Research Paper

Node Pruning by Jump Points Optimize A* by One Magnitude and More

Muhammad Hamza Noor[1]

[1]*AI Engineer at Frag Games*

This manuscript was compiled on September 7, 2024

Abstract

Pathfinding in grid environments is a problem commonly found in realistic Artificial Intelligence (AI) application areas such as robotics and video games. In this paper, I discuss A*- one of the most prominent and state-of-the-art pathfinding algorithm used presently in games and robotics, and its shortcomings especially when dealing with real-time pathfinding. Then I discuss a novel node pruning search strategy, specific to grids, which is fast, optimal and requires no memory overhead. This algorithm can be described as a macro operator that identifies and selectively expands only certain nodes in a grid map which are called jump points. Intermediate nodes on a path connecting two jump points are never expanded. I prove that this approach always computes optimal solutions and then undertake a thorough empirical analysis, comparing this method with standard A*. I find that searching with jump points can speed up A* by an order of magnitude and more and report significant improvement over the current state of the art.

Keywords: *Pathfinding Algorithms, Artificial Intelligence, A* Algorithm, A* Algorithm Optimization, Jump Point Search, JPS*

Corresponding author: Muhammad Hamza Noor, AI Engineer at Frag Games *E-mail address:* hamzarandhawa553@gmail.com

1. Introduction

One of the greatest challenges in the design of the realistic Artificial Intelligence (AI) in robotics and computer games is agent movement. Pathfinding strategies are usually employed as the core of any AI movement system. Pathfinding strategies have the responsibility of finding a path from any coordinate in the game world to another. Systems such as this take in a starting point and a destination; they then find a series of points that together comprise a path to the destination. A games' AI pathfinder usually employs some sort of precomputed data structure to guide the movement. At its simplest, this could be just a list of locations within the game that the agent is allowed to move to. Pathfinding inevitably leads to a drain on CPU resources especially if the algorithm wastes valuable time and CPU resources searching for a path that turns out not to exist.

Section 2 will highlight what game maps are and how useful information is extracted from these maps to use in pathfinding. Section 3 will show how pathfinding algorithms use this extracted information to return paths through the map when given a start and a goal position. As the A* pathfinding algorithm is such a major player in the computer games and robotics, it will be outlined in detail in Section 4. Section 5 will discuss the complexity of A* pathfinding techniques, particularly with their ability to find paths in real-time in big and complex grids.

In Section 6 and 7, I explain the node pruning technique Jump Point Search (JPS) proposed by Daniel Harabor and Alban Grastien in 2014 [HaraborGrastien14] [11], and **I prove by implementing Jump point Search for Game engines and put into the experiment in Unity Game Engine with A* and shows the results with empirical analysis that Jump Point Search speeds up the path finding of A* by one magnitude and more**. Then I conclude my research in Section 8.

2. Game World Geometry

Typically, the world geometry in a game is stored in a structure called a map. Maps usually contain all the polygons that make up the game environment. In a lot of cases, in order to cut down the search space of the game world for the pathfinder, the game map is broken down and simplified. The pathfinder then uses this simplified representation of the map to determine the best path from the starting point to the desired destination in the map. The most common forms of simplified representations are (1) Navigation Meshes, and (2) Waypoints.

2.1. Navigation Meshes

A navigation mesh is a set of convex polygons that describe the "walkable" surface of a 3D environment [Board&Ducker02] [3]. Algorithms have been developed to abstract the information required to generate Navigation Meshes for any given map. Navigation Meshes generated by such algorithms are composed of convex polygons which when assembled together represent the shape of the map analogous to a floor plan. The polygons in a mesh have to be convex since this guarantees that the AI agent can move in a single straight line from any point in one polygon to the center point of any adjacent polygon [WhiteChristensen02] [7]. Each of the convex polygons can then be used as nodes for a pathfinding algorithm. A navigation mesh path consists of a list of adjacent nodes to travel on. Convexity guarantees that with a valid path the AI agent can simply walk in a straight line from one node to the next on the list. Navigation Meshes are useful when dealing with static worlds, but they are unable to cope with dynamic worlds (or worlds that change).

2.2. Waypoints

The waypoint system for navigation is a collection of nodes (points of visibility) with links between them. Travelling from one waypoint to another is a sub problem with a simple solution. All places reachable from waypoints should be reachable from any waypoint by travelling along one or more other waypoints, thus creating a grid or path that the AI agent can walk on. If an AI agent wants to get from A to B it walks to the closest waypoint seen from position A, then uses a pre-calculated route to walk to the waypoint closest to position B and then tries to find its path from there. Usually, the designer manually places these waypoint nodes in a map to get the most efficient representation. This system has the benefit of representing the map with the least amount of nodes for the pathfinder to deal with.

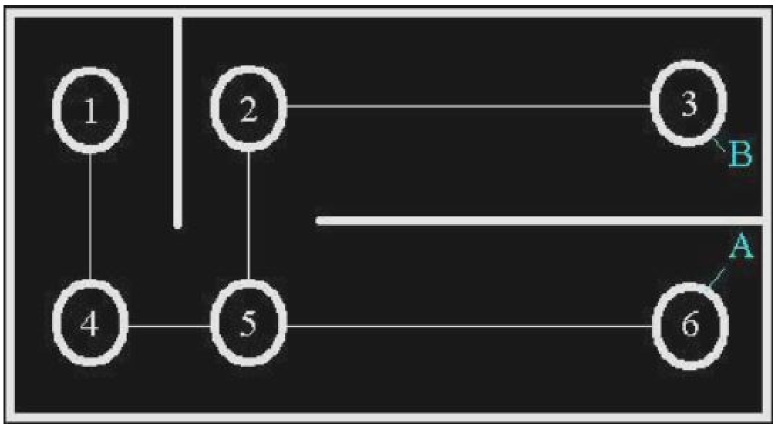


Figure 2.2.0.1. Waypoints

Figure 2.2.0.1 shows how a simple scene might be represented with waypoints. Table 2.2.0.1 shows the routing information contained within each waypoint. The path from (A) to (B) can be executed as follows. A straight-line path from (A) to the nearest waypoint is calculated (P1) then a straight-line path is calculated from (B) to the nearest waypoint (P2). These waypoints are 6 and 3 respectively. Then looking at the linking information, a pathfinding system will find the path as follows P1, waypoint 6, waypoint 5, waypoint 2, waypoint 3, P2

Way Point Number	Link Information
1	4
2	3, 5
3	2
4	1, 5
5	2, 4
6	5

Table 2.2.0.1. Routing Information

2.3. Graph Theory

Pathfinding algorithms can be used once the geometry of a game world has been encoded as a map and pre-processed to produce either a Navigation Mesh or a set of Waypoints. Since the polygons in the navigation mesh and the points in the waypoint system are all connected in some way, they are like points or nodes in a graph. So, all the pathfinding algorithm has to do is transverse the graph until it finds the endpoint it is looking for. Conceptually, a graph G is composed of two sets, and can be written as G = (V,E) where:

- V – Vertices: A set of discreet points in n-space, but this usually corresponds to a 3D map.
- E – Edges: A set of connections between the vertices, which can be either directed or not

Together with this structural definition, pathfinding algorithms also generally need to know about the properties of these elements. For example, the length, travel-time or general cost of every edge needs to be known. (From this point on, cost will refer to the distance between two nodes)

3. Pathfinding

In many game designs, AI is about moving agents/bots around in a virtual world. It is of no use to develop complex systems for high-level decision making if an agent cannot find its way around a set of obstacles to implement that decision. On the other hand, if an AI agent can understand how to move around the obstacles in the virtual world, even simple decision-making structures can look impressive. Thus the pathfinding system has the responsibility of understanding the possibilities for movement within the virtual world. A pathfinder will define a path through a virtual world to solve a given set of constraints. An example of a set of constraints might be to find the shortest path to take an agent from its current position to the target position. Pathfinding systems typically use the pre-processed representations of the virtual world as their search space.

3.1. Approaches to Pathfinding

There are many different approaches to pathfinding and for our purposes it is not necessary to detail each one. Pathfinding can be divided into two main categories, undirected and directed. The main features of each type will be outlined in the next section.

3.1.1. Undirected

This approach is analogous to a rat in a maze running around blindly trying to find a way out. The rat spends no time planning a way out and puts all its energy into moving around. Thus the rat might never find a way out and so uses most of the time going down dead ends. Thus, a design based completely on this concept would not be useful in creating a believable behaviour for an AI agent. It does however prove useful in getting an agent to move quickly while in the background a more sophisticated algorithm finds a better path.

