Smart Moves: Intelligent Pathfinding

By Bryan Stout
Published in *Game Developer* Magazine, July, 1997

PathDemo
(Oct'96 Archive)
[Get Demo]

Of all the decisions involved in computer-game AI, the most common is probably pathfinding -- looking for a good route for moving an entity from here to there. The entity can be a single person, a vehicle, or a combat unit; the genre can be an action game, a simulator, a role-playing game, or a strategy game. But any game in which the computer is responsible for moving things around has to solve the pathfinding problem.

And this is not a trivial problem. Questions about pathfinding are regularly seen in online game programming forums, and the entities in several games move in less than intelligent paths. However, although pathfinding is not trivial, there are some well-established, solid algorithms that deserve to be known better in the game community.

Several pathfinding algorithms are not very efficient, but studying them serves us by introducing concepts incrementally. We can then understand how different shortcomings are overcome.

To demonstrate the workings of the algorithms visually, I have developed a program in Delphi 2.0 called “PathDemo.” It is available for readers to download. The article and demo assume that the playing space is represented with square tiles. You can adapt the concepts in the algorithms to other tilings, such as hexagons; ideas for adapting them to continuous spaces are discussed at the end of the article.

Pathfinding on the Move

The typical problem in pathfinding is obstacle avoidance. The simplest approach to the problem is to ignore the obstacles until one bumps into them. The algorithm would look something like this:

```
while not at the goal
pick a direction to move toward the goal
if that direction is clear for movement
move there
else
pick another direction according to an avoidance strategy
```

This approach is simple because it makes few demands: all that

needs to be known are the relative positions of the entity and its goal, and whether the immediate vicinity is blocked. For many game situations, this is good enough.

Different obstacle-avoidance strategies include:

Movement in a random direction. If the obstacles are all small and convex, the entity (shown as a green dot) can probably get around them by moving a little bit away and trying again, until it reaches the goal (shown as a red dot). Figure 1a shows this strategy at work. A problem arises with this method if the obstacles are large or if they are concave, as is seen in Figure 1b, the entity can get completely stuck, or at least waste a lot of time before it stumbles onto a way around. One way to avoid this: if a problem is too hard to deal with, alter the game so it never comes up. That is, make sure there are never any concave obstacles.

Tracing around the obstacle. Fortunately, there are other ways to get around. If the obstacle is large, one can do the equivalent of placing a hand against the wall and following the outline of the obstacle until it is skirted. Figure 2a shows how well this can deal with large obstacles. The problem with this technique comes in deciding when to stop tracing. A typical heuristic may be: “Stop tracing when you are heading in the direction you wanted to go when you started tracing.” This would work in many situations, but Figure 2b shows how one may end up constantly circling around without finding the way out.

Robust tracing. A more robust heuristic comes from work on mobile robots: “When blocked, calculate the equation of the line from your current position to the goal. Trace until that line is again crossed. Abort if you end up at the starting position again.” This method is guaranteed to find a way around the obstacle if there is one, as is seen in Figure 3a. (If the original point of blockage is between you and the goal when you cross the line, be sure not to stop tracing, or more circling will result.) Figure 3b shows the downside of this approach: it will often take more time tracing the obstacle than is needed, making it look pretty simple-mindedthough not as simple as endless circling. A happy compromise would be to combine both approaches: always use the simpler heuristic for stopping the tracing first, but if circling is detected, switch to the robust heuristic.

Looking Before You Leap

Although the obstacle-skirting techniques discussed above can often do a passable or even adequate job, there are situations where the only intelligent approach is to plan the entire route before the first step is taken. In addition, these methods do little to handle the problem of weighted regions, where the difficulty is not so much avoiding obstacles as finding the cheapest path among several choices where the terrain can vary in its cost.

Fortunately, the fields of Graph Theory and conventional AI have

several algorithms that can be used to handle both difficult obstacles and weighted regions. In the literature, many of these algorithms are presented in terms of changing between states, or traversing the nodes of a graph. They are often used in solving a variety of problems, including puzzles like the 15-puzzle or Rubik's cube, where a state is an arrangement of the tiles or cubes, and neighboring states (or adjacent nodes) are visited by sliding one tile or rotating one cube face. Applying these algorithms to pathfinding in geometric space requires a simple adaptation: a state or a graph node stands for the entity being in a particular tile, and moving to adjacent tiles corresponds to moving to the neighboring states, or adjacent nodes.

