# פרויקט גמר באלגוריתמים מתקדמים לתכנון ותזמון מערכות נבונות

## ניווט מבוך שחקן נגד סוכן

## Maze Navigator-Player Vs. Agent

**מאת**
**טום קובטון, ליאור מסטורוב, רועי שמשוני**

**המרכז האקדמי רופין, מדעי המחשב, שנה ג'**
**2025, התשפ"ה**

**לינק להדגמה ביוטיוב:** https://youtu.be/c0xS11T9JAw

**לינק להצגת המצגת:** https://youtu.be/flaMTzC-ES0

**לינק למשחק:** https://github.com/Awesome-Project-Boris/MazeChaser

**הערה: המשחק במלואו זמין בקובץ ה zip המצורף ( Algo.exe )**

## תוכן עניינים:

# ***הצגת הבעיה***

קבוצתנו תכננה ליצור משחק ניווט מבוך שבו שחקן אנושי השולט בתוכנה מנסה להגיע לסוף מבוך, בזמן שאחריו "רודף" סוכן ( AI ) היכול לנצל את אותן היכולות המיוחדות ( Powerups ) שהשחקן יכול לנצל. מימוש היכולות לשחקן עוקב אחרי צעדים לוגיים ברורים ועוקבים - ללכת, לנצל יכולות מיוחדות, להפעיל שיקול דעת, ליצור מסלולים קצרים ביותר חדשים על מנת להגיע לסוף המבוך, ולהטעות / לעכב את הסוכן.

הבעיה הגיעה כשפעולת הסוכן נכנס לתמונה - בעזרת התממשקות מצד ה - Script שלו ל - Scripts אחרים ( כמו שיכול השחקן ) ניתן לגרום לו לבצע את כל הפעולות האלה - מלבד **הפעלת שיקול דעת.** וכאן נכנסת לתמונה הבעיה - יש צורך בכתיבת כלי אלגוריתמי לשיקול הצעדים של הסוכן - ההגדרה שלו כאסטרטג, Hunter / Racer ( צייד השחקן אל מול פשיטה לעבר נקודת הסיום על מנת לעשות לשחקן interception ), והתממשקות עם אלגוריתמי חיפוש והשוואה למימוש השיקולים שלו.

# ***דרך הפתרון***

ראשית, היינו צריכים דרך למפות את המבוך - איפה השחקן לעומת הסוכן? איפה הסוכן לעומת השחקן? איפה שניהם לעומת נקודת היעד של המבוך?

התכנון הראשוני היה להשתמש באלגוריתם A* - אך לבסוף השתמשנו באלגוריתם **BFS** הבסיסי - מכיוון שהקשתות ( מעברים בין רצפות המבוך ) אינם ממושקלים, יותר זול להשתמש באלגוריתם BFS. חישוב משקלים לעדיפות תנועת הסוכן במקרה זה ( עדיפות לפעילות בהתחשב בעוד פרמטרים ) התבצעה בתוך אלגוריתם Minimax בצורת היוריסטית בהמשך.

הסוכן "מודע" למסלול הקצר ביותר לשחקן - הוא פונה ל - PathFinder שהוא Script הניווט על מנת לקבל את המסלול הקצר ביותר לשחקן - אך עדיין חסר לנו את המוח של הסוכן, זה שמציב תנאים למהי פעולה טובה ומקבל היוריסטיקות התורמות להבנת הרעיון "מהו צעד טוב?".

את העבודה הכבדה ( ה"חשיבה" / אסטרטגיה ) מבצע האלגוריתם השני העיקרי בו משתמשת המערכת -

האלגוריתם **Minimax** מקבל מעין Packets של מידע מהסוכן - מיקומי entities, זמני תורות קפואים, העתק של המבוך, יכולות מיוחדות של שניהם וכו'. המידע נשלח אל הפונקציה GetBestAction, שבתורה פונה אל הפונקציה GetPossibleActions שבודקת מה הסוכן יכול לעשות בנקודה שבה הוא נמצא. עבור כל פעולה אפשרית לסוכן, הפונקציה Minimax מורצת פעמיים רקורסיבית על מנת למצוא את הפעולה ה "חכמה" ביותר. בכל ביצוע של הפונקציה מופעלת הפונקציה GetStateAfterAction שבודקת מה יכולות להיות ההשלכות של התור שנבדק.

לאחר שפונקציית Minimax מסיימת לרוץ, התוצאות נבדקות ב EvaluateState ( המוח של ה Script ). כל מצב נבדק ומקבל Score לפי פרמטרים מסוימים - קרבה לשחקן, קרבת השחקן לסיום, בונוס לניקוד אם השחקן קפוא במקום וההפך אם הסוכן קפוא וכו' ). הפלט של התהליך הזה יהיה הוראות ביצוע לסוכן ( מה הצעד הבא שהוא נדרש לעשות ).

בעיה רצינית במימוש הפתרון הייתה איזון הסוכן - לעיתים היה אפתי מדי בשימושו ביכולות מיוחדות, ולפעמים אגרסיבי מידי. לפעמים הלוגיקה של Hunter / Racer נתקלה ביותר מידי באגים וכשלים לוגיים שעלתה המסקנה לוותר על Racer לגמרי. לפעמים הסוכן ביצע פעולות לא הגיוניות על מנת להשיג שיפורים קלים מדי במסלול הקצר ביותר לעבר השחקן - מה שלקח לא מעט הצבת תנאים ומשחקים עם ההיוריסטיקות עצמן.

# ***האלגוריתמים בשימוש***

1. אלגוריתם **BFS - Breadth First Search**

נערך לוגית לזיהוי המבוך ותכונותיו - יצירת המבוך התבצעה באמצעות פלאגין חיצוני למנוע פיתוח המשחקים Unity. האלגוריתם מצא שוב ושוב מסלולים שאינם הגיוניים מכיוון שיצירת המבוך כללה יצירת קירות "כפולים" - בין שני פנלים של רצפות, על אותה הפאה לפעמים נבנו קירות, לפעמים רק אחד מהם זוהה. לאחר התגברות על הבעיה הזו, האלגוריתם מצא את היעדים שלו בקלות וביעילות.

הוחלט לא להשתמש באלגוריתם A* כנאמר מקודם מכיוון שהניווט במבוך עצמו הוא ניווט בגרף עם קשתות במשקל שווה, למרות שאת חלק מהגיון קיצור הדרך יכול היה להיכנס לניווט עצמו, אך העדפנו לבצע אותו ב - Minimax.

האלגוריתם קיים על מנת למצוא 3 מסלולים עיקריים ( שחקן-סוף, סוכן-שחקן, סוכן-סוף ) ונקרא גם בין תורות ( אם למשל השחקן החליט לשבור קיר ופרקטית לשנות את המבוך ).

2. אלגוריתם **Minimax**

האלגוריתם הוא חלק קטן ממכלול הנעזר בשלל פונקציות עזר על מנת לקבל, בסופו של דבר, את המהלך הכי טוב שהסוכן יבצע. הסוכן יקבל את המידע מהאלגוריתם, ויצבע על לוח המשחק את המסלול הקצר ביותר ( כחול ), אך גם מסלול מעט יותר אסטרטגי של "מה אם ( אעשה פעולה כזו או כזו? )". במהלך Playtesting אכן נצפה הסוכן סוטה מהמסלול הקצר ביותר ולפעמים "מטייל" בלוח ומחכה להזדמנות טובה להשתמש ב - Powerup ולדחוק את השחקן לפינה.

הסוכן מתעדף שימוש יחסית אינטנסיבי ב - Powerups ( מה שלפעמים בא לרעתו, אך מפעיל לחץ רב על השחקן ). נבדקה האפשרות להפעיל את הרקורסיה ב Minimax עד 4 פעמים, אך הורגשה פגיעה משמעותית בביצועים ועלה חשש לקריסת המשחק, ואמנם הסוכן ביצע מהלכים מעט יותר מרשימים, הפגיעה בביצועים לא הצדיקה עלייה מ - 2.

# ***קוד מעשי***

במסמך זה לא אצרף את כל קטעי הקוד, הם יצורפו לקובץ ה - zip. סה"כ ללא קבצי המשחק עצמו, מספר ה - Scripts הוא 23 - כמחצית מהם Scripts השייכים ליצירת המבוך ומתוכם יש סקריפטים לא רלוונטים שלא ימחקו מחשש לשבירת הלוגיקה של הפלאגין ככלל.

