

4

PATHFINDING

Game characters usually need to move around their level. Sometimes this movement is set in stone by the developers, such as a patrol route that a guard can follow blindly or a small fenced region in which a dog can randomly wander around. Fixed routes are simple to implement, but can easily be fooled if an object is pushed in the way. Free wandering characters can appear aimless and can easily get stuck.

More complex characters don't know in advance where they'll need to move. A unit in a real-time strategy game may be ordered to any point on the map by the player at any time, a patrolling guard in a stealth game may need to move to its nearest alarm point to call for reinforcements, and a platform game may require opponents to chase the player across a chasm using available platforms.

For each of these characters the AI must be able to calculate a suitable route through the game level to get from where it is now to its goal. We'd like the route to be sensible and as short or rapid as possible (it doesn't look smart if your character walks from the kitchen to the lounge via the attic).

This is pathfinding, sometimes called path planning, and it is everywhere in game AI.

In our model of game AI (Figure 4.1), pathfinding sits on the border between decision making and movement. Often, it is used simply to work out where to move to reach a goal; the goal is decided by another bit of AI, and the pathfinder simply works out how to get there. To accomplish this, it can be embedded in a movement control system so that it is only called when it is needed to plan a route. This is discussed in Chapter 3 on movement algorithms.

But pathfinding can also be placed in the driving seat, making decisions about where to move as well as how to get there. We'll look at a variation of pathfinding, open goal pathfinding, that can be used to work out both the path and the destination.

The vast majority of games use pathfinding solutions based on an algorithm called A*. Although it's efficient and easy to implement, A* can't work directly with the game level data. It

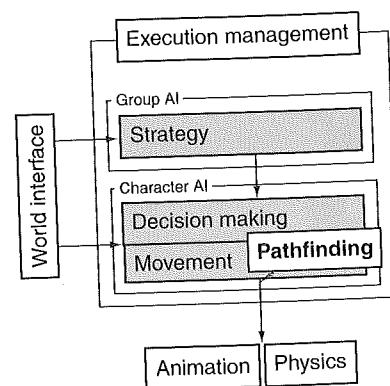


Figure 4.1 The AI model

requires that the game level be represented in a particular data structure: a directed non-negative weighted graph.

This chapter introduces the graph data structure and then looks at the older brother of the A* algorithm, the Dijkstra algorithm. Although Dijkstra is more often used in tactical decision making than in pathfinding, it is a simpler version of A*, so we'll cover it here on the way to the full A* algorithm.

Because the graph data structure isn't the way that most games would naturally represent their level data, we'll look in some detail at the knowledge representation issues involved in turning the level geometry into pathfinding data. Finally, we'll look at a handful of the many tens of useful variations of the basic A* algorithm.

4.1 THE PATHFINDING GRAPH

Neither A* nor Dijkstra (nor their many variations) can work directly on the geometry that makes up a game level. They rely on a simplified version of the level to be represented in the form of a graph. If the simplification is done well (and we'll look at how later in the chapter), then the plan returned by the pathfinder will be useful when translated back into game terms. On the other hand, in the simplification we throw away information, and that might be significant information. Poor simplification can mean that the final path isn't so good.

Pathfinding algorithms use a type of graph called a directed non-negative weighted graph. We'll work up to a description of the full pathfinding graph via simpler graph structures.

4.1.1 GRAPHS

A graph is a mathematical structure often represented diagrammatically. It has nothing to do with the more common use of the word "graph" to mean any diagram, such as a pie chart or histogram.

