IAJ Project 2 - Pathfinding

This report has been made in accordance to the guidelines provided for the IAJ's (Inteligência Artificial para Jogos) second project.

Authors

- Diogo Silva (98776)
- Carlos Marques (98639)
- Rui Melo (98823)

Index

A*

A* Algorithm

Starting off with the base A* algorithm, we basically implemented it using as basis the code given by the professor for Lab 3. As such several support methods were already implemented.

We did, however, alter the **GetNeighbourList** method both to simplify, and attempt to improve its effieciency. Originally this method presented a plethora of *if* conditions and convoluted code that, despite working, could also be achieved with a single nested and a couple inline conditions:

```
 int startX = currentNode.x == 0 ? 0 : -1;
int endX = currentNode.x == grid.getWidth() ? 0 : 1;
int startY = currentNode.y == 0 ? 0 : -1;
int endY = currentNode.y == grid.getHeight() ? 0 : 1;
for (int xx = startX; xx <= endX; xx++){
    for(int yy = startY; yy <= endY; yy++){
            if(xx == 0 && yy == 0){
                        continue; // Dont add yourself as a neighbour
        }
                neighbourList.Add(GetNode(currentNode.x+xx, currentNode.y+yy));
    }
}
```

As for the **Search** itself, it was implemented basically as presented in the theoretical slides, with the inclusion of the option to retrieve the *partial path*, and the processing of **15** nodes per frame. Other additions include the computation of debug and analysis values such as the max open list size.

For the closed list we utilized the **ClosedDictionary**, as specified in Lab 4, and for the open list we utilized the (rather unoptimal) **SimpleUnorderedList**, as it wasn't requested in any lab or guideline that we utilized any other more appropriate data structure (like a PriorityHeap). This could, however, be easily changed as the PriorityHeap is implemented and ready to be swapped into A* if needed.

Euclidean Heuristic

Initially we implemented the Euclidean Heuristic by simply computing the Euclidean Distance as such:

```
 float x = (goalNode.x - node.x);
float y = (goalNode.y - node.y);

return Mathf.Sqrt(x * x + y * y);
```

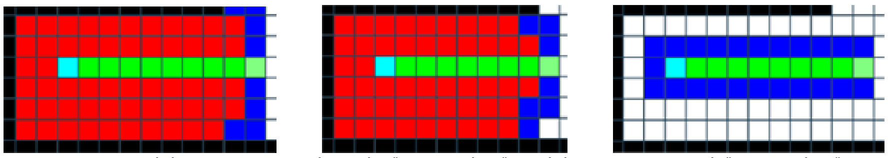
This made it so the heuristic was admissable, since it would always output a value smaller than the actual cost of going from *node* to *goalNode*. There was a problem however. The given *x* and *y* coordinates passed to this heuristic were the coordinates post **quantization** (i.e, they're integer values ranging from 0 to width or height). This is valid, and again, makes the heuristic admissable, but it doesn't make it all too optimal due to the fact that for our pathfinding algorithms we normalize the costs of both straight and diagonal movements to a static **10** for straight jumps and **14** for diagonals (which is helpful since our grid cells have a varying size depending on how big of a map we're using).

To improve this heuristic we altered it by multiplying the linear distance between the x and y coordinates by the cost of moving in a straight direction, as such:

```
 float x = (goalNode.x - node.x) * MOVE_STRAIGHT_COST;
float y = (goalNode.y - node.y) * MOVE_STRAIGHT_COST;

return Mathf.RoundToInt(Mathf.Sqrt(x * x + y * y));
```

This will output an Euclidian Heuristic that is "up to scale" with our defined cost values. Furthermore it is still admissable since it will always be smaller or equal to the actual cost of going from *node* to *goalNode*, and more optimal than the prior algorithm since the values are going to be more inline with the actual costs of reaching *goalNode* from *node* (with the movement costs being as defined). Note also that we do a *RoundToInt* before returning due to the fact that our defined diagonal cost movement is an integer number (14). As such, if we used the non-rounded value our heuristic would be over-estimating by a few decimals the actual cost, hence making it non-admissable.


From left to right, the fill using ZeroHeuristic, Euclidean Heuristic without the "cost scaling", Euclidean Heuristic with "cost scaling".

Tie Breaking

Our final improvement to the base A* was making it so, in occasion that two nodes produce the same *fValue*, we pick the one with the lowest *hValue* (i.e, the one that is, according to our heuristic, closer to the goal node). We did this, simply by adding a new method to our open data structures (more specifically, in the case of A*, to our *SimpleUnorderedNodeList*). Initially we had already been provided with a *PeekBest* method that used a LINQ query to get the node with the lowest *fValue*.