Working from the simplest algorithms to the more robust, we have:

Breadth-first search. Beginning at the start node, this algorithm first examines all immediate neighboring nodes, then all nodes two steps away, then three, and so on, until a goal node is found. Typically, each node's unexamined neighboring nodes are pushed onto an Open list, which is usually a FIFO (first-in-first-out) queue. The algorithm would go something like what is shown in Listing 1. Figure 4 shows how the search proceeds. We can see that it does find its way around obstacles, and in fact it is guaranteed to find a shortest paththat is, one of several paths that tie for the shortest in lengthif all steps have the same cost. There are a couple of obvious problems. One is that it fans out in all directions equally, instead of directing its search towards the goal; the other is that all steps are not equalat least the diagonal steps should be longer than the orthogonal ones.

Bidirectional breadth-first search. This enhances the simple breadth-first search by starting two simultaneous breadth-first searches from the start and the goal nodes and stopping when a node from one end's search finds a neighboring node marked from the other end's search. As seen in Figure 5, this can save substantial work from simple breadth-first search (typically by a factor of 2), but it is still quite inefficient. Tricks like this are good to remember, though, since they may come in handy elsewhere.

Dijkstra's algorithm. E. Dijkstra developed a classic algorithm for traversing graphs with edges of differing weights. At each step, it looks at the unprocessed node closest to the start node, looks at that node's neighbors, and sets or updates their respective distances from the start. This has two advantages to the breadth-first search: it takes a path's length or cost into account and updates the goodness of nodes if better paths to them are found. To implement this, the Open list is changed from a FIFO queue to a priority queue, where the node popped is the one with the best scorehere, the one with the lowest cost path from the start. (See Listing 2.) We see in Figure 6 that Dijkstra's algorithm adapts well to terrain cost. However, it still has the weakness of breadth-width search in ignoring the direction to the goal.

Depth-first search. This search is the complement to breadth-first search; instead of visiting all a node's siblings before any children, it

visits all of a node's descendants before any of its siblings. To make sure the search terminates, we must add a cutoff at some depth. We can use the same code for this search as for breadth-first search, if we add a depth parameter to keep track of each node's depth and change Open from a FIFO queue to a LIFO (last-in-first-out) stack. In fact, we can eliminate the Open list entirely and instead make the search a recursive routine, which would save the memory used for Open. We need to make sure each tile is marked as "visited" on the way out, and is unmarked on the way back, to avoid generating paths that visit the same tile twice. In fact, Figure 7 shows that we need to do more than that: the algorithm still can tangle around itself and waste time in a maddening way. For geometric pathfinding, we can add two enhancements. One would be to label each tile with the length of the cheapest path found to it yet; the algorithm would then never visit it again unless it had a cheaper path, or one just as cheap but searching to a greater depth. The second would be to have the search always look first at the children in the direction of the goal. With these two enhancements checked, one sees that the depth-first search finds a path quickly. Even weighted paths can be handled by making the depth cut-off equal the total accumulated cost rather than the total distance.

Iterative-deepening depth-first search. Actually, there is still one fly in the depth-first ointmentpicking the right depth cutoff. If it is too low, it will not reach the goal; if too high, it will potentially waste time exploring blind avenues too far, or find a weighted path which is too costly. These problems are solved by doing iterative deepening, a technique that carries out a depth-first search with increasing depth: first one, then two, and so on until the goal is found. In the pathfinding domain, we can enhance this by starting with a depth equal to the straight-line distance from the start to the goal. This search is asymptotically optimal among brute force searches in both space and time.

