AI Tutorial 2:
Path-Finding and Crowd Management

Summary

The concept of optimal-path calculation for a node graph is introduced and explained. This concept is mapped to the area of game navigation, and then extended into the exploration of management of large numbers of entities.

New Concepts

Graph Search, The Game Environment as a Graph, Minimum-Cost Path-Finding, Navigation for Groups, Flocking, Embedded Navigation Data

Introduction

Path-finding is the process of finding a route through an environment. Once calculated, that route can then be used by an AI agent to move to a target point, or it can be communicated to the player to show the recommended way to proceed. Some examples of path-finding in games are:

- The ghosts eyes heading back to base after Pac-Man has eaten them.
- The bad guys in Deus Ex or Metal Gear Solid heading toward the place where an alarm has been tripped, or the player has been spotted.
- Yorda running to the player when called for by Ico.
- The golden thread showing where to go next in Fable.
- The route plotted on the map to the desired destination in Red Dead Redemption.

In this tutorial we will discuss a path-finding algorithm, and use it to provide routes through a simple two-dimensional environment. There are three main steps in providing path-finding technology for AI agents in a game:

- Represent the environment as a graph of small navigable units or nodes.
- Find a route connecting a series of these nodes from the starting point to the target location.
- Move the AI agent along that route convincingly.

The algorithm which we will consider is known as A*. This is probably the most widely used path-finding algorithm within games development. In order to understand what the algorithm is doing, we shall first consider the concept of our game environment as a node graph. Following that, we shall introduce the algorithm, before moving on to practical approaches to moving large numbers of entities.

Representation of the Environment

Let us consider the idea of a game map. We can normally identify key elements of the map, for navigation purposes, in terms of a vertex/node. This simplest approach is to subdivide our environment into equally sized squares/hexagons. This allows us to algorithmically determine edges, rather than having to plot them. A more sophisticated and context-related approach is to identify junctions on our map that connect roads. The roads themselves then become edges, and our graph becomes smaller (because we only consider junctions, not every element of the environment, including the bits which aren't passable).

The most sophisticated approach commonly employed in game development is the navigation mesh, which blends these concepts - in this approach, nodes are regions, rather than points, and their shape is drawn from the environment itself, rather than forced (as in the grid case). This has the benefits of the latter approach (smaller graph size) and allows for more advanced approaches to representing different types of terrain within our environment.

Whichever approach we take, we need to represent the nodes of our environment in some fashion. See the example struct `mapNode`.

```
1  struct mapNode {
2     int nodeID;
3     int[] connectedNodes;
4     Vector3 position;
5     int terrainType;
6     int coverType;
7     bool passable;
8     bool isOnFire;
9     bool isFullOfDeathOhTheHumanity
10    ...
11 };
```

mapNode

As illustrated, there's a lot of data we can bake down into the graph which can describe the properties of a given location. This information isn't merely useful in path-planning, but can also inform agent behaviours as discussed in an earlier tutorial (e.g., an NPC of class Fireman pathing across a node with isOnFire = true could trigger his 'put out the damn fire' behaviour).

For the purposes of navigation, however, the essential elements of a node structure/class resemble:

- Unique Node ID
- List of Nodes to which the node is connected

- Position
- Something which lets us know if the node is passable

The reason this isn't definitive largely comes down to the fact you can represent your graph any way you choose, so long as you can infer the important data. For example, if representing a chessboard, you can infer that (unless the square is the first or last square on a given row or column), it is connected to the node to its right, the node to its left, the node above, or the node below - this allows you to bypass holding an array of connected nodes to explore. Similarly, you don't need to store whether or not a node is 'passable' if your graph dynamically deletes impassable nodes from the connected node list - and so on, and so forth.

Now we have an idea of the manner in which we represent our environment on a graph, we can look at an illustrative example (see Figure 1). For this tutorial we will use a grid of squares to represent the environment. Each node then has eight directly-connected nodes (ie the eight squares surrounding it). As we are modelling the problem as a square grid, we have reduced our search area to a simple two dimensional array.

Each item in the array represents one of the squares on the grid (a node), and its status is recorded as passable or impassable. As you can probably tell, this permits us to functionally reduce the complexity of our node structure to a simple 'isPassable' boolean, and an x and y coordinate (since we can infer connections).

