The A\* (pronounced A-star) algorithm can be complicated for beginners. While there are many articles on the web that explain A\*, most are written for people who understand the basics already. This article is for the true beginner.

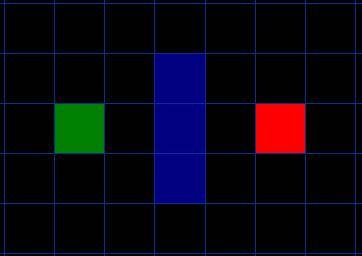
This article does not try to be the definitive work on the subject. Instead it describes the fundamentals and prepares you to go out and read all of those other materials and understand what they are talking about. Links to some of the best are provided at the end of this article, under Further Reading.

Finally, this article is not program-specific. You should be able to adapt what's here to any computer language. As you might expect, however, I have included a link to a sample program at the end of this article. The sample package contains two versions: one in C++ and one in Blitz Basic. It also contains executables if you just want to see A\* in action.

But we are getting ahead of ourselves. Let's start at the beginning ...

**Introduction: The Search Area**

Let’s assume that we have someone who wants to get from point A to point B. Let’s assume that a wall separates the two points. This is illustrated below, with green being the starting point A, and red being the ending point B, and the blue filled squares being the wall in between.

    
[Figure 1]

The first thing you should notice is that we have divided our search area into a square grid. Simplifying the search area, as we have done here, is the first step in pathfinding. This particular method reduces our search area to a simple two dimensional array. Each item in the array represents one of the squares on the grid, 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 person moves from the center of one square to the center of the next until the target is reached.

These center points are called “nodes”. When you read about pathfinding elsewhere, you will often see people discussing nodes. Why not just call them squares? Because it is possible to divide up your pathfinding area into something other than squares. They could be rectangles, hexagons, triangles, or any shape, really. And the nodes could be placed anywhere within the shapes – in the center or along the edges, or anywhere else. We are using this system, however, because it is the simplest.

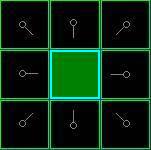
**Starting the Search**

Once we have simplified our search area into a manageable number of nodes, as we have done with the grid layout above, the next step is to conduct a search to find the shortest path. We do this by starting at point A, checking the adjacent squares, and generally searching outward until we find our target.

We begin the search by doing the following:

1. Begin at the starting point A and add it to an “open list” of squares to be considered. The open list is kind of like a shopping list. Right now there is just one item on the list, but we will have more later. It contains squares that might fall along the path you want to take, but maybe not. Basically, this is a list of squares that need to be checked out.
2. Look at all the reachable or walkable squares adjacent to the starting point, ignoring squares with walls, water, or other illegal terrain. Add them to the open list, too. For each of these squares, save point A as its “parent square”. This parent square stuff is important when we want to trace our path. It will be explained more later.
3. Drop the starting square A from your open list, and add it to a “closed list” of squares that you don’t need to look at again for now.

At this point, you should have something like the following illustration. In this illustration, the dark green square in the center is your starting square. It is outlined in light blue to indicate that the square has been added to the closed list. All of the adjacent squares are now on the open list of squares to be checked, and they are outlined in light green. Each has a gray pointer that points back to its parent, which is the starting square.

   
  [Figure 2]

Next, we choose one of the adjacent squares on the open list and more or less repeat the earlier process, as described below. But which square do we choose? The one with the lowest F cost.

**Path Scoring**

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

F = G + H

where

* G = the movement cost to move from the starting point A to a given square on the grid, following the path generated to get there.
* H = the estimated movement cost to move from that given square on the grid to the final destination, point B. This is often referred to as the heuristic, which can be a bit confusing. The reason why it is called that is because it is a guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). You are given one way to calculate H in this tutorial, but there are many others that you can find in other articles on the web.

Our path is generated by repeatedly going through our open list and choosing the square with the lowest F score. This process will be described in more detail a bit further in the article. First let’s look more closely at how we calculate the equation.