Best-first search. This is the first heuristic search considered, meaning that it takes into account domain knowledge to guide its efforts. It is similar to Dijkstra's algorithm, except that instead of the nodes in Open being scored by their distance from the start, they are scored by an estimate of the distance remaining to the goal. This cost also does not require possible updating as Dijkstra's does. Figure 8 shows its performance. It is easily the fastest of the forward-planning searches we have examined so far, heading in the most direct manner to the goal. We also see its weaknesses. In Figure 8a, we see that it does not take into account the accumulated cost of the terrain, plowing straight through a costly area rather than going around it. And in Figure 8b, we see that the path it finds around the obstacle is not direct, but weaves around it in a manner reminiscent of the hand-tracing techniques seen above.

The Star of the Search Algorithms (A* Search)

The best-established algorithm for the general searching of optimal paths is A* (pronounced "A-star"). This heuristic search ranks each

node by an estimate of the best route that goes through that node. The typical formula is expressed as:

f(n) = g(n) + h(n)

where: f(n)is the score assigned to node n g(n)is the actual cheapest cost of arriving at n from the starth(n)is the heuristic estimate of the cost to the goal from n

So it combines the tracking of the previous path length of Dijkstra's algorithm, with the heuristic estimate of the remaining path from best-first search. The algorithm proper is seen in Listing 3. Since some nodes may be processed more than oncefrom finding better paths to them laterwe use a new list called Closed to keep track of them.

A* has a couple interesting properties. It is guaranteed to find the shortest path, as long as the heuristic estimate, h(n), is admissiblethat is, it is never greater than the true remaining distance to the goal. It makes the most efficient use of the heuristic function: no search that uses the same heuristic function h(n) and finds optimal paths will expand fewer nodes than A*, not counting tie-breaking among nodes of equal cost. In Figures 9a, 9b, and 9c we see how A* deals with situations that gave problems to other search algorithms.

How Do I Use A*?

A* turns out to be very flexible in practice. Consider the different parts of the algorithm.

The state would often be the tile or position the entity occupies. But if needed, it can represent orientation and velocity as well (for example, for finding a path for a tank or most any vehicletheir turn radius gets worse the faster they go).

Neighboring states would vary depending on the game and the local situation. Adjacent positions may be excluded because they are impassable or are between the neighbors. Some terrain can be passable for certain units but not for others; units that cannot turn quickly cannot go to all neighboring tiles.

The cost of going from one position to another can represent many things: the simple distance between the positions; the cost in time or movement points or fuel between them; penalties for traveling through undesirable places (such as points within range of enemy artillery); bonuses for traveling through desirable places (such as exploring new terrain or imposing control over uncontrolled locations); and aesthetic considerationsfor example, if diagonal moves are just as cheap as orthogonal moves, you may still want to make them cost more, so that the routes chosen look more direct and natural.

The estimate is usually the minimum distance between the current

node and the goal multiplied by the minimum cost between nodes. This guarantees that h(n) is admissible. (In a map of square tiles where units may only occupy points in the grid, the minimum distance would not be the Euclidean distance, but the minimum number of orthogonal and diagonal moves between the two points.)

The goal does not have to be a single location but can consist of multiple locations. The estimate for a node would then be the minimum of the estimate for all possible goals. Search cutoffs can be included easily, to cover limits in path cost, path distance, or both. From my own direct experience, I have seen the A* star search work very well for finding a variety of types of paths in wargames and strategy games.

The Limitations of A*

There are situations where A* may not perform very well, for a variety of reasons. The more or less real-time requirements of games, plus the limitations of the available memory and processor time in some of them, may make it hard even for A* to work well. A large map may require thousands of entries in the Open and Closed list, and there may not be room enough for that. Even if there is enough memory for them, the algorithms used for manipulating them may be inefficient.

The quality of A*'s search depends on the quality of the heuristic estimate h(n). If h is very close to the true cost of the remaining path, its efficiency will be high; on the other hand, if it is too low, its efficiency gets very bad. In fact, breadth-first search is an A* search, with h being trivially zero for all nodesthis certainly underestimates the remaining path cost, and while it will find the optimum path, it will do so slowly. In Figure 10a, we see that while searching in expensive terrain (shaded area), the frontier of nodes searched looks similar to Dijkstra's algorithm; in 10b, with the heuristic increased, the search is more focused.

Let's look at ways to make the A* search more efficient in problem areas.