The path is found by figuring out which squares we should take to get from A to B. Once the path is found, our agent moves from the centre of one square to the centre of the next until the target is reached.

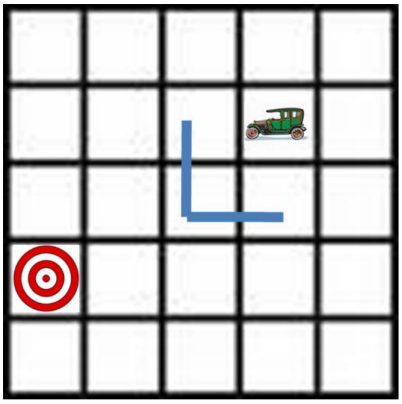

Figure 1: Example Scenario

The first thing to do is to set up the array, which represents the grid of squares. The data is read from a file. It would be easy to hardcode this data, as there isnt much of it, but it is worthwhile using a data-driven approach, so that the same executable can be used to resolve many path-finding problems (so long as they map to this geometry).

The data file contains the size of the grid to be used (number of rows and number of columns), followed by the grid of data. A navigable node is marked with a 0, an unnavigable node with a 1, and the start and end points marked with A and B respectively. An example is included as Figure 2.

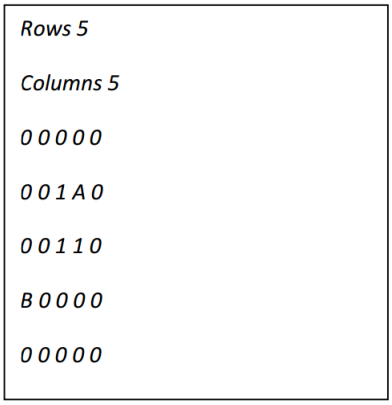

Figure 2: Example Data File

The A* Algorithm

Having looked at the way we can encode our graph, let's have a look at the A* algorithm in detail. The formula which underpins A* is

$$f = g + h \tag{1}$$

where:

- f is the total cost for the node under consideration
- g is the movement cost from the starting point to the node under consideration, following the path generated to get there
- h is the estimated movement cost from the node under consideration to the final destination, point B. This is referred to as the heuristic, as it is a guess. We dont know the actual distance until we find the path, because obstacles can be in the way.

Note: We use simple calculations for g and h (and, consequently, f) in this tutorial - costs are uniform, and the heuristic is the Manhattan distance.

A more complex graph (one which doesn't just uniformly distribute nodes, or which has different terrain types) might employ more complex equations (e.g. an additional weighting factor to the cost of traversing a muddy node), but in so doing it can make AI more sophisticated in an elegant fashion (e.g. instead of a clunky FSM telling them to stick to cover, a weighting factor which favours covered environments will lead them to path through those locations instinctively).

Characteristics of A*

A* is a best-first search. What this means is that the heuristic permits the algorithm to explore nodes which are most likely to give the optimal solution first, and the algorithm only explores other nodes if those 'promising' solutions turn out to be inefficient. This can happen as a function of graph geometry, especially in game environments. Aside from the formula above, an A* implementation normally maintains two dynamic lists: the *Open List* and the *Closed List*.

The Open List is a list of nodes of which the algorithm is aware, but which have not yet been explored: this means that they have a computed f-value, but their connected nodes have not yet been considered. When the algorithm begins, the only node on the Open List is the start node.

The Closed List is a list of nodes which the algorithm has explored. All nodes in the Closed List were originally in the Open List, before being moved to the Closed List. The final path will always be constructed from nodes on the Closed List.

Our path is generated by repeatedly going through the Open List and choosing the node with the lowest f-value. The connections of that node (its children) are then added to the Open List, and an f-value calculated for them, before the parent node is pushed onto the Closed List as a candidate for the final path. Algorithm 1 explains this in detail.