What we did was create a new method **PeekBestTieBreaking**, that gets the node with the lowest *fValue*, and in case of tie, gets the node with the lowest *fValue* AND lowest *hValue* instead. This was accomplished by adding some more conditions to the LINQ query, as such:

```
NodeRecord best = this.NodeRecords.Aggregate((nodeRecord1, nodeRecord2) => (nodeRecord1.fCost < nodeRecord2.fCost ? nodeRecord1 :
(nodeRecord1.fCost > nodeRecord2.fCost ? nodeRecord2 : (nodeRecord1.hCost < nodeRecord2.hCost ? nodeRecord1 : nodeRecord2))));
```

NodeArray A*

Node Array and Map Preprocessing

Before implementing the NodeArray A* we had to first implement the **NodeRecordArray** data structure itself. The class "skeleton" had already been provided to us for Lab 4, but we still had to implement all of the actual methods.

Firstly we started by adding a new variable to our **NodeRecord** class - *NodeIndex* - aswell as a static variable - *next_index* - which is initialized at 0 and increments every time a new node is created (meaning each node's index is automatically attributed upon creation).

The actual instantiation of the **NodeRecordArray** is handled by the **NodeArrayAStarPathfinding** class which posesses a method - *MapPreprocessing* - used to register and add all nodes to its NodeRecordArray

```
 List<NodeRecord> nodes = new List<NodeRecord>();
for (int x = 0; x < grid.getWidth(); x++)
    for (int y = 0; y < grid.getHeight(); y++)
        {
        //Add records to list
                NodeRecord node = GetNode(x, y);
                nodes.Add(node);
        }
this.NodeRecords = new NodeRecordArray(nodes);
```

NodeArray A* Algorithm

The NodeArray A* Algorithm was implemented with efficiency in mind, both in terms of avoiding unecessary searches and processing, but also in terms of memory - since all of our nodes have already been created and are stored in our NodeRecordArray, creating additional nodes would be a waste.

This class was implemented as a subclass of the **AStarPathfinding** class, which served as a super class for both this and for our **JPSPathfinding**.

The most notable difference between the **NodeArrayAStarPathfinding** and it's super class lies in the *ProcessChildNode* method. More specifically, in the fact that instead of iterating through our open and closed lists to check if the current neighbour node we're checking is in any of these, we simply get the node from our NodeRecordArray and check it's status (which is either set to Open, Closed or Unvisited).

```
 var node = this.NodeRecords.GetNodeRecord(neighbourNode);
if (node.status == NodeStatus.Open){
    // Child is in open
    if (node.fCost > fCost){
        (...)
    }
    return;
}else if (node.status == NodeStatus.Closed){
    // Child is in closed
    if (node.fCost > fCost){
        (...)
    }
    return;
}
```

It should also be noted that inherently this algorithm also does a type of tie breaking, in the sense that the Open list, implemented using a Priority Heap, adds nodes to the list, firstly ranked by *fCost* and then by *hCost* in the case of ties. This happens because the *CompareTo* function in the **NodeRecord** class was altered to have this in mind.

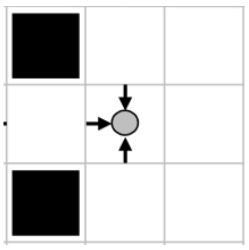
JPS+

Map Preprocessing

JPS+ gets most of its efficiency during runtime from all of the preprocessing that has to be done to the map. More specifically, we need to **generate the primary jump points**, and **compute the distances** from each node to each jump point through sweeps.

Before this, however, we had to add two new variables to our NodeRecords - **directions** and **distances**.

The first is a boolean array with size 4 in which each index corresponds to the cardinal directions North, South, East, West (in this order). The index is set to True if this node is a primary jump point entering from the given direction. For example, in the image below, that specified Jump Point node would have it's **directions** array set to [True, True, False, True].



The latter, **distances**, is a String-Int Dictionary that maps each possible direction (North, South, East, West, NorthEast, SouthEast, NorthWest, SouthWest) to an integer value corresponding to node's distance to a jump point (or wall). Note that this could have also been implemented with just an integer array rather than a dictionary, but for the sake of preserving our sanities when debugging and implementing the JPS+ runtime algorithm, we chose to have it be a dictionary instead (and would've done the same for the **directions** variable aswel, if not for hubris).

The **directions** variable (i.e, marking nodes as jump points) is done on the *checkJumpPoint* method, meanwhile the **distances** are set on our *sweepLeftRight* (sweeps left to right and right to left), *sweepUpDown* (sweeps down to up and up to down), *sweepUpDiagonally* (sweeps down to up right and left) and *sweepDownDiagonally* (sweeps up to down right and left). All of these methods are called in the **JPSPlusPathfinding** class method *MapPreprocessing*.

