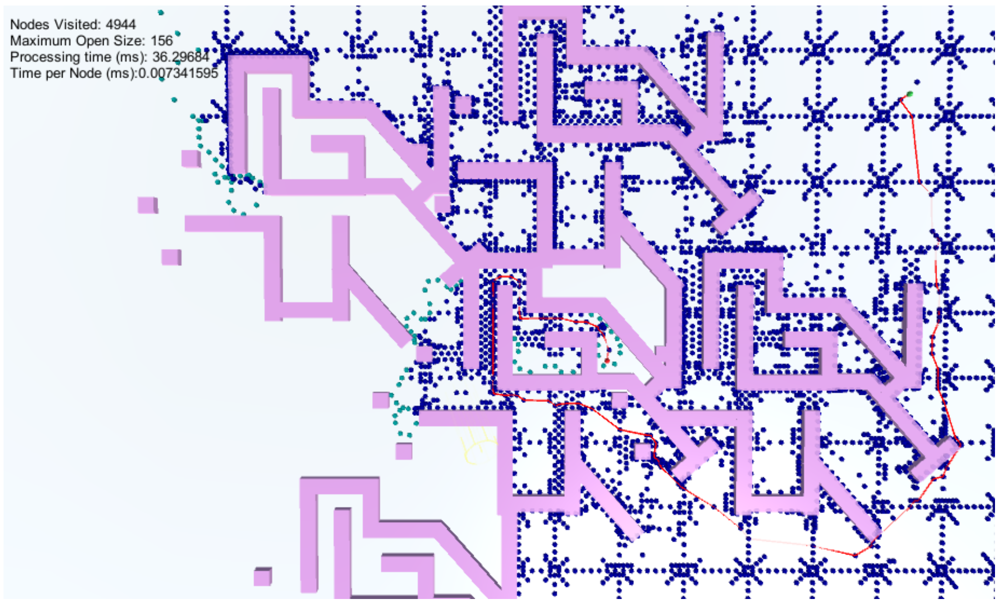
TÉCNICO LISBOA

# Inteligência Artificial para Jogos

2016-2017

Laboratory Guide

## Lab 4 – Optimizing A* for Pathfinding



```
Nodes Visited: 4944
Maximum Open Size: 156
Processing time (ms): 36.29684
Time per Node (ms):0.007341595
```

## 1) Explore the provided scene and the source code

a) Open the downloaded project in Unity and explore the scene *Pathfinding* . This scene is similar to Lab 3, but much bigger and with more obstacles and harder routes.In this laboratory you will analyze differences in efficiency by using different data structures for the Open and Closet sets, and will also implement the NodeArray A* algorithm.

b) Take another look at the classes, because some of them changed. Analyze the new NodeRecord class, the new IOpenSet and IClosedSet interfaces, and the Priority Queue implementation provided. Start by copying your implementation of A*, the heuristic, and any other data structure you implemented for Lab 3 to this project. There are some small modifications you will have to do to make your code to compile again. They are explained in the source code.

c) Analyze the code for the NodeRecordAStar class, and for the NodeRecordArray you will have to implement some of their methods later.

## 2) Compare the Unordered List for Closed set vs Dictionary for Closed Set

a) In lab 3, you implemented a Dictionary/Hastable for the ClosedSet. We will compare the efficiency between using the original unordered list and the dictionary. It is very important that you select more or less the same path when comparing the two data structures. Select a path that is complex enough (i.e. with a large number of processed nodes: 2000-3000 would be great) and that your computer can handle. Specify a number of 50-100 nodes processed per frame (the most your computer can handle without stalling). Your search algorithm will then run over a set of frames. When running the profiler, you will have many frames where the search algorithm is doing the search. Select one where the time spent is average (i.e. avoid the frames with the lowest and with the highest values).

b) Start by running the A* pathfinding algorithm with the unordered list for closed, and fill the next table with the corresponding information.

| **Method** | **Calls** | **Execution time** |
|---|---|---|
| `A*Pathfinding.Search` | 1 | |
| `GetBestAndRemove` | | |
| `AddToOpen` | | |
| `SearchInOpen` | | |
| `RemoveFromOpen` | | |
| `Replace` | | |
| `AddToClosed` | | |
| `SearchInClosed` | | |
| `RemoveFromClosed` | | |

c) Now do the same, but using the dictionary for the closed list, filling the next table. Try to explain why optimizing the closed set is so important (hint: look at the size of the closed set). Which of the three methods for the closed set is the most relevant?