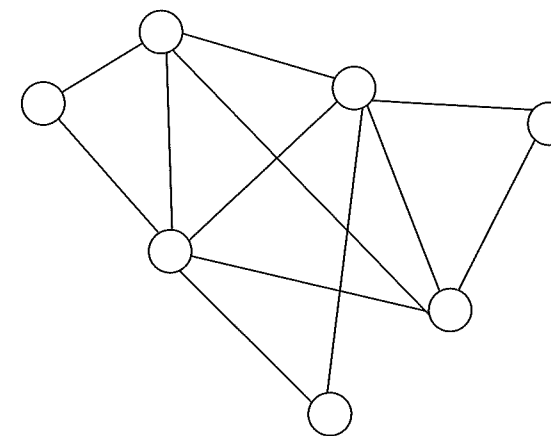


Figure 4.2 A general graph

A graph consists of two different types of element: nodes are often drawn as points or circles in a graph diagram, while connections link nodes together with lines. Figure 4.2 shows a graph structure.

Formally, the graph consists of a set of nodes and a set of connections, where a connection is simply an unordered pair of nodes (the nodes on either end of the connection).

For pathfinding, each node usually represents a region of the game level, such as a room, a section of corridor, a platform, or a small region of outdoor space. Connections show which locations are connected. If a room adjoins a corridor, then the node representing the room will have a connection to the node representing the corridor. In this way the whole game level is split into regions, which are connected together. Later in the chapter, we'll see a way of representing the game level as a graph that doesn't follow this model, but in most cases this is the approach taken.

To get from one location in the level to another, we use connections. If we can go directly from our starting node to our target node, then life is simple. Otherwise, we may have to use connections to travel through intermediate nodes on the way.

A path through the graph consists of zero or more connections. If the start and end node are the same, then there are no connections in the path. If the nodes are connected, then only one connection is needed, and so on.

4.1.2 WEIGHTED GRAPHS

A weighted graph is made up of nodes and connections, just like the general graph. In addition to a pair of nodes for each connection, we add a numerical value. In mathematical graph theory this is called the *weight*, and in game applications it is more commonly called the *cost* (although the graph is still called a "weighted graph" rather than a "costed graph").

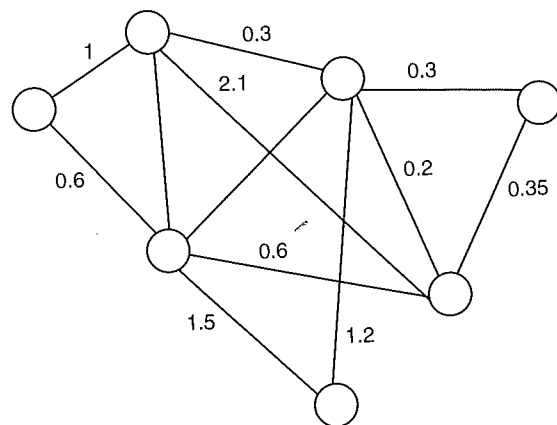


Figure 4.3 A weighted graph

Drawing the graph (Figure 4.3), we see that each connection is labeled with an associated cost value.

The costs in a pathfinding graph often represent time or distance. If a node representing a platform is a long distance from a node representing the next platform, then the cost of the connection will be large. Similarly, moving between two rooms that are both covered in traps will take a long time, so the cost will be large.

The costs in a graph can represent more than just time or distance. We will see a number of applications of pathfinding to situations where the cost is a combination of time, distance, and other factors.

For a whole route through a graph, from a start node to a target node, we can work out the total path cost. It is simply the sum of the costs of each connection in the route. In Figure 4.4, if we are heading from node A to node C, via node B, and if the costs are 4 from A to B and 5 from B to C, then the total cost of the route is 9.

Representative Points in a Region

You might notice immediately that if two regions are connected (such as a room and a corridor), then the distance between them (and therefore the time to move between them) will be zero. If you are standing in a doorway, then moving from the room side of the doorway to the corridor side is instant. So shouldn't all connections have a zero cost?

We tend to measure connection distances or times from a representative point in each region. So we pick the center of the room and the center of the corridor. If the room is large and the corridor is long, then there is likely to be a large distance between their center points, so the cost will be large.

