*Lab Exercise: A Pathfinding in Unity**

*Gentle Introduction to the A (“A Star”) Algorithm**

The A* algorithm (often pronounced “A star”) was originally invented in 1968 by computer scientists **Nils Nilsson**, **Peter Hart**, and **Bertram Raphael**. At the time, they were working on the Shakey the Robot research project at Stanford, developing a navigation system that could find optimized paths through an environment. They had been experimenting with two algorithms, which they simply referred to as “A1” and “A2.” It turned out that “A2” performed best, so they named it “A*.”

Over the years, A* has become synonymous with pathfinding in many areas of computer science, especially **artificial intelligence (AI)** for games. It’s widely used for planning character actions or determining the best route through a game world, although its applications extend far beyond gaming.

Despite its importance, A* can seem intimidating when you first learn about it—especially when you start coding it from scratch. Many developers opt to use existing libraries without really understanding the underlying mechanics. However, A* is actually quite intuitive. The key idea is this: the algorithm always tries to expand the path that appears **most likely** to lead to the goal, based on both how far you’ve traveled so far and how far you likely have left to go.

A Simple Maze Example

Imagine a grid-based maze with a **start** position (S) in one corner and a **goal** (G) in another. You can only move up, down, left, or right. If you were asked to sketch a path from S to G, you’d probably try to move *closer* to G at every step. But what happens if there’s a wall blocking that seemingly direct route? Once you realize there’s an obstacle, you might need to backtrack a bit and try a different path—maybe even move away from G initially—until you can circumvent the obstacle and keep moving closer again.

This sort of reasoning is exactly what A* does in an automated way.

The G, H, and F Values

A* keeps track of three main values for each grid cell (or *node*):

1. **G:** The cost (or distance) from the start node to the current node.
2. **H:** The estimated cost from the current node to the goal (often called the *heuristic*).
3. **F:** The sum of G and H, i.e., **F = G + H**.

When A* explores the grid, it looks at the neighbors of the current position and calculates these values for each neighbor. The key is to pick the neighbor with the **lowest F** value to explore next, because a small F suggests a promising route—both relatively close to the start (low G) and seemingly close to the goal (low H).

Estimating "H"

The heuristic (H) can be calculated in different ways. Two common heuristics are:

- **Manhattan Distance** (when you can only move in four directions—no diagonals).

