João Silva 98737, Rafael Henriques 89530, Tiago Jacinto 87127

Level 1 - A*

For the A* algorithm we tested five different implementations of the algorithm: 1-Without tie breaking and unordered list for both sets; 2-Without tie breaking and with closed set dictionary; 3-With tie breaking and with closed set dictionary; 4- Without tie breaking and with closed set dictionary and Open Priority Heap; 5- With tie breaking and with closed set dictionary and Open Priority Heap;

On implementation 1 of A* we explored a total of 2756 nodes , had an open list with a maximum size of 184 nodes and a total processing time of 4,746s, on 2 we had the same explored nodes and open list but with a processing time of 3,556s. .From 1 to 2 we added a dictionary for the closed set, which decreased the execution time of the SearchInClosed method by approximately 10 times. The complexity of the A* algorithm is, in the worst case, O(b^d), in which the *b* represents the branching factor and *d* represents the depth of the solution. Regarding the A* with dictionary, because our dictionary uses a hash table, its complexity is O(1). This happens, because the complexity of accessing a dictionary is not related with the size of the hash table, but rather with the size of the key. Since we decided to use a sole integer as key for the dictionary, our complexity is O(1) thus making the algorithm much faster than O(b^d).

	1-Without tie breaking & unordered list for both sets		2-Without tie breaking & closed set dictionary	
Method	**Calls**	**Execution time** (ms)	**Calls**	**Execution time** (ms)
A*Pathfinding.Search	1	214,18	1	24,42
GetBestAndRemove	50	4,62	50	3,95
AddToOpen	41	0,03	51	0,02
SearchInOpen	337	16,08	367	15,80
RemoveFromOpen(not the method but removing from open set through remove)	50	0,54	50	0,39
Replace	43	0,00	49	0,00
AddToClosed	50	0,03	50	0,04
SearchInClosed	337	187,62	367	0,32
RemoveFromClosed	0	0,00	0	0,00

For the changes between 2 and 3 we added tie breaking. We tested two different ways to preform tie breaking. First, we used a small *p* to give a different weight to the heuristic, but we the algorithm didn´t improve much. Then we tried simply choosing the node with the smaller heuristic, and once again, the improvement wasn´t much either. Since the difference is very small, we are not presenting it in the tables.

Implementation 4 had the same nodes but a total processing time of 3,379s. For implementation 4, we added a Priority Heap in the open set and took out tie breaking. Doing this reduced significantly the execution time of the GetBestAndRemove() method. This happened because the complexity of accessing the best node of a priority heap is O(1), making the algorithm faster in the GetBestAndRemove() method.

Finally, for the last change, we added to the previous implementation tie breaking. The total processing time was 3,401s. As expected, similarly to the third implementation, the results did not change much. The table with the data for these 2 implementations is shown below

	4-Without tie breaking & closed set dictionary & open PriorityHeap		5-With tie breaking & closed set dictionary & open PriorityHeap	
Method	**Calls**	**Execution time** (ms)	**Calls**	**Execution time (ms)**
A*Pathfinding.Search	1	19,08	1	19,23
GetBestAndRemove	50	0,64	50	0,69
AddToOpen	44	0,08	66	0,26
SearchInOpen	349	13,59	339	12,95
RemoveFromOpen(not the method but removing from open set through remove)	50	0,63	50	0,68
Replace	31	0,47	45	0,72
AddToClosed	50	0,04	50	0,17
SearchInClosed	349	0,3	339	0,29
RemoveFromClosed	0	0,00	0	0,00

Level 2 - Node Array A*

For the 6th performance test, we used the NodeArray A*. As we can see in the table, the method cut a significant amount of execution time from the SearchInOpen method as well as some time from the SearchinClosed. This is happening because the algorithm is accessing the nodes on the node array through an index that is stored on each node (given by height of grid*x + y), making the searches O(1) since the nodes are accessed instantly. The total processing time was 3,37s.

6 - NodeArray A* with tie breaking and openPriorityHeap		
Method	**Calls**	**Execution time** (ms)
A*Pathfinding.Search	1	5,32
GetBestAndRemove	50	0,64
AddToOpen	56	0,26
SearchInOpen	324	0,06
RemoveFromOpen(not the method but dequeue from heap)	50	0,62
Replace	40	0,65
AddToClosed	50	0,01
SearchInClosed	324	0,05
RemoveFromClosed	0	0,00

Level 3 – JPS+

We implemented JPS+ using a priority heap for the open list and a closed dictionary for the closed list. Since it wasn't explicit in the book or the slides, the way we did the primary jump point was by searching the grid for obstacle cells, if the obstacle cell has a walkable cell in the diagonal, and the two cardinal directions touching that diagonal are also walkable, then that diagonal is a primary jump point.