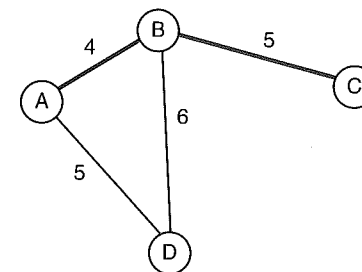


Figure 4.4 Total path cost

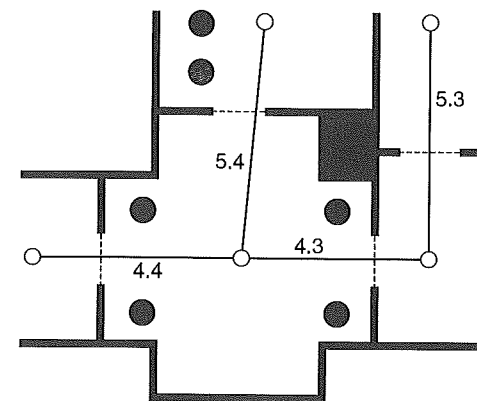


Figure 4.5 Weighted graph overlaid onto level geometry

You will often see this in diagrams of pathfinding graphs, such as Figure 4.5: a representative point is marked in each region.

A complete analysis of this approach will be left to a later section. It is one of the subtleties of representing the game level for the pathfinder, and we'll return to the issues it causes at some length.

The Non-Negative Constraint

It doesn't seem to make sense to have negative costs. You can't have a negative distance between two points, and it can't take a negative amount of time to move there.

Mathematical graph theory does allow negative weights, however, and they have direct applications in some practical problems. These problems are entirely outside of normal game development, and all of them are beyond the scope of this book. Writing algorithms that can work with negative weights is typically more complex than for those with strictly non-negative weights.

In particular, the Dijkstra and A* algorithms should only be used with non-negative weights. It is possible to construct a graph with negative weights such that a pathfinding algorithm will return a sensible result. In the majority of cases, however, Dijkstra and A* would go into an infinite loop. This is not an error in the algorithms. Mathematically, there is no such thing as a shortest path across many graphs with negative weights; a solution simply doesn't exist.

When we use the term "cost" in this book, it means a non-negative weight. Costs are always positive. We will never need to use negative weights or the algorithms that can cope with them. We've never needed to use them in any game development project we've worked on, and we can't foresee a situation when we might.

4.1.3 DIRECTED WEIGHTED GRAPHS

For many situations a weighted graph is sufficient to represent a game level, and we have seen implementations that use this format. We can go one stage further, however. The major pathfinding algorithms support the use of a more complex form of graph, the directed graph (see Figure 4.6), which is often useful to developers.

So far we've assumed that if it is possible to move between node A and node B (the room and corridor, for example), then it is possible to move from node B to node A. Connections go both

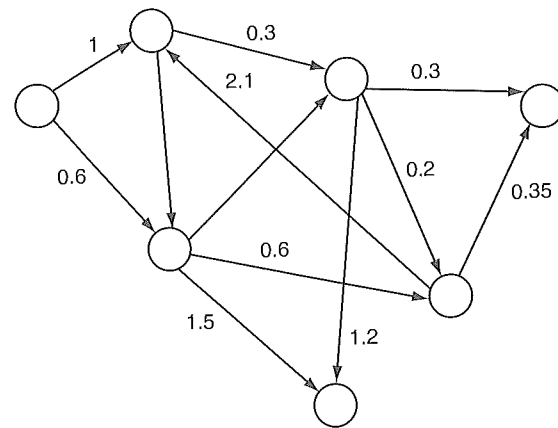


Figure 4.6 A directed weighted graph

ways, and the cost is the same in both directions. Directed graphs instead assume that connections are in one direction only. If you can get from node A to node B, and vice versa, then there will be two connections in the graph: one for A to B and one for B to A.

This is useful in many situations. First, it is not always the case that the ability to move from A to B implies that B is reachable from A. If node A represents an elevated walkway and node B represents the floor of the warehouse underneath it, then a character can easily drop from A to B but will not be able to jump back up again.