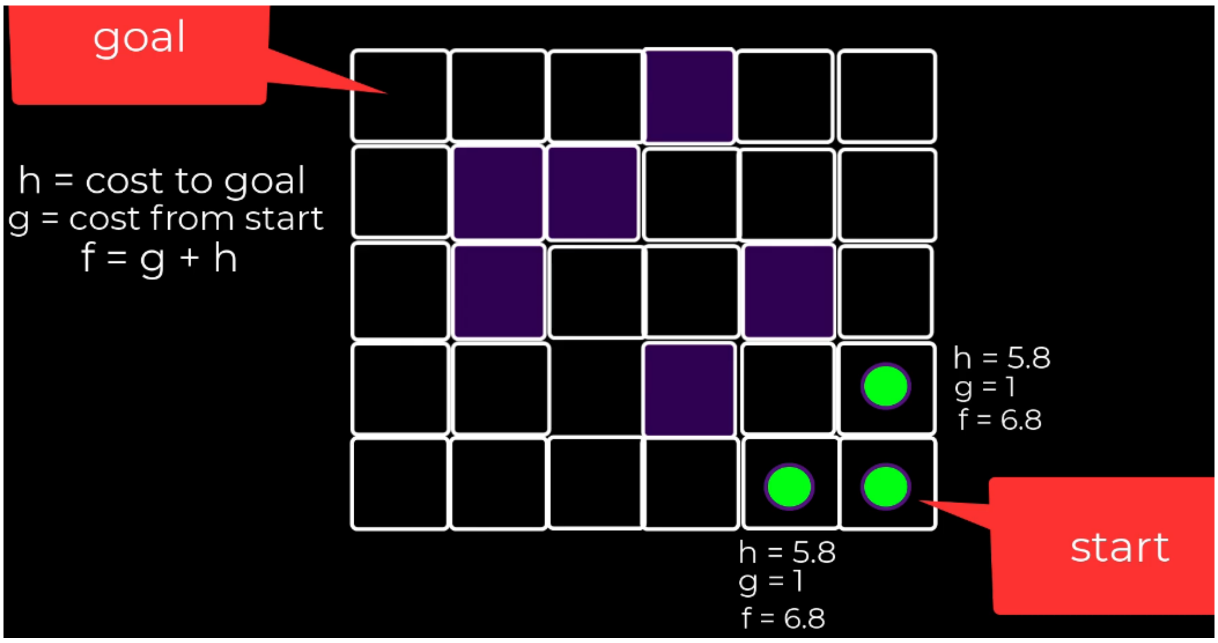
  Manhattan distance

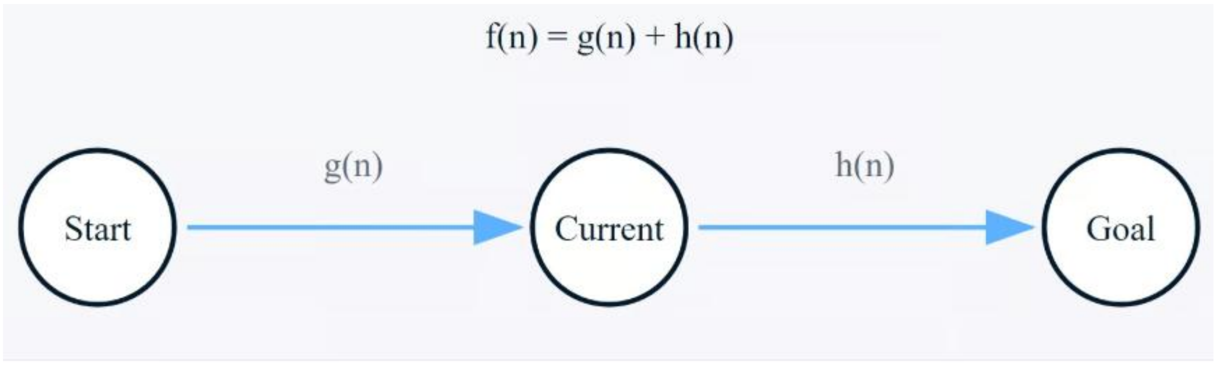
  $$h(n) = |x_1 - x_2| + |y_1 - y_2|$$

- **Euclidean Distance** (straight-line distance, which might be used if diagonal movement is allowed).

  Euclidean distance

  $$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$





A algorithm Cost Function*

Open and Closed Lists

To manage which nodes get explored, A* uses two sets:

- **Open List:** Nodes that have been discovered and are waiting to be evaluated.
- **Closed List:** Nodes already visited and finalized.

Initially, the start node goes on the open list. A* repeatedly takes the node from the open list that has the lowest F value and moves it to the closed list. When a node is expanded (processed), its neighbors are added to the open list (if they haven't already been processed). This continues until the goal is found or the open list is empty (meaning no path exists).

The A* algorithm maintains two essential lists

Open list:

- Contains nodes that need to be evaluated
- Sorted by f(n) value (lowest first)
- New nodes are added as they're discovered

Closed list:

- Contains already evaluated nodes
- Helps avoid re-evaluating nodes
- Used to reconstruct the final path

The algorithm continually selects the node with the lowest f(n) value from the open list, evaluates it, and moves it to the closed list until it reaches the goal node or determines no path exists.

