



A Star Algorithm

Yazan Mahmoud Barakat

411010698

CSC313?

A*

Introduction About Path Finding

Problem To Solve : Path Finding

The idea of path planning do exist in a lot of different fields such as AI in games, robots, and even maps, In the world of game development and while making an AI for a game you will face a famous challenge and It Is how is your AI agent going to move inside your game, There Is a Lot of Technics and Your Solution to Choose May Be Different Depending on The Type of Game You Are Making.

Some Game Engines Have Built in Functionality to Help You with Your AI But in Our Case, We Are Going to Cover One of The Best Algorithms Which is A Star (A*)

What Is A*

A* Is an Algorithm that is mostly use in pathfinding applications and it is one of the most used algorithms for AI inside video games, it is known to be one of the best algorithms for path planning and it uses a combination of heuristic searching and searching based on the shortest path

What Is the Time Complexity Of A*

A star is *Dijkstra with a heuristic* that fulfills some properties You can select different heuristic functions that lead to different time complexities. The simplest heuristic is straight line distance. However, there is also more advanced stuff like landmarks heuristic for example one of the functions can be the next :

$$F(v) = h(v) + g(v)$$

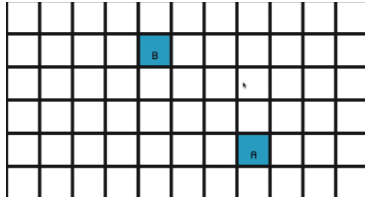
Where $h(v)$ is heuristic distance of the cell to the goal state, and $g(v)$ is the length of the path from the initial state to the goal state through the selected sequence of cells

Advantages vs Disadvantages of A*

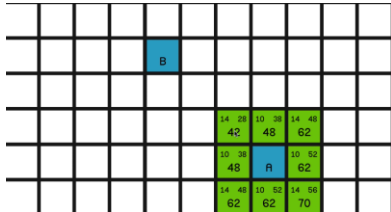
Advantages	Disadvantages
It is complete and optimal.	This algorithm is complete if the branching factor is finite and every action has a fixed cost.
It is the best one from other techniques	The speed execution of the A* search is highly dependent on the accuracy of the heuristic algorithm that is used to compute $h(n)$.
It is optimally efficient, i.e., there is no other optimal algorithm guaranteed to expand fewer nodes than A*.	It has complexity problems.

How A* Works

Let's Consider We Have Grid Based System and We Want to Find the Closest Path Between Point A and B



In This System Each Cell Surrounding the Point a Get 3 values



G Cost: Is: How Far This Cell Is from The Starting Point

H Cost : Is How Far This Cell Is from The End Point

F Cost : Is G Cost + H Cost

The Algorithm Will Chose the Cell with The Lowest F Cost and Will Mark It as Closed (Part of The Path)

And After That It Will Calculate All the Values Again for The New Cell We Have



In the end we will have our path

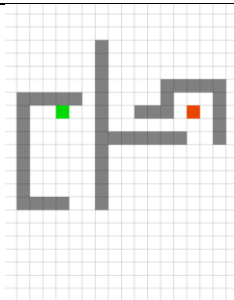
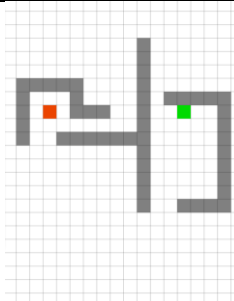
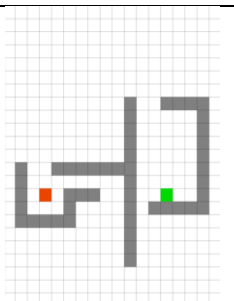


Here Is Another Example as If We Had an Obstacle

				72 10 82	62 14 76	52 24 76	48 34 82	52 44 96		
				68 0 68	58 10 68	48 20 68	38 30 68	34 40 74	38 50 88	
		58 24 82						24 44 68	28 54 82	
		54 28 82	44 24 68	34 20 54	24 24 48	14 28 42	10 38 48	14 48 62	24 58 82	
		58 38 96	40 34 74	30 30 60	20 34 54	10 38 48	10 52 A	10 52 62	20 62 82	
			44 44 88	34 40 74	24 44 68	14 48 62	10 52 62	14 56 70	24 66 90	

