The path to the right decision: An investigation into using heuristic pathfinding algorithms for decision making in game AI

Ashley Smith

December 28, 2019

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras justo velit, vestibulum sit amet turpis in, interdum rhoncus magna. Proin pulvinar posuere iaculis. Duis vulputate tristique arcu, id pretium ante blandit ut. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nam augue tellus, mattis quis consequat id, facilisis eu lectus. Vivamus euismod non quam sed condimentum. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Phasellus vitae consequat nisi. Morbi vulputate tellus ut nibh vulputate, vitae blandit ex faucibus.

Contents

1 Introduction

1.0.1 Video games and artificial intelligence

Games are good for the global economy. Newzoo (2019) reports that in October 2019 the global games market was worth $148 billion. In order to remain competitive, developers are pushed to make bigger, better and more complex games. The evolution of the technologies available has allowed for a greater number of elements to be simulated in the game world which could potentially increase a game's worth (Blow, 2004). The entry barrier to creating these good-looking and well executed games has been lifted with the establishment of engines like Unity (Unity Technologies, 2005) and Unreal (Epic Games, 1998) to the point where even non-programmers can get involved using textual or visual scripting.

For many years, only the game's graphics were considered important (Blow, 2004; Yap, 2002), however, good physics systems or competent AI (Artificial Intelligence) are now recognised as a way of improving the end-user's experience just as much (Blow, 2004). Game AI is different from the normal, academic AI as it simulates behaviour and aims to be believable and fun, whereas an academic AI aims to achieve a level of intelligence or autonomy to excel at a given task (Nareyek, 2004, p. 60).

1.0.2 The need for game AI

AI has multiple uses within a games context but the majority of use cases employ AI to control the characters featured in a level of the game. Laird and VanLent (2001, p. 16) said that whether these characters are replacements for opposing players or characters that act as companions, villains and plot devices, "human-level AI can expand the types of experiences people have playing computer games". Using AI opens up the opportunity for increasing the difficulty of the game, and could be perceived as the kind of challenges that make games fun (Buro, 2004, p. 2). Laird and VanLent (2001, p. 16) also hypothesised that utilising such an AI is a good step towards the development of enjoyable and challenging gameplay, and potentially to "completely new genres" (Laird & VanLent, 2001, p. 17).

"Customers value great AI" (Nareyek, 2004, p. 60) and so it's important to choose a suitable approach that fulfils the expectations of the player thus the requirements of the game (Millington, 2019, p. 19). Academic AI can be made using algorithms inspired by biology such as neural networks or genetic algorithms and trained through iteration or

with datasets. These approaches aren't used in game AI because of the high requirements to train the AI to interact with a specific game (Nareyek, 2004, p. 64), moreover, it is easy to train the AI to play too strongly ruin the game (Tozour, 2002, p. 13). Instead, game AI developers embrace simpler, non-learning algorithms due to them being easier to understand, implement and debug (Tozour, 2002, p. 7).

1.0.3 Approaches to game AI development

The most basic forms of AI used in games take the form of a series of if-then statements and are known as a 'production rule systems' (Tozour, 2002). These statements are organised in a list and the AI uses the behaviour of the rules that evaluate to *true*. The result is a very basic AI that is not only limited to what actions it can take but also when it can take them. Similarly, decision trees combine the same if-then style with branching structures to create game AI (Nareyek, 2004, p. 62), and the tree and subtrees are recursively traversed until a leaf node is found with the desired behaviour. This process is very easy to understand and implement (Millington, 2019, p. 295), and the branching structure makes visualisation of the process more intuitive than the basic list used in a production rule system. Because of this, many consider decision trees to be one of, if not the simplest techniques to making AI (Millington, 2019, p. 295; Tozour, 2002, p. 7).

DIAGRAM OF DECISION TREES HERE

FSMs, or Finite State Machines, are the most common approach to game AI (Orkin, 2006, p. 1; Millington, 2019, p. 309) due to being easy to understand and the efficacy of their output. FSMs consist of a directed graph where each node represents a state and the edges represent the transitions between them (Tozour, 2002, p. 6). A character can only be in one state at time and has no memory of any previous states (Colledanchise, Marzinotto, & Ögren, 2014); each state represents an expected behaviour and determines what they do and the conditions to switch to a different state (Diller, Ferguson, Leung, Benyo, & Foley, 2004, p. 3). In the right environment, the impression of a well thought out FSM could compete with that a neural network, at a fraction of the time and resource costs, despite not always arriving at the optimal decisions (Sweetser & Wiles, 2002). However, each new behaviour requires the creation of a new state and the conditions of which this state integrates and transitions into other states, making expansion and maintenance cumbersome (Sweetser and Wiles, 2002, p. 2; Lim, Baumgarten, and Colton, 2010, p. 3). There's no easy way to combine the tests inside of FSMs and selecting the conditions for a

state transition is still very much a process that must be done by hand (Millington, 2019, p. 313).

FSM DIAGRAM

The need for more flexibility in game AI has lead to the creation adaptation of modular algorithms such as behaviour trees (Lim et al., 2010, p. 1). Like decision trees, the recursive structure of a behaviour tree is simple to understand and implement while also being high level, allowing for more sophisticated AI to be created in a modular fashion through the use of subtrees and different node types (Shoulson, Garcia, Jones, Mead, & Badler, 2011, p. 144), each performs an action or check and then proceeds to succeed or fail (Lim et al., 2010, p. 4). It is these types that make the AI process at a higher level than standard decision trees, and when combined with leaf nodes that perform checks and actions to build trees, and then combined again to make trees containing subtrees, the simplicity and elegance of this technique certainly demonstrates why behaviour trees are getting attention (Shoulson et al., 2011, p. 144).

