AI Pathfinding in Video Games:

JPS+ with Recursive Goal Bounding

by Joshua Jay Espinosa

May 2016

Submitted to the *Department of Computer Science*

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Iona College, N.Y.

Thesis Advisor: Smiljana Petrovic, Ph.D

_____	_____
Author	Date
_____	_____
Advisor	Date
_____	_____
Department Chair	Date

Contact: jespinosa2@gaels.iona.edu
Date: May 2016

AI Pathfinding in Video Games:

JPS+ with Recursive Goal Bounding

by Joshua Jay Espinosa

Abstract

Artificial Intelligence is used in video games to help in-game objects find a goal. Specifically, 'Pathfinding' is the automated technique of choosing the shortest path between two points, a starting node and a goal node. The A* Search was primarily used for pathfinding until recent developments with JPS+ and Goal Bounds algorithms that improved the speed up to anywhere from 20 to 1000 times that of A*. The JPS+ eliminates redundant paths by identifying and preprocessing 'jump points'. Goal bounds establishes where the goal is and avoids exploring points in the wrong direction.

This thesis strives to prove or disprove that the goal bounding algorithm can be used recursively during precomputation in video games to eliminate unnecessary jump points that are not visited during the search for the goal. Considering fewer jump points saves a substantial amount of pathfinding time during runtime. Considered points can be called 'subgoals' as they are bounds within bounds. If the size of the goal bounds is further reduced, even more jump points could be eliminated, which results in less time taken to find the shortest path.

Acknowledgements

I'd like to extend my deepest and most profound gratitude to my advisor, Dr. Smiljana Petrovic. Without her, I would never have accomplished all I have. I am indebted to her wisdom, compassion, patience and expertise. Dr. Petrovic's presence and intelligence are an inspiration and I will follow that inspiration to make her proud.

Thank you, Steve Rabin. Without your profound foundational work in the industry, this thesis would not have been possible.

I would like to thank the departmental chairs of my time, Dr. Schiaffino and Dr. Bailie, along with faculty for allowing me to work and study in the Computer Science Department granting me this precious knowledge and skill set.

I also would like to thank my professional network of Conference Associates from the Game Developer's Conference, whose optimism and knowledge kept me motivated.

Lastly, I extend my thanks to my mother, my sister, and Stephanie Rucci, who were there behind the scenes in those gritty sleepless nights of work, always pushing me to keep going and keep smiling even when faced with the seemingly impossible.

Honorable Mentions
Dr. Lubomir Ivanov, Brother J.K. Devlin, Sister Kathleen Deignan, Willmor Pena

IONA COLLEGE

Table of Contents

INTRODUCTION

Thesis statement:

We can improve the performance of pathfinding algorithms during runtime if we can create goal bounds within goal bounds in the precomputation of a grid-based map thus further pruning nodes that to do not ultimately end up on the optimal path.

Problem Definition

In the world of gaming, there are many things that prove vexing to the player and take away from the overall experience. As the extent of our technology grows, nuisances progressively dissipate and make room for more enjoyable sessions.

As expected, a game is almost nothing without its artificial intelligence (AI). Without good AI systems, the game will be as exciting as a word processor.

Pathfinding is one area in AI in gaming that has been the subject of extensive research and improvement. Now, previous implications of pathfinding were not faulty. However, they were inefficient. The A* algorithm used to take reign. If an object in the game had to be automated to find a goal, it was most likely powered by the A* algorithm. There was an issue with this, however. A*, in most cases, was a fairly neat and quick algorithm. Despite this, there is a downside to everything. A* spent time processing almost each and every crevasse of the grid-based maps in order to find the most optimal path. Given the dynamism of gaming, this could take an even longer amount of time if there are other factors such as live moving obstacles or a misplaced collider. It could take so long that it can seem like the game is frozen in an eternal loop when it is

actually trying to find the goal as seen in a clip from L.A. Noire from Podhraski's article where the player's avatar cannot find the door and awkwardly shuffles around.[1]

Pathfinding has been optimized with JPS in order to prune many of the path candidates and avoid redundancy. Then, JPS was taken to the next level with JPS+ which requires preprocessed data to learn from. This operated on some problems one hundred times faster than A*! Goal Bounding further improves performance. With Goal Bounding, box bounds are calculated by backtracking from the current point. Bounds yield areas which have been visited or don't include the goal. Such areas are eliminated from the list of path candidates. With Goal Bounds, the search cannot go in the wrong direction. Factoring in the precomputed data, the ignoring of about seven invalid directions (assuming we can go diagonal), the pruning of redundancy, and the jump point jumping can bring search speed up to 1000x faster than A*.

JPS was used in the stead of A* but Steve Rabin took it to entirely new levels by combining JPS+ with Goal Bounding to make it one thousand times faster![2] Yet, technology still grows. It is possible to make this algorithm even faster! With JPS+ eliminating the redundant optimal paths and goal bounding eliminating going in the wrong direction, it can be said that using subgoals within the space of the goal bounds can eliminate a few steps in the goal bounding computation and increase the speed even further. This can help us to ignore jump points that do not help us find the best path.

What is "Subgoal Bounding" or Recursive Goal Bounding?

"Subgoal bounding" can be proposed as an alternative to increase speed further. It is possible to prune the nodes further with an expansion to the goal bounding algorithm, which we shall refer to as the "subgoal bounding algorithm". Subgoal Bounds (not to be confused with subgoal graphs) can be implemented by running goal bounds on each of the jump points established from the starting node. When the goal bounds are

set up within goal bounds, they create subgoal bounds. Only assessing what falls within goal bound and subgoal bound space, we can prune out the last family of jump points which get calculated but ultimately do not end up on the optimal path.

BACKGROUND AND FOUNDATIONAL WORK

Summaries

Foundational work

Introduction to A by Amit Patel*[3]

This article strictly goes over how A* works. It also includes interactive tools to take you step by step through the steps of A*. With the A* search, one must calculate an "f" value for each of the eight nodes surrounding the node in question. That which has the lowest "f" value will be put on the open list and taken as a node on the optimal path. To get the "f" value, the search must first acquire the heuristic values "g" and "h" and add them together. Remember this must be done on each of the surrounding nodes. The "g" value is how far the node is from the starting node. The "h" value is how far the node is away from the goal node. When running these calculations, A* can be comparatively slow.

Online Graph Pruning for Pathfinding on Grid Maps by Daniel Harabor and Alban Grastien[4]

In this article, Daniel Harabor's explains the identification of jump points.

Here, Harabor goes into depth on how to identify jump points in the JPS algorithm. He explains that a node will be a jump point if it satisfies the following conditions:

- The node in question is the goal node (of course)
- This node has at LEAST one forced neighbor
- You are moving diagonally and there is a node in the jump paths that counts as a jump point to your current node based on the conditions above. (Basically, if you are moving from *node x* and *node y* has a jump point *node z*, then *y will be a jump point of x as it has potential jump points itself.)*

