Final System Report

John P. Baggs and Matthew S. Renner

System Functionality:

Final System

Navigation Mesh:

The Navigation Mesh is a 2D representation of the three dimensional environment of the game level. It will be a collection of convex polygons that represent only the traversable areas of the level. It is made by getting a list of surface polygons, merging polygons, subdividing polygons, re-merging again, and finding connecting neighbor polygons. The gathering of surface polygons is when every polygon whose normal says it is a floor polygon is put into a list. The first merge of polygons is done with a shared edge. If two polygons share an edge, meaning they share two vertices, then they are merged. This is done by going in a clockwise fashion around the first polygon until it reaches the first shared vertex, then it adds the second polygon's vertices until it reaches the second shared vertex. This creates a new polygon that will be available to merge with the remaining polygons. When the initial merging is done all that is left is big polygons that represent the entire level. Subdividing of the polygons uses a binary space partition tree to cut out an obstacle from the polygon. It goes through a list of the obstacles and with each obstacle finds all the polygons that have part of that obstacle inside it. Then going around the obstacle and the polygon the subdivision algorithm finds a point on the edge of the polygon that intersects with an edge of obstacle. This becomes the first point of the new polygon. The subdivision algorithm then continues until all the edges of the obstacle are used and all the new polygons are free of the obstacle. These new polygons are added to the list of polygons that represent the surface of the level and can be used again to remove another obstacle. When all obstacles have been removed from the list of polygons the next step is trying to re-merge any two polygons who share two vertices using the method described above. The last step is to find what polygons are neighbors with each other. There are three distinctions we make for polygons being neighbors. A neighbor can share two vertices, or have one side contained inside a side on the other polygon, or have an overlapping side from one polygon to another. Each neighbor connection is checked against the width of the agent so no connection is made that the agent could not fit through going from one polygon to another.

Search Agent:

For this final system we have implemented multiple search agents. Each one is a variant of A* that changes some detail of the implementation to try to improve an aspect of the metrics of the system. These searches are along with A*: A* optimal, Bi-Directional, Bi-Directional optimal, Fringe, Beam, Dynamic Weighted, and Dynamic Weighted BDBOP.

A* is a heuristic based search which implements backtracking using an open list and a closed list. When it goes through an iteration it looks at the open list and finds the node that has the lowest total cost function. This cost function is the cost to reach the current node plus the cost from the current node to the goal. The cost from the current node to the goal is the heuristic cost. For this project we used the straight-line distance for each search since it is a good approximation of the actual cost. When it gets the node with the lowest total cost it checks to see if that node has the goal. If it doesn't have the goal it is placed on the closed list, saying it has been looked at it. Then the neighbors are added to the open list and it goes through the iteration again.

Bi-Directional search is a form of A* that uses two A* searches, usually simultaneously. One search goes from the agent to the goal while the second search goes from the goal to the agent. The search ends when the two searches meet or one search has found its goal. The two searches have meet if the current node for one search is in the closed list of the other search. Once either one of the two ending states have been reached a path can be constructed using recursive pre-order and post-order traversals.

A* optimal and Bi-Directional optimal searches act like the regular version of the searches but the implementation is changed to make them run faster. For the optimization of the searches the closed list was removed and replaced with a lookup table. This took out the need for a O(n) linear search of the list to see if a node was on it. When the search needs to see if a node is on the closed list it simply looks up the node's id on the look up table. The open list has a lookup to see if a node is on it, so again this lookup was replaced to remove the expensive linear search for the O (1) search of a lookup table. The final improvement was the implementation of a Min-Heap for the open table. This speeds up the insertion (O(n)) and deletion (O (1)) for the open list to an insertion of $O(\log_2(n))$ and a deletion of $O(\log_2(n))$.

Fringe Search is a combination of iterative deepening search and A*. Fringe Search uses a now and later list instead of a closed and open list. The now list is the list of nodes it can look at and the later list is nodes that will be considered on the next iteration. While going through the now list it keeps a track of the max value that it wanted to expand but couldn't because of the limitations of the IDA*. When the iteration is done it changes the max depth the search can go to to the new max value it wants to expand. It goes through the now list in a sequential fashion expanding if it's allowed each time removing nodes and either putting them on the later list or not. Once the now list is empty the two lists are swapped and the search is started again with the new now list, which is really the later list from the last iteration. This continues till either the goal is found or both lists are empty.