All of these algorithms were basically implemented as specified in the GameAIPro2 book.

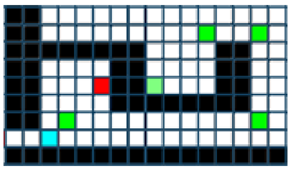
It should also be stated that our JPS+ implementation was done using the NodeRecordArray for both the Open and Closed sets.

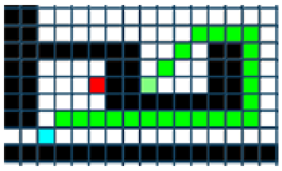
JPS+ Algorithm

After doing all the preprocessing, the JPS+ algorithm was implemented as specified in the GameAIPro2 book (with adaptations so that it would work with the NodeRecordArray and our other classe's structures). Several support functions had to be created such as *inGeneralDirection* and *inExactDirection* and so on. It should also be stated that, for Open and Closed lists we used our **NodeRecordArray** and, just like the NodeRecordArrayAStarPathfinding class, the **JPSPlusPathfinding** class also inherits from AStarPathfinding.

For our algorithm to work properly we had to add yet another variable to our node records - **travelingDirection**. Since JPS+ introduces the notion of removing redundant paths by considering which directions to move on depending on the direction the node was entered. As such, this variable holds the direction that the current node's parent was traveling to reach the current node (the exception to this is the starting node which has no traveling direction, since it has no nodes that traveled to it).

The **travelingDirection** variable is also important for our *CalculatePath* method which had to be altered for this class (I'd also like to remember that this class is in fact a subclass of AStarPathfinding). A property of JPS+ is that it only considers Jump Point Nodes (or Target Jump Points), and as such, when the algorithm finds the path, it will have just the disconnected nodes, as seen in the left image. With the travelingDirection, however, we can post-process our path to include all the intermediate nodes and achieve an output as specified in the image on the right.

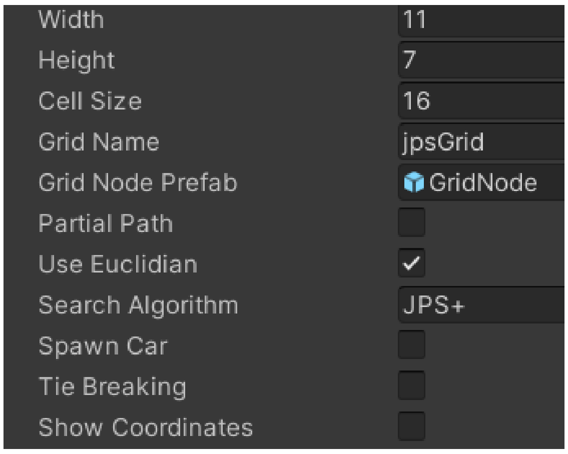




Several other adaptations were made for the algorithm to work properly (such as using the aforementioned DIAGONAL_JUMP_COST and STRAIGHT_JUMP_COST to compute the givenCost) that are too extensive to include in this report.

Settings

Before proceeding further, we should explain the available settings and variables of our Pathfinding Manager

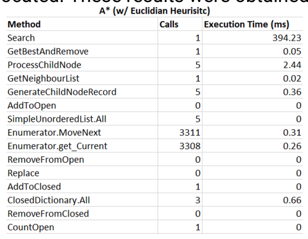


Width	11
Height	7
Cell Size	16
Grid Name	jpsGrid
Grid Node Prefab	GridNode
Partial Path	☐
Use Euclidian	✓
Search Algorithm	JPS+
Spawn Car	☐
Tie Breaking	☐
Show Coordinates	☐

- **Partial Path**
  - Check to make it so the partial path is returned
- **Use Euclidian**
  - If checked the algorithms will use the Euclidean Heuristic. If its not, then they will be using the Zero Heuristic
- **Search Algorithm**
  - Used to select which pathfinding algorithm should be used.
  - Valid values are - A*, NodeArrayA* or JPS+.
- **Spawn Car**
  - If checked, when the search algorithm finishes a car will be spawned and start following the created path
- **Tie Breaking**
  - If checked, A* will use Tie Breaking via heuristic value checking
- **Use Priority Movement**
  - If checked, the car's movement will include priority movement and dynamic avoid obsticles, else, it just uses Arrive Movement.