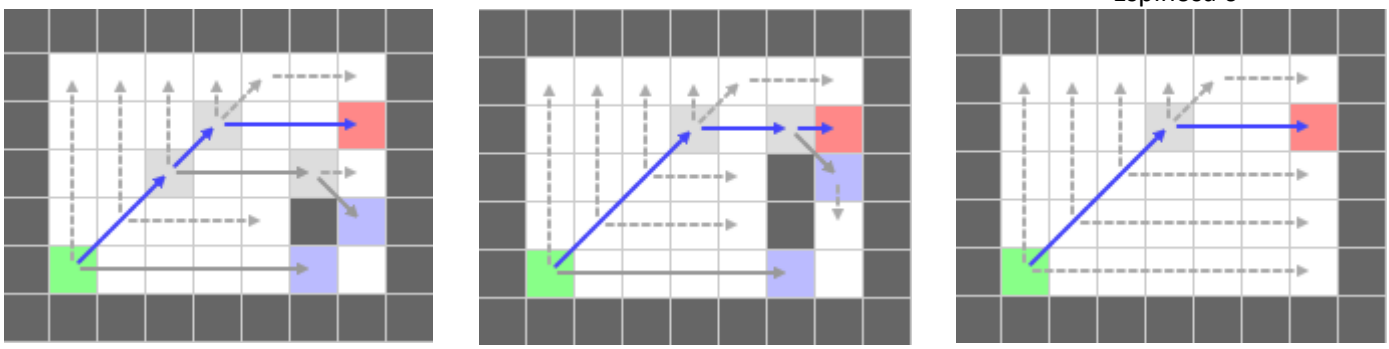
The JPS Pathfinding System by Daniel Harabor and Alban Grastien[5]

Harabor and Grastien's article on the Jump Point Search is a general overview of the JPS Pathfinding System. Before understanding JPS+, one must understand the base of the Jump Point Search. Harabor explains

that A* differs from JPS as in A*, one must consider the permutations of every expanded node. JPS avoids this by eliminating symmetric expansions. Furthermore, JPS requires no preprocessing making it an optimal algorithm candidate for pathfinding.

The article moves on to explain the theory behind JPS. The algorithm uses "pruning" and enforces its pruning rules to make decisions. If we move to the right from node *p*, for example, at the new location we can prune node *p*, its neighbors, and the neighbors of those neighbors leaving the current node we are at with only one natural neighbor. These pruned neighbors are removed from the queue and the natural neighbor is the only one taken into mind. Harabor and Grastien move on to explain how this pruning works diagonally (diagonal movements leave room for three natural neighbors to assess), and in the case of an obstacle (if there is an obstacle, one must explore the nodes beyond it making this a *forced neighbor*). After explaining the pruning rules, the articles goes on to explain the jumping rules in JPS. Each node undergoes the JPS algorithm explained above in order to find jump points. Essentially they state that the algorithm runs recursively to find it the jump path reaches an obstacle or not. If the path ends up reaching an obstacle, the path is a failed path and not added to the open list (already reducing operations). If the path does not reach an obstacle, if the node produces a forced neighbor, or if the node holds a jump path that leads to the goal, this is noted as a jump point. Examples can be seen in Figure 1 below.

Figure 1



(a) (b) (c)

(Examples generated using Nathan Witmer's JPS tool)

Harabor and Grastien use a similar example to Figure 1 in their article. Figure 1(c) shows the jump point (gray) is established by the fact that the node's straight jump leads to the goal. All other nodes are ignored. Figures 1(a) and 1(b) show the same situation but with obstacles. More jump points are established (steel blue is recently discovered jump points) as they have forced neighbors and the forced neighbors must be assessed.

The article finally touches upon JPS+. JPS+ was introduced due to the fact that identifying the jump points could cause a bottleneck (blockage). With JPS+, a jump point is computed in each direction of the node (including diagonals, which is 8 precomputed jump points). We then add jump point successors to nodes where we cross the row or column that our goal is in. That is all that is spoken on JPS+.

Shortest Path; Jump Point Search by Daniel Harabor[6]

This article expands on what Daniel Harabor spoke of in his "The JPS Pathfinding System" article. The beginning of the article mirrors what he speaks of in the other article in regard to what JPS is and its pruning rules. He goes a bit more into depth on the jumping rules in this article, however. He states "jump points . . . have neighbors that cannot be reached by an alternative symmetric path: the optimal path must go through the current node."

He then explains the performance speed of JPS in comparison to A* using real world examples. It changes depending on the type of game. There is a factor of 3-15 times greater at adaptive depth, 2-30 times faster for Baldur's Gate, and 3-26 times faster for Dragon Age. However, it is still an increase in speed at all times.

Subgoal Graphs for Optimal Pathfinding in Eight-Neighbor Grids by Peter Noevig and Carlos Hernandez[7]

This article goes into depth on using preprocessing to established subgoal graphs that can be used to optimize pathfinding. This algorithm differs from the major algorithms such as JPS+ as it is more of a "path planner" than a "pathfinder". The content in this article does not directly tie in with my thesis but it delivers excellent insight. Just as I am attempting to do, subgoal graphs eliminate the consideration of redundant jump points. Noevig and Hernandez compare it to the JPS algorithm by stating "the jump points of JPS are similar to our subgoals, except that the agent always moves on a straight line from one jump point to the next one and thus moves in only one direction whereas the agent can move from one subgoal to the next one in a combination of two directions." A visual example can be seen below in Figure 2:

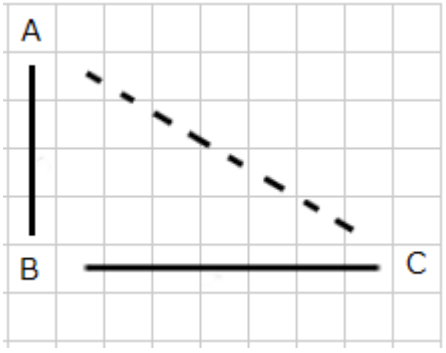


Figure 2

Here, we see jump points A, B, and C. If we were to just use standard JPS, we would first just to from A to B with a "jump cost" of 1 and a travel cost of 5. Then we move to C totaling the amount of jumps to 2 and a

total travel cost of 13. C is a subgoal of A. Given there are no obstacles we can jump right to this subgoal with at approximately half the cost. We save time jumping from point to point and find a shorter path!

Instead of evaluating a path to one goal node during execution, these subgoals are placed during pre-processing. This preprocessing takes about 7 seconds at minimum (though additional algorithms can be run increasing time but improving accuracy). With these subgoals, you work with a smaller set of nodes during execution (one no longer has to assess the all of the neighbors of each node at runtime anyone, they can simply assess distances between subgoal nodes) and this is good. Let's revisit the examples used in the summary of Harabor's articles seen in Figure 3:

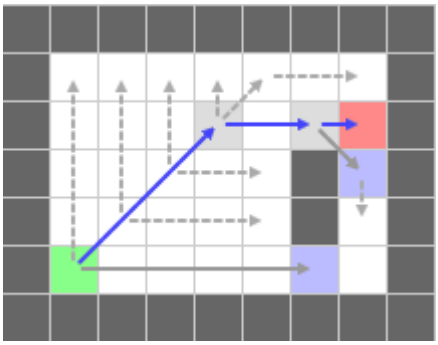


Figure 3

Here we have two additional jump points that are practically useless. We should be able to discard them but the algorithm does not call for it. We can even more jump points if we follow the idea being subgoal graphs. We wouldn't be able to jump right to the goal node but instead the jump point generated right before it. This is a small grid example so it would only be one less jump. However, imagine the possibilities. It is bittersweet as subgoals seem to take a while during preprocessing but provide a smoother and quicker experience during runtime. Yet, loading times are common and expected in games so utilizing subgoals may just be a prime choice despite its setback. While the game is being loaded into memory, the precomputation can take place.

Noevig and Hernandez move forward to explain how subgoals operate during execution. Using these subgoal graphs would actually be an alternative to JPS/JPS+ in the way Noevig and Hernandez explain it. This is not the goal of this thesis, however. It works similarly to how JPS+ places jump points on each corner of the grid. Instead we put subgoals at the corners of *obstacles* and then add jump edges to subgoals that can be reached to by one another. This forms a simple subgoal graph that can be used during execution.