Beam search is created with a very simple change to A*. For our beam search we used the optimal version of A*. The change to go from A* to beam search is a limit on the open list size. If the open list is at a certain size, set by the designers of the search, then when it tries to add a new node it has to remove one from the list. The node with the largest total cost function is removed from the list and the new node is added. This simple change keeps the worry of space complexity out of the minds of the designers.

Dynamic weighted search is another simple change to A*. This time the idea behind the creation is not the space complexity but instead the time complexity of the search. The algorithms main thought in the start of the search is to get moving. It does this by putting a weight, that changes as the search goes on, to the heuristic cost for the total cost function. As the search goes on the weight gets lower and lower until the real cost of the heuristic becomes the deciding factor in the total cost function again like A*. The weight we used was a function that takes the number of nodes that were expanded to get to the current node and the total number of nodes the search thinks it will take to get to the goal into account. It divides number of nodes expanded by the total number of nodes the search thinks it will take to get to the goal. The quotient is then subtracted from one and multiplied by a constant used to cap the weighting of h. Because the time complexity was the main factor in creating this search it is not guaranteed to find an optimal path.

Dynamic weighted BDBOP search is the search we created. It combines the aspects we believed made the most successful search for both time and space complexity. It uses aspects from dynamic weighted, BI-Directional, beam, and the optimized version of BI-Directional. The dynamic weighted aspect is so the search can get going to speed it up. The BI-Directional part is to help decrease the number of nodes visited by the search, in turn also speeding it up. The beam search was used to keep the space complexity down since BI-Directional's max queue size can be big since there are two queues active at the same time.

Lastly the optimized version of BI-Directional was added to again speed up the time for the search.

Data:

For the purpose of comparing the searches used the program can be ran multiple times with each search or with all the searches on a level. If the user wants they can let the program run 1000 times per search on a map. Also the user can choose to run all the algorithms on the loaded level. When the algorithm is done running it will send the information for both the navigation mesh and the search to a file that can be used later to compare the searches as the user sees fit. The data that is written for the search agent is the time it took to complete, the number of nodes visited, the max queue size, the path length in number of polygons for the final path, and the path cost for the final path. For the navigation mesh the data that is written to the file is the time it took to either load or create, number of initial polygons, number of obstacles, and the number of final polygons in the navigation mesh.

System Evaluation:

Navigation Mesh:

For this final system we did not evaluate our navigation mesh. Before we implemented the navigation mesh we did research on search space representation of 3D environments and did a small evaluation before choosing which type of search space to go with. The navigation mesh, unlike the square grid or triangle search spaces, represents only the walkable areas of the environment. With a square grid you would have to check each time before you add a node to the open list if the square has an obstacle in it or over it. This adds to the time the search takes. For the triangle search space, usually no triangles are kept if they have an obstacle in them, but this can lead to an incomplete representation of the environment and still a lot of nodes to search through. With the navigation mesh the full environment can be represented and only a small amount of polygons, compared to the amount of polygons needed to represent the environment with a triangle or square grid search space. With this evaluation we saw that a navigation mesh was the best way to represent the entire traversable environment and the best way to decrease the size of the search space. This all being said the navigation mesh is not perfect for our implementation. Our navigation mesh cannot deal with inclines, or obstacles that are not squares. It can deal with multiple obstacles which allows us to build complex levels that can stress the search algorithms