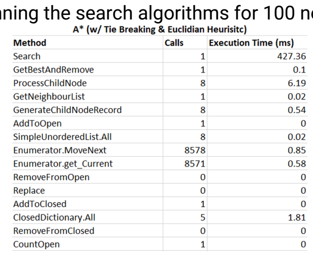
Comparison and Analysis

After all of our algorithms had been implemented and properly tested we proceeded to analyze their behaviours and compare them against each other. Bellow follow two pictures. The first has some results obtained from running Unity's Deep Profiler + Call Stack in order to analyze which functions each algorithm's search function was calling and where the time was being allocated. These results were obtained running the search algorithms for 100 nodes on each frame.



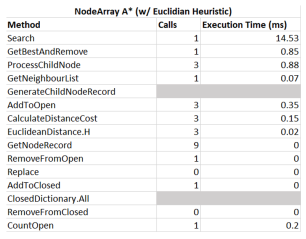
A* (w/ Euclidian Heurisitc)

Method	Calls	Execution Time (ms)
Search	1	394.23
GetBestAndRemove	1	0.05
ProcessChildNode	5	2.44
GetNeighbourList	1	0.02
GenerateChildNodeRecord	5	0.36
AddToOpen	0	0
SimpleUnorderedList.All	5	0
Enumerator.MoveNext	3311	0.31
Enumerator.get_Current	3308	0.26
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	1	0
ClosedDictionary.All	3	0.66
RemoveFromClosed	0	0
CountOpen	1	0



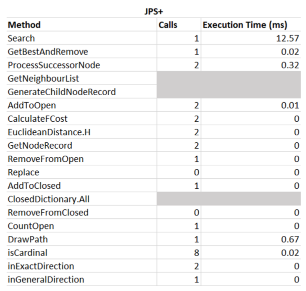
A* (w/ Tie Breaking & Euclidian Heurisitc)

Method	Calls	Execution Time (ms)
Search	1	427.36
GetBestAndRemove	1	0.1
ProcessChildNode	8	6.19
GetNeighbourList	1	0.02
GenerateChildNodeRecord	8	0.54
AddToOpen	1	0
SimpleUnorderedList.All	8	0.02
Enumerator.MoveNext	8578	0.85
Enumerator.get_Current	8571	0.58
RemoveFromOpen	0	0
Replace	0	0
AddToClosed	1	0
ClosedDictionary.All	5	1.81
RemoveFromClosed	0	0
CountOpen	1	0



NodeArray A* (w/ Euclidian Heuristic)

Method	Calls	Execution Time (ms)
Search	1	14.53
GetBestAndRemove	1	0.85
ProcessChildNode	3	0.88
GetNeighbourList	1	0.07
GenerateChildNodeRecord		
AddToOpen	3	0.35
CalculateDistanceCost	3	0.15
EuclideanDistance.H	3	0.02
GetNodeRecord	9	0
RemoveFromOpen	1	0
Replace	0	0
AddToClosed	1	0
ClosedDictionary.All		
RemoveFromClosed	0	0
CountOpen	1	0.2

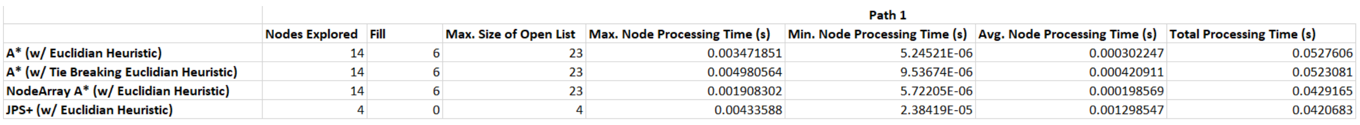


JPS+

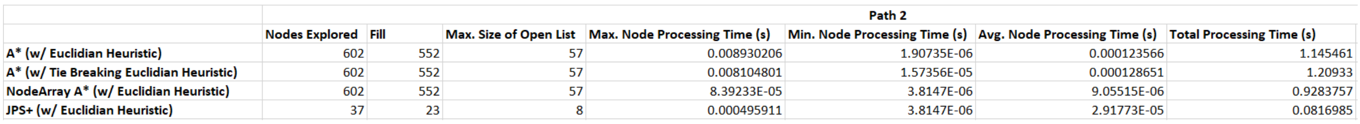
Method	Calls	Execution Time (ms)
Search	1	12.57
GetBestAndRemove	1	0.02
ProcessSuccessorNode	2	0.32
GetNeighbourList		
GenerateChildNodeRecord		
AddToOpen	2	0.01
CalculateFCost	2	0
EuclideanDistance.H	2	0
GetNodeRecord	2	0
RemoveFromOpen	1	0
Replace	0	0
AddToClosed	1	0
ClosedDictionary.All		
RemoveFromClosed	0	0
CountOpen	1	0
DrawPath	1	0.67
isCardinal	8	0.02
inExactDirection	2	0
inGeneralDirection	1	0

Some conclusions could be taken, such as comparing the execution time of certain methods between algorithms and realizing that some algorithms call methods that others do and vice versa (for example the NodeArray A* algorithm doesn't bother generating child node records, since it bases itself around not having to generate more nodes than the ones created during pre processing). We find that this isn't the optimal way to compare our algorithms however, as we had to randomly pick a frame from the profiler to analyze for each of the algorithms (aiming to never pick neither the one that peaked or valleyed in terms of CPU usage).

