|  |  |  |
| --- | --- | --- |
| Pathfinding in Games and Agile Development | December 2  2012 | |
| A portfolio of three laboratory exercises, including background of the subject and discussion of methods and results for each of the three exercises. | | Portfolio of Work |

Table of Contents

[Introduction 1](#_Toc341968771)

[Week 6: GPSA and Hashing 2](#_Toc341968772)

[Introduction 2](#_Toc341968773)

[Work Done 2](#_Toc341968774)

[Discussion 2](#_Toc341968775)

[Results 2](#_Toc341968776)

[Methods 3](#_Toc341968777)

[Conclusions 3](#_Toc341968778)

[Week 7: The Lee Algorithm 3](#_Toc341968779)

[Introduction 3](#_Toc341968780)

[Work Done 4](#_Toc341968781)

[Discussion 4](#_Toc341968782)

[Results 4](#_Toc341968783)

[Methods 4](#_Toc341968784)

[Conclusion 4](#_Toc341968785)

[Week 8: The A\* Algorithm 4](#_Toc341968786)

[Introduction 4](#_Toc341968787)

[Work Done 4](#_Toc341968788)

[Discussion 4](#_Toc341968789)

[Results 4](#_Toc341968790)

[Methods 4](#_Toc341968791)

[Conclusion 4](#_Toc341968792)

[Conclusion 4](#_Toc341968793)

[Bibliography 4](#_Toc341968794)

# Introduction

Pathfinding is of particular importance in modern games, as environments become ever larger and more complex the AI routines developed to negotiate such environments must necessarily become more powerful and relatively less taxing to compute. Pathfinding in games can find its roots in Dijkstra’s algorithm for finding the shortest path through a weighted graph, though such complete solutions can be extremely taxing to compute in real-time, such as in games. The amount of CPU cycles dedicated to AI in general, of which Pathfinding can be considered a subset, is growing slower than the environments that must be negotiated as studios strive for ever greater graphical scope and fidelity AI must take something of a back-seat.

The following reflective logs have been chosen for their interesting features and pertinence to the subject of Pathfinding. Both the Lee and A\* algorithm are used within games as the main Pathfinding method and exploring STL structures such as has been done in the Hashing lab is another primary goal of this module.

# Week 6: GPSA and Hashing

## Introduction

Hashing is the process of splitting a large search group into a number of smaller groups by using the properties of the individual items to divide them into like groups. An example may be to take a large list of names and split them into groups for the letter than each begins with. This means that when we come to search the group we go not have to search through the entire list, instead we check the first character of the name for which we are searching and just search the small group of items that are share that hash.

By using the characteristic of the items to form a certifiably unique number we can ensure that, no matter the size of the list, our search times will not increase significantly.

The example used in this lab is finding all unique board states for the game of Noughts and Crosses.

## Work Done

The original implementation of the code to be modified employed a duplicates list, this list held a copy of every board state that was produced by the algorithm. Every time a state was created it was compared with the complete duplicates list to ascertain whether the newly created state was in fact a unique state and previously unchecked.

This code was first modified to make use of a Boolean array as a hash table. Then, in an attempt to improve the space efficiency of the algorithm, an STL map was substituted. Discussion and comparison of these methods will follow.

## Discussion

### Results

First, a side by side comparison of the three different solutions:

|  |  |  |
| --- | --- | --- |
|  | Debug | Release |
| Unaltered GPSA | 0.59s | 0.153s |
| Hash Table (Bool Array) | 0.007s | 0.001s |
| STL Map | 0.058s | 0.001s |

Table : Speed Comparison

It can be seen that both the Hash table and the STL map are substantially faster than storing the whole board state structure and checking against each one. For the sample sized used there is no discernible difference in speed between the STL map and the Boolean array, though as the map uses a binary tree it may not scale to very large numbers of items as well as an array. On the other hand the hashing function used has a large range and thus the array’s size is extremely large for the amount of data it is containing.

In this case the array has a space efficiency of less than 4% while not performing noticeably faster than the STL map.

### Methods

A further alternative would be a hash\_map structure, which is optimised specifically for use as a hash table. This is included in Visual Studio by default but is not part of the STL and would not therefore be guaranteed to be cross platform. Thus it was not included in this comparison.

The conversion of the code from a Boolean array to using an STL map was entirely trivial, as the STL map uses almost identical syntax to any normal array.

The hashing function itself was an interesting feature of the algorithm. In its essence it assigns a ternary number to each position on the board to define the three states each position can be in, it then pushes these into a string and casts it into an integer. This yields a maximum size of 39 or 19,683. This is why the array is 20,000 elements in size.

## Conclusions

For problems of a similar magnitude to this it seems most reasonable to use an STL map as the look-up times are not slow enough to warrant the massive reduction in space efficiency produced by a switch to the Boolean array. For situations where the number of elements is orders of magnitude larger then it may be worth retesting the array and map for speed. If there is a space efficient hashing function then it is highly likely that the Boolean array is the best choice, but for any situation where the hashing function leads to a sparsely populated array it is likely that the map would be preferable.

# Week 7: The Lee Algorithm

## Introduction

The Lee algorithm, or the wavefront or flood-fill algorithm, is a breadth first method for finding the shortest path between two points. It expands out from the start point assigning each new position a value to represent the distance (or number of iterations) from that point to the start point. Once the designated end point has been reached the algorithm backtracks from the end point to the beginning by choosing the node with the lowest distance value.

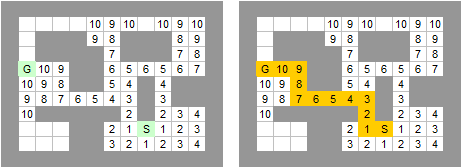


Figure : Lee Algorithm (Ericson, 2008)

## Work Done

An algorithm based on the principles of the Lee algorithm was produced. The algorithm will find the shortest path on an n x n grid, using only NSEW movement. If there is no viable path then the algorithm notifies the user as such.

The algorithm differs from the classical Lee in two areas. The first difference is in the use of distance. The algorithm does not actually make use of the distance at all, instead pushing all new nodes on the wavefront onto a FIFO queue, with every node being assigned a pointer to its parent node (the node from which it was first reached), this use of parent references is the second way in which the produced algorithm differs from the normal Lee algorithm. This representation ensures that any node will have a parent with the lowest ‘distance’ to the start point and allows the backtracking portion of the algorithm to simply follow a chain of parent references back to the start point.

## Discussion

### Results

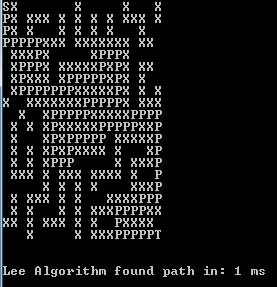
The algorithm performs well in the tests conducted, finding the shortest path through a 1000x1000 grid of randomly placed barriers in less than 0.1s.

Figure : Lee Algorithm Output

### Methods

## Conclusion

# Week 8: The A\* Algorithm

## Introduction

## Work Done

## Discussion

### Results

### Methods

## Conclusion

# Conclusion

# Bibliography

Ericson, C., 2008. *realtimecollision.net.* [Online]   
Available at: http://realtimecollisiondetection.net/blog/?p=83  
[Accessed 29 November 2012].

Ericson, C., 2008. *realtimecollision.net.* [Online]   
Available at: http://realtimecollisiondetection.net/blog/?p=56  
[Accessed 29 November 2012].

Nash, A., 2010. *AIGameDev.net.* [Online]   
Available at: http://aigamedev.com/open/tutorial/theta-star-any-angle-paths/#AAlgorithm  
[Accessed 29 November 2012].