# Amit’s A* Pages

From Red Blob Games

Home Blog Links Twitter About Search

The problem we’re trying to solve is to get a game object from the starting point to a goal. *Pathfinding* addresses the problem of finding a good path from the starting point to the goal—avoiding obstacles, avoiding enemies, and minimizing costs (fuel, time, distance, equipment, money, etc.). *Movement* addresses the problem of taking a path and moving along it. It’s possible to spend your efforts on only one of these. At one extreme, a sophisticated pathfinder coupled with a trivial movement algorithm would find a path when the object begins to move and the object would follow that path, oblivious to everything else. At the other extreme, a movement-only system would not look ahead to find a path (instead, the initial “path” would be a straight line), but instead take one step at a time, considering the local environment at every point. Best results are achieved by using both pathfinding and movement algorithms.

## Pathfinding #

## Other topics #

There are many other topics related to pathfinding.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# Movement costs for pathfinders

From Amit’s Thoughts on Pathfinding

Home Blog Links Twitter About Search

When using a pathfinding algorithm, you may want to treat map spaces as something other than *clear* and *blocked*. Often there is more information available, such as the difficulty of moving through that area. For example, swamps and mountains may be more difficult to pass than grasslands and desert. With some algorithms, like A*, you can put this encode this information into the cost function. Listed below are some ideas for movement costs that might be useful.

## Altitude #

High altitudes (such as mountains) can have a higher movement cost than low altitudes. With this cost function, your units will try to stay in the lowlands whenever possible. For example, if the source and destination are both at high altitudes, the unit might move downhill, travel for a while, and then move back uphill.

### Moving uphill #

Instead of high altitudes having a high cost, *moving* uphill can have a high cost. This avoids the odd situation described above. With this cost function, units try to avoid moving uphill. Faced with the same situation, the unit will try to avoid moving back uphill at the end; it can do this by staying at a high altitude throughout its travels. A cost function such as this one may be good for units such as soldiers, which can move downhill easily but have a hard time going uphill.

### Moving up- or downhill #

Some units, such as tanks, have a hard time moving uphill *or* downhill. You can assign a high cost to moving downhill, and an even higher cost to moving uphill. The units will try to avoid changing altitudes.

## Terrain #

You may want different types of terrain to have different movement costs.

## Forests, mountains, and hills #

Instead of using altitudes, you may want to use terrain types, as in Civilization. Each terrain type can have a movement cost associated with it. This movement table might apply to all units, or different movement tables could be associated with each unit type. For example, soldiers might have no trouble moving through forests, but tanks might have a very hard time. A fancier method is to assign movement costs to *changing* terrain. Going from grassland into mountains could be more expensive than going from hills to mountains, which could be more expensive than going from mountains to mountains.

## Roads #

In many games, the primary purpose of roads is to make movement possible or easier. After choosing a movement cost function, you can add a road modifier to it. One possibility is to divide the cost by some constant (such as two); another is to assign a constant cost to movement along a road.

I strongly advise that you do *not* make road movement free (zero-cost). This confuses pathfinding algorithms such as A*, because it introduces the possibility that the shortest path from one point to another is along a winding road that seems to lead nowhere. The algorithm has to search a very wide area to make sure that no such roads exist. Note that in the game Civilization, railroads had zero-cost movement, but when using the “Auto Goto” function, railroads had a non-zero cost. This is evidence that a pathfinding algorithm was being used.

## Walls or other barriers #

Instead of checking both movement costs and for obstacles in your pathfinding algorithm, you can use movement costs. Just assign a very high movement cost to any obstacle. When expanding nodes (in the A* algorithm), check if the cost is too high; if it is, then throw the node out.

## Sloped Land #

Instead of using *movement* up and down hills, you might want to make movement on any hill expensive. To do this, compute the overall slope of the terrain (by looking at the maximum difference between the current tail and its neighbors), and use that as part of the movement cost. Land that is very steep will have a high cost and land that is shallow will have a low cost. This approach differs from the movement uphill/downhill cost in that it looks for *land* that is steep, while the previous approach looked for units that *move* in a steep direction. In particular, if you're on a hill and can move left or right without going up or down, the uphill/downhill approach will consider it a low cost, while this approach will consider it a high cost (because the land is steep even if you aren't going up or down).

Sloped land costs may not make sense for unit movement, but you can use pathfinding for more than finding paths for units. I use it for finding paths for roads, canals, bridges, and so on. If you want to build these items on flat land, you can take land slope into account when finding a path for a road or canal. See the section on applications for more ideas.

## Enemies and friendly units #

Another modifier can help you avoid enemy units. Using influence maps, you can keep track of areas that are near enemy or friendly units, have recently killed soldiers, have been recently explored, are close to an escape route, or have been traversed recently. (Age of Empires 2 uses influence maps to influence pathfinding.) An influence map might have a positive value for friendly units and a negative value for enemy units. By increasing the movement cost whenever you are in negative territory, you can influence your units to stay away from the enemy.

Even more complicated (and perhaps not possible with influence maps) is to look at *visibility*: is your unit visible by an enemy unit? Is your unit detectable in some other way? Is it possible for that enemy unit to fire on you?

## Marked beacons #

If your map is designed and not automatically generated, you can add extra information to it. For example, ancient trade routes often would pass particular points, which often became trading towns. These places are **beacons**, places that are known to be along good paths. The distance to a beacon would be added to the movement cost as a way to influence paths to favor beacons.

Good choices for beacons include lighthouses, cities, mountain passes, and bridges.

## Fuel consumption #

**Note:**

To keep the state space small, you need to round off the fuel value to a coarse measurement unit. Unfortunately, this makes the search less effective.

In addition to looking at the *time* it takes to go somewhere, you may want to consider the *fuel* it takes. The fuel consumption may be given more weight when the unit's fuel level is lower.

To track the fuel usage through the map, you need to use state space, as described in . The state would be the pair <location, fuel>. However, state space can become very large, so it may be worth looking at alternatives to using ordinary A*.

One alternative is A* with Bounded Costs (ABC). With ABC, you can assign a bound ("20 gallons") to a cost ("fuel").

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# User experience with shortest paths

From Amit's Thoughts on Pathfinding

What's most important in the game is the user. You want the user to have fun! You don't want him (or her) to feel like the computer is cheating, or that the game units aren't behaving properly.

## Dumb movement #

If the pathfinding doesn't work well, the user will end up moving the units manually. *Avoid this!* In Civilization, the rules for the game allowed for zero-cost movement along railroads. However, the pathfinder had a non-zero movement cost. The result was that users avoided using the pathfinder, and instead moved units manually on the railroads. In Command and Conquer, units could get stuck in "U" shaped traps so users would have to guide the units manually. A dumb pathfinder will annoy your users and make them move units themselves, so make your pathfinder decent!

## Smart movement #

Making units too smart is almost as bad as making units too dumb. If the player has to deal with fog of war but the pathfinder has access to the entire map, the units will mysteriously know where to go even though the user does not. That's a clear sign to the user that something odd is going on. On the other hand, it gives better paths. A compromise is to scale up the movement costs on unexplored areas. For example, if your normal movement costs are 1 for grass, 3 for forest, and 7 for mountains, set them differently on unexplored areas: 5 for grass, 6 for forest, 7 for mountains. The unit will take mountains vs. grass into account, but not too much; it'll be a subtle hint. Raising costs for moving through unexplored areas will also tend to make the unit stay in explored territory as much as possible. You might want to do the opposite for "scout" units: they should *prefer* unexplored areas.

Try to keep your units balanced between too dumb and too smart. The goal should be to make it match what the user might have done to move the unit around.

## Multithreading #

You can use multithreading to improve the user experience. When a unit needs a path, allow it to start moving in a straight line towards the goal, and add a request to a pathfinding queue. In another (low priority) thread, pull requests off the queue and find paths. Your units will start moving immediately, so the user won't be left wondering if something is wrong, and you won't have a high CPU load (which will slow down the rest of the game) while the path is being calculated.

## Multiple units #

If your game allows multiple units in a group to move together, try to make the movement look interesting. You can find a single path for them all to follow, and then have them all follow the path individually, but this will lead to either a line of units or units trying to pass each other. Instead, vary the paths a little so that they can walk in parallel. Alternatively, pick one "leader" unit to move along the path and have the other units use a separately programmed "follow" behavior. This following could be as simple as moving towards the leader but stay some distance away, or it could be as involved as flocking.

## Multiple waypoints #

Even given the optimal path, the player may prefer a different path. You may allow the player to mark waypoints on the path: instead of simply clicking on a destination, the player would click on two or three points along the way to the destination. (Many real-time strategy games use shift-click for this operation.) You now have three or four smaller paths to compute, and you save some time. The player also has some control over the overall path—for example, your pathfinder may have found a path to the west of some mountains, but for safety's sake, the player wants to stay to the east of the mountains (near friendly guard towers).

The main change in unit movement code will be that instead of a single destination, you will have a list of destinations. Find a path to the first destination. Once you get there, remove it from the list and find a path to the next destination. This reduces latency and improves throughput as well.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# Applications

From Amit's Thoughts on Pathfinding

Home Blog Links Twitter About Search

In addition to finding a path for a unit to move along, pathfinding can be used for several other purposes.

## Exploration #

If part of your cost function penalizes paths that are on known territory, paths are more likely to go through unexplored territory. These paths are good for scout units.

## Spying #

If part of the cost function penalizes paths near the enemy's watchtowers and other units, your unit will tend to stay in hiding. Note however that to work well, you may have to update the path periodically to take into account enemy unit movements.

## Road building #

Historically, roads have been built along paths that are often used. As the paths are used more and more often, vegetation is removed and replaced with dirt, and later with stone or other material. One application of pathfinding is to find roads. Given places that people want to go (cities, lakes, springs, sources of minerals, and so on), find paths randomly between these important locations. After finding hundreds or perhaps thousands of paths, determine which spaces on the map most often occur on paths. Turn those spaces into roads. Repeat the experiment, with the pathfinder preferring roads, and you will find more roads to build. This technique can work for multiple types of roads as well (highways, roads, dirt paths): the most commonly used spaces would become highways and less commonly used spaces would become roads or dirt paths.

Pathfinding is also used for building roads over mountains that avoid extreme slopes. This article shows how setting the movement cost to the square of the slope makes A* find a “natural” looking path up and over a mountain pass.

## Terrain analysis #

Combining influence maps, pathfinding, and line of sight can give you interesting ways to analyze terrain.

Using the same approach as road building, we can use pathfinding to determine what areas are the most likely to be traversed given some set of source and destination points. These points, and areas near them, tend to be strategically important. Clash of Civilizations uses this for their Map AI.

By further analysing the common paths we find, we can find ambush sites—locations along a path that do not have line-of-sight access to the location N steps further along the path. Placing an ambush at one of these points means the enemy will not see you until they are within distance N, so you can ambush with a large force.

## City building #

Cities often form around natural resources such as farmland or sources of mineral wealth. As people from these cities trade with each other, they need trading routes. Use pathfinding to find their trading routes, and then mark a day’s worth of travel on these routes. After a caravan travels for a day, it will need a place to stop: a perfect place for a city! Cities that lie along more than one travel route are great places for trading villages, which eventually grow into cities.

A combination of the road building and city building may be useful for producing realistic maps, either for scenarios or for randomized maps.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

---

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# AI techniques

From Amit’s Thoughts on Pathfinding

Home Blog Links Twitter About Search

Pathfinding is often associated with AI, because the A* algorithm and many other pathfinding algorithms were developed by AI researchers. Several biology-inspired AI techniques are currently popular, and I receive questions about why I don’t use them. Neural Networks model a brain learning by example—given a set of right answers, it learns the general patterns. Reinforcement Learning models a brain learning by experience—given some set of actions and an eventual reward or punishment, it learns which actions are good or bad. Genetic Algorithms model evolution by natural selection—given some set of agents, let the better ones live and the worse ones die. Typically, genetic algorithms do not allow agents to learn during their lifetimes, while neural networks allow agents to learn only during their lifetimes. Reinforcement learning allows agents to learn during their lifetimes and share knowledge with other agents.

## Neural Networks #

Neural networks are structures that can be “trained” to recognize patterns in inputs. They are a way to implement function approximation: given $y_1 = f(x_1)$, $y_2 = f(x_2)$, ..., $y_n = f(x_n)$, construct a function f’ that approximates f. The approximate function f’ is typically *smooth*: for x’ close to x, we will expect that f’(x’) is close to f’(x). Function approximation serves two purposes:

- **Size:** the representation of the approximate function can be significantly smaller than the true function.
- **Generalization:** the approximate function can be used on inputs for which we do not know the value of the function.

Neural networks typically take a vector of input values and produce a vector of output values. Inside, they train weights of “neurons”. Neural networks use *supervised learning*, in which inputs and outputs are known and the goal is to build a representation of a function that will approximate the input to output mapping.

In pathfinding, the function is f(start, goal) = path. We do not already know the output paths. We could compute them in some way, perhaps by using A*. But if we are able to compute a path given (start, goal), then we already know the function f, so why bother approximating it? There is no use in generalizing f because we know it completely. The only potential benefit would be in reducing the size of the representation of f. The representation of f is a fairly simple algorithm, which takes little space, so I don’t think that’s useful either. In addition, neural networks produce a fixed-size output, whereas paths are variable sized.

Instead, function approximation may be useful to construct components of pathfinding. It may be that the movement cost function is unknown. For example, the cost of moving across an orc-filled forest may not be known without actually performing the movement and fighting the battles. Using function approximation, each time the forest is crossed, the movement cost f(number of orcs, size of forest) could be measured and fed into the neural network. For future pathfinding sessions, the new movement costs could be used to find better paths. Even when the function is unknown, function approximation is useful primarily when the function varies from game to game. If a single movement cost applies every time someone plays the game, the game developer can precompute it beforehand.

Another function that is could benefit from approximation is the heuristic. The heuristic function in A* should estimate the minimum cost of reaching the destination. If a unit is moving along path P = $p_1$, $p_2$, ..., $p_n$, then after the path is traversed, we can feed n updates, g($p_i$, $p_n$) = (actual cost of moving from i to n), to the approximation function h. As the heuristic gets better, A* will be able to run quicker.

Neural networks, although not useful for pathfinding itself, can be used for the functions used by A*. Both movement and the heuristic are functions that can be measured and therefore fed back into the function approximation.

## Genetic Algorithms #

**Note:**

Function approximation can be transformed into a function optimization problem. To find f’(x) that approximates f(x), set g(f’) = Sum of $(f'(x)-f(x))^2$ over all input x.

Genetic Algorithms allow you to explore a space of parameters to find solutions that score well according to a “fitness function”. They are a way to implement *function optimization*: given a function g(x) (where x is typically a vector of parameter values), find the value of x that maximizes (or minimizes) g(x). This is an *unsupervised learning* problem—the right answer is not known beforehand. For pathfinding, given a starting position and a goal, x is the path between the two and g(x) is the cost of that path. Simple optimization approaches like hill-climbing will change x in ways that increase g(x). Unfortunately in some problems, you reach “local maxima”, values of x for which no nearby x has a greater value of g, but some faraway value of x is better. Genetic algorithms improve upon hill-climbing by maintaining multiple x, and using evolution-inspired approaches like mutation and cross-over to alter x. Both hill-climbing and genetic algorithms can be used to learn the best value of x. For pathfinding, however, we already have an algorithm (A*) to find the best x, so function optimization approaches are not needed.

Genetic Programming takes genetic algorithms a step further, and treats *programs* as the parameters. For example, you would breeding pathfinding *algorithms* instead of *paths*, and your fitness function would rate each algorithm based on how well it does. For pathfinding, we already have a good algorithm and we do not need to evolve a new one.

It may be that as with neural networks, genetic algorithms can be applied to some portion of the pathfinding problem. However, I do not know of any uses in this context. Instead, a more promising approach seems to be to use pathfinding, for which solutions are known, as one of many tools available to evolving agents.

## Reinforcement Learning #

Like genetic algorithms, Reinforcement Learning is an unsupervised learning problem. However, unlike genetic algorithms, agents can learn during their lifetimes; it’s not necessary to wait to see if they “live” or “die”. Also, it’s possible for multiple agents experiencing different things to share what they’ve learned. Reinforcement learning has some similarities to the core of A*. In A*, reaching the end goal is propagated back to mark all the choices that were made along the path; other choices are discarded. In reinforcement learning, every state can be evaluated and its reward (or punishment) is propagated back to mark all the choices that were made leading up to that state. The propagation is made using a value function, which is somewhat like the heuristic function in A*, except that it's updated as the agents try new things and learn what works. One of the key advantages of reinforcement learning and genetic algorithms over simpler approaches is that there is a choice made between *exploring* new things and *exploiting* the information learned so far. In genetic algorithms, the exploration via mutation; in reinforcement learning, the exploration is via exlicitly allowing the probability of choosing new actions.

As with genetic algorithms, I don't believe reinforcement learning should be used for the pathfinding problem itself, but instead as a guide for teaching agents how to behave in the game world.

---

---

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

---

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# References

From Amit’s Thoughts on Pathfinding

Home Blog Links Twitter About Search

This section is quite incomplete.

General graph searching algorithms can be used for pathfinding. Many algorithms textbooks describe graph searching algorithms that do not use heuristics (breadth-first search, depth-first search, Dijkstra’s). Reading about them may help in understanding A*, which is a variant of Dijkstra’s. Many AI textbooks will address graph searching algorithms that do use heuristics (best-first search, A*).

For non-graph-search algorithms, see John Lonningdal’s web page (via Wayback Machine, since his site is no longer up).

To learn more about unit movement after a path has been found, see Pottinger’s articles on unit movement and group movement. Also highly recommended are Craig Reynold’s pages on steering and flocking. Geoff Howland has a great article about unit movement in general.

The grid-based pathfinding competition results paper describes optimizations for A* running on unweighted grids: contraction hierarchies, subgoal graphs, jump point search, visibility graphs, compressed path databases, and more.

Patrick Lester has a page describing a two-level pathfinder. Hierarchical Planning A* (HPA*) can turn a grid representation into a simplified graph.

Clearance-based pathfinding annotates the graph with the sizes of the objects that can pass through there. This is useful if you want small units to be able to pass through an area but large units to be blocked.

There’s an interesting paper describing how to find Simplest Paths instead of Shortest Paths.

This StackOverflow question includes a summary of lots of variants of A*.

Triangulation A* converts a polygonal obstacle representation into a navigation mesh using triangles, and Triangulation Reduction A* simplifies the resulting pathfinding graph by removing nodes.

Here are some other papers I haven't classified:

Real-time heuristic search augments A* and other algorithms with additional data to speed up pathfinding.

Fringe Search looks at large sets of nodes (the fringe or frontier) at a time. They greatly reduce the cost of processing each node. A* sorts one node at a time (either on insertion and deletion from the set, or both), and batch sorting is faster. But even faster is not sorting at all. In the batches of nodes that Fringe search is processing, there's no need to sort them. The downside is that more nodes have to be processed, sometimes more than once. But if you can make processing them really cheap, then it's okay to process lots of nodes.

Contraction Hierarchies (long paper) can find paths faster by adding shortcut edges. Also great reading is the related work section, which summarizes A* improvements from landmarks, arc flags, transit nodes, highway hierarchies, and other approaches.

Arc flags (note to self: need a better link here) restrict the set of edges expanded in the main pathfinding loop. First divide the world into regions, then precalculate which edges are part of a shortest path to each region. Instead of looking at *all* edges to neighbors, look at only the edges that are part of a shortest path to the region the goal is in. Steve Rabin calls this approach "goal bounding".