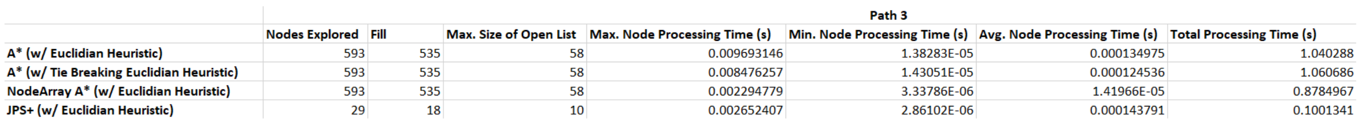
Following are results obtained by running each algorithm (maximum 25 nodes per frame) on the same paths and recording their debug values. More specifically we recorded the total number of nodes explored, the fill (by subtracting the number of nodes explored minus the nodes that were in the actual path), the maximum size of the open list, the total processing time until the search was complete, and the minimum, maximum and average time it took to process a node. All these paths were done on the provided giantGrid (100x70 nodes)



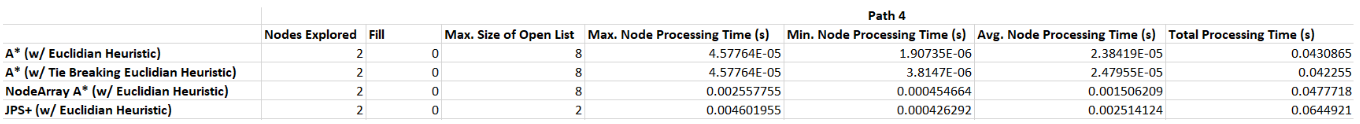
	Path 1						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	14	6	23	0.003471851	5.24521E-06	0.000302247	0.0527606
A* (w/ Tie Breaking Euclidian Heuristic)	14	6	23	0.004980564	9.53674E-06	0.000420911	0.0523081
NodeArray A* (w/ Euclidian Heuristic)	14	6	23	0.001908302	5.72205E-06	0.000198569	0.0429165
JPS+ (w/ Euclidian Heuristic)	4	0	4	0.00433588	2.38419E-05	0.001298547	0.0420683



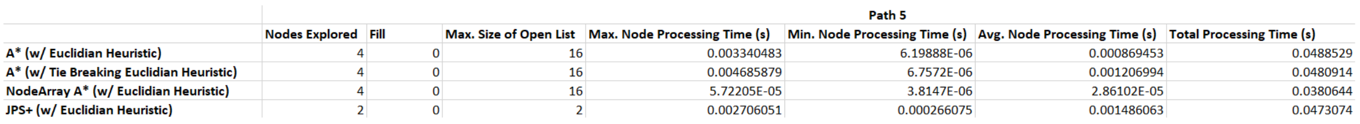
	Path 2						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	602	552	57	0.008930206	1.90735E-06	0.000123566	1.145461
A* (w/ Tie Breaking Euclidian Heuristic)	602	552	57	0.008104801	1.57356E-05	0.000128651	1.20933
NodeArray A* (w/ Euclidian Heuristic)	602	552	57	8.39233E-05	3.8147E-06	9.05515E-06	0.9283757
JPS+ (w/ Euclidian Heuristic)	37	23	8	0.000495911	3.8147E-06	2.91773E-05	0.0816985



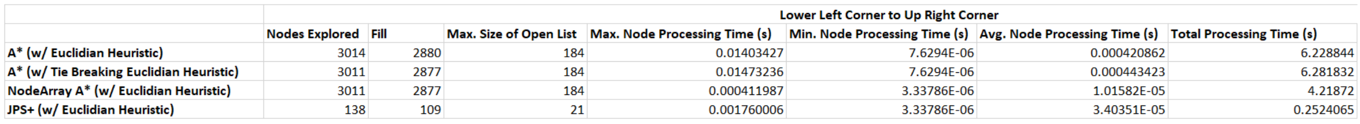
	Path 3						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	593	535	58	0.009693146	1.38283E-05	0.000134975	1.040288
A* (w/ Tie Breaking Euclidian Heuristic)	593	535	58	0.008476257	1.43051E-05	0.000124536	1.060686
NodeArray A* (w/ Euclidian Heuristic)	593	535	58	0.002294779	3.33786E-06	1.41966E-05	0.8784967
JPS+ (w/ Euclidian Heuristic)	29	18	10	0.002652407	2.86102E-06	0.000143791	0.1001341



