Artificial Intelligence Tutorial 1: Path Finding

Introduction to Path-Finding

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:

- 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 set 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.

Overview of A*

The A* algorithm represents the environment as a set of interconnected nodes. Each node could be a shape on a grid, or a set of points in a network. This set of connected nodes is often referred to as the *node graph*.

In calculating an optimal path between two distant positions on the grid, the algorithm begins at the start node, and calculates the progress which would be made by moving to each of the directly-connected nodes. The best of those nodes is selected, and the algorithm performs the same calculation on each of the nodes directly connected to this new node. This iterative process is repeated until the destination node is reached. Tracing back through the nodes to the start position gives the optimal path.

The "progress" is calculated as a combination of how far we have travelled to get to the node, and an estimate of the remaining distance to the target node. This estimate is known as the *heuristic*, and can be calculated in many ways, which leads to various flavours of A*. For this tutorial we will

consider the most straightforward heuristic. This will make more sense as we see the algorithm work through the example problem.

Represent the environment

The problem which we will address in this tutorial is to plot a route around a very simple obstacle. Figure 1 illustrates the problem to be solved. We must find a route from the start point A (the car) to the end point B (the target), going around the shaded wall.

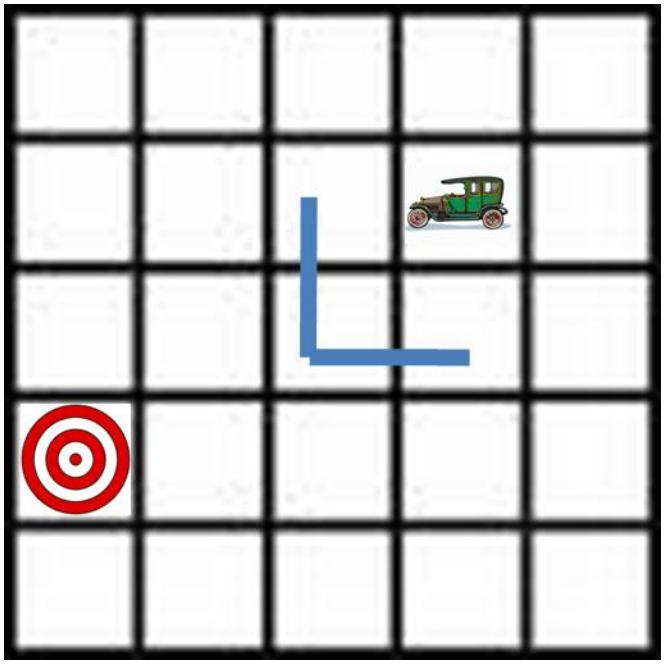


FIGURE 1 – *Problem 1*

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 (ie a node), and its status is recorded as walkable or unwalkable. 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.

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 isn't 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.

Code Exercise 1 - *Read the data.*

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.

DATA 1 – *Data for Problem 1*

```
Rows 5
Columns 5
0 0 0 0 0
0 0 1 A 0
0 0 1 1 0
B 0 0 0 0
0 0 0 0 0
```

Initialise the search

Before we start searching the grid for a path, we need to set up some data structures that will be used to store the results of the path-tests carried out along the way.

The first item is a list of the nodes which the algorithm is currently interested in (i.e. they are potential candidates for inclusion in the path). This is known as the Open List. Initially the only node in the Open List is the start node. Once the algorithm has finished considering a node on the Open List, it is transferred to another list of nodes, known as the Closed List. The final path will be constructed from nodes in the Closed List.

We also introduce the concept of Parent nodes at this stage. Each node contains a reference to the previous node in the chain which led the algorithm to that point. Once the path search is complete, the parent of each node will lead the algorithm back from the target node all the way to the start node, thereby providing the final path data.

Code Exercise 2 – *Initialise the search. As well as a node class, three structures are required – an array of the nodes themselves, and two vectors, one containing the Open List, the other containing the Closed List. Remember that using vectors requires inclusion of the stl library.*

The A* Algorithm

We will now look at the A*algorithm in detail.

