The heuristic function `h(n)` tells A* an *estimate* of the minimum cost from any vertex `n` to the goal. It's important to choose a good heuristic function.

A*'s Use of the Heuristic

The heuristic can be used to control A*'s behavior.

- At one extreme, if `h(n)` is 0, then only `g(n)` plays a role, and A* turns into Dijkstra's algorithm, which is guaranteed to find a shortest path.
- If `h(n)` is always lower than (or equal to) the cost of moving from `n` to the goal, then A* is guaranteed to find a shortest path. The lower `h(n)` is, the more node A* expands, making it slower.
- If `h(n)` is exactly equal to the cost of moving from `n` to the goal, then A* will only follow the best path and never expand anything else, making it very fast. Although you can't make this happen in all cases, you can make it exact in some special cases. It's nice to know that given perfect information, A* will behave perfectly.
- If `h(n)` is sometimes greater than the cost of moving from `n` to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
- At the other extreme, if `h(n)` is very high relative to `g(n)`, then only `h(n)` plays a role, and A* turns into Best-First-Search.

So we have an interesting situation in that we can decide what we want to get out of A*. At exactly the right point, we'll get shortest paths really quickly. If we're too low, then we'll continue to get shortest paths, but it'll slow down. If we're too high, then we give up shortest paths, but A* will run faster.

In a game, this property of A* can be very useful. For example, you may find that in some situations, you would rather have a "good" path than a "perfect" path. To shift the balance between `g(n)` and `h(n)`, you can modify either one.

N
Technically,
should be call
heuristic is an
the actual cos
continue to cal
implementatic
the game
community do
A fi

Speed or accuracy?

A*'s ability to vary its behavior based on the heuristic and cost functions can be very useful in a game. The tradeoff between speed and accuracy can be exploited to make your game faster. For most games, you don't *really* need the **best** path between two points. You just need something that's close. What you need may depend on what's going on in the game, or how fast the computer is.

Suppose your game has two types of terrain, Flat and Mountain, and the movement costs are 1 for flat land and 3 for mountains, A* is going to search three times as far along flat land as it does along mountainous land. This is because it's *possible* that there is a path along flat terrain that goes around the mountains. You can speed up A*'s search by using 1.5 as the heuristic distance between two map spaces. A* will then compare 3 to 1.5, and it won't look as bad as comparing 3 to 1. It is not as dissatisfied with mountainous terrain, so it won't spend as much time trying to find a way around it. Alternatively, you can speed up up A*'s search by decreasing the amount it searches for paths around mountains—just tell A* that the movement cost on mountains is 2 instead of 3. Now it will search only twice as far along the flat terrain as along mountainous terrain. Either approach gives up ideal paths to get something quicker.

The choice between speed and accuracy does not have to be static. You can choose dynamically based on the CPU speed, the fraction of time going into pathfinding, the number of units on the map, the importance of the unit, the size of the group, the difficulty level, or any other factor. One way to make the tradeoff dynamic is to build a heuristic function that assumes the minimum cost to travel one grid space is 1 and then build a cost function that scales:

```
g'(n) = 1 + alpha * (g(n) - 1)
```

If `alpha` is 0, then the modified cost function will always be 1. At this setting, terrain costs are completely ignored, and A* works at the level of simple passable/unpassable grid spaces. If `alpha` is 1, then the original cost function will be used, and you get the full benefit of A*. You can set `alpha` anywhere in between.

You should also consider switching from the heuristic returning the *absolute* minimum cost to

returning the *expected* minimum cost. For example, if most of your map is grasslands with a movement cost of 2 but some spaces on the map are roads with a movement cost of 1, then you might consider having the heuristic assume no roads, and return `2 * distance`.

The choice between speed and accuracy does not have to be global. You can choose some things dynamically based on the importance of having accuracy in some region of the map. For example, it may be more important to choose a good path near the current location, on the assumption that we might end up recalculating the path or changing direction at some point, so why bother being accurate about the faraway part of the path? Or perhaps it's not so important to have the shortest path in a safe area of the map, but when sneaking past an enemy village, safety and quickness are essential.