Main loop

The core of A* is its main loop, which continues until either:

- The goal is reached (success)
- The open list becomes empty (failure - no path exists)

During each iteration, the algorithm:

1. Selects the most promising node (lowest f value) from the open list
2. Moves it to the closed list
3. Examines all neighboring nodes

Neighbor evaluation

For each neighbor, the algorithm:

- Skips nodes already in the closed list
- Calculates a tentative g score
- Updates node values if a better path is found
- Adds new nodes to the open list

Walking Through an Example

1. **Start at S:**
    - G = 0 (we haven't traveled anywhere yet)
    - H = distance from S to G (the heuristic)
    - F = G + H
2. **Evaluate Neighbors:**
   For each walkable neighbor (up, down, left, right), calculate its G, H, and F. Put these neighbors on the open list.
3. **Pick the Next Node:**
   From the open list, choose the node with the lowest F. Move it to the closed list (meaning we won't revisit it).
4. **Repeat:**
    - Expand the chosen node's neighbors.
    - If a neighbor isn't on the open or closed list, add it (with its calculated G, H, F).

    - If it’s already on the open list but you just found a path with a lower G (better path from S), update its G and F, and change its parent to the current node.

5. **Goal Found:**
   Continue until you finally pick the goal node from the open list. At that point, you can trace back through each node’s “parent” to reconstruct the path.

In practice, as the algorithm progresses, it “fans out” around the start position and tends to focus more on exploring nodes that seem closer to the goal. It may occasionally “backtrack” or expand nodes that initially looked less promising, but only when it needs to verify whether a longer path might actually become shorter due to walls or other obstacles.

Why A* Is Powerful

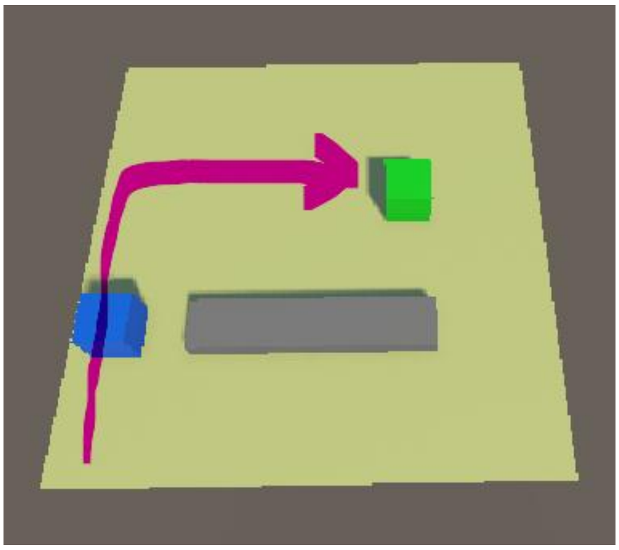
A* finds *optimal* paths given a properly designed heuristic (one that never overestimates the distance to the goal). It achieves this without looking too far ahead in unnecessary directions, making it both efficient and accurate for many pathfinding problems. In addition to games, A* is used for navigation in robotics, GIS (geographic information systems), and various route-planning applications.

While you might rely on existing libraries, understanding how A* works underneath helps you debug pathfinding issues and even tailor the algorithm to specific needs—like factoring in different types of costs (e.g., terrain difficulty or damage to a game character).

Objective

Create a Unity project that visualizes the *A pathfinding algorithm** to find the optimal path in a grid-based maze.

Create this environment:


Blue cube = player

Green cube = goal

The goal is to find the optimal path with the A* algorithm so the player will move automatically to the goal.

1/ Your task:

Try to understand the maze.cs & FindPathAStar.cs

```
 void Update() {
    if (Input.GetKeyDown(KeyCode.P)) {
       BeginSearch();
       hasStarted = true;
    }
    if (hasStarted)
       if (Input.GetKeyDown(KeyCode.C)) Search(lastPos);
 }
```

In the input method you see that we build up the closed list of the A* algorithm. Try to create the closed list.

2/ Your task: Now you do one search iteration with a key press. Your task is after the start positions are set, the search algorithm will calculate the best path and your player will follow the best path after searching automatically.

Your task 2.1: *Develop a path reconstruction method:*

Once the goal is reached, the algorithm works backward through the parent references to construct the optimal path from start to goal.

This systematic approach ensures that A* will always find the optimal path if:

1. A path actually exists between the start and goal nodes

Use the close list, and go from goal node to start node through the parents.

Move through the path with a coroutine iterating per second node per node.

