Fundamentals of Artificial Intelligence in Game Development

B.C. Bridger
Chris S. Groskopf

University of Georgia
Athens, Georgia
Contact: idlemind79@hotmail.com

Abstract
In effort to achieve higher levels of realism in game play many companies are beginning to put more emphases on the creation of engaging, challenging opponents which is necessitating the use of several algorithms from the field of artificial intelligence. This paper will review these techniques, their implementation, and discuss who and how they are being used within the game development industry.

INTRODUCTION

What makes a game fun? Intellectual stimulation? Realisticness? Its replay value? All of these are necessary conditions for a game to be successful, and all are areas in which artificial intelligence provides significant contributions. The goal of AI in game development is to create intelligent opponents, or agents, that can compete with the gamer in a "brains vs. brains" scenario in lieu of the "brains vs. brawn" situations so commonly encountered in older games. This necessitates a definition of intelligence, which for the scope of this paper will simply be agents that are realistic in appearance and action from the point of view of the gamer. Does this mean that an agent who scratches himself while he is idle appears more realistic than one that just stands there? While technically the answer to this question is "yes," what we are hoping to achieve with AI is an agent that can construct plans, execute these plans, and learn from the result. The reason that we are forced to use techniques from AI is that games produce a unique element to programs: uncertainty. We will begin our look at how games attempt to deal with this with by looking at search. This section is divided into two sub-sections: basics and advanced. Those who are familiar with search and common search strategies may wish to skip ahead to the advanced section.

SEARCH BASICS

Consider a game of chess. At the start of the game, there are 32 pieces and 64 positions on the board. It is conceivable that you could list all of the possible moves from this point. If you expanded that list to include all possibilities from those positions, you would be building an example of what is known as a search tree. Any problem that has a discrete number of possible states as well as rules governing passage from one state to another can be represented by a search tree and thus solved using an appropriate search algorithm. The problem is that generating a complete tree is rarely feasible due to computational and/or time constraints. Continuing with chess as our example, if you just considered all of the pieces that can occupy any position on the board (i.e. leave out the pawns and bishops) you would still be left with approximately 1.53 * 10^21 possible boards. Even if you assumed that you could always beat your opponent in twenty moves (a very short game) and attempted to draw a tree only forty moves deep, you would still end up with around 25^40 possible situations, or nodes (assuming a branching factor of 25). So what we need is a way to generate a tree as deep as possible given our constraints. The way we go about this is to expand nodes in some kind of reasonable manor. Here there are two possibilities, in a preset pattern (depth first or breadth first) or by intelligently guessing which is the best and expanding it first. The latter of these, heuristic search, is what we will concentrate on for reasons that will become apparent. We will use what is called a heuristic to assign desirability values to each node in our attempts to choose the best one. For clarity, we will concentrate on creating a path finding algorithm using the A* technique (respectively, this is the most common use and method of search in game development). Remember that all a search needs is a set of possible conditions and rules governing passage from one state to another. Lets assume that we are on a 5X5 grid and can move into any adjacent position (Increasing the field size or adding obstacles does not significantly impact the difficulty of the algorithm). We will also need to define an initial position as well as a goal, so lets choose 1,1 and 5,5. When our search starts, the first row will