Scale

A* computes `f(n) = g(n) + h(n)`. To add two values, those two values need to be at the same scale. If `g(n)` is measured in hours and `h(n)` is measured in meters, then A* is going to consider `g` or `h` too much or too little, and you either won't get as good paths or you A* will run slower than it could.

Exact heuristics

If your heuristic is exactly equal to the distance along the optimal path, you'll see A* expand very few nodes, as in the diagram shown in the next section. What's happening inside A* is that it is computing `f(n) = g(n) + h(n)` at every node. When `h(n)` exactly matches `g(n)`, the value of `f(n)` doesn't change along the path. All nodes not on the right path will have a higher value of `f` than nodes that are on the right path. Since A* doesn't consider higher-valued `f` nodes until it has considered lower-valued `f` nodes, it never strays off the shortest path.

Precomputed exact heuristic

One way to construct an exact heuristic is to precompute the length of the shortest path between every pair of points. This is not feasible for most game maps. However, there are ways to approximate this heuristic:

- Fit a coarse grid on top of the fine grid. Precompute the shortest path between any pair of coarse grid locations.
- Precompute the shortest path between any pair of waypoints. This is a generalization of the coarse grid approach.

Then add in a heuristic `h'` that estimates the cost of going from any location to nearby waypoints. (The latter too can be precomputed if desired.) The final heuristic will be:

```
h(n) = h'(n, w1) + distance(w1, w2) + h'(w2, goal)
```

or if you want a better but more expensive heuristic, evaluate the above with all pairs `w1, w2` that are close to the node and the goal, respectively.

Linear exact heuristic

In a special circumstance, you can make the heuristic exact without precomputing anything. If you have a map with no obstacles and no slow terrain, then the shortest path from the starting point to the goal should be a straight line.

If you're using a simple heuristic (one which does not know about the obstacles on the map), it should match the exact heuristic. If it doesn't, then you may have a problem with scale or the type of heuristic you chose.

Heuristics for grid maps

On a grid, there are well-known heuristic functions to use.

Use the distance heuristic that matches the allowed movement:

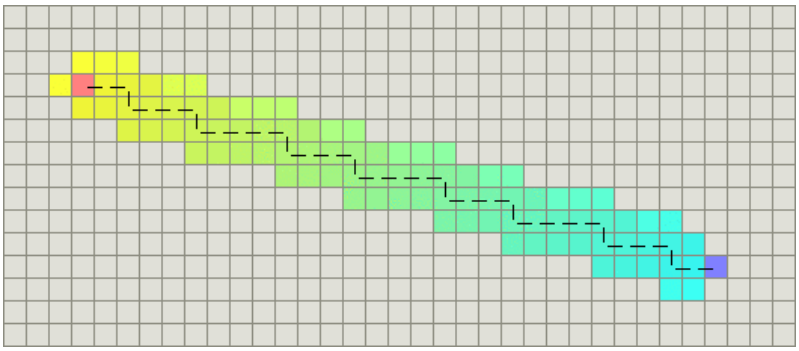
- On a square grid that allows **4 directions** of movement, use Manhattan distance (L_1).
- On a square grid that allows **8 directions** of movement, use Diagonal distance (L_∞).
- On a square grid that allows **any direction** of movement, you might or might not want Euclidean distance (L_2). If A* is finding paths on the grid but you are allowing movement not on the grid, you may want to consider other representations of the map.
- On a hexagon grid that allows **6 directions** of movement, use Manhattan distance adapted to hexagonal grids.

Manhattan distance

The standard heuristic for a square grid is the Manhattan distance. Look at your cost function and find the minimum cost `D` for moving from one space to an adjacent space. Therefore, the heuristic in my game should be `D` times the Manhattan distance:

```
h(n) = D * (abs(n.x-goal.x) + abs(n.y-goal.y))
```

You should use the scale that matches your cost function.