Biased Cost Pathfinding alters the movement costs of areas where other units are going to move, so that subsequent paths avoid colliding with those units.

Parallel Ripple Search is designed for multi-core pathfinding, without the sort bottleneck that A* has.

Corridor Maps are a way to construct a pathfinding graph that greatly reduces the number of nodes, especially in maps with lots of corridors.

Learning Real-Time A* updates the heuristic as it explores the map. Prioritized

Learning Real-Time A* prioritizes the search to favor areas where it is learning more.

IDA* runs faster with a hex grid than a square grid(!).

Compressed Path Databases tests how well all-pairs shortest path (Floyd-Warshall or Johnson's Algorithm) on grids can be compressed. It applies if you're using a grid and the map isn't changing; I suspect you'd be better off reducing the graph size first.

Probabilistic Roadmaps (PRMs) build pathfinding graphs from polygonal obstacle maps.

---

---

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

---

From Red Blob Games

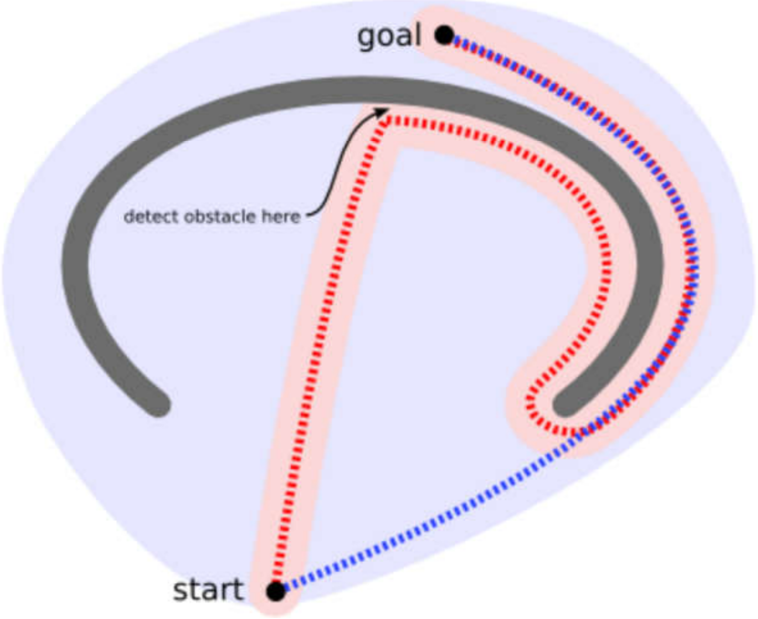
I started writing this in 1997; last modified: 02 Mar 2018

# Introduction to A*

From Amit’s Thoughts on Pathfinding

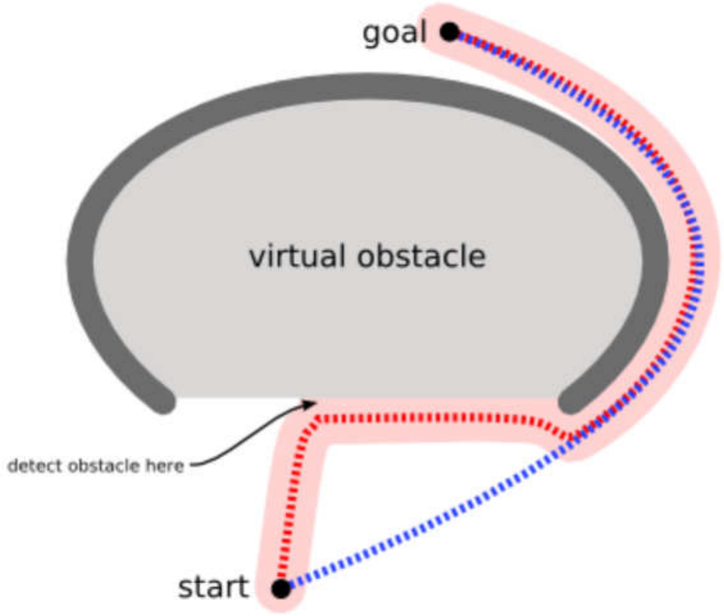
Home Blog Links Twitter About Search

Movement for a single object seems easy. Pathfinding is complex. Why bother with pathfinding? Consider the following situation:



The unit is initially at the bottom of the map and wants to get to the top. There is nothing in the area it scans (shown in pink) to indicate that the unit should not move up, so it continues on its way. Near the top, it detects an obstacle and changes direction. It then finds its way around the “U”-shaped obstacle, following the red path. In contrast, a pathfinder would have scanned a larger area (shown in light blue), but found a shorter path (blue), never sending the unit into the concave shaped obstacle.

You can however extend a movement algorithm to work around traps like the one shown above. Either avoid creating concave obstacles, or mark their convex hulls as dangerous (to be entered only if the goal is inside):

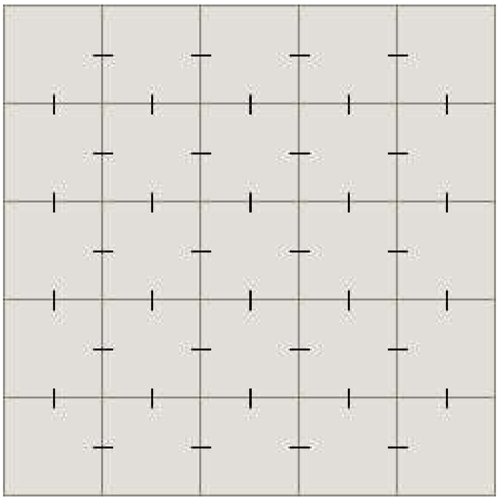


Pathfinders let you plan ahead rather than waiting until the last moment to discover there's a problem. There's a tradeoff between planning with pathfinders and reacting with movement algorithms. Planning generally is slower but gives better results; movement is generally faster but can get stuck. If the game world is changing often, planning ahead is less valuable. I recommend using both: pathfinding for big picture, slow changing obstacles, and long paths; and movement for local area, fast changing, and short paths.

## Algorithms #

*I have written a newer version of this one page, but not the rest of the pages. It has interactive diagrams and sample code.*

The pathfinding algorithms from computer science textbooks work on *graphs* in the mathematical sense—a set of vertices with edges connecting them. A tiled game map can be considered a graph with each tile being a vertex and edges drawn between tiles that are adjacent to each other:

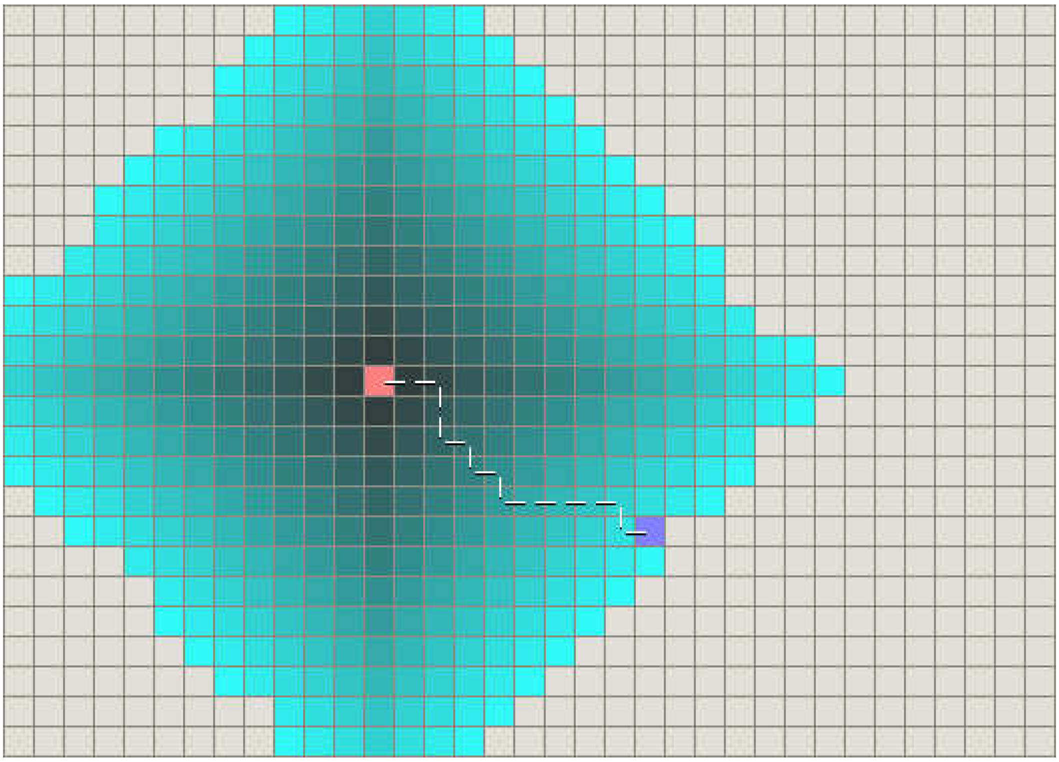


For now, I will assume that we're using two-dimensional grids. If you haven't worked with graphs before, see this primer. Later on, I'll discuss how to build other kinds of graphs out of your game world.

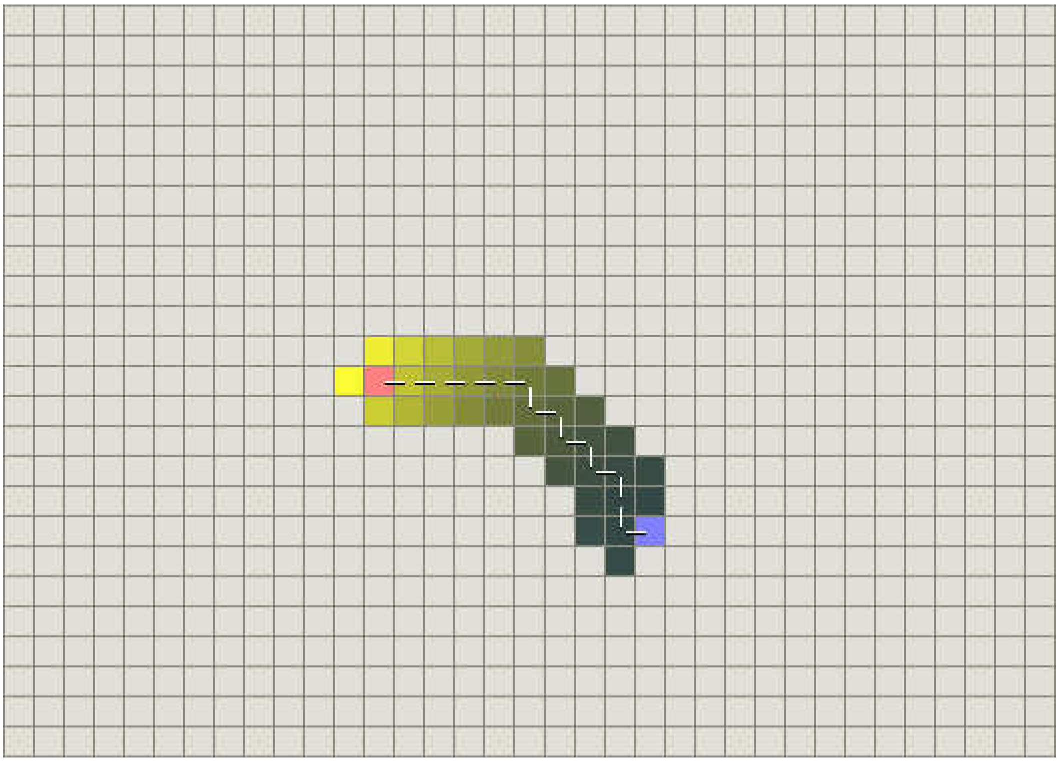
Most pathfinding algorithms from AI or Algorithms research are designed for arbitrary graphs rather than grid-based games. We'd like to find something that can take advantage of the nature of a game map. There are some things we consider common sense, but that algorithms don't understand. We know something about distances: in general, as two things get farther apart, it will take longer to move from one to the other, assuming there are no wormholes. We know something about directions: if your destination is to the east, the best path is more likely to be found by walking to the east than by walking to the west. On grids, we know something about symmetry: most of the time, moving north then east is the same as moving east then north. This additional information can help us make pathfinding algorithms run faster.

## Dijkstra's Algorithm and Best-First-Search #

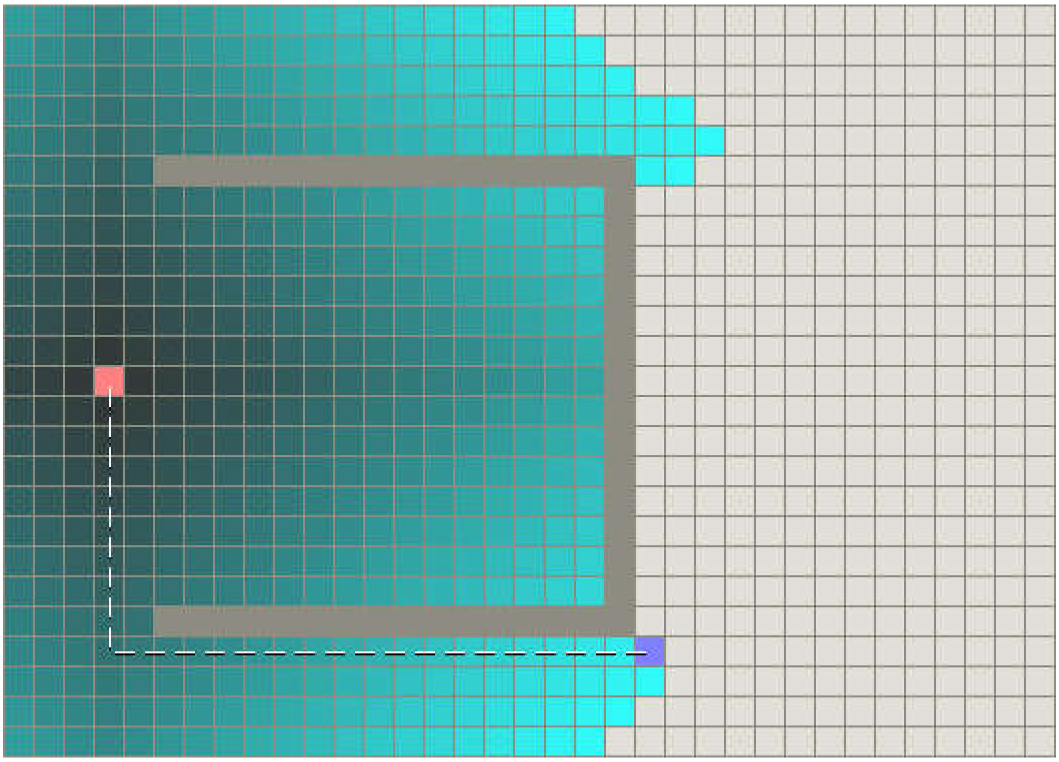
Dijkstra's Algorithm works by visiting vertices in the graph starting with the object's starting point. It then repeatedly examines the closest not-yet-examined vertex, adding its vertices to the set of vertices to be examined. It expands outwards from the starting point until it reaches the goal. Dijkstra's Algorithm is guaranteed to find a shortest path from the starting point to the goal, as long as none of the edges have a negative cost. (I write "a shortest path" because there are often multiple equivalently-short paths.) In the following diagram, the pink square is the starting point, the blue square is the goal, and the teal areas show what areas Dijkstra’s Algorithm scanned. The lightest teal areas are those farthest from the starting point, and thus form the “frontier” of exploration:



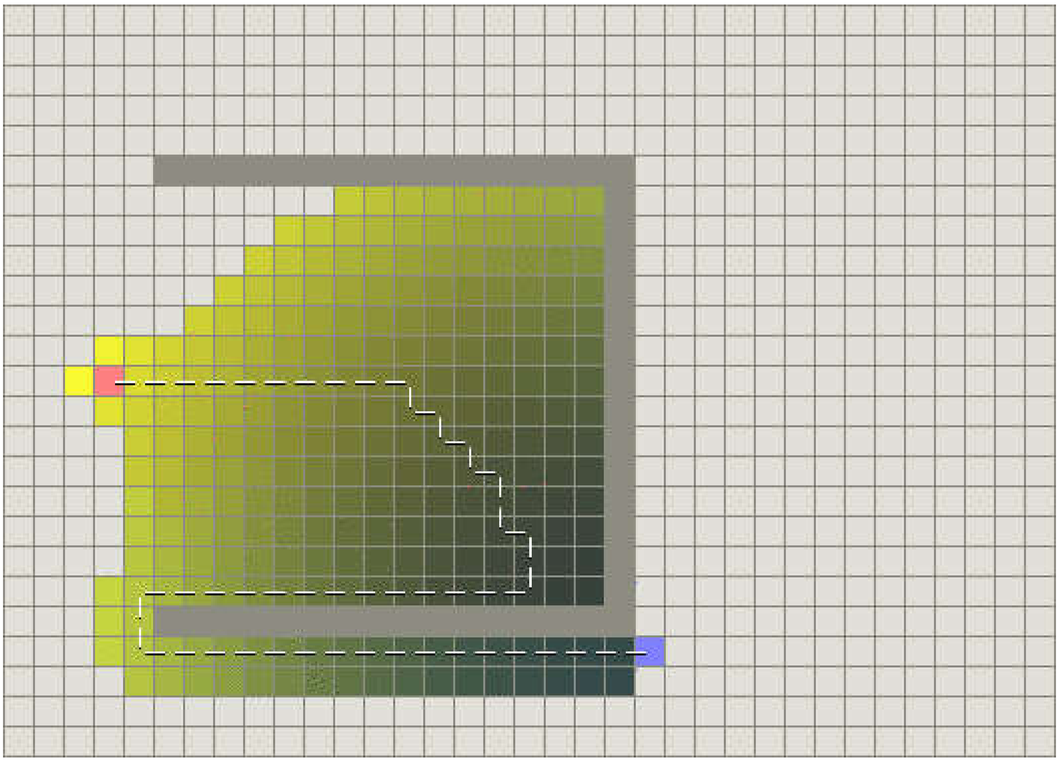
The Greedy Best-First-Search algorithm works in a similar way, except that it has some estimate (called a *heuristic*) of how far from the goal any vertex is. Instead of selecting the vertex closest to the starting point, it selects the vertex closest to the goal. Greedy Best-First-Search is *not* guaranteed to find a shortest path. However, it runs much quicker than Dijkstra’s Algorithm because it uses the heuristic function to guide its way towards the goal very quickly. For example, if the goal is to the south of the starting position, Greedy Best-First-Search will tend to focus on paths that lead southwards. In the following diagram, yellow represents those nodes with a high heuristic value (high cost to get to the goal) and black represents nodes with a low heuristic value (low cost to get to the goal). It shows that Greedy Best-First-Search can find paths very quickly compared to Dijkstra’s Algorithm:



However, both of these examples illustrate the simplest case—when the map has no obstacles, and the shortest path really is a straight line. Let's consider the concave obstacle as described in the previous section. Dijkstra's Algorithm works harder but is guaranteed to find a shortest path:



Greedy Best-First-Search on the other hand does less work but its path is clearly not as good:



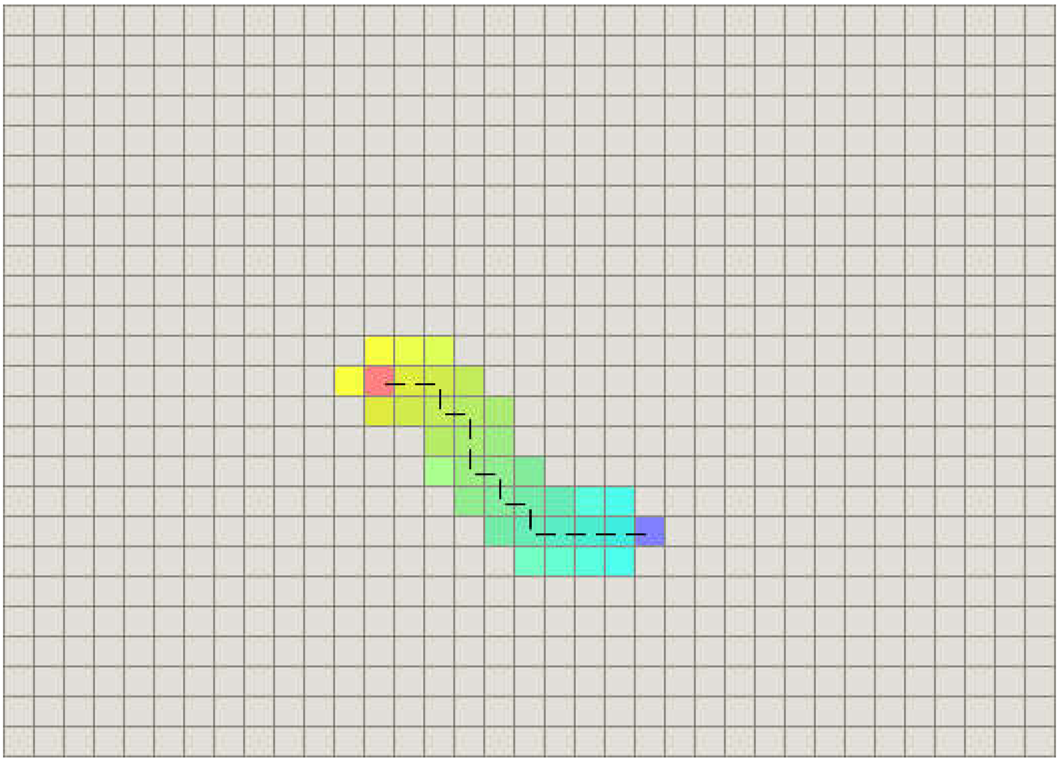
The trouble is that Greedy Best-First-Search is "greedy" and tries to move towards the goal even if it's not the right path. Since it only considers the cost to get to the goal and ignores the cost of the path so far, it keeps going even if the path it's on has become really long.