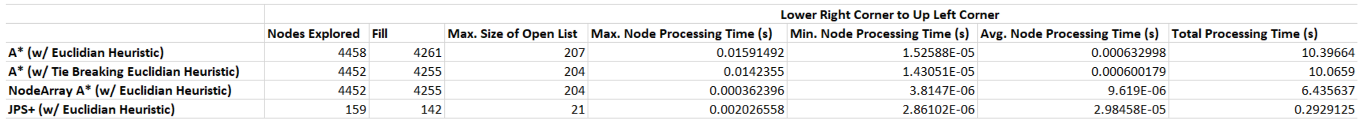
	Path 4						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	2	0	8	4.57764E-05	1.90735E-06	2.38419E-05	0.0430865
A* (w/ Tie Breaking Euclidian Heuristic)	2	0	8	4.57764E-05	3.8147E-06	2.47955E-05	0.042255
NodeArray A* (w/ Euclidian Heuristic)	2	0	8	0.002557755	0.000454664	0.001506209	0.0477718
JPS+ (w/ Euclidian Heuristic)	2	0	2	0.004601955	0.000426292	0.002514124	0.0644921



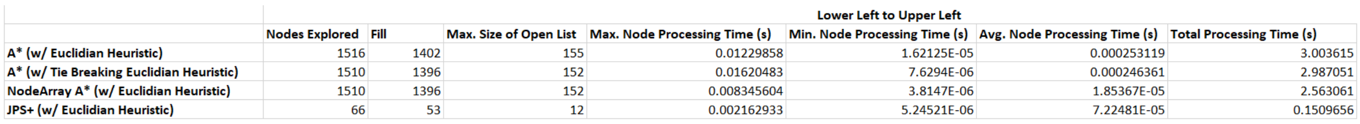
	Path 5						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	4	0	16	0.003340483	6.19888E-06	0.000869453	0.0488529
A* (w/ Tie Breaking Euclidian Heuristic)	4	0	16	0.004685879	6.7572E-06	0.001206994	0.0480914
NodeArray A* (w/ Euclidian Heuristic)	4	0	16	5.72205E-05	3.8147E-06	2.86102E-05	0.0380644
JPS+ (w/ Euclidian Heuristic)	2	0	2	0.002706051	0.000266075	0.001486063	0.0473074



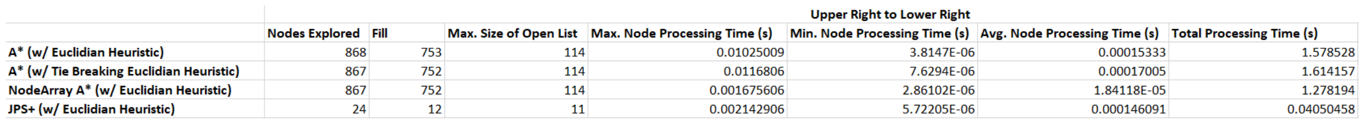
	Lower Left Corner to Up Right Corner						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	3014	2880	184	0.01403427	7.6294E-06	0.000420862	6.228844
A* (w/ Tie Breaking Euclidian Heuristic)	3011	2877	184	0.01473236	7.6294E-06	0.000443423	6.281832
NodeArray A* (w/ Euclidian Heuristic)	3011	2877	184	0.000411987	3.33786E-06	1.01582E-05	4.21872
JPS+ (w/ Euclidian Heuristic)	138	109	21	0.001760006	3.33786E-06	3.40351E-05	0.2524065



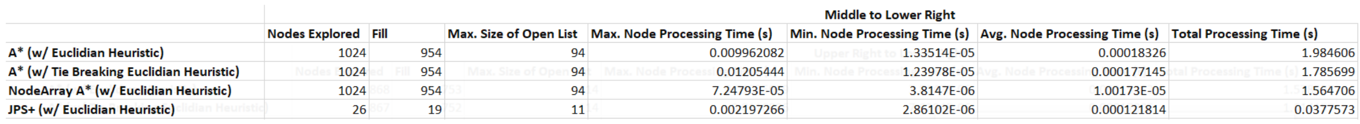
	Lower Right Corner to Up Left Corner						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	4458	4261	207	0.01591492	1.52588E-05	0.000632998	10.39664
A* (w/ Tie Breaking Euclidian Heuristic)	4452	4255	204	0.0142355	1.43051E-05	0.000600179	10.0659
NodeArray A* (w/ Euclidian Heuristic)	4452	4255	204	0.000362396	3.8147E-06	9.619E-06	6.435637
JPS+ (w/ Euclidian Heuristic)	159	142	21	0.002026558	2.86102E-06	2.98458E-05	0.2929125