They move on to explain all different types of subgoal graphs (two-level, further improvements, and experiments).

Directly Relevant Work

JPS+ with Goal Bounding: Over 1000x Faster than A by Steve Rabin*[8]

This presentation session is the real back bone behind my research. This is the latest and largest improvement in this field of study to date. This is especially useful as Rabin explains the technicalities behind JPS+, how it ties in with goal bounding, and the potential of all the algorithms to work with one another. He explains that JPS+ alone is just about 3-20 times faster than A* due to its pruning methods. However, we can precompute a bunch of data to improve it. He precomputes goal bounds using his technique of "goal bounding" which avoids going in the wrong direction. Goal bounding can be applied to multiple algorithms so it gives freedom for more precomputing or the altering of JPS+. This precomputing on Goal Bounding is slow, however. It runs on $O(n^2)$.

Rabin moves on to explain JPS and JPS+ first with detailed explanation of how to precompute the numbers for the jump point nodes that are placed on the open list in JPS+. There are four types of jump points: Primary, Straight, Diagonal, and Target Jump Points. Primary Jump Points are those established by forced neighbors. All or no jump points can be primary. The straight jump points are jump points that can make a straight jump that will reach another primary jump point (direction is important, as it must be forced neighbor).

Diagonal works similarly except it calculates the distance to straight jump points diagonally. Our target jump point is assessed during runtime. This jump point places jump points as far in each direction as possible. Target jump points are placed into the open list if one of the nodes in questions crosses the jump (row or column) to the corner nodes that the goal established.

Goal bounding constricts the search space. If we are at a starting point and we are going left, we want to consider what are all the nodes that we can get to optimally by going left, right, up, down, top right, etc. If the goal is NOT in these "optimal nodes", we should not go in that bounded direction. Just by assessing left and right, you can eliminate computing almost half them map! We calculate goal bounds for each edge of the node. We backtrack in a Dijkstra floodfill method until we can go no further. Then we have our goal bound.

JPS+ with Goal Bounding only works on grids (using octal heuristics).

Other Notable Sources

Jump Point Search Explained by Nathan Witmer[9]

A useful article for understanding JPS and the background behind it (including A*). Covers everything listed above but it especially useful for its pathfinding tool to show live runtime of A* vs JPS.

How to Speed Up A Pathfinding with the Jump Point Search Algorithm by Tomislav Podhraski*[10]

This is practically the same article by Nathan Witmer except with a live tool that records the time that the algorithms take to find the goal instead of just showing the steps in live action.

Literature Review (Relevance of Proposed Work)

JPS+ is fairly young (created in 2011). Furthermore, its tie with Rabin's Goal Bounding is almost infantile (dating just a few months back earlier this year) as explained in his seminar. Due to this, most of the literature is primarily for some form of foundation to move forward with my work. I will use all of the algorithms mentioned (A*, JPS, JPS+, Subgoal graphs, Dijkstra floodfill, and Goal Bounding) as leverage to reinvent the

way we use Rabin's JPS+ with Goal Bounding method. After review, it seems that no one has attempted to integrate a subgoal bound system into Rabin's already existing method making my work distinct and relevant.

IMPLEMENTATION

METHODOLOGY OF EVALUATION

Algorithms

The following algorithms will be used:

1. JPS/JPS+
    a. Used to precompute jump points
2. Goal Bounding
    a. Precomputation algorithm that when combined with JPS+ makes the fastest method of pathfinding known to date
3. Subgoal Graphs
    a. Different from the proposed work of "subgoal bounds"
    b. Will not be used directly. The idea behind it is used to compute subgoal *bounds*.
    c. Can be used later on if time allows in order to optimize the path finding instead of JPS+

Languages and Software

The following languages and software will be used:

1. Unity5 Engine
    a. This will be used to create scenes that will visually represent and tests of this new algorithm
2. C#
    a. Language of choice and one of the languages of Unity
    b. Any implementation will be written in this

3. C++
    a. Preexisting implementations of JPS+ with Goal Bounding already exist but they are in C++. They will be converted into C# for Unity.

Sample Sizes

Bioware has been generous enough to disclose some of their maps for research purposes.[11] These grid maps are directly from their games and can be used or remodeled to test my work on a live, relevant scale. The samples can be seen and downloaded at the following link: http://www.movingai.com/benchmarks/

Other than this, there is no other major sample sizes to work from. We know that the latest JPS+ with Goal Bounding method could be 1000x faster than A*. JPS is about 3 times faster than A* and all of the precomputed data from JPS+ makes it 100x faster. Goal bounding gives it that final kick to 1000x. That is our platform.

Method of Evaluation and Expected Results

Procedure

1. Recreate the JPS+ with Goal Bounding algorithm within the space of the Unity Editor.
2. Expand on the goal bounding code
    a. Take the precomputed jump point(s) within each goal bound and calculate goal bounds for them
3. See if we can somehow get the goal before runtime
4. Assess how the subgoal bounds work with the newly generated target jump points that are created during runtime with JPS+
5. Should it work, test on multiple maps.

Experiments and Expected Results

1. See how JPS+ with Goal Bounding fairs against A* in the Unity Engine, just to confirm its increase in speed.

   a. In other settings, the difference is apparent. There is no place where I can find JPS+ with Goal Bounding specifically next to A*. I know it will be faster, it is just good to have.

2. Test to see if we can establish bounds simultaneously on each jump point within the major goal bound. Is this efficient or should we do one at a time?
   a. I expect simultaneously to work best but there then leaves room for the data to get "over processed" (meaning we'd be doing extra work for no reason and actually going backward). It may be best to just choose one jump point to make bounds and go from there.
3. Test to see if more layers of subgoal bounds means more accuracy
   a. I expect it to work well. However, there has to be some sort of cutoff point as it will soon just turn into single nodes!

Implementation

The work began in Unity3D using C# to provide a nice visual representation. The initial plan was to recreate the JPS+ with Goal Bounding algorithm from the ground up in order to test both said algorithm and the subgoal bounding algorithm. The major issue with this was that the algorithm would be able to run fine on these platforms but there would be no collection of significant data. These algorithms needed to be run on actual maps of varying sizes rather than small scale, statically generated maps for testing. The topic of the project quickly went from building a testing this subgoal bounding algorithm into building a JPS+ with Goal Bounding port for Unity3D maps. To save time and focus on the research at hand, the primary programming language was switched to C++.

All existing directly relevant foundational work had been done in C++. By making the switch to C++, there were more available resources to use in order to get real and significant data to work with. The JPS+ with Goal Bounding source code given by Steven Rabin was especially useful. It not only provided a (nearly) functional program to build off of but more importantly this source code included map parsing algorithms which were essential to the strives in research.

The maps used in this work were acquired from the game *Dragon Age: Origins* which were provided for research purposes through Moving AI's Pathfinding Benchmark Problems[12]. These maps were provided in an ASCII grid map as seen below in Figure 5.

