Inteligência Artificial Para Jogos

2º Projeto: Efficient pathfinding in Room based scenarios and Path Following

Grupo: 12

Nome: Daniel Amado - Número: 75629

Nome: Gonçalo Castilho - Número: 75305

Nome: Tiago Gomes - Número: 76079

Introdução

Este projeto foi realizado no âmbito da cadeira de Inteligência Artificial para Jogos e encontra-se divido em duas partes essenciais, sendo a primeira parte a criação de um algoritmo eficiente de procura de caminhos e de seguida implementar o movimento de maneira a que personagem siga o caminho mais eficiente até ao ponto marcado no mapa. Inicialmente foi implementado os vários algoritmos pedidos, sendo estes o A* e o Node Array A*. Após a realização destes foi desenvolvida a heurística Gateway. De seguida foi implementado então o movimento do personagem para garantir que seguisse o caminho correto até ao objetivo. Por fim, mas não menos importante foram efetuadas diferentes otimizações de maneira a melhorar a eficiência do projeto.

*Algoritmo A**

O projeto além dos objetivos iniciais tinha diferentes níveis para seguir uma ordem estruturada. Esse primeiro nível era implementar o algoritmo A*.

Este algoritmo calcula o caminho a escolher através de pesos. Estes pesos consistem na soma entre uma função que dá o custo de um caminho entre dois nós e uma função heurística que faz uma estimativa do custo de um dado nó até há solução. Só no caso de a heurística ser admissível é que é garantido a melhor solução.

No caso deste projeto já tínhamos um ponto de partida para efetuá-lo sendo que já nos eram dadas as estruturas e alguns métodos auxiliares para implementar esta procura.

*Algoritmo Node Array A**

A implementação deste algoritmo tem como base o Algoritmo A* sendo que contem uma diferença importante no que toca as listas de nós abertos e fechados. Ao contrário do algoritmo A* que continha duas listas, que podiam ser de tipos diferentes, este algoritmo contem apenas uma classe que tem uma lista com todos os nós, e em que cada nó pode ser acedido usando um identificador único e tambem uma lista de nós abertos. Neste algoritmo cada nó que esta guardado na primeira lista contem uma variavel que corresponde ao seu estado que pode ser Aberto, Fechado ou Não visitado. Usando o estado de cada nó podemos evitar operações desnecessárias para adicionar nós a lista de abertos assim como que os nós fechados apenas é necessário modificar o estado do nó em vez de adicionar a uma lista.

Todo o resto do algoritmo funciona de igual forma ao A* original.

Heurística Gateway

Após a implementação das procuras dedicámo-nos a implementação das heurísticas que iriam ser necessárias para as compararmos e percebermos quais melhor se adaptam ao cenário fornecido pelo professor.

O primeiro passo passou por implementar o *Cluster Graph,* sendo que este seria necessário para fornecer informações há heurística. Ora esta estrutura de dados estava quase

implementada tendo já os *clusters* e as *gateways*, apenas faltava introduzir os valores das distâncias entre as *gateways*, para acedermos rapidamente a estes valores que eram usados pela heurística. Para isso o que fizemos foi uma procura com o algoritmo A*, usando uma Heuristica com a distancia Euclidiana, em que nos dava a caminho com o menor custo entre duas *gateways* e inseríamos os valores.

No fim desta etapa passámos então para a implementação da heurística. Esta heurística consistia em dados dois nós, sendo que inicialmente verificamos se os nós pertencem a algum cluster, caso não pertençam ou caso os nós estejam no mesmo cluster aplicamos a distância Euclidiana, caso contrário o que fazemos é vemos a distância do nó até ás gateways e de seguida dessas gateways até ás gateways do cluster do nó objetivo e por fim fazemos novamente a distancia euclidiana dessas gateways até ao nó objetivo, como podemos ver nesta formula:

➔ h(n, g) = $minGi \in G\ Cluster\ n\ ,Gj \in G(Cluster\ g\ )$ (h'(n,Gi) + H(Gi , Gj) + h'(Gj, g))

Resultados:

Após a implementação das várias procuras e das heurísticas passamos para a parte de obtenção de resultados e comparação entre estes. Foram feitos dois tipos de procuras sendo que os nós iniciais e nós objetivos foram sempre muito próximos entre as experiências, havendo, no entanto, um erro humano devido a não ser possível carregar no mesmo ponto do mapa duas vezes com a mesma precisão.