Alternative algorithms

There Is Many Different Algorithms for Path Finding Each One with Its Own Pros and Cons for Example

Algorithm	Will Always Return the Shortest Path	Support Movement Cost	Support Multiple Start / End Point	Time Complexity	Visualization
Breadth First Search	Yes	No	Yes	$O(V^2)$	
Dijkstra	Yes	Yes	Yes	$O(V^2)$	
A*	Hard To Decide (heuristic)	Yes	No	$O(bd)$	

Algorithm Pseudocode Code

Open List to Store the Nodes That Is Not evaluated yet

Closed List to Store the Nodes That has been evaluated

Loop

Current Node in open with the lowest f cost

Remove current from Open

Add current to closed

If(current is the end node)

Return path found

Foreach neighbor of the current node

If neighbor is not traversable or neighbor is in closed

Skip to the next neighbor

If new path to neighbor is shorter or neighbor is not in open

Set f cost of neighbor

Set parent of neighbor to current

If neighbor is not in open

Add neighbor to open

To Apply This Algorithm, we would need two more Classes , the first one to identify what is a node in the world and its properties like what is the f cost , g cost , are we allowed to walk here or not so We Created the class Node

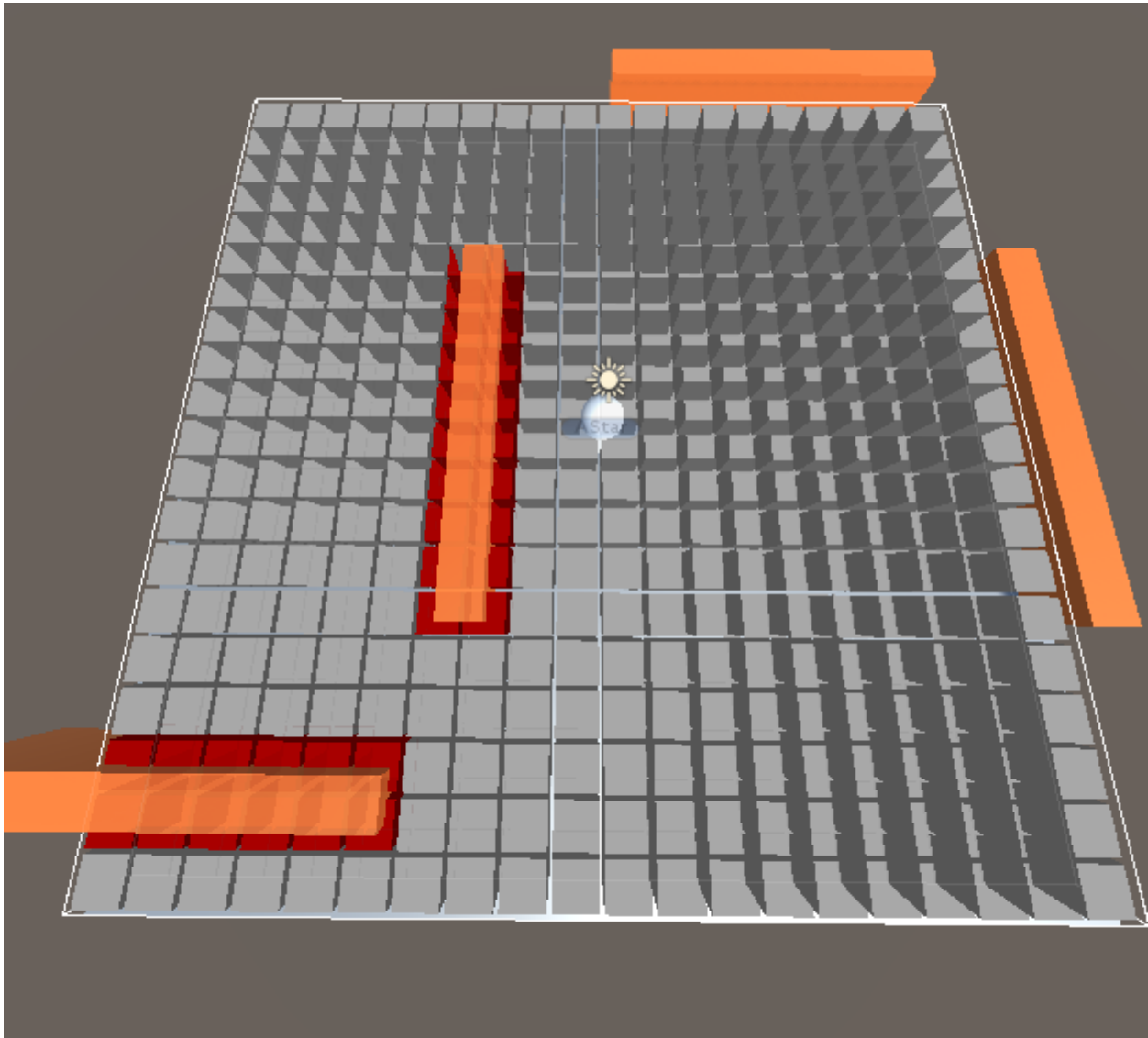
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Node
{
    public int gridX, gridY;
    public int gCost, hCost;