Your task2.2: *Implement the system automically start the search once the start and end goal is selected.*

Your task2.3: *Implement a method so the player will walk automatically (after the A*search) to the goal (use a coroutine, so you do 1 step per second).*

Step 1: Unity Setup

1. **Scene Setup:**
    - Add a **Plane** to act as the ground (GameObject > 3D Object > Plane).
2. **Create Prefabs:**
    - **Blue Cube:** Represents the **player**.
    - **Green Cube:** Represents the **goal**.
    - **Path Cube:** Represents the path visualization.
3. **Materials:**
    - Create blue and green materials for the player and goal.

Step 2: Scripts

MapLocation.cs

Defines a grid location.

```
using UnityEngine;

public struct MapLocation
{
    public int x, z;

    public MapLocation(int x, int z)
    {
        this.x = x;
        this.z = z;
    }

    public override bool Equals(object obj)
    {
        if (obj is MapLocation ml)
        {
            return ml.x == x && ml.z == z;
        }
        return false;
    }

    public override int GetHashCode()
    {
        return x ^ z;
    }
}
```

PathMarker.cs

Represents a grid node.

```
using UnityEngine;

public class PathMarker
{
    public MapLocation location;
    public float G, H, F;
    public GameObject marker;
    public PathMarker parent;

    public PathMarker(MapLocation l, float g, float h, float f,
    {
        location = l;
        G = g;
        H = h;
        F = f;
        this.marker = marker;
        parent = p;
    }

    public override bool Equals(object obj)
    {
        if (obj is PathMarker pm)
        {
            return location.Equals(pm.location);
        }
        return false;
    }
}
```

We override Equals() to properly compare PathMarker objects. We simply compare their underlying location, which in turn has its own Equals() method.

Maze.cs

Generates the grid-based maze.

```
public class Maze : MonoBehaviour
{
    public int width = 10, depth = 10;
    public int[,] map;

    void Start()
    {
        GenerateMaze();
    }

    public void GenerateMaze()
    {
        map = new int[width, depth];
        for (int x = 0; x < width; x++)
        {
            for (int z = 0; z < depth; z++)
            {
                map[x, z] = Random.Range(0, 3) == 0 ? 1 : 0; //
                if (map[x, z] == 1)
                {
                    Instantiate(GameObject.CreatePrimi
                }
            }
        }
    }
```

```
public List<MapLocation> GetNeighbors(MapLocation loc)
{
    List<MapLocation> neighbors = new List<MapLocation>();
    if (loc.x > 0 && map[loc.x - 1, loc.z] == 0) neighbors.A
    if (loc.x < width - 1 && map[loc.x + 1, loc.z] == 0) nei
    if (loc.z > 0 && map[loc.x, loc.z - 1] == 0) neighbors.A
    if (loc.z < depth - 1 && map[loc.x, loc.z + 1] == 0) nei
    return neighbors;
}
```

FindPathAStar.cs

Implements the A* algorithm. we'll write the **FindPathAStar** class that uses PathMarker objects to systematically explore the maze. A* will keep track of two key lists:

- **Open List:** Nodes that are candidates for exploration.
- **Closed List:** Nodes we've already visited and finalized.

We'll calculate **G**, **H**, and **F** values for each neighbor, pick the node with the lowest **F**, and continue until we reach our goal.

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

public class FindPathAStar : MonoBehaviour
{
    public GameObject startPrefab, goalPrefab, pathPrefab;
    public Maze maze;

    private List<PathMarker> open = new List<PathMarker>();
    private List<PathMarker> closed = new List<PathMarker>();

    private PathMarker startNode, goalNode, lastPos;
    private bool done = false;