There are two main undirected approaches that improve efficiency. These are Breadth-first search and Depth-first search respectively, they are well known search algorithms as detailed for example in [RusselNorvig95] [2]. Breadth-first search treats the virtual world as a large connected graph of nodes. It expands all nodes that are connected to the current node and then in turn expands all the nodes connected to these new nodes. Therefore if there is a path, the breadth-first approach will find it. In addition if there are several paths it will return the shallowest solution first. The depth-first approach is the opposite of breadth-first searching in that it looks at all the children of each node before it looks at the rest, thus creating a linear path to the goal. Only when the search hits a dead end does it go back and expand nodes at shallower levels. For problems that have many solutions the depth-first method is usually better as it has a good chance of finding a solution after exploring only a small portion of the search space. For clarity the two approaches will be explained using a simple map shown in Figure 3.1.1.1.

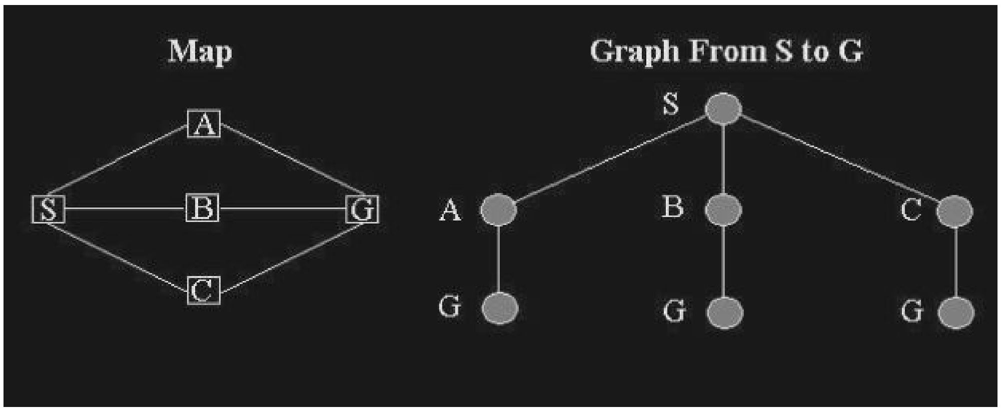


Figure 3.1.1.1. Shows a waypoint representation of a simple map and its corresponding complete search tree from the start (S) to the goal (G).

Figure 3.1.1.2 shows how the two approaches would search the tree to find a path. In this example the breadth-first took four iterations while the depth-first search finds a path in two. This is because the problem has many solutions, which the depth-first approach is best, suited to. The main drawback in these two approaches is that they do not consider the cost of the path but are effective if no cost variables are involved.

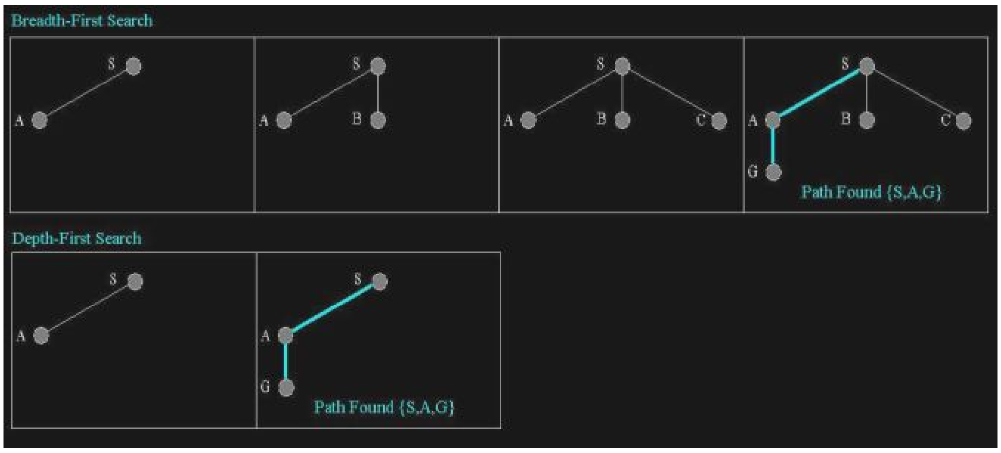


Figure 3.1.1.2

3.1.2. Directed

Directed approaches to pathfinding all have one thing in common, they do not go blindly through the maze. In other words they all have some method of assessing their progress from all the adjacent nodes before picking one of them. This is referred to as assessing the cost of getting to the adjacent node. Typically the cost in game maps is measured by the distance between the nodes. Most of the algorithms used will find a solution to the problem but not always the most efficient solution i.e. the shortest path. The main strategies for directed pathfinding algorithms are:

- Uniform cost search g(n) modifies the search to always choose the lowest cost next node. This minimises the cost of the path so far, it is optimal and complete, but can be very inefficient.
- Heuristic search h(n) estimates the cost from the next node to the goal. This cuts the search cost considerably but it is neither optimal nor complete.

The two most commonly employed algorithms for directed pathfinding in games use one or more of these strategies. These directed algorithms are known as Dijkstra and A* respectively [RusselNorvig95] [2]. Dijkstra's algorithm uses the uniform cost strategy to find the optimal path while the A* algorithm combines both strategies thereby minimizing the total path cost. Thus A* returns an optimal path and is generally much more efficient than Dijkstra making it the backbone behind almost all pathfinding designs in computer games. Since A* is the most commonly used algorithm in the pathfinding arena it will be outlined in more detail later in this report.

The following example in Figure 3.1.2.1 compares the effectiveness of Dijkstra with A*. This uses the same map from Figure 3.1.1.1 and its corresponding search tree from start (S) to the goal (G). However this time the diagram shows the cost of travelling along a particular path.

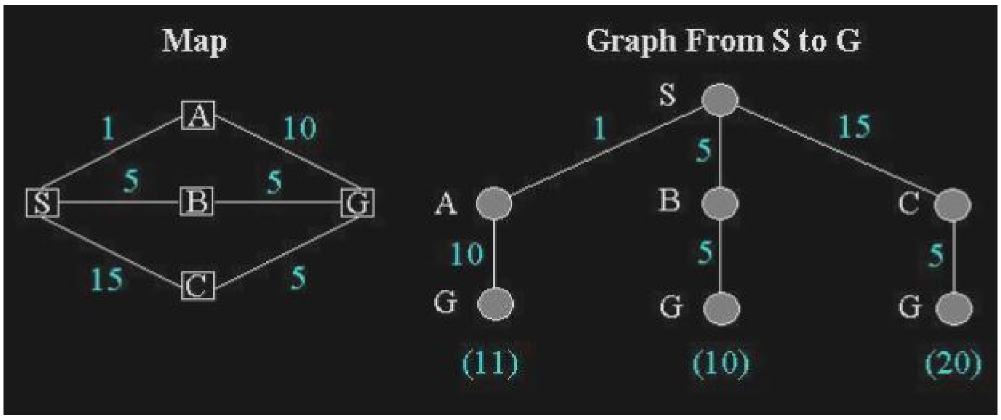


Figure 3.1.2.1

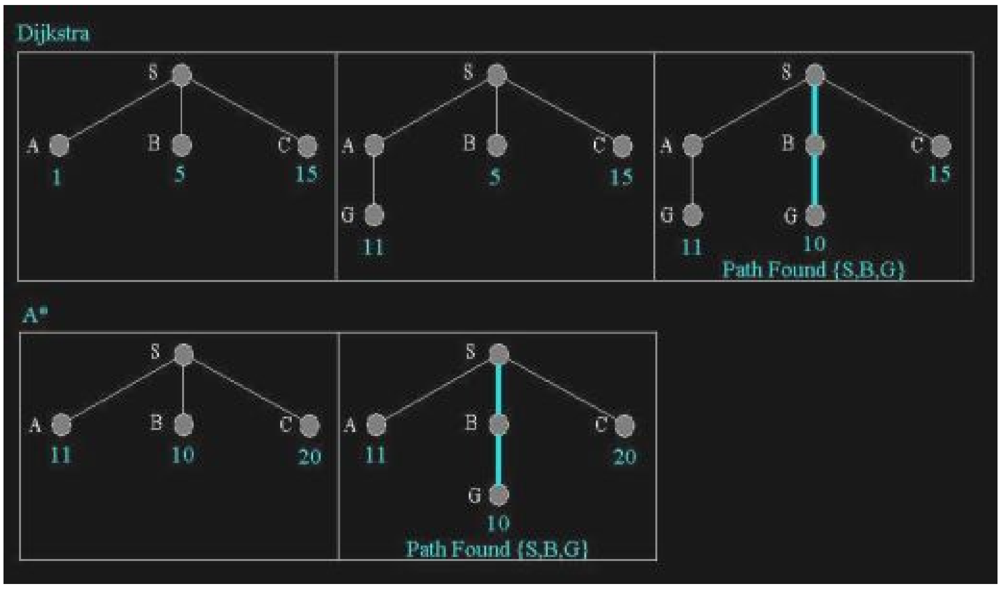


Figure 3.1.2.2

Figure 3.1.2.2 illustrates how Dijkstra and A* would search the tree to find a path given the costs indicated in Figure 3.1.2.1. In this example, Dijkstra took three iterations while A* search finds a path in two and finds the shortest path i.e. the optimal solution. Given that the first stage shown in Figure 3.1.2.2 for both Dijkstra and A* actually represents three iterations, as each node connected to the start node (S) would take one iteration to expand, the total iterations for Dijkstra and A* are six and five respectively. When compared to the Breadth-first and Depthfirst algorithms, which took five and two iterations respectively to find a path, Dijkstra and A* took more iterations but they both returned optimal paths while breadth-first and depth-first did not. In most cases, it is desirable to have agents that finds optimal pathways as following suboptimal pathways may be perceived as a lack of intelligence by a human player.