    public int fCost{get { return gCost + hCost;
    }}}

    public Node parent;
    public bool walkable;
    public Vector3 worldNodePos;
    public Node(bool _walkable , Vector3 _worldNodePos , int _gridX , int _gridY)
    {
        gridX = _gridX ;
        gridY = _gridY;
        walkable = _walkable;
        worldNodePos = _worldNodePos;
    }
}
```


And After That We Need Class to Give the Algorithm the world size and it will cut in in a grid and check for each node stats so We Created the Grid Class



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

public class Grid : MonoBehaviour
{
    public LayerMask unwalkableMask;
    public Vector2 gridWorldSize;
    public GameObject walls;
    public float nodeRadius;
    Node[,] grid;

    float nodeDiameter;
    int gridSizeX, gridSizeY;
    public List<Vector3> prevPos = new List<Vector3>();
    public List<GameObject> obs = new List<GameObject>();

    void Start()
```

```

{
    nodeDiameter = nodeRadius * 2;
    gridSizeX = Mathf.RoundToInt(gridWorldSize.x / nodeDiameter);
    gridSizeY = Mathf.RoundToInt(gridWorldSize.y / nodeDiameter);
    CreateGrid();
    GetObjectsInLayer(walls,7);
}
// Update is called every frame, if the MonoBehaviour is enabled.
protected void Update()
{
    for (int i = 0; i < prevPos.Count; i++)
    {
        if(obs[i].transform.position.x != prevPos[i].x)
        {
            CreateGrid();
        }
    }
}
private void GetObjectsInLayer(GameObject root, int layer)
{
    foreach (Transform t in root.GetComponentInChildren<Transform>())
    {
        if (t.gameObject.layer == layer)
        {
            prevPos.Add(t.gameObject.transform.localPosition);
            obs.Add(t.gameObject);
        }
    }
}

void CreateGrid()
{
    grid = new Node[gridSizeX, gridSizeY];
    Vector3 worldBottomLeft =
        transform.position - Vector3.right * gridWorldSize.x / 2 - Vector3.forward *
gridWorldSize.y / 2;

    for (int x = 0; x < gridSizeX; x++)
    {
        for (int y = 0; y < gridSizeY; y++)
        {
            Vector3 worldPoint = worldBottomLeft + Vector3.right *
(x * nodeDiameter + nodeRadius) + Vector3.forward * (y * nodeDiameter +
nodeRadius);
            bool walkable = !(Physics.CheckSphere(worldPoint, nodeRadius,
unwalkableMask));
            grid[x, y] = new Node(walkable, worldPoint ,x,y);
        }
    }

    public List<Node> GetNeighboursNodes(Node node)
    {
        List<Node> Neighbours = new List<Node>();
        for (int x = -1; x <= 1; x++) {
            for (int y = -1; y <= 1; y++) {
                if(x == 0 && y == 0 ) continue;

                int checkX = node.gridX + x;
                int checkY = node.gridY + y;
                if(checkX >= 0 && checkX < gridSizeX && checkY >= 0 && checkY <
gridSizeY)
                {
                    Neighbours.Add(grid[checkX,checkY]);
                }
            }
        }
        return Neighbours;
    }
}