רשימת ה - Scripts המצורפים לפי הסדר:

1. MazeGenerator - ראש הפאזל ליצירת המבוך
2. MazePathfiner - נווט המבוך
3. AIController - גוף הסוכן
4. AIAction ( type ) - הגוף של פעולה שנשקלת עבור הסוכן
5. MiniMaxGameState - ( type ) - הגוף של מצב המשחק לשיקול האלגוריתם
6. MinimaxController - המוח ופונקציות העזר להחלטת פעולות הסוכן

## ***MazeSpawner ( script )***

```
using UnityEngine;
using System.Collections.Generic;

public class MazeSpawner : MonoBehaviour
{
    // --- Public Fields for Unity Inspector ---
    public enum MazeGenerationAlgorithm { PureRecursive, RecursiveTree,
RandomTree, OldestTree, RecursiveDivision, }
    public MazeGenerationAlgorithm Algorithm =
MazeGenerationAlgorithm.PureRecursive;
    public bool FullRandom;
    public int RandomSeed;
    public int Rows;
    public int Columns;
    public float CellWidth;
    public float CellHeight;
    public bool AddGaps;

    [Header("Prefabs")]
    public GameObject Floor = null;
    public GameObject Wall = null;
    public GameObject Pillar = null;
    public GameObject PlayerPrefab = null;
    public GameObject EnemyPrefab = null;
    public GameObject EndGoalPrefab = null;

    // --- Public Properties ---
    public BasicMazeGenerator MazeGenerator { get; private set; }

    // --- Private Fields ---
    private Dictionary<Vector2Int, TileInfo> tileInfos = new
Dictionary<Vector2Int, TileInfo>();

    public void BeginSpawning()
    {
        // Set the random seed if not using a fully random maze
        if (!FullRandom)
        {
            Random.seed = RandomSeed;
```

```
        }
        switch (Algorithm)
        {
            case MazeGenerationAlgorithm.PureRecursive:
                MazeGenerator = new RecursiveMazeGenerator(Rows,
Columns);
                break;
            case MazeGenerationAlgorithm.RecursiveTree:
                MazeGenerator = new RecursiveTreeMazeGenerator(Rows,
Columns);
                break;
            case MazeGenerationAlgorithm.RandomTree:
                MazeGenerator = new RandomTreeMazeGenerator(Rows,
Columns);
                break;
            case MazeGenerationAlgorithm.OldestTree:
                MazeGenerator = new OldestTreeMazeGenerator(Rows,
Columns);
                break;
            case MazeGenerationAlgorithm.RecursiveDivision:
                MazeGenerator = new DivisionMazeGenerator(Rows,
Columns);
                break;
        }
        MazeGenerator.GenerateMaze();

        // --- PASS 1: Instantiate all Floor Tiles ---
        for (int row = 0; row < Rows; row++)
        {
            for (int column = 0; column < Columns; column++)
            {
                float x = column * (CellWidth + (AddGaps ? 0.2f : 0));
                float z = row * (CellHeight + (AddGaps ? 0.2f : 0));

                GameObject tmp_floor = Instantiate(Floor, new
Vector3(x, 0, z), Quaternion.Euler(0, 0, 0));
                tmp_floor.transform.parent = transform;

                TileInfo currentTileInfo =
tmp_floor.GetComponent<TileInfo>();
                if (currentTileInfo != null)
                {
```

```
                    tileInfos[new Vector2Int(column, row)] =
currentTileInfo;
                }
            }
        }

        // --- PASS 2: Instantiate Walls and link them to the existing
TileInfos ---

        for (int row = 0; row < Rows; row++)
        {
            for (int column = 0; column < Columns; column++)
            {
                MazeCell cell = MazeGenerator.GetMazeCell(row, column);
                TileInfo currentTileInfo = tileInfos[new
Vector2Int(column, row)];
                float x = column * (CellWidth + (AddGaps ? 0.2f : 0));
                float z = row * (CellHeight + (AddGaps ? 0.2f : 0));

                // This logic for WallRight is correct and should stay.
                if (cell.WallRight)
                {
                    GameObject tmp_wall = Instantiate(Wall, new
Vector3(x + CellWidth / 2, 0, z) + Wall.transform.position,
Quaternion.Euler(0, 90, 0));
                    tmp_wall.transform.parent = transform;

                    // This part is new from our refactoring and is
correct.
                    currentTileInfo.WallObjects[Direction.Right] =
tmp_wall;
                    var neighborKey = new Vector2Int(column + 1, row);
                    if (tileInfos.ContainsKey(neighborKey)) {
tileInfos[neighborKey].WallObjects[Direction.Left] = tmp_wall; }
                }

                // This logic for WallFront is correct and should stay.
                if (cell.WallFront)
                {
                    GameObject tmp_wall = Instantiate(Wall, new
Vector3(x, 0, z + CellHeight / 2) + Wall.transform.position,
Quaternion.Euler(0, 0, 0));
                    tmp_wall.transform.parent = transform;
```

```
                    currentTileInfo.WallObjects[Direction.Front] =
tmp_wall;
                    var neighborKey = new Vector2Int(column, row + 1);
                    if (tileInfos.ContainsKey(neighborKey)) {
tileInfos[neighborKey].WallObjects[Direction.Back] = tmp_wall; }
                }

                if (cell.WallLeft)
                {
                    GameObject tmp_wall = Instantiate(Wall, new
Vector3(x - CellWidth / 2, 0, z) + Wall.transform.position,
Quaternion.Euler(0, 270, 0));
                    tmp_wall.transform.parent = transform;

                    // The linking logic for this wall is already
handled by the WallRight of the neighbor,
                    // but we can add it here for completeness if
needed in the future.
                    currentTileInfo.WallObjects[Direction.Left] =
tmp_wall;
                }

                if (cell.WallBack)
                {
                    GameObject tmp_wall = Instantiate(Wall, new
Vector3(x, 0, z - CellHeight / 2) + Wall.transform.position,
Quaternion.Euler(0, 180, 0));
                    tmp_wall.transform.parent = transform;

                    currentTileInfo.WallObjects[Direction.Back] =
tmp_wall;
                }
            }
        }

        // --- 3. Instantiate Pillars (Optional) ---
        if (Pillar != null)
        {
            for (int row = 0; row < Rows + 1; row++)
            {
                for (int column = 0; column < Columns + 1; column++)
                {
```

```
                    float x = column * (CellWidth + (AddGaps ? 0.2f :
0));
                    float z = row * (CellHeight + (AddGaps ? 0.2f :
0));

                    GameObject tmp_pillar = Instantiate(Pillar, new
Vector3(x - CellWidth / 2, 0, z - CellHeight / 2),
Quaternion.identity);
                    tmp_pillar.transform.parent = transform;
                }
            }
        }

        // --- 4. Strategically Place and Instantiate Entities ---
        Vector2Int goalPosition = new Vector2Int(Columns - 1, Rows -
1);
        MazeGenerator.GetMazeCell(goalPosition.y,
goalPosition.x).IsGoal = true;
        Vector2Int playerStartPos =
MazePathfinder.FindFurthestCell(goalPosition.y, goalPosition.x,
MazeGenerator);
        Vector2Int enemyStartPos = FindBestEnemySpawn(playerStartPos,
goalPosition);

        System.Func<int, int, Vector3> GetWorldPos = (row, col) => {
            return new Vector3(col * (CellWidth + (AddGaps ? 0.2f :
0)), 1, row * (CellHeight + (AddGaps ? 0.2f : 0)));
        };

        // --- 5. Spawn Player, AI, and Goal ---
        if (EndGoalPrefab != null) { Instantiate(EndGoalPrefab,
GetWorldPos(goalPosition.y, goalPosition.x), Quaternion.identity,
transform); }
        GameObject playerObj = Instantiate(PlayerPrefab,
GetWorldPos(playerStartPos.y, playerStartPos.x), Quaternion.identity,
transform);
        GameObject aiObj = Instantiate(EnemyPrefab,
GetWorldPos(enemyStartPos.y, enemyStartPos.x), Quaternion.identity,
transform);

        if (playerObj != null && aiObj != null)
        {
```

```
GameManager.Instance.InitializeGame(playerObj.GetComponent<PlayerContro
ller>(), aiObj.GetComponent<AIController>());
        }
    }

    // This is the enemy placement logic
    private Vector2Int FindBestEnemySpawn(Vector2Int playerPos,
Vector2Int goalPos)
    {
        var playerDistanceMap =
MazePathfinder.CalculateAllDistances(playerPos.y, playerPos.x,
MazeGenerator);
        var goalDistanceMap =
MazePathfinder.CalculateAllDistances(goalPos.y, goalPos.x,
MazeGenerator);
        var playerToGoalPath =
MazePathfinder.FindShortestPath(playerPos.y, playerPos.x, goalPos.y,
goalPos.x, MazeGenerator);
        var playerToGoalPathLookup = new
HashSet<Vector2Int>(playerToGoalPath);

        List<Vector2Int> candidateCells = new List<Vector2Int>();
        for (int r = 0; r < Rows; r++)
        {
            for (int c = 0; c < Columns; c++)
            {
                var currentPos = new Vector2Int(c, r);
                if (playerDistanceMap.ContainsKey(currentPos) &&
goalDistanceMap.ContainsKey(currentPos) &&
!playerToGoalPathLookup.Contains(currentPos) &&
                    playerDistanceMap[currentPos] >= 10 &&
goalDistanceMap[currentPos] >= 8)
                {
                    candidateCells.Add(currentPos);
                }
            }
        }
```

```
        if (candidateCells.Count > 0)
        {
            return candidateCells[Random.Range(0,
candidateCells.Count)];
        }
        // Fallback if no candidates are found
        return MazePathfinder.FindFurthestCell(playerPos.y,
playerPos.x, MazeGenerator);
    }

    public TileInfo GetTileInfo(int row, int col)
    {
        var key = new Vector2Int(col, row);
        return tileInfos.ContainsKey(key) ? tileInfos[key] : null;
    }

    public GameObject GetFloorTile(int row, int col)
    {
        // This method relies on the 'tileInfos' dictionary and the
'TileInfo' component.
        // It gets the TileInfo for a grid position and returns its
GameObject.
        TileInfo tileInfo = GetTileInfo(row, col);
        if (tileInfo != null)
        {
            return tileInfo.gameObject;
        }
        return null;
    }
}
```

