3.1

# Basic A* Pathfinding Made Simple

***James Matthews—Generation5***

**jmatthews@generation5.org**

The A* algorithm has been the source of much confusion and mystery within the game programming community. While the goals and basic theory of A* are relatively simple to understand, the implementation of the algorithm can be a nightmare to realize. This article will hopefully clarify the theory *and* the implementation of the A* algorithm. We will look first at the basics of A* as they apply to two-dimensional maps, and then study a C++ class that implements the algorithm.

## An Overview

The A* (pronounced *a-star*) algorithm will find a path between two points on a map. While many different pathing algorithms exist, A* will find the shortest path, if one exists, and will do so relatively quickly—which is what sets it apart from the others. There are *many* flavors of A*, but they all build around the basic algorithm presented here.

A* is a *directed* algorithm, meaning it doesn't blindly search for a path (like a rat in a maze), but instead assesses the best direction to explore, sometimes backtracking to try alternative means. This is what makes the A* algorithm so flexible.

### Terms

Before we delve into A*'s particulars, we should define a few terms.

A **map** (or **graph**) is the space that A* is using to find a path between two positions. This doesn't necessarily have to be a map in the literal sense of the meaning. The map could be comprised of squares or hexagons, be a three-dimensional area, or even a spatial representation of game trees. The important thing to understand is that the map is the area within which A* works.

**Nodes** are the structures that represent positions on the map. However, the map will be in a data structure independent of the nodes. The nodes store information critical to the A* algorithm as well as positional information; thus, nodes act as a bookkeeping device to store the progress of the pathfinding search. It is important to remember that two or more nodes can correspond to the same position on the map.

The **distance** (or **heuristic**) is used to determine the "suitability" of the node being explored. We will use the term *distance*, since this article will primarily be dealing with applying A* to traditional two-dimensional maps.

The **cost** of a node is probably the hardest term to define—an analogy is probably best. When traveling large distances, various factors are taken into account (time, energy, money, or scenery) that affect whether a certain path is to be taken. Possible paths between the start and goal nodes will have associated costs, and it is the job of A* to minimize these costs. Note that there are no set algorithms or equations to determine the distance and cost of a node; they are *completely* application-dependent.

## The A* Algorithm

We will now venture into the theory surrounding the A* algorithm. A* traverses the map by creating nodes that correspond to the various positions it explores. Remember that these nodes are for recording the progress of the search. In addition to holding the map location, each of these nodes has three main attributes commonly called *f*, *g*, and *h*, sometimes referred to as the *fitness*, *goal*, and *heuristic* values, respectively. The following describes each in more detail:

- *g* is the cost to get from the starting node to this node. Many different paths go from the start node to this map location, but this node represents a single path to it.
- *h* is the estimated cost to get from this node to the goal. In this setting, *h* stands for *heuristic* and means *educated guess*, since we don't really know the cost (that's why we're looking for a path).
- *f* is the sum of *g* and *h*. *f* represents our best guess for the cost of this path going through this node. The lower the value of *f*, the better we *think* the path is.

At this point, you might ask why we are measuring some distances and guessing at other distances. The purpose of *f*, *g*, and *h* is to quantify how promising a path is up to this node. Component *g* is something we can calculate perfectly. It is the cost required to get to this current node. Since we've explored all nodes that led to this one, we know the value of *g* exactly. However, component *h* is a completely different beast. Since we don't know how much farther it is to the goal from this node, we are forced to guess. The better our guess, the closer *f* is to the true value, and the quicker A* finds the goal with little wasted effort.

Additionally, A* maintains two lists, an *Open* list and a *Closed* list. The Open list consists of nodes that *have not* been explored, whereas the Closed list consists of all nodes that *have* been explored. A node is considered "explored" if the algorithm has looked at every node connected to this one, calculated their *f*, *g* and *h* values, and placed them on the Open list to be explored in the future.

Open and Closed lists are required because nodes are not unique. For example, if you start at (0,0) and move to (0,1), it is perfectly valid to move back to (0,0). You must, therefore, keep track of what nodes have been explored and created—this is what the Open and Closed lists are for. As mentioned earlier, nodes simply mark the state and progress of the search.

In pathing, this distinction becomes important because there can be many different ways to navigate to the same point. For example, if a pathway branches into two but converges again later, the algorithm must determine which branch to take.