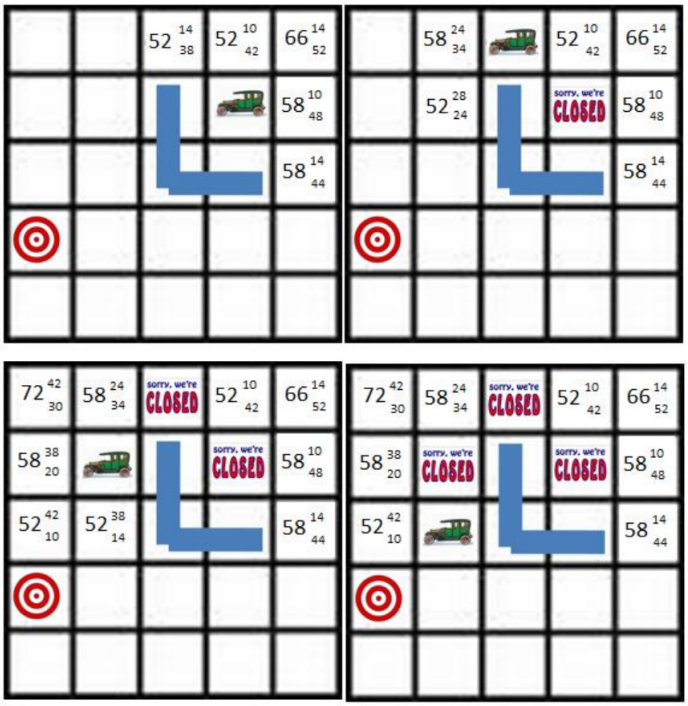

Figure 3: Solution

For the purposes of the example scenario (Figure 1), the cost of moving from one node to its neighbour is 10 for a vertical or horizontal neighbour, and 14 for a diagonal neighbour. It would be more accurate to use 1.0 and 1.4121 (i.e. the square root of 2.0); however, as will be discussed at the end of this tutorial, path-finding is very processor-intensive so using integers can lead to a significant optimisation over using floating point arithmetic, dependant on hardware. Figure 3 illustrates how our system employs the above algorithm to solve the scenario.

Following the Path

Once the path through the nodes has been determined, it is likely that a character or vehicle will need to use that path. In our example, the path is simply a set of straight lines connecting the centre of each grid-square along the path. Following that path precisely will result in very robotic movement that may be okay for Wall-E or a Dalek, but characters and vehicles dont move so mechanically. Most games require a much smoother path-following behaviour.

There are a number of ways of achieving this. A simple method is to take the calculated path and use that as a set of targets for the character to head towards. When the character gets sufficiently close to the target waypoint, it will start to be influenced by the following waypoint and gradually change its direction of movement toward that. This will smooth out the corners of the path.

Another method is to use the waypoints to build a smooth spline, and have the character progress along that. A spline is a curved line connecting a number of points. Depending on the settings used in calculating the spline, the path may not connect the exact positions of the waypoints, but it will follow the general shape.

Once a smooth route has been determined, the character or vehicle must move along it convincingly. Vehicle games will usually use the vehicles handling model and physics engine to drive the

Algorithm 1 The A* Algorithm

```
procedure INITIALIZATION(A, B, Open List, Closed List)
    Let A be the Start Node
    Let B be the Goal Node
    Compute g and h for A                ▷ g will normally be 0, and h will depend on your heuristic
    Assign f to A, where f = g + h
    Add A to the Open List               ▷ At this point, A will be the only node on the Open List
end procedure

procedure A* SEARCH LOOP(Open List, Closed List, Node Graph, B)
    Let P be the best node on the Open List                          ▷ Node of lowest f-value
    while Open List is not empty do
        if P = B then
            Go to Generate Path                                      ▷ If P = B, we have a solution
        else
            Let n be the number of nodes connected to P
            for i = 0; i = n; i++ do
                Let Q be the i^th Node connected to P                ▷ Data for Q is taken from Node Graph
                Compute g and h for Q
                Assign f to Q, where f = g + h
                if Q is on the Open List and f(Q)_new ≥ f(Q)_old then
                    Do nothing                                       ▷ We already have a path from A to Q which is cheaper
                else if Q is on the Open List and f(Q)_new < f(Q)_old then
                    Set parent of Q on the Open List to P            ▷ If this happens, our heuristic is broken
                else
                    Add Q to the Open List with parent P
                end if
            end for
            Pop P off the Open List
            Push P onto the Closed List                              ▷ P is a candidate for the final path
            Let P be the new best node on the Open List
        end if
    end while          ▷ If the Open List is empty, all nodes have been explored and no path exists
end procedure

procedure GENERATE PATH(Closed List, Path, A, B)
    Let R = B
    while R ≠ A do
        Add R to Path
        Let S be the parent of R
        R = S
    end while
    Add A to Path                        ▷ At this point, the child of A (first step) is already on the Path
end procedure
```

vehicle in effect providing steering, acceleration and brake inputs to the handling model as though a player were driving it. Characters must walk or run along the path, so the appropriate animation must be played, and the animation update synched to the movement speed, so that each foot appears to be static on the ground, and not moonwalking.