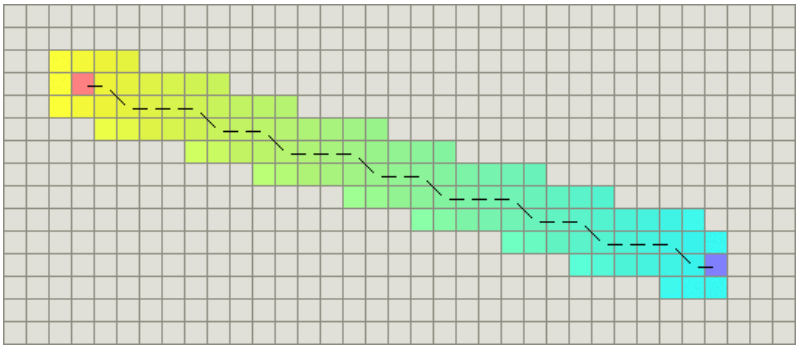


(Note: the above image has a tie-breaker added to the heuristic.}

Diagonal distance

If on your map you allow diagonal movement you need a different heuristic (sometimes called the Chebyshev distance). The Manhattan distance for (4 east, 4 north) will be 8*D. However, you could simply move (4 northeast) instead, so the heuristic should be 4*D. This function handles diagonals, assuming that both straight and diagonal movement costs D:

```
h(n) = D * max(abs(n.x-goal.x), abs(n.y-goal.y))
```



If the cost of diagonal movement is not D, but something like D2 = sqrt(2)*D, the above heuristic won't be right for you. You will instead want something more sophisticated:

```
h_diagonal(n) = min(abs(n.x-goal.x), abs(n.y-goal.y))
h_straight(n) = (abs(n.x-goal.x) + abs(n.y-goal.y))
h(n) = D2 * h_diagonal(n) + D * (h_straight(n) - 2*h_diagonal(n))
```

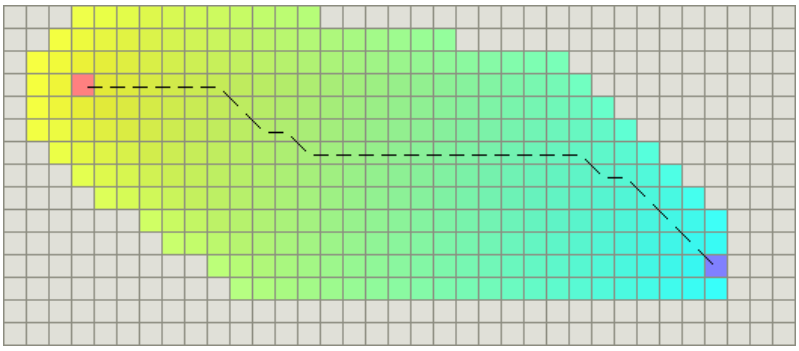
Here we compute `h_diagonal(n)` = the number of steps you can take along a diagonal, `h_straight(n)` = the Manhattan distance, and then combine the two by considering all diagonal steps to cost D2, and then all remaining straight steps (note that this is the number of straight steps in the Manhattan distance, minus two straight steps for each diagonal step we took instead) cost D.

Euclidean distance

If your units can move at any angle (instead of grid directions), then you should probably use a straight line distance:

```
h(n) = D * sqrt((n.x-goal.x)^2 + (n.y-goal.y)^2)
```

However, if this is the case, then you may have trouble with using A* directly because the cost function `g` will not match the heuristic function `h`. Since Euclidean distance is shorter than Manhattan or diagonal distance, you will still get shortest paths, but A* will take longer to run:

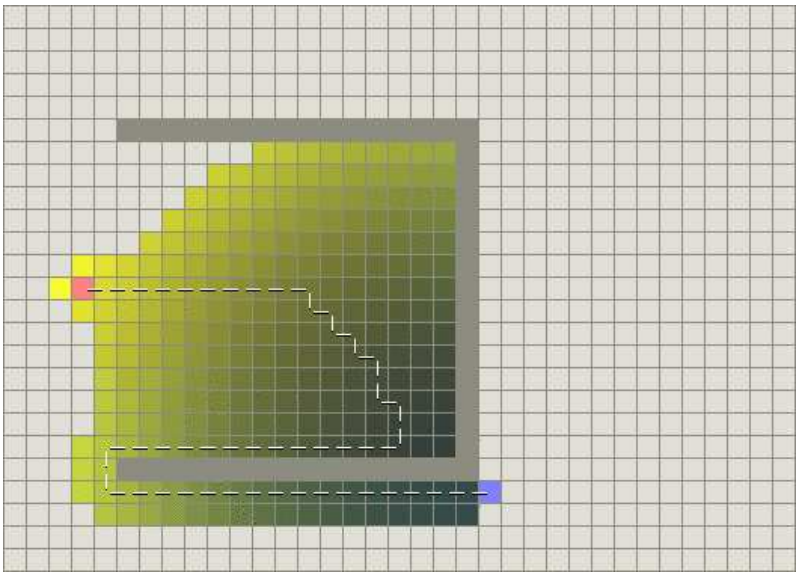


Euclidean distance, squared

I've seen several A* web pages recommend that you avoid the expensive square root in the Euclidean distance by just using distance-squared:

```
h(n) = D * ((n.x-goal.x)^2 + (n.y-goal.y)^2)
```

Do not do this! This definitely runs into the scale problem. When A* computes `f(n) = g(n) + h(n)`, the square of distance will be much higher than the cost `g` and you will end up with an overestimating heuristic. For longer distances, this will approach the extreme of `g(n)` not contributing to `f(n)`, and A* will degrade into (Greedy) Best-First-Search:



To attempt to fix this you can scale the heuristic down. However, then you run into the opposite problem: for shorter distances, the heuristic will be too small compared to `g(n)` and A* will degrade into Dijkstra's algorithm.

If, after profiling, you find the cost of the square root is significant, either use a fast square root approximation with Euclidean distance or use the diagonal distance as an approximation to Euclidean.

Breaking ties

In some maps there are many paths with the same length. For example, in flat areas without variation in terrain, using a grid will lead to many equal-length paths. A* might explore all the paths with the same `f` value, instead of just one. (If you do have lots of such areas, there might be other techniques that are better than using A* on a grid.)

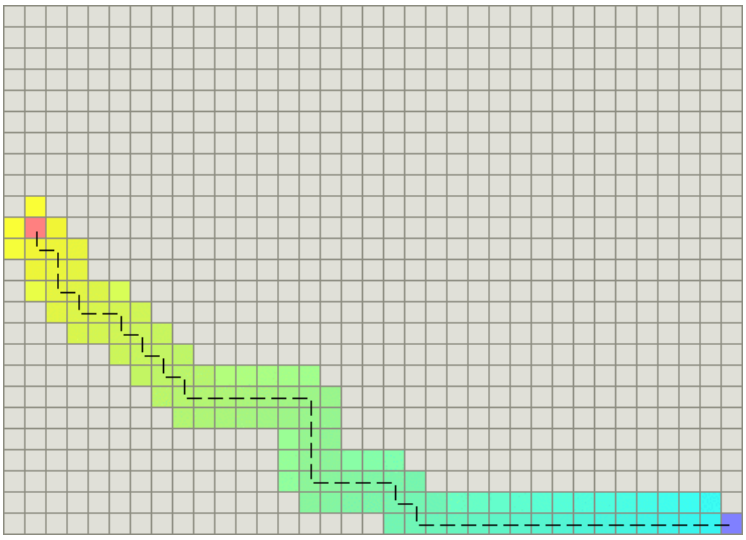
Ties in `f` values.