### The Algorithm

Now let us look at A* broken down into pseudo-code.

```
1. Let $P$ = the starting point.
2. Assign $f$, $g$ and $h$ values to P.
3. Add $P$ to the Open list. At this point, $P$ is the only node on the Open list.
4. Let $B$ = the best node from the Open list (best node has the lowest $f$-value).
   a. If $B$ is the goal node, then quit—a path has been found.
   b. If the Open list is empty, then quit—a path cannot be found.
5. Let C = a valid node connected to B.
   a. Assign $f$, $g$, and $h$ values to $C$.
   b. Check whether $C$ is on the Open or Closed list.
      i. If so, check whether the new path is more efficient (lower $f$-value).
         1. If so, update the path.
      ii. Else, add $C$ to the Open list.
   c. Repeat step 5 for all valid children of $B$.
6. Repeat from step 4.
```

### A Simple Example

Certain steps within the algorithm might not make immediate sense (such as *5bi*), but for the moment, a very simple step-through should clarify most of the algorithm. Look at the example map shown in Figure 3.1.1.

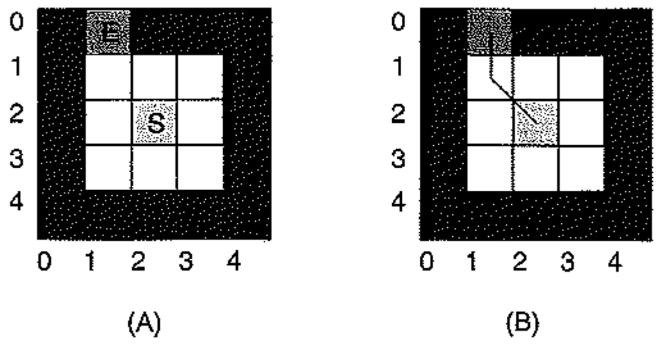

**FIGURE 3.1.1** *A) Very simple map. B) Path solution.*

The center point is the starting position (S), and the offset gray point is the end position (E). The values *f*, *g*, and *h* are simple to assign for the starting point. Value *g* is zero since there is no cost associated with the first node. Value *h* is calculated differently for each application, but for map-based problems, something simple like the combined cost of the horizontal and vertical differences (called the Manhattan Distance) is sufficient, since this is a reasonable guess for the remaining cost. Therefore, if *(dx,dy)* is the destination point and *(sx,sy)* is the starting point:

$h = | dx\text{-}sx | + | dy\text{-}sy |$

For our problem, *(sx,sy)* is (2,2) and *(dx,dy)* is (1,0), so *h* is calculated as follows:

$h = | 1-2 | + | 0-2 |$
$h = 1 + 2 = 3$

Since *h* is 3 and g is 0, then *f*, which is the sum of *g* and *h*, equals 3. Generating the children is simple since all children are valid (all eight adjacent cells can be traveled to) and all are new nodes to be added to the Open list. The *g*-value for each child node will be 1, since *g* is the cost of getting to the parent (0) plus the cost of moving a position on the map (in this case, one tile). The *h*-value will be different for each node, but it is easy to see that *(1,1)* will have the lowest score since it is the closest to our goal node. Therefore, *(1,1)* is the best child node and will be the next to be explored.

Node *(1,1)* has four valid children: *(1,0) (1,2) (2,1)* and *(2,2)*. Now we have to determine which nodes are on the Open or Closed lists, and which are new nodes. Node *(2,2)* is on the Closed list. It was our original starting point, and all its children have been opened up. Nodes *(1,2)* and *(2,1)* are on the Open list since their children have yet to be explored (and are children of *(2,2)*), and, finally, *(1,0)* is a new node.

After assigning *f*, *g*, and *h* values to the nodes, it is evident that *(1,0)* will have the best score, and upon the next iteration of the A* algorithm, it is discovered that *(1,0)* is the goal node.

## CAStar—A C++ Class for the A* Algorithm

CAStar is an example C++ class that implements the A* algorithm. It is a little more complex than a standard class, since it allows the programmer to supply his own cost and validity functions, as well as a variety of callback function pointers. We will not look at all of the class member functions; instead, we will focus on the node data structure and two important member functions. These two member functions, `LinkChild` and `UpdateParents`, handle most aspects of A*.