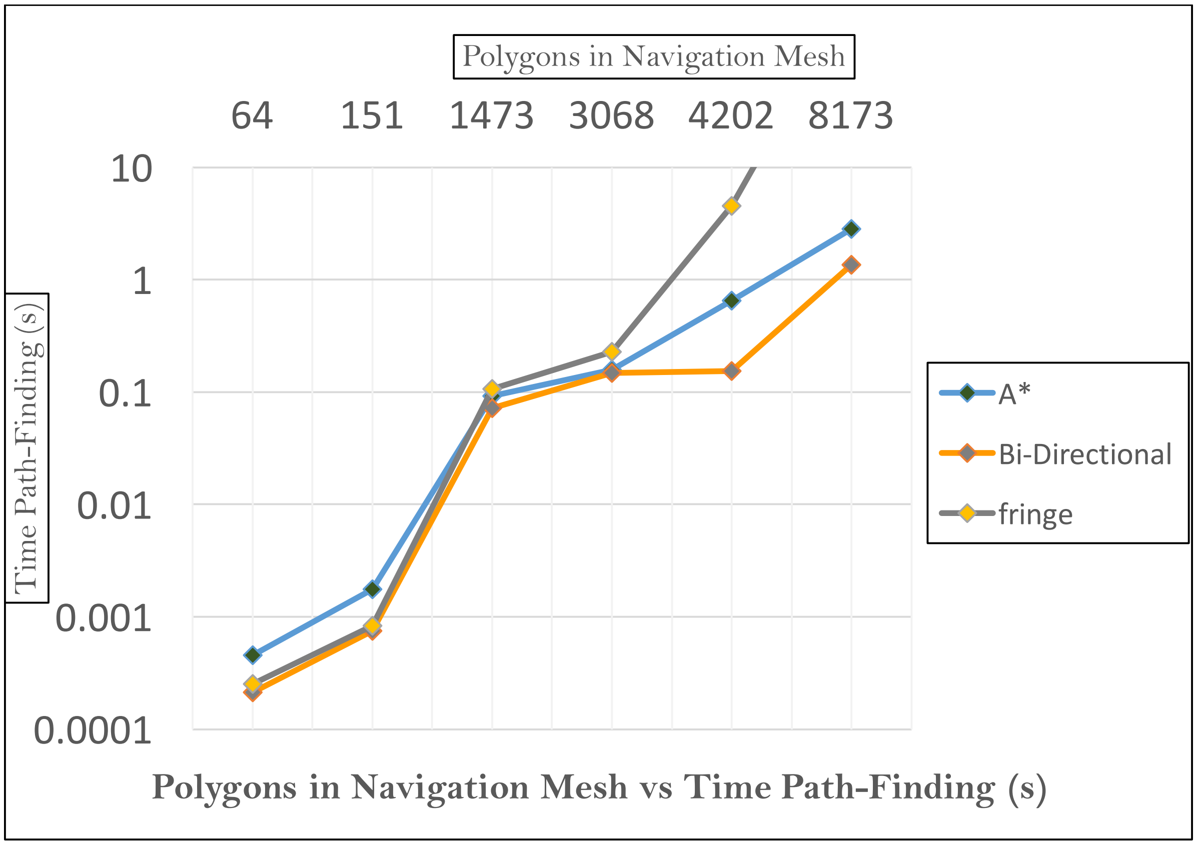
Search Algorithms:

Each search algorithm was measured on the same metrics: time taken to do the search, number of nodes visited, max size of the queue, and the path cost in units. After these metrics we looked at the path itself. This path tells us if the algorithm gives us an optimal solution or not and, if not optimal, how bad is it compared to the optimal solution. A* is the standard path finding algorithm of the video game industry. As the data shows A* is quick, complete, and optimal. Now here in this research quick is a relative term. A* is quick compared to searches like uninformed searches, but compared to most of the other searches we tested it is not quick. This being said A* is still an easy to implement search that will always work and find a path if there is one. Since A* has been used for so many years it is pointless to evaluate it. Instead we will evaluate each search against A* and against the other searches. So we can compare the algorithms against A* we will first show the data from A* below:

A*					
Level Name	Polygon Count	Time (s)	Nodes Visited	Max Nodes In Queue	Path Cost (units)
PlainScene	1	6.00E-06	1	1	0
PlainScenewithCube	4	1.85E-05	4	2	20.01665
PlainSceneWithtwoCubes2	25	6.11E-05	19	5	29.2007
TwoPlainScene	32	7.42E-05	19	8	48.336
MultiPlaneSceneWithCubes	64	0.000455725	59	17	63.57973
MultipPlaneSceneWithRandomCubes	151	0.001749542	111	29	90.30271
Xanadu	1473	0.091969133	1433	39	3437.157
HugePlaneWithRandomCubes1	3068	0.157871528	1807	86	642.2347
TryAnotherHugeWithDegrees	4202	0.652782071	3922	139	1483.118
Xanadu2	5903	0.496834003	2459	688	31376.67
TryAnotherHuge	8173	2.817510368	7957	234	4004.154