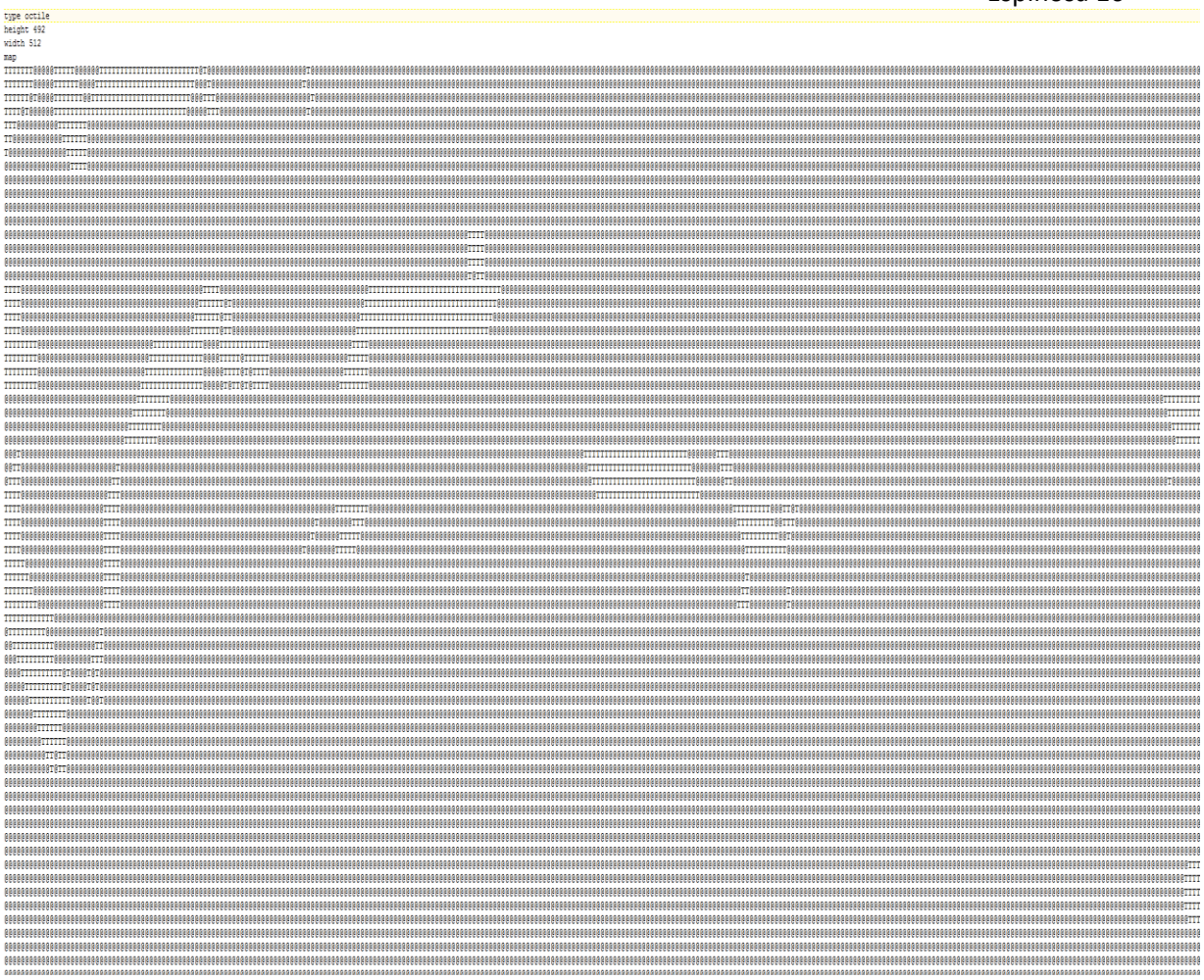


Figure 5.1 – Brecilian Forest; ‘brc100d.map’ file contents. ASCII grid

Key:

- . = passable terrain
- G = passable terrain
- @ = out of bounds
- O = out of bounds
- T = trees (unpassable)
- S = swamp
- W = water

Figure 5.2 – ‘.map’ file ASCII key.

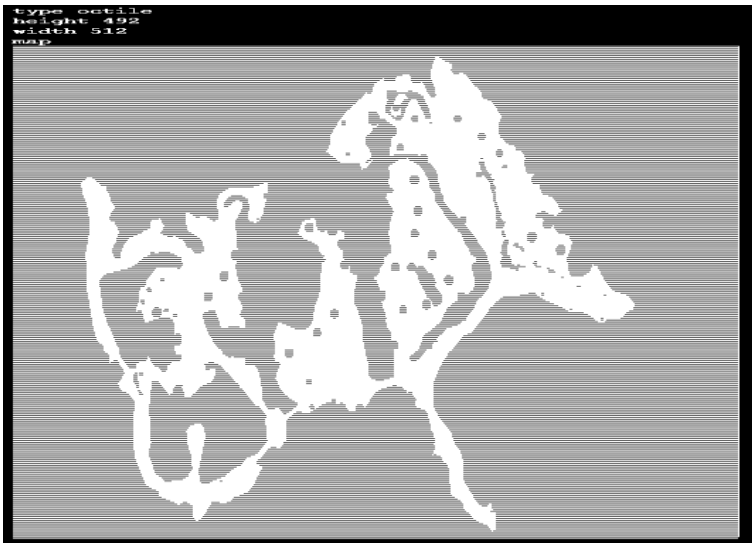


Figure 5.2 – Brecilian Forest; grid zoomed out, condensed view of ASCII grid 'brc100d.map'

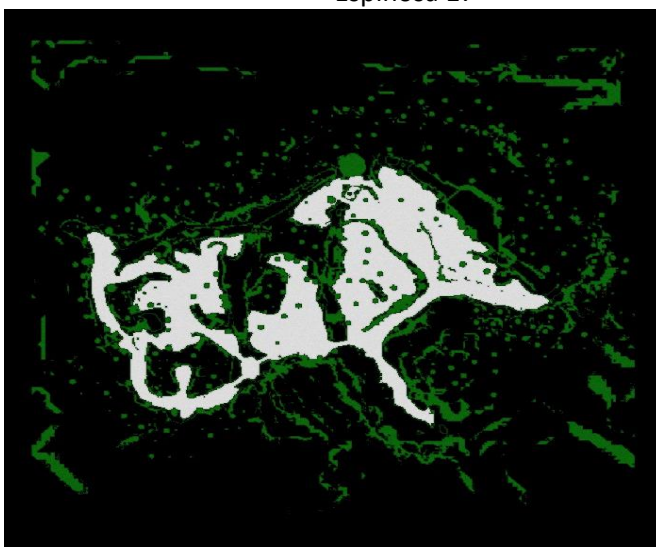


Figure 5.3 – Brecilian Forest; graphical representation of 'brc100d.map'

If work continued in Unity3D, the ASCII characters in these files would need to be parsed and maps would need to be dynamically generated. Of course, this is not impossible but, as stated, this would have strayed far from the research at hand whereas the code in C++ was already set to parse these maps.

With this toolset switch, it was time to test the JPS+ with Goal Bounding before any further progress could be made. The source code was not as straight forward as the algorithm entailed (most likely to promote organization and versatility), thus, adding a layer of complexity. The project included the following files with their corresponding headers:

1. Main – Runs pathfinding based on information from all other files (.pre files, .scen files, .map files, which file to write to, etc)
    a. Entry – mainly for writing precomputation file
2. Map – for creating map to use in endeavors
3. Precompute Map – Goal Bounding

a. JPS+ -- Established the jump points for path optimization and for basing goal bound off of
b. Dijkstra Floodfill – runs a Dijkstra floodfill for goal bounding
    i. Bucket Priority Queue – data structure for handling bucket
        1. Unsorted Priority Queue – base data structure used for bucket priority queue
            a. Pathfinding Node – established node structs to use while calculating optimal path

4. Timer – non-standard constructed timer specific for this project
5. Scenario Loader – parses scenario files to run test problems on different start and goal nodes on maps
6. Miscellaneous Data Structure Files
    a. Fast Stack
    b. FPUtil
    c. Generic Heap