Wouldn't it be nice to combine the best of both? A* was developed in 1968 to combine heuristic approaches like Greedy Best-First-Search and formal approaches like Dijsktra's Algorithm. It's a little unusual in that heuristic approaches usually give you an approximate way to solve problems without guaranteeing that you get the best answer. However, A* is built on top of the heuristic, and although the heuristic itself does not give you a guarantee, A* *can* guarantee a shortest path.

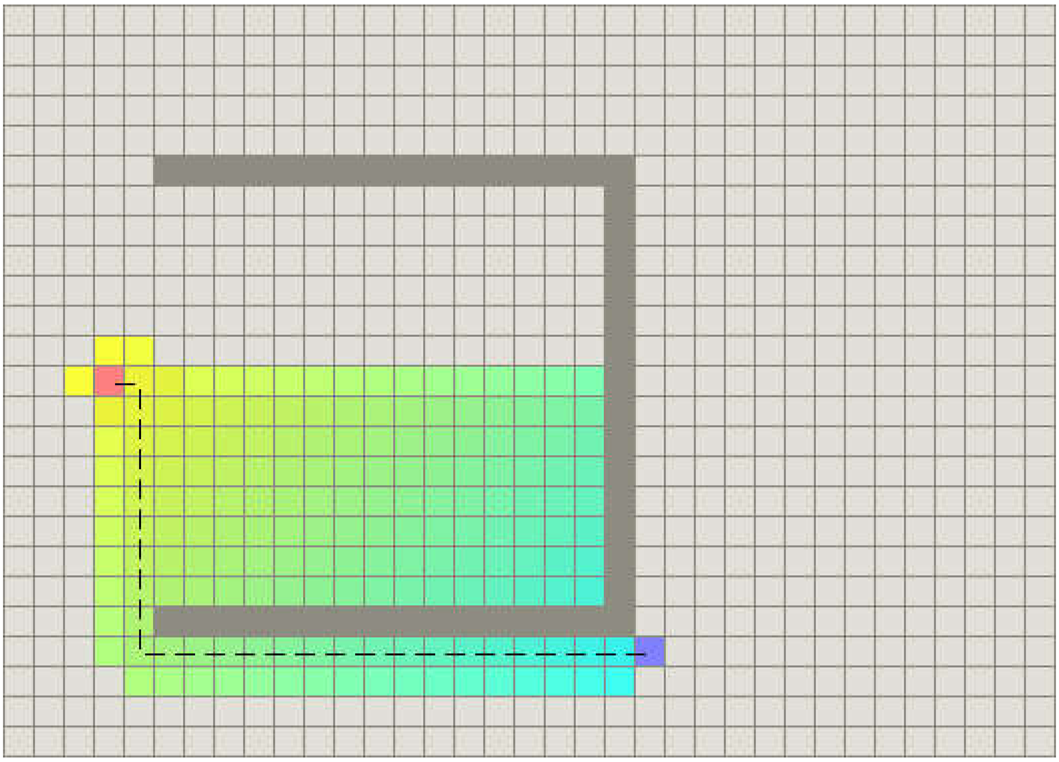
## The A* Algorithm #

I will be focusing on the **A* Algorithm**. A* is the most popular choice for pathfinding, because it's fairly flexible and can be used in a wide range of contexts.

A* is like Dijkstra's Algorithm in that it can be used to find a shortest path. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search:



In the example with a concave obstacle, A* finds a path as good as what Dijkstra's Algorithm found:



The secret to its success is that it combines the pieces of information that Dijkstra's Algorithm uses (favoring vertices that are close to the starting point) *and* information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, `g(n)` represents the *exact cost* of the path from the starting point to any vertex `n`, and `h(n)` represents the heuristic *estimated cost* from vertex `n` to the goal. In the above diagrams, the yellow (`h`) represents vertices far from the goal and teal (`g`) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex `n` that has the lowest `f(n) = g(n) + h(n)`.

The rest of this article will explore heuristic design, implementation, map representation, and a variety of other topics related to the use of pathfinding in games. Some sections are well-developed and others are rather incomplete.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# Heuristics

From Amit’s Thoughts on Pathfinding

Home Blog Links Twitter About Search

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 Greedy Best-First-Search.

**Note:**

Technically, the **A*** algorithm should be called simply **A** if the heuristic is an underestimate of the actual cost. However, I will continue to call it **A*** because the implementation is the same and the game programming community does not distinguish **A** from **A***.

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.

## 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 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—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 ($L_\infty$).
- 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.

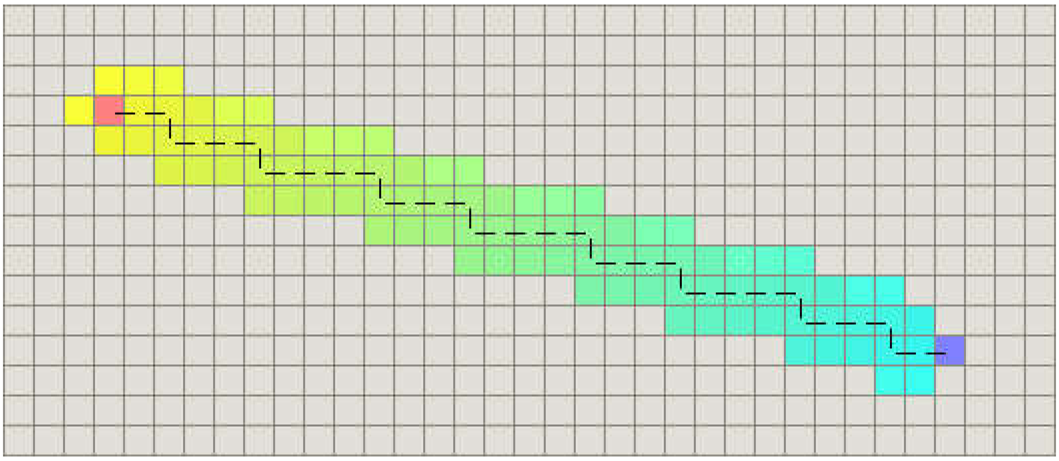
Multiply the distance in steps by the minimum cost for a step. For example, if you're measuring in meters, the distance is 3 squares, and each square is 15 meters, then the heuristic would return 3 × 15 = 45 meters. If you're measuring in time, the distance is 3 squares, and each square takes at least 4 minutes to cross, then the heuristic would return 3 × 4 = 12 minutes. The units (meters, minutes, etc.) returned by the heuristic should match the units used by the cost function.

## 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. *In the simple case, you can set `D` to be 1.* The heuristic on a square grid where you can move in 4 directions should be `D` times the Manhattan distance:

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy)
```

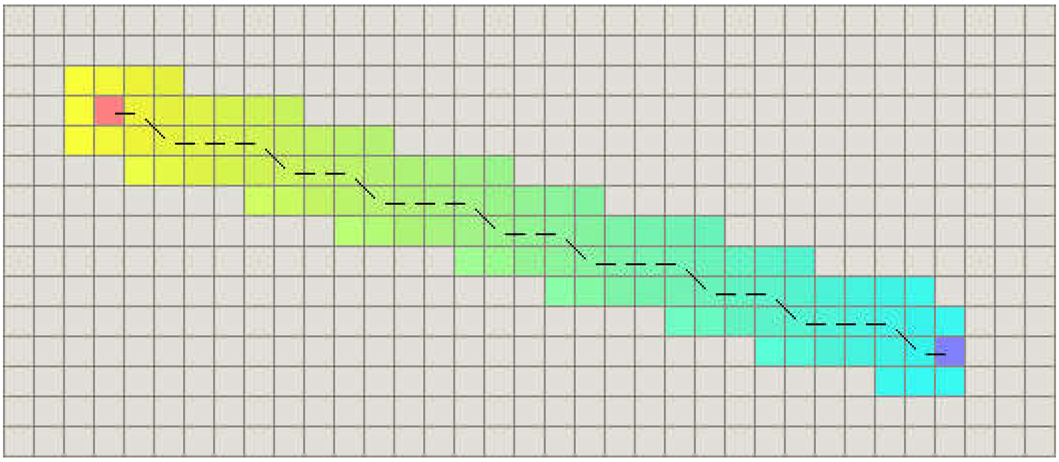
How do you pick D? Use a scale that matches your cost function. For the best paths, and an "admissible" heuristic, set D to the lowest cost between adjacent squares. In the absence of obstacles, and on terrain that has the minimum movement cost D, moving one step closer to the goal should *increase* `g` by D and *decrease* `h` by D. When you add the two, `f` (which is set to `g + h`) will stay the same; that's a sign that the heuristic and cost function scales match. You can also give up optimal paths to make A* run faster by increasing D, or by decreasing the ratio between the lowest and highest edge costs.



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

## Diagonal distance #

If your map allows diagonal movement you need a different heuristic. 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×D2, where D2 is the cost of moving diagonally.



```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

Here we compute the number of steps you take if you can't take a diagonal, then subtract the steps you save by using the diagonal. There are `min(dx, dy)` diagonal steps, and each one costs `D2` but saves you `2×D` non-diagonal steps.

When D = 1 and D2 = 1, this is called the Chebyshev distance. When D = 1 and D2 = sqrt(2), this is called the *octile distance*.

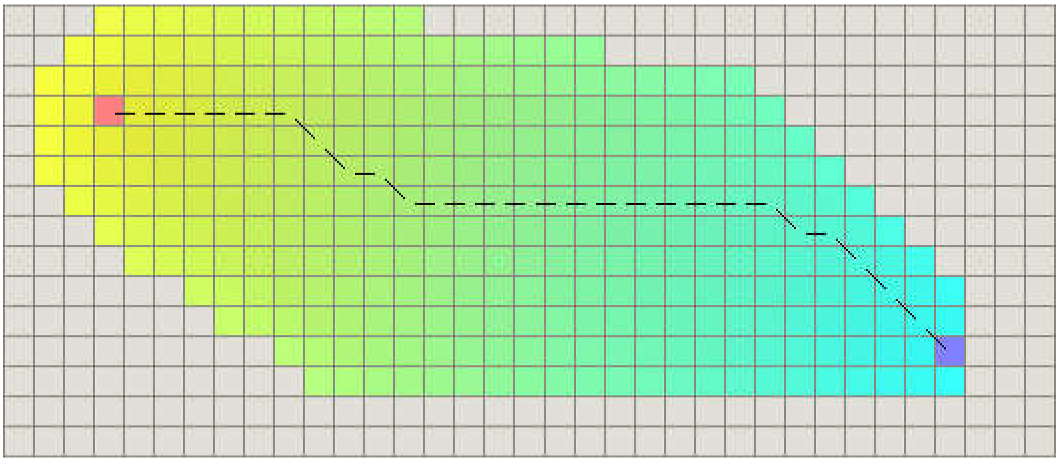
Another way to write this is `D * max(dx, dy) + (D2-1) * min(dx, dy)`. Patrick Lester writes it yet a different way, with `if (dx > dy) (D * (dx-dy) + D2 * dy) else (D * (dy-dx) + D2 * dx)`. These are all equivalent.

## Euclidean distance #

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

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * sqrt(dx * dx + dy * dy)
```

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:

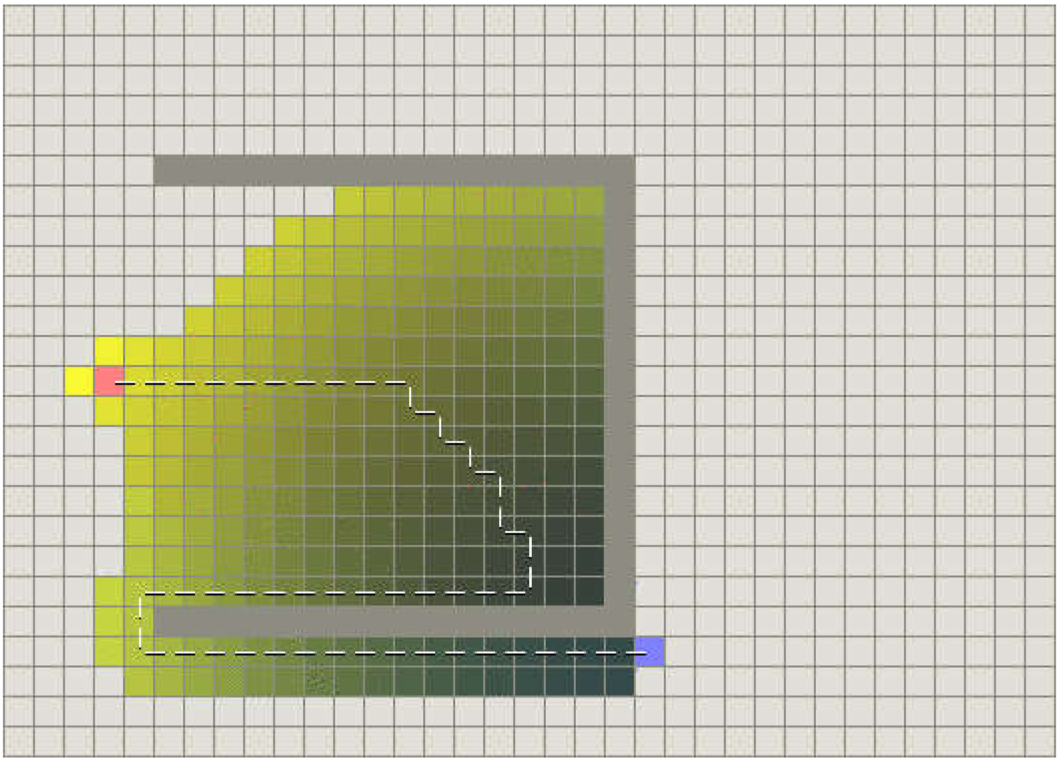


## Euclidean distance, squared #

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