***MazePathfiner ( script )***

```
using System.Collections.Generic;
using UnityEngine;


// We're using BFS to find the optimal placements for enemy and player.

public static class MazePathfinder
{
    public static Vector2Int FindFurthestCell(int startRow, int
startCol, BasicMazeGenerator generator)
    {
        Queue<Vector2Int> queue = new Queue<Vector2Int>();
        Dictionary<Vector2Int, Vector2Int?> visited = new
Dictionary<Vector2Int, Vector2Int?>();
        Vector2Int startNode = new Vector2Int(startCol, startRow);
        queue.Enqueue(startNode);
        visited[startNode] = null;
        Vector2Int furthestNode = startNode;

        while (queue.Count > 0)
        {
            Vector2Int currentNode = queue.Dequeue();
            furthestNode = currentNode;
            MazeCell currentCell = generator.GetMazeCell(currentNode.y,
currentNode.x);

            // Check neighbors
            if (!currentCell.WallRight && !visited.ContainsKey(new
Vector2Int(currentNode.x + 1, currentNode.y)))
            {
                Vector2Int neighbor = new Vector2Int(currentNode.x + 1,
currentNode.y);
                visited[neighbor] = currentNode;
                queue.Enqueue(neighbor);
            }
```

```
            if (!currentCell.WallLeft && !visited.ContainsKey(new
Vector2Int(currentNode.x - 1, currentNode.y)))
            {
                Vector2Int neighbor = new Vector2Int(currentNode.x - 1,
currentNode.y);
                visited[neighbor] = currentNode;
                queue.Enqueue(neighbor);
            }
            if (!currentCell.WallFront && !visited.ContainsKey(new
Vector2Int(currentNode.x, currentNode.y + 1)))
            {
                Vector2Int neighbor = new Vector2Int(currentNode.x,
currentNode.y + 1);
                visited[neighbor] = currentNode;
                queue.Enqueue(neighbor);
            }
            if (!currentCell.WallBack && !visited.ContainsKey(new
Vector2Int(currentNode.x, currentNode.y - 1)))
            {
                Vector2Int neighbor = new Vector2Int(currentNode.x,
currentNode.y - 1);
                visited[neighbor] = currentNode;
                queue.Enqueue(neighbor);
            }
        }
        return furthestNode;
    }


    // Finds the single shortest path between a start and end point.
    // It returns a List of coordinates representing the cells on the
path.

    public static List<Vector2Int> FindShortestPath(int startRow, int
startCol, int endRow, int endCol, BasicMazeGenerator generator)
    {
        Queue<Vector2Int> queue = new Queue<Vector2Int>();
        Dictionary<Vector2Int, Vector2Int?> visited = new
Dictionary<Vector2Int, Vector2Int?>();
        Vector2Int startNode = new Vector2Int(startCol, startRow);
        Vector2Int endNode = new Vector2Int(endCol, endRow);
```

```
        if (startNode == endNode) return new List<Vector2Int> {
startNode };

        queue.Enqueue(startNode);
        visited[startNode] = null;

        while (queue.Count > 0)
        {
            Vector2Int currentNode = queue.Dequeue();
            if (currentNode == endNode) break;

            MazeCell currentCell = generator.GetMazeCell(currentNode.y,
currentNode.x);

            // Helper to perform a strict two-way check
            System.Action<Direction> checkNeighbor = (dir) => {
                Vector2Int neighborNode;
                bool isPathClear = false;

                // Get neighbor position and check its corresponding
wall
                switch (dir)
                {
                    case Direction.Front:
                        neighborNode = new Vector2Int(currentNode.x,
currentNode.y + 1);
                        if (neighborNode.y < generator.RowCount &&
!currentCell.WallFront && !generator.GetMazeCell(neighborNode.y,
neighborNode.x).WallBack) isPathClear = true;
                        break;
                    case Direction.Back:
                        neighborNode = new Vector2Int(currentNode.x,
currentNode.y - 1);
                        if (neighborNode.y >= 0 &&
!currentCell.WallBack && !generator.GetMazeCell(neighborNode.y,
neighborNode.x).WallFront) isPathClear = true;
                        break;
                    case Direction.Right:
                        neighborNode = new Vector2Int(currentNode.x +
1, currentNode.y);
                        if (neighborNode.x < generator.ColumnCount &&
!currentCell.WallRight && !generator.GetMazeCell(neighborNode.y,
neighborNode.x).WallLeft) isPathClear = true;
```

```
                        break;
                    case Direction.Left:
                        neighborNode = new Vector2Int(currentNode.x -
1, currentNode.y);
                        if (neighborNode.x >= 0 &&
!currentCell.WallLeft && !generator.GetMazeCell(neighborNode.y,
neighborNode.x).WallRight) isPathClear = true;
                        break;
                    default:
                        return;
                }

                if (isPathClear && !visited.ContainsKey(neighborNode))
                {
                    visited[neighborNode] = currentNode;
                    queue.Enqueue(neighborNode);
                }
            };

            checkNeighbor(Direction.Front);
            checkNeighbor(Direction.Back);
            checkNeighbor(Direction.Right);
            checkNeighbor(Direction.Left);
        }

        // Backtrack from the end node to the start node to build the
path.
        List<Vector2Int> path = new List<Vector2Int>();
        Vector2Int? pathNode = endNode;
        while (pathNode != null && visited.ContainsKey(pathNode.Value))
        {
            path.Add(pathNode.Value);
            pathNode = visited[pathNode.Value];
        }
        path.Reverse();

        // Return the path only if it's valid (starts at the start
node)
        return (path.Count > 0 && path[0] == startNode) ? path : null;
    }

    // Calculates the distance from every cell in the maze to the
NEAREST cell on a given path.
```

```
    // This uses a "multi-source" BFS starting from all path cells at
once.

    public static Dictionary<Vector2Int, int>
CalculateDistancesFromPath(List<Vector2Int> path, BasicMazeGenerator
generator)
    {
        Dictionary<Vector2Int, int> distances = new
Dictionary<Vector2Int, int>();
        Queue<Vector2Int> queue = new Queue<Vector2Int>();

        // Start the search from EVERY cell on the path, with a
distance of 0.

        foreach (var pathCell in path)
        {
            if (!distances.ContainsKey(pathCell))
            {
                distances[pathCell] = 0;
                queue.Enqueue(pathCell);
            }
        }

        // Standard BFS

        while (queue.Count > 0)
        {
            Vector2Int currentNode = queue.Dequeue();
            MazeCell currentCell = generator.GetMazeCell(currentNode.y,
currentNode.x);

            System.Action<Vector2Int> checkNeighbor = (neighborNode) =>
{
                if (!distances.ContainsKey(neighborNode))
                {
                    distances[neighborNode] = distances[currentNode] +
1;
                    queue.Enqueue(neighborNode);
                }
            };

            if (!currentCell.WallRight) checkNeighbor(new
Vector2Int(currentNode.x + 1, currentNode.y));
```

```
            if (!currentCell.WallLeft) checkNeighbor(new
Vector2Int(currentNode.x - 1, currentNode.y));
            if (!currentCell.WallFront) checkNeighbor(new
Vector2Int(currentNode.x, currentNode.y + 1));
            if (!currentCell.WallBack) checkNeighbor(new
Vector2Int(currentNode.x, currentNode.y - 1));
        }
        return distances;
    }

    // This method returns a map of distances from the start cell to
all other reachable cells.

    public static Dictionary<Vector2Int, int> CalculateAllDistances(int
startRow, int startCol, BasicMazeGenerator generator)
    {
        Dictionary<Vector2Int, int> distances = new
Dictionary<Vector2Int, int>();
        Queue<Vector2Int> queue = new Queue<Vector2Int>();

        Vector2Int startNode = new Vector2Int(startCol, startRow);

        queue.Enqueue(startNode);
        distances[startNode] = 0;

        while (queue.Count > 0)
        {
            Vector2Int currentNode = queue.Dequeue();
            MazeCell currentCell = generator.GetMazeCell(currentNode.y,
currentNode.x);

            // Check neighbors
            System.Action<Vector2Int> checkNeighbor = (neighborNode) =>
{
                if (!distances.ContainsKey(neighborNode))
                {
                    distances[neighborNode] = distances[currentNode] +
1;
                    queue.Enqueue(neighborNode);
                }
            };
```

```
            if (!currentCell.WallRight) checkNeighbor(new
Vector2Int(currentNode.x + 1, currentNode.y));
            if (!currentCell.WallLeft) checkNeighbor(new
Vector2Int(currentNode.x - 1, currentNode.y));
            if (!currentCell.WallFront) checkNeighbor(new
Vector2Int(currentNode.x, currentNode.y + 1));
            if (!currentCell.WallBack) checkNeighbor(new
Vector2Int(currentNode.x, currentNode.y - 1));
        }

        return distances;
    }

    // This is a new, overloaded version of FindShortestPath.
    // It allows us to run a path search on a temporary, modified copy
of the maze data.

    public static List<Vector2Int> FindShortestPath(int startRow, int
startCol, int endRow, int endCol, MazeCell[,] mazeData, int rows, int
columns)
    {
        Queue<Vector2Int> queue = new Queue<Vector2Int>();
        Dictionary<Vector2Int, Vector2Int?> visited = new
Dictionary<Vector2Int, Vector2Int?>();
        Vector2Int startNode = new Vector2Int(startCol, startRow);
        Vector2Int endNode = new Vector2Int(endCol, endRow);

        queue.Enqueue(startNode);
        visited[startNode] = null;

        while (queue.Count > 0)
        {
            Vector2Int currentNode = queue.Dequeue();
            if (currentNode == endNode) break;

            // The only change is here: we access the mazeData array
directly.

            MazeCell currentCell = mazeData[currentNode.y,
currentNode.x];

            System.Action<Vector2Int> checkNeighbor = (neighborNode) =>
{
```

```
                if (neighborNode.y >= 0 && neighborNode.y < rows &&
neighborNode.x >= 0 && neighborNode.x < columns &&
!visited.ContainsKey(neighborNode))
                {
                    visited[neighborNode] = currentNode;
                    queue.Enqueue(neighborNode);
                }
            };

            if (!currentCell.WallRight) checkNeighbor(new
Vector2Int(currentNode.x + 1, currentNode.y));
            if (!currentCell.WallLeft) checkNeighbor(new
Vector2Int(currentNode.x - 1, currentNode.y));
            if (!currentCell.WallFront) checkNeighbor(new
Vector2Int(currentNode.x, currentNode.y + 1));
            if (!currentCell.WallBack) checkNeighbor(new
Vector2Int(currentNode.x, currentNode.y - 1));
        }

        List<Vector2Int> path = new List<Vector2Int>();
        Vector2Int? pathNode = endNode;
        while (pathNode != null && visited.ContainsKey(pathNode.Value))
        {
            path.Add(pathNode.Value);
            pathNode = visited[pathNode.Value];
        }
        path.Reverse();
        return path;
    }
}
```