```

```

    }
    public Node NodeFromWorldPoint(Vector3 worldPosition)
    {
        float percentX = (worldPosition.x + gridWorldSize.x / 2) / gridWorldSize.x;
        float percentY = (worldPosition.z + gridWorldSize.y / 2) / gridWorldSize.y;
        percentX = Mathf.Clamp01(percentX);
        percentY = Mathf.Clamp01(percentY);

        int x = Mathf.RoundToInt((gridSizeX - 1) * percentX);
        int y = Mathf.RoundToInt((gridSizeY - 1) * percentY);
        return grid[x, y];
    }

    public List<Node> path;
    void OnDrawGizmos()
    {
        Gizmos.DrawWireCube(transform.position, new Vector3(gridWorldSize.x, 1, gridWorldSize.y));

        if (grid != null)
        {
            foreach (Node n in grid)
            {
                Gizmos.color = (n.walkable) ? Color.white : Color.red;
                if(path != null)
                {
                    if(path.Contains(n))
                    {
                        Gizmos.color = Color.green;
                    }
                }
                Gizmos.DrawCube(n.worldNodePos, Vector3.one * (nodeDiameter - .1f));
            }
        }
    }
}

```

Applied Code in C#

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

public class AStarPathFinding : MonoBehaviour
{
    Grid;
    public Transform player;
    public Transform target;
    protected void Awake()
    {
        grid = GetComponent<Grid>();
    }
    void FindPath(Vector3 startPos, Vector3 targetPos)
    {
        List<Node> openedSet = new List<Node>();
        HashSet<Node> closedSet = new HashSet<Node>();

        Node startNode = grid.NodeFromWorldPoint(startPos);
        openedSet.Add(startNode);
        Node endNode = grid.NodeFromWorldPoint(targetPos);
        while (openedSet.Count > 0)
        {
            Node currentNode = openedSet[0];
            for (int i = 0; i < openedSet.Count; i++) {
                if(openedSet[i].fCost < currentNode.fCost ||
                    openedSet[i].fCost == currentNode.fCost &&
openedSet[i].hCost < currentNode.hCost)
                {
                    currentNode = openedSet[i];
                }
            }
            openedSet.Remove(currentNode);
            closedSet.Add(currentNode);
            if(currentNode == endNode)
            {
                RetractPath(startNode , endNode);
                return;
            }
            foreach (Node neighbour in
grid.GetNeighboursNodes(currentNode))
            {
                if(!neighbour.walkable ||
closedSet.Contains(neighbour)) continue;
                int movementCostToNeighbour = currentNode.gCost +
GetDistance(currentNode, neighbour);
                if(movementCostToNeighbour < neighbour.gCost ||
!openedSet.Contains(neighbour) )
                {
                    neighbour.gCost = movementCostToNeighbour;
                    neighbour.hCost = GetDistance(neighbour,
neighbour.parent = currentNode;
endNode);
```

```

        if(!openedSet.Contains(neighbour))
        {
            openedSet.Add(neighbour);
        }
    }
}