```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx * dx + dy * dy)
```

**Do not do this!** This definitely runs into the scale problem. The scale of `g` and `h` need to match, because you're adding them together to form `f`. 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.

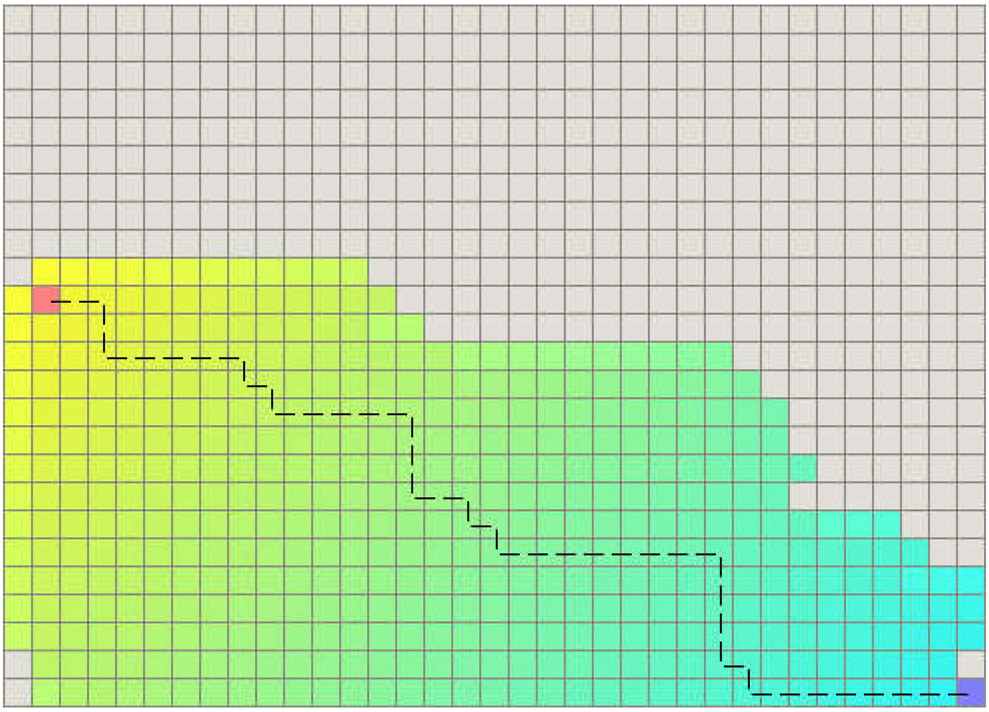
## Multiple goals #

If you want to search for *any* of several goals, 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.

If you want to search for spot near a single goal, ask A* search to find a path to the center of the goal area. While processing nodes from the OPEN set, exit when you pull a node that is near enough.

## Breaking ties #

In some grid 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 only one.



Ties in `f` values.

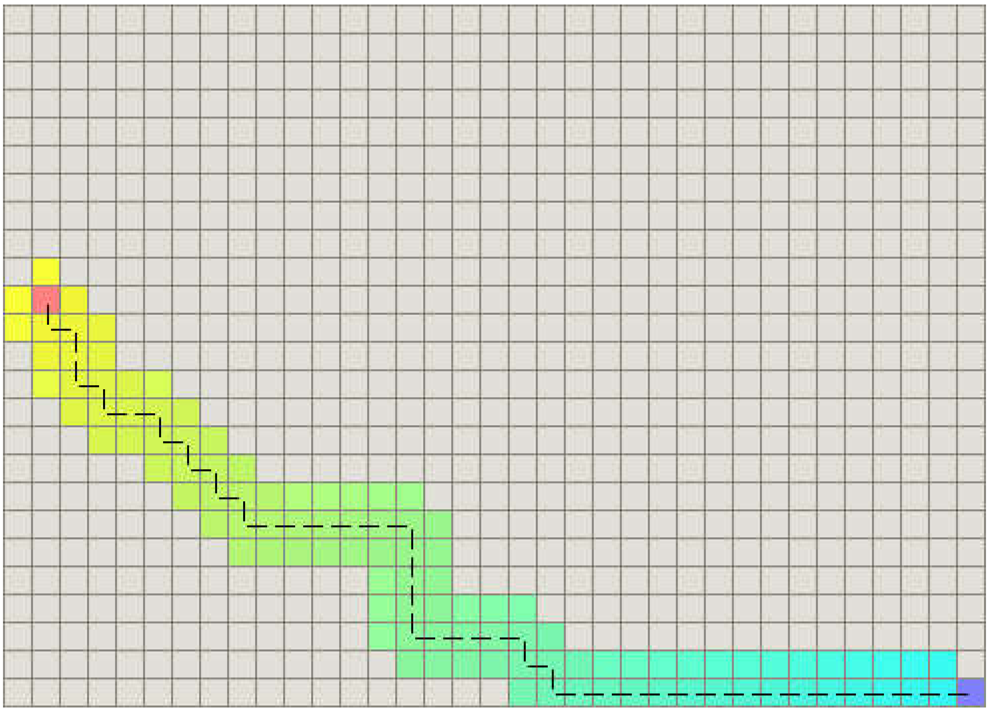
The quick hack to work around this problem is to either adjust the `g` or `h` values. The tie breaker needs to be deterministic with respect to the vertex (*i.e.,* it shouldn't be a random number), and it needs to make the `f` values differ. Since A* sorts by `f` value, making them different means only 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.

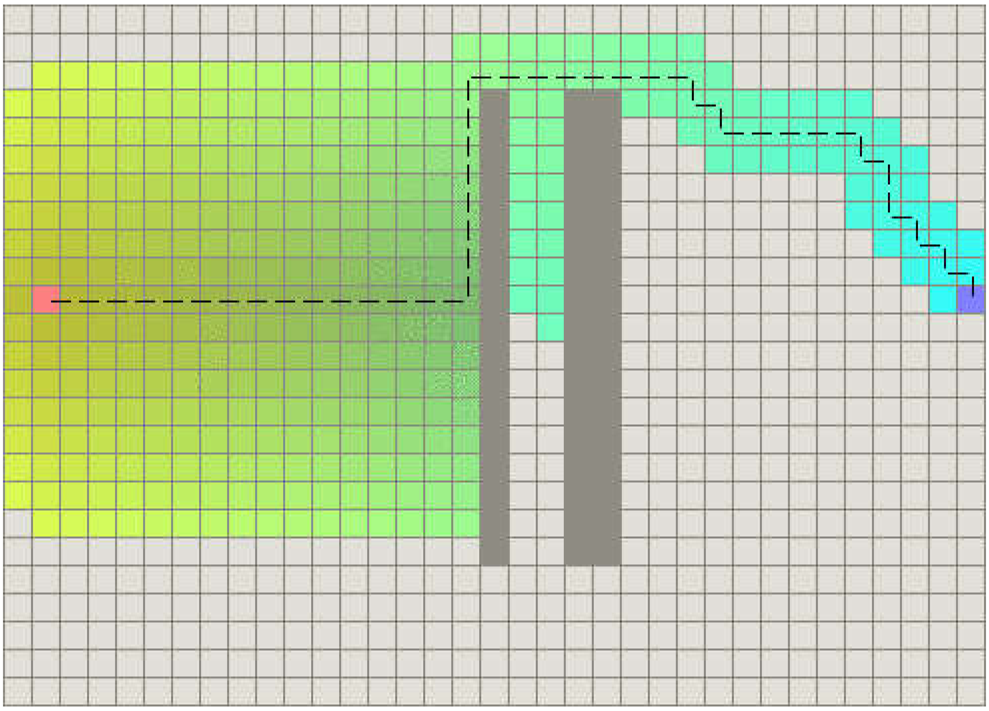
```
heuristic *= (1.0 + p)
```

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. (Note that this slightly breaks "admissibility" of the heuristic but in games it almost never matters.) 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.

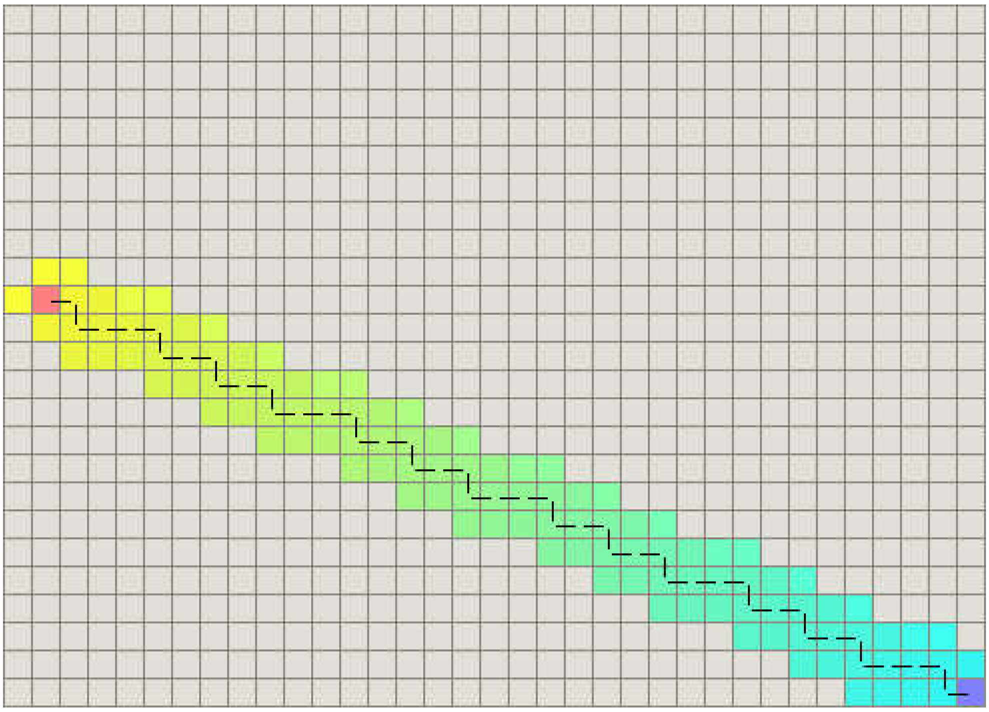
Steven van Dijk suggests that a more straightforward way to do this would to pass `h` to the comparison function. When the `f` values are equal, the comparison function would break the tie by looking at `h`.

Another way to break ties is to add a deterministic random number to the heuristic or edge costs. (One way to choose a deterministic random number is to compute a hash of the coordinates.) 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:

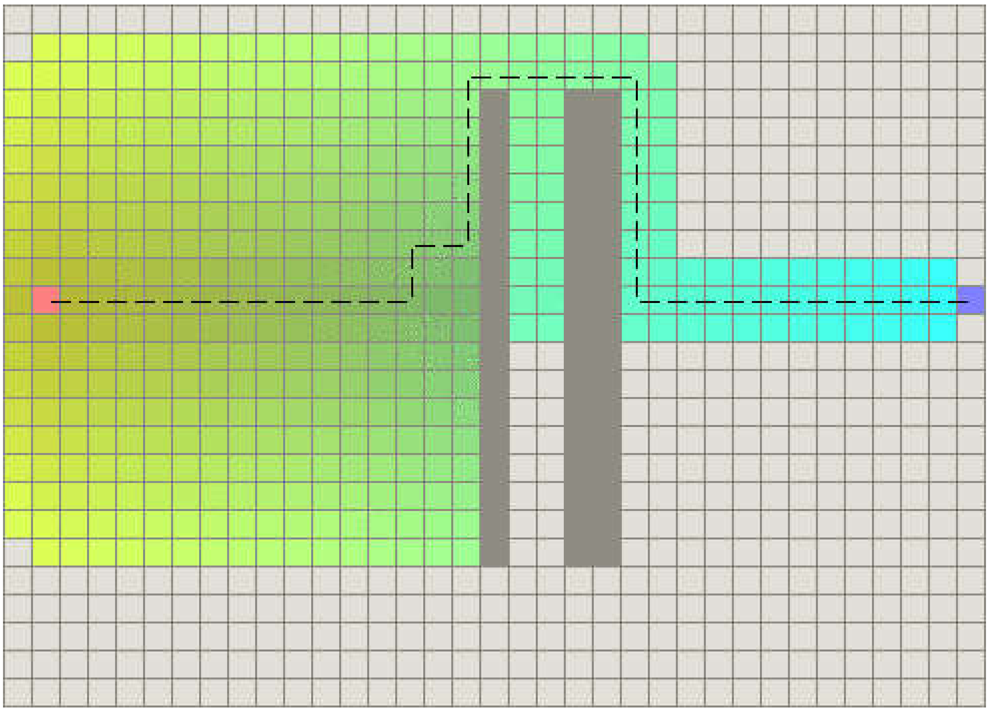
```
dx1 = current.x - goal.x
dy1 = current.y - goal.y
dx2 = start.x - goal.x
dy2 = start.y - goal.y
cross = abs(dx1*dy2 - dx2*dy1)
heuristic += cross*0.001
```

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 will look 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 [or try this mirror or 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.

And yet another way to break ties on grids is to minimize turns. The change in x,y from the *parent* to the *current* node tells you what direction you were moving in. For all edges being considered from *current* to *neighbor*, if the change in x,y is different than the one from parent to current, then add a small penalty to the movement cost.

**The above modifications to the heuristic are a "band aid" fix to an underlying inefficiency.** Ties occur when there are lots of paths that are equally good, leading to a large number of nodes to explore. Consider ways to "work smarter, not harder":

- Alternate map representations can solve the problem by **reducing the number of nodes in the graph**. Collapsing multiple nodes into one, or by remove all but the important nodes. Rectangular Symmetry Reduction is a way to do this on square grids; also look at "framed quad trees". Hierarchical pathfinding uses a high level graph with few nodes to find most of the path, then a low level graph with more nodes to refine the path.
- Some approaches leave the number of nodes alone but **reduce the number of nodes visited**. Jump Point Search skips over large areas of nodes that would contain lots of ties; it's designed for square grids. Skip links add "shortcut" edges that skip over areas of the map. The AlphA* algorithm adds some depth-first searching to the usual breadth-first behavior of A*, so that it can explore a single path instead of processing all of them simultaneously.
- Fringe Search (PDF) solves the problem instead by **making node processing fast**. Instead of keeping the OPEN set sorted and visiting nodes one at a time, it processes nodes in batches, expanding only the nodes that have low f--values. This is related to the HOT queues approach.

## Approximate heuristics #

A heuristic that has the exact distance is ideal for making A* fast but it’s usually impractical. We can often preprocess the graph to construct an approximate distance, and use that approximation in the A* heuristic.

ALT A* uses “landmarks” and the triangle inequality to preprocess the pathfinding graph in order to make pathfinding much faster. ALT also does a few other things, but the heuristic improvement is the part that got my attention. It’s surprisingly simple to implement, sometimes under 15 lines of code, and produces impressive speedups.

The name “landmark” is a little misleading. These points need to be placed on the outer edges of the map. Some authors call it “differential heuristics”.

The landmark approach stores lots of data that could be compressed. The Compressed Differential Heuristic shows the results of compressing the landmark data. You can store a lot more landmarks in the same space, so you get improved heuristic values.

Landmarks may be a special case of a more general approach. This paper explores transforming a map into a map where a regular distance metric works.

Distance oracles seem to be related but I haven’t looked into them yet.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# Implementation notes

From Amit's Thoughts on Pathfinding

Home Blog Links Twitter About Search

## Sketch #

The A* algorithm, stripped of all the code, is fairly simple. There are two sets, OPEN and CLOSED. The OPEN set contains those nodes that are candidates for examining. Initially, the OPEN set contains only one element: the starting position. The CLOSED set contains those nodes that have already been examined. Initially, the CLOSED set is empty. Graphically, the OPEN set is the "frontier" and the CLOSED set is the "interior" of the visited areas. Each node also keeps a pointer to its parent node so that we can determine how it was found.

There is a main loop that repeatedly pulls out the best node `n` in OPEN (the node with the lowest `f` value) and examines it. If `n` is the goal, then we're done. Otherwise, node `n` is removed from OPEN and added to CLOSED. Then, its neighbors `n'` are examined. A neighbor that is in CLOSED has already been seen, so we don't need to look at it [1]. A neighbor that is in OPEN is scheduled to be looked at, so we don't need to look at it now. Otherwise, we add it to OPEN, with its parent set to `n`. The path cost to `n'`, `g(n')`, will be set to `g(n) + movementcost(n, n')`.

**I go into a lot more detail** here, with interactive diagrams.

[1] I'm skipping a small detail here. You do need to check to see if the node's `g` value can be lowered, and if so, you re-open it.

```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current, neighbor)
    if neighbor in OPEN and cost less than g(neighbor):
      remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor): [2]
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to g(neighbor) + h(neighbor)
      set neighbor's parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

[2] This should never happen if you have an consistent admissible heuristic. However in games we often have inadmissible heuristics.

**See Python and C++ implementations here.**

## Connectivity #

If your game has situations in which the start and goal are not connected at all by the graph, A* will take a long time to run, since it has to explore every node connected from the start before it realizes there's no path. Calculate the Connected Components first and only use A* if the start and goal are in the same region.

## Performance #

The main loop of A* reads from a priority queue, analyzes it, and inserts nodes back into the priority queue. In addition, it tracks which nodes have been visited. To improve performance, consider:

- Can you decrease the size of the graph? This will reduce the number of nodes that are processed, both those on the path and those that don't end up on the final path. Consider navigation meshes instead of grids. Consider hierarchical map representations.
- Can you improve the accuracy of the heuristic? This will reduce the number of nodes that are not on the final path. The closer the hheuristic to the actual path length (not the distance), the fewer nodes A* will explore. Consider these heuristics for grids. Consider ALT (A*, Landmarks, Triangle Inequality) for graphs in general (including grids).
- Can you make the priority queue faster? Consider other data structures for

your priority queue. Consider processing nodes in batches, as fringe search does. Consider approximate sorting.
- Can you make the heuristic faster? The heuristic function is called for every open node. Consider caching its result. Consider inlining the call to it.

For grid maps, see these suggestions.

## Source code and demos #

### Demos #

These demos run in your browser:

- **I have written an introduction to A* with interactive demos.**
- I have written Flash demos for square grids, hexagonal grids, and triangular grids. The Actionscript 3 code for these demos is available here (see Pathfinder.as for the main algorithm, and Graph.as for the abstract interface to graphs).
- This site has demos of A*, Breadth-First Search, Dijkstra’s Algorithm, and Greedy Best-First Search on road maps (not grids)
- This library has lots of optimizations for grid maps.
- This Javascript A* demo lets you change the road weight; source code on github, using the MIT open source license. The calculation is interruptible so you can run a few iterations per frame.
- James Macgill’s Java applet.
- This interactive demo lets you choose A* or Dijkstra’s Algorithm.
- This demo is nice and has Javascript source code.
- This demo is nice and also has code available.
- Another Javascript demo
- This page describes Jump Point Search and also has an online demo.
- This Actionscript A* tutorial has a demo near the end.
- This demo is in Javascript with readable source but I don’t know the license for the source code.
- This A* demo and this Jump Point Search demo use Unity.

### Code #

If you're using C++, be sure to look at Recast, from Mikko Mononen.

If you're planning to implement graph search yourself, **here's my guide to Python and C++ implementations**.

I've collected some links to source code but haven't looked into these projects and can't make specific recommendations. **These are old** links. I started collecting them in 1997, before Google existed, before Github or npm or pypy or other ways of finding code online. **I no longer maintain this list.**

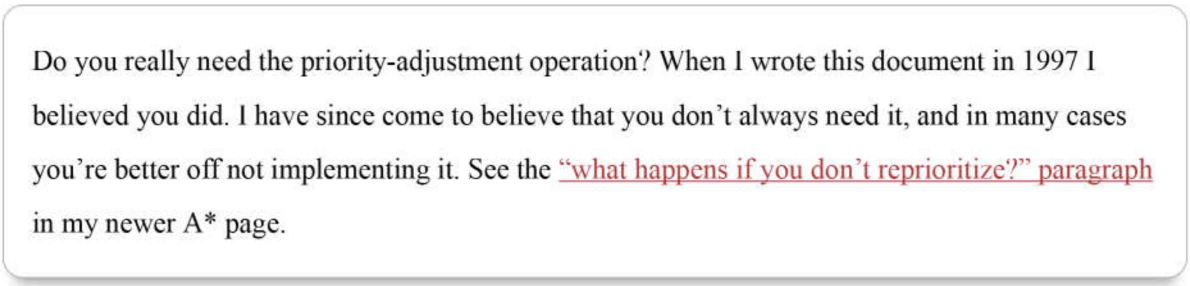
- C++: [1][2][3][4]
- Java: [1][2][3]
- Javascript: [1][2][3][4][5][6] (extremely fast pathfinding for grid maps)
- Python: [1]
- Objective C + Cocos2D: parts [1] and [2]
- Lua: [1][2]more
- Ruby: [1][2][3]
- C#: [1][priority queue helper class]
- Unity: [1]
- Assembly: [1]
- Actionscript 3 (Flash): [1][2][3][4][5]
- Flex (Flash): [1]
- Go: [1]
- Prolog: [1]
- Processing: [1]

## Set representation #

What's the first thing you'll think of using for the OPEN and CLOSED sets? If you're like me, you probably thought "array". You may have thought "linked list", too. There are many different data structures we can use, but to pick one we should look at what operations are needed.

There are three main operations we perform on the OPEN set: the main loop repeatedly finds the best node and removes it; the neighbor visiting will check whether a node is in the set; and the neighbor visiting will insert new nodes. Insertion and remove-best are operations typical of a priority queue.

The choice of data structure depends not only on the operations but on the number of times each operations runs. The membership test runs once for each neighbor for each node visited. Insertion runs once for each node being considered. Remove-best runs once for each node visited. Most nodes that are considered will be visited; the ones that are not are the *fringe* of the search space. When evaluating the cost of operations on these data structures, we need to consider the maximum size of the fringe (F).



Do you really need the priority-adjustment operation? When I wrote this document in 1997 I believed you did. I have since come to believe that you don't always need it, and in many cases you're better off not implementing it. See the "what happens if you don't reprioritize?" paragraph in my newer A* page.

In addition, *there's a fourth operation*, which is relatively rare but still needs to be implemented. If the node being examined is already in the OPEN set (which happens frequently), and if its `f` value is better than the one already in the OPEN set (which is rare), then the value in the OPEN set must be adjustment. The adjustment operation involves removing the node (which is not the best `f`) and re-inserting it. These two steps may be optimized into an increase-priority operation that moves the node (this is also called decrease-key).