Many directed pathfinding designs use a feature known as Quick Paths. This is an undirected algorithm that gets the agent moving while in the background a more complicated directed pathfinder assesses the optimal path to the destination. Once the optimal path is found a "slice path" is computed which connects the quick path to the full optimal path. Thus creating the illusion that the agent computed the full path from the start. [Higgins02] [5].

4. A* Pathfinding Algorithm

A* (pronounced a-star) is a directed algorithm, meaning that it does not blindly search for a path (like a rat in a maze) [Matthews02] [6]. Instead it assesses the best direction to explore, sometimes backtracking to try alternatives. This means that A* will not only find a path between two points (if one exists!) but it will find the shortest path if one exists and do so relatively quickly.

4.1. How It Works

The game map has to be prepared or pre-processed before the A* algorithm can work. This involves breaking the map into different points or locations, which are called nodes. These can be waypoints, the polygons of a navigation mesh or the polygons of an area awareness system.

These nodes are used to record the progress of the search. In addition to holding the map location, . each node has three other attributes. These are fitness, goal and heuristic commonly known as f, g, and h respectively. Different values can be assigned to paths between the nodes. Typically these values would represent the distances between the nodes. The attributes g, h, and f are defined as follows:

- g is the cost of getting from the start node to the current node i.e. the sum of all the values in the path between the start and the current node.
- h stands for heuristic which is an estimated cost from the current node to the goal node (usually the straight line distance from this node to the goal).
- f is the sum of g and h and is the best estimate of the cost of the path going through the current node. In essence the lower the value of f the more efficient the path.

The purpose of f, g, and h is to quantify how promising a path is up to the present node. Additionally A* maintains two lists, an Open and a Closed list. The Open list contains all the nodes in the map that have not been fully explored yet, whereas the Closed list consists of all the nodes that have been fully explored. A node is considered fully explored when the algorithm has looked at every node linked to it. Nodes therefore simply mark the state and progress of the search.

4.2. The A* Algorithm

```
1. Let P = starting point.
2. Assign f, g and h values to P.
3. Add P to the Open list. At this point, P is the only node on the Open list.
4. Let B = the best node from the Open list (i.e. the node that has the lowest fvalue).
    (a) If B is the goal node, then quit – a path has been found.
    (b) If the Open list is empty, then quit – a path cannot be found.
5. Let C = a valid node connected to B.
    (a) Assign f, g, and h values to C.
    (b) Check whether C is on the Open or Closed list.
        i. If so, check whether the new path is more efficient (i.e. has a lower f-value).
            A. If so, update the path.
        ii. Else, add C to the Open list.
    (c) Repeat step 5 for all valid children of B.
6. Repeat from step 4.
```

A Simple example to illustrate the pseudo code of A*. The following step-through example should help to clarify how the A* algorithm works (see Figure 4.2.0.1).

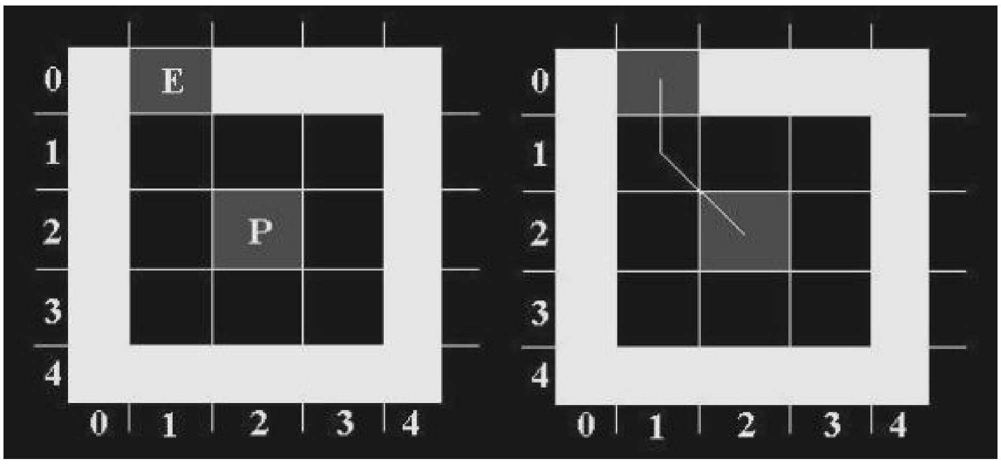


Figure 4.2.0.1

Let the center (2,2) node be the starting point (P), and the offset grey node (0,1) the end position (E). The h-value is calculated differently depending on the application. However for his example, h will be the combined cost of the vertical and horizontal distances from the present node to (E). Therefore h = | dx-cx | + | dy-cy | where (dx,dy) is the destination node and (cx,cy) is the current node. At the start, since P(2,2) is the only node that the algorithm knows, it places it in the Open list as shown in Table 4.2.0.1.

Open List	**Closed List**
P(2,2)	Empty

Table 4.2.0.1

Node	g-value	h-value	f-value
(1,1)	0 (Nodes to travel through)	1	1
(2,1)	0	2	2
(3,1)	0	3	3
(1,2)	0	2	2
(3,2)	0	4	4
(1,3)	0	3	3
(2,1)	0	4	4
(3,3)	0	5	5

Table 4.2.0.2. Routing Information

There are eight neighbouring nodes to p(2,2). These are (1,1), (2,1), (3,1), (1,2), (3,2), (1,3), (2,3), (3,3) respectively. If any of these nodes is not already in the Open list it is added to it. Then each node in the Open list is checked to see if it is the end node E(1,0) and if not, then its f-value is calculated (f = g + h).

As can be seen from Table 4.2.0.2, Node (1,1) has the lowest f-value and is therefore the next node to be selected by the A* algorithm. Since all the neighbouring nodes to P(2,2) have been looked at, P(2,2) is added to the Closed list (as shown in Table 4.2.0.3)

Open List	Closed List
(1,1), (2,1), (3,1), (1,2), (3,2), (1,3), (2,3), (3,3)	P(2,2)

Table 4.2.0.3

There are four neighbouring nodes to (1,1) which are E(1,0), (2,1), (1,2), (2,2) respectively. Since E(1,0) is the only node, which is not on either of the lists, it is now looked at. Given that all the neighbours of (1,1) have been looked at, it is added to the Closed list. Since E(1,0) is the end node, a path has therefore been found and it is added to the Closed list. This path is found by back-tracking through the nodes in the Closed list from the goal node to the start node P (2,2), (1,1), E(1,0) . This algorithm will always find the shortest path if one exists [Matthews02] [6].

5. Limitations Of A*

A* requires a large amount of CPU resources if there are many nodes to search through as is the case in large maps which have become popular in the newer games. In sequential programs, this may cause a slight delay in the game. This delay is compounded if A* is searching for paths for multiple AI agents and/or when the agent has to move from one side of the map to the other. This drain on CPU resources may cause the game to freeze until the optimal path is found. Game designers overcome these problems by tweaking the game so as to avoid these situations [Cain02][4].

The inclusion of dynamic objects to the map is also a major problem when using A*. For example, once a path has been calculated, if a dynamic object then blocks the path or target movies, the agent would have no knowledge of this and would continue on as normal and walk straight into the object or to the location from where the target has moved. Simply reapplying the A* algorithm every time a node is blocked would cause an excessive drain on the CPU. Research has been conducted to extend the A* algorithm to deal with this problem most notably the D* algorithm (which is short for dynamic A*) [Stentz94][1]. This allows for the fact that node costs may change as the AI agent moves across the map and presents an approach for modifying the cost estimates in real time. However the drawback to this approach is that it adds further to the drain on the CPU and forces a limit on the dynamic objects that can be introduced to performance-sensitive applications such as video games.