BEHAVIOUR TREE DIAGRAM

1.0.4 The relationship of pathfinding and game AI

One common requirement for game AI is for the characters to be able to traverse the areas of the game in a way which meets the player's expectations logically and efficiently — a task known as pathfinding. Regardless of what an AI decides to do, a pathfinding mechanic needs to be in place to allow the AI to navigate to where it needs to go, manoeuvring around obstacles while still taking a sensible route (Graham, McCabe, & Sheridan, 2003, p. 60). For most games, the algorithm of choice is A* as it is the de-facto standard pathfinding algorithm (Millington, 2019, p. 197; Botea, Müller, and Schaeffer, 2004, p. 2; Nareyek, 2004, p. 64; Leigh, Louis, and Miles, 2007, p. 73).

The A* algorithm analyses the game's map and generates a path from one location to another while minimising a *cost* value — this value can represent anything but usually it represents the time or distance to travel along a given route (Yap, 2002, p. 44). This means that the pathfinding algorithm itself doesn't decide where to go, only how to get there and the manner in which it does so. When asked to calculate a path, a pathfinding algorithm is provided a graph of nodes to determine which nodes can be reached from which (Nareyek, 2004, p. 61). The algorithm isn't concerned in what the data represents

(2D or 3D coordinate data), the data's form (a tree, a graph or a list of connections) or the unit of measurement to calculate the weight of an edge (length, traversal time or monetary cost), as long as it is equipped with the right functionality to digest this information (Millington, 2019, p. 277; Graham et al., 2003, p. 60).

With the pathfinding process taking place after the decision has been made, the opportunity to involve the data gathered from pathfinding algorithm is missed. Often, the AI will decide to approach the nearest object, but obstacles in the way mean that the cost of navigating to the destination is greater than some alternative. While implementing the algorithm isn't difficult, ensuring the AI generates a path to the correct destination is difficult to do well (Forbus, Mahoney, & Dill, 2002). Perhaps on the way to the destination, the character has to also navigate past traps or other hazards where it will need to decide whether to avoid or pass through — decisions that A* isn't fully equipped to deal with on it's own. If the AI wanted to factor in these hazards and real cost values, it would have to perform a more advanced check on where to travel too, maybe even using the pathfinding algorithm multiple times to definitely make sure that it wants to take the generated path, potentially ruining the game's speed and overall performance.

Pathfinding algorithms are actually general purpose search algorithms applied to a spatial context (Cui and Shi, 2011, p. 125; Orkin, 2003, p. 6; Yap, 2002, p. 46); there is no such restriction that these algorithms should be limited to pathfinding when in a games context. Millington (2019, p. 197) said that "pathfinding can also be placed in the driving seat, making decisions about where to move as well as how to get there". With search algorithms having the flexibility of being able to traverse graphs of nodes representing any kind of data, it's no stretch to imagine the A* search algorithm being applied to a graph containing the same tests and actions found in decision and behaviour trees in order to generate a 'path' of actions rather than a path of spatial data (Higgins, 2002, p. 114). This is paper aims to investigate and re-engineer A* to make decisions rather than paths in a game context.

1.1 Literature Review

1.1.1 Dijkstra's algorithm: A graph and tree search algorithm

A search algorithm is a recursive method designed to find a match for a piece of data within a collection such as an array, graph or tree (Friedman, Bentley, & Finkel, 1976). A

piece of data is provided and the search algorithm typically returns whether it is present and it's location. Dijkstra's algorithm (1959) is a search algorithm that operates on trees and graphs (which are then interpreted as trees). The algorithm calculates the shortest difference from any node on the graph to any other node, and can be terminated early to avoid unnecessary computation if a destination is provided and found.

Dijkstra's algorithm works through the recursive summation and comparison of distance values starting from the a given start node (Dijkstra, 1959, p. 269). Each neighbouring node is added to the open list, then, the current node's distance from the start is added to the length between the current node and its neighbour. If this tentative value is lower than the current distance value of the neighbour, it replaces it. When all the neighbours have been considered, the node with the lowest tentative value on the graph is selected and permanently 'visited' and are removed from the open list (Dijkstra, 1959). This process is repeated until either there are no open nodes left or the destination, if provided, has been visited, and thus a path and the distance from the start to the destination can be retrieved. The problem with Dijkstra's algorithm is that it always selects the node with the lowest tentative distance value, meaning that the algorithm has no notion of direction and is calculating the lowest distance to nodes that may not be relevant to getting to the destination (Millington, 2019, p. 214).

DIJKSTRA'S ALGORITHM DIAGRAM

1.1.2 A* algorithm: A heuristic best-first search algorithm

A* is an improvement of Dijkstra's algorithm in this regard (Hart, Nilsson, & Raphael, 1968, p. 101) — while it doesn't stray far from how Dijkstra's algorithm works in the sense that it operates using a tentative distance value and it keeps track of the nodes that have and haven't been visited, it does extend the algorithm using what's known as a heuristic approach (Cui & Shi, 2011, p. 126). "Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal" (Pearl, 1984, p. 3). This means that in a pathfinding situation, a heuristic function could estimate the distance to the goal, by ignoring walls and measuring in a straight line, to direct the algorithm in the right direction and avoid evaluating routes that travel in the wrong direction to make the process more efficient (Cui & Shi, 2011, p. 127). Heuristics enable A* to perform a best-first search (Yap, 2002, p. 46), as the heuristic now has the power to select which node is the best to

evaluate and prioritise over the others (Russell & Norvig, 2016, p. 94).