All of this was loaded into Visual Studio 2015 and was almost ready to run. However, though the code was available for public use it was not written as a ready tool. Having been developed between 2014-2015, the existing source code needed to be updated to eliminate deprecated C++ libraries, data structures, and methods in 2016. The major issue was correcting the issues with the hash_map data structure. Since the implementation of this algorithm, this had since been deprecated and needed to be updated in both code and the standard library. After this, the code was operational but only statically read maps that were included within the repository. Some code to prompt the user for a map was created in the main file so that maps could be downloaded and chosen to test (see Figure 6).

```
Enter a map name: arena.map
Begin preprocessing map: Maps
Writing to file 'Maps\arena.m
Goal Bounding Preprocessing
Row: 0
Row: 1
Row: 2
Row: 3
Row: 4
Row: 5
Row: 6
Row: 7
Row: 8
Row: 9
Row: 10
Row: 11
Row: 12
```

Figure 6 – arena.map being entered and preprocessing beginning

```
subopt  0.999992        valid

GPPC    Maps\arena.map.scen     total-time      98.000000       max-time-step
98.000000       time-20-moves   98.000000       total-len       61.325400
subopt  0.999992        valid

GPPC    Maps\arena.map.scen     total-time      105.000000      max-time-step
105.000000      time-20-moves   105.000000      total-len       61.153800
subopt  0.999992        valid

GPPC    Maps\arena.map.scen     total-time      128.000000      max-time-step
128.000000      time-20-moves   128.000000      total-len       60.911200
subopt  0.999992        valid

GPPC    Maps\arena.map.scen     total-time      191.000000      max-time-step
191.000000      time-20-moves   191.000000      total-len       61.325400
subopt  0.999992        valid

GPPC    Maps\arena.map.scen     total-time      98.000000       max-time-step
98.000000       time-20-moves   98.000000       total-len       62.153800
subopt  0.999992        valid
Total map time: 11787.000000,   Maps\arena.map
```

Figure 7.1 – Results with standard goal bounding for the last few benchmark problems in the scenario files for arena.map

```
15      maps/dao/arena.map      49      49      1       40      47      3       61.3259
15      maps/dao/arena.map      49      49      1       41      46      2       61.1543
15      maps/dao/arena.map      49      49      1       45      47      9       60.9117
15      maps/dao/arena.map      49      49      1       7       47      44      61.3259
15      maps/dao/arena.map      49      49      1       7       47      46      62.1543
```

Figure 7.2 – Last five scenarios for arena.map (see Figure 3.3 for explanation)

Bucket	map	map width	map height	start x-coordinate	start y-coordinate	goal x-coordinate	goal y-coordinate	optimal length
0	maps/dao/arena.map	49	49	1	11	1	12	1
0	maps/dao/arena.map	49	49	1	13	4	12	3.41421

Figure 7.3 – Key and example for scenario files from Moving AI Pathfinding Benchmarks[13]

Now that maps could be tested dynamically, a couple of maps of varying scales were put through the algorithm and seemed to work efficiently (see Figure 7). It was then time to move on to implementing the subgoal bounding algorithm.

First, the actual place in code where the jump points were calculated needed to be found. This was done in the JPSPlus.cpp file in eight separate functions that checked each direction for jump points. The snippet of code can be seen below:

```
void JPSPlus::SearchDown(PathfindingNode * currentNode, int jumpDistance)
{
	int row = currentNode->m_row;
	int col = currentNode->m_col;

	// Consider straight line to Goal
	if (col == m_goalCol && row < m_goalRow)
	{
		int absJumpDistance = jumpDistance;
		if (absJumpDistance < 0) { absJumpDistance = -absJumpDistance; }

		if ((row + absJumpDistance) >= m_goalRow)
		{
			unsigned int diff = m_goalRow - row;
			unsigned int givenCost = currentNode->m_givenCost + FIXED_POINT_SHIFT(diff);
			PathfindingNode * newSuccessor = m_goalNode;
			//myFile.open("C:/Users/jespinosa2/Desktop/JumpPoints.txt");
			//myFile << "Testing\n";
			myFile << "Down Successor: ( Row ";
			myFile << m_goalNode->m_row;
			myFile << " , Column ";
			myFile << m_goalNode->m_col;
			myFile << " );\n\n";
			////myFile.close();
			PushNewNode(newSuccessor, currentNode, Down, givenCost);
			return;
		}
	}

	if (jumpDistance > 0)
	{
		// Directly jump
		int newRow = row + jumpDistance;
		unsigned int givenCost = currentNode->m_givenCost + FIXED_POINT_SHIFT(jumpDistance);
```

```
                PathfindingNode * newSuccessor = &m_mapNodes[newRow][col];

                //myFile << "Testing\n";
                myFile << "Down Successor: ( Row ";
                myFile << newRow;
                myFile << " , Column ";
                myFile << col;
                myFile << " );\n\n";
                ////myFile.close();
                PushNewNode(newSuccessor, currentNode, Down, givenCost);
        }
}

void JPSPlus::SearchDownRight(PathfindingNode * currentNode, int jumpDistance)
{
        int row = currentNode->m_row;
        int col = currentNode->m_col;



// . . .

//Functions continue on
```

Figure 8 – Jump Point pushing source code

These jump points were printed to a file to get a feel for how many recursions would need to be done in order to properly implement the subgoal bounding algorithm. When printed to a file using arena.map (see Figure 6 for all maps used in testing), the file ended up having approximately 64,000 pages of jump points. This was due to the multiple scenarios giving varying jump points. The first scenario gave a round of eight jump points (see Figure 9).

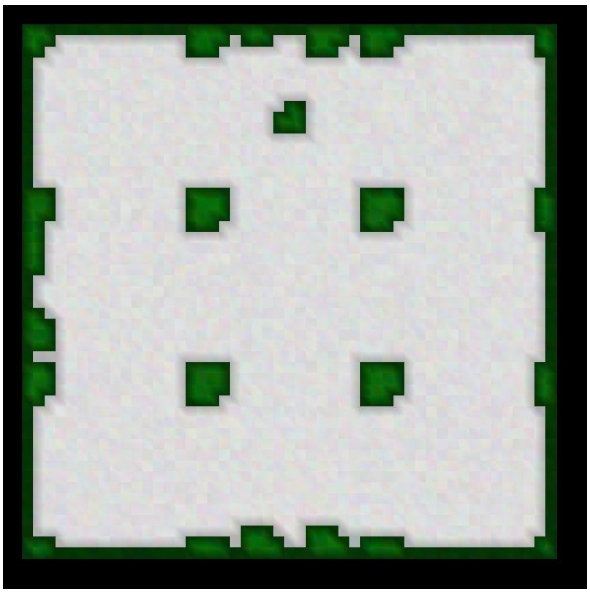
```
Down Successor: ( Row 12 , Column 1 );
Down Right Successor: ( Row 13 , Column 3 );
Up Successor: ( Row 10 , Column 1 );
Up Right Successor: ( Row 12 , Column 2 );
Up Right Successor: ( Row 12 , Column 4 );
Up Right Successor: ( Row 3 , Column 2 );
Up Right Successor: ( Row 3 , Column 18 );
Up Right Successor: ( Row 2 , Column 3 );
Up Successor: ( Row 1 , Column 3 );
```

Figure 9 – Jump Points for arena.map. The first eight pertain to the first problem in the scenario file and the last one is the beginning of jump points for the second scenario.

It was attempted to now run the subgoal bounding algorithm. However, there were two major algorithm setbacks. The issue was the fact that goal bounding algorithm relied on the jump points of the starting node to determine where the goal bound would be. Using the jump points as starting nodes on the recursion would not work due to the fact that they did not have jump points themselves. Also, the ASCII grid would need to be parsed on each iteration. The subgoal bounding algorithm was meant to serve as an expansion JPS+ with Goal Bounding algorithm but in order to get the desired result, each of the algorithms would all need to be deconstructed. Deconstruction of an algorithm with foundation spanning years back would not be a feasible feat in the time given. Thus, theoretical alternatives were tested to assess to see if it would be worth pursuing that next step in future research.