These problems in performance-sensitive applications such as video games cause performance issues and make the game lag if CPU resources are limited. A solution to these problems in A* is to improve A* by a novel search strategy, specific to grids, which is fast, optimal and requires no memory overhead. This algorithm can be described as a macro operator that identifies and selectively expands only certain nodes in a grid map which we call jump points. Intermediate nodes on a path connecting two jump points are never expanded. We prove that this approach always computes optimal solutions and then undertake a thorough empirical analysis. We find that searching with jump points can speed up A* by an order of magnitude and more and report significant improvement over the current state of the art. [HaraborGrastien14] [11]

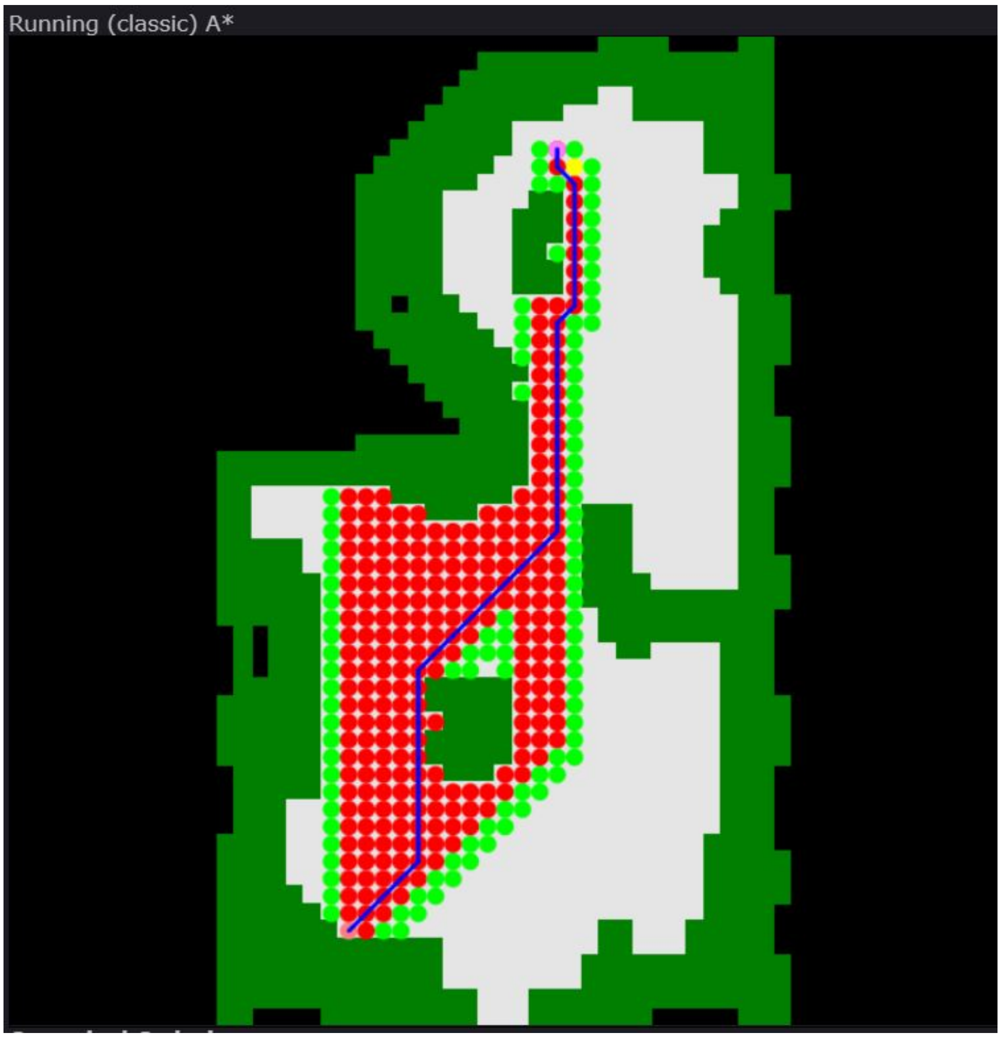


Figure 5.0.0.1

Figure 5.0.0.1 shows Nodes exploration in A* from the start point to the target above. Red circles are the nodes expanded from heap while finding path. To find path, A* explored a great portion of the grid nodes by putting them into Heap and taking them out on the basis of least f-cost i.e. the nodes with least f-cost get explored first. Inserting and pulling a node out of heap costs LogN each time, so doing it for so many nodes of the map for each AI agent puts so much performance overload on the CPU by occupying resources and Random Access Memory(RAM).

Time and Space Complexity of A*:

One major practical drawback of A* is its Time and Space Complexity which is $\mathbf{O}(\mathbf{b}^{\mathbf{d}})$ where:
1. **b** is the branching factor (the average number of successors per state), as it stores all generated nodes in memory.
2. **d** is the depth of the solution (the length of the shortest path).

6. Jump Point Search

Widely employed in areas such as robotics [Lee&Yu09] [9], artificial intelligence [Wang&Botea09] [8] and video games, the ubiquitous undirected uniform-cost grid map is a highly popular method for representing pathfinding environments. Regular in nature, this domain typically features a high degree of path symmetry [Harabor&Botea][10]. Symmetry in this case manifests itself as paths (or path segments) which share the same start and end point, have the same length and are otherwise identical save for the order in which moves occur. Unless handled properly, symmetry can force search algorithms to evaluate many equivalent states and prevent real progress toward the goal. In this paper, we deal with such path symmetries by developing a macro operator that selectively expands only certain nodes from the grid, which we call jump points. Moving from one jump point to the next involves travelling in a fixed direction while repeatedly applying a set of simple neighbour pruning rules until either a dead-end or a jump point is reached. Because we do not expand any intermediate nodes between jump points, this strategy can have a dramatic positive effect on search performance. Furthermore, computed solutions are guaranteed to be optimal. Jump point pruning is fast, requires no preprocessing and introduces no memory overheads. It is also largely orthogonal to many existing speedup techniques applicable to grid maps. [HaraborGrastien14] [11]

6.1. Jump Points

In this section, we explore a search strategy for speeding up optimal search by selectively expanding only certain nodes on a grid map which are termed jump points. We give an example of the basic idea in Figure 6.1.0.1.

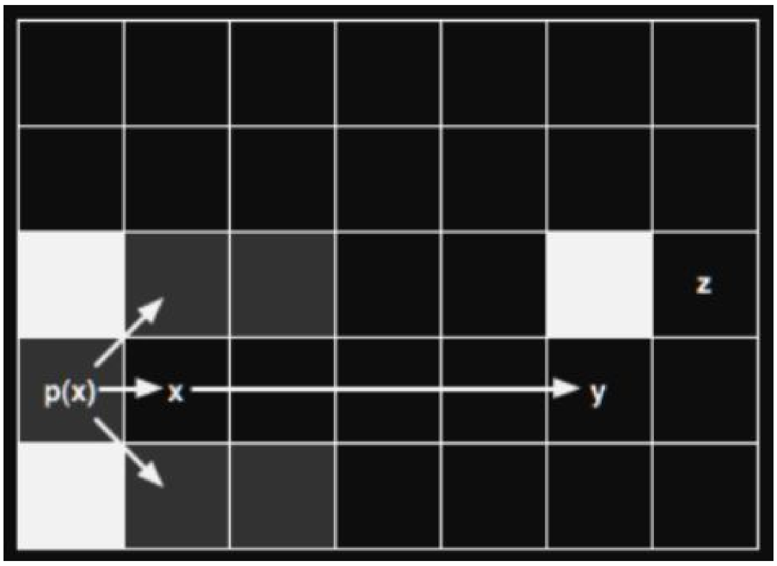


Figure 6.1.0.1

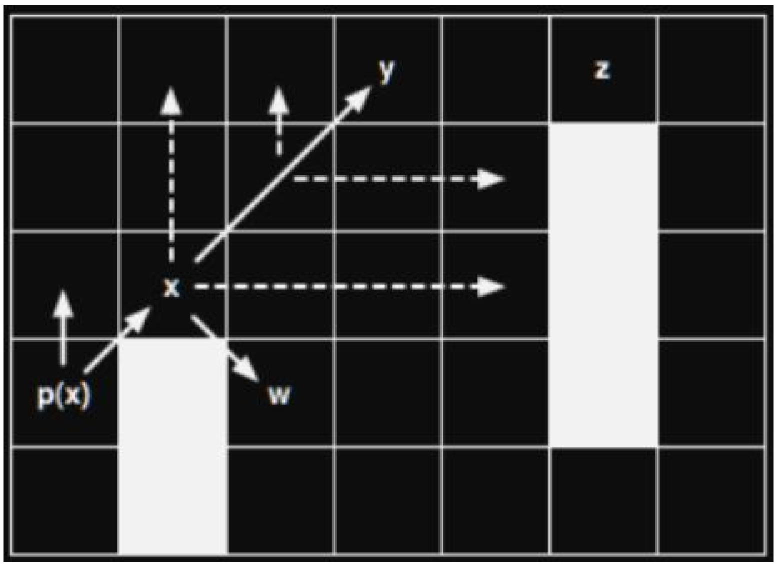


Figure 6.1.0.2

Examples of straight 6.1.0.1 and diagonal 6.1.0.2 jump points. Dashed lines indicate a sequence of interim node evaluations that reached a dead end. Strong lines indicate eventual successor nodes.