First, let us look at the node data structure:

```
class _asNode {
      public:
      _asNode(int, int);
      int f,g,h;
      int x,y;
      int numchildren;
```

```
        int number;

        _asNode *parent;
        _asNode *next;
        _asNode *children[8];
        void    *dataptr;
    };
```

The node data structure implemented as a mini-class to aid member variable initialization (see the source files on the CD-ROM). The member variables are self-explanatory: `f`, `g`, and `h` values, `x` and `y` variables for positional information, `numchildren` to track the number of children, and `number`, a unique identifier for each map position.

Following, we have a pointer to the parent of the node. The pointer labeled `next` is used in the Open and Closed lists (implemented as linked-lists). We then have an array of pointers to the children (pathfinding on a grid requires an array size of eight). The final variable is a void pointer that the programmers can use to associate some form of data with the node.

### CAStar::LinkChild

`LinkChild` takes two pointers to `_asNode` structures. One denotes the parent node (`node`), and the other is a temporary node (`temp`) that only has its `x` and `y` variables initialized. `LinkChild` implements steps *5a* and *5b* of the original pseudo-code.

```
void CAStar::LinkChild(_asNode *node, _asNode *temp)
{
   int x = temp->x;
   int y = temp->y;
   int g = node->g +
           udFunc(udCost, node, temp, 0, m_pCBData);
   int num = Coord2Num(x,y);
```

First, we retrieve the coordinate information from `temp`. Notice how we calculate *g* by using the parent's *g-value* and then calling the user-defined cost function, `udCost`. The last line generates the unique identifier for our node position.

```
   _asNode *check = NULL;

   if (check = CheckList(m_pOpen, num)) {
      node->children[node->numchildren++] = check;

      if (g < check->g) {
         check->parent = node;
         check->g = g;
         check->f = g + check->h;
      }
   } else if (check = CheckList(m_pClosed, num)) {
      node->children[node->numchildren++] = check;

      if (g < check->g) {
            check->parent = node;
            check->g = g;
            check->f = g + check->h;

            UpdateParents(check);
         }
      }
```

If you refer back to our pseudo-code, you will see that we must first check whether the node exists on either the Open or Closed lists. `CheckList` takes a pointer to a list head and a unique identifier to search for; if it finds the identifier, it returns the pointer of the node with which it is associated.

If it is found on the Open list, we add it to the array of `node`'s children. We then check whether the *g* calculated from the new node is smaller than `check`'s *g*. Remember that although `check` and `temp` correspond to the same position on the map, the paths by which they were reached can be *very* different.

If the node is found on the Closed list, we add it to `node`'s children. We do a similar check to see whether the *g-value* is lower. If it is, then we have to change not only the current parent pointer, but also *all* connected nodes to update their `f`, `g`, `h` values and possibly their parent pointers, too. We will look at the function that performs this after we finish with `LinkChild`.

```
      else {
         _asNode *newnode = new _asNode(x,y);
         newnode->parent = node;
         newnode->g = g;
         newnode->h = abs(x-m_iDX) + abs(y-m_iDY);
         newnode->f = newnode->g + newnode->h;
         newnode->number = Coord2Num(x,y);

         AddToOpen(newnode);

         node->children[node->numchildren++] = newnode;
      }
   }
```

Finally, if it is neither on the Open or Closed list, we create a new node and assign the `f`, `g`, and `h` values. We then add it to the Open list before updating its parent's child pointer array.

### CAStar::UpdateParents

`UpdateParents` takes a node as its single parameter and propagates the necessary changes up the A* tree. This implements step *5bi1* of our algorithm!

```
void CAStar::UpdateParents(_asNode *node)
{
   int g = node->g, c = node->numchildren;

   _asNode *kid = NULL;
   for (int i=0;i<c;i++) {
      kid = node->children[i];
      if (g+1 < kid->g) {
         kid->g = g+1;
         kid->f = kid->g + kid->h;
         kid->parent = node;

         Push(kid);
      }
   }
```

This is the first half of the algorithm. It is fairly easy to see *what* the algorithm does. The question is, *why* does it do it? Remember that node's *g*-value was updated before the call to UpdateParents. Therefore, we check all children to see whether we can improve on their *g*-value as well. Since we have to propagate the changes back, any updated node is placed on a stack to be recalled in the latter half of the algorithm.

```
   _asNode *parent;
   while (m_pStack) {
      parent = Pop();
      c = parent->numchildren;
      for (int i=0;i<c;i++) {
         kid = parent->children[i];

         if (parent->g+1 < kid->g) {
            kid->g = parent->g +
             udFunc(udCost, parent, kid, 0, m_pCBData);
            kid->f = kid->g + kid->h;
            kid->parent = parent;

            Push(kid);
         }
      }
   }
}
```

The rest of the algorithm is basically the same as the first half, but instead of using node's values, we are popping nodes off the stack. Again, if we update a node, we must push it back onto the stack to continue the propagation.