For the subgoal bounding algorithm to be necessary and work properly, there would need to be a steady number of jump points. The eight jump points from arena.map seemed like a good number of recursions to test under (other maps had more than eight jump points in their realistic scenarios). We ran the subgoal bounding algorithm by running the goal bounding algorithm at the worst case of the smallest map. This way, data sample

sets of time could be used in order to determine whether or not research should press forward. The results are seen in the proceeding section.



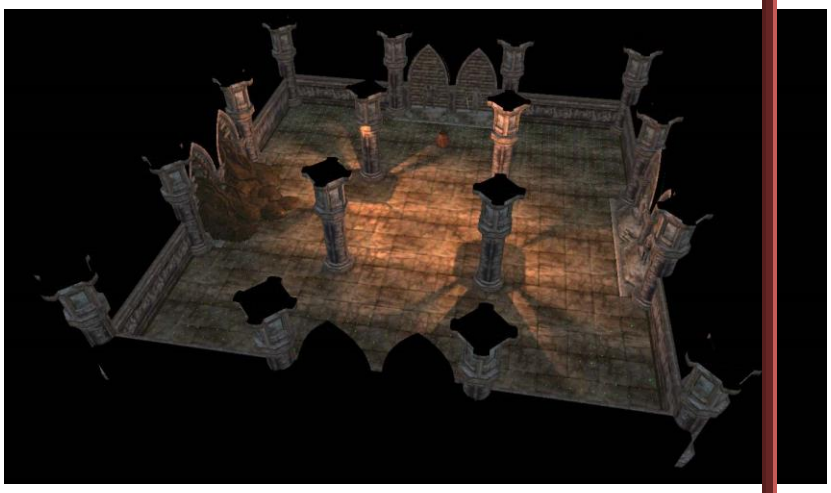
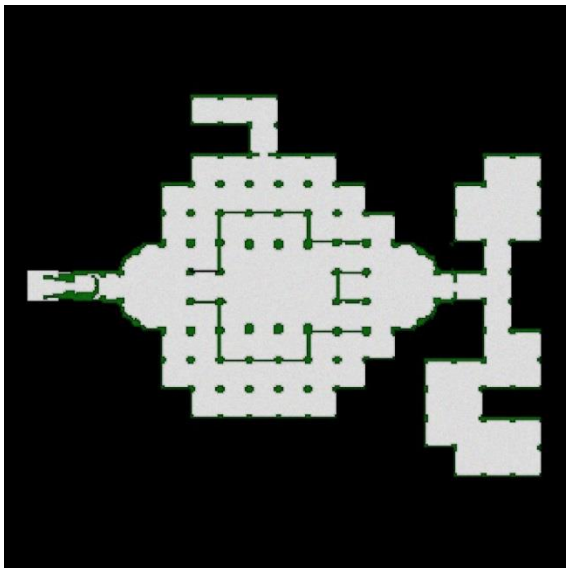


Figure 10.1 – arena.map graphical representation and corresponding 3D scene, “Small Castle Arena”



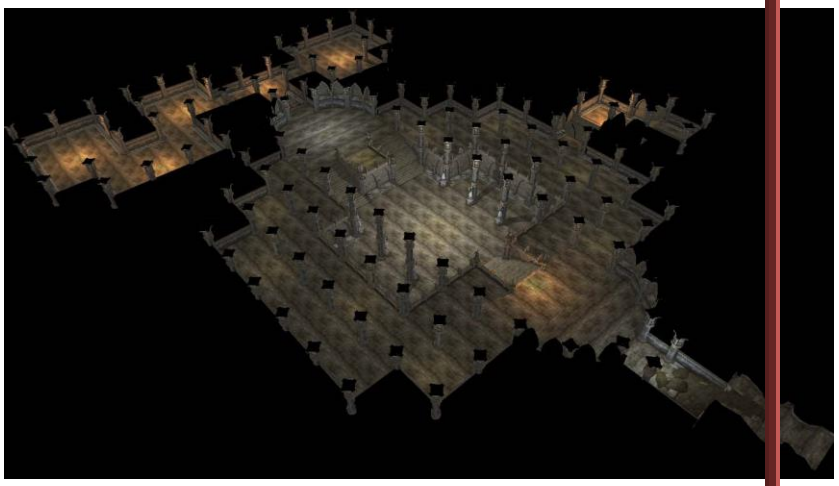


Figure 10.2 – arena2.map graphical representation and corresponding 3D scene, “Large Castle Arena”

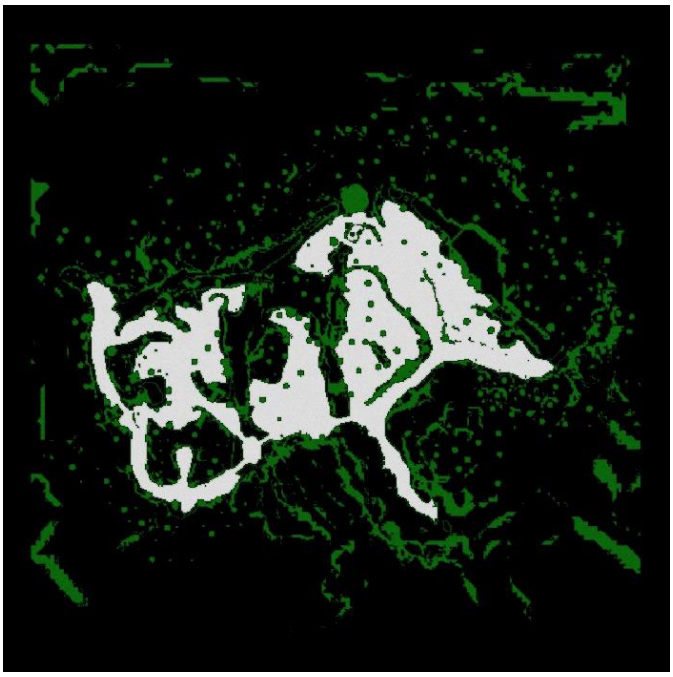
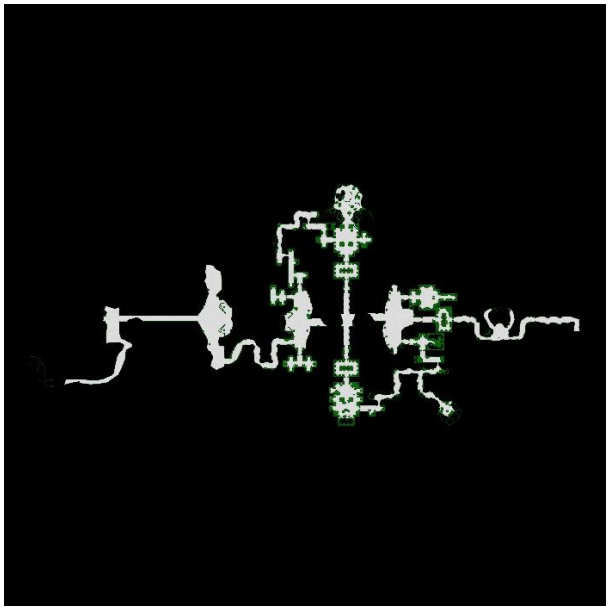


Figure 10.3 – brc100d.map graphical representation and corresponding 3D scene, “West Brecilian Forest”