consist of all positions possible from 1,1, namely 1,2, 2,2, and 2,1. The second row will expand these three and consist of 18 nodes, 97 in the third, and so on. So to avoid having to build all of this, how will we choose which node to expand? There are two common ways of attributing values to a node. The first is the cost associated with the expansion of the node, and the second it a heuristic value. A* uses the sum of these two. In most instances, the cost of a node will be its depth. This attempts to force the algorithm to find the shallowest, or optimal, answer. But, you can manipulate the costs to suit your needs. Say, for instance that you wanted the resulting path to move on a northerly direction because you suspected that there were enemy troops to the south. You could simply assign lower costs to nodes that move in a northerly direction than those that did not. The second value, a heuristic, must be defined by the programmer. In two-dimensional pathfinders, the Manhattan distance is the preferred heuristic of many developers. This value consists of the sum of the distance of the current coordinates from those of the goal coordinates. More specifically, if your game doesn't allow diagonal movement, a MD heuristic function will look something like abs(dx)+abs(dy) , or max(abs(dx), abs(dy)) if it does. Now, remember that it is the sum of the cost and heuristic that A* uses to evaluate each node, and we are ready to implement our pathfinder. We will need to create two lists (or whichever linked data type you prefer). The first of these will be an open list consisting of nodes that have been generated but not explored. The second will possess all nodes explored. Note that these are lists of lists as each element in the sub-list contains the position, its value, and a parent pointer to the node to which it was expanded from. Now all that we need is a queuing function to choose the next node to expand based on values of those that have not been explored. Once the goal node is generated, we can simply follow the parent pointers back to find the path. Now that we are familiar with how A* works, we can explain why it is preferred. First, it is one of the most efficient heuristic searches available in terms of time and computational resources required. Second, it is guaranteed to find a solution if one exists, and moreover that solution will be optimal providing that the heuristic function never overestimates. We are also in a position to explain why use search at all. Instead of this complicated pathfinder, why not just continually move in the direction of the goal. To answer this question, consider a situation in which you must move away from the goal in order to reach it (i.e. when there is an obstacle between you and your destination). Using this type of algorithm, you will run into the obstacle and move back and forth infinitely. Simply stated, if the goal was to the west, you would run into the wall and continually move north, south, north, south, etc. because the program is only looking for the position that is closest to the goal. Rest assured that examples can be easily contrived that will fail other "cheap fixes." Now lets look at some other search algorithms. Most informed (heuristic) searches currently used are variations of the A* algorithm above. For example, iterative deepening A* (IDA*). On each iteration with this method, a depth first search is performed based on evaluations identical to A* until either a solution is found or an inputted function cost (cost + heuristic value) is reached. As would be expected, this algorithm has its advantages and disadvantages. IDA* was designed to allow search deeper into a tree using less memory and it does this well. However, in order to conserve memory, it "forgets" what it expanded after each iteration, only remembering the cost limit available. Thus, in trees where there exist redundant nodes, it is doomed to repeat itself. From this problem arose SMA* (simplified memory-bounded A*) which simply uses all available memory to search. It "forgets" nothing until it runs out of allotted memory at which point it drops nodes from the open list starting with those that appear least promising. We are now ready to move into some more advanced techniques.