Transforming the Search Space

Perhaps the most important improvement one can make is to restructure the problem to be solved, making it an easier problem. Macro-operators are sequences of steps that belong together and can be combined into a single step, making the search take bigger steps at a time. For example, airplanes take a series of steps in order to change their orientation and altitude. A common sequence may be used as a single change of state operator, rather than using the smaller steps individually. In addition, search and general problem-solving methods can be greatly simplified if they are reduced to sub-problems, whose individual solutions are fairly simple. In the case of pathfinding, a map can be broken down into large contiguous areas whose connectivity is known. One or two border tiles between each pair of adjacent areas are chosen; then the route is first laid out

in by a search among adjacent areas, in each of which a route is found from one border point to another.

For example, in a strategic map of Europe, a path-finder searching for a land route from Madrid to Athens would probably waste a fair amount of time looking down the boot of Italy. Using countries as areas, a hierarchical search would first determine that the route would go from Spain to France to Italy to Yugoslavia (looking at an old map) to Greece; and then the route through Italy would only need to connect Italy's border with France, to Italy's border with Yugoslavia. As another example, routes from one part of a building to another can be broken down into a path of rooms and hallways to take, and then the paths between doors in each room.

It is much easier to choose areas in predefined maps than to have the computer figure them out for randomly generated maps. Note also that the examples discussed deal mainly with obstacle avoidance; for weighted regions, it is trickier to assign useful regions, especially for the computer (it may not very useful, either).

Storing It Better

Even if the A* search is relatively efficient by itself, it can be slowed down by inefficient algorithms handling the data structures. Regarding the search, two major data structures are involved.

The first is the representation of the playing area. Many questions have to be addressed. How will the playing field be represented? Will the areas accessible from each spotand the costs of moving therebe represented directly in the map or in a separate structure, or calculated when needed? How will features in the area be represented? Are they directly in the map, or separate structures? How can the search algorithm access necessary information quickly? There are too many variables concerning the type of game and the hardware and software environment to give much detail about these questions here.

The second major structure involved is the node or state of the search, and this can be dealt with more explicitly. At the lower level is the search state structure. Fields a developer might wish to include in it are:

- The location (coordinates) of the map position being considered at this state of the search.
- Other relevant attributes of the entity, such as orientation and velocity.
- The cost of the best path from the source to this location.
- The length of the path up to this position.
- The estimate of the cost to the goal (or closest goal) from this location.
- The score of this state, used to pick the next state to pop off Open.
- A limit for the length of the search path, or its cost, or both, if

applicable.
- A reference (pointer or index) to the parent of this nodethat is, the node that led to this one.

Additional references to other nodes, as needed by the data structure used for storing the Open and Closed lists; for example, “next” and maybe “previous” pointers for linked lists, “right,” “left,” and “parent” pointers for binary trees.

Another issue to consider is when to allocate the memory for these structures; the answer depends on the demands and constraints of the game, hardware, and operating system.

On the higher level are the aggregate data structures the Open and Closed lists. Although keeping them as separate structures is typical, it is possible to keep them in the same structure, with a flag in the node to show if it is open or not. The sorts of operations that need to be done in the Closed list are:

```
Insert a new node.
Remove an arbitrary node.
Search for a node having certain attributes (location,
speed, direction).
Clear the list at the end of the search.
```

The Open list does all these, and in addition will:

- Pop the node with the best score.
- Change the score of a node.

The Open list can be thought of as a priority queue, where the next item popped off is the one with the the highest priority in our case, the best score. Given the operations listed, there are several possible representations to consider: a linear, unordered array; an unordered linked list; a sorted array; a sorted linked list; a heap (the structure used in a heap sort); a balanced binary search tree.

There are several types of binary search trees: 2-3-4 trees, red-black trees, height-balanced trees (AVL trees), and weight-balanced trees. Heaps and balanced search trees have the advantage of logarithmic times for insertion, deletion, and search; however, if the number of nodes is rarely large, they may not be worth the overhead they require.

Fine-Tuning Your Search Engine

There are also ways of tweaking the search algorithm to help get good results while working with limited resources:

Beam search. One way of dealing with restricted memory is to limit the number of nodes on the Open list; when it is full and a new node is to be inserted, simply drop the node with the worst rating. The

Closed list could also be eliminated, if each tile stores its best path length. There is no promise of an optimal path since the node leading to it may be dropped, but it may still allow finding a reasonable path.