Of course, when following a smoothed-out version of the path, there could well be obstacles introduced as corners are cut off. If this happens then the characters collision detection and response routines come into play. It may be adequate to nudge the character along the path using the collision routines, or it may be more appropriate to instigate a path-finding algorithm at a much higher level of detail. It may be easiest to remodel the environment to remove the snags.

Computational Cost

Path-finding can be an expensive business, in terms of both memory and processing power. For the purposes of this tutorial, we have concentrated on a simple two-dimensional environment. The algorithms discussed are applicable to the complex environments that are developed for games on all platforms, but bear in mind that the memory and processor requirements scale up with the complexity and size of the environment.

The first stage (representing the environment as a set of nodes) is likely to lead to a large memory footprint. If the environment is static then this can be pre-calculated and loaded in as part of the world data, so there is no processor cost at run-time. However, if the environment is dynamic then the node graph will need to be updated while the game is running, which of course adds extra work for the processors. Some examples of dynamic environments are a wall that can be blown up to create a new path, or a street that can be blocked by the player parking a truck across it. If the game is set in an open world, which is spooled in from disk as the player traverses it, then the local node data should also be spooled in, which adds further complication to the memory requirements of the path-finding system.

The second stage (finding a route through the nodes) is likely to be very processor intensive, as there may well be a high number of nodes, and possible routes, to churn through in order to find the optimal path. The good news is that path-finding calculations are unlikely to be sufficiently crucial to need to be completed for every frame of the game (in the way that rendering and physics calculations need to be), so the cost can and should be spread over a number of frames. There could also be multiple agents all looking for a path at the same time, so again the cost needs to be spread across multiple frames, with the most urgent agents being attended to first (typically the one that the player is most likely to see up close).

As path-finding is such an expensive business, it is always worth considering whether or not it is actually required for the game that is being developed. There may well be cheaper alternatives that are suited to the particular game design being worked on. For example, if the game design requires a heavily scripted set-piece, it may be more appropriate to attach the agents to pre-set paths.

Some games also employ more than one layer of path-finding. The world could be divided into large nodes, for longer distance routes, but each local node could be further broken down into smaller nodes for shorter term path-finding. Often the higher level routes are pre-computed an example may be an open city driving game where the routes between sections of city are pre-computed (probably using freeways), but the local detail is worked out as the game requires it (e.g., using alleyways and pedestrian areas).

String-Pulling

We've talked a little about splines as a way of navigating our A* path, which is often appropriate as a post-processing step for believable navigation, but it isn't the only - or, indeed, best - way of making our entities move appropriately through our environment.

Consider a scenario where our agent is moving across a three-by-three grid obstructed at the centre, as shown in the figure below, where G denotes our goal node:

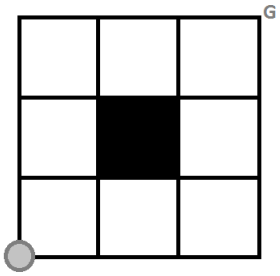

Figure 4: Example Scenario

A standard A* approach, allowing for diagonal motion, would produce the path shown in Figure 5.

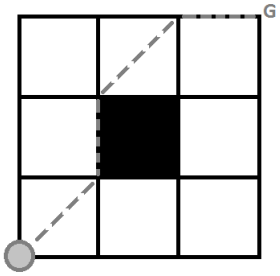

Figure 5: A* Solution to Scenario

But this path is far less natural than a path which leads in a straight diagonal to the top left corner of the obstruction, and then a straight diagonal from that corner to the goal, as shown in Figure 6.

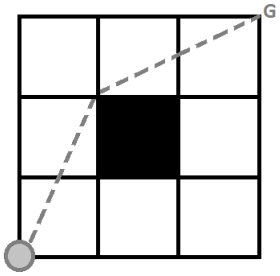

Figure 6: String-Pulling Solution to Scenario

So the question for us is how do we go about creating this second, obviously superior path? The first part of the answer lies in defining our vertices as the corners of the grid, rather than the centres. This doesn't really increase the size of our graph, if we think about it.