As procuras efetuadas foram a A* com a heurística Euclidiana, associada a diferentes tipos de estruturas para armazenar e aceder aos nós abertos e fechados. Entre estas estão as estruturas para a lista de nós abertos, SimpleUnorderedList, NodePriorityHeap, LeftPriorityList, RightPriorityList. No caso da lista de nós fechados temos as listas SimpleUnorderedList e Dictionary.

Além disso foi testado a procura Node Array A* com as duas heuristicas implementadas para o projeto.

Foram testadas as diferentes combinações e pode-se visualizar os diferentes resultados na tabela 1.

Algoritmo	Heurística	Lista(open/closed)	Nodes Visitados	Máximo de Open Nodes	Processing Time(ms)	Processing Time for Node (ms)
A*	Euclidean Distance	NodePriorityHeap, Dictionary	6985	208	1279	0.1831
		LeftPriorityList, Dictionary	7034	211	2182	0.3102
		RightPriorityList, Dictionary	7050	207	2098	0.2976
		SimpleUnorderedList, SimpleUnorderedList	6985	208	36365	5.2062
NodeArray A*	Euclidean Distance		7154	210	1190	0.1663
	Gateway		1414	229	401	0.2836
	Gateway Opt.		1393	229	227	0.1630

Comparações:

Comparando inicialmente os diferentes tipos de estruturas usados na procura A*, na métrica de tempo de processamento verificámos que usar Listas desordenadas prejudica imenso a performance do caminho devido a ter que procurar na lista sempre qual é o nó com o valor mais pequeno. Entre as diferentes estruturas que temos aqui representadas a que efetuou melhores tempos foi a NodePriorityHeap, sendo que tanto a Left como a Right PriorityList fizeram quase 1000 ms a mais, além disso, no pior caso temos as listas desordenadas que demoraram 36365ms a serem percorridas, o que é uma diferença muito significativa em relação ás outras estruturas. Em termos de nós visitados nenhuma estrutura se destacou sendo que todas elas rondaram os mesmos valores, sendo a pior por uma diferença mínima a RightPriorityList. Este facto também se aplica ao número máximo de nós da lista de abertos. Estas duas métricas são muito semelhantes devido ao facto de a heurística não mudar daí que os nós que têm que explorar serão os mesmos, havendo apenas uma pequena discrepância devido ao erro humano. Esta métrica pode ser observada nas imagens abaixo, sendo que os nós azuis representam os nós explorados e os verdes que estão na lista de abertos.

Quanto ao tempo de processamento de tempo por cada nó ai podemos verificar que esta métrica é diretamente proporcional ao tempo de processamento, isto deve-se ao facto de os nós explorados serem muito semelhantes e neste caso podemos verificar que as mudanças de estruturas influenciam imenso o tempo de exploração por nó, tal como observado enquanto foi tratada a métrica do tempo de processamento.

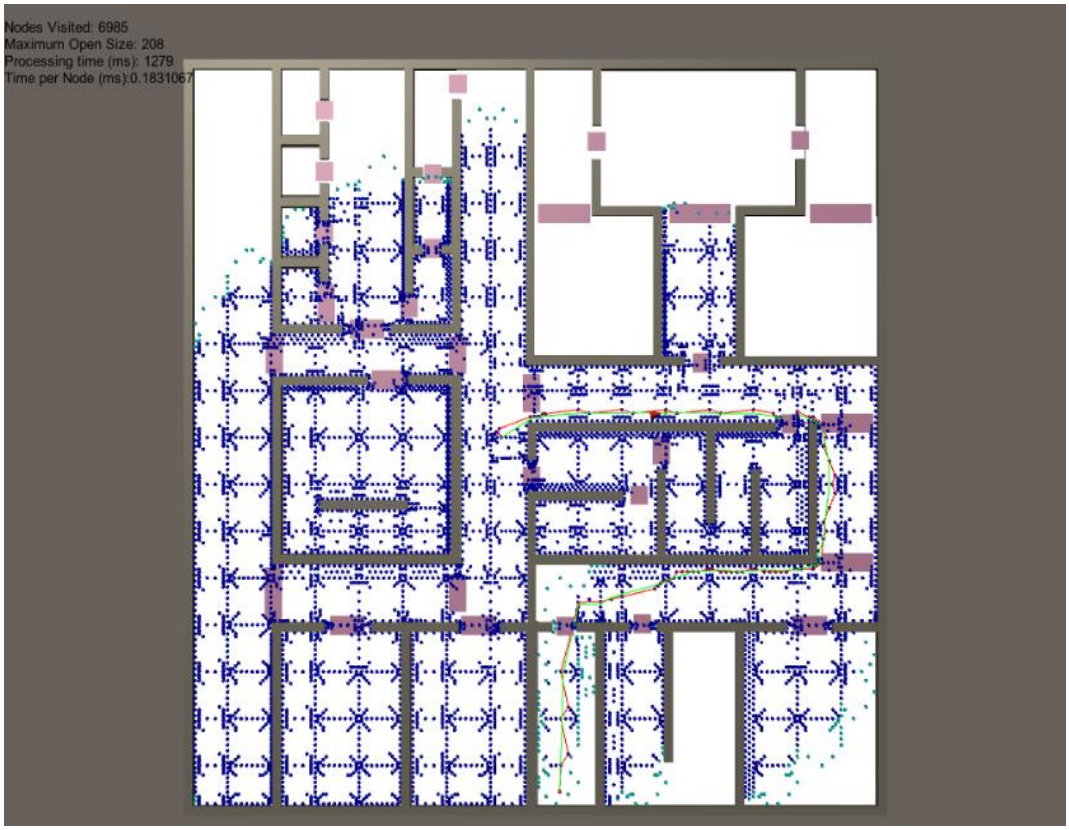
Após esta comparações entre estruturas passamos para a comparação entre heuristicas. Sendo que as duas heuristicas que vamos comparar são a gateway e a distância euclidiana.