The first evaluation we will make is of Bi-Directional search. This search is slightly faster than A* for every level we built. This is without using threading. If threading was an option, the Bi-Directional search would have not have been slightly faster but incredibly faster because of the parallel processing. The speed up from A* comes from the number of nodes visited by the search. The more nodes that the search has to visit the slower it becomes. For every level but one Bi-Directional visits less nodes than A*. It is only slightly less than A* except for the level Xanadu2 in which Bi-Directional visited less than 40% of the nodes as A*. Xanadu2 is a huge but straight and narrow level. The agent is at one end of the level while the goal is at the other end of the level. Because Bi-Directional uses two searches and goes until they meet somewhere in the middle, avoiding the last half of their respective searches, it expands fewer nodes. This is due to the fact that A* visits more nodes as it gets nearer to the goal and is searching for the best path, but when using Bi-Directional the last half of the search is essentially bypassed. The max size of the queue is how many nodes are held in the open list at one time. For Bi-Directional it holds more nodes than A* at one time because it has two separate A* searches with two separate open lists. On most environments Bi-Directional has the same path cost as A*. For the environments in which Bi-Directional's path cost is different than A* it is always higher. This means that Bi-Directional is not as optimal as A*. The amount that it adds to the path cost is so small that you wouldn't be able to tell if it wasn't optimal unless you did a brute force search and carefully compared all the path costs to one another. To finish for Bi-Directional, it is a complete algorithm because it uses basic A* for both of its searches. The data below is Bi-Directional's for the environments:

Bi-Directional					
Level Name	Polygon Count	Time (s)	Nodes Visited	Max Nodes In Queue	Path Cost (units)
PlainScene	1	4.35162E-06	1	0	0
PlainScenewithCube	4	1.21927E-05	4	2	20.01665
PlainSceneWithtwoCubes2	25	4.72918E-05	16	10	29.2007
TwoPlainScene	32	9.91254E-05	19	13	48.336
MultiPlaneSceneWithCubes	64	0.000212059	33	25	63.57973
MultipPlaneSceneWithRandomCubes	151	0.000750937	70	41	90.30271
Xanadu	1473	0.071714423	1415	30	3445.417
HugePlaneWithRandomCubes1	3068	0.148339511	1920	172	643.4597
TryAnotherHugeWithDegrees	4202	0.153098145	1995	191	1483.118
Xanadu2	5903	0.061081362	920	412	31376.67
TryAnotherHuge	8173	1.360682614	6159	307	4005.499