As described above, G is the movement cost to move from the starting point to the given square using the path generated to get there. In this example, we will assign a cost of 10 to each horizontal or vertical square moved, and a cost of 14 for a diagonal move. We use these numbers because the actual distance to move diagonally is the square root of 2 (don’t be scared), or roughly 1.414 times the cost of moving horizontally or vertically. We use 10 and 14 for simplicity’s sake. The ratio is about right, and we avoid having to calculate square roots and we avoid decimals. This isn’t just because we are dumb and don’t like math. Using whole numbers like these is a lot faster for the computer, too. As you will soon find out, pathfinding can be very slow if you don’t use short cuts like these.

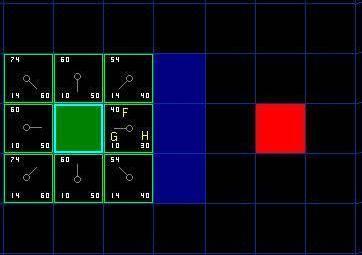
Since we are calculating the G cost along a specific path to a given square, the way to figure out the G cost of that square is to take the G cost of its parent, and then add 10 or 14 depending on whether it is diagonal or orthogonal (non-diagonal) from that parent square. The need for this method will become apparent a little further on in this example, as we get more than one square away from the starting square.

H can be estimated in a variety of ways. The method we use here is called the Manhattan method, where you calculate the total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way. We then multiply the total by 10, our cost for moving one square horizontally or vertically. This is (probably) called the Manhattan method because it is like calculating the number of city blocks from one place to another, where you can’t cut across the block diagonally.

Reading this description, you might guess that the heuristic is merely a rough estimate of the remaining distance between the current square and the target "as the crow flies." This isn't the case. We are actually trying to estimate the remaining distance along the path (which is usually farther). The closer our estimate is to the actual remaining distance, the faster the algorithm will be. If we overestimate this distance, however, it is not guaranteed to give us the shortest path. In such cases, we have what is called an "inadmissible heuristic."

Technically, in this example, the Manhattan method is inadmissible because it slightly overestimates the remaining distance. But we will use it anyway because it is a lot easier to understand for our purposes, and because it is only a slight overestimation. On the rare occasion when the resulting path is not the shortest possible, it will be nearly as short. Want to know more? You can find equations and additional notes on heuristics [here](http://www.policyalmanac.org/games/heuristics.htm).

F is calculated by adding G and H. The results of the first step in our search can be seen in the illustration below. The F, G, and H scores are written in each square. As is indicated in the square to the immediate right of the starting square, F is printed in the top left, G is printed in the bottom left, and H is printed in the bottom right.

   
  [Figure 3]

So let’s look at some of these squares. In the square with the letters in it, G = 10. This is because it is just one square from the starting square in a horizontal direction. The squares immediately above, below, and to the left of the starting square all have the same G score of 10. The diagonal squares have G scores of 14.

The H scores are calculated by estimating the Manhattan distance to the red target square, moving only horizontally and vertically and ignoring the wall that is in the way. Using this method, the square to the immediate right of the start is 3 squares from the red square, for a H score of 30. The square just above this square is 4 squares away (remember, only move horizontally and vertically) for an H score of 40. You can probably see how the H scores are calculated for the other squares.

The F score for each square, again, is simply calculated by adding G and H together.

**Continuing the Search**

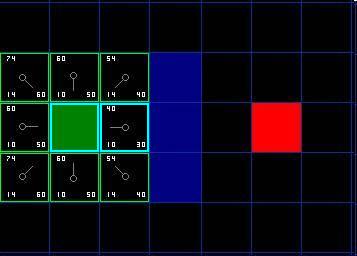
To continue the search, we simply choose the lowest F score square from all those that are on the open list. We then do the following with the selected square:

4) Drop it from the open list and add it to the closed list.

5) Check all of the adjacent squares. Ignoring those that are on the closed list or unwalkable (terrain with walls, water, or other illegal terrain), add squares to the open list if they are not on the open list already. Make the selected square the “parent” of the new squares.

6) If an adjacent square is already on the open list, check to see if this path to that square is a better one. In other words, check to see if the G score for that square is lower if we use the current square to get there. If not, don’t do anything.   
    On the other hand, if the G cost of the new path is lower, change the parent of the adjacent square to the selected square (in the diagram above, change the direction of the pointer to point at the selected square). Finally, recalculate both the F and G scores of that square. If this seems confusing, you will see it illustrated below.