Como podemos observar na tabela, a heurística gateway destaca-se em quase todas as métricas, tirando o tempo de processamento por nó. Esta diferença deve-se a forma como a heurística é utilizada, e devido a termos as distancias entre gateways disponíveis a partida numa tabela. Como temos acesso ás distâncias e não precisamos de calculá-las ganhamos tempo que no caso da heurística euclidiana seria usado para calcular a distancia sempre que visita um nó. Além disso devido a heurística gateway ser mais focada devido as distancias entre pontos predefinidos do mapa que são as gateway, garante que menos nós vão ser explorados e atinge o resultado mais rapidamente.

Por fim em relação aos algoritmos implementados verificamos que as procuras são muito semelhantes em termos de nós explorados, apenas destacando que no caso do NodeArrayA* conseguiu finalizar primeiro que o A*, algo que foi testado nas mesmas situações.

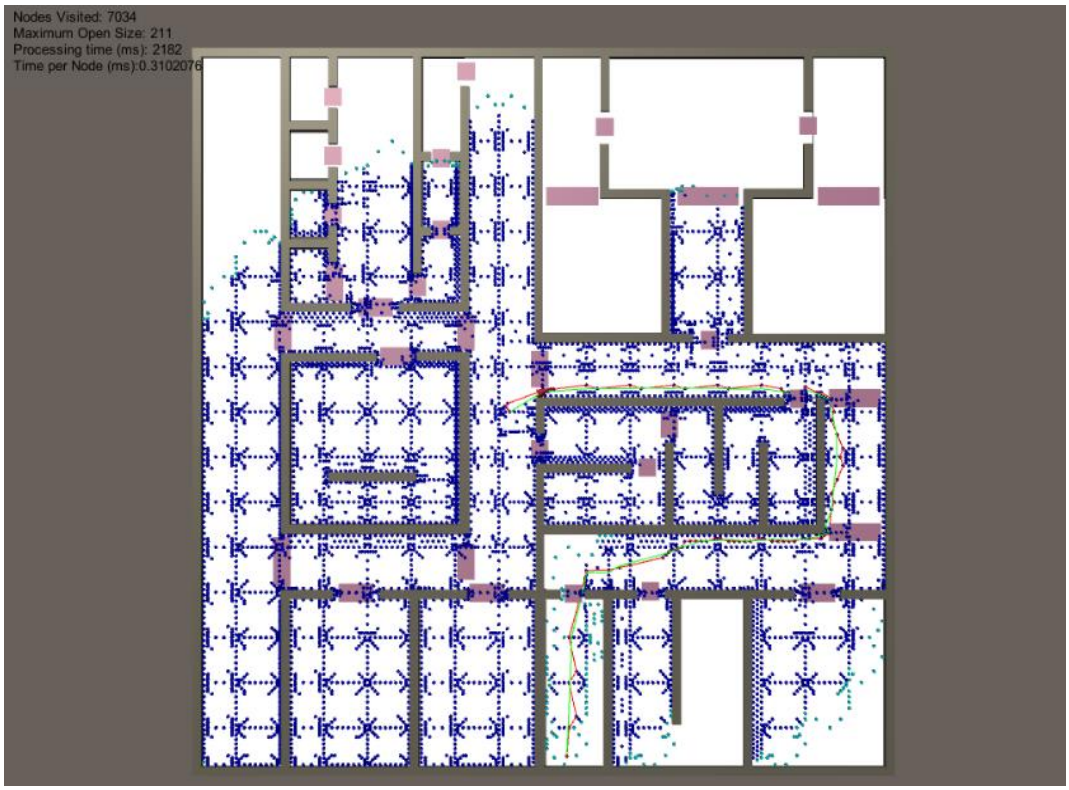
\/ A* com Euclidean Distance `(new NodePriorityHeap (), new DictionaryNodeList(), new EuclideanDistanceHeuristic())`