Here the search is expanding a node x which has as its parent p(x); the direction of travel from p(x) to x is a straight move to the right. When expanding x we may notice that there is little point to evaluating any neighbour highlighted grey as the path induced by such a move is always dominated by (i.e. no better than) an alternative path which mentions p(x) but not x. I will make this idea more precise in the next section but for now it is sufficient to observe that the only non-dominated neighbour of x lies immediately to the right. Rather than generating this neighbour and adding it to the open list, as in the classical A* algorithm, It is proposed to simply step to the right and continue moving in this direction until we encounter a node such as y; which has at least one other non-dominated neighbour (here z). If we find a node such as y (a jump point) we generate it as a successor of x and assign it a g-value (or cost-so-far) of g(y) = g(x) + dist(x, y). Alternatively, if we reach an obstacle we conclude that further search in this direction is fruitless and generate nothing. In the remainder of this section we will develop a macrostep operator which speeds up node expansion by identifying jump point successors in the case of both straight and diagonal moves. First it will be necessary to define a series of pruning rules to determine whether a node should be generated or skipped. This will allow us to make precise the notion of a jump point and give a detailed description of the jump points algorithm. Then, we prove that the process of "jumping" nodes, such as x in Figure 6.1.0.1, has no effect on the optimality of the search. [HaraborGrastien14][11]

6.2. Algorithmic Description

6.2.1. Horizontal and Vertical Scan

Horizontal (and vertical) scanning is the simplest to explain. The discussion below only covers horizontal scanning from left to right, but the other three directions are easy to derive by changing the scanning direction, and/or substituting left/right for up/down.

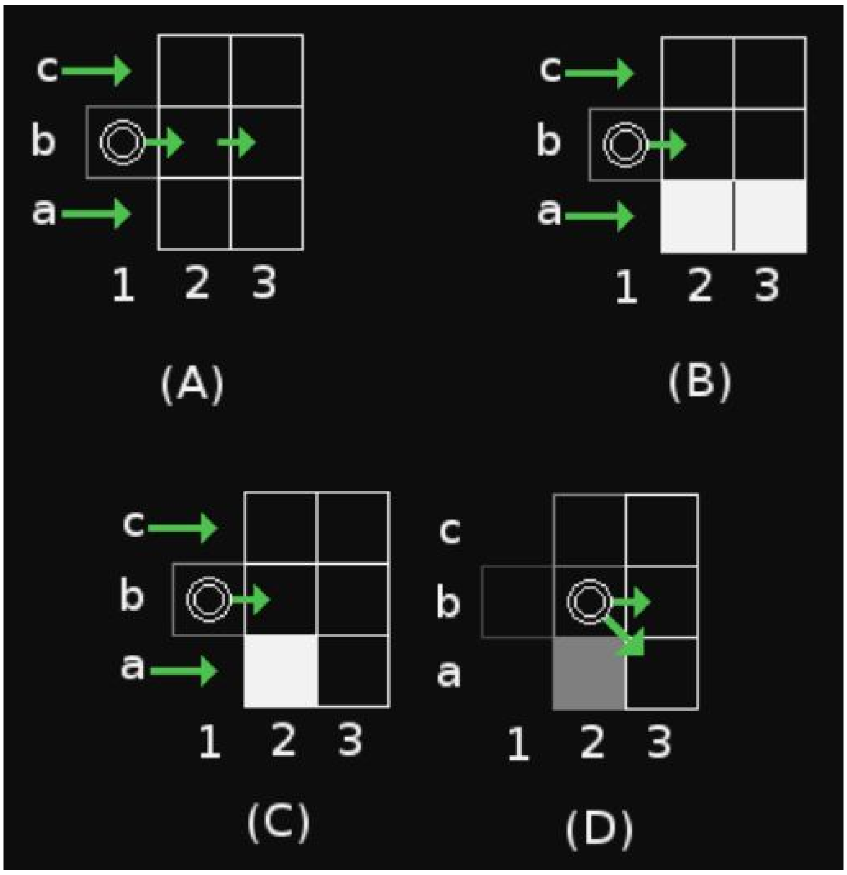


Figure 6.2.1.1. The (A) picture shows the global idea. The algorithm scans a single row from left to right. Each horizontal scan handles a different row. In the section about diagonal scan below, it will be explained how all rows are searched

At this time, assume the goal is to only scan the b row, rows a and c are done at some other time. The scan starts from a position that has already been done, in this case b1. Such a position is called a parent. The scan goes to the right, as indicated by the green arrow leaving from the b1 position. The (position, direction) pair is also the element stored in open and closed lists. It is possible to have several pairs at the same position but with a different direction in a list.

The goal of each step in the scan is to decide whether the next point (b2 in the picture) is interesting enough to create a new entry in the open list. If it is not, you continue scanning (from b2 to b3, and further). If a position is interesting enough, new entries (new jump points) are made in the list, and the current scan ends.

Positions above and below the parent (a1 and c1) are covered already due to having a parent at the b1 position, these can be ignored. In Figure 6.2.1.1 [A] picture, position b2 is in open space, the a and c rows are handled by other scans, nothing to see here, we can move on [to b3 and further]. The picture is the same.

The a row is non-passable, the scan at the a row has stopped before, but that is not relevant while scanning the b row.

The 6.2.1.1 [C] picture shows an 'interesting' situation. The scan at the a row has stopped already due to the presence of the non-passable cell at a2 [or earlier]. If we just continue moving to the right without doing anything, position a3 would not be searched. Therefore, the right action here is to stop at position b2, and add two new pairs to the open list, namely (b2, right) and (b2, right-down) as shown in picture [D] of figure 6.2.1.1. The former makes sure the horizontal scan is continued if useful, the latter starts a search at the a3 position (diagonally down).

After adding both new points, this scan is over and a new point and direction is selected from the open list. The row below is not the only row to check. The row above is treated similarly, except 'down' becomes 'up'.

Two new points are created when c2 is non-passable and c3 is passable. (This may happen at the same time as a2 being non-passable and a3 being passable. In that case, three jump points will be created at b2, for directions right-up, right, and right-down.) Last but not least, the horizontal scan is terminated when the scan runs into a non-passable cell, or reaches the end of the map. In both cases, nothing special is done, besides terminating the horizontal scan at the row.

6.2.2. Code of the Horizontal Scan

```csharp
List<JumpPointNode> HorizontalSearch(Vector2 pos, Node parent, int horDir, int dist, Node destination)
{
    int x0 = (int)pos.x, y0 = (int)pos.y;

    List<JumpPointNode> nodes = new List<JumpPointNode>();
    for (int xi = x0; xi < gridComponent.GetGridSizeX() && xi >= 0; xi += horDir)
    {
        int x1 = xi + horDir;
        if (!IsValid(x1, y0))
            return nodes;
        Node g = grid[x1, y0];
        if (!g.walkable) return nodes;
        if (g == destination)
        {
            g.parent = parent;
            nodes.Add(new JumpPointNode(g, horDir, 0 , dist, parent));
            return nodes;
        }
        dist += horVertCost;
        int x2 = x1 + horDir;

        if (IsValid(x1, y0-1) && !grid[x1, y0 - 1].walkable && IsValid(x2, y0-1) && grid[x2, y0 - 1].walkable)
        {
            if (closedSet.Contains(grid[x1, y0]))
            {
                if (dist < grid[x1, y0].gCost)
                {
                    nodes.Add(new JumpPointNode(grid[x1, y0], horDir, -1, dist, parent));
                    grid[x2, y0 - 1].gCost = dist + diagonalCost;

                }
                else
                    return nodes;
            }
            else if (dist < grid[x1, y0].gCost)
            {
                nodes.Add(new JumpPointNode(grid[x1, y0], horDir, -1, dist, parent));
                grid[x2, y0 - 1].gCost = dist + diagonalCost;
            }
        }
        if (IsValid(x1, y0+1) && !grid[x1, y0 + 1].walkable && IsValid(x2, y0+1) && grid[x2, y0 + 1].walkable)
        {
            if (closedSet.Contains(grid[x1, y0]))
            {
                if (dist < grid[x1, y0].gCost)
                {
                    nodes.Add(new JumpPointNode(grid[x1, y0], horDir, +1, dist, parent));
                    grid[x2, y0 + 1].gCost = dist + diagonalCost;
                }
                else
                    return nodes;
            }
            else if (dist < grid[x1, y0].gCost)
            {
                nodes.Add(new JumpPointNode(grid[x1, y0], horDir, +1, dist, parent));
                grid[x2, y0 + 1].gCost = dist + diagonalCost;
            }
        }
        if (nodes.Count > 0)
        {
            nodes.Add(new JumpPointNode(grid[x1, y0], horDir, 0, dist, parent));
            return nodes;
        }
    }
    return nodes;
}
```

Code 1. C# Horizontal Scan

Coordinate (x0, y0) is at the parent position, x1 is next to the parent, and x2 is two tiles from the parent in the scan direction. The code is quite straightforward. After checking for the off-map and obstacle cases at x1, the non-passable and passable checks are done, first above the y0 row, then below it. If either case adds a node to the nodes result, the continuing horizontal scan is also added, all nodes are returned. Code of the vertical scan works similarly.