    void Start()
    {
        BeginSearch();
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.C) && lastPos != null && !
        {
            Search(lastPos);
        }
    }
```

Within our FindPathAStar class, we need to reference the maze itself and prepare a few essential variables for visualization. Here's a quick look at the fields we'll create:

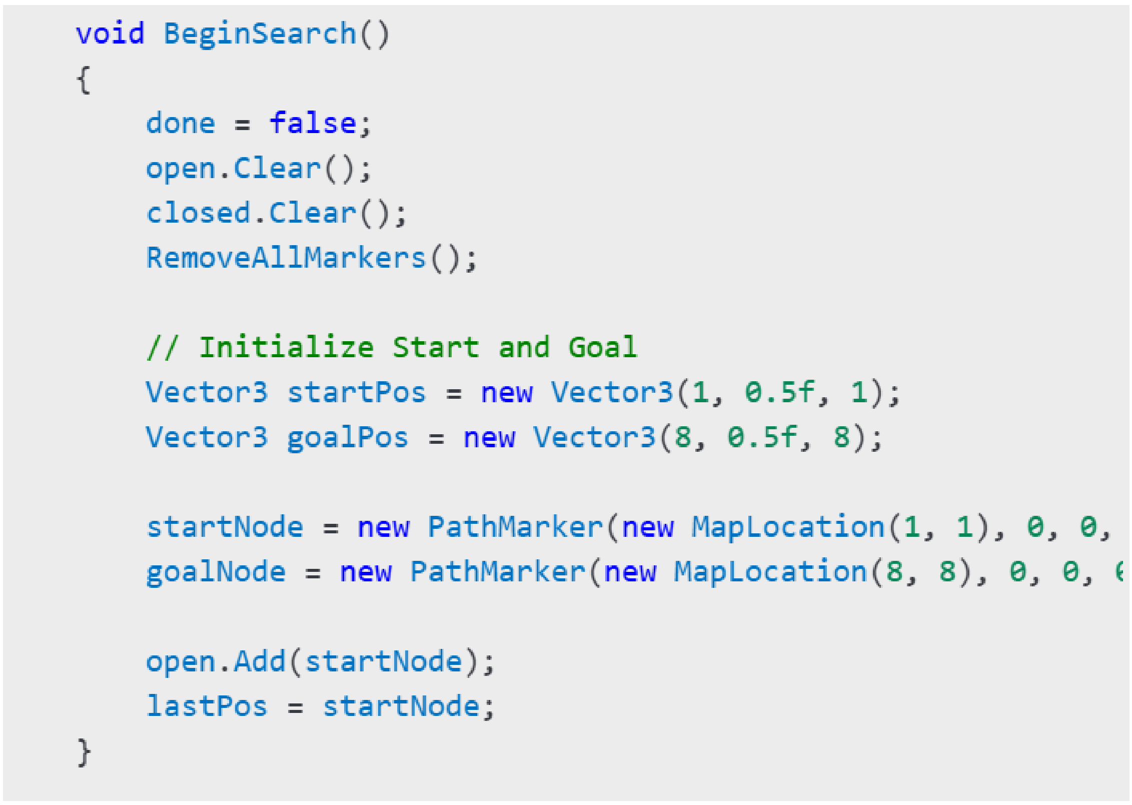
```
public Maze maze;

private List<PathMarker> open = new List<PathMarker>();

private List<PathMarker> closed = new List<PathMarker>();


public GameObject start;

public GameObject end;

public GameObject pathP;


private PathMarker startNode;

private PathMarker goalNode;

private PathMarker lastPos;

private bool done = false;
```

- **maze**: Points to the script that generates our maze.
- **open** & **closed**: The two critical lists in A*:
    - **Open** holds nodes that we still need to explore.
    - **Closed** holds nodes that we've finished processing.
- **start**, **end**, **pathP**: Prefabs for displaying the start point, end point, and path markers in the maze.
- **startNode** & **goalNode**: References to our chosen start and goal positions.
- **lastPos**: Tracks the most recently expanded node.
- **done**: A flag to indicate when we've found a path (or completed our search).

Once these variables are in place, make sure to **attach** this script to your Maze GameObject in the Unity hierarchy. Then drag in the required references (the maze script itself, the prefabs for start/end/path, and the materials for open/closed markers) in the Inspector.

Our BeginSearch() method does the heavy lifting to place our **start** and **goal** markers randomly in the maze. Here are the main steps:

1. **Reset** the pathfinding state.
2. **Remove** any existing markers from previous runs.
3. **Gather** all valid (walkable) maze cells and shuffle them.
4. **Pick** the first two cells in the shuffled list as your start and goal positions.
5. **Instantiate** the corresponding markers (start and end prefabs) and create PathMarker objects to represent them.