{



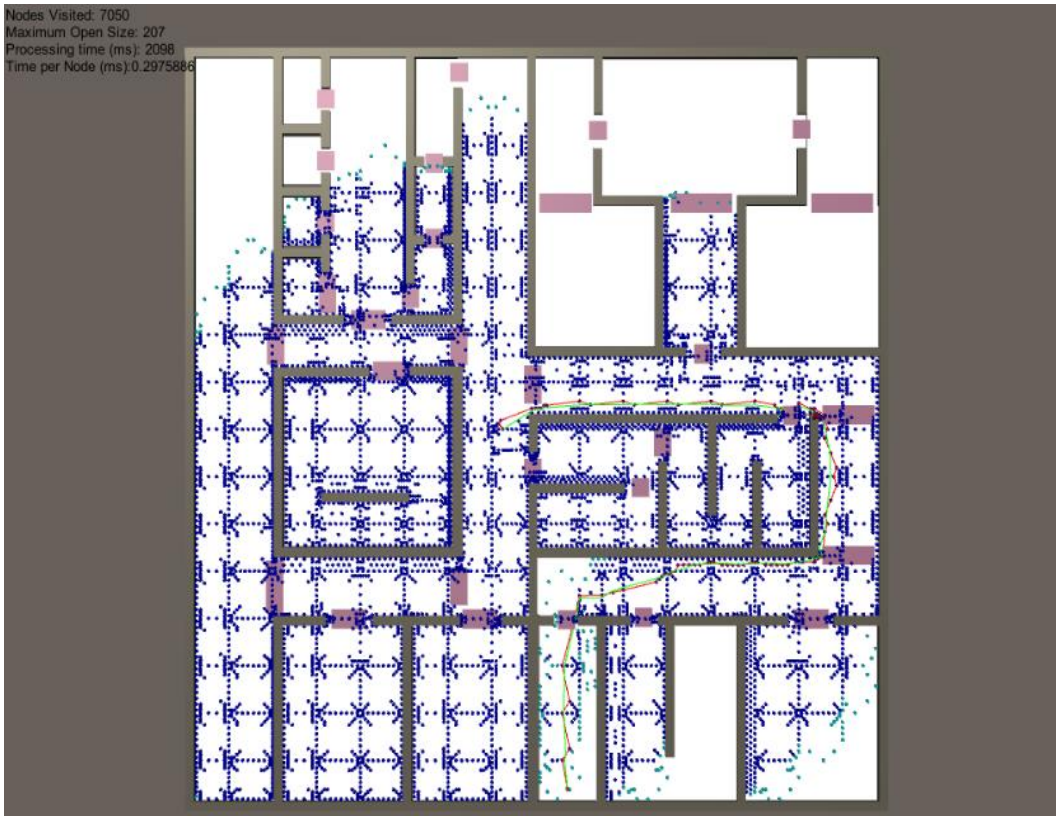
\/ A* com Euclidean Distance `(new LeftPriorityList(), new DictionaryNodeList(), new EuclideanDistanceHeuristic())`