When identifying the next node to expand, A* will take this heuristic distance into account using the formula $f(n) = g(n) + h(n)$ (Hart et al., 1968, p. 102; Russell and Norvig, 2016, p. 95), where $g(n)$ is the real distance the algorithm has calculated from the start node to node n and is the distance to be minimised while finding the goal, $h(n)$ is the heuristic distance from the current node n and the destination, and $f(n)$ is the combination of these two metrics forming an estimate of the distance from the start node to the destination if travelling through node n, also known as the fitness value (Hart et al., 1968; Millington, 2019; Graham et al., 2003, p. 64).

This heuristic component of A* transforms it into a family of algorithms where applying a different heuristic selects a different algorithm (Hart et al., 1968, p. 107), moreover, implementing A* and using a heuristic that returns a constant value for all nodes reverts A* back into Dijkstra's algorithm (Lester, 2005, p. 10; Millington, 2019, p. 237). Conversely, implementing a well-designed heuristic method can be used to guarantee optimal solutions, and using a heuristic that is somewhere in-between can output results with varying degrees of accuracy in exchange for faster execution (Millington, 2019, p. 219). The implementation of a good heuristic can be difficult, as making the heuristic take more factors into account for accuracy has the drawback of making the algorithm less efficient overall with the heuristic being frequently used throughout the process.

On the other hand, Graham et al. (2003, p. 68) argue that one of the constraints of games the industry is the "over-reliance" on the A* pathfinding algorithm and describe the development of its many extensions as a way of avoiding the discovery of new techniques. The pathfinding process can require a lot of CPU resources, sometimes to the point of stalling the game, when applied to larger graphs (Cui and Shi, 2011, p. 127; Stentz, 1996, p. 110; Graham et al., 2003, p. 67). This indicates that the performance of A* can vary depending on various factors, and is why it is important to optimise A* by selecting an suitable storage mechanism for the graph and internal node storage as well as using a reasonable heuristic that balances efficiency and efficacy (Millington, 2019, p. 228).

CRITICAL: A* DIAGRAM

1.1.3 Processing a character's perception of the game world

These approaches need a way to digest information about the character and its environment that is relevant to the decision making process, and selecting this information is just as important as the form it is delivered in (Cui & Shi, 2011, p. 126). This extraction of world data can vary in difficulty (Diller et al., 2004, p. 3) and is done for both performance and gameplay needs. All game AI solutions need the world to be re-interpreted to be better suited for both decision making (Buro, 2004, p. 2) and pathfinding (Diller et al., 2004, p. 3). Particular geometry of the map may need to be interpreted as vantage points, choke points or safe spots; particular formations of enemy units may need to be not only counted but also assessed for tactical strengths, whether engaging the units head-on is better than running away to a better location, and finally, the interpretation of time such as whether there is enough time to navigate to what would otherwise be a better location for fighting the enemy (Buro, 2004). An AI would then take this abstraction of the world, combine it with the character's data and then consider what the character should be doing — if the character is in an 'attacking' state and the player is nearby then the decision would involve getting into a position and hitting or shooting at the player.

If A* is to be used for game AI it also needs to operate on non-positional data. In standard pathfinding, each node is a position in the world and the edges between these nodes are the movements to get from one to another. Substituting each node to be an AI state and each edge to be an action that causes transitions from one state to another creates a graph that A* could process. The implementation the states and actions depend on the expectations of the AI — traditional pathfinding could be implemented by having an action that triggers movement and changes the position variable in the character's state. Regardless, a graph consisting of these nodes could be difficult to process and reduce the speed of A*.

A* can suffer from performance problems when used with many agents at once, or with a large or inefficient graph of nodes (Graham et al., 2003). Many solutions have been discovered to prevent or reduce the impact on performance — one such solution is known as hierarchical pathfinding, where the game's map is simplified into chunks, (Cui & Shi, 2011, p. 126). By simplifying a group or grid of nodes into a single node that represents the whole group (like a quadtree), the pathfinding process can be applied to larger worlds as if they were actually smaller (Botea et al., 2004). This works well for standard pathfinding, but if A* is applied to something other than purely spatial world data then the nodes need

extra consideration.

DIAGRAM ABOUT HIERARCHICAL PATHFINDING

There is a large amount of actions a character can take at any given moment (Nareyek, 2004, p. 62) and so applying a search algorithm to such a large set of tasks will make the process a lot slower than traditional methods if left unchecked. Hierarchical pathfinding essentially creates more nodes to expand in the short term to reduce the total number of nodes expanded in the long term and gain a net increase in performance. If hierarchical pathfinding reduces the amount of destinations, by collecting connections between the nodes in a given area, and simplifies navigation to and from this area then this technique can also be applied to the character's state so long as these connections can be aggregated using other metrics other than purely relying on their proximities to each other. One such way of grouping could be separating 'attacking' actions from 'defending' actions so that character's deciding to play defensively can eliminate one route of evaluation rather than multiple. Splitting gameplay elements like this can get complicated — if a certain unit in a strategy game can attack from a longer range, would attacking the enemy from a distance be a defensive manoeuvre or an offensive one, and how easy is this to change on a per-unit basis (Weber, Mateas, & Jhala, 2011)?

1.1.4 Making decisions with A* using GOAP

Orkin (2003, p. 11) expressed that expectations of AI are growing with the release of every new game and that "we need to look toward more structured, formalized solutions to creating scalable, maintainable and re-usable decision making systems". Techniques used to create the Game AI featured in most games do not contain the scalability Orkin envisions (Laird & VanLent, 2001, p. 17), however, Higgins (2002, p. 117) declares that a pathfinding engine can be created generically, especially if a templated language is used, to give the code even more reusability. Implementing A* in this way would allow it to be used for both regular pathfinding and anything else that could requires searching (Higgins, 2002, p. 120). With A* being efficient and optimisable (Millington, 2019, p. 215), reusing A* for game AI has the potential to bring the benefits it usually brings to pathfinding to AI while being adaptable enough to scale up and meet expectations.