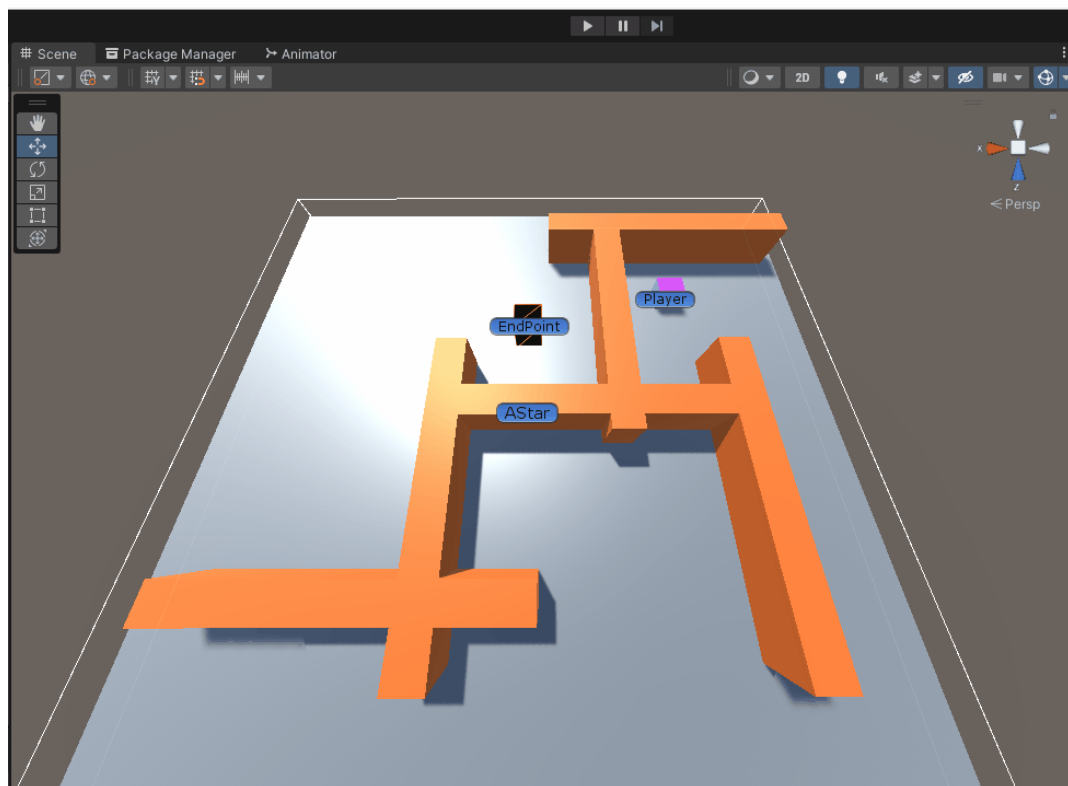
int GetDistance(Node a, Node b)
{
    int dstX = Mathf.Abs(a.gridX - b.gridX);
    int dstY = Mathf.Abs(a.gridY - b.gridY);
    if(dstX > dstY)
    {
        return 14 * dstY + 10*(dstX - dstY);
    }
    return 14* dstX + 10*(dstY - dstX);
}

void RetractPath(Node startNode , Node endNode)
{
    List<Node> path = new List<Node>();
    Node currentNode = endNode;

    while (currentNode != startNode)
    {
        path.Add(currentNode);
        currentNode= currentNode.parent;
    }
    path.Reverse();
    grid.path = path;
}

// Update is called once per frame
void Update()
{
    FindPath(player.position,target.position);
}
}

```



A Problem in my implementation

Resources

- 1- Sebastian Lague
- 2- GameDev.TV
- 3- <https://qiao.github.io/PathFinding.js/visual/>
- 4- www.ccpo.odu.edu
- 5- www.scaler.com
- 6- <https://www.geeksforgeeks.org>
- 7- <https://www.sciencedirect.com/science/article/pii/S187770581403149X>
- 8- <https://www.vtupulse.com/artificial-intelligence/a-star-search-algorithm-artificial-intelligence/>
- 9- <https://cs.stackexchange.com/questions/56176/a-graph-search-time-complexity>