{

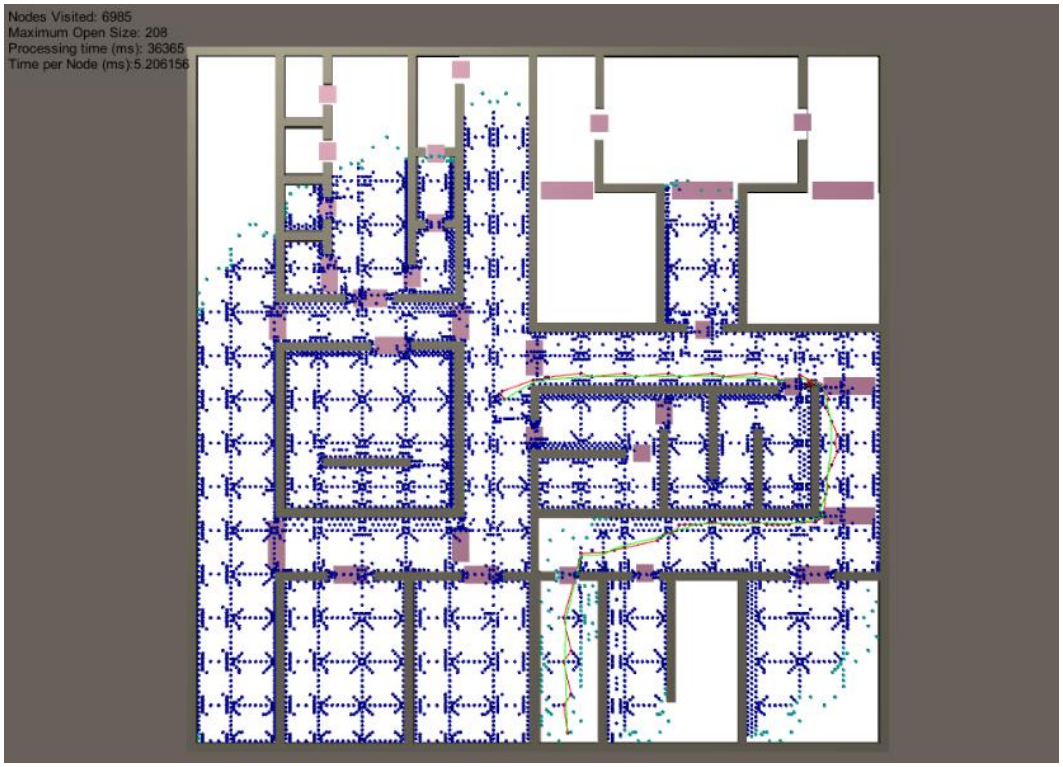


\/ A* com Euclidean Distance `(new RightPriorityList(), new DictionaryNodeList(), new EuclideanDistanceHeuristic())`