Orkin (2006, p. 1) worked on the development of the game AI for the game F.E.A.R (Monolith Productions, 2005). The approach used for this AI was called GOAP which stands for 'Goal Oriented Action Planning' and uses the A* algorithm as part of its decision

making process and "allows characters to decide not only what to do, but how to do it" (Orkin, 2003, p. 1). GOAP changes the data to be processed by A* from spatial data such as coordinates into character AI state data, such as what is found in FSMs. Therefore, the output of A* is no longer a sequence of movements but a sequence of actions also known as a plan (Orkin, 2003, p. 2; Tozour, 2002, p. 6).

GOAP essentially takes the idea of having state machines to encapsulate behaviours and replaces the hand-programming the conditions and connections of state transitions with a pathfinding process with the aim being the decoupling of states and their transitions (Orkin, 2003, p. 2). The pathfinding process uses each node to represent a state and the edges between each node as the actions that lead to those states (Orkin, 2003, p. 7) When GOAP is used, a goal is given for the character to achieve, which drives the AI to play the game rather than idly remain inactive, and what is returned is the sequence of actions that will satisfy the goal (Orkin, 2003, p. 1). An action is a representation of one thing the character will do to change the world in some way, like opening a door or picking up a weapon (Orkin, 2003); some actions have preconditions that require the execution of another action prior to it (Orkin, 2003, p. 5). A* will then find the sequence of actions, the plan, that satisfies the character's goal while minimising an arbitrary *cost* value — Orkin (2003, pp. 4–5) suggests that this process creates interesting character AI that can adapt to change while also having a code structure that is reusable, maintainable and "elegant".

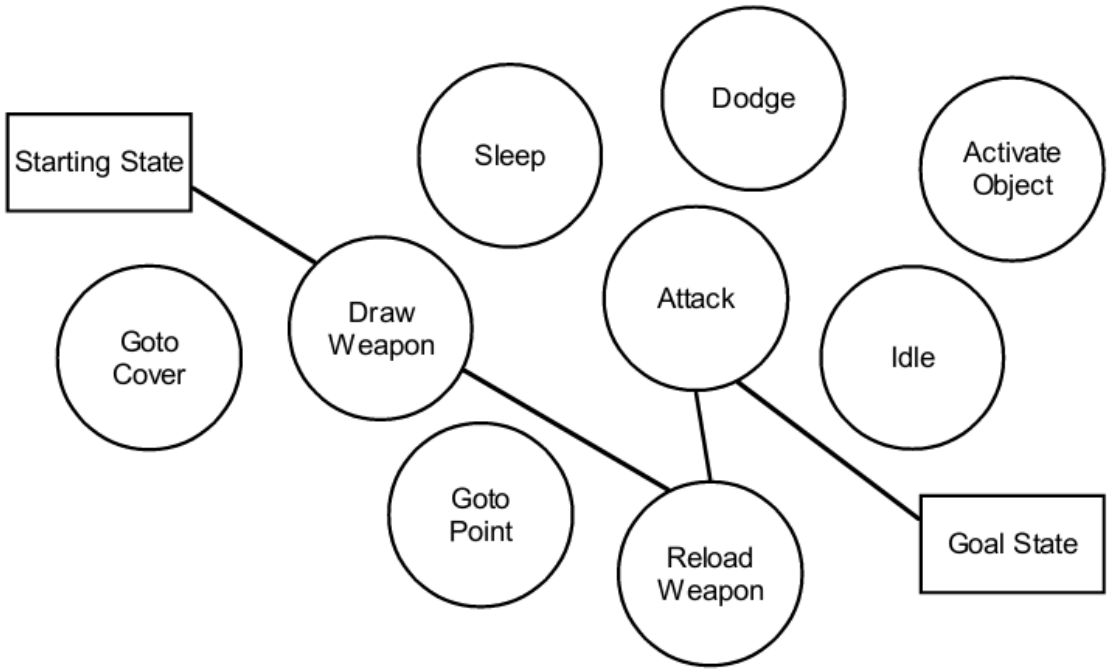

Figure 1: An example of GOAP (Orkin, 2003, p. 3)

A* being used in GOAP means that fundamental formula $f(n) = g(n) + h(n)$ (Hart et al., 1968, p. 102) is implemented in a way such that A* can perform a best-first search and expand the actions that are more likely to be useful to the situation (Orkin, 2003, p. 7). To calculate $h(n)$, Orkin (2003, p. 7) also shares that the heuristic in GOAP can be calculated as the summation of unmet conditions of the goal node, but this is rather unclear. Given the example in figure 1, "the goal is to kill an enemy" (Orkin, 2003, p. 3) could be interpreted in various ways that all seem unsuitable. If the goal condition is that the enemy is dead, the heuristic would be the same for all nodes but the goal state, making it potentially wasteful like Dijkstra's algorithm (Millington, 2019, p. 214). Alternatively, if the condition was divided into multiple conditions such as the weapon needing to be drawn, the weapon needing to be loaded, and the enemy being dead, it is implied that either the algorithm has been run previously to determine the requirements of killing an enemy, or that they were programmed by the programmer — how would the starting state know that the weapon needed to be reloaded before drawing it? While GOAP does seem elegant, the graph in figure 1 doesn't have a sizeable number of nodes; using dijkstra's algorithm wouldn't incur much of a performance cost with a graph this size. Orkin (2003) suggests that a regressive search by starting at the goal and pathing to the start makes sense, and in figure 1, it does, as only one action can attack and so it's clear that the decision making process boils down to the character attacking the enemy.