6.2.3. Diagonal Scan

The diagonal scan uses the horizontal and vertical scan as building blocks, otherwise, the basic idea is the same. Scan the area in the given direction from an already covered starting point, until the entire area is done or until new jump points are found.

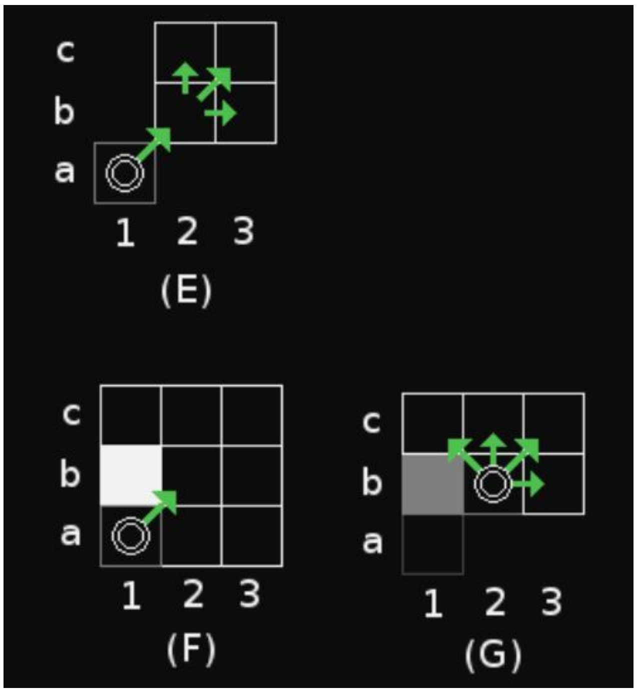


Figure 6.2.3.1

The scan direction explained here in 6.2.3.1 is diagonally to the right and up. Other scan directions are easily derived by changing 'right' with 'left', and/or 'up' with 'down'. Picture [E] shows the general idea.

Starting from position a1, the goal is to decide if position b2 is a jump point. There are two ways how that can happen. The first way is if a2 (or b1) itself is an 'interesting' position. The second way is if up or to the right new jump points are found.

The first way is shown in picture [F]. When position b1 is non-passable, and c1 is passable, a new diagonal search from position b2 up and to the left must be started. In addition, all scans that would be otherwise performed in the diagonal scan from position a1 must be added. This leads to four new jump points, as shown in picture [G].

Note that due to symmetry, similar reasoning causes new jump points for searching to the right and down, if a2 is non-passable and a3 is passable. (As with the horizontal scan, both c1 and a3 can be new directions to search at the same time as well.) The second way of getting a jump point at position b2 is if there are interesting points further up or to the right.

To find these, a horizontal scan to the right is performed starting from b2, followed by a vertical scan up from the same position. If both scans do not result in new jump points, position b2 is considered done, and the diagonal scan moves to examine the next cell at c3 and so on, until a non-passable cell or the end of the map.

6.2.4. Code of the Diagonal Scan

```
List<JumpPointNode> DiagonalSearch(Vector2 pos, Node parent,int horDir, int verDir, int dist, Node destination)
{
    int x0 = (int)pos.x, y0 = (int)pos.y;
    List<JumpPointNode> nodes = new List<JumpPointNode>();
    for (int xi = x0, yi = y0;
         xi >= 0 && xi < gridComponent.GetGridSizeX() && yi >= 0 && yi < gridComponent.GetGridSizeY(); xi+= horDir,
      yi += verDir)
    {
        int x1 = xi + horDir;
        int y1 = yi + verDir;
        if (!IsValid(x1, y1))
            return nodes;
        Node g = grid[x1, y1];
        if (!g.walkable) return nodes;
        if (g == destination)
        {
            g.parent = parent;
            nodes.Add(new JumpPointNode(g, horDir, verDir , dist, parent));
            return nodes;
        }
        dist += diagonalCost;
        int x2 = x1 + horDir, y2 = y1 + verDir;
        if (IsValid(x0, y1) && !grid[x0, y1].walkable && IsValid(x0, y2) && grid[x0, y2] != null && grid[x0, y2].
     walkable)
        {
            if (closedSet.Contains(grid[x1, y1]))
            {
                if (dist < grid[x1, y1].gCost)
                {
                    nodes.Add(new JumpPointNode(grid[x1, y1], -horDir, verDir, dist, parent));
                    grid[x0, y2].gCost = dist + diagonalCost;
                }
                else
```

```
                    return nodes;
            }
            else if (dist < grid[x1, y1].gCost)
            {
                nodes.Add(new JumpPointNode(grid[x1, y1], -horDir, verDir, dist, parent));
                grid[x0, y2].gCost = dist + diagonalCost;
            }
        }

        if (IsValid(x1, y0) && !grid[x1, y0].walkable &&IsValid(x2, y0) && grid[x2, y0] != null && grid[x2, y0].
     walkable)
        {
            if (closedSet.Contains(grid[x1, y1]))
            {
                if (dist < grid[x1, y1].gCost)
                {
                    nodes.Add(new JumpPointNode(grid[x1, y1], horDir, -verDir, dist, parent));
                    grid[x2, y0].gCost = dist + diagonalCost;
                }
                else
                    return nodes;
            }
            else if (dist < grid[x1, y1].gCost)
            {
                nodes.Add(new JumpPointNode(grid[x1, y1], horDir, -verDir, dist, parent));
                grid[x2, y0].gCost = dist + diagonalCost;
            }
        }

        if (nodes.Count == 0)
        {
            g.parent = parent;
            List<JumpPointNode> subNodes = HorizontalSearch(new Vector2(x1, y1), g, horDir, dist, destination);
            if (subNodes.Count > 0)
            {
                nodes.AddRange(subNodes);
            }
                subNodes = VerticalSearch(new Vector2(x1, y1), g, verDir, dist, destination);
            if (subNodes.Count > 0)
            {
                nodes.AddRange(subNodes);
            }
        }
        else if (nodes.Count > 0)
        {
            nodes.Add(new JumpPointNode(grid[x1, y1], 0, verDir, dist, parent));
            nodes.Add(new JumpPointNode(grid[x1, y1], horDir, verDir, dist, parent));
            nodes.Add(new JumpPointNode(grid[x1, y1], horDir, 0, dist, parent));

            return nodes;
        }
    }
    return nodes;
}
```

Code 2. C# Diagonal Scan

The same coordinate system as with the horizontal scan 1 is used here as well. (x0, y0) is the parent position, (x1, y1) is one diagonal step further, and (x2, y2) is two diagonal steps. After map boundaries, obstacle, and destination-reached checking, first checks are done if (x1, y1) itself should be a jump point due to obstacles. Then it performs a horizontal scan, followed by a vertical scan. Most of the code is detection that a new point was created, skipping the remaining actions, and then creating new jump points for the skipped actions. Also, if jump points got added in the horizontal or vertical search, their parent reference is set to the intermediate point. This is discussed further in the next section.

6.2.5. Creating Jump Points

Creating jump points at an intermediate position, such as at b2 when the horizontal or vertical scan results in new points has a second use. It's a record of how you get back to the starting point. Consider the following situation:

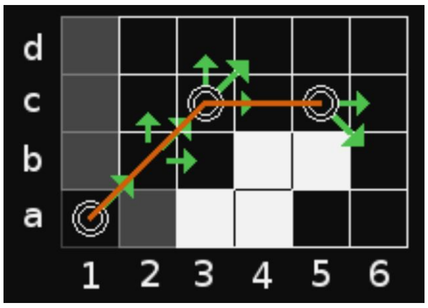


Figure 6.2.5.1

Here in 6.2.5.1, a diagonal scan started at a1. At b2 nothing was found. At c3, the horizontal scan resulted in new jump points at position c5. By adding a jump point at position c3 as well, it is easy to store the path back from position c5, as you can see with the yellow line. The simplest way is to store a pointer to the previous (parent) jump point. In the code, I use special jump points for this, which are only stored in the closed list (if no suitable node could be found instead) by means of the get_closed_node method.

6.2.6. Starting Point

Finally, a small note about the starting point. In all the discussions before, the parent was at a position which was already done. In addition, scan directions make assumptions about other scans covering the other parts. To handle all these requirements, you first need to check the starting point is not the destination. Secondly, make eight new jump points, all starting from the starting position but in a different direction. Finally, pick the first point from the open list to initiate the search.