## ***AIController ( script )***

```
using UnityEngine;
using System.Collections.Generic;
using System.Linq;

[RequireComponent(typeof(TileMovement))]
[RequireComponent(typeof(MinimaxController))]

public class AIController : MonoBehaviour
{
    private List<GameObject> highlightedPathTiles = new
List<GameObject>(); // Main path to consider
    private List<GameObject> highlightedStrategicPathTiles = new
List<GameObject>(); // Alternative wildcard path to consider

    private Dictionary<GameObject, Color> originalTileColors = new
Dictionary<GameObject, Color>(); // Original color of tiles

    public int TurnsFrozen { get; set; } = 0;

    [Header("AI Configuration")]
    [Tooltip("How many turns ahead the AI will think. 2 is a good
starting point.")]
    [Range(0, 4)]
    public int AIDepth = 2;

    private TileMovement tileMovement;
    private MinimaxController minimaxController;
    private PlayerController player;
    private MazeSpawner mazeSpawner;
    private PowerupManager powerupManager;

    public void Initialize()
    {
```

```
        tileMovement = GetComponent<TileMovement>();
        minimaxController = GetComponent<MinimaxController>();

        // Getting component references

        player = GameManager.Instance.player;
        mazeSpawner = FindObjectOfType<MazeSpawner>();
        powerupManager = PowerupManager.Instance;
    }

    public void TakeTurn()
    {
        if (tileMovement.GetCurrentGridPosition() ==
player.GetComponent<TileMovement>().GetCurrentGridPosition())
        {
            Debug.Log("CAUGHT! The AI has moved onto the player's
tile.");
            GameManager.Instance.EndGame(false); // Player loses
            return;
        }

        ClearAllHighlights();

        if (TurnsFrozen > 0)
        {
            Debug.Log("[AI] Turn skipped (frozen).");
            TurnsFrozen--;
            ClearPreviousPath(); // Clear the path if frozen
            GameManager.Instance.EndAITurn();
            return;
        }

        // --- VISUALIZER CALL ---

        // At the start of the turn, this shows the path the AI is
considering.

        vvar chasePath = MazePathfinder.FindShortestPath(
            tileMovement.GetCurrentGridPosition().y,
tileMovement.GetCurrentGridPosition().x,

player.GetComponent<TileMovement>().GetCurrentGridPosition().y,
player.GetComponent<TileMovement>().GetCurrentGridPosition().x,
```

```
            mazeSpawner.MazeGenerator
        );
        // Draw the main chase path in blue.
        VisualizePath(chasePath, new Color(0.9f, 0.9f, 1.0f));


        // 1. We figure out how's the game going and the parameters for
Minimax's condsideration

        MinimaxGameState currentState = new MinimaxGameState
        {
            AIPos = tileMovement.GetCurrentGridPosition(),
            PlayerPos =
player.GetComponent<TileMovement>().GetCurrentGridPosition(),
            AITurnsFrozen = this.TurnsFrozen,
            PlayerTurnsFrozen = player.TurnsFrozen,
            AIPowerups = new List<Powerup>(powerupManager.AIPowerups),
            PlayerPowerups = new
List<Powerup>(powerupManager.PlayerPowerups),
            MazeGrid = mazeSpawner.MazeGenerator.MazeGrid,
            MazeRows = mazeSpawner.Rows,
            MazeColumns = mazeSpawner.Columns
        };

        // 2. We ask the Minimax brain for the best action.

        AIAction bestAction =
minimaxController.GetBestAction(currentState, AIDepth);

        // 2.5. If we don't get a suitable "best" action, we just tell
the AI to move down the path to the player.

        if (bestAction == null)
        {
            Debug.LogWarning("[AI] Minimax returned no best action.
Using fallback: move towards player.");
            var path =
MazePathfinder.FindShortestPath(currentState.AIPos.y,
currentState.AIPos.x, currentState.PlayerPos.y,
currentState.PlayerPos.x, mazeSpawner.MazeGenerator);
            if (path != null && path.Count > 1)
            {
                // Create a simple move action from the path
```

```
                bestAction = new AIAction
                {
                    Type = ActionType.Move,
                    MoveDirection = GetDirectionFromVector(path[1] -
currentState.AIPos)
                };
            }
            else
            {
                // If there's truly no action and no path, end the
turn.
                Debug.LogError("[AI] Fallback failed: No path to
player. AI is trapped. Ending turn.");
                GameManager.Instance.EndAITurn();
                return;
            }
        }

         Color boldColor = new Color(0.6f, 0.6f, 1.0f);
 Color dimColor = new Color(0.9f, 0.9f, 1.0f);

 // Determine the final, chosen path based on the AI's action.
 List<Vector2Int> finalPath;
 if (bestAction.Type == ActionType.UsePowerup &&
(bestAction.PowerupType == PowerupType.BreakWall ||
bestAction.PowerupType == PowerupType.Jump))
 {
     // The best action was strategic, so the "final path" is the one
AFTER the power-up.
     var stateAfterAction =
minimaxController.GetStateAfterAction(currentState, bestAction, true);
     finalPath =
minimaxController.FindShortestPathSimulated(stateAfterAction.AIPos,
stateAfterAction.PlayerPos, stateAfterAction);

     // In this case, we also want to show the "alternative" dumb path.
     var alternativePath = MazePathfinder.FindShortestPath(
         currentState.AIPos.y, currentState.AIPos.x,
         currentState.PlayerPos.y, currentState.PlayerPos.x,
         mazeSpawner.MazeGenerator
     );
     // Draw the alternative path first with the dim color.
     VisualizePath(alternativePath, dimColor);
```

```
 }
 else
 {
     // The best action was a simple move, so the final path is the
direct chase path.
     finalPath = MazePathfinder.FindShortestPath(
         currentState.AIPos.y, currentState.AIPos.x,
         currentState.PlayerPos.y, currentState.PlayerPos.x,
         mazeSpawner.MazeGenerator
     );
 }

 // Draw the final, chosen path with the BOLD color.
 // The VisualizePath logic will automatically prevent it from
overwriting tiles
 // that were already colored by the dim "alternative" path if they
overlap.
 // To ensure the main path is always prominent, we will draw it last.

 VisualizePath(finalPath, boldColor);

        Debug.Log($"[AI DECISION]: {bestAction}");

        if (bestAction.Type == ActionType.UsePowerup &&
(bestAction.PowerupType == PowerupType.BreakWall ||
bestAction.PowerupType == PowerupType.Jump))
        {
            var stateAfterAction =
minimaxController.GetStateAfterAction(currentState, bestAction, true);
            var strategicPath =