```
void BeginSearch()
{
    done = false;
    open.Clear();
    closed.Clear();
    RemoveAllMarkers();

    // Initialize Start and Goal
    Vector3 startPos = new Vector3(1, 0.5f, 1);
    Vector3 goalPos = new Vector3(8, 0.5f, 8);

    startNode = new PathMarker(new MapLocation(1, 1), 0, 0,
    goalNode = new PathMarker(new MapLocation(8, 8), 0, 0,

    open.Add(startNode);
    lastPos = startNode;
}
```

Testing the Setup

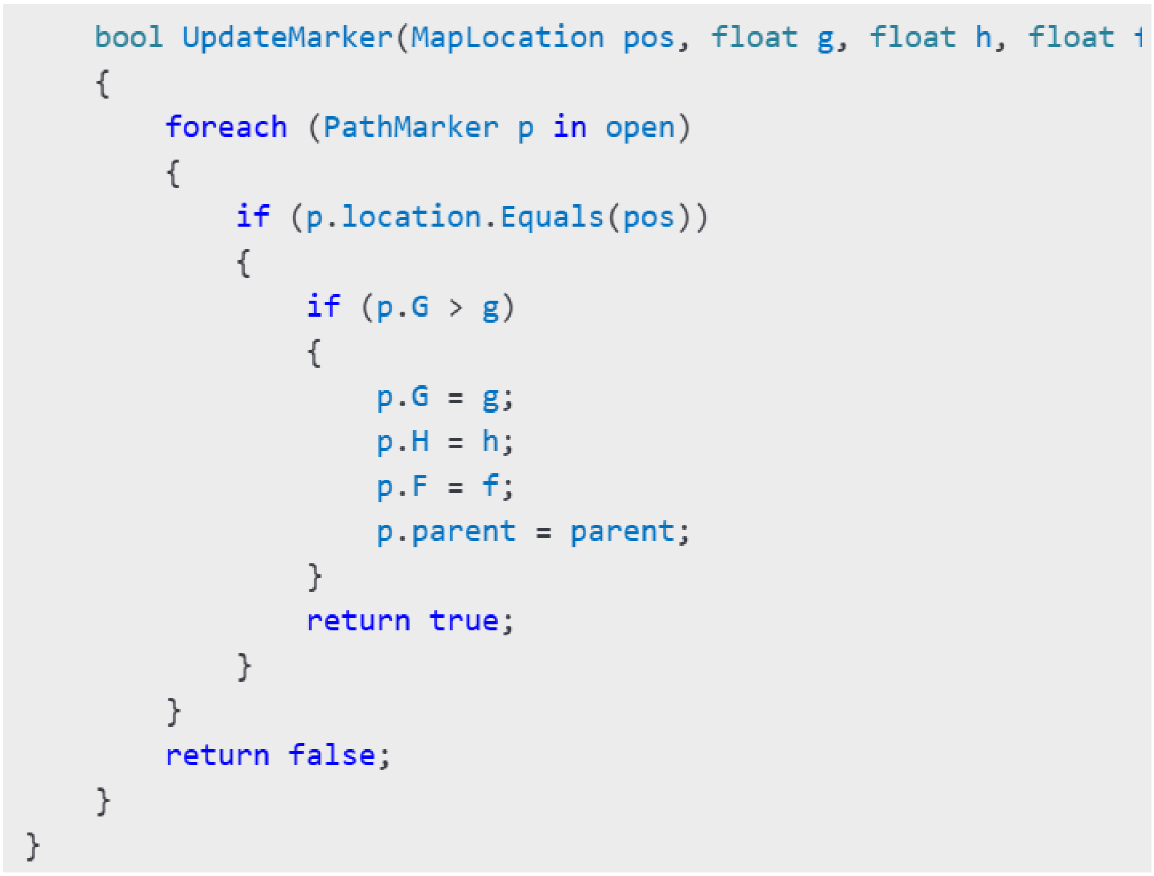
In Update(), add a quick key check to call BeginSearch() whenever you press **P**:

```
void Update()
{
  if (Input.GetKeyDown(KeyCode.P))
  {
     BeginSearch();
  }
}
```

With this in place, press **Play** and **P** in the Game view. You should see your randomly generated maze appear, along with the **Start** and **Goal** markers placed in valid locations.