To solve this problem, we need to either adjust the `g` or `h` values; it is usually easier to adjust `h`. The tie breaker needs to be deterministic with respect to the vertex (*i.e.,* it shouldn't just be a random number), and it needs to make the `f` values differ. Since A* sorts by `f` value, making them different means only one of the "equivalent" `f` values will be explored.

One way to break ties is to nudge the scale of `h` slightly. If we scale it downwards, then `f` will increase as we move towards the goal. Unfortunately, this means that A* will prefer to expand vertices close to the starting point instead of vertices close to the goal. We can instead scale `h` upwards slightly (even by 0.1%). A* will prefer to expand vertices close to the goal.

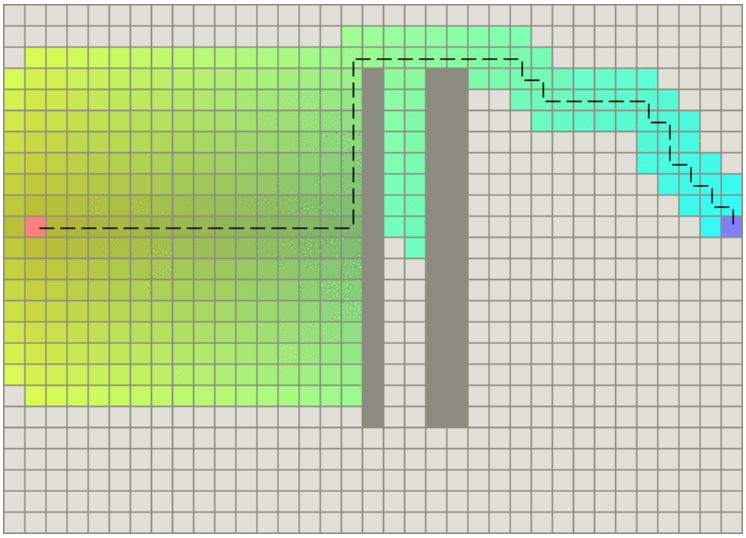
```
heuristic *= (1.0 + p)
```

The factor `p` should be chosen so that `p` < *(minimum cost of taking one step) / (expected maximum path length)*. Assuming that you don't expect the paths to be more than 1000 steps long, you can choose p = 1/1000. The result of this tie-breaking nudge is that A* explores far less of the map than previously:



Tie-breaking scaling added to heuristic.

When there are obstacles of course it still has to explore to find a way around them, but note that after the obstacle is passed, A* explores very little:



Tie-breaking scaling added to heuristic, works nicely with obstacles.

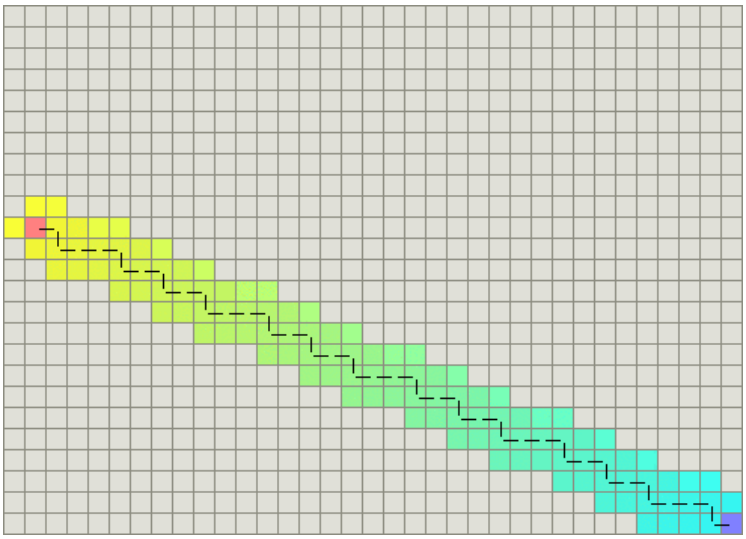
Steven van Dijk suggests that a more straightforward way to do this would to pass `h` to the comparison function. When the `f` values are equal, the comparison function would break the tie by looking at `h`.