Another problem of GOAP is that the high-level nature of the approach may not allow for enough control of the game (Stanciu & PETRUŞEL, 2012, p. 87) — while making abstractions of the world is necessary for AI to process the information (Buro, 2004, p. 2) and thinking on a higher level is beneficial for making more human-like decisions, the precision of the actions taken by the AI is both low level and very noticeable if incorrect (Graham et al., 2003, p. 60). Figure 1 shows a decision making process that decides what to do but not necessarily where to do it. When the action "Goto Point" in figure 1 gets executed, there's no indication of how the destinations are generated other than satisfying the preconditions for another action (for example, moving to an object to activate it) (Orkin, 2003, p. 7).

This isn't specific to GOAP as it applies to every AI approach where the generation of these locations isn't part of the AI. Is the character's AI deciding where to go, or just deciding that it has to go somewhere? In order for "Goto Cover" to act in a similar way to the "Goto Point" action, a goal or action precondition would either have to require the character to be out of line-of-sight, or be using one of the designated cover spots —

this is how F.E.A.R (2005) approached it (Orkin, 2006, p. 12). The former would require calculating the closest position that allows the escape of line-of-sight. The latter of these would require set locations to be marked as points of interest and then A* would be employed to find which cover spot would be the quickest to navigate to. Both of these would have their own problems, and wrapping the entire pathfinding process as a single action would mean that the information from the pathfinding request would not be used in the decision making at all. Without utilising the AI's ability to create an impression of thought, the information given to these actions could be poorly chosen and "may be perceived as a lack of intelligence by a human player" (Graham et al., 2003, p. 63).

1.1.5 Defining the notion of cost in the context of game AI

Cost, sometimes referred to as weight, is a term that will continue to be used when talking about A* as it is the the metric that governs the searching process (Graham et al., 2003, p. 60). *Cost* doesn't have to be a numeric value, as long as it can be compared and combined correctly with other *cost* values, however, one numeric restriction of cost is that it cannot, or rather should not, be negative. The method A* uses to determine if a route should be expanded before another is if its *cost* value is lower — when only positive values are added together it is assumed that there isn't a way for a *cost* to decrease in value. While in mathematics it is entirely possible and valid for these values to be negative, the problems that make this necessary are not applicable to games (Millington, 2019, p. 202).

In the formula $f(n) = g(n) + h(n)$ (Hart et al., 1968), each function returns a *cost* — the combination of the goal and heuristic *cost* values, also known as the fitness value, is used select the next open node to evaluate (Russell & Norvig, 2016, p. 94). If a and b are two nodes in a graph and $f(a) < f(b)$, A* will interpret this as node a being a more logical choice to expand (Orkin, 2003, p. 7). Node b won't be evaluated unless the route derived from node a results in dead ends or more routes which have worse fitness values than node b. With the order of execution being so dependent on $f(n)$, it is important that the definition of the *cost* of an action is calculated correctly. It is difficult to design such a system when everything is arbitrary, but metrics such as distance can be used to prioritise actions with lower distances such as movement; resources such as gold or mana could be used in game specific interactions to minimise spending important resource, and the time spent doing the action can also be factored in to drive the character to resolve the situation as quickly as possible (Lester, 2005, p. 8). Millington (2019) said that "the cost

function is a blend of many different concerns, and there can be different cost functions for different characters in a game" and that this is called "tactical pathfinding". The ability to change the *cost* per character further demonstrates the adaptability of A* — changing the way a character interacts with the world is just changing these *cost* functions so that the A* prioritises one action over another.

Some decisions are more troublesome to weigh than others. With the constraint of non-negativity, what would the *cost* be of an ability that regenerates mana instead of expending it? The only way apply reductions to values in this way would be to have a baseline *cost* for an action and then add or subtract from it, however, this does mean that this baseline value would dictate the maximum value of the reduction and so forward planning is necessary to ensure that all reductions can be applied in a balanced way. Another difficult type of decision to way are ones that don't have inherent characteristics; with a good goal for AI being unpredictable, surprising behaviour (Scott, 2002, p. 17), how would the incentive for performing strategies like flanking and ambushing be created, and how would it compare to the *cost* for attacking an enemy directly head-on? Orkin (2006, p. 14) said that in F.E.A.R (Monolith Productions, 2005) this behaviour just came naturally from using a dynamic decision making system such as GOAP and that while they were prepared to implement "complex squad behaviour", none was implemented — the squad's ability to ambush "emerged" from the combination of the "squad level" decisions and the individual character's decisions which gave the illusion of complex behaviour. Orkin doesn't mention much more on this subject and so GOAP's ability to surprise remains uncertain.

Harmon (2002, p. 405) suggests that an "opportunity cost" could be introduced which represents the other actions that cannot be taken as a result of taking the given, mutually exclusive action, the objective being to apply penalties to actions due to the fact that an alternative and possibly better course of action is available. While this makes sense when it is difficult to reward the AI for performing these strategies, but adding penalties to other actions could get complex, there is no discussion of this concept from Orkin (2006) from his work on F.E.A.R. Although, it is unclear whether Harmon means to apply penalties based on the current action or the other actions, and if it's the latter, is this a flat *cost* shared amongst all actions or does it try to give smaller penalties to the better tasks somehow? Regardless, to calculate this *cost* some mechanism of the AI, such as the weighting function used to determine $g(n)$ or possibly the nodes themselves, needs to have knowledge of other nodes so that this opportunity cost can be determined and used, yet again requiring forward planning.

