Build a game AI

GAMes and AI

Sam McKay 4906379

2016

[Abstract 2](#_Toc443839483)

[Introduction 2](#_Toc443839484)

[Game and AI type 3](#_Toc443839485)

[Background theory 4](#_Toc443839486)

[Methodology 5](#_Toc443839487)

[3rd party additions 6](#_Toc443839488)

[Conclusion 7](#_Toc443839489)

[References 8](#_Toc443839490)

[Appendix 9](#_Toc443839491)

# Abstract

Almost all modern games rely upon some form of Artificial Intelligence (AI), ranging from simple pathfinding algorithms to fully fledged decision making enemies with teamwork abilities. One of the most common forms of AI found in most games is pathfinding, allowing AI agents to find the quickest route between two points.

# Introduction

The aim of this task is to implement a weighted A\* pathfinding algorithm which steers an AI agent over a varied map avoiding any objects. This report covers the necessary background research required to complete the algorithm as well as a detailed look at the implementation.

The popular indie game development engine Unity (Unity, 2015) has been used in order to provide a better visual representation of the algorithms progress. The entire program was written using C#, including the many prerequisites for the algorithm.

# Game and AI type

The AI shown here is an A\* pathfinding algorithm designed for use in a shooter game be it a top down or first person shooter. However, as a very generic algorithm it can easily be used for a large variety of game genres.

# Background theory

A star pathfinding(A\*) is one of the most common AI algorithms used in games. Its ability to efficiently find the quickest route between two points has many uses in games. A\* is an evolved version of the Breadth first search which uses an expanding frontier. The frontier is initially composed of the start locations neighbours, connecting nodes. Then simply select a node in the frontier, mark it as visited and then add all the node’s unvisited neighbours to the frontier consequently expanding it. This can be repeated until the frontier is empty or until we reach our destination node.

The main problem with using just breadth first for pathfinding is that you cannot add weighting to the nodes so an adapted version called Dijkstra’s Algorithm is used. Now a priority queue is used to represent the frontier allowing nodes to be added based on their cost.

Another improvement on this is called Greedy Best-First Search, this uses a heuristic based on the Manhattan distance to find the fastest route by picking the next neighbour based on its proximity to the goal node. Whilst this works great when there are very few obstacles in the way the paths produced are not the most efficient in a complex map.

The solution is to this is the A\* algorithm which “uses both the actual distance from the start and the estimated distance to the goal” (Red Blob Games, 2016). (POSSIBLY ADD MORE?)

# Methodology

There are several prerequisites for implementing the A\* algorithm, in order to minimise these the Unity game engine was used to visualise the end result.

Before the main A\* loop can be started, several other classes and data structures were required including, *SquareGrid*, *PriorityQueue*, *Location*. All of which are required to effectively implement the A algorithm. As well as these, Dictionaries were used to store key value pairs for *cameFrom* and *costSoFar*. This made it considerably easier and quicker to store and access all information about nodes and the costs to get to them.

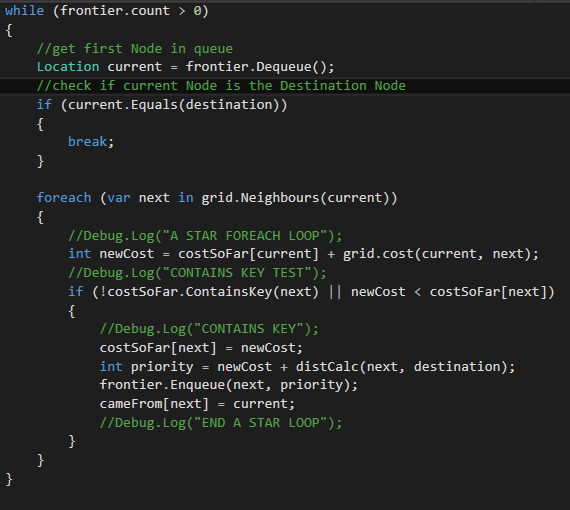
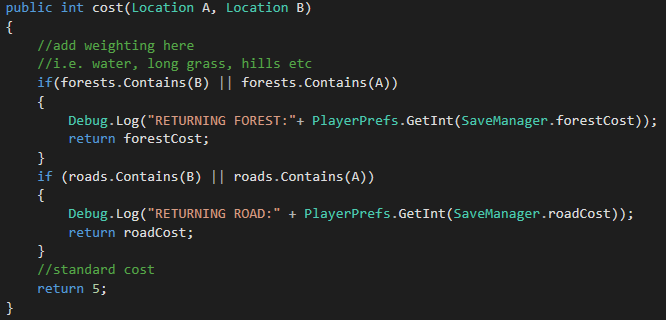


Figure A Star Loop

With all of the setup complete the A\* loop itself becomes a very simple loop which runs as long as the frontier isn’t empty. The first element in the frontier, a priority queue, is set to be the current location, a quick check is then made to see if the destination has been reached. This allows the program to break out of the loop as soon as the destination has been found, massively improving efficiency since continuing once the best route has been found is wasteful.

The next stage is to loop through the nodes neighbours, finding the cost to travel to each one and whether this route to that neighbour is the fastest route. If it is the quickest route to that node, then its added to the frontier. This makes it possible to visit nodes more than once, an incredibly useful feature since getting to one node via a slightly different route could prove quicker when there are weighted nodes. This is where A\*’s advantage over Breadth-first is demonstrated since Breadth-first wouldn’t even consider a node that has already been looked visited.

In order to calculate weighted costs a function was implemented to return a cost based on the node type. This was done by storing all nodes in appropriate HashSets. When the loop asks for the cost to travel to the node the below function will check if that Node exists any of the weighted categories, i.e. forests, roads. To demonstrate the advantage of weighting the default node will return a value of 5, allowing a “lighter” weight (1-3) to be applied to a *RoadNode*. In practice this means that the pathfinder will prefer travelling on roads as its quicker than a standard tile. As *forestNodes* are “heavily” weighted the path finder will attempt to avoid them unless it is quicker to travel through the forest than around it.



This has a many uses not only in games but also in real life, for example satnavs would use this in order to avoid congested areas or slower roads.

# 3rd party additions

Unity3D was used for visual representation of the pathfinding algorithm, all other assets used are included in Unity’s “standard assets” pack. No other scripts were used.

# Conclusion

# References

# Appendix