Okay, so let’s see how this works. Of our initial 9 squares, we have 8 left on the open list after the starting square was switched to the closed list.  Of these, the one with the lowest F cost is the one to the immediate right of the starting square, with an F score of 40. So we select this square as our next square. It is highlight in blue in the following illustration.

    
[Figure 4]

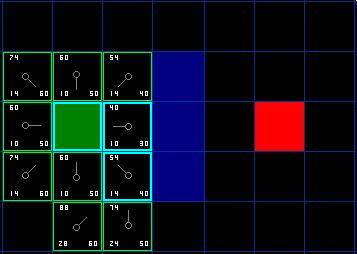
First, we drop it from our open list and add it to our closed list (that’s why it’s now highlighted in blue).  Then we check the adjacent squares. Well, the ones to the immediate right of this square are wall squares, so we ignore those. The one to the immediate left is the starting square. That’s on the closed list, so we ignore that, too.

The other four squares are already on the open list, so we need to check if the paths to those squares are any better using this square to get there, using G scores as our point of reference. Let’s look at the square right above our selected square. Its current G score is 14. If we instead went through the current square to get there, the G score would be equal to 20 (10, which is the G score to get to the current square, plus 10 more to go vertically to the one just above it). A G score of 20 is higher than 14, so this is not a better path. That should make sense if you look at the diagram. It’s more direct to get to that square from the starting square by simply moving one square diagonally to get there, rather than moving horizontally one square, and then vertically one square.

When we repeat this process for all 4 of the adjacent squares already on the open list, we find that none of the paths are improved by going through the current square, so we don’t change anything. So now that we looked at all of the adjacent squares, we are done with this square, and ready to move to the next square.

So we go through the list of squares on our open list, which is now down to 7 squares, and we pick the one with the lowest F cost. Interestingly, in this case, there are two squares with a score of 54. So which do we choose? It doesn’t really matter. For the purposes of speed, it can be faster to choose the last one you added to the open list. This biases the search in favor of squares that get found later on in the search, when you have gotten closer to the target. But it doesn’t really matter. (Differing treatment of ties is why two versions of A\* may find different paths of equal length.)

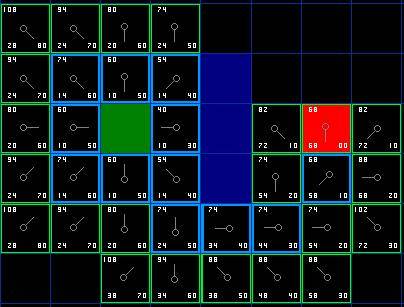
So let’s choose the one just below, and to the right of the starting square, as is shown in the following illustration.

    
[Figure 5]

This time, when we check the adjacent squares we find that the one to the immediate right is a wall square, so we ignore that. The same goes for the one just above that. We also ignore the square just below the wall. Why? Because you can’t get to that square directly from the current square without cutting across the corner of the nearby wall. You really need to go down first and then move over to that square, moving around the corner in the process. (Note: This rule on cutting corners is optional. Its use depends on how your nodes are placed.)

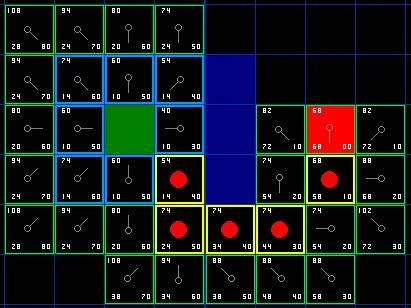
That leaves five other squares.  The other two squares below the current square aren’t already on the open list, so we add them and the current square becomes their parent. Of the other three squares, two are already on the closed list (the starting square, and the one just above the current square, both highlighted in blue in the diagram), so we ignore them. And the last square, to the immediate left of the current square, is checked to see if the G score is any lower if you go through the current square to get there. No dice. So we’re done and ready to check the next square on our open list.

We repeat this process until we add the target square to the closed list, at which point it looks something like the illustration below.

    
[Figure 6]

Note that the parent square for the square two squares below the starting square has changed from the previous illustration. Before it had a G score of 28 and pointed back to the square above it and to the right. Now it has a score of 20 and points to the square just above it. This happened somewhere along the way on our search, where the G score was checked and it turned out to be lower using a new path – so the parent was switched and the G and F scores were recalculated. While this change doesn’t seem too important in this example, there are plenty of possible situations where this constant checking will make all the difference in determining the best path to your target.