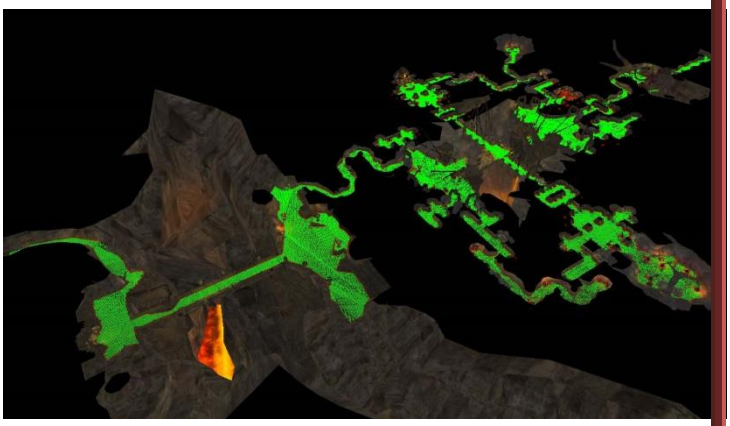


Figure 10.4 – orz900d.map graphical representation and corresponding 3D scene, “The Deep Roads: The Dead Trenches ”

RESULTS

Results

As explained in the Implementation section, the precomputation of the maps were run on the worst case on the smallest map. This means that each map ran the subgoal bounding algorithm through eight cycles of recursion. The algorithm was programmed to collect the time it took to run the algorithm and the results can be seen in the tables below.

	"arena.map" Timing (small size)	
	Standard (Goal Bounding)	Recursive (Subgoal Bounding)
	233	1572
	225	1812
	227	1779
	223	1811
	239	1813
Average	229.4 ms (0.22 sec)	1757.4 ms (1.75 sec)

Figure 11.1 – arena.map timing results

	"arena2.map" Timing (medium size)	
	Standard (Goal Bounding)	Recursive (Subgoal Bounding)
	33822	278825
	34332	278613
	37123	248002
	34434	257023
	34931	278333
Average	34928.4 ms (34 sec)	268159.2 ms (4.46 min)

Figure 11.2 – arena2.map timing results

	"brc100d.map" Timing (large size)	
	Standard (Goal Bounding)	Recursive (Subgoal Bounding)
	67333	570000
	67440	544902
	68738	549847
	67703	560027
	68640	544902
Average	67970.8 ms (1.13 min)	553935.6 ms (9.17 min)

Figure 11.3 – brc100d.map timing results

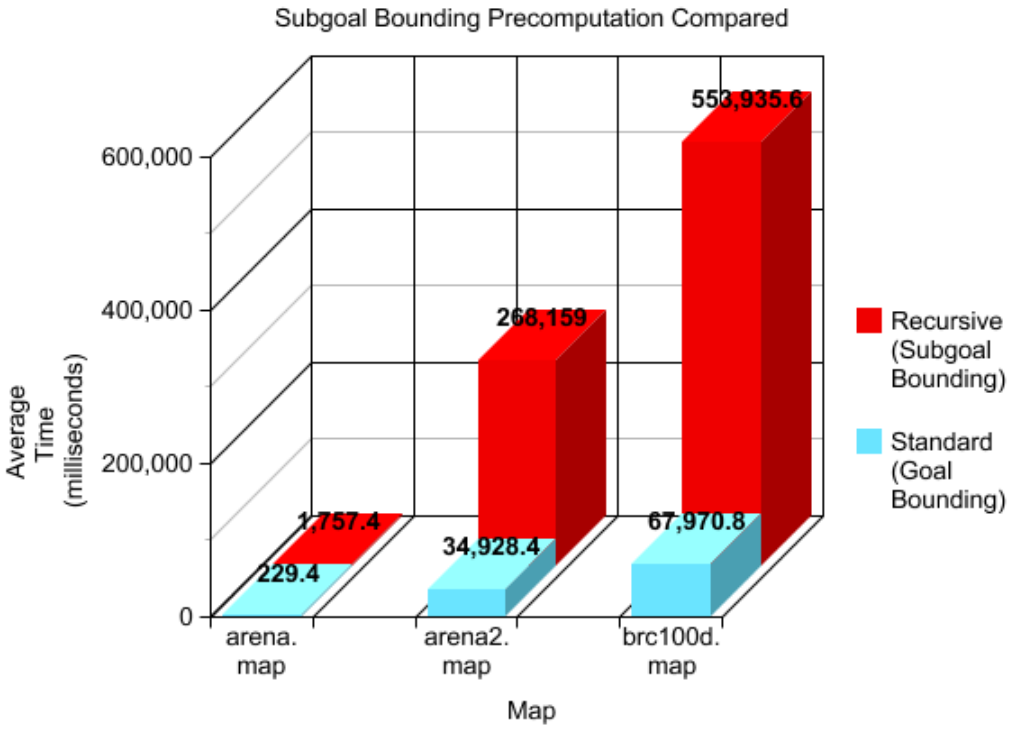


Figure 12 – Bar graph presentation of data provided in Figure 1.

As seen, the precomputation took far too long to compute the goal bounds for each of the nodes in question. In addition, the larger the map, the more prevalent the time drawback was. Looking at Figure 11.1, you can see that arena.map is fairly small in size. As it was only a 49x49 grid, goal bounding took mere milliseconds. When running on maps of more common sizes, the difference seen.

Goal bounding was fairly slow to begin with as it ran on $O(n^2)$. The recursive subgoal bounding algorithm ended up running on $O(n^2*n)$. Though this may not seem like a lot, the difference is clear when demonstrated in a graphical format (see Figure 12 and Figure 13).

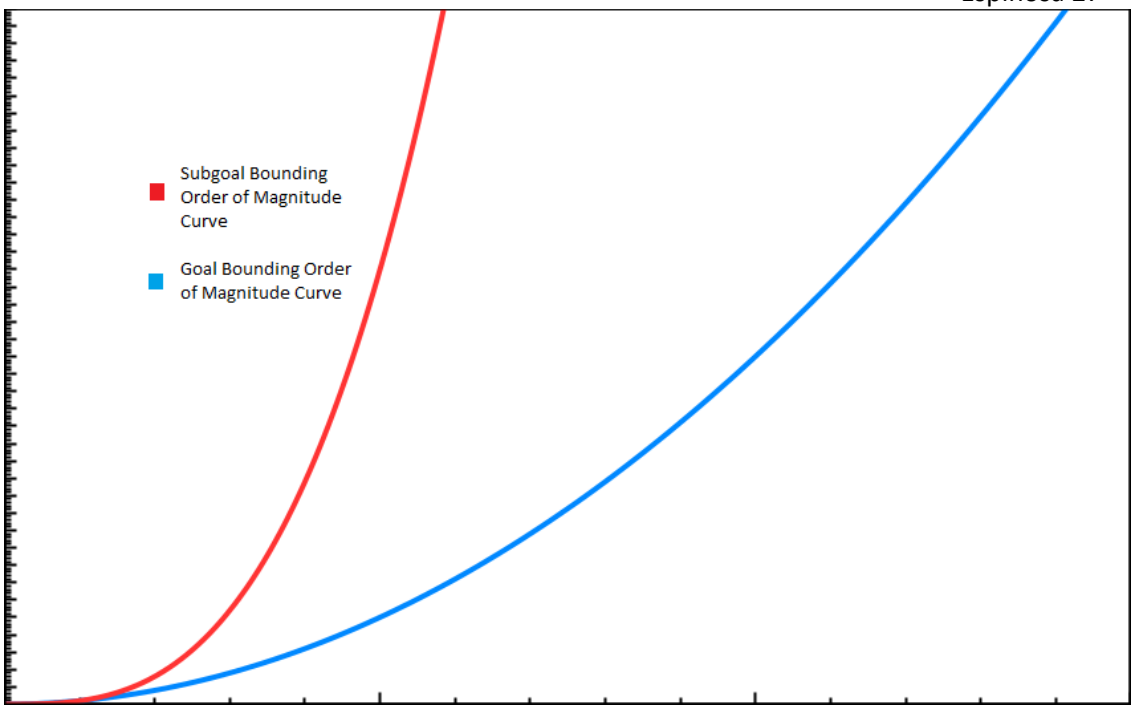


Figure 13 – Order of magnitude comparison between the two algorithms

Though it was expected that the precomputation would take a longer, such an increase in wait time is preposterous to consider as alternative to save computational time during execution. The largest map available, orz900d.map, was tested and took approximately two hours to precompute. Waiting for your game's map to load for two hours just to save a few seconds deciding the optimal path during runtime is simply infeasible.