Next, we need to think about the manner in which A* works. Currently, we define the edges between our vertices when we first create the map. But what if we allowed ourselves to create edges of our own? For example, Figure 7 indicates with Xs which nodes our start node is connected to by default. In order to obtain the path we want, there needs to be an additional edge created, which attaching the top left hand corner of the obstruction (the Star Node) to our Start Node.

In order to create this edge, we leverage what we already know about computer graphics, and extend our A* algorithm. Consider how A* computes our G-value for a given node - we look at the G-value of the considered node's parent, and then add the cost of moving from the parent to the node. But what if we looked at the parent of our node's parent? This is one approach to string-pulling.

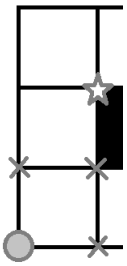

Figure 7: Connections to our Start Node

When we first explore a node, unless it is the start node, it has a parent - it has to, since we've reached the point where we're considering it. But the shortest distance we can move (and the most natural path we can take, in most scenarios) might well be found by looking at previous parents, so that's what we do.

We look at our considered node's parent's parent, and perform a line of sight check between that 'grandparent' and our considered node. If there is line of sight between the two, we then look at the grandparent's parent and perform another line of sight check. We continue this process until we find a parent node which does *not* have direct line of sight with our considered node - and we make the child of that parent (e.g. the last node which *does* have line of sight) the parent of our considered node, creating an edge between the two. In this way, it should be obvious how our Star Node in Figure 7 becomes a child of our Start Node.

Since the shortest distance between two points is always a straight line, this will (in most scenarios) provide a path which is far more natural and far more efficient than a traditional A* path by cutting out 'unnecessary' nodes (and the edges between them). If we leverage what we know from Graphics about pixel checks, and the fact that most navigation in our game will take place in two dimensions, this path computation can still be *very* cheap. And, if you want, you can still spline about the selected vertices/nodes in order to make the navigation seem even more natural.

We can enhance this still further with the generation of navigation meshes, which will be discussed in a future tutorial.

Crowd Navigation

One issue in video-games engineering which is often an issue when integrating AI techniques, particularly for navigation, is the management of large numbers of agents. Optimal path planning techniques (such as A*) are very expensive operations. Computing A* paths for a given number of agents poses no issue; if we scale this up to computing paths for a citys population, it becomes problematic. Moreover, that ignores the issue of correctness in the solution (if we simply run independent A* algorithms for every entity in our game, they might well walk through one another, become trapped if they cannot walk through one another due to physics, etc.). Attempting to secure a correct and optimal solution for many agents in a system is a significant complexity problem.

That said, many games have navigation of thousands of entities through their environment, and they have to be doing it somehow! This is an active area of AI research, and one of the few areas which has become prevalent in game engineering (due to its focus on timely results). It also, in some ways, ties into physics, as many of the principles can be considered analogous to attraction and repulsion.

The premise of crowd navigation (which can be considered an application of the wider area of swarm behaviours) revolves around moving a number of agents in a cohesive group (or collective of cohesive groups) from their current location towards some arbitrary waypoint (or goal). We say a number, because crowd navigation techniques can apply just as readily to a small squad as they can to a city population so long as the squad remains concentrated.

An important aspect of this is that most methods of crowd navigation can plug neatly into an implementation of heuristic path planning. In such cases (as we discuss in detail below), the heuristic path planner acts as a guide for the swarm, rather than an absolute route-planner.

Of the many approaches one can take, we shall discuss:

- Fixed group motion
- River flow approach
- Flocking, via the Boids Algorithm
- Ant colony algorithms, and other nature-based approaches

A key element of understanding to take away from these discussions is that largely they revolve around making use of data we have already computed - in the case of flocking, for example, direction of the flock as a whole is informed by a single computed path, while the motion of individual entities within the flock is computed independently within that context. As such, efficient group movement can be considered 'avoiding reinventing the wheel' - what we want are groups which move believably, but don't chew through processor cycles.