minimaxController.FindShortestPathSimulated(stateAfterAction.AIPos,
stateAfterAction.PlayerPos, stateAfterAction);
            // Draw the strategic path in a different color, on top of
the blue path.
            VisualizePath(strategicPath, new Color(0.6f, 0.6f, 1.0f));
        }

        if (bestAction.Type == ActionType.UsePowerup &&
(bestAction.PowerupType == PowerupType.BreakWall ||
bestAction.PowerupType == PowerupType.Jump))
        {
            // Create a temporary clone of the maze to simulate the
wall break
```

```
            var stateAfterAction =
minimaxController.GetStateAfterAction(currentState, bestAction, true);
            var strategicPath =
minimaxController.FindShortestPathSimulated(stateAfterAction.AIPos,
stateAfterAction.PlayerPos, stateAfterAction);
            VisualizeStrategicPath(strategicPath);
        }

        if (bestAction.Type == ActionType.UsePowerup)
        {
            string powerupName =
System.Text.RegularExpressions.Regex.Replace(bestAction.PowerupType.ToS
tring(), "(\\B[A-Z])", " $1"); // We get the name of the powerup being
used
            string chatMessage = $"Agent used {powerupName} at turn
{GameManager.Instance.GetTurnNumber()}!";
            UIManager.Instance.AddToChatHistory(chatMessage);
        }

        // 3. The Agent executes the move

        if (bestAction.Type == ActionType.Move)
        {

tileMovement.AttemptMove(GetVectorFromDirection(bestAction.MoveDirectio
n), (success) =>
            {
                GameManager.Instance.EndAITurn();
            });
        }
        else if (bestAction.Type == ActionType.UsePowerup)
        {
            ExecutePowerupAction(bestAction);
            GameManager.Instance.EndAITurn();
        }
    }



    private void ExecutePowerupAction(AIAction action)
    {
        // Use a direct reference to the AI's inventory
        var aiInventory = powerupManager.AIPowerups;
```

```
        switch (action.PowerupType)
        {
            case PowerupType.BreakWall:
                powerupManager.ExecuteBreakWall(aiInventory,
action.PowerupSlot, tileMovement.GetCurrentGridPosition(),
action.PowerupTargetDirection);
                break;
            case PowerupType.Jump:
                powerupManager.ExecuteJump(aiInventory,
action.PowerupSlot, tileMovement.GetCurrentGridPosition(),
action.PowerupTargetDirection, this);
                break;
            case PowerupType.Dash:
                // Dash requires a callback to end the turn after the
animation.
                List<Vector2Int> dashPath =
powerupManager.CalculateDashPath(tileMovement.GetCurrentGridPosition(),
action.PowerupTargetDirection);
                aiInventory.RemoveAt(action.PowerupSlot); // Consume
powerup

UIManager.Instance.UpdatePowerupDisplay(powerupManager.PlayerPowerups,
powerupManager.AIPowerups);
                tileMovement.ExecuteDash(dashPath, () => {
GameManager.Instance.EndAITurn(); });
                return; // Return early because the callback will end
the turn.
            case PowerupType.Freeze:
                powerupManager.ExecuteFreeze(aiInventory,
action.PowerupSlot, player);
                break;
            case PowerupType.Teleport:
                powerupManager.ExecuteTeleport(aiInventory,
action.PowerupSlot, player, this);
                break;
        }
    }

    private Vector2Int GetVectorFromDirection(Direction dir)
    {
        switch (dir)
        {
```

```
            case Direction.Front: return Vector2Int.up;
            case Direction.Back: return Vector2Int.down;
            case Direction.Right: return Vector2Int.right;
            case Direction.Left: return Vector2Int.left;
            default: return Vector2Int.zero;
        }
    }

    private Direction GetDirectionFromVector(Vector2Int dir)
    {
        if (dir == Vector2Int.up) return Direction.Front;
        if (dir == Vector2Int.down) return Direction.Back;
        if (dir == Vector2Int.right) return Direction.Right;
        if (dir == Vector2Int.left) return Direction.Left;
        return Direction.Start;
    }


    // Path debugging


// In AIController.cs

    private void ClearPreviousPath()
    {
        foreach (var tile in highlightedPathTiles)
        {
            // SAFETY CHECK: Only try to revert the color if the tile
still exists
            // and we have its original color stored in our dictionary.
            if (tile != null && originalTileColors.ContainsKey(tile))
            {
                tile.GetComponent<Renderer>().material.color =
originalTileColors[tile];
            }
        }
        foreach (var tile in highlightedStrategicPathTiles)
        {
            // Add the same safety check here for the strategic path.
            if (tile != null && originalTileColors.ContainsKey(tile))
            {
                tile.GetComponent<Renderer>().material.color =
originalTileColors[tile];
```

```
            }
        }

        highlightedPathTiles.Clear();
        highlightedStrategicPathTiles.Clear();

    }


private void ClearAllHighlights()
{
    foreach (var entry in originalTileColors)
    {
        // Key is the GameObject, Value is the original Color
        if (entry.Key != null)
        {
            entry.Key.GetComponent<Renderer>().material.color =
entry.Value;
        }
    }
    // After reverting all colors, we clear the dictionary for the next
turn.
    originalTileColors.Clear();
}

private void VisualizePath(List<Vector2Int> path, Color highlightColor)
{
    if (path == null) return;

    foreach (var pos in path)
    {
        GameObject tileObject = mazeSpawner.GetFloorTile(pos.y, pos.x);
        if (tileObject != null)
        {
            var tileRenderer = tileObject.GetComponent<Renderer>();
            if (tileRenderer != null)
            {
                // IMPORTANT: Only store the tile's color if it's the
first time
                // we are highlighting it this turn.
                if (!originalTileColors.ContainsKey(tileObject))
                {
```

```
                    originalTileColors.Add(tileObject,
tileRenderer.material.color);
                }
                // Apply the highlight color.
                tileRenderer.material.color = highlightColor;
            }
        }
    }
}

void OnTriggerEnter(Collider other)
    {
        // Check if the object we touched has a PlayerController script
on it.
        if (other.GetComponent<PlayerController>() != null)
        {
            Debug.Log("CAUGHT! The AI has collided with the player.");
            // Tell the GameManager the player lost.
            GameManager.Instance.EndGame(false);
        }
    }
}
```