ADVANCED SEARCH

For any informed search to provide an optimal solution, it must have three things, an admissible heuristic, complete knowledge of the domain that it is searching, and a clear definition of the problem. Moreover, heuristic searches are limited by the fact that if any new information is received, then they will essentially have to backtrack and start from scratch tying up valuable computational resources. We will examine the latter of these problems first. Originally designed for robotics, a new search method known as dynamic A*, or D*, has emerged to deal with the problem of unknown, partially known, or changing environments. This is especially applicable to game development because it removes the need to give the agent the unfair advantage of complete spatial knowledge of its environment. We will concentrate on a D* path finding algorithm as this is the most applicable example. Consider the event that you are traversing a tunnel only to find an unexpected obstacle. Instead of backtracking in attempts to find other solutions and ignoring the possibility of a path in an unknown region using existing desirability function node values, D* allows you to reassign values to all unexplored nodes based on the new information. The implementation resembles A* in that you still have an open and closed set of nodes and a PROCESS_STATE function, but adds a new function, MODIFY_COST. This enables the algorithm to raise and lower cost functions on nodes in the open list if their path cost as determined in PROCESS_STATE changes. More information on D*, including implementation, is available at http://www.frc.ri.cmu.edu/~axs.

A relatively new area of AI, evolutionary programming, is providing us with ways to handle some of our other problems such as lack of heuristics, ill-defined problems, and vast search spaces. Genetic algorithms take a more biological approach to problem solving by creating a digital DNA representation for a problem and evolving it until it reaches an optimal, or near optimal solution. This evolution process is achieved by taking a set (population) of potential solutions (individuals) and evolving them through a series of genetic operators. In human DNA, each individual is represented by sub-strings built of an alphabet consisting of A, G, T, and C (adenine, guanine, thymine, and cytosine). Problem representation in GAs mimic this representation, with the exception that the alphabet used is most often binary. Once an initial population is created according to these specifications, individuals are filtered through a fitness function to determine their quality, and if this value is not satisfactory, then the data is evolved again. This Darwinian type of evolution occurs by a process of reproduction. Two strings with the highest fitness values "mate". This "natural selection" ensures that successful strings (ones with a high fitness value) survive the evolutionary process while genes with poor fitness values eventually perish. The mating process of the two chromosomes is achieved by swapping segments which are randomly chosen in a process collectively called "crossover." For example, consider two parent individuals with genes numbered 1 to X. The crossover function would choose a randomly generated number between one and X, lets say 25. Then, two offspring individuals would be created, one with genes numbered 1 to 25 from the first parent over 26 to X from the second; and the other would be the first 25 of the second parent over the rest of the first. Thus, a GA can take an initial population of solutions and mutate them with (hopefully) better results each time. When does evolution stop? The three most common methods are when the variation from one generation to the next reaches a predefined level of stability, after a chromosome is created that reaches a specified fitness value, or simply upon completion of a pre-specified number of evolutions. The downside to genetic algorithms is that when they first run, they rely largely on sheer luck as it can take several evolutions before significant fitness vales are assigned. However, GAs are extremely versatile. They can be applied to almost any problem, including the ability to learn.

MACHINE LEARNING

As more and more developers turn their efforts to the development of artificial life, it is becoming obvious that one of the fundamental qualities of intelligence is the ability to learn. Knowledge acquisition and representation is currently predominately being attempted with adaptive technologies such as genetic algorithms and neural networks. Following the previous section, it should be fairly obvious how a GA can be used for learning. The qualities which proved successful (i.e. speed) are retained, and those that are not are minimized or dropped (like over aggressiveness) through the evolutionary process. Neural nets, much like GAs, mimic biological learning, or at least what is known about it. In the human brain, there are basic cells called neurons that are composed of a core, dendrites for receiving information, and axons which serve to pass information to other neurons. In each neuron is an electro-chemical sensor to detect the strength of incoming charges. If the input is strong enough, then the neuron will fire to all cells to which it is connected; otherwise no output occurs and the charge dissipates. It is thought that learning takes place by modification of the connections between neurons so that certain input only channels through specified paths. This is where neural nets differ; they have an unchangeable structure, or a predefined number of units that can act as neurons and a pre-specified number of links. This restraint is overcome by attaching weighted values to the connections, so although connections can never be created or destroyed, their importance and the signal they receive can be increased or reduced. Signal qualities can also be manipulated inside the cell by using a sigmoid function to determine the firing point in lieu of a step function. Thus, we can continue to mimic the "wet ware" by adjusting these values based upon the desired output and the actual output of the network. However, this type of network requires a mediator to judge and evaluate its performance. Teuvo Kohonen pioneered the field of unsupervised learning in 1982 with the introduction of the Kohonen network. The difference in this system was that the output neurons compete with each other much like in a genetic algorithm. Although this process goes beyond the scope of this paper, it will suffice to say that weight to the "winning" connection is increased so that when a similar input is received it will be likely to go to this neuron again.

Another method of learning worthy of mention was first used in the game Galapagos by Anark. For their main character, they developed an agent that learned from Non-stationary Entropic Reduction Mapping (NERM). NERM allows Mendel, the agent, to receive feedback from his environment, which he uses to adjust his behavior. He is self-organizing, meaning that he receives no justification for the feedback; he simply acts to minimize the negative, much like a pet animal. What this means is he has the ability to learn from the environment even if he is not interacting with the user. Creatures, by Millennium Interactive, is an organic life simulator that perhaps best demonstrates the potential of GA's and neural nets in the gaming industry. In the game, creatures, called Norns, are born and raised by the gamer. While growing up, Norns must learn how to speak, interact with others, entertain themselves, etc. The GA's come into play when they breed. Every aspect of a Norn is represented in its DNA, which is passed down in reproduction (several web sites have sprung up to allow this to be done online with other gamers). Creator Toby Simpson recently received an email about a Norn which appeared more intelligent that others. Proving the flexibility of GA's, he examined it to find that it had spontaneously developed additional "lobes" for the AI to work with! So, at this point, we have demonstrated how artificial intelligence can create gaming agents that can reason and learn. Our other focus was on realisticness. The game development industry is beginning to put emphasis on individual AI, such as artificial emotion, independent thought, and other like topics.