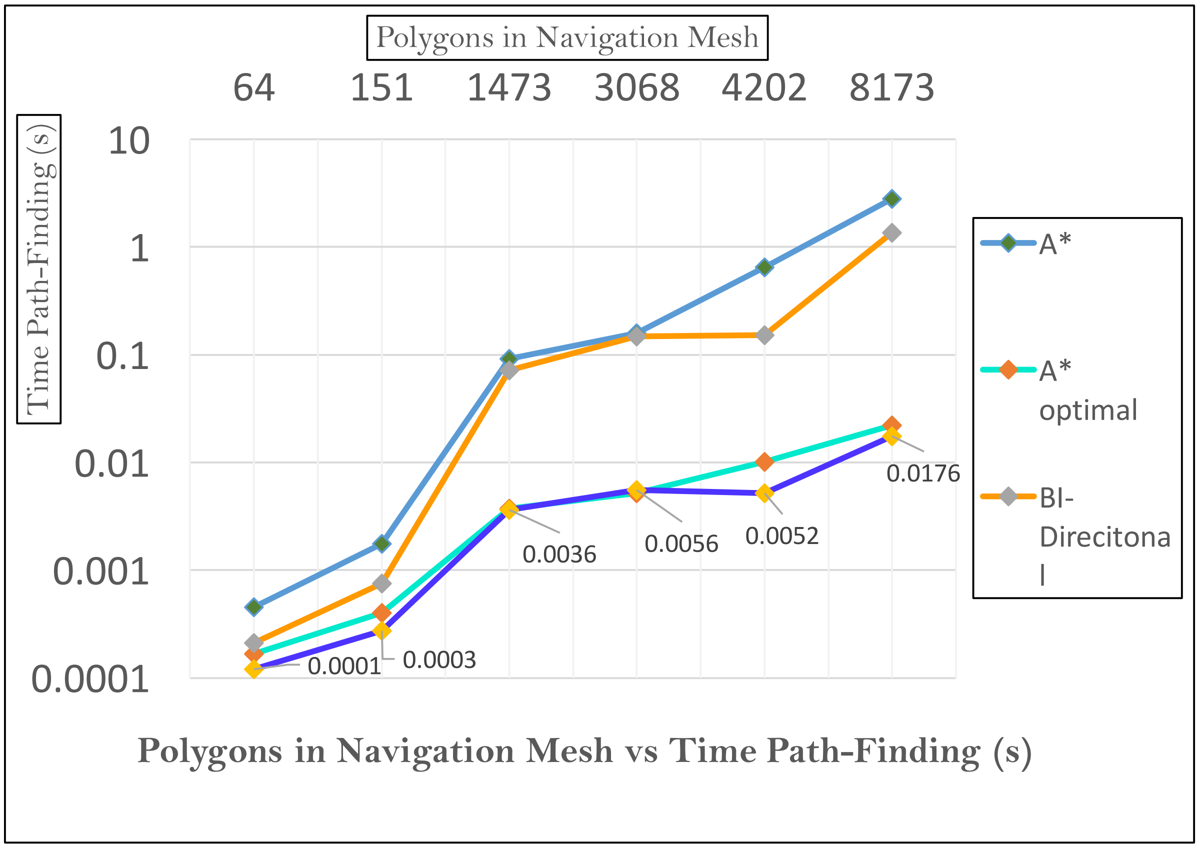
Fringe search is by far the worst of all the searches for time complexity, nodes visited, and max size of the queue. Since Fringe search uses IDA* it has to research nodes over and over again until it expands enough nodes for the goal to be in the set it can look at. For the time complexity of Fringe compared to A*, Fringe search stays only a little higher than A* when the number of searchable nodes are small, but once the searchable node count gets higher Fringe search's time complexity explodes. This is extremely apparent on the last environment where Fringe search took over 344 seconds. As stated before this is because it has to keep expanding the nodes it has already expanded because of the nature of IDA*. The max size the queue is also greater than A*. This says that it takes more memory to run Fringe search. Just like the time complexity, the space complexity starts off not much higher than A*, but as the searchable node count becomes greater the space complexity rockets upwards. The path cost is either the same as A* or greater. This shows that Fringe search is not optimal. The most interesting aspect of Fringe search is that it can be faster than A* or Bi-Directional on straight and narrow levels. Xanadu2 that was talked about earlier is a prime example of such a level. Because of the nature of the the Xanadu2 and IDA*, Fringe search expands straight down until it hits the goal making it fast. Below is the data corresponding to Fringe search:

Fringe					
Level Name	Polygon Count	Time (s)	Nodes Visited	Max Nodes In Queue	Path Cost (units)
PlainScene	1	7.66397E-06	1	0	0
PlainScenewithCube	4	1.56193E-05	7	2	20.01665
PlainSceneWithtwoCubes2	25	7.3904E-05	44	5	29.2007
TwoPlainScene	32	0.000130017	41	8	48.336
MultiPlaneSceneWithCubes	64	0.000254081	74	18	66.07505
MultipPlaneSceneWithRandomCubes	151	0.000829333	201	52	100.5537
Xanadu	1473	0.106760867	19506	648	3474.831
HugePlaneWithRandomCubes1	3068	0.227698225	14128	1065	700.7254
TryAnotherHugeWithDegrees	4202	4.509115664	95462	4521	1483.118
Xanadu2	5903	0.03554867	2948	685	32732.87
TryAnotherHuge	8173	344.1079	972765	32068	4050.381

The next evaluation is of A*'s and Bi-Directional's optimal versions. Here the optimal versions of A* and BI-Directional are always faster than their original counterparts. They visit the same amount of nodes or slightly more than their non-optimal versions, but because of the three simple optimizations we did to the code the search algorithms performed roughly 100 times faster than their non-optimized counterparts. When you consider how many times paths are constructed while playing a game this is a huge factor to consider when evaluating your search algorithms. For both the Bi-Directional and A* optimal versions, the max size of the queue and the path costs are the same or slightly more than the non-optimized versions. Just like the non-optimized version comparisons, Bi-Directional optimized performs better, time wise, than A* optimal. Below is the data for both Bi-Directional optimal and A* optimal:

A* optimal					
Level Name	Polygon Count	Time (s)	Nodes Visited	Max Nodes In Queue	Path Cost (units)
PlainScene	1	1.25875E-05	1	1	0
PlainScenewithCube	4	1.49674E-05	4	2	20.01665
PlainSceneWithtwoCubes2	25	4.33235E-05	19	5	29.2007
TwoPlainScene	32	7.11746E-05	19	8	48.336
MultiPlaneSceneWithCubes	64	0.000166576	59	17	63.57973
MultipPlaneSceneWithRandomCubes	151	0.00040083	111	29	90.30271
Xanadu	1473	0.003723265	1450	39	3437.157
HugePlaneWithRandomCubes1	3068	0.005179307	1810	86	642.2347
TryAnotherHugeWithDegrees	4202	0.010066425	3937	139	1483.118
Xanadu2	5903	0.014454339	2458	688	31376.67
TryAnotherHuge	8173	0.02206523	7965	234	4004.154

Bi-Directional optimal					
Level Name	Polygon Count	Time (s)	Nodes Visited	Max Nodes In Queue	Path Cost (units)
PlainScene	1	4.09508E-06	1	0	0
PlainScenewithCube	4	8.15678E-06	4	2	20.01665
PlainSceneWithtwoCubes2	25	3.28608E-05	16	10	29.2007
TwoPlainScene	32	6.88863E-05	21	11	48.336
MultiPlaneSceneWithCubes	64	0.000120726	33	25	63.57973
MultipPlaneSceneWithRandomCubes	151	0.000272712	70	41	90.30271
Xanadu	1473	0.003606461	1461	25	3437.157
HugePlaneWithRandomCubes1	3068	0.005564553	1922	172	643.4597
TryAnotherHugeWithDegrees	4202	0.005192899	1999	191	1483.118
Xanadu2	5903	0.003437756	920	412	31376.67
TryAnotherHuge	8173	0.017555074	6164	307	4005.499



Beam search is based off of space complexity considerations. Because beam search is A*, but with a cap on the max size of the queue, it performs the same as A* in most aspects. The beam search is using the optimized version of A* and was compared to A* optimal. The time complexity is around the same time as A* optimal. The big difference between A* optimal and Beam search is when the max size of the queue becomes huge for A* optimal. The max size of the queue we used for Beam search was 230 nodes. A* optimal had a max size of the queue at 638. This helps space complexity greatly if the max queue size stays lower. The problem with putting a maximum cap on the queue is that the search is no longer optimal and is also not complete. The search is not optimal because it might throw away certain nodes that it needed to make the optimal path. The search is not complete because of the max queue size cap. If the max queue size cap is too small then it will throw out nodes that it needed to complete the search. This situation calls for empirical testing for every environment to make sure that the search can find a path. The designer that is using Beam search needs to run the search on all the environments to make sure the cap they put on the queue will allow for a path to be found every time. If a path cannot be found the max queue size cap needs to be increased. Below is the data for Beam search on every environment developed:

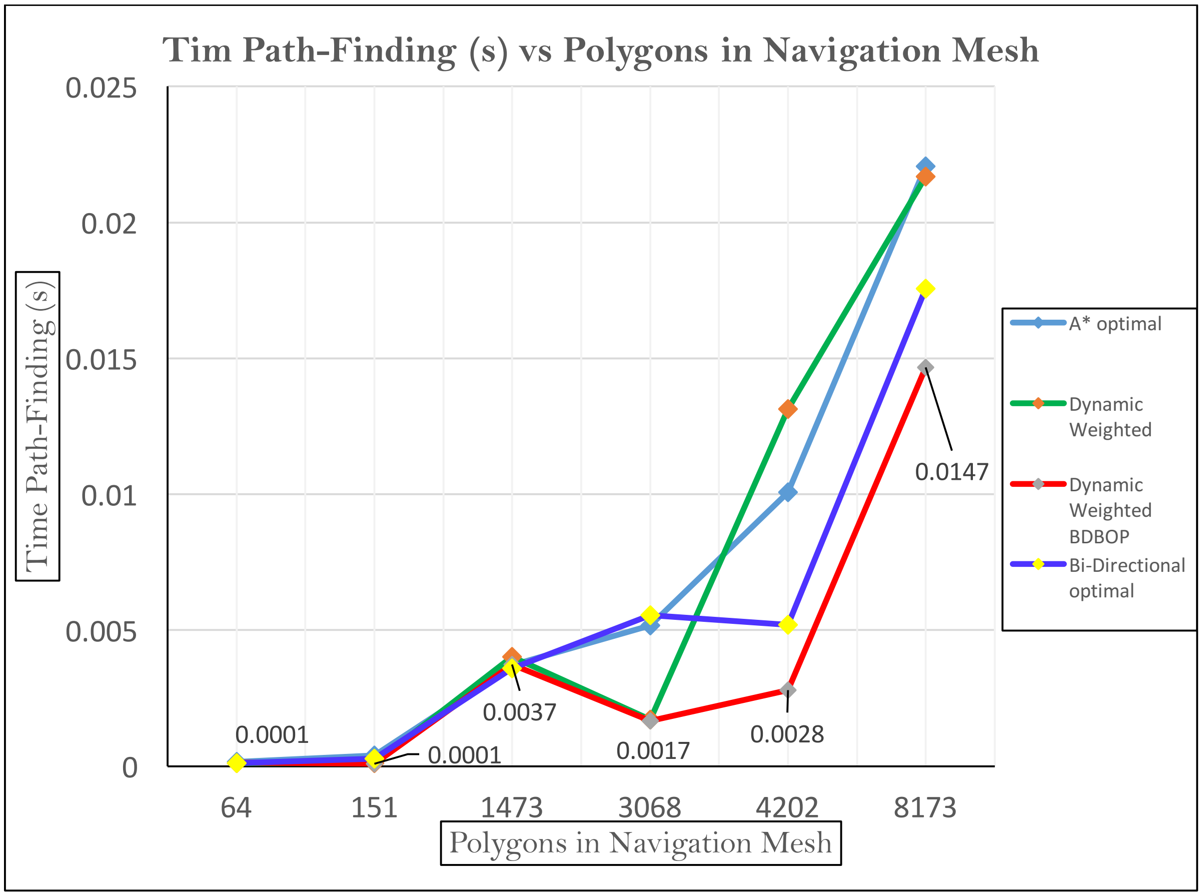
Beam					
Level Name	Polygon Count	Time (s)	Nodes Visited	Max Nodes In Queue	Path Cost (units)
PlainScene	1	1.1605E-05	1	1	0
PlainScenewithCube	4	1.32608E-05	4	2	20.01665
PlainSceneWithtwoCubes2	25	3.98817E-05	19	5	29.2007
TwoPlainScene	32	6.72641E-05	19	8	48.336
MultiPlaneSceneWithCubes	64	0.000178422	59	17	63.57973
MultipPlaneSceneWithRandomCubes	151	0.000389614	111	29	90.30271
Xanadu	1473	0.003887091	1450	39	3437.157
HugePlaneWithRandomCubes1	3068	0.005209413	1810	86	642.2347
TryAnotherHugeWithDegrees	4202	0.011437898	3937	139	1483.118
Xanadu2	5903	0.007759266	1671	230	31376.67
TryAnotherHuge	8173	0.022468253	7960	230	4004.154

Dynamic Weighted A* search is a major time complexity improvement over A* optimal. Because of Dynamic Weighted A*'s attribute of wanting to get the search going with the weighted heuristic, the search runs faster than A* optimal. The search visits less nodes than A* optimal except for one environment where it visited the same as A* optimal. This is the main reason for Dynamic Weighted A* being faster than A* optimal. Dynamic Weighted A* does not perform faster than A* optimal on every environment and this is because the size of the queue is bigger than A* optimal for every environment except Xanadu2. The queue size being bigger slows it down because it has to make an operation of O($\log_2 n$) for insertion and deletion and even though $\log_2 n$ is a small operation it still makes a difference when the queue is bigger. The path cost for Dynamic Weighted A* is going to be greater than A* optimal. This tells us that Dynamic Weighted A* is not an optimal search, but it is complete because it does not throw out any nodes it might use. Below is the data for Dynamic Weighted A* for the environments that were developed:

Dynamic Weighted A*					
Level Name	Polygon Count	Time (s)	Nodes Visited	Max Nodes In Queue	Path Cost (units)
PlainScene	1	1.14284E-05	1	1	0
PlainScenewithCube	4	1.36528E-05	3	2	20.01665
PlainSceneWithtwoCubes2	25	4.20008E-05	19	7	29.2007
TwoPlainScene	32	5.99418E-05	12	11	55.96035
MultiPlaneSceneWithCubes	64	0.000114689	23	19	63.57973
MultipPlaneSceneWithRandomCubes	151	0.000108925	15	32	103.7901
Xanadu	1473	0.004015337	1241	131	3760.751
HugePlaneWithRandomCubes1	3068	0.001712589	394	134	681.0273
TryAnotherHugeWithDegrees	4202	0.013145534	3833	224	1483.118
Xanadu2	5903	0.006726547	97	316	31495.76
TryAnotherHuge	8173	0.021681417	6423	329	5312.438