```
void Search(PathMarker thisNode)
{
    if (thisNode.Equals(goalNode))
    {
        done = true;
        Debug.Log("Goal Found!");
        ShowPath(thisNode);
        return;
    }

    List<MapLocation> neighbors = maze.GetNeighbors(thisNode
    foreach (MapLocation loc in neighbors)
    {
        if (closed.Any(node => node.location.Equals(loc)))
            continue;

        float g = thisNode.G + 1; // Cost from start to neig
        float h = Mathf.Abs(loc.x - goalNode.location.x) + M
        float f = g + h;

        if (!UpdateMarker(loc, g, h, f, thisNode))
        {
            GameObject pathMarker = Instantiate(pathPrefab,
            open.Add(new PathMarker(loc, g, h, f, pathMarker
        }
    }
```

```
        open = open.OrderBy(p => p.F).ToList();
        PathMarker nextNode = open[0];
        open.RemoveAt(0);
        closed.Add(nextNode);
        lastPos = nextNode;
    }

    void ShowPath(PathMarker node)
    {
        while (node != null)
        {
            Instantiate(pathPrefab, new Vector3(node.location.x,
            node = node.parent;
        }
    }

    void RemoveAllMarkers()
    {
        GameObject[] markers = GameObject.FindGameObjectsWithTag
        foreach (GameObject m in markers)
        {
            Destroy(m);
        }
    }
```



```
    bool UpdateMarker(MapLocation pos, float g, float h, float f
    {
        foreach (PathMarker p in open)
        {
            if (p.location.Equals(pos))
            {
                if (p.G > g)
                {
                    p.G = g;
                    p.H = h;
                    p.F = f;
                    p.parent = parent;
                }
                return true;
            }
        }
        return false;
    }
}
```

Updating Existing Nodes

Before adding a newly found neighbor to the **Open** list, we must check if it already exists there. If it does, we simply **update** its G, H, and F values—no need to create a duplicate marker:

```
bool UpdateMarker(MapLocation pos, float g, float h, float f, PathMarker parent)

{

  foreach (PathMarker p in open)

  {

    if (p.location.Equals(pos))

    {

      p.G = g;

      p.H = h;

      p.F = f;

      p.parent = parent;

      return true;
```

```
        }
    }
    return false;
}
```

This function scans through the **Open** list, looking for an existing marker at the same location. If found, it updates the node and returns true. If not, it returns false so we can add a **new** node instead.

3. Adding Neighbors to the Open List

Inside our main search routine, once we calculate a neighbor's G, H, and F, we do:

```
if (!UpdateMarker(neighbor, g, h, f, node))
{
    // If UpdateMarker returned false, the neighbor wasn't found
    // so we create a new marker and add it to the open list.
    open.Add(new PathMarker(neighbor, g, h, f, pathBlock, node));
}
```

The pathBlock is the newly instantiated visual object for that node. We also keep track of the node that generated this neighbor by setting it as the neighbor's **parent**.

4. Picking the Next Node

With the neighbor processing complete, we sort the Open list so that the node with the **lowest F** ends up at the front. We can use LINQ for this:

```
open = open
    .OrderBy(p => p.F)
    .ThenByDescending(p => p.H) // Ties on F get sorted by H
    .ToList();
```

- **OrderBy** sorts primarily by F.
- **ThenByDescending** provides a secondary sort by H (useful if multiple nodes share the same F).

We then grab the first node (the one with the smallest F) as our next candidate:

```
PathMarker pm = open[0];

closed.Add(pm);

open.RemoveAt(0);
```

Finally, we update lastPos so the search will begin from this new node during the next iteration:

```
lastPos = pm;
```