ARTIFICIAL LIFE

Even after we have succeeded in creating a sufficiently intelligent agent, our job as game developers is not complete. If you will remember, one of our original goals was to create a "realistic" agent and environment. This is a-life, an emulation of biological behavior, to which several areas contribute. Arguably, emotion is a predominant characteristic that makes for realistic behavior in lieu of robotic type actions. Making agents unique used to mean only appearance, for example making a character in a basketball game look like Michael Jordan. However, we are beginning to see a change with the emergence of a field known as artificial emotion. Leading this field is Ian Wilson, who has divided emotions into three categories: momentary, mood, and personality, which take precedence in that order. The latter of these is the underlying feature that will typically be used in decision making. One method that is currently being used works as follows: the actions outlined above in the paper are used to generate not one, but a set of possible actions. Then, each agent compares the possibilities to its personality and chooses the action that is most fitting to him/her. It is important to note that the possibilities are weighted so that the majority of agents will follow the action given, or preferred. Consider the event that the gamer tells his team to charge the enemy. The agents could be presented with a set of possible weighted (so that most would follow the instruction) choices that included charging, staying idle, and running. Thus, an agent that was excessively passive could sit still, one that was scared could run, while the majority charged as told. While it might agitate the gamer to see some of his military run from battle when told to charge, all would agree that this is what would happen in real life. The other two categories of emotion, mood and momentary emotion, must be triggered. Thus, like humans, something can spark an action to an agent that it normally would not do. For example, in our war game, an agent with an aggressive personality who is following orders to sit in a bunker may become impatient and agitated and thus charge the line because he had been idle too long. According to developer Steve Woodcock, this type of realism is quickly being brought to the forefront of game AI. Currently, it is being implemented in one of two ways. The first is as implemented in games like Microsoft's Close Combat. This is a war simulation game that provides each GI an individual personality much like described above. The second type is best demonstrated by P.F. Magic's Dogz, and Catz series in which the gamer creates an autonomous agent by defining amounts of certain traits, and watching the agent evolve in its environment. These games go as far as to give each agent body language, tempers, etc. Another approach to implementing AE is being pioneered by Electronic Arts with The Sims. This game is similar to Dogz and Catz in that the gamer develops an agent and then watches it grow in a simulated environment. The difference in the implementation is the use of what developer Will Wright calls a "behavioral engine" in which actions and behaviors associated with an object are coded into the object itself. So, a football will contain code that states how it is to be used and when one would want to use it. The problem with AE is twofold. First, it is computationally expensive and many companies are not providing sufficient resources for any true advancement to be made. Secondly, with shoot 'em up games and the like dominating the market, there has been no real need. Who wants to see a timid demon while playing Doom? Of course, a-life is not merely concerned with the individual; it must necessarily be interested in the coexistence of all agents in the environment. In this area, it has made some remarkable advances. The simplest of these is flocking, in which several characters can engage in some group action even if they do not know why they are doing so. This is comparable to a swarm of bees or a flock of birds. However, perhaps the most intriguing advances in a-life as it relates to games is inter-agent communication - agents working together to achieve a common goal. Consider a shooter game in which a monster could realize that he needed help and go get a buddy to assist him. Or, imagine playing Doom and a monster organizes all of the others against you! While these ideas are promising, work in this area is insufficient to predict future possibilities.

FUZZY LOGIC

What ties all of these principles together is fuzzy logic. Look around the room. Do you see any tall people? Any thick books? You probably answered yes to at least one of these, so you must know exactly what height is "tall", or how many pages constitute a "thick" book. Obviously this is absurd. Fuzzy logic is essentially a multi-valued logic system that permits intermediate values to be defined between conventional evaluations like yes/no, true/false, black/white, etc. It allows notions like pretty hot and kind of cold to be formulated mathematically and processed. This is achieved by representing values in something that more resembles a bell-shaped graph in lieu of a step function. For example, one cannot represent "young" on a step function because to do so necessitates knowing the day in which one ceases to be young. What we need, and is provided by fuzzy logic, is a function that shows that a person is less young on his twentieth birthday than he was the day before, or the day before that. Thus fuzzy logic, as opposed to conventional logic, proves that a big rat is not the same size as a big elephant. Moreover, it retains the use of conventional ideals like negation, conjunction, disjunction, etc. and the truth-values that they imply. These types of distinctions are applicable to our intelligent agent because it allows it to realize that there is not always a "right" decision. Similarly, it allows for more freedom as problems and states can be represented at intermediate values. An interesting company, louderthanabomb (louderthanabomb.com) has designed a fuzzy logic editor specifically for game design. More information on this topic can be found at The Fuzzy Logic Laboratorium (www.flll.uni-linz.ac.at).