7 - JPS+		
Method	**Calls**	**Execution time** (ms)
A*Pathfinding.Search	1	2,31
GetBestAndRemove	50	0,18
AddToOpen	60	0,14
SearchInOpen	70	0,75
RemoveFromOpen(not the method but dequeue from heap)	50	0,17
Replace	1	0,00
AddToClosed	50	0,07
SearchInClosed	70	0,04
RemoveFromClosed	0	0,00

The first thing to note when seeing this table is that although the execution time between JPS and NodeArray A* is only half when looking at a single frame, JPS only runs for two frames (the second frame it processes another 34 nodes).The total processing time was 0,03s, with the total Explored nodes being 83 and the maximum size of the open list being 14.

The method "GetBestAndRemove" takes less time since the Priority Heap is much shorter (since we only store jump points). The add functions have almost the same execution time, the difference in time in the "AddToOpen" function may be due to the lists' size being smaller. The execution times of the Search functions take much longer when considering the number of calls, but since the list of nodes to search through is also smaller the difference in time is not too significant, since it is called less times throughout the algorithm and searches a smaller list.

JPS avoids redundant paths, so it is rare to have a need for replacements, which makes the Replace function rarely used. The remove from closed function will never be used since nodes are not reopened in this algorithm. After this analysis we can conclude that JPS+ is the fastest since it processes the least nodes, and each node is processed faster.

Level 5 - PathFollowing

The implementation of Pathfollowing was achieved in two parts, the DynamicFollow algorithm and the Pathsmoothing.

<u>DynamicFollow:</u>

The two main methods that need to be implemented in DynamicFollow are from the Path class, GetParam and GetPosition. We imagined the path as line segments, which are formed from two nodes of the path. **GetParam** finds the line segment closest to the current position of the character. To achieve this, we calculated the angles formed by the character and the

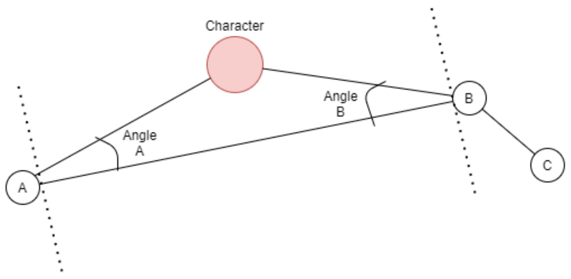

Figure 1

end points (A, B) of a line segment as seen in figure 1. If these angles are below 90 degrees that means the character will be somewhere within the bounds marked by stripes.

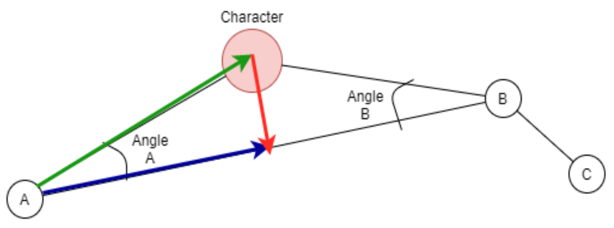

Figure 2

Then we project the vector from A to Character into vector AB which results in the blue vector represented in figure 2. Finally, we can get the closest point as a result of the sum between A and the projection vector. The distance from the character to this point is then calculated and the line segment that will be chosen is the one where this distance is the smallest. We considered that the param that has to be returned is the index of the node that represents the starting point of the line segment within the path list plus the magnitude of the blue vector (projection) divided by the magnitude of the vector AB. This means that while the character is within a line segment his param will be between the index of the start of that line segment and its end.

GetPosition can now be easily implemented by simply truncating the param value to an integer which will result in the index of a node which is immediately returned.

__**Pathsmoothing**__**:**

The path found by A* or Node Array A* contains many unnecessary nodes which cause the path following to be slower. Our solution removes not only the unnecessary nodes but also changes some bits of the path so that there are less collisions with the obstacles. To remove the unnecessary nodes, we go through each node that forms the rectangle (outlined with red stripes) between two target nodes, in this example 0 and 2 from figure 3. And it searches for nodes that are not walkable. If the algorithm does not find any not walkable nodes, then it removes the node in between. In figure 3 node 1 was not removed since the obstacle is within the rectangle. Finally, to improve collision avoidance in turns we also substitute the diagonal nodes and add a single node in the corners, for example in figure 3 we add a node below 1 and remove 1 and 2. Otherwise the character would arrive/seek from 1 to 2 and overlap the obstacle.

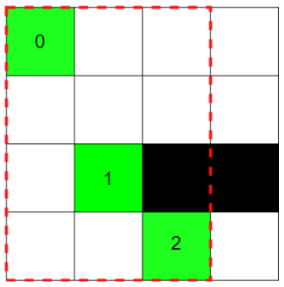

Figure 3

Level 6 – Optimizations

The first optimization we did was to change the Euclidean distance heuristic into an octile one (where our movement restrictions are counted into the heuristic) but there were no changes to the processing time. The Total explored nodes went up by 100 in A* and by 6 in JPS+, but the maximum size of the open list went down to 174 on A* and stayed the same on JPS+. We concluded that although the heuristic is more accurate, it doesn't seem to make much difference.