Another way to break ties is to compute a hash of the coordinates, and use that to adjust the heuristic. This breaks more ties than adjusting `h` as above. Thanks to Cris Fuhrman for suggesting this.

A different way to break ties is to prefer paths that are along the straight line from the starting point to the goal:

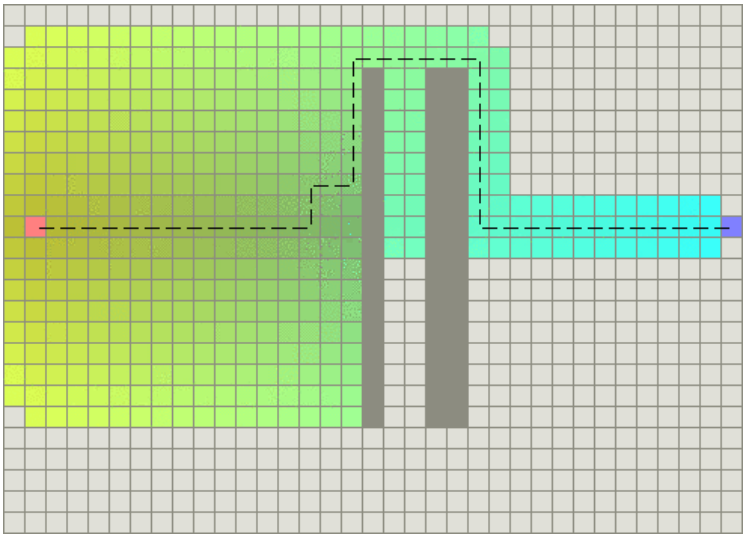
```
dx1 = current.x - goal.x
dy1 = current.y - goal.y
dx2 = start.x - goal.x
dy2 = start.y - goal.y
cross = abs(dx1*dy2 - dx2*dy1)
heuristic += cross*0.001
```

This code computes the vector cross-product between the start to goal vector and the current point to goal vector. When these vectors don't line up, the cross product will be larger. The result is that this code will give some slight preference to a path that lies along the straight line path from the start to the goal. When there are no obstacles, A* not only explores less of the map, the path looks very nice as well:



Tie-breaking cross-product added to heuristic, produces pretty paths.

However, because this tie-breaker prefers paths along the straight line from the starting point to the goal, weird things happen when going around obstacles (note that the path is still optimal; it just looks strange):



Tie-breaking cross-product added to heuristic, less pretty with obstacles.

To interactively explore the improvement from this tie breaker, see James Macgill's A* applet [if that link does not work, try this mirror]. Use "Clear" to clear the map, and choose two points on opposite corners of the map. When you use the "Classic A*" method, you will see the effect of ties. When you use the "Fudge" method, you will see the effect of the above cross product added to the heuristic.

Yet another way to break ties is to carefully construct your A* priority queue so that *new* insertions with a specific `f` value are always ranked better (lower) than *old* insertions with the same `f` value.

You may also want to read about the AlphA* algorithm, which has a more sophisticated way to break ties, yet still maintain bounds on the optimality of the resulting path. AlphA* is adaptive and is likely to perform better than either of the two tie-breakers I described above. However, the tie-breakers I described are extremely easy to implement, so start with them first, and try AlphA* if you need something better.

Searching for an area

If you want to search for any spot near some goal, instead of one particular space, you could construct a heuristic `h'(x)` that is the minimum of `h1(x), h2(x), h3(x), ...` where `h1, h2, h3` are heuristics to each of the nearby spots. However, a faster way is to just let A* search for the center of the goal area. Once you pull *any* nearby space from the OPEN set, you can stop and construct a path.

Back: Introduction Up: Home Next: Implementation

Email me at amitp@cs.stanford.edu, or post a public comment:

From Amit's Game Programming Site
Last modified: Sat Jan 29 12:59:41 2011