The second, more conceptual thing to take away is that crowd management is not really about intelligent path-finding; it is about motion control. If we think about A*, we are intelligently building a path for our agent to explicitly follow over however many subsequent frames it takes our agent to follow it. By contrast, the computation for most crowd navigation is reactive, taking place at a frame-by-frame level, ensuring that the entity is always moving in the right direction but rarely with any firm idea of where it's going or how it's getting there.

Moving as a Fixed Group

This approach to the problem of navigating multiple agents is an optimisation well-suited to small groups (say, a squad in an RTS, or a party in an RPG). It involves pre-defining an arrangement of your multiple agents, performing a heuristic path plan for the group as a whole, and the group moving as a collective with the aim of remaining as close to their fixed arrangement as possible. Figure 8 shows an example of this in action:

Figure 8: Movement as a Fixed Group

The method used to implement this behaviour can vary depending on application (it can, for example, be considered a special case of flocking which we discuss later), but the process is as follows:

1. Define arrangement of entities (this will likely be an array of coordinates, one point per entity, relative to a centre-point). We shall call this a 'squad'.
2. Using the centre-point as your starting position, perform an A* search to determine a path to the squads destination.
3. Begin moving the centre-point of the squad along this route. This motion will, by definition, change the absolute coordinates of the entity position array but their relative coordinates will remain the same.

4. Apply a force to each member of the squad, in the direction of the updated location of its specific point.

5. When the centre-point reaches its destination, after a given time period (long enough for entities to reach their own points relative to the centre-point), cease applying the force. Note: employ some form of hysteresis in order to avoid oscillation.

This approach is very efficient, and plugs neatly into your physics system. It is, however, vulnerable to issues when deployed in the wild. It is possible, for example, for an entity to become separated from the squad (envision a case where that entitys point winds up on the inside of a building, potentially trapping that entity inside it while the rest of the squad moves on). Similarly, it can generate behaviours that lack realism (an entity constantly walking diagonally into a wall because its target point is on the other side of the wall).

These aberrant behaviours can be dealt with a number of ways. The brute force approach would be through use of additional heuristic path planning. In this case, an agent separated from the squad might calculate its own A* path towards the squads goal, and revert to squad navigation once it reaches the squad (this is employed in some older RPGs).

An alternative solution might be and additional forces applied to the entities. In this approach, entities can be dissuadedfrom walking into walls by the introduction of a repulsive force normal to the wall (basically, pushing the agent away, making its net force parallel to the wall). A more sophisticated variant of this would involve nearest-neighbour and line-of-sight checks, with squad members experiencing forces which push them back into line of sight with their squad-mates if this is lost.

There are other variations to this approach. One might keep track of the entity points and adapt them in real-time based on whether or not they have moved into impassable terrain, or use the overall squad path as a means of employing much shorter A* paths for individual entities (each entity plotting an A* path to its updated point, rather than to the distant destination, updated periodically throughout travel). Again, the key is minimising computational expense while maintaining a group behaviour.

Embedded Map Data: A Flowing River

This approach is generally suitable for large groups of agents that are intended to navigate (either together as a group, or independently) to a fixed, small number of destinations. In this case, we define these locations during the design phase of the game. Information regarding how to approach these locations, either precomputed or obtained during runtime, is embedded (baked) into map data to influence entity behaviour.

An example might be a city-based game, where the population is expected to flee into bunkers in the event of an air-raid. The city is modelled, navigationally, as a mesh or grid (each node on the graph representing a section of roadway). When the air-raid siren is heard, each entity begins to navigate towards its nearest bunker. If a bunker is destroyed while they're en route, they must change to the next-nearest, and so on.

This approach essentially involves solving a graph search from every grid square in our environment, to every bunker; this search does not need to be heuristically guided - in fact, in some cases, it might be better (more efficient) for it not to be - if you're interested, investigate the Dijkstra search algorithm (the algorithm A* is based on).

However you perform the search, it involves heavy pre-computation (though not particularly heavy; solving the problem for a 1024-node map will take less than a second of pre-computation, and *can* be done at compile-time rather than run-time). This information will be stored solely in terms of the first direction each grid square determines; each bunker will have its own map. A simple example, with two bunkers and twenty-five grid squares, is shown in Figure 9.