Second, having two connections in different directions means that there can be two different costs. Let's take the walkway example again but add a ladder. Thinking about costs in terms of time, it takes almost no time at all to fall off the walkway, but it may take several seconds to climb back up the ladder. Because costs are associated with each connection, this can be simply represented: the connection from A (the walkway) to B (the floor) has a small cost, and the connection from B to A has a larger cost.

Mathematically, a directed graph is identical to a non-directed graph, except that the pair of nodes that makes up a connection is now ordered. Whereas a connection (node A, node B, cost) in a non-directed graph is identical to (node B, node A, cost) (so long as the costs are equal), in a directed graph they are different connections.

4.1.4 TERMINOLOGY

Terminology for graphs varies. In mathematical texts you often see vertices rather than nodes and edges rather than connections (and, as we've already seen, weights rather than costs). Many AI developers who actively research pathfinding use this terminology from exposure to the mathematical literature. It can be confusing in a game development context because vertices more commonly mean something altogether different.

There is no agreed terminology for pathfinding graphs in games articles and seminars. We have seen locations and even "dots" for nodes, and we have seen arcs, paths, links, and "lines" for connections.

We will use the nodes and connections terminology throughout this chapter because it is common, relatively meaningful (unlike dots and lines), and unambiguous (arcs and vertices both have meaning in game graphics).

In addition, while we have talked about directed non-negative weighted graphs, almost all pathfinding literature just calls them graphs and assumes that you know what kind of graph is meant. We'll do the same.

4.1.5 REPRESENTATION

We need to represent our graph in such a way that pathfinding algorithms such as A* and Dijkstra can work with it.

As we will see, the algorithms need to find out the outgoing connections from any given node. And for each such connection, they need to have access to its cost and destination.

We can represent the graph to our algorithms using the following interface:

```

1 class Graph:
2     # Returns an array of connections (of class
3     # Connection) outgoing from the given node
4     def getConnections(fromNode)
5
6 class Connection:
7     # Returns the non-negative cost of the
8     # connection
9     def getCost()
10
11    # Returns the node that this connection came
12    # from
13    def getFromNode()
14
15    # Returns the node that this connection leads to
16    def getToNode()

```

The graph class simply returns an array of connection objects for any node that is queried. From these objects the end node and cost can be retrieved.

A simple implementation of this class would store the connections for each node and simply return the list. Each connection would have the cost and end node stored in memory.

A more complex implementation might calculate the cost only when it is required, using information from the current structure of the game level.

Notice that there is no specific data type for a node in this interface, because we don't need to specify one. In many cases it is sufficient just to give nodes a unique number and to use integers as the data type. In fact, we will see that this is a particularly powerful implementation because it opens up some specific, very fast optimizations of the A* algorithm.

4.2 DIJKSTRA

The Dijkstra algorithm is named for Edsger Dijkstra, the mathematician who devised it (and the same man who coined the famous programming phrase "GOTO considered harmful").

Dijkstra's algorithm wasn't originally designed for pathfinding as games understand it. It was designed to solve a problem in mathematical graph theory, confusingly called "shortest path."

Where pathfinding in games has one start point and one goal point, the shortest path algorithm is designed to find the shortest routes to everywhere from an initial point. The solution to this problem will include a solution to the pathfinding problem (we've found the shortest route to everywhere, after all), but it is wasteful if we are going to throw away all the other routes. It can be modified to generate only the path we are interested in, but is still quite inefficient doing that.

Because of these issues, we have seen Dijkstra used only once in production pathfinding, not as the main pathfinding algorithm but to analyze general properties of a level in the very complex pathfinding system of a military simulation.

Nonetheless, it is an important algorithm for tactical analysis (covered in Chapter 6, Tactical and Strategic AI) and has uses in a handful of other areas of game AI. We will examine it here because it is a simpler version of the main pathfinding algorithm A*.

4.2.1 THE PROBLEM

Given a graph (a directed non-negative weighted graph) and two nodes (called *start* and *goal*) in that graph, we would like to generate a path such that the total path cost of that path is minimal among all possible paths from start to goal.

