2015

Literature Review

GOAP

SAM MCKAY, ILLIAN GEORGIEV, AARON SWISS-HAMLET, JOSE MANUEL DIEZ AND VLAD-EUGEN TANASE

Contents

Strips

In order to fully implement a goal orientated action planning system we have researched heavily into the STRIPS methodology, this is one of the original methods of doing this and with slight updates can be utilised within our project.

Strips uses a regressive search algorithm to create an action plan, this works by have a desired state or goal and working backwards to fulfil the pre-requisites of the goal. For example, if the goal is to not be cold then a pre-requisite could be too be near a fire, in order to achieve that it would need wood, to get wood it has cut down a tree and to do that it needs to find an axe. So in this case its plan would be:

Find axe
Cut down tree
Make fire
Sit near fire

These goals can be in the form of a needs system i.e. hunger, energy, warmth, thirst and so on. These would each have a weight which would indicate their importance and the AI would prioritise those with a greater weight. This means that the search algorithm would need to recalculate when a step in its plan has changed in case the goal has changed, this however could make it more efficient in case it has discovered a quicker way of achieving the goal since it calculated the last plan.

Steering

Steering is useful for simplistic applications while still giving the impression of complex actions. The main implementation method involves vectors(math context). Steering model can be applied to collision avoidance, as well as following and tracking other entities and paths.

Steering has many different implementations. From flocking algorithms such as Boids to path tracking, collision avoidance, containment etc. Steering implements vectors to determine the direction in which an AI agent should move. These vectors are subject to change based on the implementation of steering and the current world state.
A good implementation of steering is Anti Gravity movement and Vector Field movement. Both methods work in roughly the same manner. The AI agent has a direction/steer vector which it is trying to keep to.

Vector field/Flow field uses objects around the world and the goal position to create a grid of direction vectors which alter the original direction vector of the AI to have it avoid obstacles. This is useful for avoiding big and difficult to go through terrain.
Vector field method is somewhat limited by the resolution of the grid. An ideal solution is to have a good midpoint. Another possible issue with this method is moving objects.
While static terrain requires only one calculation in the case the grid resolution is high a big moving object that covers multiple grid squares will have to affect all the cells around its perimeter and even a bit beyond.

Similar to "Obstacle avoidance" described by Craig Reynolds, Anti Gravity works on roughly the same principle as Vector Fields however instead of a grid of vectors the AI agent tracks the distance between each object it can collide with and applies a steering vector opposite the direction of that object.
This approach is also similar to what is used in the Boids AI model developed by Craig Reynolds. Except in this case all other "entities" are static and only the Separation and Alignment rules are taken into account.
Path following allows for the use of predetermined paths in the game world. The agent sill uses vectors to steer and implements the use of a future position. The future position usually will have an offset from the path this offset is used to correct the AI direction vector. While not 100% accurate to the path the AI will stay within a certain distance of it.
Steering can also be implemented in the context of moving entities. Following, intercepting, queuing and running away from. These behaviors work on the same principles as the ones listed above with the change that the entities/objects used as a reference for the calculations are not static.

Combining different steering implementations can give the AI agent multiple ways to navigate around the world based on information supplied to it by its sensors.