7. Performance

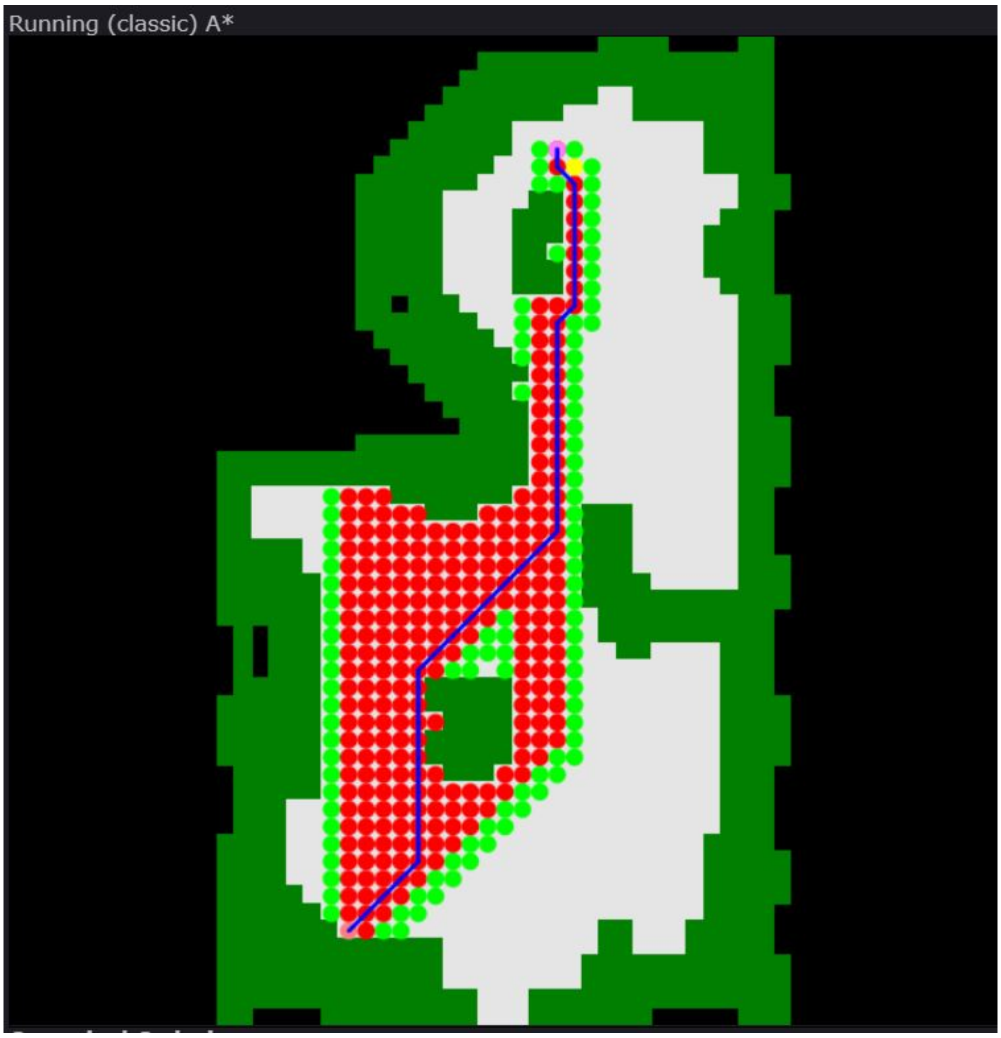


Figure 7.0.0.1. Path Found by A* Algorithm

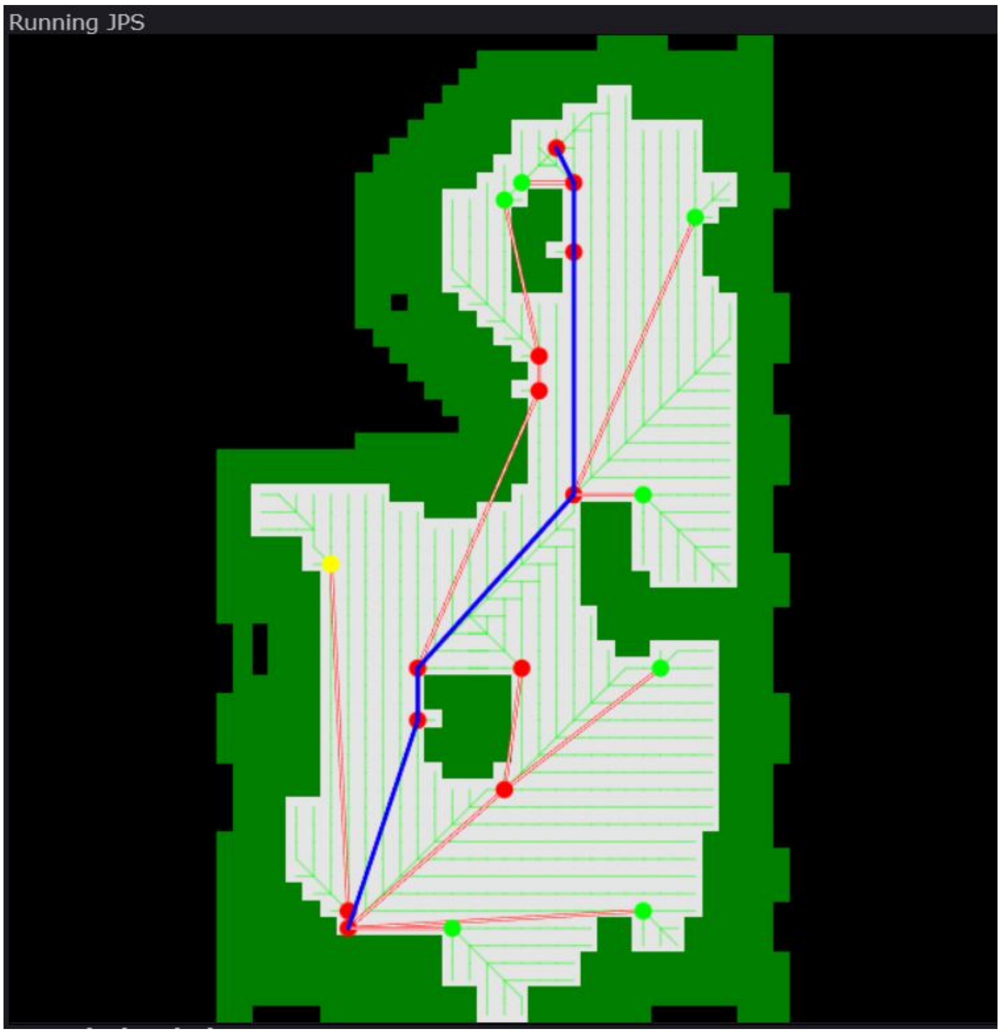


Figure 7.0.0.2. Path Found by Jump Point Search (JPS) Algorithm

Figure 7.0.0.1 and Figure 7.0.0.2 shows the same path found by A* and JPS respectively by exploring nodes by putting them into heap. Red nodes are those which have been taken out of heap and put into closed set. And Green Circles are the nodes which were not taken out of heap and explored further.

It can be seen that JPS also found the shortest and optimal path but JPS put so many less nodes into the heap and consumed very less time and memory.

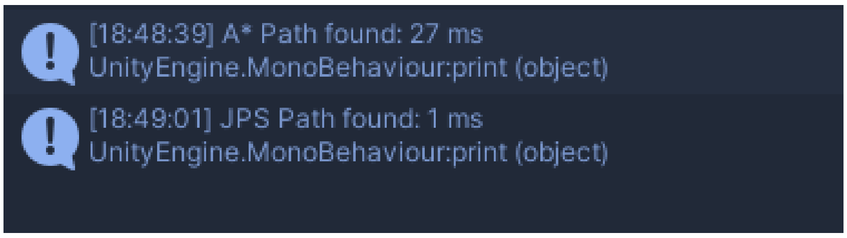
After implementing JPS, I tested standard A* and JPS both on different grid maps of sizes and formations in Unity, giving both similar starting and target positions. My finding is that JPS takes less Time and Space by one magnitude and more.

7.1. Results

This was the result of the time taken by both algorithms to find the path on the following Grid sizes:

7.1.1. On the Grid Map of nodes 100 x 100

A* found path in 27 milliseconds. JPS found the same path in less than 1 millisecond. JPS found the path more than 27 times faster than standard A*.