GOAP uses a simple "action cost" system to prioritise which tasks to carry out first (Orkin, 2006, p. 11). Using an integer or floating point value is ideal for the type of *cost* as it can be added and compared with other values trivially. However, GOAP's method of creating unique characters is to assign different actions to make them act differently (Orkin, 2006, p. 8) as opposed to having them evaluate situations differently as Millington (2019) suggested. Giving characters different actions will implicitly change how they prioritise the same situation as they will be given new routes to explore, and for most games this might be the best idea. It could be a good idea to split the *cost* type into a set of values to give actions different values. Throwing a grenade and shooting a pistol could be considered equivalent in difficulty to perform, but typically grenades have more firepower in exchange for there being less available; reducing these actions to a single value could be difficult depending on the approach, but splitting the value into a set would allow the programmers to keep track of firepower and usage penalties separately. When needed for comparison, this set would evaluate to the sum of it's parts — but certain characters or situations could choose to adjust or completely omit parts of the *cost* so that actions are perceived in different ways depending on how a character views a certain cost. For a fast character, the *cost* to move might be halved and the *cost* of being exposed doubled in order to give the impression of an agile and hard-to-hit character.

1.1.6 Generation, selection and application of goal state nodes

When the game world has been processed, the actions a character can perform have been laid out and the methods of evaluating courses of action have been provided, the only things remaining that A* needs to function are the start and goal node states for the actual decision. The starting state is trivial as it is simply the current state of the character and world; the goal state node requires more consideration than that though. In standard pathfinding, the output is the shortest path from the starting node to the goal node, if such a path exists (Nareyek, 2004, p. 61). For game AI though, the goal needs to represent how the character or world should be — or rather the objective outcome of the decision making process. This objective can be difficult to ascertain as a goal node could represent anything; what seems to be a simple goal such 'win the game' becomes a rigorous series of tests to both calculate the *cost* reaching the goal than another (Harmon, 2002, p. 403). On the other hand, objectives that are too small or disconnected may not combine correctly to form this over-arching goal of winning the game. A balance is needed, whether that means the objective is to chase the player or defend an area, the objective

needs to be focused on winning without being vague.

While Orkin (2003) talks about in great depth about how GOAP handles making a decision to satisfy goal conditions, but doesn't describe where these objectives come from. A "squad coordinator" is mentioned (Orkin, 2003, p. 13) that organises multiple agents into squads when they're close together, but the actual goals of the AI can come from both the character's individual AI and the squad coordinator. GOAP doesn't replace an FSM, so it could be inferred that the character's state when a goal is reached determines the next goal and thus the character's behaviour, with the initial state being defaulted, scripted or randomised (Orkin, 2003, p. 2). Alternatively, game AI could be created in layers, and the output of one layer could be the desired goal of another, but this would mean that this problem propagates to the highest level.

Another talking point regarding goals is the amount of designated goal state nodes in the graph. There is typically only a single goal in standard pathfinding, but it is possible, maybe even advantageous, for some games to contain multiple goal nodes in a graph (Millington, 2019, p. 272). Instead of checking for a match with a certain location (or state, when developing AI), a method that checks whether a given node meets the requirements of being classified as a goal could be used instead; the heuristic would also need revision as it would need to calculate the heuristic *cost* to reach the 'nearest' goal rather than just the single, given goal (Millington, 2019, p. 272). Having multiple goals and goal types (Higgins, 2002, p. 121) would grant the ability for the AI to re-route to a different goal if it's easier and therefore accomplish the same task in multiple different ways without creating generic goals; this has its drawbacks though, one being the need for a more intricate and potentially confusing implementation and design of the AI needs, the other being the creation of balancing difficulties to ensure goals are prioritised as expected.

1.1.7 Summary

Search algorithms such as A* are generic, maintainable and versatile and are therefore theoretically suitable replacements for FSMs and behaviour trees for implementing game AI. While GOAP does use A* for part of it's decision making process, it isn't a complete solution and still separates decision making from the pathfinding process. This is acceptable and valid as GOAP is for generating a sequence of actions whereas pathfinding is strictly for navigating the map in order to perform these actions. Unfortunately, some information found during pathfinding that could be considered useful for decision making

is lost unless explicitly communicated — a decision might request to navigate to some location, but the path generated might be longer than expected and a different course of action could have been more appropriate. Without replanning, GOAP's disconnect between these systems could result in the wrong decisions being made.

In this paper, the mechanisms of the A* search algorithm are examined and re-engineered, through the substitution of input and output types, to investigate the modularity and adaptability of an AI that uses search algorithms to make decisions while actively involving pathfinding in the process as opposed to keeping these systems separate. Several approaches to defining goals and heuristic methods will be used to visualise the effects they have on a squad-controlling game AI. The aim of using this approach is to bring decision-making and pathfinding closer together and therefore simplifying the overarching process of perceiving, deciding and interacting in the game world.

1.2 Methods and Methodologies

1.2.1 Algorithm selection

As discussed, this paper will be using A* as the pathfinding and decision making algorithm of choice. A* is already used in many games for pathfinding mechanics (Millington, 2019, p. 197) and so it could be inferred that any developers intending to implement game AI using a search algorithm would prefer to reuse the A* algorithm specifically. A* is more of a family of algorithms rather than just one (Hart et al., 1968, p. 107) — implementing A* naturally grants access to using Dijkstra's algorithm (1959) when given a constant heuristic (Lester, 2005, p. 10). Similarly, providing different heuristic functions allows the adjustment of the effectiveness and efficiency of A*'s output (Hart et al., 1968, p. 107).

There are multiple adaptations and extensions of A* because of its wide usage. IDA* (Iterative Deepening A*) is an algorithm based on A* that eliminates the need for the internal storage (Korf, 1985, p. 36) to reduce the amount of memory used during execution at the potential cost of computational speed (Botea et al., 2004, p. 2; Yap, 2002, p. 44). This is done by creating a threshold initially set to $threshold = h(start)$ and terminating the current iteration when $f(n) > threshhold$, which then updates the next iteration's threshold to the minimum $f(n)$ that exceeded the the current threshold (Korf, 1985, p. 103). It is this repetition of expansion that can make this algorithm inefficient (Yap, 2002, p. 46), but the lack of overhead from storing nodes could have the opposite effect

and make the algorithm run faster than A* in some situations (Korf, 1985, p. 106).