## ***AIAction ( script )***

```
using UnityEngine;

public enum ActionType { Move, UsePowerup }

/// Represents a single, potential action a character can take,
/// complete with all necessary information for execution and evaluation.

public class AIAction
{
    public ActionType Type { get; set; }
-
    public Direction MoveDirection { get; set; }

    //  Fields for a 'UsePowerup' action

    public int PowerupSlot { get; set; } // The inventory index (0, 1, or 2)
    public PowerupType PowerupType { get; set; }
    public Direction PowerupTargetDirection { get; set; } // For directional
powerups (BreakWall, Jump, Dash)

    public float Score { get; set; }

    // Helper for debugging to easily see what action the AI chose.

    public override string ToString()
    {
        if (Type == ActionType.Move)
        {
            return $"Action: Move {MoveDirection}, Score: {Score:F2}";
        }
        else
        {
            string target = (PowerupTargetDirection != Direction.Start &&
PowerupTargetDirection != 0) ? $" towards {PowerupTargetDirection}" : "";
            return $"Action: Use {PowerupType}{target} (from slot
{PowerupSlot}), Score: {Score:F2}";
```

```
        }
    }
}
```

## ***MiniMaxGameState ( script )***

```
using UnityEngine;
using System.Collections.Generic;


/// A snapshot of all relevant information for the Minimax algorithm to
evaluate a possible future.
/// This class is designed to be cloned and modified during simulation
without affecting the live game.


public class MinimaxGameState
{
    // Character States

    public Vector2Int AIPos { get; set; }
    public Vector2Int PlayerPos { get; set; }
    public int AITurnsFrozen { get; set; }
    public int PlayerTurnsFrozen { get; set; }

    // Inventories

    public List<Powerup> AIPowerups { get; set; }
    public List<Powerup> PlayerPowerups { get; set; }

    // Maze State

    public MazeCell[,] MazeGrid { get; set; }
    public int MazeRows { get; set; }
    public int MazeColumns { get; set; }


    /// Creates a deep copy of this game state for simulation.
```

```
    /// Essential for allowing the Minimax algorithm to explore
different futures.

    public MinimaxGameState Clone()
    {
        // Create new lists for powerups to avoid modifying the
original inventories.

        var aiPowerupsCopy = new List<Powerup>();
        foreach (var p in this.AIPowerups) { aiPowerupsCopy.Add(new
Powerup(p)); }

        var playerPowerupsCopy = new List<Powerup>();
        foreach (var p in this.PlayerPowerups) {
playerPowerupsCopy.Add(new Powerup(p)); }

        // Create a deep copy of the maze grid. This is crucial for
simulating wall breaks.

        MazeCell[,] mazeGridCopy = new MazeCell[MazeRows, MazeColumns];
        for (int r = 0; r < MazeRows; r++)
        {
            for (int c = 0; c < MazeColumns; c++)
            {
                mazeGridCopy[r, c] = new MazeCell(this.MazeGrid[r, c]);
            }
        }

        return new MinimaxGameState
        {
            AIPos = this.AIPos,
            PlayerPos = this.PlayerPos,
            AITurnsFrozen = this.AITurnsFrozen,
            PlayerTurnsFrozen = this.PlayerTurnsFrozen,
            AIPowerups = aiPowerupsCopy,
            PlayerPowerups = playerPowerupsCopy,
            MazeGrid = mazeGridCopy,
            MazeRows = this.MazeRows,
            MazeColumns = this.MazeColumns
        };
    }
}
```