Path scoring

The key to determining which squares to use when finding the path is the following equation:

$f = g + h$

where

- f is the total cost for the node under consideration

- *g* is the movement cost from the starting point A 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 don't know the actual distance until we find the path, because obstacles can be in the way. We use a simple way to calculate *h* in this tutorial, but there are many others to choose from, depending on the application of the A* algorithm.

Our path is generated by repeatedly going through our open list and choosing the node with the lowest *f* score.

For the purposes of this tutorial, 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 leads to a significant optimisation on using floating point arithmetic.

The path-finding loop

The logic for the main loop of the A* Algorithm is as follows:

1. Let A = starting point.

2. Assign f, g and h values to A.

3. Add A to the Open list. At this point, A is the only node on the Open list.

4. Let P = the best node from the Open list (i.e. the node with the lowest f-value).

    a. If P is the goal node (i.e. B), then go to step 7 – a path has been found.

    b. If the Open list is empty, then quit – a path cannot be found

5. Let Q = a valid node directly connected to P.

    a. Calculate f, g, and h values for Q.

    b. Check whether Q is on the Open or Closed list.

        i. If so, check whether the new path is more efficient (i.e. has a lower f-value).

            1. If so update the path by setting P as Q's parent, and assign the new f, g, and h values to Q .

            2. Else, do nothing (ie Q's current parent provides the more efficient path).

        ii. Else, add Q to the Open list, set P as Q's parent and assign the f, g, and h values to Q.

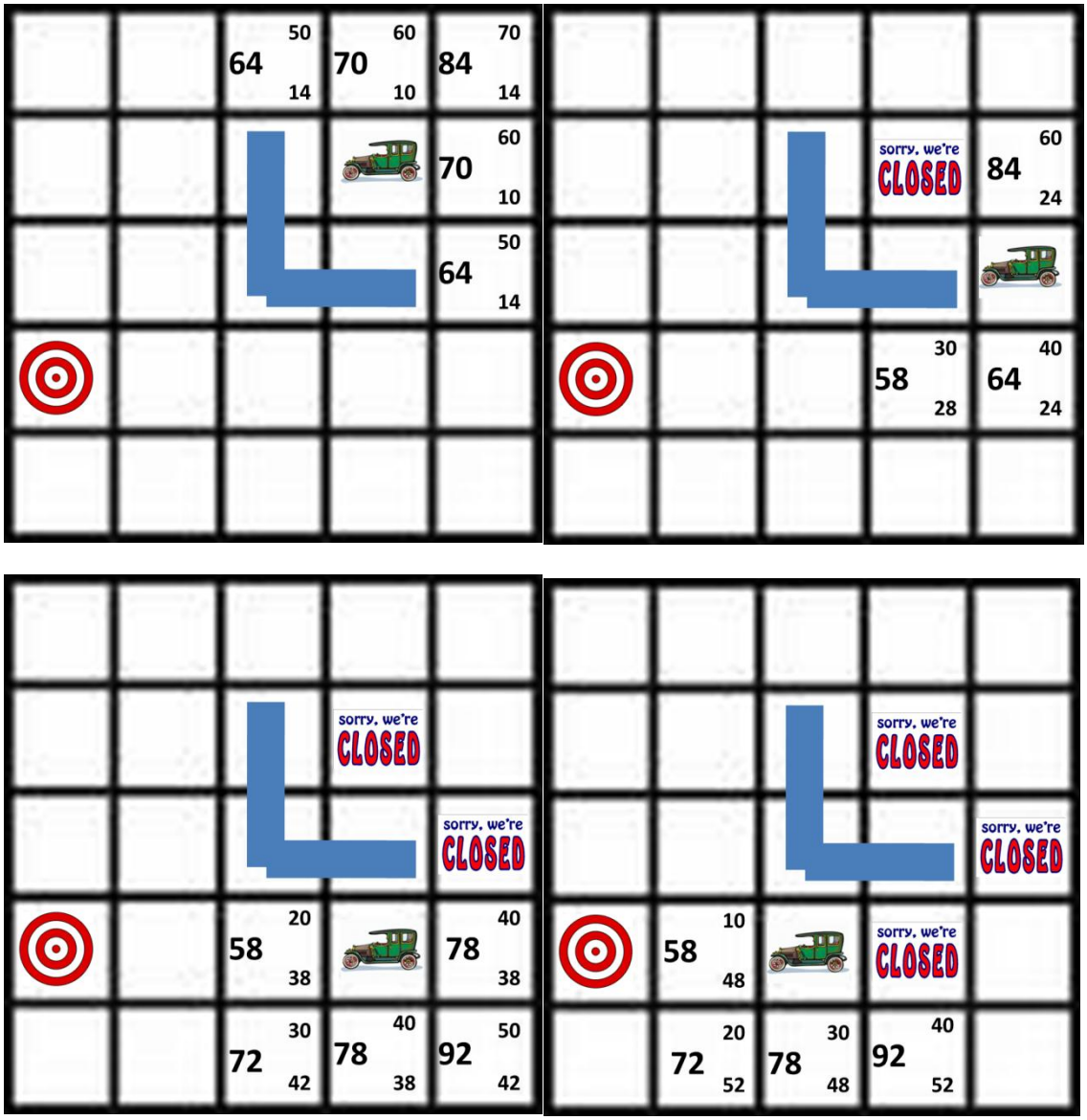
    c. Repeat step 5 for all valid directly-connected nodes of P.