D* (Dynamic A*) is an adaptation of A* that operates under the assumption that the cost of navigating from one node to another can change, such as when the environment is only partially known (Stentz, 1997, pp. 1–2). The navigation of these environments require frequent map updates and the constant regeneration of paths, a process which is already expensive. D* aims to generate optimal paths efficiently for full, partial or no knowledge of the map (Stentz, 1997, p. 2). To do this, D* mainly features the functions $PROCESS-STATE$ and $MODIFY-COST$, the former to calculate paths to the goal and the latter to modify and register costs and their affected states (Stentz, 1997, p. 3).

MAYBE DIAGRAM COMPARING ALGORITHMS

These adaptations could be used instead of A*, however every extension of A* comes with its own benefits and drawbacks. Only the game AI developer knows which search algorithm is most relevant for their use case, therefore for the purposes of this investigation only the standard A* algorithm implementation will be used as it is the most commonly used algorithm and will be applicable to most games.

1.2.2 Language selection

For a lot of developers, the choice of programming language comes from the requirements of their chosen game engine. Unity (Unity Technologies, 2005) uses C# or JavaScript, Unreal (Epic Games, 1998) uses C++, and some engines implement their own language, one example being Godot (Linietsky & Manzur, 2014). Godot gives developers a choice between C++, C#, languages that have bindings written to Godot's GDNative module, and finally their own language called GDScript which writes like Python. Most engines take advantage of C++'s performance even if they don't use it as a scripting language — Unreal, Unity and Godot all use C++ as part of their engine.

Unity (Unity Technologies, 2005) and Unreal (Epic Games, 1998) are both popular, so naturally C# and C++ are both used a lot in the industry. Blow (2004, p. 30) said that C++ was used in most games, most likely due to C++'s speed of code execution being useful for resource intensive engines and video games. C++ also features a template system which grants access to generic programming; templated classes and functions are only defined once but can be reused with different parameter types. This is very suitable for this paper's use case, while a degree of genericness can be achieved through base classes

and virtual functions, C++ templates are an alternative that doesn't require the use of inheritance or the grouping of classes (Higgins, 2002, p. 117). Templated code is used to generate a new class or function for a given type at compile time, meaning that one implementation of the A* algorithm can be given parameters of any type have multiple uses throughout a single program (Higgins, 2002, p. 120).

A* could also be implemented with C#, but C#'s generics do not grant the same freedoms that C++'s templates do. C# is a higher level language, so elements such as memory management are either hidden or contained in abstractions in an effort to make these things simpler. The abstractions of high level languages can be reasonable and advantageous for some developers and games, but with pathfinding being a vital element to a lot of games and is therefore commonly built into game engines it could be said that features of a low level language like memory management or C++ templates are more applicable for implementing pathfinding algorithms. It is for this reason that C++ is to be used for the purposes of this research.

1.2.3 Tool and framework selections

To research game AI, a game environment needs to be created so that the AI's interactions can be observed. This requires making some sort of video game and so an appropriate tool needs to be chosen. Established game engines are viable candidates as they trivialise the process of programming the necessities for a standard video game: opening a window, loading resources, rendering the environment and handling input. Even though a simple text adventure can be programmed with very basic programming knowledge, creating and visualising the choices for the AI to make would be time consuming and hard to debug. A library like the Open Graphics Library, or OpenGL (Silicon Graphics, 1992), allows 2D and 3D graphics to be used in a video game and would make visualisation of the AI and environment easier. Alternatively, using an established engine introduces technology such as debugging along with the basic functionality — this technology could be considered redundant use in research as simply running the project would require the installation of a relatively large piece of software.

SDL (Simple DirectMedia Layer, Lantinga, 1998) and SFML (Simple Fast Multimedia Library, Gomilia, 2007) are two libraries that could be considered a good compromise between low level graphics API usage and full-featured commercial engines. Both SDL and SFML implement essentials like opening a window, rendering graphics and responding

to input without introducing unnecessary functionality. Programming a full game using these frameworks requires greater technical knowledge and effort than using a traditional game engine, but for the purposes of this research, fewer game systems are necessary and so using SDL and SFML is more acceptable. SDL is used as a starting point for game engines just as much as it is used for standalone games; when paired with a graphics API like OpenGL, SDL facilitates the creation of the game's window and input handling while providing basic rendering functionality with the expectation that the developer will use the graphics API to draw their game.

SFML on the other hand is more of a wrapper for OpenGL specifically, and contains an assortment of useful predefined classes to be used when rendering 2D graphics game. SFML is much larger when compared to SDL because of this, but for a basic game it would be nonsensical to reimplement low level graphical functionality just to get objects appearing in the game's window. With all things considered, SFML seems to be an intelligent choice for this experiment — it provides a suitable amount of basic and additional functionality for building a simple 2D game without requiring the installation of a game engine.

END

- Tools: - SDL vs SFML: - Both of these implement rendering not much difference - This project focuses on the implementation of A* and so the framework for making the actual game doesn't matter so much. I used SFML because the defined classes for sprites are nicer to work with than SDL and because the library has more functionality less work needs to be done in getting a basic game rendering - Imgui - Easy to implement - Allows for debugging and analysing the algorithm - Can be used to make level making / editing - Used in many games

- Research methodologies: Qualitative (yes) vs quantitative - Why case studies? - Researcher studies subjects in natural environment rather than isolating variables - Caters to both qualitative and quantitative - How to case study: - Define objectives of case study - Define the case itself and the research questions - Draw up a table of possible cases and the questions to be asked to explore them - Determine what shouldn't be done (Baxter & Jack, 2008, p. 546) - Keeps study in scope - What will be studied - How will data be recorded - How will this answer the question

2 Design, Development and Evaluation

3 Design (14–15 pages when combined with development)

3.1 Development (14–15 pages when combined with design)

3.2 Results and Evaluation (11–12 pages including critical review)

4 Conclusions

4.1 Conclusions and Critical Review

References

Baxter, P., & Jack, S. (2008). Qualitative case study methodology: Study design and implementation for novice researchers. The qualitative report, 13(4), 544–559.