**My recommendation:** The best generic choice is a binary heap. If you have a binary heap library available, use it. If not, start with sorted arrays or unsorted arrays, and switch to binary heaps if you want more performance. If you have more than 10,000 elements in your OPEN set, then consider more complicated structures such as a bucketing system.

## Unsorted arrays or linked lists #

The simplest data structure is an unsorted array or list. Membership test is slow, O(F) to scan the entire structure. Insertion is fast, O(1) to append to the end. Finding the best element is slow, O(F) to scan the entire structure. Removing the best element is O(F) for arrays and O(1) for linked lists. The increase-priority operation is O(F) to find the node and O(1) to change its value.

## Sorted arrays #

To make remove-best fast, we can keep the array sorted. Membership is then O(log F), since we can use binary search. Insertion is slow, O(F) to move all the elements to make space for the new one. Finding the best is fast, O(1) since it's already at the end. And removing the best is O(1) if we make sure the best sorts to the *end* of the array. The increase-priority operation is O(log F) to find the node and O(F) to change its value/position.

Make sure the array is sorted so that the best element is at the end.

## Sorted linked lists #

With sorted arrays, insertion is slow. If we use a linked list, we can make that fast. Membership in the linked list is slow, O(F) to scan the list. Insertion is fast, O(1) to insert a new link, but it was O(F) to find the right position for it. Finding the best remains fast, O(1) because the best is at the end. Removing the best also is O(1). The increase-priority operation is O(F) to find the node and O(1) to change its value/position.

## Binary heaps #

A binary heap (not to be confused with a memory heap) is a tree structure that is stored in an array. Unlike most trees, which use pointers to refer to children, the binary heap uses indexing to find children.

In a binary heap, membership is O(F), as you have to scan the entire structure. Insertion is O(log F) and remove-best is O(log F).

The increase-priority operation is tricky, with O(F) to find the node and surprisingly, only O(log F) to increase-priority it. Unfortunately most priority queue libraries don't include this operation. Fortunately, *it's not strictly necessary*. So I recommend not worrying about it unless you absolutely need to. Instead of increasing priority, insert a new element into the priority queue. You'll potentially end up processing the node twice but that's relatively cheap compared to implementing increase-priority.

In C++, use the priority_queue class, which doesn't have increase-priority, or Boost's mutable priority queue, which does. In Python, use the heapq library.

You can combine a hash table or indexed array for membership and a priority queue for managing priorities; see the hybrid section below.

As part of my newer A* tutorial, I have a complete A* implementation in Python and C++ using binary heaps for the priorities and hash tables for the for membership. I *do not implement increase-priority* and explain why in the optimization section.

A variant of the binary heap is a d-ary heap, which has more than 2 children per node. Inserts and increase-priority become a little bit faster, but removals become a little bit slower. They may have better cache performance.

**I have only used binary heaps and bucket approaches for my own pathfinding projects.** If binary heaps aren't good enough, then consider pairing heaps, sequence heaps, or a bucket based approach. The paper Priority Queues and Dijkstra's Algorithm is worth a read if you are unable to shrink the graph and need a faster priority queue.

## Sorted skip lists #

Searching an unsorted linked list is slow. We can make that faster if we use a skip list instead of a linked list. With a skip list, membership is fast if you have the sort key: O(log F). Insertion is O(1) like a linked list if you know where to insert. Finding the best node is fast if the sort key is `f`, O(1), and removing a node is O(1). The increase-priority operation involves finding a node, removing it, and reinserting it.

If we use skip lists with the map location as the key, membership is O(log F), insertion is O(1) after we've performed the membership test, finding the best node is O(F), and removing a node is O(1). This is better than unsorted linked lists in that membership is faster.

If we use skip lists with the `f` value as the key, membership is O(F), insertion is O(1), finding the best node is O(1), and removing a node is O(1). This is no better than sorted linked lists.

## Indexed arrays #

If the set of nodes is finite and reasonably sized, we can use a direct indexing structure, where an index function `i(n)` maps each node `n` to an index into an array. Unlike the unsorted and sorted arrays, which have a size corresponding to the largest size of OPEN, with an indexed array the array size is always `max(i(n))` over all `n`. If your function is dense (*i.e.,* there are no indices unused), then `max(i(n))` will be the number of nodes in your graph. Whenever your map is a grid, it's easy to make the function dense.

Assuming `i(n)` is O(1), membership test is O(1), as we merely have to check whether `Array[i(n)]` contains any data. Insertion is O(1), as we set `Array[i(n)]`. Find and remove best is O(numnodes), since we have to search the entire structure. The increase-priority operation is O(1).

## Hash tables #

Indexed arrays take up a lot of memory to store all the nodes that are *not* in the OPEN set. An alternative is to use a hash table, with a hash function `h(n)` that maps each node `n` into a hash code. Keep the hash table twice as big as N to keep the chance of collisions low. Assuming `h(n)` is O(1), membership test is expected O(1), insertion is expected O(1), and remove best is O(numnodes), since we have to search the entire structure. The increase-priority operation is O(1).

Hash tables are best for set membership but not for managing priorities. In my my newer A* tutorial, I use hash tables for membership and binary heaps for priorities. I combined the OPEN and CLOSED sets into one, which I call VISITED.

## Splay trees #

Heaps are a tree-based structure with expected O(log F) time operations. However, the problem is that with A*, the common behavior is that you have a low cost node that is removed (causing O(log F) behavior, since values have to move up from the very bottom of the tree) followed by low cost nodes that are added (causing O(log F) behavior, since these values are added at the bottom and bubble up to the very top). The *expected case* behavior of heaps here is equivalent to the *worst case* behavior. We may be able to do better if we find a data structure where *expected case* is better, even if the *worst case* is no better.

Splay trees are a self adjusting tree structure. Any access to a node in the tree tends to bring that node up to the top. The result is a “caching” effect, where rarely used nodes go to the bottom and don’t slow down operations. It doesn’t matter how big your splay tree is, because your operations are only as slow as your “cache size”. In A*, the low cost nodes are used a lot, and the high cost nodes aren’t used for a long time, so those high cost nodes can move to the bottom of the tree.

With splay trees, membership, insertion, remove-best, and increase-priority are all expected O(log F), worst case O(F). Typically however, the caching keeps the worst case from occurring. Dijkstra’s Algorithm and A* with an underestimating heuristic however have some peculiar characteristics that may keep splay trees from being the best. In particular, `f(n') >= f(n)` for nodes `n` and neighboring node `n'`. When this happens, it may be that the insertions all occur on one side of the tree and end up putting it out of balance. I have not tested this.

## HOT queues #

There’s another data structure that may be better than heaps in theory, and its ideas may be useful in practice. Often, you can restrict the range of values that would be in the priority queue. Given a restricted range, there are often better algorithms. For example, sorting can be done on arbitrary values in O(N log N) time, but when there is a fixed range the bucket or radix sorts can perform sorting in O(N) time.

We can use HOT Queues (Heap On Top queues) to take advantage of `f(n') >= f(n)` where `n'` is a neighbor of `n`. We are removing the node `n` with minimal `f(n)`, and inserting neighbors `n'` with `f(n) <= f(n') <= f(n) + delta` where `delta <= C`. The constant `C` is the *maximum* change in cost from one point to an adjacent point. Since `f(n)` was the minimal `f` value in the OPEN set, and everything being inserted is `<= f(n) + delta`, we know that all `f` values in the OPEN set are within a range of `0 .. delta`. As in bucket/radix sort, we can keep “buckets” to sort the nodes in the OPEN set.

With HOT queues, the topmost bucket uses a binary heap and all other buckets are unsorted arrays. Membership test is O(F) because we don’t know which bucket the node is in. Insertion and remove-best in the top bucket are O(log

(F/K)). Insertion into other buckets is O(1), remove-best never occurs! If the top bucket empties, then we need to convert the next bucket, an unsorted array, into a binary heap. It turns out this operation (“heapify”) can be run in O(F/K) time. The increase-priority operation is best treated as a O(F/K) removal followed by an O(log (F/K)) or O(1) insertion.

In A*, many of the nodes we put into OPEN we never actually need. HOT Queues are a big win because the elements that are not needed are inserted in O(1) time. Only elements that are needed get heapified (which is not too expensive). The only operation that is more than O(1) is node deletion from the heap, which is only O(log (F/K)).

In addition, if `C` is small, we can set K = C, and then we do not even need a heap for the smallest bucket, since all the nodes in a bucket have the same `f` value. Insertion and remove-best are both O(1) time! One person reported that HOT queues are as fast as heaps for at most 800 nodes in the OPEN set, and are 20% faster when there are at most 1500 nodes. I would expect that HOT queues get faster as the number of nodes increases.

A cheap variant of a HOT queue is a two-level queue: put good nodes into one data structure (a heap or an array) and put bad nodes into another data structure (an array or a linked list). Since most nodes put into OPEN are “bad”, they are never examined, and there’s no harm putting them into the big array.

## Pairing heaps #

Fibonacci heaps are good priority queues for A*, in theory. However, in practice they’re not used. A pairing heap can be thought of as a simplified Fibonacci heap. They are said to work well in practice; I have never used them.

Here’s the original paper describing them.

## Soft heaps #

A soft heap is a type of heap that gives the nodes in approximately the right order. By approximating, it can provide results faster than a regular heap.

I haven't tried soft heaps. The algorithms described in this paper seem fairly short and straightforward to implement. However this answer on stackoverflow says it's useful in theory but "unlikely to be useful in practice".

For pathfinding in games, we often do not need the *exact* shortest path and usually would prefer to have a reasonably short path computed quickly. So this may be one of the applications where it *is* useful in practice. I don't know yet; if you have studied this data structure, please email me.

## Sequence heaps #

I haven't looked into them. See Peter Sanders's paper Fast Priority Queues for Cached Memory.

*"Sequence heaps may currently be the fastest available data structure for large comparison based priority queues both in cached and external memory This is particularly true if the queue elements are small and if we do not need deletion of arbitrary elements or decreasing keys."*

## Data Structure Comparison #

It is important to keep in mind that we are not merely looking for asymptotic ("big O") behavior. We also want to look for a low constant. To see why, consider an algorithm that is O(log F) and another that is O(F), where F is the number of elements in the heap. It may be that on your machine, an implementation of the first algorithm takes 10,000 * log(F) seconds, while an implementation of the second one takes 2 * F seconds. For F = 256, the first would take 80,000 seconds and the second would take 512 seconds. The "faster" algorithm takes more time in this case, and would only start to be faster when F > 200,000.

*You cannot merely compare two algorithms.* You should also compare the implementations of those algorithms. You also have to know what size your data might be. In the above example, the first implementation is faster for F > 200,000, but if in your game, F stays under 30,000, then the second implementation would have been better.

None of the basic data structures is entirely satisfactory. Unsorted arrays or lists make insertion very cheap and membership and removal very expensive. Sorted arrays or lists make membership somewhat cheap, removal very cheap and insertion very expensive. Binary heaps make insertion and removal somewhat cheap, but membership is very expensive. Splay trees make everything somewhat cheap. HOT queues make insertions cheap, removals fairly cheap, and membership tests somewhat cheap. Indexed arrays make membership and insertion very cheap, but removals are incredibly expensive, and they can also take up a lot of memory. Hash tables perform similarly to indexed arrays, but they can take up a lot less memory in the common case, and removals are merely expensive instead of extremely expensive.

For a good list of pointers to more advanced priority queue papers and implementations, see Lee Killough's Priority Queues page.

### Hybrid representations #

To get the best performance, you will want a hybrid data structure. For my A* code, I used an indexed array for O(1) membership test and a binary heap for O(log F) insertion and O(log F) remove-best. For increase-priority, I used the indexed array for an O(1) test whether I really needed to perform the change in priority (by storing the `g` value in the indexed array), and then in those rare cases that I did need to increase priority, I used the O(F) increase-priority on the binary heap. You can also use the indexed array to store the location in the heap of each node; this would give you O(log F) for increase-priority.

In my newer A* tutorial, I use a hash table for membership and a binary heap for priorities. I further simplified by combining OPEN and CLOSED into the same set, which I call Visited.

## Interaction with the game loop #

Interactive (especially real-time) games introduce requirements that affect your ability to compute the best path. It may be more important to get *any* answer than to get the *best* answer. Still, all other things being equal, a shorter path is better than a longer one.

In general, computing the part of the path close to the starting point is more important than the path close to the goal. The principle of *immediate start*: get the unit moving as soon as possible, even along a suboptimal path, and then compute a better path later. In real-time games, the *latency* of A* is often more important than its *throughput*.

Units can be programmed to follow either their instincts (simple movement) or their brains (a precalculated path). Units will follow their instincts unless their brains tell them otherwise. (This approach is used in nature and also in Rodney Brook's robot architecture.) Instead of calculating all the paths at once, limit the game to finding one path every one, two, or three game cycles. Then let the units start walking according to instinct (which could simply be moving in a straight line towards the goal), and come back later to find paths for them. This approach allows you to even out the cost of pathfinding so that it doesn't occur all at once.

## Early exit #

It is possible to exit early from the A* main loop and get a partial path. Normally, the loop exits when it finds the goal node. However, at any point before that, it can return a path to the currently best node in OPEN. That node is our best chance of getting to the goal, so it's a reasonable place to go.

Candidates for early exit include having examined some number of nodes, having spent some number of milliseconds in the A* algorithm, or exploring a node some distance away from the starting position. When using path splicing, the spliced path should be given a smaller limit than a full path.

## Interruptible algorithm #

If few objects need pathfinding services or if the data structures used to store the OPEN and CLOSED sets are small, it can be feasible to store the state of the algorithm, exit to the game loop, then continue where A* left off.

## Group movement #

Path requests do not arrive evenly distributed. A common situation in a real-time strategy game is for the player to select multiple units and order them to move to the same goal. This puts a high load on the pathfinding system.

**See Also:** Putting paths together is called path splicing in another section of these notes.

In this situation, it is very likely that the path found for one will be useful for other units. One idea is to find a path `P` from the center of the units to the center of the destinations. Then, use most of that path for all the units, but replace the first 10 steps and the last 10 steps by paths that are found for each individual unit. Unit *i* will receive a path from its starting location to `P[10]`, followed by the shared path `P[10..len(P)-10]`, followed by a path from `P[len(P)-10]` to the destination.

The paths found for each unit are short (approximately 10 steps on average), and the long path is shared. Most of the path is found only once and shared among all the units. However, the user may not be impressed if he sees all the units moving on the same path. To improve the appearance of the system, make the units follow slightly different paths. One way to do this is to alter the paths themselves, by choosing adjacent locations.

Another approach is to make the units aware of each other (perhaps by picking a “leader” unit randomly, or by picking the one with the best sense of what’s going on), and only find a path for the leader. Then use a flocking algorithm to make them move in a group.

There are some variants of A* that handle a moving destination, or updated knowledge of the destination. Some of them might be adapted to handle multiple units going to the same destination, by performing A* in reverse (finding a path from the destination to the unit).

## Refinement #

If the map contains few obstacles, but instead contains terrain of varying costs, then an initial path can be computed by treating terrain as being cheaper than normal. For instance, if grasslands are cost 1, hills are cost 2, and mountains are cost 3, then A* will consider walking through 3 grasslands to avoid 1 mountain. Instead, compute an initial path by treating grasslands as 1, hills as 1.1, and mountains as 1.2. A* will then spend less time trying to avoid the mountains, and it will find a path quicker. (It approximates the benefits of an exact heuristic.) Once a path is known, the unit can start moving, and the game loop can continue. When spare CPU is available, compute a better path using the real movement costs.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# Variants of A*

From Amit's Thoughts on Pathfinding

## Beam search #

In the main A* loop, the OPEN set stores all the nodes that may need to be searched to find a path. The **Beam Search** is a variation of A* that places a limit on the size of the OPEN set. If the set becomes too large, the node with the worst chances of giving a good path is dropped. One drawback is that you have to keep your set sorted to do this, which limits the kinds of data structures you'd choose.

## Iterative deepening #

Iterative Deepening is an approach used in many AI algorithms to start with an approximate answer, then make it more accurate. The name comes from game tree searches, where you look some number of moves ahead (for example, in Chess). You can try to deepen the tree by looking ahead more moves. Once your answer doesn't change or improve much, you assume that you have a pretty good answer, and it won't improve when you try to make it more accurate again. In IDA*, the "depth" is a cutoff for `f` values. When the `f` value is too large, the node won't even be considered (*i.e.,* it won't be added to the OPEN set). The first time through you process very few nodes. Each subsequent pass, you increase the number of nodes you visit. If you find that the path improves, then you continue to increase the cutoff; otherwise, you can stop. For more details, read these lecture nodes on IDA*.

I personally don't see much need for IDA* for finding paths on game maps. ID algorithms tend to increase computation time while reducing memory requirements. In map pathfinding, however, the "nodes" are very small—they are simply coordinates. I don't see a big win from not storing those nodes.

## Dynamic weighting #

With dynamic weighting, you assume that at the beginning of your search, it's more important to get (anywhere) quickly; at the end of the search, it's more important to get to the goal.

```
f(p) = g(p) + w(p) * h(p)
```

There is a weight (`w >= 1`) associated with the heuristic. As you get closer to the goal, you decrease the weight; this decreases the importance of the heuristic, and increases the relative importance of the actual cost of the path.

## Bandwidth search #

There are two properties about *Bandwidth Search* that some people may find useful. This variation assumes that `h` is an *overestimate*, but that it doesn't overestimate by more than some number `e`. If this is the case in your search, then the path you get will have a cost that doesn't exceed the best path's cost by more than `e`. Once again, the better you make your heuristic, the better your solution will be.

Another property you get is that if you can drop some nodes in the OPEN set. Whenever `h+d` is greater then the true cost of the path (for some `d`), you can drop any node that has an `f` value that's at least `e+d` higher than the `f` value of the best node in OPEN. This is a strange property. You have a "band" of good values for `f`; everything outside this band can be dropped, because there is a guarantee that it will not be on the best path.

Curiously, you can use different heuristics for the two properties, and things still work out. You can use one heuristic to guarantee that your path isn't too bad, and another one to determine what to drop in the OPEN set.

*Note:* When I wrote this in 1997, Bandwidth search looked potentially useful, but I've never used it and I don't see much written about it in the game industry, so I will probably remove this section. You can search Google for more information, especially from textbooks.

## Bidirectional search #

Instead of searching from the start to the finish, you can start two searches in parallel—one from start to finish, and one from finish to start. When they meet, you should have a good path.

It’s a good idea that will help in some situations. The idea behind bidirectional searches is that searching results in a “tree” that fans out over the map. A big tree is much worse than two small trees, so it’s better to have two small search trees.

The *front-to-front* variation links the two searches together. Instead of choosing the best forward-search node—`g(start,x) + h(x,goal)`—or the best backward-search node—`g(y,goal) + h(start,y)`—this algorithm chooses a pair of nodes with the best `g(start,x) + h(x,y) + g(y,goal)`.

The *retargeting* approach abandons simultaneous searches in the forward and backward directions. Instead, it performs a forward search for a short time, chooses the best forward candidate, and then performs a backward search—not to the starting point, but to that candidate. After a while, it chooses a best backward candidate and performs a forward search from the best forward candidate to the best backward candidate. This process continues until the two candidates are the same point.

Front-to-End Bidirectional Heuristic Search with Near-Optimal Node Expansions is a recent result with a near-optimal bidirectional variant of A*.