6. Repeat from step 4.

7. Starting at B, construct a path back to A, by using the parent node of each node. This is the optimal path that has been found.

Our Example

These diagrams show how the algorithm solves our example problem.



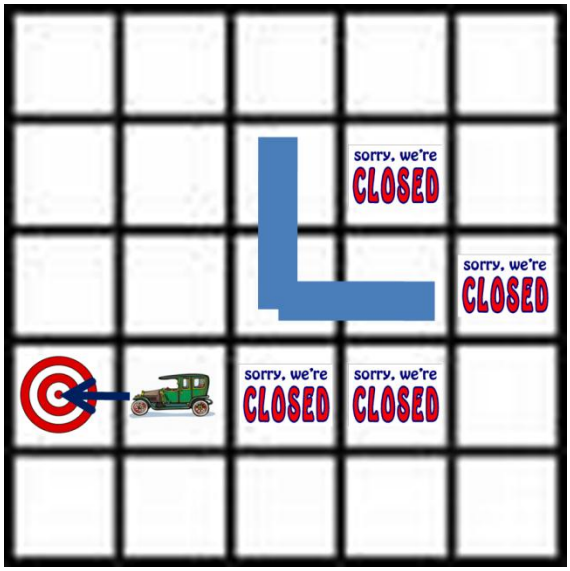


FIGURE 2: *Solving the problem.*

Code Exercise 3 – *Implement the algorithm, as described in steps 1 to 7, in C++. The output is a text string listing the coordinates of each node in the final path.*

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 don't 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 vehicle's handling model and physics engine to drive the 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 character's 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.

A caveat: it’s not cheap

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 (eg using alleyways and pedestrian areas).

Further work

1. Implement the A* algorithm as described above. Input should be a data file of the format shown, and the output should be a list of node coordinates along the calculated path.

2. Try some different obstacle shapes by changing the content of the data file. Your executable should be able to solve any path-finding problem in that format. Also try extending the size of the grid.
3. Introduce the concept of slow terrain. Some parts of the environment could take longer to traverse than others (eg the ground is boggy). Make sure you do this in a data-driven way (ie introduce a new type of node in the input data), and adapt the code to deal with this.
4. Introduce a pair of teleport nodes, which act as a shortcut from one part of the map to another. Again, do this in a data-driven manner.
5. Draw a representation of the grid, and the calculated path on screen.
6. Rewrite the whole thing to work on a 3D grid of cubes.