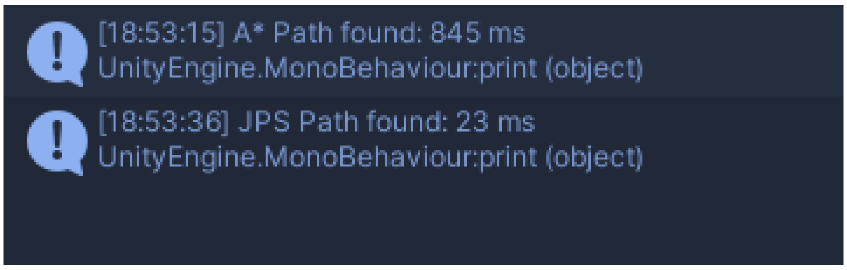


```
[18:48:39] A* Path found: 27 ms
UnityEngine.MonoBehaviour:print (object)
[18:49:01] JPS Path found: 1 ms
UnityEngine.MonoBehaviour:print (object)
```

Figure 7.1.1.1. Comparison of A* and JPS Performance

7.1.2. On the Grid Map of nodes 1000 x 1000

A* found path in 845 milliseconds. JPS found the same path in 23 milliseconds. JPS found the path 36.7 times faster than standard A*

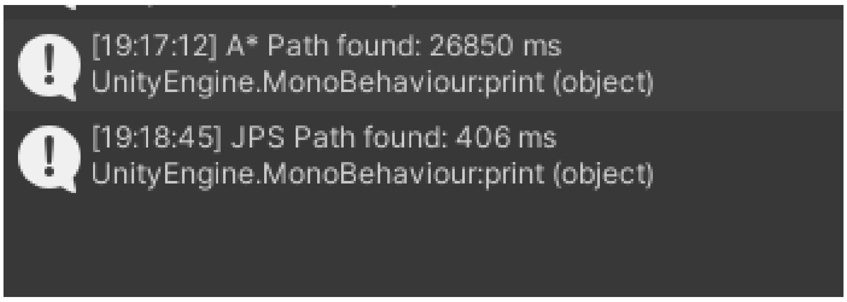


```
[18:53:15] A* Path found: 845 ms
UnityEngine.MonoBehaviour:print (object)
[18:53:36] JPS Path found: 23 ms
UnityEngine.MonoBehaviour:print (object)
```

Figure 7.1.2.1. Comparison of A* and JPS Performance

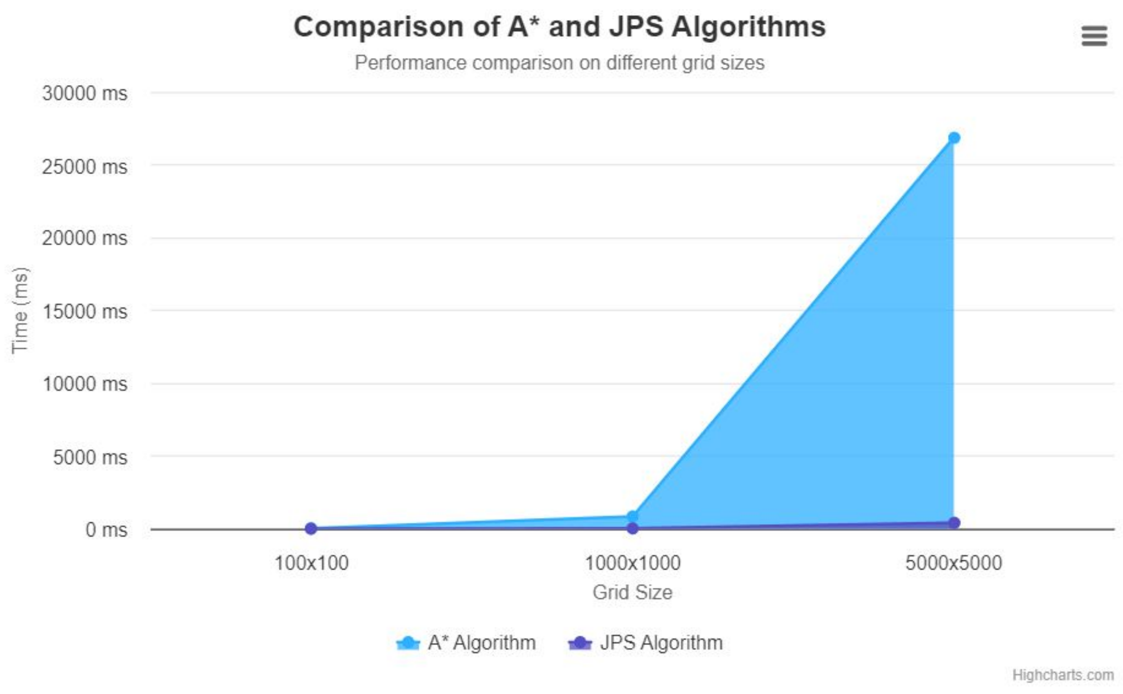
7.1.3. On the Grid Map of nodes 5000 x 5000 299

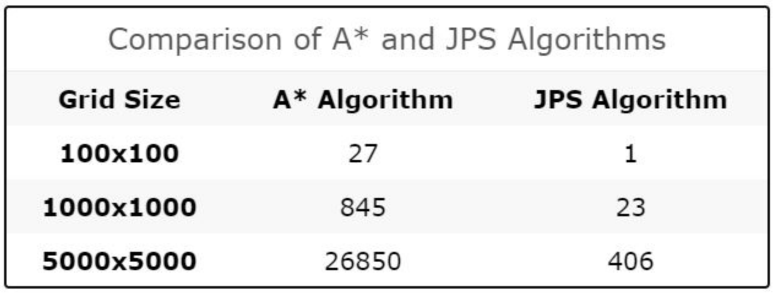
A* found path in 26,850 milliseconds. JPS found the same path in 406 milliseconds. JPS found the path 66.1 times faster than standard A*. 300



```
[19:17:12] A* Path found: 26850 ms
UnityEngine.MonoBehaviour:print (object)
[19:18:45] JPS Path found: 406 ms
UnityEngine.MonoBehaviour:print (object)
```

Figure 7.1.3.1. Comparison of A* and JPS Performance





Comparison of A* and JPS Algorithms

Grid Size	A* Algorithm	JPS Algorithm
100x100	27	1
1000x1000	845	23
5000x5000	26850	406

Figure 7.1.3.2. Visual Representation of Time Taken by A* and JPS

8. Conclusion

In pursuit of finding a solution to the problem of too much time and resources taking in A* path-finding, I found this node pruning strategy for speeding up pathfinding on undirected uniform-cost grid maps an effective solution. This algorithm identifies and selectively expands only certain nodes from a grid map which are called jump points. Moving between jump points involves only travelling in a fixed direction, either straight or diagonal. I prove that intermediate nodes on a path between two jump points never need to be expanded and "jumping" over them does not affect the optimality of search.

This method is unique in the pathfinding literature in that it has very few disadvantages: it is simple, yet highly effective; it preserves optimality, yet requires no extra memory; it is extremely fast, yet requires no preprocessing. I am unaware of any other algorithm which has all these features.

This solution will work and give optimal paths only in uniform costs grid maps. Further research is needed to answer how the Jump Point Search technique can be used in a grid map with variable costs as the current technique of Jump Point Search does not give any solution to find paths with variable costs of nodes in the map.

References

[1] A. Stentz, *Optimal and Efficient Path Planning forPartially-known Environments*, In proceedings of the IEEE International Conference on Robotics and Automation, May 1994.

[2] S. Russel and P. Norvig, *ArtificialIntelligence A Modern Approach*, Published by Prentice-Hall.Inc, Dec. 1995.

[3] B. Board and M. Ducker, *Area Navigation: Expanding the Path-Finding Paradigm*, Published by GameProgramming Gems 3 and Charles River Media., Dec. 2002.

[4] T. Cain, *Practical Optimizations for A**, Published by AI GameProgramming Wisdom and Charles River Media, Dec. 2002.

[5] D. Higgins, *Generic A* Pathfinding*, Published by AI GameProgramming Wisdom and Charles River Media, Dec. 2002.

[6] J. Matthews, *Basic A* Pathfinding Made Simple*, Published by AI GameProgramming Wisdom and Charles River Media., Dec. 2002.

[7] S. White and C. Christensen, *A Fast Approach to Navigation Meshes*, Published by GameProgramming Gems 3 and Charles River Media., Dec. 2002.

[8] A. B. K. Wang, *Tractable multi-agent path planning on grid maps.* In IJCAI, 1870–1875, May 2009.

[9] J. Lee and W. Yu, *A coarse-to-fine approach for fast path finding for mobile robots*, In IROS, 5414 –5419. Pochter, N.; Zohar, A.; Rosenschein, J. S.; and Felner, A. 2010. Search space reduction using swamp hierarchies. In AAAI, May 2009.

[10] A. B. K. Wang, *Breaking path symmetries in 4-connected grid maps*, In AIIDE, 33–38, May 2010.

[11] D. Harabor and A. Grastien, *Online Graph Pruning for Pathfinding on Grid Maps*, Published by NICTA and The Australian National University., Dec. 2014.