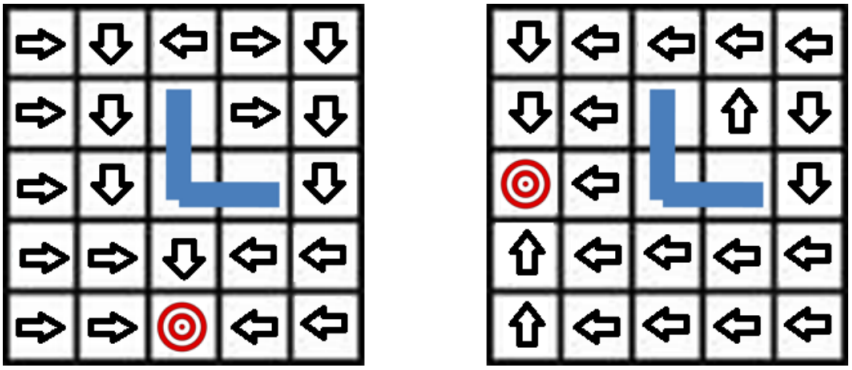

Figure 9: Embedded Map Data

The left-hand diagram shows embedded data computed for Bunker 1 (the south-most bunker, located at (3,0)); the right-hand digram shows embedded data computed for Bunker 2 (the west-most bunker, located at (0,3)). An agent at square (1,3) is closest to the Bunker 2, and is directed west. If Bunker 2 is destroyed, that agent will instead be directed south, in accordance with the data for Bunker 1.

This data can be employed with any of the other navigation systems discussed in this lecture (in the same way that Sort and Sweep can be applied as a bolt-on to any broad phase collision check). If we consider fixed group motion, this baked data can be the basis for guiding our group. It can also be the basis for avoiding some of the pitfalls with that approach (the direction of motion can be applied as an additional force to entities, keeping them from entering impassable terrain and so on).

Similarly, it can be used to inform our flocking algorithm, by providing directional data that does not need re-computing on the fly. It can even supplant computation of traditional path-finding for a single agent, in a scenario where only a finite number of changes can occur in the environment.

Another benefit is that this approach maps neatly (if sub-optimally) to destructible terrain. The introduction of another passable grid square (for example, the destruction of one of the walls in the example above), only involves the computation of the route from this newly passable step and the squares immediately adjacent to it. This approach to destructible terrain might give sub-optimal answers; a re-computation of specific regions based on destroyed terrain is more expensive, but provides better results. And, of course, if you know which terrain is destructible, you always have the option of baking in additional maps to be referenced when and if that terrain is destroyed.

Flocking: It's for the Boids

The flocking algorithm (proposed by Reynolds in 1987) is an $\mathcal{O}(n^2)$ complexity algorithm (in its simplest implementation) which determines the direction of motion for every entity in a flock. The basic premise revolves around taking averages of certain properties of the flock, and using vector addition and normalisation to settle on a new direction for every entity in each frame of the update. We should remember that AI does not need to update every rendered frame, or every physics system update; it needs only update often enough to convince the player that the entity is behaving.

In terms of a vanilla flocking algorithm, three factors are considered when determining a new directional update of the entities (known as boids):

- Separation: How far an entity is from its nearest entities, to avoid crowding each other. This is important, as without this consideration eventually all Boids will wind up occupying the same position (or clustered into a massive mess of collision checks, if they have physical presence). This is a short-range repulsion.
- Alignment: The average direction of the flock is calculated (vector sum, then normalised).

- Cohesion: Steering towards the average position of the flock (mean positional vector sum). This is a long-range attraction

Figure 10 shows a visual example, where circles are boids and the diamond is the flock's centre of motion; the black arrow indicates the average direction of the flock, and smaller arrows indicate the attractive forces acting on each boid.

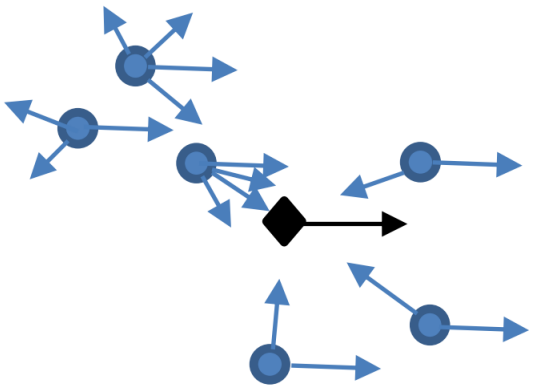

Figure 10: A Flock of Boids

Weighting factors can be applied to vary the importance of these three elements. Similarly, the range at which separation becomes a consideration can vary on the basis of how cohesive we wish a flock to appear. It is important to remember that contributions should be normalised when we consider our final direction; particularly if this is then going to be mapped to a velocity multiplier for our physics. Optimisations to flocking, again involving nearest-neighbour checks, can significantly reduce computation.