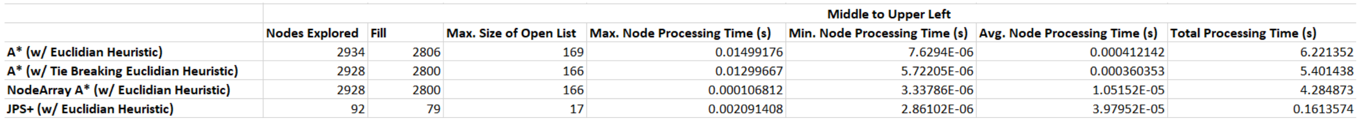
	Lower Left to Upper Left						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	1516	1402	155	0.01229858	1.62125E-05	0.000253119	3.003615
A* (w/ Tie Breaking Euclidian Heuristic)	1510	1396	152	0.01620483	7.6294E-06	0.000246361	2.987051
NodeArray A* (w/ Euclidian Heuristic)	1510	1396	152	0.008345604	3.8147E-06	1.85367E-05	2.563061
JPS+ (w/ Euclidian Heuristic)	66	53	12	0.002162933	5.24521E-06	7.22481E-05	0.1509656



	Upper Right to Lower Right						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	868	753	114	0.01025009	3.8147E-06	0.00015333	1.578528
A* (w/ Tie Breaking Euclidian Heuristic)	867	752	114	0.0116806	7.6294E-06	0.00017005	1.614157
NodeArray A* (w/ Euclidian Heuristic)	867	752	114	0.001675606	2.86102E-06	1.84118E-05	1.278194
JPS+ (w/ Euclidian Heuristic)	24	12	11	0.002142906	5.72205E-06	0.000146091	0.04050458



	Middle to Lower Right						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	1024	954	94	0.009962082	1.33514E-05	0.00018326	1.984606
A* (w/ Tie Breaking Euclidian Heuristic)	1024	954	94	0.01205444	1.23978E-05	0.000177145	1.785699
NodeArray A* (w/ Euclidian Heuristic)	1024	954	94	7.24793E-05	3.8147E-06	1.00173E-05	1.564706
JPS+ (w/ Euclidian Heuristic)	26	19	11	0.002197266	2.86102E-06	0.000121814	0.0377573



	Middle to Upper Left						
	Nodes Explored	Fill	Max. Size of Open List	Max. Node Processing Time (s)	Min. Node Processing Time (s)	Avg. Node Processing Time (s)	Total Processing Time (s)
A* (w/ Euclidian Heuristic)	2934	2806	169	0.01499176	7.6294E-06	0.000412142	6.221352
A* (w/ Tie Breaking Euclidian Heuristic)	2928	2800	166	0.01299667	5.72205E-06	0.000360353	5.401438
NodeArray A* (w/ Euclidian Heuristic)	2928	2800	166	0.000106812	3.33786E-06	1.05152E-05	4.284873
JPS+ (w/ Euclidian Heuristic)	92	79	17	0.002091408	2.86102E-06	3.97952E-05	0.1613574

As we can easily see, JPS+ absolutely demolishes all other algorithms in every front. It obtains processing times that are much smaller comparatively (except for paths that consisted of 2-8 nodes, at which point we could confidently say that the difference in processing time is due to margin of error). In terms of nodes explored, fill and max size of open list, no matter the path, JPS+ always obtained better results (i.e smaller ones).

This however, was to be expected. **JPS+** is able to perform much faster at runtime due to all the preprocessing it does on the map. Only jump points are explored and added to the open list, unlike the other algorithms that indiscriminately allow for the exploration of any node (conditionally, obviously). We don't even need to generate or compute neighbours since all we need to do to check where we can go next is check the direction we entered the current node from and its distances variable.

All other results obtained were, aswell, consistently expectable. **A*** was the node that performed the worst out of all the search algorithms, which wasn't surprising, as this was served as a base class for all other nodes and posessed no "tricks" like the JPS+ or "optimizations" like the NodeArrayA*.