***MinimaxController ( script )***

```
using UnityEngine;
using System.Collections.Generic;
using System.Linq;

[RequireComponent(typeof(AIController))]
public class MinimaxController : MonoBehaviour
{

    /// Looks at a game state and generates a complete list of all
possible actions
    /// for the currently active character (AI or Player).

    /// Gets a game state to analyze, and whose turn is it

    /// Returns a list of all valid AIAction objects.


    private List<AIAction> GetPossibleActions(MinimaxGameState state,
bool isAITurn)
    {
        var actions = new List<AIAction>();
        var characterPos = isAITurn ? state.AIPos : state.PlayerPos;
        var powerups = isAITurn ? state.AIPowerups :
state.PlayerPowerups;

        if ((isAITurn && state.AITurnsFrozen > 0) || (!isAITurn &&
state.PlayerTurnsFrozen > 0))
        {
            return actions;
        }

        // A helper to perform the strict two-way wall check
```

```
        System.Func<Direction, bool> isPathClear = (dir) => {
            Vector2Int neighborPos = characterPos +
GetVectorFromDirection(dir);
            if (neighborPos.x < 0 || neighborPos.x >= state.MazeColumns
|| neighborPos.y < 0 || neighborPos.y >= state.MazeRows)
                return false; // Out of bounds

            MazeCell currentCell = state.MazeGrid[characterPos.y,
characterPos.x];
            MazeCell neighborCell = state.MazeGrid[neighborPos.y,
neighborPos.x];

            if (dir == Direction.Front && !currentCell.WallFront &&
!neighborCell.WallBack) return true;
            if (dir == Direction.Back && !currentCell.WallBack &&
!neighborCell.WallFront) return true;
            if (dir == Direction.Right && !currentCell.WallRight &&
!neighborCell.WallLeft) return true;
            if (dir == Direction.Left && !currentCell.WallLeft &&
!neighborCell.WallRight) return true;

            return false;
        };

        // ACTION TYPE 1: Standard Moves

        if (isPathClear(Direction.Front)) actions.Add(new AIAction {
Type = ActionType.Move, MoveDirection = Direction.Front });
        if (isPathClear(Direction.Back)) actions.Add(new AIAction {
Type = ActionType.Move, MoveDirection = Direction.Back });
        if (isPathClear(Direction.Right)) actions.Add(new AIAction {
Type = ActionType.Move, MoveDirection = Direction.Right });
        if (isPathClear(Direction.Left)) actions.Add(new AIAction {
Type = ActionType.Move, MoveDirection = Direction.Left });

        // ACTION TYPE 2: Power-Up Usage

        for (int i = 0; i < powerups.Count; i++)
        {
            Powerup powerup = powerups[i];

            System.Action<Direction> addDirectionalPowerup = (dir) => {
```

```
                Vector2Int targetPos = characterPos +
GetVectorFromDirection(dir);
                if (targetPos.x < 0 || targetPos.x >= state.MazeColumns
|| targetPos.y < 0 || targetPos.y >= state.MazeRows)
                    return; // Target is out of bounds

                bool wallExists = !isPathClear(dir);

                if (powerup.Type == PowerupType.BreakWall &&
wallExists)
                {
                    actions.Add(new AIAction { Type =
ActionType.UsePowerup, PowerupSlot = i, PowerupType = powerup.Type,
PowerupTargetDirection = dir });
                }
                else if (powerup.Type == PowerupType.Jump &&
wallExists)
                {
                    actions.Add(new AIAction { Type =
ActionType.UsePowerup, PowerupSlot = i, PowerupType = powerup.Type,
PowerupTargetDirection = dir });
                }
                else if (powerup.Type == PowerupType.Dash) // Dash can
be used on open paths
                {
                    actions.Add(new AIAction { Type =
ActionType.UsePowerup, PowerupSlot = i, PowerupType = powerup.Type,
PowerupTargetDirection = dir });
                }
            };

            switch (powerup.Type)
            {
                case PowerupType.BreakWall:
                case PowerupType.Jump:
                case PowerupType.Dash:
                    addDirectionalPowerup(Direction.Front);
                    addDirectionalPowerup(Direction.Back);
                    addDirectionalPowerup(Direction.Right);
                    addDirectionalPowerup(Direction.Left);
                    break;

                case PowerupType.Freeze:
```

```
                case PowerupType.Teleport:
                    actions.Add(new AIAction { Type =
ActionType.UsePowerup, PowerupSlot = i, PowerupType = powerup.Type });
                    break;
            }
        }
        return actions;
    }


    /// The simulation engine. Takes a state and an action, and returns
the resulting new state.

    public MinimaxGameState GetStateAfterAction(MinimaxGameState
previousState, AIAction action, bool isAITurn)
    {
        // Creating a copy for the state to sandbox it

        MinimaxGameState newState = previousState.Clone();

        var actingPowerups = isAITurn ? newState.AIPowerups :
newState.PlayerPowerups;

        if (action.Type == ActionType.Move)
        {
            Vector2Int newPos = (isAITurn ? newState.AIPos :
newState.PlayerPos) + GetVectorFromDirection(action.MoveDirection);
            if (isAITurn) newState.AIPos = newPos;
            else newState.PlayerPos = newPos;
        }
        else if (action.Type == ActionType.UsePowerup)
        {
            // The powerup is consumed

            Powerup usedPowerup = actingPowerups[action.PowerupSlot];
            actingPowerups.RemoveAt(action.PowerupSlot);

            switch (action.PowerupType)
            {
                case PowerupType.Jump:
                    Vector2Int jumpPos = (isAITurn ? newState.AIPos :
newState.PlayerPos) +
GetVectorFromDirection(action.PowerupTargetDirection);
```

```
                    if (isAITurn) newState.AIPos = jumpPos;
                    else newState.PlayerPos = jumpPos;
                    break;

                case PowerupType.Freeze:
                    if (isAITurn) newState.PlayerTurnsFrozen =
usedPowerup.FreezeDuration;
                    else newState.AITurnsFrozen =
usedPowerup.FreezeDuration;
                    break;

                case PowerupType.BreakWall:
                    Vector2Int breakPos = isAITurn ? newState.AIPos :
newState.PlayerPos;
                    SimulateWallBreak(newState, breakPos,
action.PowerupTargetDirection);
                    break;

                case PowerupType.Teleport:
                    Vector2Int goalPos = new
Vector2Int(newState.MazeColumns - 1, newState.MazeRows - 1);

                    // For a deterministic simulation, we assume
Teleport sends the opponent
                    // to the worst possible spot for them (furthest
from the goal).

                    Vector2Int newTeleportPos =
FindFurthestCellSimulated(goalPos, newState);

                    if (isAITurn) newState.PlayerPos = newTeleportPos;
// AI teleports Player
                    else newState.AIPos = newTeleportPos; // Player
teleports AI
                    break;

                case PowerupType.Dash:
                    Vector2Int startPos = isAITurn ? newState.AIPos :
newState.PlayerPos;
                    List<Vector2Int> path =
CalculateDashPathSimulated(startPos, action.PowerupTargetDirection,
newState);
```

```
                    Vector2Int endPos = (path.Count > 0) ? path.Last()
: startPos;

                    if (isAITurn) newState.AIPos = endPos;
                    else newState.PlayerPos = endPos;
                    break;
            }
        }

        // Return the modified state, representing the future.
        return newState;
    }

    // Helper methods for simulation

    private void SimulateWallBreak(MinimaxGameState state, Vector2Int
pos, Direction dir)
    {
        Vector2Int neighborPos = pos;
        Direction oppositeDir = Direction.Start;

        switch (dir)
        {
            case Direction.Front: neighborPos.y++; oppositeDir =
Direction.Back; break;
            case Direction.Back: neighborPos.y--; oppositeDir =
Direction.Front; break;
            case Direction.Right: neighborPos.x++; oppositeDir =
Direction.Left; break;
            case Direction.Left: neighborPos.x--; oppositeDir =
Direction.Right; break;
        }

        // Check bounds

        if (neighborPos.x < 0 || neighborPos.x >= state.MazeColumns ||
neighborPos.y < 0 || neighborPos.y >= state.MazeRows) return;

        // Update walls in the cloned maze grid

        state.MazeGrid[pos.y, pos.x].SetWall(dir, false);
        state.MazeGrid[neighborPos.y,
neighborPos.x].SetWall(oppositeDir, false);
```

```
    }

    private Vector2Int GetVectorFromDirection(Direction dir)
    {
        switch (dir)
        {
            case Direction.Front: return Vector2Int.up;
            case Direction.Back: return Vector2Int.down;
            case Direction.Right: return Vector2Int.right;
            case Direction.Left: return Vector2Int.left;
            default: return Vector2Int.zero;
        }
    }


    public List<Vector2Int> FindShortestPathSimulated(Vector2Int
startNode, Vector2Int endNode, MinimaxGameState state)
    {
        Queue<Vector2Int> queue = new Queue<Vector2Int>();
        Dictionary<Vector2Int, Vector2Int?> visited = new
Dictionary<Vector2Int, Vector2Int?>();

        if (startNode == endNode) return new List<Vector2Int> {
startNode };

        queue.Enqueue(startNode);
        visited[startNode] = null;

        while (queue.Count > 0)
        {
            Vector2Int currentNode = queue.Dequeue();
            if (currentNode == endNode) break;

            MazeCell currentCell = state.MazeGrid[currentNode.y,
currentNode.x];

            // Helper to perform a strict two-way check within the
simulated state

            System.Action<Direction> checkNeighbor = (dir) => {
                Vector2Int neighborNode = currentNode +
GetVectorFromDirection(dir);
```

```
                // Check bounds first

                if (neighborNode.x < 0 || neighborNode.x >=
state.MazeColumns || neighborNode.y < 0 || neighborNode.y >=
state.MazeRows) return;

                MazeCell neighborCell = state.MazeGrid[neighborNode.y,
neighborNode.x];
                bool isPathClear = false;

                if (dir == Direction.Front && !currentCell.WallFront &&
!neighborCell.WallBack) isPathClear = true;
                else if (dir == Direction.Back && !currentCell.WallBack
&& !neighborCell.WallFront) isPathClear = true;
                else if (dir == Direction.Right &&
!currentCell.WallRight && !neighborCell.WallLeft) isPathClear = true;
                else if (dir == Direction.Left && !currentCell.WallLeft
&& !neighborCell.WallRight) isPathClear = true;

                if (isPathClear && !visited.ContainsKey(neighborNode))
                {
                    visited[neighborNode] = currentNode;
                    queue.Enqueue(neighborNode);
                }
            };

            checkNeighbor(Direction.Front);
            checkNeighbor(Direction.Back);