As regards alignment, this is a factor we can bend to our will slightly, and tie our flocking into intelligent path planning. If we consider the river flow described above, it is easy to see how the directions we chart as a result of this might map handily to a flock of entities moving about the environment in the context of what their alignment should be. We can, of course, trend towards that alignment rather than forcing it, which will give our Boids a very natural curve to their motion (spline). This works equally well for a single path plan.

We can extend our flocking to allow for the separation of the flock into multiple flocks; this can be done through the application of an upper boundary (or distance-dependent weighting factor) for cohesion, or intelligent finite state machine design, or a combination of those elements. We can also adapt the techniques in flocking to allow for a more reactive squad (as discussed above), in conjunction with some finite state controls.

Ant Colonies, and Other Animals

Many algorithms applied in research AI which can be mapped to navigation draw their inspiration from the behaviours of colonies in nature. Flocking, described above, is such an algorithm; we separate it from this discussion primarily because of its ease of use relative to more biologically grounded systems, and also because many of these systems are more accurately considered means of affecting the alignment of a flock, than navigation methods in and of themselves.

The Ant Colony algorithms (and there are many) have, at their basis, a principle of a nave collection of entities which, slowly, become more familiar with their environment due to exploration. The premise of these is simple:

- An ant colony is comprised of a great many ants (our entities)
- At the beginning of our simulation, ants have no knowledge of their environment.
- Ants will search the environment for food; upon finding food, they will return to the colony and leave a pheromone trail down the path they travelled (this can be considered analogous to a weighting factor in intelligent path planning).
- If another ant, while searching for food, crosses the pheromone trail it is likely to try and follow it, rather than continue wandering.
- If it successfully finds food, it will leave its own pheromone trail as it returns to the colony.
- Over time, pheromone trails will naturally decay; if a source of food is emptied, ants which follow the trail will not renew it, and eventually will not be especially likely to follow it.

Ant Colonies are a more complex navigational algorithm than others we have discussed, as they are strategic in nature. By this, we mean that the ants are searching as they travel for something in the environment, rather than attempting to path to a known destination. As such, it is a waste to employ ant colonies purely as a means of navigating from a known origin to a known goal this would be inefficient, and we would be better employing an embedded data solution, or some other algorithm discussed earlier.

Where ant colonies can be applicable are scenarios where your navigation is strategic in nature. If you consider a harvesting entity for a real-time strategy, with an automation mode, it is obvious that multiples of these entities can benefit from the inclusion of an approach of this kind. The question then becomes "Is implementing a pheromone trail more efficient than simply flagging the first source of gold I find, and having all of my gold-harvesting NPCs automatically go there to mine?"; that depends upon the nature of your game, and the number of entities your AI will be controlling. Similarly, it is related to the rarity of resource nodes in your environment.

Many other algorithms and optimisations have similar bases in biology. Some examples are:

- Artificial Bee Colony algorithm
- Intelligent Water Drops algorithm
- Particle Swarm Optimisation (and Multi-swarm Optimisation)

For the purposes of engineering a real-time solution, however, many of these approaches require a strategic element to the problem we are trying to solve (as with the ants) in order to show any benefit over the methods discussed earlier in the tutorial.

Implementation

Check the Practical Tasks handout for today. Implement the A* algorithm, or some other best-path graph search. Consider how you might embed data in your node class to better guide entities about an environment. Investigate the boids algorithm and attempt to implement it. Consider how you might locate an agent's nearest neighbour in a squad - does this check map to anything you might have considered in physics (hint hint)?

Summary

We have considered the problem of intelligent single-agent navigation, and a heuristic-based solution to it. We have gone on to explore multi-agent navigation, and proposed solutions which can be computed in real-time. We have highlighted the strengths and weaknesses of these solutions, and emphasised that they are only a small taste of what's out there in terms of group navigation solutions.