| **Method** | **Calls** | **Execution time** |
|---|---|---|
| `A*Pathfinding.Search` | 1 | |
| `GetBestAndRemove` | | |
| `AddToOpen` | | |
| `SearchInOpen` | | |
| `RemoveFromOpen` | | |
| `Replace` | | |
| `AddToClosed` | | |
| `SearchInClosed` | | |
| `RemoveFromClosed` | | |

## 3) Comparing PriorityLists vs Unordered Lists for Open Set

a) Implement the LeftPriorityList and RightPriorityList classes. The LeftPriorityList orders node records from the lowest f to the highest, while the RightPriorityList orders node records from the highest f to the lowest. Perform profiling for both versions of the Priority lists and fill out the next table. Use the dictionary for the closed set.

| **LeftPriorityList** | | | **RightPriorityList** | | |
|---|---|---|---|---|---|
| **Method** | **Calls** | **Execution time** | **Method** | **Calls** | **Execution time** |
| `A*Pathfinding.Search` | 1 | | `A*Pathfinding.Search` | 1 | |
| `GetBestAndRemove` | | | `GetBestAndRemove` | | |
| `AddToOpen` | | | `AddToOpen` | | |
| `SearchInOpen` | | | `SearchInOpen` | | |
| `RemoveFromOpen` | | | `RemoveFromOpen` | | |
| `Replace` | | | `Replace` | | |
| `AddToClosed` | | | `AddToClosed` | | |
| `SearchInClosed` | | | `SearchInClosed` | | |
| `RemoveFromClosed` | | | `RemoveFromClosed` | | |

b) Compare the values of the LeftPriorityList with the ones of the unordered list. While the search execution is better, the time required to add a node increased substantially. It is important for you to realize that some data structures are more optimized for some operations than others. It is not possible to have a data structure where all operations are super-efficient. So we usually have to choose the data structure that optimizes the operations that are used the most. Looking at the previous table, what operations are performed the most?

c) Which PriorityList obtained the best results? Why?

## 4) Comparing Priority Heap for Open Set

a) Run the A* pathfinding algorithm with the provided PriorityHeap data structure. Fill the table below and compare with the other data sets.

| Method | Calls | Execution time |
|---|---|---|
| `A*Pathfinding.Search` | 1 | |
| `GetBestAndRemove` | | |
| `AddToOpen` | | |
| `SearchInOpen` | | |
| `RemoveFromOpen` | | |
| `Replace` | | |
| `AddToClosed` | | |
| `SearchInClosed` | | |
| `RemoveFromClosed` | | |

## 5) Implement and Compare Node Array A*

a) Complete the implementation of the NodeArrayAStarPathfinding. You will also need to complete the implementation of the NodeRecordArray. The NodeRecordArray is the data structure used for both Open Set and Closed Set in Node Array A*. Take a look at the theoretical slides in order to know what you need to do. The basic idea is that node records for all nodes are initially created, and you can use the node Index to access the corresponding node record almost immediately from the Node Record Array. The node record has a flag that specifies the current state of the Node (unvisited, open, closed). When a new search is performed all the NodeRecords have to be initialized to unvisited. We still need an Open data structure because we need to order the nodes to get the best ones (and you don't want to do this for the whole array). But for the Open you can just use a PriorityHeap. **Note:** to make this algorithm as fast as possible you should avoid creating new NodeRecords. Reuse the ones in the array. Also take that into consideration when implementing the NodeRecordArray.

b) Once you manage to implement the new algorithm, change the Pathfinding Manager to use the new search algorithm and run the same path as for the previous versions of the algorithm. Fill out the last table. Compare this version of the algorithm with the previous one. Compare this version with your initial implementation (i.e. the one in table 1). How much faster is this one? Which methods benefited the most? You will see that choosing the right data structure can have a huge impact in the algorithm efficiency.

| Method | Calls | Execution time |
|---|---|---|
| `NodeArrayA*.Search` | 1 | |
| `GetBestAndRemove` | | |
| `AddToOpen` | | |
| `SearchInOpen` | | |
| `RemoveFromOpen` | | |
| `Replace` | | |
| `AddToClosed` | | |
| `SearchInClosed` | | |
| `RemoveFromClosed` | | |