There may be any number of paths with the same minimal cost. Figure 4.7 has 10 possible paths, all with the same minimal cost. When there is more than one optimal path, we only expect one to be returned, and we don't care which one it is.

Recall that the path we expect to be returned consists of a set of connections, not nodes. Two nodes may be linked by more than one connection, and each connection may have a different cost (it may be possible to either fall off a walkway or climb down a ladder, for example). We therefore need to know which connections to use; a list of nodes will not suffice.

Many games don't make this distinction. There is, at most, one connection between any pair of nodes. After all, if there are two connections between a pair of nodes, the pathfinder should always take the one with the lower cost. In some applications, however, the costs change over the course of the game or between different characters, and keeping track of multiple connections is useful.

There is no more work in the algorithm to cope with multiple connections. And for those applications where it is significant, it is often essential. We'll always assume a path consists of connections.

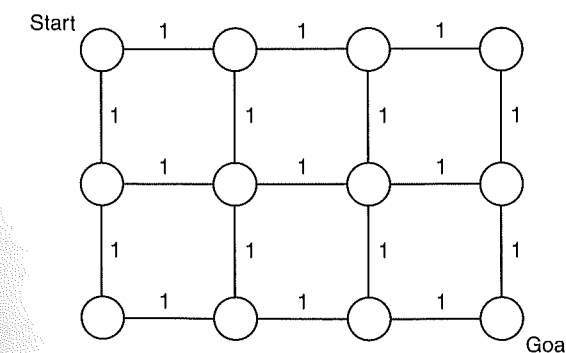


Figure 4.7 All optimal paths

4.2.2 THE ALGORITHM

Informally, Dijkstra works by spreading out from the start node along its connections. As it spreads out to more distant nodes, it keeps a record of the direction it came from (imagine it drawing chalk arrows on the floor to indicate the way back to the start). Eventually, it will reach the goal node and can follow the arrows back to its start point to generate the complete route. Because of the way Dijkstra regulates the spreading process, it guarantees that the chalk arrows always point back along the shortest route to the start.

Let's break this down in more detail.

Dijkstra works in iterations. At each iteration it considers one node of the graph and follows its outgoing connections. At the first iteration it considers the start node. At successive iterations it chooses a node to consider using an algorithm we'll discuss shortly. We'll call each iteration's node the "current node."

Processing the Current Node

During an iteration, Dijkstra considers each outgoing connection from the current node. For each connection it finds the end node and stores the total cost of the path so far (we'll call this the "cost-so-far"), along with the connection it arrived there from.

In the first iteration, where the start node is the current node, the total cost-so-far for each connection's end node is simply the cost of the connection. Figure 4.8 shows the situation after the first iteration. Each node connected to the start node has a cost-so-far equal to the cost of the connection that led there, as well as a record of which connection that was.

For iterations after the first, the cost-so-far for the end node of each connection is the sum of the connection cost and the cost-so-far of the current node (i.e., the node from which the connection originated). Figure 4.9 shows another iteration of the same graph. Here the cost-so-far stored in node E is the sum of cost-so-far from node B and the connection cost of Connection IV from B to E.