            checkNeighbor(Direction.Right);
            checkNeighbor(Direction.Left);
        }

        List<Vector2Int> path = new List<Vector2Int>();
        Vector2Int? pathNode = endNode;
        while (pathNode != null && visited.ContainsKey(pathNode.Value))
        {
            path.Add(pathNode.Value);
            pathNode = visited[pathNode.Value];
        }
        path.Reverse();

        return (path.Count > 0 && path[0] == startNode) ? path : null;
    }
```

```
    private Vector2Int FindFurthestCellSimulated(Vector2Int startNode,
MinimaxGameState state)
    {
        Queue<Vector2Int> queue = new Queue<Vector2Int>();
        HashSet<Vector2Int> visited = new HashSet<Vector2Int>();
        queue.Enqueue(startNode);
        visited.Add(startNode);
        Vector2Int furthestNode = startNode;

        while (queue.Count > 0)
        {
            Vector2Int currentNode = queue.Dequeue();
            furthestNode = currentNode;
            MazeCell currentCell = state.MazeGrid[currentNode.y,
currentNode.x];

            // Check neighbors (abbreviated for clarity)
            Vector2Int[] neighbors = new Vector2Int[] {
                new Vector2Int(currentNode.x + 1, currentNode.y), new
Vector2Int(currentNode.x - 1, currentNode.y),
                new Vector2Int(currentNode.x, currentNode.y + 1), new
Vector2Int(currentNode.x, currentNode.y - 1)
            };
            if (!currentCell.WallRight &&
!visited.Contains(neighbors[0])) { visited.Add(neighbors[0]);
queue.Enqueue(neighbors[0]); }
            if (!currentCell.WallLeft &&
!visited.Contains(neighbors[1])) { visited.Add(neighbors[1]);
queue.Enqueue(neighbors[1]); }
            if (!currentCell.WallFront &&
!visited.Contains(neighbors[2])) { visited.Add(neighbors[2]);
queue.Enqueue(neighbors[2]); }
            if (!currentCell.WallBack &&
!visited.Contains(neighbors[3])) { visited.Add(neighbors[3]);
queue.Enqueue(neighbors[3]); }
        }
        return furthestNode;
    }


    /// The Agent's "brain". Analyzes a game state and returns a score,
higher scores are better for the AI.
```

```
    private float EvaluateState(MinimaxGameState state)
    {
        Vector2Int goalPos = new Vector2Int(state.MazeColumns - 1,
state.MazeRows - 1);

        //  Terminal State Check (Game Over Conditions are Absolute)

        if (state.AIPos == state.PlayerPos) return 10000f;
        if (state.PlayerPos == goalPos) return -10000f;
        if (state.AIPos == goalPos) return -5000f;

        //  Path Calculations

        int aiPathToPlayer = FindShortestPathSimulated(state.AIPos,
state.PlayerPos, state)?.Count ?? 999;
        int playerPathToGoal =
FindShortestPathSimulated(state.PlayerPos, goalPos, state)?.Count ??
999;

        //  Core Chase Scoring

        float finalScore = -aiPathToPlayer * 10f;

        // URGENCY HEURISTIC
        // If the player is very close to winning, the AI must panic.

        if (playerPathToGoal < 5)
        {
            // The penalty increases exponentially the closer the
player gets.
            // 4 steps away = -100 penalty. 1 step away = -400 penalty.
            // This will force the AI to prioritize stopping the player
above all else.

            float urgencyPenalty = (5 - playerPathToGoal) * 100f;
            finalScore -= urgencyPenalty;
        }

        // Strategic Modifiers for Power-ups
        if (state.PlayerTurnsFrozen > 0)
        {
```

```
            finalScore += 80f;
        }
        if (state.AITurnsFrozen > 0)
        {
            finalScore -= 80f;
        }

        return finalScore;
    }



    // From here the agent pulls the best possible action it can
preform


    public AIAction GetBestAction(MinimaxGameState currentState, int
depth)
    {
        AIAction bestAction = null;
        float bestScore = float.MinValue;

        // Check for an immediate win before any complex thinking

        foreach (AIAction immediateAction in
GetPossibleActions(currentState, true))
        {
            if (immediateAction.Type == ActionType.Move)
            {
                if (currentState.AIPos +
GetVectorFromDirection(immediateAction.MoveDirection) ==
currentState.PlayerPos)
                {
                    Debug.Log("[AI STRATEGY]: Found an immediate kill
shot. Overriding Minimax.");
                    immediateAction.Score = 10000f;
                    return immediateAction;
                }
            }
        }

        //  Setup for Tie-breaker and willingness factor
```

```
        List<Vector2Int> idealPath =
FindShortestPathSimulated(currentState.AIPos, currentState.PlayerPos,
currentState);
        Vector2Int idealNextStep = (idealPath != null &&
idealPath.Count > 1) ? idealPath[1] : currentState.AIPos;

        // Willingness - how hard does the agent want to user its
powerups?

        Vector2Int goalPos = new Vector2Int(currentState.MazeColumns -
1, currentState.MazeRows - 1);
        int playerPathToGoal =
FindShortestPathSimulated(currentState.PlayerPos, goalPos,
currentState)?.Count ?? 999;

        // The agent's willingness to use power-ups scales from 20% to
100% based on the player's proximity to the goal.

        float willingness = Mathf.InverseLerp(30f, 5f,
playerPathToGoal); // Ranges from 0.0 (far) to 1.0 (close)
        float willingnessModifier = 0.2f + (willingness * 0.8f); //
Scales from a base of 0.2 up to 1.0

        //  Main Evaluation Loop

        foreach (AIAction action in GetPossibleActions(currentState,
true))
        {
            MinimaxGameState newState =
GetStateAfterAction(currentState, action, true);
            float score = Minimax(newState, depth - 1, false);
            float strategicBonus = 0f;

            if (action.Type == ActionType.UsePowerup)
            {
                switch (action.PowerupType)
                {
                    case PowerupType.BreakWall:
                    case PowerupType.Jump:
                        int pathLengthBefore = idealPath?.Count ?? 999;
                        int pathLengthAfter =
FindShortestPathSimulated(newState.AIPos, newState.PlayerPos,
newState)?.Count ?? 999;
```

```
                        int pathImprovement = pathLengthBefore -
pathLengthAfter;
                        if (pathImprovement >= 4) { strategicBonus =
pathImprovement * 15f; }
                        break;
                    case PowerupType.Teleport:
                        int playerPathToGoalBefore = playerPathToGoal;
                        int playerPathToGoalAfter =
FindShortestPathSimulated(newState.PlayerPos, goalPos, newState)?.Count
?? 999;
                        int defensiveImprovement =
playerPathToGoalAfter - playerPathToGoalBefore;
                        if (defensiveImprovement > 0) { strategicBonus
= defensiveImprovement * 10f; }
                        break;
                    case PowerupType.Freeze:
                        strategicBonus = 45f;
                        break;
                }

                // The calculated bonus is now scaled by the AI's
current "willingness" to act.

                score += strategicBonus * willingnessModifier;
            }

            // Apply tie-breaker for moves

            if (action.Type == ActionType.Move && (currentState.AIPos +
GetVectorFromDirection(action.MoveDirection) == idealNextStep))
            {
                score += 0.01f;
            }

            if (score > bestScore)
            {
                bestScore = score;
                action.Score = score;
                bestAction = action;
            }
        }
        return bestAction;
    }
```

```
    // The Core Recursive Algorithm

    private float Minimax(MinimaxGameState state, int depth, bool
isMaximizingPlayer)
    {
        // Base Case: If we've reached max depth or the game is over,
return the state's value

        if (depth == 0)
        {
            return EvaluateState(state);
        }
        // A quick check to see if the game ended prematurely

        float terminalScore = EvaluateState(state);
        if (Mathf.Abs(terminalScore) >= 1000f)
        {
            return terminalScore;
        }

        List<AIAction> possibleActions = GetPossibleActions(state,
isMaximizingPlayer);
        if (possibleActions.Count == 0)
        {
            return EvaluateState(state);
        }

        // Recursive Step

        if (isMaximizingPlayer) // Agent's turn
        {
            float bestScore = float.MinValue;
            foreach (var action in possibleActions)
            {
                MinimaxGameState newState = GetStateAfterAction(state,
action, true);
                float score = Minimax(newState, depth - 1, false);
                bestScore = Mathf.Max(bestScore, score);
            }
            return bestScore;
        }
        else // The Player's turn
```

```
        {
            float bestScore = float.MaxValue;
            foreach (var action in possibleActions)
            {
                MinimaxGameState newState = GetStateAfterAction(state,
action, false);
                float score = Minimax(newState, depth - 1, true);
                bestScore = Mathf.Min(bestScore, score);
            }
            return bestScore;
        }
    }

    private List<Vector2Int> CalculateDashPathSimulated(Vector2Int
startPos, Direction direction, MinimaxGameState state)
    {
        var path = new List<Vector2Int>();
        var currentPos = startPos;

        // Loop for a maximum distance to prevent infinite loops in odd
cases

        for (int i = 0; i < Mathf.Max(state.MazeRows,
state.MazeColumns); i++)
        {
            Vector2Int neighborPos = currentPos +
GetVectorFromDirection(direction);

            // Stop if the next tile is out of bounds

            if (neighborPos.x < 0 || neighborPos.x >= state.MazeColumns
|| neighborPos.y < 0 || neighborPos.y >= state.MazeRows)
            {
                break;
            }

            // Perform the same strict, two-way wall check

            MazeCell currentCell = state.MazeGrid[currentPos.y,
currentPos.x];
            MazeCell neighborCell = state.MazeGrid[neighborPos.y,
neighborPos.x];
            bool isPathClear = false;
```

```
            if (direction == Direction.Front && !currentCell.WallFront
&& !neighborCell.WallBack) isPathClear = true;
            else if (direction == Direction.Back &&
!currentCell.WallBack && !neighborCell.WallFront) isPathClear = true;
            else if (direction == Direction.Right &&
!currentCell.WallRight && !neighborCell.WallLeft) isPathClear = true;
            else if (direction == Direction.Left &&
!currentCell.WallLeft && !neighborCell.WallRight) isPathClear = true;

            // If a wall is in the way, the dash stops.

            if (!isPathClear)
            {
                break;
            }

            // Otherwise, add the tile to the path and continue from
the new position.

            path.Add(neighborPos);
            currentPos = neighborPos;
        }
        return path;
    }
}
```