Blow, J. (2004). Game development: Harder than you think. Queue, 1(10), 28.

Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. Journal of game development, 1(1), 7–28.

Buro, M. (2004). Call for ai research in rts games. In Proceedings of the aaai-04 workshop on challenges (pp. 139–142). AAAI press.

Colledanchise, M., Marzinotto, A., & Ögren, P. (2014). Performance analysis of stochastic behavior trees. In 2014 ieee international conference on robotics and automation (icra) (pp. 3265–3272). IEEE.

Cui, X., & Shi, H. (2011). A*-based pathfinding in modern computer games. International Journal of C 11(1), 125–130.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische mathematik, 1(1), 269–271.

Diller, D. E., Ferguson, W., Leung, A. M., Benyo, B., & Foley, D. (2004). Behavior modeling in commercial games. In Proceedings of the 2004 conference on behavior representation in m (pp. 17–20).

Epic Games. (1998). Unreal Engine (Version 4.23.1). Retrieved November 30, 2019, from https://www.unrealengine.com/en-US/

Forbus, K. D., Mahoney, J. V., & Dill, K. (2002). How qualitative spatial reasoning can improve strategy game ais. IEEE Intelligent Systems, 17(4), 25–30.

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1976). An algorithm for finding best matches in logarithmic time. ACM Trans. Math. Software, 3(SLAC-PUB-1549-REV. 2), 209–226.

Gomilia, L. (2007). Simple and Fast Multimedia Library (SFML) (Version 2.5.1). Retrieved December 28, 2019, from https://sfml-dev.org

Graham, R., McCabe, H., & Sheridan, S. (2003). Pathfinding in computer games. The ITB Journal, 4(2), 6.

Harmon, V. (2002). An economic approach to goal-directed reasoning in an rts. AI Game Programming 402–410.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics, 4(2), 100–107.

Higgins, D. (2002). Generic a* pathfinding. AI Game Programming Wisdom, 114–121.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. Artificial intelligence, 27(1), 97–109.

Laird, J., & VanLent, M. (2001). Human-level ai's killer application: Interactive computer games. AI magazine, 22(2), 15–15.

Lantinga, S. (1998). Simple DirectMedia Layer (SDL) (Version 2.0.10). Retrieved December 28, 2019, from https://libsdl.org

Leigh, R., Louis, S. J., & Miles, C. (2007). Using a genetic algorithm to explore a*-like pathfinding algorithms. In 2007 ieee symposium on computational intelligence and games (pp. 72–79). IEEE.

Lester, P. (2005). A* pathfinding for beginners. online]. GameDev WebSite. http://www. gamedev. net

Lim, C.-U., Baumgarten, R., & Colton, S. (2010). Evolving behaviour trees for the commercial game defcon. In European conference on the applications of evolutionary computation (pp. 100–110). Springer.

Linietsky, J., & Manzur, A. (2014). Godot (Version 3.1.2). Retrieved December 28, 2019, from https://godotengine.org

Millington, I. (2019). Ai for games. CRC Press.

Monolith Productions. (2005). F.E.A.R.

Nareyek, A. (2004). Ai in computer games. Queue, 1(10), 58.

Newzoo. (2019). 2019 global games market per device and segment. Retrieved November 30, 2019, from https://newzoo.com/key-numbers/

Orkin, J. (2003). Applying goal-oriented action planning to games. AI game programming wisdom, 2, 217–228.

Orkin, J. (2006). Three states and a plan: The ai of fear. In Game developers conference (Vol. 2006, p. 4).

Pearl, J. (1984). Heuristics: Intelligent search strategies for computer problem solving.

Russell, S. J., & Norvig, P. (2016). Artificial intelligence: A modern approach. Malaysia; Pearson Education Limited,

Scott, B. (2002). The illusion of intelligence. AI game programming wisdom, 1, 16–20.

Shoulson, A., Garcia, F. M., Jones, M., Mead, R., & Badler, N. I. (2011). Parameterizing behavior trees. In International conference on motion in games (pp. 144–155). Springer.

Silicon Graphics. (1992). Open Graphics Library (OpenGL) (Version 4.7). Retrieved December 28, 2019, from https://opengl.org

Stanciu, P.-L., & PETRUŞEL, R. (2012). Implementing recommendation algorithms for decision making processes. Informatica Economica, 16(3).

Stentz, A. (1996). Map-based strategies for robot navigation in unknown environments. In Aaai spring symposium on planning with incomplete information for robot problems (pp. 110–116).

Stentz, A. (1997). Optimal and efficient path planning for partially known environments. In Intelligent unmanned ground vehicles (pp. 203–220). Springer.

Sweetser, P., & Wiles, J. (2002). Current ai in games: A review. Australian Journal of Intelligent Inform 8(1), 24–42.

Tozour, P. (2002). The evolution of game ai. AI game programming wisdom, 1, 3–15.

Unity Technologies. (2005). Unity Engine (Version 2019.2.14). Retrieved November 30, 2019, from https://unity.com

Weber, B. G., Mateas, M., & Jhala, A. (2011). Building human-level ai for real-time strategy games. In 2011 aaai fall symposium series.

Yap, P. (2002). Grid-based path-finding. In Conference of the canadian society for computational stud (pp. 44–55). Springer.