Dynamic Weighted BDBOP is a search that was created to have a better time complexity than A* and not give up the space complexity like Dynamic Weighted A*. For every level but one Dynamic Weighted BDBOP out performs for time complexity A*. The environment in which it does not it was behind by only .00002 seconds. The maps where it was faster than A* optimal it was faster on some by .0034 seconds and higher. The attributes from both Dynamic Weighted A* and Bi-Directional contribute the most to the time complexity speed up. Dynamic Weighted gives the search the head start and the idea about not caring about the path being optimal at the beginning of the search. This put together with Bi-Directional gives the search a double dose of head start to allow it to get going extremely fast and get to a point where it is almost done just from the head start. Bi-Directional also visited less nodes than A* which helps speed up Dynamic Weighted BDBOP. The space complexity is saved because of the queue capping aspect of Beam search. For Dynamic Weighted BDBOP the max queue size cap could be lower than Beam search which we placed at 100 for each of the two search agents used. This made a total of 200 in which only one environment was able to get the queue size to that limit. The only down fall with Dynamic Weighted BDBOP is the path cost. The path cost is greater than A*, not by much in some environments but at the most at over 1000 units. The level in which the path cost difference was over 1000 was a massive level where a higher cost might not matter much to the developers. This does mean that Dynamic Weighted BDBOP is not optimal and because of the addition of the max queue size cap Dynamic Weighted BDBOP is not complete. It is complete more often with smaller queue limits than Beam search, but it is still not complete all the time. Below is the data for Dynamic Weighted BDBOP for all developed environments:

Dynamic Weighted BDBOP					
Level Name	Polygon Count	Time (s)	Nodes Visited	Max Nodes In Queue	Path Cost (units)
PlainScene	1	3.64923E-06	1	0	0
PlainScenewithCube	4	8.75807E-06	3	2	20.01665
PlainSceneWithtwoCubes2	25	3.12166E-05	15	9	29.2007
TwoPlainScene	32	6.89998E-05	17	18	58.10884
MultiPlaneSceneWithCubes	64	0.000143726	23	19	63.57973
MultipPlaneSceneWithRandomCubes	151	9.57684E-05	15	32	103.7901
Xanadu	1473	0.003744285	1405	59	3862.1
HugePlaneWithRandomCubes1	3068	0.001677637	397	122	676.8464
TryAnotherHugeWithDegrees	4202	0.002796114	824	168	1483.118
Xanadu2	5903	0.001222544	97	100	31495.76
TryAnotherHuge	8173	0.014675078	4564	200	5809.746



The last thing to talk about is the idea of an optimal solution. For all searches except for A* an optimal solution was not found every time. Is this a bad thing? Not in every case. The need for an optimal solution depends on why the search is necessary. If the path is the path for GPS than an optimal solution always needs to be found. But in a video game an optimal solution does not always need to be found. The main concern in a videogame is time complexity, space complexity, and completeness. In most video games today space complexity is becoming less and less necessary so that concerns can be thrown out the window. Also if a navigation mesh is used the search space will never be so big that the space the search takes up will be an issue. So in a game where a navigation mesh is used and the environment is not the size of the world the two biggest concerns is time complexity and completeness. With Dynamic Weighted BDBOP the time complexity is not an issue. It is faster than all the other searches by a great deal, which will help the game perform at a faster rate. With completeness just like Beam search it can be an issue. But because space complexity for the searches is no longer an issue we can make the cap large like 100 for each search agent or take it out completely. If the developer wants to keep the space complexity in the search, then they must set a cap that will work for their environments and keep their space complexity. It can be argued that humans do not always choose the path that is the shortest, most optimal. Since video games now are trying to have the artificial intelligence agents, or NPCs, act more humanly, to make the player experience better, the path from the search not being optimal is not a bad thing. The path Dynamic Weighted BDBOP finds is close enough to the optimal solution that it is still not a bad path to take, but since it is not optimal it allows the agent to look more humanly on its path.