## Dynamic A* and Lifelong Planning A* #

There are variants of A* that allow for changes to the world after the initial path is computed. D* is intended for use when you don’t have complete information. If you don’t have all the information, A* can make mistakes; D*’s contribution is that it can correct those mistakes without taking much time. LPA* is intended for use when the costs are changing. With A*, the path may be invalidated by changes to the map; LPA* can re-use previous A* computations to produce a new path.

*However*, both D* and LPA* require a lot of space—essentially you run A* and keep around its internal information (`OPEN/CLOSED` sets, path tree, `g` values), and then when the map changes, D* or LPA* will tell you if you need to adjust your path to take into account the map changes.

For a game with lots of moving units, you usually don't want to keep all that information around, so D* and LPA* aren't applicable. They were designed for robotics, where there is only one robot—you don't need to reuse the memory for some other robot's path. If your game has only one or a small number of units, you may want to investigate D* or LPA*.

- Overview of D*
- D* Paper 1
- D* Paper 2
- Lifelong planning overview
- Lifelong planning paper (PDF)
- Lifelong planning A* applet

## Jump Point Search #

Many of the techniques for speeding up A* are really about reducing the number of nodes. In a square grid with uniform costs it's quite a waste to look at all the individual grid spaces one at a time. One approach is to build a graph of key points (such as corners) and use that for pathfinding. However, you don't want to precompute a waypoint graph, look at Jump Point Search, a variant of A* that can skip ahead on square grids. When considering children of the current node for possible inclusion in the OPEN set, Jump Point Search skips ahead to faraway nodes that are visible from the current node. Each step is more expensive but there are fewer of them, reducing the number of nodes in the OPEN set. See this blog post for details, this blog post for a nice visual explanation, and this discussion on reddit of pros and cons.

Also see Rectangular Symmetry Reduction, which analyzes the map and embeds jumps into the graph itself. Both techniques were developed for square grids. Here's the algorithm extended for hexagonal grids.

## Theta* #

Sometimes grids are used for pathfinding because the map is made on a grid, not because you actually want movement on a grid. A* would run faster and produce better paths if given a graph of key points (such as corners) instead of the grid. However if you don't want to precompute the graph of corners, you can use Theta*, a variant of A* that runs on square grids, to find paths that don't strictly follow the grid. When building *parent* pointers, Theta* will point directly to an ancestor if there's a line of sight to that node, and skips the nodes in between. Unlike path smoothing, which straightens out paths after they're found by A*, Theta* can analyze those paths as part of the A* process. This can lead to shorter paths than postprocessing a grid path into an any-angle path. This article is a reasonable introduction to the algorithm; also see Lazy Theta*.

The ideas from Theta* likely can be applied to navigation meshes as well.

Also see Block A*, which claims to be much faster than Theta* by using a hierarchical approach.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# Dealing with moving obstacles

From Amit’s Thoughts on Pathfinding

Home Blog Links Twitter About Search

A pathfinding algorithm will compute a path around stationary obstacles, but what if the obstacles move? By the time a unit reaches a particular point, an obstacle may no longer be there, or a new obstacle may be there. If the typical obstacle can be routed around, use a separate obstacle avoidance algorithm (steering) along with your pathfinder. The pathfinder will find the desired path, and then while following it, move around obstacles. If however obstacles can cause the path to change significantly, consider using the pathfinder for obstacle avoidance.

## Recalculating paths #

As time passes we expect the game world to change. A path found some time ago may no longer be the optimal path. It may be worth updating old paths with new information. Listed below are some criteria that could be used for determining when a recalculation is needed:

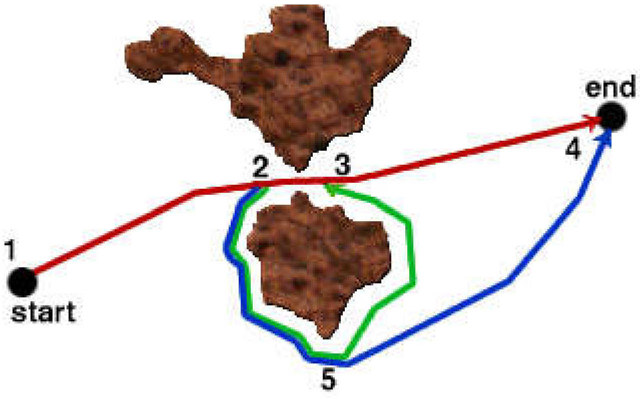
- Every $N$ steps: this guarantees that the information used to calculate the path is not more than $N$ steps old.
- Whenever extra CPU time is available: this allows dynamic adjustment of path quality; as more units are deployed, or if the game is running on a slower computer, CPU usage per unit can be decreased.
- Whenever the unit turns a corner or passes a waypoint.
- Whenever the world near the unit has changed.

The main drawback of path recalculation is that a lot of path information is thrown away. For example, if the path is 100 steps long and it is recalculated every 10 steps, the total number of path steps is 100+90+80+70+60+50+40+30+20+10 = 550. For a path of $M$ steps, approximately $M$^$^{2}$ path steps are computed over time. Therefore path recalculation is not a good idea if you expect to have many long paths. It would be better to reuse the path information instead of throwing it away.

## Path splicing #

When a path needs to be recalculated, it means the world is changing. Given a changing world, nearby parts of the map are better known than faraway parts of the map. We can follow a *local repair strategy*: find a good path nearby, and assume the path farther away need not be recomputed until we get closer to it. Instead of recalculating the entire path, we can recalculate the first *M* steps of the path:

1. Let `p[1]..p[N]` be the remainder of the path (*N* steps)
2. Compute a new path from `p[1]` to `p[M]`
3. *Splice* this new path into the old path by removing `p[1]..p[M]` and inserting the new path in its place



Since `p[1]` and `p[M]` are fewer than *M* steps apart, it's unlikely that the new path will be long. Unfortunately, situations can arise in which the new path is long and not very good. The accompanying figure shows such a situation. The original red path is 1-2-3-4; brown areas are obstacles. If we reach 2 and discover that the path from 2 to 3 has been blocked, path splicing would replace 2-3 with the green path 2-3-5 and splice it in, resulting in the unit moving along path 1-2-5-3-4. We can see this is not a good path; the blue path 1-2-5-4 would be better.

Bad paths can often be detected by looking at the length of the new path. If it is significantly longer than *M*, it could be bad. A simple solution is to add a limit (maximum path length) to the path finding algorithm. If a short path isn't found, the algorithm returns an error code; in this case, use path recalculation instead of path splicing to get a path such as 1-2-5-4.

> **Implementation Note:**
> Store the path in *reverse* order: it is easy to remove the beginning of the path and splice in a new path with a different length; because both operations occur at the *end* of the array. Essentially you treat the array as a *stack* where the top element is the next move to make.

For cases not involving these situations, for a path with *N* steps, path splicing will compute *2N* to *3N* path steps, depending on how often a new path is spliced in. This is a fairly low cost for the ability to respond to changes in the world. Surprisingly, the cost is independent of *M*, the number of steps for splicing. Instead of affecting CPU time, *M* controls a tradeoff between responsiveness and path quality. If *M* is high, the unit's movement will not respond quickly to changes in the map. If *M* is too low, the paths being spliced out may be too short to allow the replacement path to go around the obstacle cleanly; more suboptimal paths (such as 1-2-5-3-4) will be found. Try different values of *M* and different criteria for splicing (such as every 3/4 *M* steps) to see what's right for your map.

Path splicing is significantly faster than path recalculation, but it does not respond well to major changes in the path. It is possible to detect many of these situations and use path recalculation instead. It also has a few variables that can be adjusted, such as *M* and the choice of when to find a new path, so it can be adjusted (even at run-time) for different conditions. Path splicing also does not handle situations where the units must coordinate in order to pass each other.

## Watching for map changes #

An alternative to recalculating all or part of the path at certain intervals is to have changes to the map trigger a recalculation. The map can be divided into regions, and every unit can express an interest in certain regions. (All the regions that contain part of the path could be of interest, or only nearby regions that contain part of the path.) Whenever an obstacle enters or leaves a region, that region is marked *changed*, and all units that have an interest in that region are notified, so that paths can be recalculated to take into account the change in obstacles.

Many variations of this technique are possible. For example, instead of immediately notifying the units, only notify them at regular intervals. Many changes can be grouped into one notification, so that excessive path recalculations are not needed. Another example is for the unit to poll the regions instead of the regions notifying the unit.

Watching for map changes allows units to avoid recalculation whenever the obstacles on the map do not change, so consider it if you have many regions that do not change often.

## Predicting obstacle movement #

If obstacle movement can be predicted, it is possible to take into account future position of obstacles for pathfinding. An algorithm such as A* has a cost function that determines how difficult it is to pass a point on the map. A* can be modified to keep track of the time required to reach a point (determined by current path length), and this time can be passed in to the cost function. The cost function can then take time into account, and use the predicted obstacle position at that time to determine whether the map space is impassable. This modification is not perfect, however, as it will not take into account the possibility of waiting at a point for the obstacle to move out of the way, and A* is not designed to differentiate between paths along the same route, but at different points in time.

Cooperative Pathfinding A* formalizes this, building a table of the paths of all other units and treating them as obstacles. Also see the Windowed Hierarchical Cooperative A* algorithm.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# Space used by precalculated paths

From Amit's Thoughts on Pathfinding

Home Blog Links Twitter About Search

Sometimes, it's not the time needed to calculate a path, but the space used by paths for hundreds of units that is the limiting factor. Pathfinders require space for the algorithm to run, plus space to store a path. The temporary space required for the algorithm to run (with A*, the OPEN and CLOSED sets) typicaly is larger than the space required to store the resulting path. By restricting your game to compute only one path at a time, you can minimize the amount of temporary space needed. In addition, the choice of data structure for your OPEN and CLOSED sets can make a big difference for minimizing temporary space. This section will instead focus on minimizing the space used by the resulting paths.

## Locations vs. directions #

A path can be either locations or directions. Locations take more space, but have the advantage that it is easy to determine an arbitrary location or direction in the path without traversing the path. When storing directions, only the direction can be determined easily; the location can only be determined by going through the entire path, following the directions. In a typical grid map, locations may be stored with two 16-bit integers, making each step take 32 bits. There are far fewer directions, however, so they can take less space. If the unit can move in only four directions, each step takes only 2 bits; if the unit can move in six or eight directions, each step takes 3 bits. Either of these is a significant savings over storing locations in the path. Hannu Kankaanpaa suggests that you can further reduce the space requirement by storing the *relative* direction ("turn right 60 degrees") instead of the absolute direction ("go north"). Some relative directions may not make sense for some types of units. For example, if your unit is moving north, it's unlikely that the next step is to go south. In a six directional game, you have only five meaningful directions. On some maps, perhaps only three of those directions (straight, left 60 degrees, right 60 degrees) make sense, but on other maps, turning right 120 degrees may be a valid move (for example, when going up a steep mountain path with switchbacks).

## Path compression #

Once a path can be found, it can be compressed in some way. A general purpose compression algorithm could be used, but will not be discussed here. A compression algorithm specific to paths could be used to shorten either location-based paths or direction-based paths. Before deciding, look at typical paths in your game to decide which kind of compression will work best. In addition, consider ease of implementation (and debugging), the size of the code, and whether it really matters. If you have a limit of 300 units and only 50 are walking at any one time, and paths are short (100 steps), the total memory requirement might only be <50k anyway, and not worth worrying about compression.

### Location storage #

In maps where obstacles rather than terrain are the main influence in determining paths, there may be many straight-line segments in the path. If this is the case, then a path need contain only the endpoints (sometimes called *waypoints*) of those line segments. Movement consists of examining the next point on the path and moving in a straight line towards it.

### Direction storage #

When directions are stored, it may be the case that a particular direction is followed many times in a row. You can take advantage of that common pattern to store the path in less space.

One way to store the path is to store both a direction and a number which indicates how many times the unit should move in that direction. Unlike the optimization for location storage, this optimization can make things worse if a direction is not taken many times in a row. Also, for many straight lines where location compression is useful, direction compression is not, since the line may not be aligned with one of the walking directions. With relative directions, you can eliminate “keep going forward” as a possible direction. Hannu Kankaanpaa points out that with an eight direction map, you can eliminate forwards, backwards, and the 135 degree left and right turns (assuming your map allows it), and you can then store each direction with only two bits.

Another way to store the path is to use variable length encoding. The idea is to use a single bit (0) for the most common step: go straight. Use a 1 to mark a turn, and follow the 1 by some number of bits to represent the turn. In a four directional map, you can turn only left or right, so you might use 10 for left and 11 for right.

Variable length encoding is more general and may compress better than run length encoding for mixed paths, but not as well for long straight paths. The sequence (north, straight six steps, turn left, straight three steps, turn right, straight five steps, turn left, straight two steps) is represented as [(NORTH, 6), (WEST, 3), (NORTH, 5), (WEST, 2)] with run length encoding. If each direction is two bits and each distance is eight bits, this path requires 40 bits to store. With variable length encoding, you use one bit for each step and two bits for each turn—[NORTH 0 0 0 0 0 0 10 0 0 0 11 0 0 0 0 0 10 0 0]—a total of 24 bits. If the initial direction and each turn imply one step, you can save one bit per turn, resulting in 20 bits to store the path. However, longer paths can take more space with variable length encoding. The sequence (north, straight two hundred steps) is [(NORTH, 200)] with run length encoding, a total of 10 bits. The same sequence with variable length encoding is [NORTH 0 0 ...], a total of 202 bits.

## Computed waypoints #

A *waypoint* is a point along a path. Instead of storing every step along the way, after pathfinding a post-processing step can collapse multiple steps into a single waypoint, usually at places where the path changes direction or at major locations like cities. The movement algorithm will then follow a path between waypoints.

## Limited path length #

Given that map conditions or orders may change, it may not make sense to store a long path, since at some point the remainder of the path may not be of any use. Each unit can store a fixed number of steps at the beginning of the path, and then use path recalculation when the path has almost been traversed. This approach allows for control of the amount of data used per unit.

## Summary #

Paths can potentially take up a lot of space in a game, especially when the paths are long and there are many units present. Path compression, waypoints, and beacons reduce the space requirements by storing many steps in a small amount of data. Waypoints rely on straight-line segments being common so that we have to store only the endpoints, while beacons rely on there being well-known paths calculated beforehand between specially marked places on the map. If paths still take up too much space, the path length can be limited, resulting in the classic time-space tradeoff: to save space, information can be forgotten and recalculated later.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018

# Map representations

From Amit’s Thoughts on Pathfinding

Home Blog Links Twitter About Search

Through most of this document I’ve assumed that A* was being used on a grid of some sort, where the “nodes” given to A* were grid locations and the “edges” were directions you could travel from a grid location. However, A* was designed to work with arbitrary graphs, not only grids. There are a variety of map representations that can be used with A*.

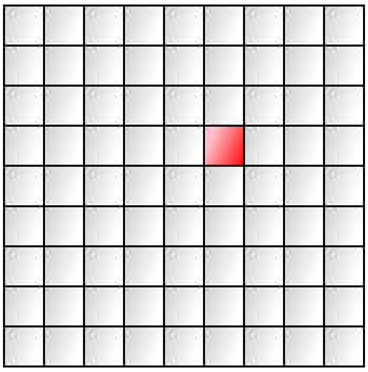
**The map representation can make a huge difference in the performance and path quality.**

Pathfinding algorithms tend to be worse than *linear*: if you double the distance needed to travel, it takes *more* than twice as long to find the path. You can think of pathfinding as searching some area like a circle—when the circle’s diameter doubles, it has *four* times the area. In general, the fewer nodes in your map representation, the faster A* will be. Also, the more closely your nodes match the positions that units will move to, the better your path quality will be.

The map representation used for pathfinding does not have to be the same as the representation used for other things in the game. However, using the same representation is a good starting point, until you find that you need better paths or more performance.

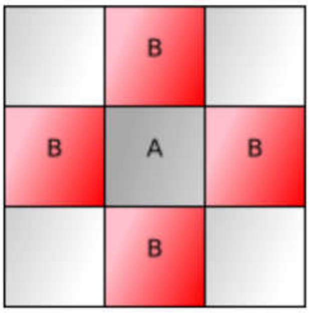
## Grids #

A grid map uses a uniform subdivision of the world into small regular shapes sometimes called “tiles”. Common grids in use are square, triangular, and hexagonal. Grids are simple and easy to understand, and many games use them for world representation; thus, I have focused on them in this document.



I used grids for BlobCity because the movement costs were different in each grid location. If your movement costs are uniform across large areas of space (as in the examples I've used in this document), then using grids can be quite wasteful. There's no need to have A* move one step at a time when it can skip across the large area to the other side. Pathfinding on a grid also yields a path on grids, which can be postprocessed to remove the jagged movement. However, if your units aren't constrained to move on a grid, or if your world doesn't even use grids, then pathfinding on a grid may not be the best choice.

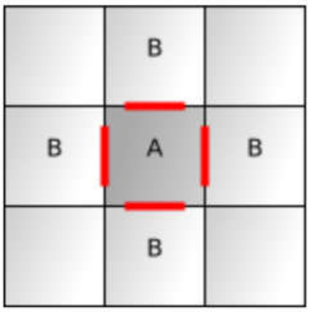
## Tile movement #



Even within grids, you have a choice of tiles, edges, and vertices for movement. Tiles are the default choice, especially for games in which units only move to the center of a tile. In this diagram, the unit at A can move to any of the spots marked B. You may also wish to allow diagonal movement, with the same or higher movement cost.

If you're using grids for pathfinding, your units are not constrained to grids, *and* movement costs are uniform, you may want to straighten the paths by moving in a straight line from one node to a node far ahead when there are no obstacles between the two.

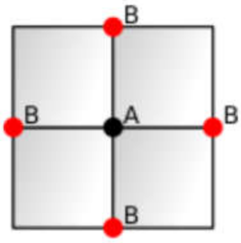
## Edge movement #



If your units can move anywhere within a grid space, or if the tiles are large, think about whether edges or vertices would be a better choice for your application.

A unit usually enters a tile at one of the edges (often in the middle) and exits the tile at another edge. With pathfinding on tiles, the unit moves to the center of the tile, but with pathfinding on edges, the unit will move directly from one edge to the other. I wrote a java applet demo of road drawing between edges; that might help illustrate how edges can be used.

## Vertex movement #

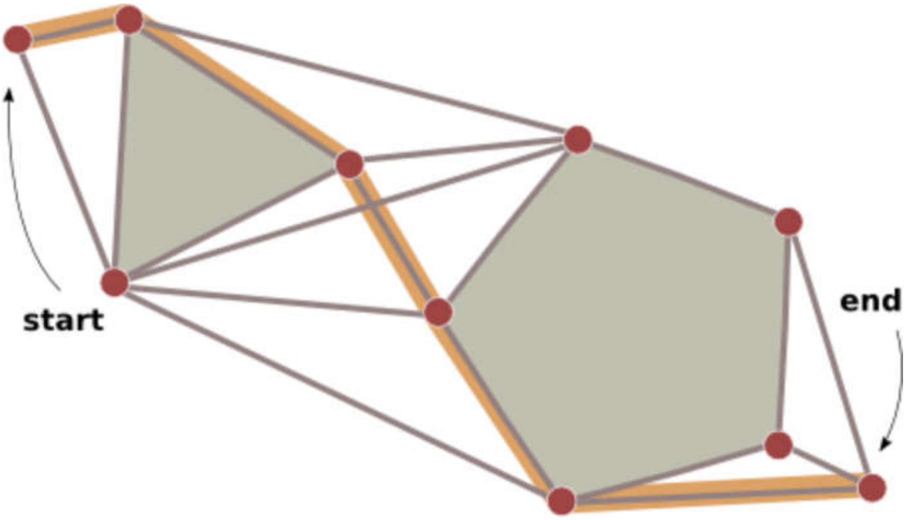


Obstacles in a grid system typically have their corners at vertices. The shortest path around an obstacle will be to go around the corners. With pathfinding on vertices, the unit moves from corner to corner. This produces the least wasted movement, but paths need to be adjusted to account for the size of the unit.