PLANNING

In order for the agent to effectively accomplish anything, it must first construct a plan. Most planning algorithms are based on the STRIPS model which stems from situation calculus. In STRIPS, each state is represented by a predicate, for example location(Agent1,Tunnel1); and the state of the environment is represented by a conjunction of these predicates. Thus to construct a plan, a sequence of states must be created. Each action consists of a precondition, a post condition, (or effect) and a modify_condition function. Obviously, the precondition defines what states must exist to execute the action, and the post condition defines the situation that results. The modify_conditions function changes the current world state to reflect the changes resulting from the action. For example, if part of the world condition was location(X,Y) and the action to move from Y to Z was performed, then not only would location(X,Z) need to be added to the state of the environment, but the previous condition would need to be removed to prevent a conflict. We will consider a world with three actions: move_to(X,Y), ensure_loaded(X,Z), fire(X,Z,A). In a situation with an initial state if at(A1, R2), at(A2,R1); if agent A1 decided to fire a weapon at A2, the final plan would look similar to: move_to(A1,R1), ensure_loaded(A1,Gun), fire(A1,Gun,A2). The steps to develop this plan go beyond the scope of this paper, but it will suffice to say that a plan is generated that ensures no conflict between the pre and post conditions. Then, the variables are unified to create an executable plan.

CUSTOM GAMES

A recent trend in many shooter and war simulation games has been to allow the user to create his or her own agents through extensible AI, which allows the gamer to actually manipulate the source code. Thus, a player can manipulate existing agents in a game, or create entirely new ones. This was first attempted by ID with their war simulator Quake. Packaged with the game was their scripting language, Quake C, which allowed gamers to create both new levels and creatures, and even define their actions in certain situations. As would be expected, this spawned an entire subculture of "bot builders" as they came to be known. Extensible AI has been taken even further by companies such as GT Interactive and Grimmware. These two companies are actually exposing the game's API (Application Program Interface) to the player, allowing for full manipulation of all included characters.

CONCLUSION

According to surveys taken by developer Steve Woodcock in 1997, around 24% of game development companies were employing full time AI programmers who received approximately 5% of total processor time. In the same study one year later, the numbers were up to 46% and 10%, respectively. This definitively proves that AI is, in a large part, the future of the gaming industry. This future will rely largely on efficiently incorporating of all the techniques mentioned in this paper; each of which involves moving away from a rule-based approach to AI. The added bonus of this progression is that it greatly decreases predictability. Autonomous agents are surprisingly spontaneous, especially when they are able to mutate on their own through biological technologies. For now, we need to realize that if we play a game that claims to use AI, and find it too easy, or not realistic enough; it is not the fault of the AI, but rather the developers. We do not need to discredit the field (as some have suggested) simply because some of its implementations fail. Aside from this type of critique, another problem which AI is beginning to overcome is lack of development time. As many of the AI routines cannot be developed until late in the game development process, developers have been traditionally expected to turn out large amounts work at the last minute with less than ideal results. However, with the gradual realization of the importance of this genre, developers are beginning to see a change. By allotting AI sufficient resources in games, and providing developers with the necessary time needed in the development phase, the future of this field appears very promising.

BIBLIOGRAPHY

1.www.gameai.com
2.www.gamedev.net
3.www.gamasutra.com
4.www.frc.ri.cmu.edu/~axs/
5.www-cs-students.stanford.edu/~amitp/gameprog.html
h5.ttp://forum.swarthmore.edu/~jay/learn-game/index.html
7.www.ide.hk-r.se/~pdv/mypublications.html
8.Russell, S. and Norvig, P (1995) Artificial Intelligence A Modern Approach, Prentice Hall
9.www.flll.uni-linz.ac.at
10.www.louderthanabomb.com
11.www.alife.com
12.http//ai.about.com/compute/ai/library/weekly/aa070598.htm
13.www.gamasutra.com/features/19990507/artificial%5Femotion%5F01.htm
14.http://ai.about.com/compute/ai/library/weekly/aa121598.htm
15.www.ma.umist.ac.uk/dsumpter/beesim/