Iterative-deepening A*. The iterative-deepening technique used for depth-first search (IDDFS) as mentioned above can also be used for an A* search. This entirely eliminates the Open and Closed lists. Do a simple recursive search, keep track of the accumulated path cost g(n), and cut off the search when the rating f(n) = g(n) + h(n) exceeds the limit. Begin the first iteration with the cutoff equal to h(start), and in each succeeding iteration, make the new cutoff the smallest f(n) value which exceeded the old cutoff. Similar to IDDFS among brute-force searches, IDA* is asymptotically optimal in space and time usage among heuristic searches.

Inadmissible heuristic h(n). As discussed above, if the heuristic estimate h(n) of the remaining path cost is too low, then A* can be quite inefficient. But if the estimate is too high, then the path found is not guaranteed to be optimal and may be abysmal. In games where the range of terrain cost is widefrom swamps to freewaysyou may try experimenting with various intermediate cost estimates to find the right balance between the efficiency of the search and the quality of the resulting path.

There are also other algorithms that are variations of A*. Having toyed with some of them in PathDemo, I believe that they are not very useful for the geometric pathfinding domain.

What if I'm in a Smooth World?

All these search methods have assumed a playing area composed of square or hexagonal tiles. What if the game play area is continuous? What if the positions of both entities and obstacles are stored as floats, and can be as finely determined as the resolution of the screen? Figure 11a shows a sample layout. For answers to these search conditions, we can look at the field of robotics and see what sort of approaches are used for the path-planning of mobile robots. Not surprisingly, many approaches find some way to reduce the continuous space into a few important discrete choices for consideration. After this, they typically use A* to search among them for a desirable path. Ways of quantizing the space include: Tiles. A simple approach is to slap a tile grid on top of the space. Tiles that contain all or part of an obstacle are labeled as blocked; a fringe of tiles touching the blocked tiles is also labeled as blocked to allow a buffer of movement without collision. This representation is also useful for weighted regions problems. See Figure 11b.

Points of visibility. For obstacle avoidance problems, you can focus on the critical points, namely those near the vertices of the obstacles (with enough space away from them to avoid collisions), with points being considered connected if they are visible from each other (that is, with no obstacle between them). For any path, the search considers only the critical points as intermediate steps between start

and goal. See Figure 11c.

Convex polygons. For obstacle avoidance, the space not occupied by polygonal obstacles can be broken up into convex polygons; the intermediate spots in the search can be the centers of the polygons, or spots on the borders of the polygons. Schemes for decomposing the space include: C-Cells (each vertex is connected to the nearest visible vertex; these lines partition the space) and Maximum-Area decomposition (each convex vertex of an obstacle projects the edges forming the vertex to the nearest obstacles or walls; between these two segments and the segment joining to the nearest visible vertex, the shortest is chosen). See Figure 11d. For weighted regions problems, the space is divided into polygons of homogeneous traversal cost. The points to aim for when crossing boundaries are computed using Snell's Law of Refraction. This approach avoids the irregular paths found by other means.

Quadtrees. Similar to the convex polygons, the space is divided into squares. Each square that isn't close to being homogenous is divided into four smaller squares, recursively. The centers of these squares are used for searching a path. See Figure 11e.

Generalized cylinders. The space between adjacent obstacles is considered a cylinder whose shape changes along its axis. The axis traversing the space between each adjacent pair of obstacles (including walls) is computed, and the axes are the paths used in the search. See Figure 11f.

Potential fields. An approach that does not quantize the space, nor require complete calculation beforehand, is to consider that each obstacle has a repulsive potential field around it, whose strength is inversely proportional to the distance from it; there is also a uniform attractive force to the goal. At close regular time intervals, the sum of the attractive and repulsive vectors is computed, and the entity moves in that direction. A problem with this approach is that it may fall into a local minimum; various ways of moving out of such spots have been devised.

Bryan Stout has done work in "real" AI for Martin Marietta and in computer games for MicroProse. He is preparing a book on computer game AI to be published by Addison-Wesley.