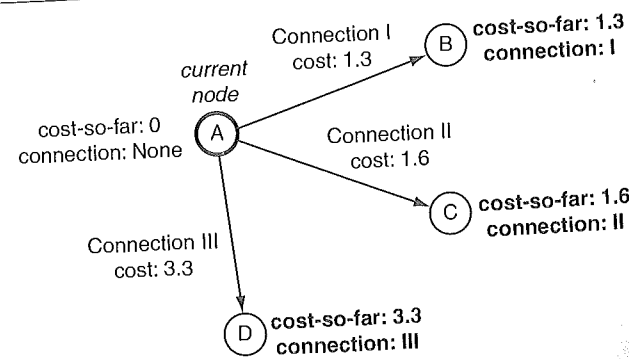


Figure 4.8 Dijkstra at the first node

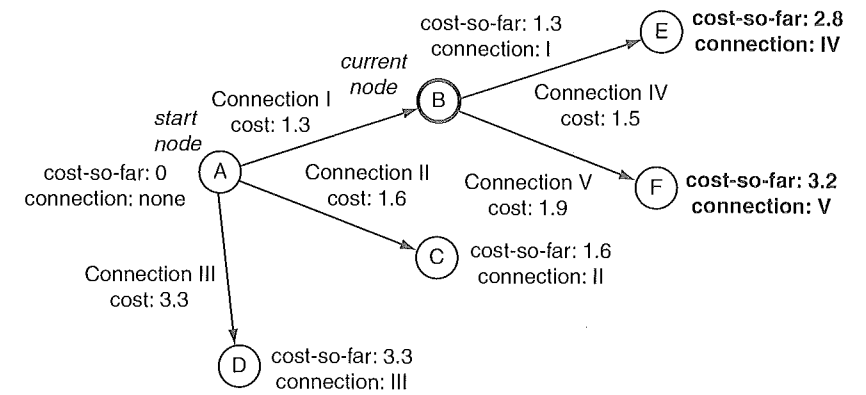


Figure 4.9 Dijkstra with a couple of nodes

In implementations of the algorithm, there is no distinction between the first and successive iterations. By setting the cost-so-far value of the start node as 0 (since the start node is at zero distance from itself), we can use one piece of code for all iterations.

The Node Lists

The algorithm keeps track of all the nodes it has seen so far in two lists: open and closed. In the open list it records all the nodes it has seen, but that haven't had their own iteration yet. It also keeps track of those nodes that have been processed in the closed list. To start with, the open list contains only the start node (with zero cost-so-far), and the closed list is empty.

Each node can be thought of as being in one of three categories: it can be in the closed list, having been processed in its own iteration; it can be in the open list, having been visited from another node, but not yet processed in its own right; or it can be in neither list. The node is sometimes said to be closed, open, or unvisited.

At each iteration, the algorithm chooses the node from the open list that has the smallest cost-so-far. This is then processed in the normal way. The processed node is then removed from the open list and placed on the closed list.

There is one complication. When we follow a connection from the current node, we've assumed that we'll end up at an unvisited node. We may instead end up at a node that is either open or closed, and we'll have to deal slightly differently with them.

Calculating Cost-So-Far for Open and Closed Nodes

If we arrive at an open or closed node during an iteration, then the node will already have a cost-so-far value and a record of the connection that led there. Simply setting these values will overwrite the previous work the algorithm has done.