Pathfinding

Unlike steering method pathfinding can be used to solve complex problems such leave a maze and find the shortest distance between the start and end points (in case there are obsticles).
Pathfinding has several limitations and drawbacks:
It requires a comprehensive grid/graph with well defined connections between cells/nodes
Movement between those is not necessarily described by the algorithm used and usually has to be implemented separately.
Algorithms used for pathfinding are very resource intensive and can take a very long time to compute.
If a path is obstructed by a dynamic object the AI will not register that and possibly get stuck.
The two main approaches are directed and undirected pathfinding.
Undirected pathfinding is not widely implemented and it uses one of two main algorithms, Breadth-first and Depth-first. They both search though the connected nodes until they find a solution with no regard as to the difficulty, time it might take to get to the final goal and while similar the two algorithms work in almost the opposite way.
Breadth-first expands all nodes, at one level, revealing all the nodes they are connected to. Only when all nodes from the current level are expanded will it move down to the next.
This is useful where the node hierarchy has multiple solutions are different depth levels. Breadth-first will find the solution that takes the least steps(node expansions).
As the name with both algorithms suggests Depth-first does not focus on expanding all nodes originating from a single node before moving on. It looks at the children nodes until it reaches a node which has no new ones originating from it. This method is much more efficient than Breadth-first in situations where you have a very uniformly shaped tree of nodes with a lot of solutions.
The directed pathfinding is more common in games since it provides the shortest/fastest route to a target location. However it is much more time consuming to compute than the undirected methods.
Directed algorithms work on a cost system described by the developer. Costs or weights can be anything from the physical distance between nodes in the game, to estimated time of completion or even arbitrary values put in by the programmers/designers to “guide” the algorithm in a certain direction.
Like with undirected pathfinding there are two main strategies to finding the shortest: uniform search and heuristic search.
Uniform search gives an optimal result picking the easiest route possible every time however it is very inefficient and may take a very long time to complete.
As the name suggests heuristic search uses a heuristic to determine the route. This is the much more efficient method however but it does not search through all possible paths and not always is the final solution the most optimal.
The two most used algorithms for directed pathfinding are Dijkstra(uniform search) and A*(combines uniform and heuristic search). Other algorithms exists which are mostly variations of the ones already mentioned.
These variations or evolutions are developed in order to solve some of the issues that are present whenever pathfinding is involved. A* itself is an evolution of Dijkstra.
PRA* is an evolution of the A* algorithm. It speeds up the A* algorithm by first generating a solution for a more abstract version of the search space. This way the A* algorithm has to look through a fraction of the original node/grid mesh.

D*(dynamic A*)this algorithm takes into account changes in the cost of a path after it has been calculated and the AI has been set in motion. This solves the problem with dynamic objects but adds a lot more strain on the CPU in order to accomplish this.

Variations of the A* algorithm go on with each variation improving the original in some way. Here are some of them and how they are influencing the original they are built upon.

Dijkstra – finds all possible paths through a graph/grid.

A* - Uses a heuristic to find the shortest/most optimal path.

IDA* - Iterative Deepening A*

This version tries to speed up A* starting with an approximate answer. With each iteration it expands more and more nodes and tries to match the overall weight/cost to the initial estimate. Somewhat based on the Depth Search algorithm for looking through trees.

D*/LPA* - Dynamic A*/ Lifelong Planning A*

These versions solve the problem of dynamic environments in exchange for requiring a lot of space. These versions keep all details of the last computed path and match it to the current graph/grid state to determine if a recomputed is needed.

PRA* - Partial Refinement A*

Creates a “lower resolution” version of the original graph/grid thus speeding up the algorithm.

LRTA* - Learning in Real-Time A*

substantially lowers the first move delay present in standard A*. It accomplishes this by first doing a very small scale calculation to determine the first step and then it finds paths to the end goal which may be suboptimal. Finding an optimal path is much more expensive compared to A*

Pathfinding gives the best solutions however it can be very resource intensive based on the implementation. Current algorithms and their variants try and balance speed, resource requirement and quality of result. A specific implementation can be chosen based on the needs of the AI agent.

Map Generation

Research conducted into random map generation for 2D games.
Easiest approach is to create tiles, however other options are Height map, Perlin Noise but those are more complicated than what we require for the project and as such we shall be sticking with the tiled approach. This will allow to create plenty of different maps in which to test our AI, we can also utilise the random generator for the creation of objects throughout the scene allowing us to spend more time upon the action planning section.

The maps will need to be exported into an XML format that can be read with a TMX reader that can be used with SFML, the advantage of having them in XML format is that we can also edit them manually should we need to.