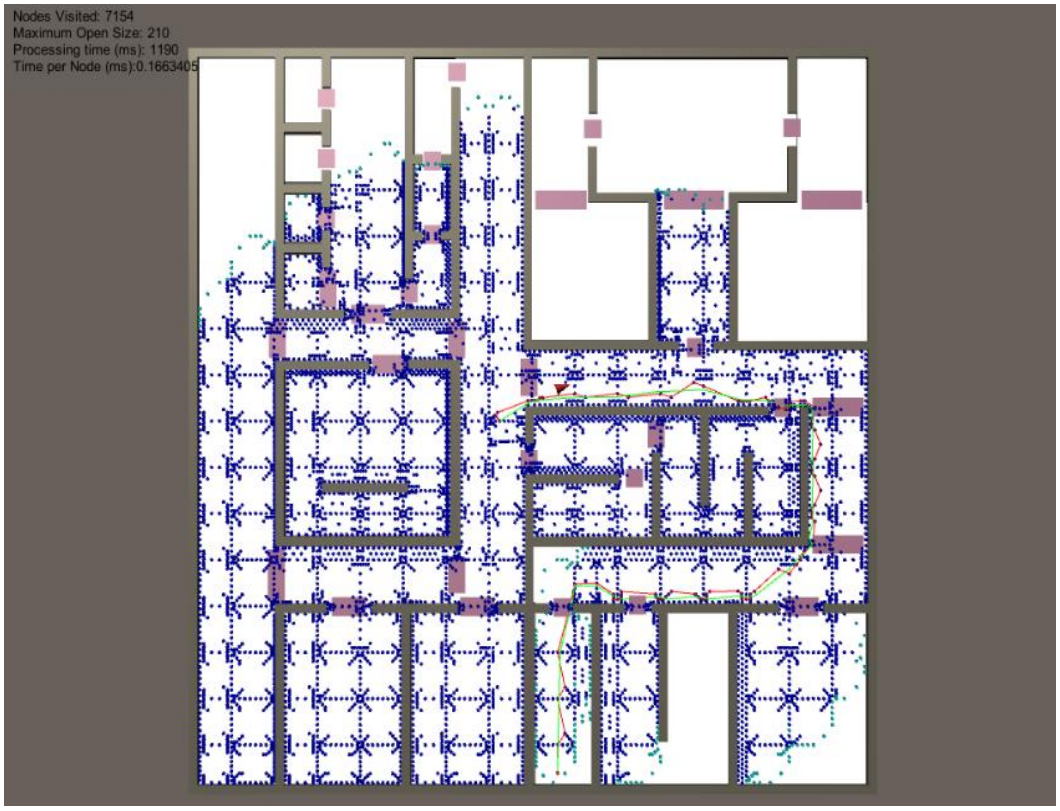
```
        {
```



\/ A* com Euclidean Distance tudo SimpleUnorderedLists



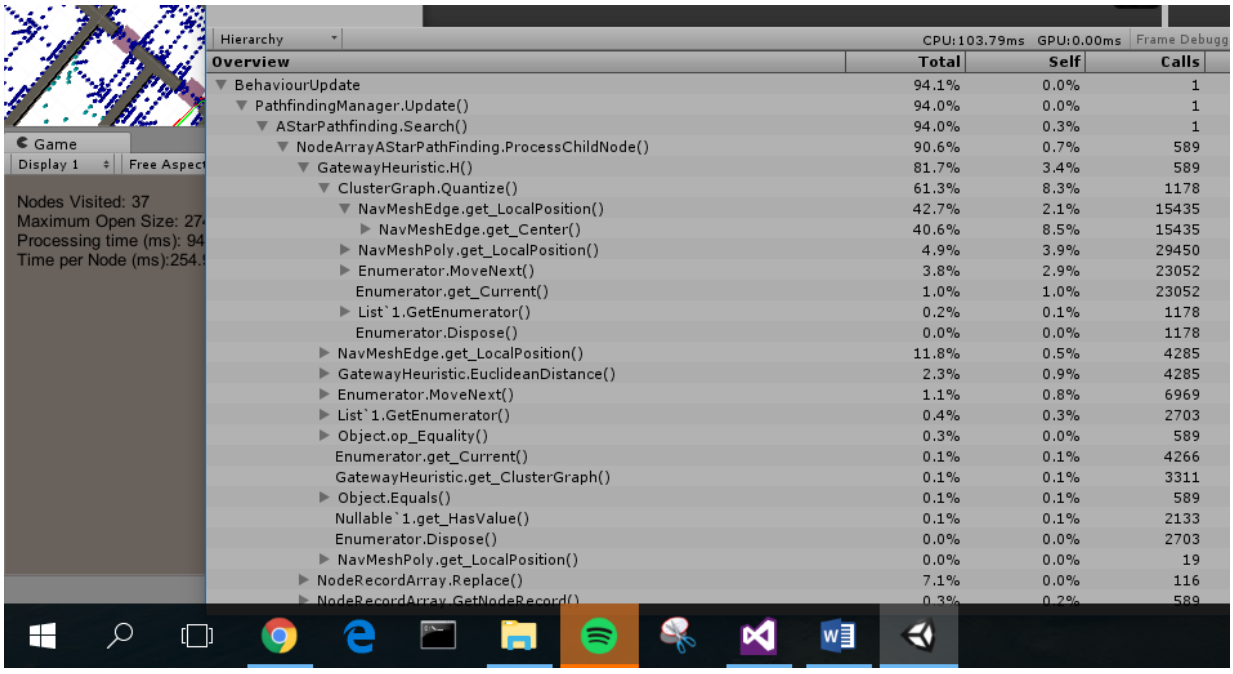
\/ NodeArray A* com Euclidean Distance



\/ NodeArray A* com Gateway

\/ NodeArray A* com Gateway e optimização

Profiller sem optimizações



Optimizações:

Tendo em conta o *profiler* verificamos que a função *Quantize* está a gastar uma grande percentagem do CPU. Ao analisarmos a função verificamos que tem um ciclo *for* que percorre os clusters todos de modo a descobrir a que cluster pertence o node. Como esta informação é estática, podemos fazer um pré processamento para ser mais rápida esta procura. Para isso criamos um dicionário que tem como *keys* os nós e os como *values* os clusters. Este dicionário é populado no início do programa quando estamos a criar os NodeRecord (aproveitamos um ciclo *for* para duas coisas). Depois desta melhoria conseguimos baixar bastante o impacto desta função.

Para além disso verificámos sempre que usávamos a função magnitude para comparações e trocámo-la pela sqrMagnitude.