### Utilizing CAStar

As mentioned, CAStar is expandable and can be easily adapted to other applications. The main advantage of CAStar is that the programmer supplies the cost and validity functions. This means that CAStar is almost ready to go for any 2D map problems.

The programmer supplies the cost and validity functions, as well as two optional notification functions by passing function pointers of the following prototype:

```
typedef int(*_asFunc)(_asNode *, _asNode *, int, void *);
```

The first two parameters are the parent and child nodes. The integer is a function-specific data item (used in callback functions), and the final pointer is the `m_pCBData` (cost and validity functions) or `m_pNCData` (notification functions) as defined by the programmer. See the A* Explorer source code and documentation on the CD-ROM for examples on how to use these features effectively.

ON THE CD

## A* Explorer

A* Explorer is a Windows program that utilizes CAStar and allows the user to explore many aspects of the A* algorithm. For example, if you would like to look at how A* solves the simple map given at the beginning of this chapter in Figure 3.1.1, do the following:

ON THE CD

1. Run A* Explorer off of the book's CD.
2. Select "File, Open," and find *very_simple.ase* in A* Explorer's directory.
3. Use the Step function (F10) to step through each iteration of A*. Look at the Open and Closed lists, as well as the A* tree itself.

Alternatively, if you would like to see how relative costing (as described in the next section) affects A*'s final path, open *relative_cost.ase* and run the A* algorithm (F9). Now, select "Relative Costing" within the "Pathing" menu and re-run A*. Notice how the path changes.

A* Explorer has a huge number of features that we don't have room to cover here, so take a look at the online help for a complete breakdown of the features, including breakpoint and conditions, map drawing, and understanding the A* tree.

## Ideas and Expansions

The A* algorithm is great because it is highly extensible and will often bend around your problem easily. The key to getting A* to work optimally lies within the cost and heuristic functions. These functions can also yield more realistic behavior if tuned properly. As a simple example, if a node's altitude determines the cost of its position, the cost function could favor traversing children of equal altitude (relative costing) as opposed to minimizing the cost (Figure 3.1.2).

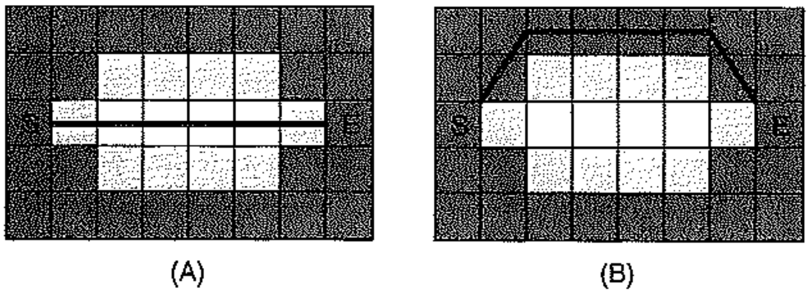

**FIGURE 3.1.2** *A) Path generated by normal costing, and B) relative costing.*

Conclusi

A* Resou

This is a good example of how altering the cost function yields more realistic behavior (in certain scenarios). By adapting the distance and child generation functions, it is easy to stretch A* to non-map specific problems. Other ideas for enthusiastic readers to explore include hexagonal or three-dimensional map support, optimizing the algorithm, and experimenting with different cost and distance functions.

Of course, one of the best places to look for additional ideas and expansions to A* lies within the other articles of this book and the *Game Programming Gems* series of books!

## Conclusion

A* is a difficult algorithm to fully understand. On paper, it looks simple, when looking at someone else's code, it *still* looks simple—but understanding it completely can be a daunting task. Hopefully, after reading this chapter, you will understand how A* works, its potential applications, and ideas on expanding and improving it. Use CAStar and A* Explorer to help further your experience and knowledge of A*.

## A* Resources on the Internet

| | |
|---|---|
| **Generation5:** | www.generation5.org/ |
| **The Game AI Page:** | www.gameai.com/ |
| **Flipcode:** | www.flipcode.com/ |