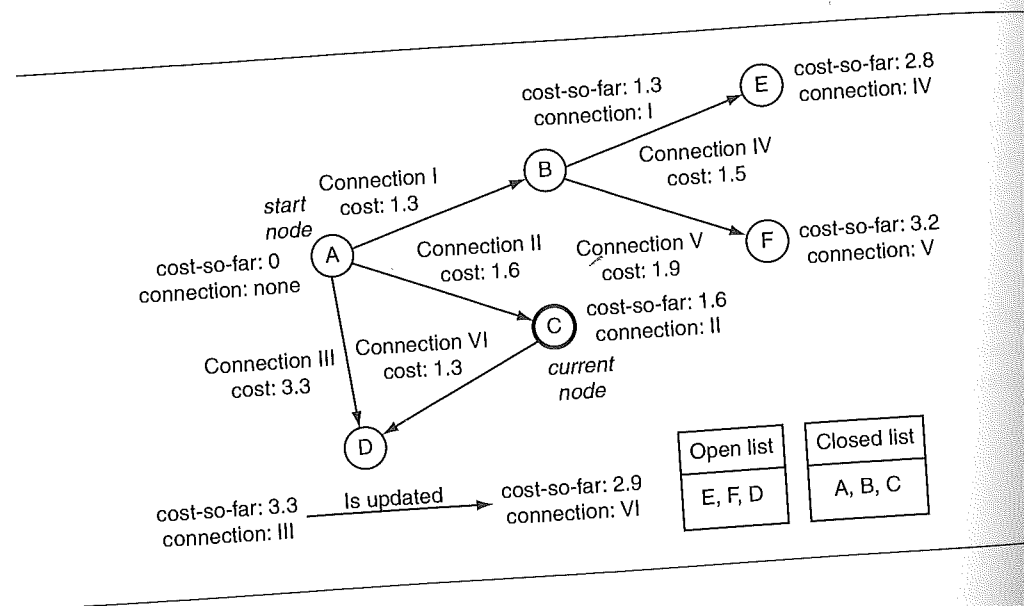


Figure 4.10 Open node update

Instead, we check if the route we've now found is better than the route that we've already found. Calculate the cost-so-far value as normal, and if it is higher than the recorded value (and it will be higher in almost all cases), then don't update the node at all and don't change what list it is on.

If the new cost-so-far value is smaller than the node's current cost-so-far, then update it with the better value, and set its connection record. The node should then be placed on the open list. If it was previously on the closed list, it should be removed from there.

Strictly speaking, Dijkstra will never find a better route to a closed node, so we could check if the node is closed first and not bother doing the cost-so-far check. A dedicated Dijkstra implementation would do this. We will see that the same is not true of the A* algorithm, however, and we will have to check for faster routes in both cases.

Figure 4.10 shows the updating of an open node in a graph. The new route, via node C, is faster, and so the record for node D is updated accordingly.

Terminating the Algorithm

The basic Dijkstra algorithm terminates when the open list is empty: it has considered every node in the graph that be reached from the start node, and they are all on the closed list.

For pathfinding, we are only interested in reaching the goal node, however, so we can stop earlier. The algorithm should terminate when the goal node is the smallest node on the open list.

Notice that this means we will have already reached the goal on a previous iteration, in order to move it onto the open list. Why not simply terminate the algorithm as soon as we've found the goal?

Consider Figure 4.10 again. If D is the goal node, then we'll first find it when we're processing node B. So if we stop here, we'll get the route A-B-D, which is not the shortest route. To make sure there can be no shorter routes, we have to wait until the goal has the smallest cost-so-far. At this point, and only then, we know that a route via any other unprocessed node (either open or unvisited) must be longer.

In practice, this rule is often broken. The first route found to the goal is very often the shortest, and even when there is a shorter route, it is usually only a tiny amount longer. For this reason, many developers implement their pathfinding algorithms to terminate as soon as the goal node is seen, rather than waiting for it to be selected from the open list.

Retrieving the Path

The final stage is to retrieve the path.

We do this by starting at the goal node and looking at the connection that was used to arrive there. We then go back and look at the start node of that connection and do the same. We continue this process, keeping track of the connections, until the original start node is reached. The list of connections is correct, but in the wrong order, so we reverse it and return the list as our solution.

Figure 4.11 shows a simple graph after the algorithm has run. The list of connections found by following the records back from the goal is reversed to give the complete path.