# Polygonal maps #

The most common alternative to grids is to use a polygonal representation. If the movement cost across large areas is uniform, and if your units can move in straight lines instead of following a grid, you may want to use a non-grid representation. You can use a non-grid graph for pathfinding even if your game uses a grid for other things.

Here's a simple example of one kind of polygonal map representation. In this example, the unit needs to move around two obstacles:



Imagine how your unit will move in this map. The shortest path will be between corners of the obstacles. So we choose those corners (red circles) as the key "navigation points" points to tell A* about; these can be computed once per map change. If your obstacles are aligned on a grid, the navigation points will be aligned with the vertices of the grid. In addition, the start and end points for pathfinding need to be in the graph; these are added once per call to A*.

In addition to the navigation points, A* needs to know which points are connected. The simple algorithm is to build a **visibility graph**: pairs of points that can be seen from each other. The simple algorithm may be fine for your needs, especially if the map doesn't change during gameplay, but you may need a more sophisticated algorithm if the simple one is too slow. In addition, since we have added the start and end navigation points to the graph, we check line of sight from those to existing vertices and each other, and add edges where needed.

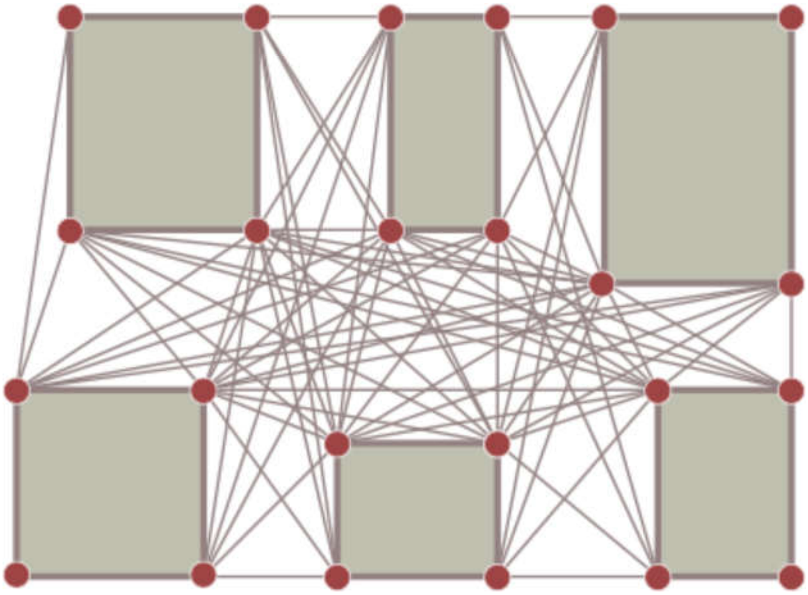
The third piece of information A* needs is travel times between the points. That will be manhattan distance or diagonal grid distance if your units move on a grid, or straight line distance if they can move directly between the navigation points.

A* will then consider paths from navigation point to navigation point. The pink line is one such path. This is *much* faster than looking for paths from grid point to grid point, when you have only a few navigation points, instead of lots of grid locations. When there are no obstacles in the way, A* will do very well—the start point and end point will be connected by an edge, and A* will find that path immediately, without expanding any other navigation points. Even when there are obstacles to consider, A* will jump from corner to corner until it finds the best path, which will still be much faster than looking for a path from a grid location to another.

Wikipedia has more about visibility graphs from the robotics literature. This slide deck is a nice introduction as well.

## Managing complexity #

The above example was rather simple and the graph is reasonable. In some maps with lots of open areas or long corridors, a problem with visibility graphs becomes apparent. A major disadvantage of connecting every pair of obstacle corners is that if there are N corners (vertices), you have up to $N^2$ edges. This example demonstrates the problem:



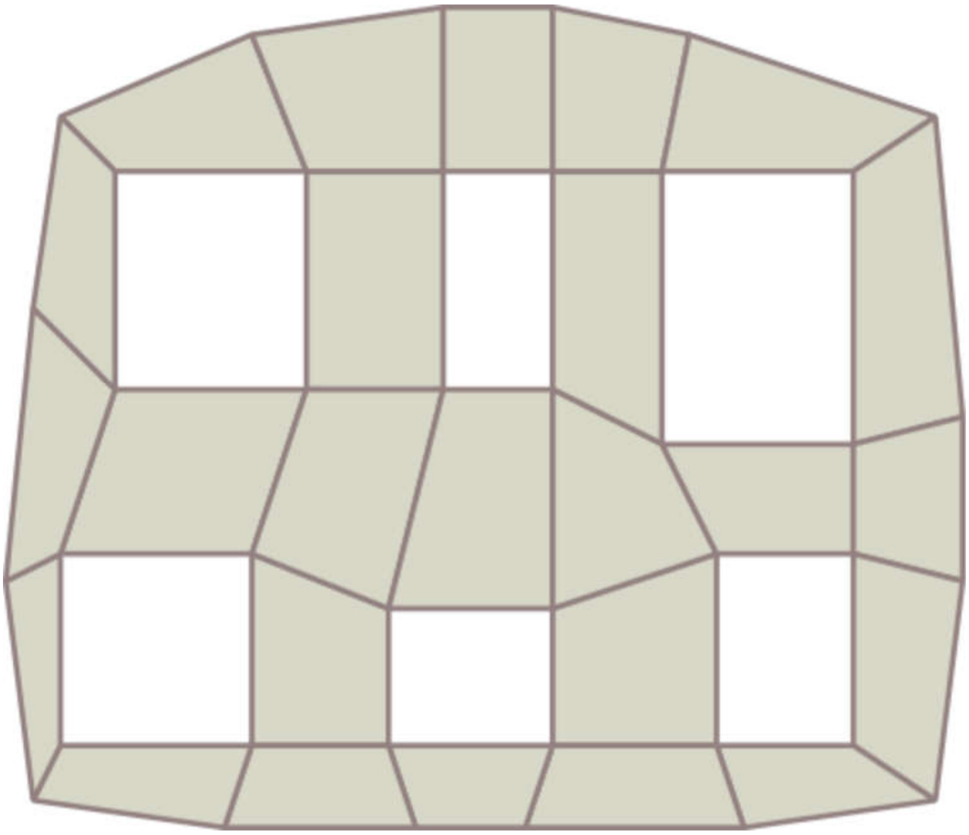
These extra edges primarily affect memory usage. Compared to grids, these edges provide “shortcuts” that greatly speed up pathfinding. There are algorithms for simplifying the graph by removing redundant edges. However, even after removing redundancies, there will still be a large number of edges.

Another disadvantage of the visibility graphs is that we have to add start/end nodes along with their new edges to the graph for every invocation of A*, and then remove them after we find a path. The nodes are easy to add but adding edges requires line of sight from the new nodes to all existing nodes, and that can be slow in large maps. One optimization is to only look at nearby nodes. Another option is to use a *reduced visibility graph* that removes the edges that aren't tangent to both vertices (these will never be in the shortest path).

## Navigation Meshes #

Instead of representing the *obstacles* with polygons, we can represent the *walkable* areas with non-overlapping polygons, also called a *navigation mesh*. The walkable areas can have additional information attached to them (such as "requires swimming" or "movement cost 2"). Obstacles don't need to be stored in this representation.

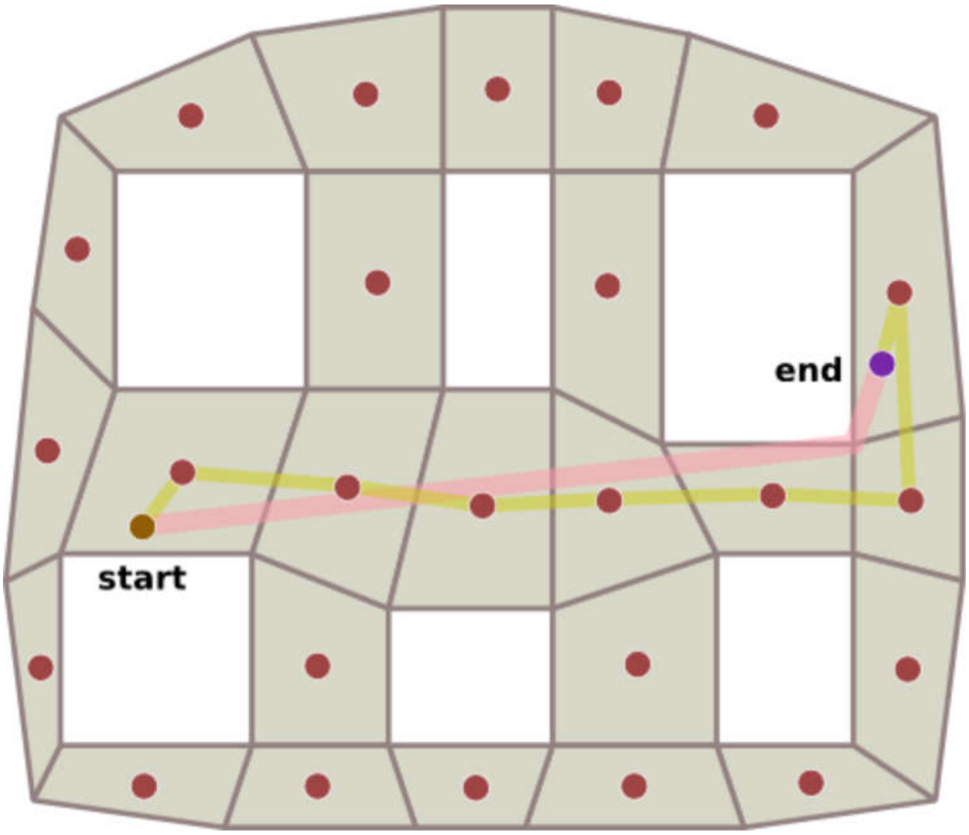
The previous example becomes this:



We can then treat this much like we treat a grid. As with a grid, we have a choice of using polygon centers, edges, or vertices as navigation points.

## Polygon movement #

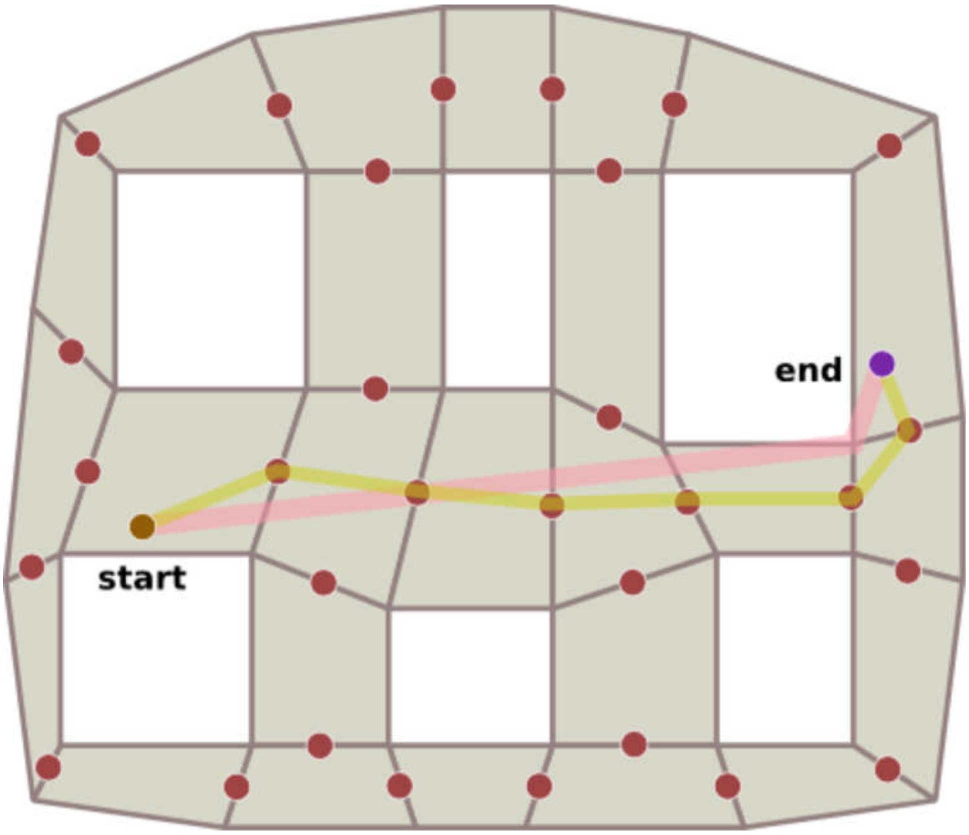
As with grids, the center of each polygon provides a reasonable set of nodes for the pathfinding graph. In addition, we have to add the start and end nodes, along with an edge to the center of the polygon we're in. In this example, the yellow path is the what we'd find using a pathfinder through the polygon centers, and the pink path is the ideal path.



The visibility graph representation would produce the pink path, which is ideal. Using a navigation mesh makes the map manageable but the path quality suffers. We can make the path look better by smoothing it.

## Polygon edge movement #

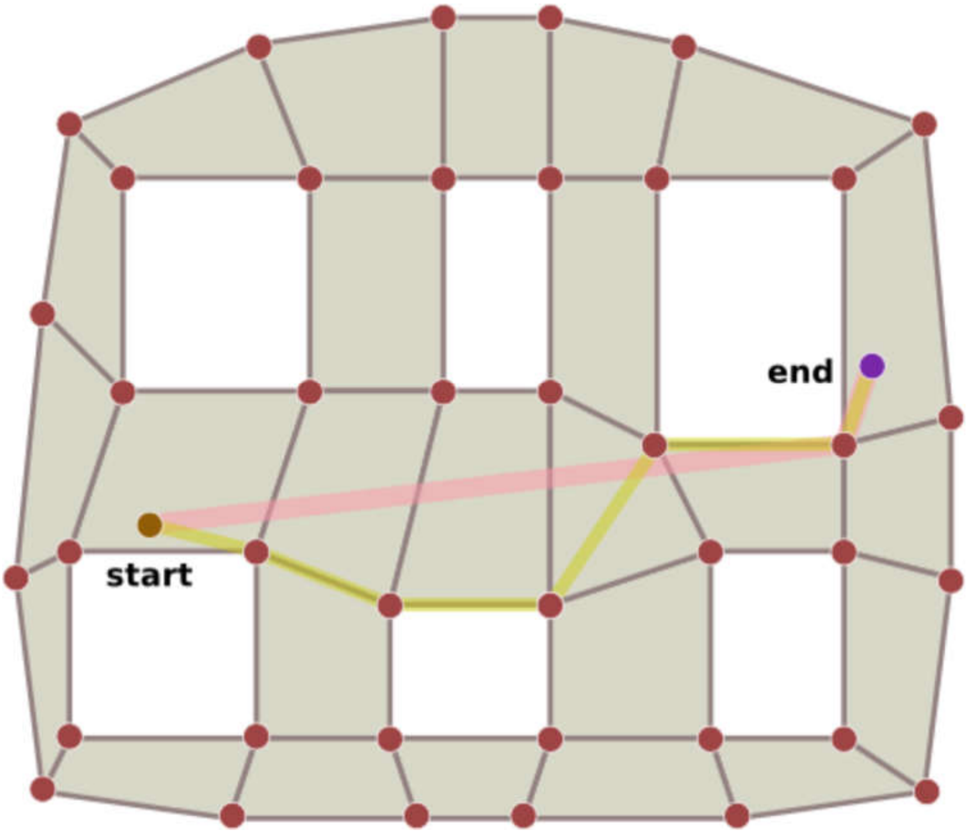
Moving to the center of the polygon is usually unnecessary. Instead, we can move through the edges between adjacent polygons. In this example, I picked the center of each edge. The yellow path is what we'd find with a pathfinder through the edge centers, and it compares pretty well to the ideal pink path.



You can pick more points along the edge to produce a better path, at increased cost.

## Polygon vertex movement #

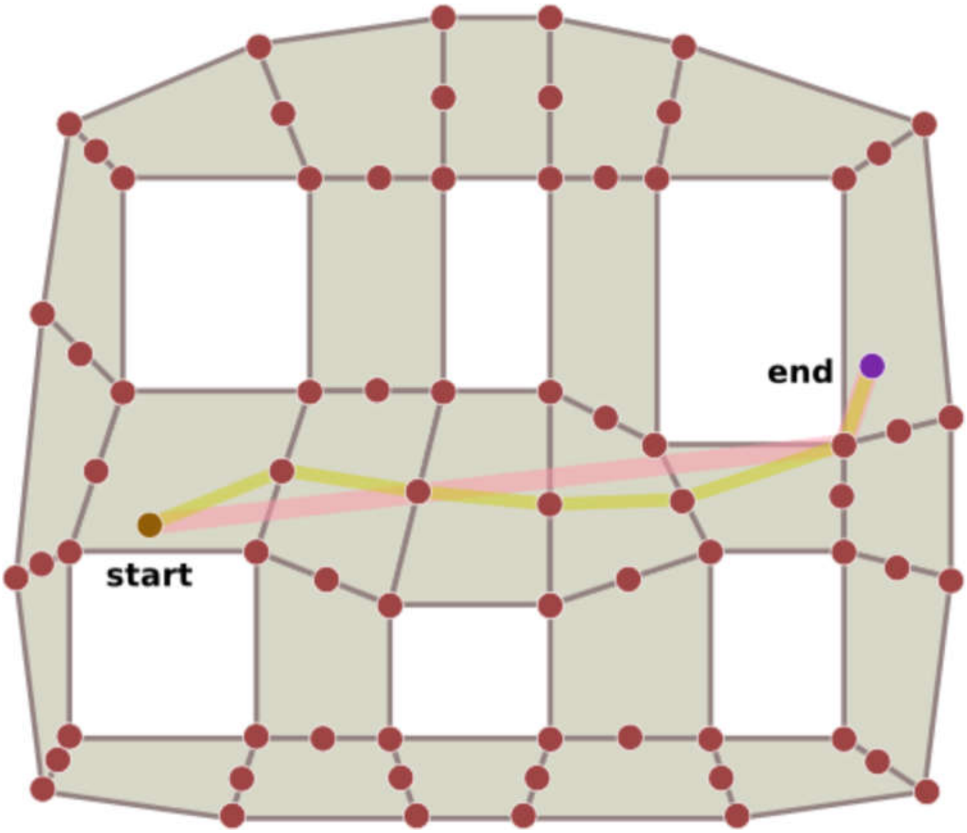
The shortest way around an obstacle is to go around the corner. This is why we used corners for the visibility graph representation. We can use vertices with navigation meshes:



There's only one obstacle in the way in this example. When we need to go around the obstacle, the yellow path goes through a vertex, as the pink (ideal) path does. However, whereas the visibility graph approach would have a straight line from the start point to the corner of the obstacle, the navigation mesh adds some more steps. These steps typically should not go through vertices, so the path looks unnatural, with "wall hugging" behavior.