So how do we determine the path? Simple, just start at the red target square, and work backwards moving from one square to its parent, following the arrows. This will eventually take you back to the starting square, and that’s your path. It should look like the following illustration. Moving from the starting square A to the destination square B is simply a matter of moving from the center of each square (the node) to the center of the next square on the path, until you reach the target.

    
[Figure 7]

**Summary of the A\* Method**

Okay, now that you have gone through the explanation, let’s lay out the step-by-step method all in one place:

1) Add the starting square (or node) to the open list.

2) Repeat the following:

a) Look for the lowest F cost square on the open list. We refer to this as the current square.

b) Switch it to the closed list.

c) For each of the 8 squares adjacent to this current square …

* If it is not walkable or if it is on the closed list, ignore it. Otherwise do the following.
* If it isn’t on the open list, add it to the open list. Make the current square the parent of this square. Record the F, G, and H costs of the square.
* If it is on the open list already, check to see if this path to that square is better, using G cost as the measure. A lower G cost means that this is a better path. If so, change the parent of the square to the current square, and recalculate the G and F scores of the square. If you are keeping your open list sorted by F score, you may need to resort the list to account for the change.

d) Stop when you:

* Add the target square to the closed list, in which case the path has been found (see note below), or
* Fail to find the target square, and the open list is empty. In this case, there is no path.

3) Save the path. Working backwards from the target square, go from each square to its parent square until you reach the starting square. That is your path.

Note: In earlier versions of this article, it was suggested that you can stop when the target square (or node) has been added to the open list, rather than the closed list.  Doing this will be faster and it will almost always give you the shortest path, but not always.  Situations where doing this could make a difference are when the movement cost to move from the second to the last node to the last (target) node can vary significantly -- as in the case of a river crossing between two nodes, for example.

As was noted in the main article, there are several ways that you can calculate the heuristic in A\*. I describe a few here. You should also check out the links at the bottom of this page.

**Manhattan Method**

This is the method used in the main article. Its primary advantage is that it will generally get you to the target faster than most alternatives. Its primary drawback is that in a 8-way pathfinder like the one in the main example, this method is inadmissible, so it is not guaranteed to give us the shortest possible path. It will almost always give the shortest path, though, and when it doesn't it won't be far off. Also, because it is overweighted compared to G, it will overpower any small modifiers that you try to add to the calculation of G like, say, a turning penalty or an influence map modifier. If you don't have those, this method is probably your best bet. The equation, where abs = absolute value, is:

H = 10\*(abs(currentX-targetX) + abs(currentY-targetY))

public EmployeeGeneralEntityFormController(IEmployeeGeneralEntityForm form)

{

try

{

\_form = form;

GetTranslations(\_form.Form, \_form.ToolTpController);

InitializeControls();

}

catch (CAReException ex)

{

TRACE.Error(ex.Message, ex);

// **ToDo: Display an error message**

}

}

public AppointmentGeneralEntityForm()

{

InitializeComponent();

if (!DesignMode)

{

\_controller = ControllerFactory.GetInstance().GetAppointmentGeneralEntityFormController(this);

Load += AppointmentGeneralEntityForm\_Load;

}

}

#endregion

public DayRegistrationFormController CreateDayRegistrationFormController(IDayRegistrationForm dayRegistrationForm)

{

DayRegistrationFormController(dayRegistrationForm);

}

**Diagonal Shortcut**

This method balances H with G. Its primary advantage is that, because it is balanced, it is able to fully consider small modifiers like a turning penalty or influence map modifier. It is also admissible. It is a bit slower than the Manhattan method, though. The equation, where abs = absolute value, is:

xDistance = abs(currentX-targetX)  
yDistance = abs(currentY-targetY)  
if xDistance > yDistance  
     H = 14\*yDistance + 10\*(xDistance-yDistance)  
else  
     H = 14\*xDistance + 10\*(yDistance-xDistance)  
end if

There are many other methods for calculating H, although these are probably two of the most commonly used. In general, the closer your estimate is to the actual remaining distance along the path to the goal, the faster A\* will find that goal. If you overestimate H compared to the actual remaining distance, however, A\* is not guaranteed to give you the best path. The trick, then, is to come as close to the actual distance as possible, without overestimating.