Research Questions and Answers

1. See how JPS+ with Goal Bounding fairs against A* in the Unity Engine, just to confirm its increase in speed.
    a. The Unity engine is no contributing factor in computational speed. As JPS+ with Goal Bounding is currently the fastest algorithm it will run faster.
2. Test to see if we can establish bounds simultaneously on each jump point within the major goal bound. Is this efficient or should we do one at a time?

a. Bounds should be established one at a time due to map constraints. As jump points do not have their own jump points to determine the size of goal bounds, the bounds must be calculated by running a floodfill through the entire map. Each time this happens, the map must be loaded once again. If this was run simultaneously, map data could potentially be lost and the bounds would be established incorrectly thus yielding inaccurate results.

3. Test to see if more layers of subgoal bounds means more accuracy
    a. One would need to run the subgoal bounding for each jump point. Running the subgoal algorithm again would yield no significant difference in accuracy but would instead waste even more time.

Conclusions

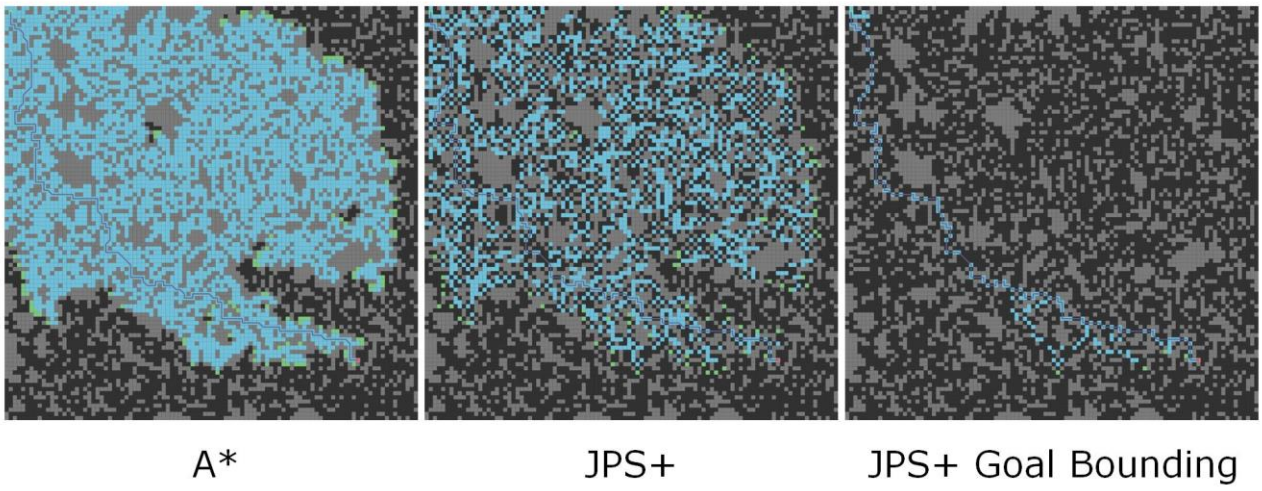


Figure 14 – Pathfinding Optimization over the years, Rabin 2015

Implementing the subgoal bounding algorithm is not an efficient way to optimize the current state of the art JPS+ with Goal Bounding algorithm. Since its release in 2015, the JPS+ with Goal Bounding algorithm has not yet been optimized in the years it has been out and for good reason. In Figure 14, we can see the evolution of optimization over the years. This algorithm was a drastic leap efficiency as you can see those considered to

be on the open list are almost all on the optimal path. No new research has been presented in this field as this seems it will be the most efficient for quite some time.

Thus, the focus should lie on how we can use it. This powerful algorithm can currently only be used on grid-based maps using C++. In the current day and age, games are being built on multiple engines and in various languages. The Unity market, for example, is becoming an industry standard. However, available assets and open source code is for A* only. Porting the algorithm to agree with these engines and their languages will be projects on their own and its important to focus attention on this. Future research can even strive to use on navigational meshes on 3D terrain instead of grid-based systems. It is time to put focus into making tools using this optimized algorithm and bring it to all platforms.

BIBLIOGRAPHY

[1] Podhraski, Tomislav. "How to Speed Up A* Pathfinding with the Jump Point Search Algorithm." *Game Development Tutorials*. 12 Mar. 2013. Web. 29 Sept. 2015. <http://gamedevelopment.tutsplus.com/tutorials/how-to-speed-up-a-pathfinding-with-the-jump-point-search-algorithm--gamedev-5818>

[2] Rabin, Steve. "JPS+ with Goal Bounding: Over 1000x Faster Than A*" Game Developer's Conference. San Francisco. 3 Mar. 2015. Web. 29 Sept. 2015.

[3] Patel, Amit. "Introduction to A* ." *Red Blob Games*. Ed. Amit Patel. N.p., n.d. Web. 18 Dec. 2015. <http://www.redblobgames.com/pathfinding/a-star/introduction.html>.

[4] Harabor, Daniel, and Alban Grastien. "Online Graph Pruning for Pathfinding on Grid Maps". In National Conference on Artificial Intelligence (AAAI), 2011.

[5] Harabor, Daniel, and Alban Grastien. "The JPS Pathfinding System." Proceedings of the Fifth Annual Symposium on Combinatorial Search: 207-08. Web. 13 Sept. 2015.

[6] Harabor, Daniel. *Shortest Path*. 7 Sept. 2011. Web. 12 Oct. 2015.

[7] Uras, Tansel, Sven Koenig, and Carlos Hernandez. "Subgoal Graphs for Optimal Pathfinding in Eight-Neighbor Grids ∗." 13 Sept. 2015.

[8] Rabin, Steve. "JPS+ with Goal Bounding: Over 1000x Faster Than A*" Game Developer's Conference. San Francisco. 3 Mar. 2015. Web. 29 Sept. 2015.

[9] Witmer, Nathan. "Jump Point Search Explained." *ZeroWidth Positive Lookahead*. Ed. Nathan Witmer. 29 Sept. 2015.

[10] Podhraski, Tomislav. "How to Speed Up A* Pathfinding with the Jump Point Search Algorithm." *Game Development Tutorials*. 12 Mar. 2013. Web. 29 Sept. 2015.

[11] Sturtevant, Nathan. *Pathfinding Benchmarks*. Web. 29 Sept. 2015 <http://www.movingai.com/benchmarks/>.

[12] Sturtevant, Nathan. *Pathfinding Benchmarks*. Web. 29 Sept. 2015 <http://www.movingai.com/benchmarks/>.

[13] Sturtevant, Nathan. *Pathfinding Benchmarks*. Web. 29 Sept. 2015 <http://www.movingai.com/benchmarks/>.