A* with tie breaking performed very similarly to A* in most occasions, but it did manage to shave off a few nodes out of both fill, total nodes explored, and sometimes even max nodes in the open list. In terms of time taken, however, there wasn't a large enough margin that would lead us to conclude A* with tie breaking was significantly better than A*. At least not the way we implemented it. It should be mentioned the way we did Tie Breaking was very simple. Perhaps another strategy such as slightly augmenting artificially the value of the hCost could produce better results, but then again, it could also make our heuristic non-admissable.

NodeArrayA* managed to obtain the same fill, total nodes explored and max nodes in open list values as A* with Tie Breaking. This was expected as, all in all, we're not really altering the way or order we process nodes (remember NodeArrayA* also inherently does tie breaking due to the way it stores the nodes in the PriorityHeap). What this algorithm does, however, is make it so we don't have to constantly create new nodes and, perhaps more importantly, makes it so we don't have to iterate over an open and closed list in order to verify whether a node is or is not in the open or closed lists. The impact of this can even be seen in the first image where the ProcessChildNode method was constantly recquiring less execution time than the ProcessChildNode methods in A* and A* with Tie Breaking. This then leads to the execution times being consistently smaller in all paths with a significant enough number of nodes (taking into account that the more nodes we have to explore, the better NodeArrayA* will be over A* and A* with Tie Breaking, since it will cause more nodes to be in the closed/open lists, hence leading the latter algorithms to have to iterate over a much larger number of nodes.

All in all, no surprises were had.

As a side note, all of the created tables have also been exported to a csv file - **P2_Tables.csv** - thats included in the project.

PathFollowing

Car Spawning

If the **Spawn Car** option is checked in the PathfindingManager then upon a search's completion one out of four possible cars will be spawned in the node in which the search started. It will immediately start following the computed path and upon reaching the goal node, stop. When clearing the grid, or after a new path is computed, the car will de-spawn and a new one will be generated. Our cars are controlled by our **MovementController** class which receives, not only the spawn location, but also an array with the real-world coordinates of each node in the path (except for the starting one, i.e, the one where the car should spawn and therefore, already is).

Movement

Our cars move using the **Dynamic Arrive** movement from the previous project.

One might easily realize that the Arrive movement won't really generate the most realistic of movements. And one would be right. If we simply set the car's target to be the next node in the path, what we'll witness is that the car accelerates briefly, and then immediately starts deaccelerating, since in our grid all of our path's nodes are next to each other. The car will basically be moving in "hiccups".

Initially, we solved this by also including the **Dynamic Seek** movement and setting our cars to move with this behaviour for each node except for the last one, at which point we swapped to Arrive. This, however, caused a lot of crashes courtesy of paths that included sharp (or even non-sharp...) turns. We then removed this movement and instead implemented **Straight Line Smoothing** which ended up solving our "hiccups" problem. There were still, however, situations in which after a sharp turn the car would skim against a wall, or pass over an obstacle.

It should also be stated that we added code into the **Pathfinding Manager** class in order to generate a Box Collider around the "wall" nodes. This was done initially for the StraightLine smoothing optimization, but also ended up being useful for the optional Priority Movement.

StraightLine Smoothing

The straight line smoothing was done as specified in the theoretical slides. We start at a node and check each node after it and the one after that and see if we can move directly from our current one to the farther, if so, we delete the middle one. We repeat this until we find a node we cannot remove, at which point we set our current node to be that one, and continue the algorithm. The way the deletion is done is by adding the nodes to be removed to a temporary array, and then removing the nodes in that array from the nodes in the movement's path. We couldn't easily remove the nodes directly from the path during the iterations of the algorithm since this could lead to problems (as we'd be altering the array we're iterating over). It should also be stated that we check whether or not we can move directly to a node by generating several intermediate points between the current node and the node we're checking, and at each point checking for any collisions using a central ray and two smaller whiskers.

This algorithm was implemented in our **cullPath** method, and managed to effectively remove the unecessary nodes from our path.

Behaviour Optimizations

For tweaking the values (which were adjusted to work optimally with the Giant Grid), we also had the idea to include a "dynamic" adjustment of the Arrive movement's slow radius that took into consideration the position of the target we just arrived at to the next target. This was based on the idea that, the farther the distance, the more we're going to be accelerating, henceforth, the faster we should start slowing down. `this.arriveMovement.SlowRadius = 6.5f + 0.015f * totalDistance;`

Besides this we also set the Arrive movements' *stop* radius to 0.0, since we don't actually want to fully stop, just slowdown between targets. We do, however, set the stop radius to 0 when dealing with our final target.

Other optimizations were attempted to avoid certain collisions, but in the end we ended up adding the option to activate **Priority Movement**, which uses both the Arrive Movement and a **Dynamic Avoid Obstacle** algorithm to attempt to avoid crashing into obstacles.