## Hybrid movement #

There aren't any restrictions on what parts of each polygon can be made into navigation points for pathfinding. You can add multiple points along an edge, and the vertices are good points too. Polygon centers are rarely useful. Here's a hybrid scheme that uses both the edge centers and vertices:



Note that to get around the obstacle, the path goes through a vertex, but elsewhere, it can go through edge centers.

## Path smoothing #

Path smoothing is fairly easy with the resulting paths, as long as the movement costs are constant. The algorithm is simple: if there’s line of sight from the navigation point *i* to point *i+2*, remove point *i+1*. Repeat this until there is no line of sight between adjacent points in the path.

What will be left is only the navigation points that go around the corners of obstacles. These are vertices of the navigation mesh. If you use path smoothing, there’s no need to use edge or polygon centers as navigation points; use only the vertices.

In the above examples, path smoothing would turn the yellow path into the pink one. However, the pathfinder has no knowledge of these shorter paths, so its decisions won’t be optimal. Shortening the path found in an approximate map representation (navigation meshes) will not always produce paths that are as good as those found in a more exact representation (visibility graphs).

## Hierarchical #

A flat map has but one level in its representation. It's rare for games to have only one level—often there is a "tile" level and then a "sub-tile" level in which objects can move within a tile. However, it's common for *pathfinding* to occur on only the larger level. You can also add higher levels such as "rooms".

Having fewer nodes in the map representation is better for pathfinding speed. One way to reduce the problem is to have multiple levels of searching. For example, to get from your home to a location in another city, you would find a path from your chair to your car, from the car to the street, from the street to a freeway, from the freeway to the edge of the city, from there to the other city, then to a street, to a parking lot, and finally to the door of the destination building. There are several levels of searching here:

- At the *street* level, you are concerned with walking from one location to a nearby location, but you do not go out on the street.
- At the *city* level, you go from one street to another until you find the freeway. You do not worry about going into buildings or parking lots, nor do you worry about going on freeways.
- At the *state* level, you go from one city to another on the freeway. You do not worry about streets within cities until you get to your destination city.

Dividing the problem into levels allows you to ignore most of your choices. When moving from city to city, it is quite tedious to consider every street in every city along the way. Instead, you ignore them all, and only consider freeways. The problem becomes small and manageable, and solving it becomes fast.

A hierarchical map has many levels in its representation. A heterogenous hierarchy typically has a fixed number of levels, each with different characteristics. Ultima V, for example, has a "world" map, on which are cities and dungeons. You can enter a city or dungeon and be in a second map level. In addition, there are "layers" of worlds on top of one another, making for a three-level hierarchy. The levels can be of different types (tile grids, visibility, navigation mesh, waypoints). A homogeneous hierarchy has an arbitrary number of levels, each with the same characteristics. Quad trees and oct trees can be considered to be homogeneous hierarchies.

In a hierarchical map, pathfinding may occur on several levels. For example, if a 1024x1024 world was divided into 64x64 “zones”, it may be reasonable to find a path from the player’s location to the edge of the zone, then from zone to zone until reaching the desired zone, then from the edge of that zone to the desired location. At the coarser levels, long paths can be found more easily because the pathfinder does not consider all of the details. When the player actually walks across each zone, the pathfinder can be invoked again to find a short path through that zone. By keeping the problem size small, the pathfinder can run quicker.

You can use multiple levels with graph-searching algorithms such as A*, but you do not need to use the same algorithm at each level. For small levels, you may be able to precompute the shortest path between all pairs of nodes (using Floyd-Warshall or other algorithms). In general, pathfinding in a hierarchical map will not produce optimal paths, but they are usually close.

A similar approach is to use varying resolution. First, plot a path with low resolution. As you get closer to a point, refine the path with a higher resolution. This approach can be used with path splicing to avoid moving obstacles.

Some papers to read: Pathfinding for Dragon Age:Origins explains several hierarchical approaches used in a commercial game, Ultrafast shortest-path queries with linear-time preprocessing by using “transit nodes” for a road graph [PDF], Transit nodes for grid maps in games, Hierarchical A*: Searching Abstraction Hierarchies Efficiently, Route Planning in Road Networks (Dominic Schultes’s PhD thesis), Hierarchical Annotated A* (part 1 and part 2 and source code).

## Wraparound maps #

If your world is spherical or toroidal, then objects can “wrap” around from one end of the map to the other. The shortest path could lie in any direction, so all directions must be explored. If using a grid, you can adapt the heuristic to consider wrapping around. Instead of `abs(x1 - x2)` you can use `min(abs(x1 - x2), (x1+mapsize) - x2, (x2+mapsize) - x1)`. This will take the `min` of three options: either staying on the map without wrapping, wrapping when `x1` is on the left side, or wrapping when `x2` is on the left side. You’d do the same for each axis that wraps. Essentially you calculate the heuristic assuming that the map is adjacent to copies of itself.

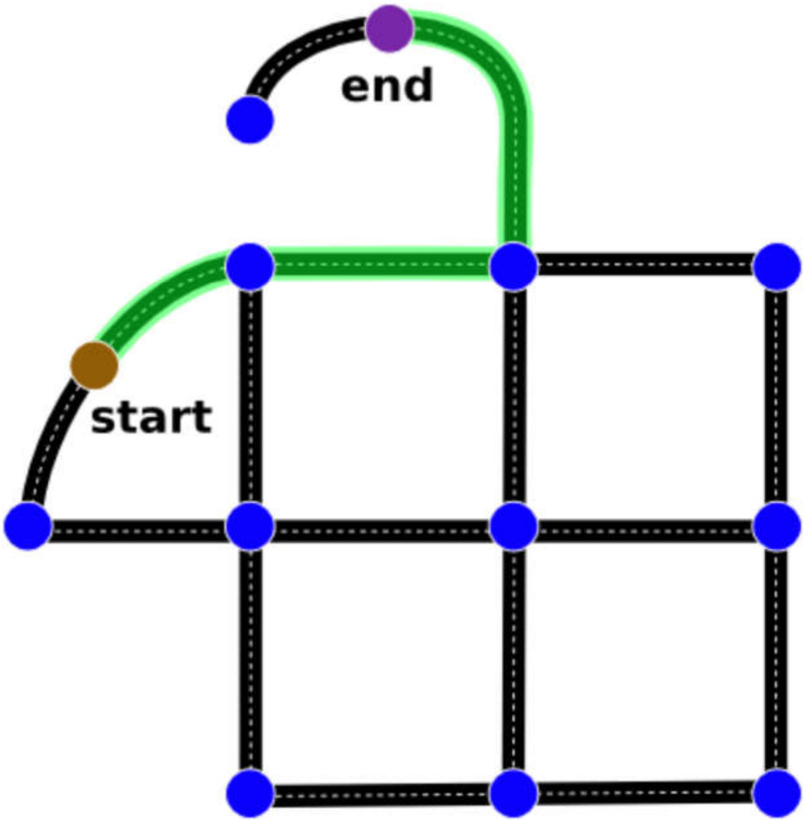
## Connected Components #

In some game maps, there's no path between the source and destination. If you ask A* to find a path, it will end up exploring a large subset of the graph before it determines that there's no path. If the map can be analyzed beforehand, mark each of the connected subgraphs with a different marker. Then, before looking for a path, check if the source and destination are both in the same subgraph. If not, then you know there's no path between them. Hierarchical pathfinding can also be useful here, especially if there are one way edges between subgraphs.

## Road maps #

If your units can only move on roads, you may want to consider giving A* the road and intersection information. Each intersection will be a node in the graph, and each road will be an edge. A* will find paths from intersection to intersection, which is much faster than using a grid representation.

For some applications, your units may not start and end on intersections. To handle this case, each time you run A*, you will need to modify the node/edge graph (this is the same technique we use with visibility graph and navigation mesh map representations). Add the starting and ending points as new nodes, and add edges between these points and their nearest intersections. After pathfinding, remove these extra nodes and edges from the graph so that the graph is ready to be used for the next invocation of A*.



In this diagram, the intersections become nodes in the pathfinding graph for A*. The edges are the roads between the nodes, and these edges should be given the driving distance along each road. In the “roads as edges” framework, you can incorporate one-way roads as unidirectional edges in the graph.

If you want to assign costs to turning, you can extend the framework a bit: instead of nodes being locations, consider nodes to be a <location, direction> pair (a point in *state space*), where the direction indicates what direction you were facing when you *arrived* at that location. Replace edges from X to Y with edges from <X, dir> to <Y, dir> to represent a straight drive, and from <X, dir1> to <X, dir2> to represent a “turn”. Each edge represents *either* a straight drive or a turn, but not both. You can then assign costs to the edges representing turns.

If you also need to take into account turn limitations, such as “only right turns”, you can use a variation of this framework in which the two types of edges are always combined. Each edge represents an optional turn followed by a straight drive. In this framework, you can represent restrictions like “you can only turn right”: include an edge from <X, north> to <Y, north> for driving straight, and an edge from <X, north> to <Z, east> for the right turn followed by a drive, but *don’t* include <X, north> to anything west, because that would mean a left turn, and don’t include anything south, because that would mean a U-turn.

In this framework, you can model a large city downtown, in which you have one-way streets, turn restrictions at certain intersections (often prohibiting U-turns and sometimes prohibiting left turns), and turn costs (to model slowing down and waiting for pedestrians before you turn right). Compared to grid maps, A* can find paths in road graphs environment fairly quickly, because there are few choices to make at each graph node, and there are relatively few nodes in the map.

For large scale road maps, be sure to read Goldberg and Harrelson's paper on ALT (A*, Landmarks, Triangle inequality) (PDF is here, or this paper.

## Skip links #

A pathfinding graph constructed from a grid typically assigns a vertex to each location and an edge to each possible movement from a location to an adjacent location. The edges are not constrained to be between adjacent vertices. A "skip link" or "shortcut link" is an edge between non-adjacent vertices. It serves to shortcut the pathfinding process.

What should the movement cost be for a skip link? There are two approaches:

- Make the cost match the movement cost of the best path. This preserves nice properties of A*, like finding optimal paths. To give A* a nudge in the right direction, break the tie between the skip link and the regular links by reducing the skip link's cost by 1% or so.
- Make the cost match the heuristic cost. This makes a much stronger impact on performance but you give up optimal paths.

Adding skip links is an approximation of a hierarchical map. It takes less effort but can often give you many of the same performance benefits.

For dungeon room-and-corridor grid maps, Rectangular Symmetry Reduction and Jump Point Search offer two different ways to build skip links. Rectangular Symmetry Reduction statically builds additional edges (which they call *macro edges*) and then uses standard graph search; Jump Point Search dynamically builds longer edges as part of the graph search algorithm. For road maps and other types of graphs Contraction Hierarchies are worth looking at; see this paper. When I wrote this page in 1997 I didn't know about contraction hierarchies, or I would've used that terminology instead of making up the term “skip links”.

## Waypoints #

A *waypoint* is a point along a path. Waypoints can specific to each path or be part of the game map. Waypoints can be entered manually or computed automatically. In many real-time strategy games, players can manually add path-specific waypoints by shift-clicking. When automatically computed along a path, waypoints can be used to compress the path representation. Map designers can manually add waypoints (or “beacons”) to a map to mark locations that are along good paths or an algorithm can be used to automatically mark waypoints on a map.

Since the goal of skip links is to make pathfinding faster when those links are used, skip links should be placed between designer-placed waypoints. This will maximize their benefit.

If there are not too many waypoints, the shortest paths between each pair of waypoints can be precomputed beforehand (using an all-pairs shortest path algorithm, not A*). The common case will then be a unit following its own path until it reaches a waypoint, and then it will follow the precomputed shortest path between waypoints, and finally it will get off the waypoint highway and follow its own path to the goal.

Using waypoints or skip links with false costs can lead to suboptimal paths. It is sometimes possible to smooth out a bad path in a post-processing step or in the movement algorithm.

## Graph Format Recommendations #

Start by pathfinding on the game world representation you already use. If that’s not satisfactory, consider transforming the game world into a different representation for pathfinding.

In many grid games, there are large areas of maps that have uniform movement costs. A* doesn’t “know” this, and wastes effort exploring them. Creating a simpler graph (navigation mesh, visibility graph, or hierarchical representation of the grid map) can help A*.

The visibility graph representation produces the best paths when movement costs are constant, and allows A* to run rather quickly, but can use lots of memory for edges. Grids allow for fine variation in movement costs (terrain, slope, penalties for dangerous areas, etc.), use very little memory for edges, but use lots of memory for nodes, and pathfinding can be slow. Navigation meshes are in between. They work well when movement costs are constant in a larger area, allow for some variation in movement costs, and produce reasonable paths. The paths are not always as short as with visibility graph representation, but they are usually reasonable. Hierarchical maps use multiple levels of representation to handle both coarse paths over long distances and detailed paths over short distances.

You can read more about navigation meshes in this well-illustrated article. Note that the article is comparing (a) keeping walkable polygons to keeping only navigation points, and (b) moving along vertices to moving along polygon centers. These are mostly orthogonal. Keeping the walkable polygons is nice for dynamically adjusting the path later, but not needed in all games. Using vertices is better for obstacle avoidance, and if you're using path smoothing it won't negatively affect path quality. Without path smoothing, edges might perform better, so consider either edges or edges+vertices.

An alternative to building a separate non-grid representation of a grid map is to use a variant of A* that better understands grid maps with uniform costs. See Jump Point Search to speed up A* on square grids and Theta* to generate non-grid movement on a grid.

Email me at redblobgames@gmail.com, or tweet to @redblobgames, or post a public comment:

From Red Blob Games
I started writing this in 1997; last modified: 02 Mar 2018

# Long and short term goals

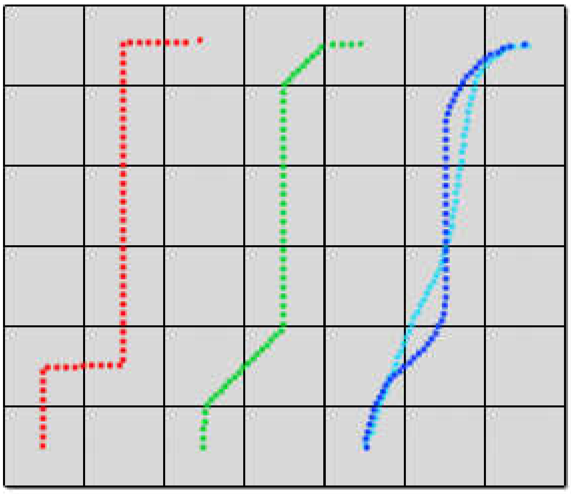
From Amit’s Thoughts on Pathfinding

I’ve concentrated on the task of finding paths from one place to another. However, an equally important question is: once I have a path, how do I move along it? The most obvious answer is moving in a straight line from one location to the next. However, you might also want to move in a curve, or have multiple levels of movement. You may want to treat the locations as a low-priority goal from which you deviate. A higher level question is: where do you want to go? Unless you first answer the higher level question, pathfinding is not very useful. Certainly, one form of goal setting is asking the user to click on the destination. However, you may have automated tasks as well—exploring, spying, attacking, and building are common ones.

## Unit movement #

I’ve concentrated on pathfinding, which reduces the problem of moving from one location to another with the many smaller problems of moving from one space to an adjacent space.

You could move in a straight line from one location to the next but there are alternatives. Consider the four movement paths on this diagram:



The red path is the standard approach: move from the center of one square to the center of the next. The green path is somewhat better: move in straight lines between the *edges* between the tiles, instead of the center of the tiles. You might also try moving in straight lines between the *corners* of tiles. The blue paths use splines, with dark blue being low order splines and light blue being a higher order spline.

Lines between corners and edges of tiles will be the shortest solution. However, the splines can make your units seem less mechanical and more “alive”. It’s a cheap trick but not a great one. There are better ways to handle movement.

If your units cannot turn easily, you may want to take that into account when plotting a movement path. Craig Reynolds has a great page about steering that has a paper about steering and Java applets demonstrating various behaviors. If you have more than one unit moving along a path, you may also want to investigate flocking. Craig recommends that instead of treating paths as a list of places your unit must visit, you treat paths as ``a guideline, from which you deviate reactively as conditions require.’‘

If you’re using grids for pathfinding, your units are not constrained to grids, *and* movement costs are uniform, you may want to straighten the paths by moving in a straight line from one node to a node far ahead when there are no obstacles between the two. If you’re using navigation meshes, look at the funnel algorithm.

## Behavior flags or stacks #

Your units may have more than one goal. For example, you may have a general goal like “spying” but also a more immediate goal like “go to the enemy headquarters”. In addition, there may be temporary goals like “avoid that patrol guard”. Here are some ideas for goals:

- **Stop**: Stay in the current location
- **Stay**: Stay in one area
- **Flee**: Move to a safe area
- **Retreat**: Move to a safe area, while fighting off enemy units
- **Explore**: Find and learn about areas for which little information is known
- **Wander**: Move around aimlessly

- **Search**: Look for a particular object
- **Spy**: Go near an object or unit to learn more about it, without being seen
- **Patrol**: Repeatedly walk through an area to make sure no enemy units go through it
- **Defend**: Stay near some object or unit to keep enemy units away
- **Guard**: Stay near the entrance to some area to keep enemy units out
- **Attack**: Move to some object or unit to capture or destroy it
- **Surround**: With other units, try to surround an enemy unit or object
- **Shun**: Move away from some object or unit
- **Avoid**: Stay away from any other units
- **Follow**: Stay near some unit as it moves around
- **Group**: Seek and form groups of units
- **Work**: Perform some task like mining, farming, or collecting

For each unit you can have a flag indicating which behavior it is to perform. To have multiple levels, keep a *behavior stack*. The top of the stack will be the most immediate goal and the bottom of the stack will be the overall goal. When you need to do something new but later want to go back to what you were doing, push a new behavior on the stack. If you instead need to do something new but *don't* want to go back to the old behavior, clear the stack. Once you are done with some goal, pop it from the stack and start performing the next behavior on the stack.

## Waiting for movement #

It is inevitable that the movement algorithm will run into obstacles that were not there during the pathfinding process. An easily implemented technique is based on the assumption that the *other* obstacle will move first. This is particularly useful when the obstacle is a friendly unit. When an obstacle is in the way, wait some amount of time for it to move. If it still hasn't moved after that period of time, recalculate a path around it or to the destination. If the obstacle is detected ahead of time, your unit can simply walk slower to give the other unit more time to get out of the way.

It is possible that two units will bump into each other, and each will wait for the other to proceed. In this case, a priority scheme can be used: assign each unit a unique number, and then make the lower numbered unit wait for the higher numbered unit. This will force one of the units to proceed if both are waiting. When obstacles are detected ahead of time, the lower numbered unit should slow down before reaching the expected point of collision.

## Coordinated movement #

The technique described above does not work when units are trying to move through a narrow corridor. If one unit stands still while the other tries to go around, the corridor can't be used by both units. One unit will block it while the other one will take a long path around.

It should be possible for the second unit to communicate with the first one, and ask it to back up. Once the corridor is clear, the second unit can pass through, and then the first unit can go through. This may be complicated to implement unless you can identify the corridors beforehand. For randomly generated maps, it could be very difficult to determine where the corridor is and how far the first unit needs to back up.

Email me at redblobgames@gmail.com, or tweet to @redblobgames,
or post a public comment:

From Red Blob Games

I started writing this in 1997; last modified: 02 Mar 2018