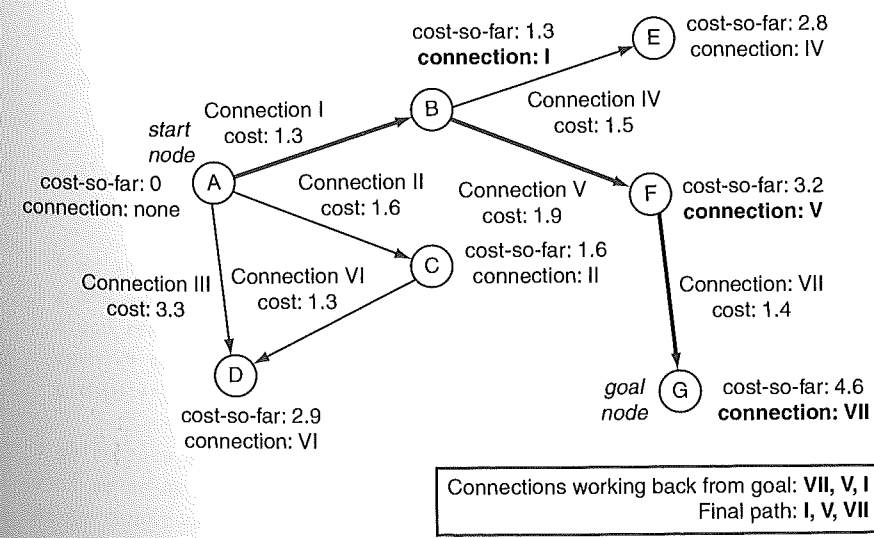


Figure 4.11 Following the connections to get a plan

4.2.3 PSEUDO-CODE

The Dijkstra pathfinder takes as input a graph (conforming to the interface given in the previous section), a start node, and an end node. It returns an array of connection objects that represent a path from the start node to the end node.

```

1 def pathfindDijkstra(graph, start, end):
2     # This structure is used to keep track of the
3     # information we need for each node
4     struct NodeRecord:
5         node
6         connection
7         costSoFar
8
9     # Initialize the record for the start node
10    startRecord = new NodeRecord()
11    startRecord.node = start
12    startRecord.connection = None
13    startRecord.costSoFar = 0
14
15    # Initialize the open and closed lists
16    open = PathfindingList()
17    open += startRecord
18    closed = PathfindingList()
19
20    # Iterate through processing each node
21    while length(open) > 0:
22
23        # Find the smallest element in the open list
24        current = open.smallestElement()
25
26        # If it is the goal node, then terminate
27        if current.node == goal: break
28
29        # Otherwise get its outgoing connections
30        connections = graph.getConnections(current)
31
32        # Loop through each connection in turn
33        for connection in connections:
34
35            # Get the cost estimate for the end node
36            endNode = connection.getToNode()
37

```

```

38    endNodeCost = current.costSoFar +
39                    connection.getCost()
40
41    # Skip if the node is closed
42    if closed.contains(endNode): continue
43
44    # .. or if it is open and we've found a worse
45    # route
46    else if open.contains(endNode):
47
48        # Here we find the record in the open list
49        # corresponding to the endNode.
50        endNodeRecord = open.find(endNode)
51
52        if endNodeRecord.cost <= endNodeCost:
53            continue
54
55        # Otherwise we know we've got an unvisited
56        # node, so make a record for it
57    else:
58        endNodeRecord = new NodeRecord()
59        endNodeRecord.node = endNode
60
61        # We're here if we need to update the node
62        # Update the cost and connection
63        endNodeRecord.cost = endNodeCost
64        endNodeRecord.connection = connection
65
66        # And add it to the open list
67        if not open.contains(endNode):
68            open += endNodeRecord
69
70        # We've finished looking at the connections for
71        # the current node, so add it to the closed list
72        # and remove it from the open list
73        open -= current
74        closed += current
75
76        # We're here if we've either found the goal, or
77        # if we've no more nodes to search, find which.
78        if current.node != goal:
79
80            # We've run out of nodes without finding the
81            # goal, so there's no solution

```



```

82     return None
83
84     else:
85
86         # Compile the list of connections in the path
87         path = []
88
89         # Work back along the path, accumulating
90         # connections
91         while current.node != start:
92             path += current.connection
93             current = current.connection.getFromNode()
94
95         # Reverse the path, and return it
96         return reverse(path)

```

Other Functions

The pathfinding list is a specialized data structure that acts very much like a regular list. It holds a set of `NodeRecord` structures and supports the following additional methods:

- The `smallestElement()` method returns the `NodeRecord` structure in the list with the lowest `costSoFar` value.
- The `contains(node)` method returns true only if the list contains a `NodeRecord` structure whose `node` member is equal to the given parameter.
- The `find(node)` method returns the `NodeRecord` structure from the list whose `node` member is equal to the given parameter.

In addition, we have used a function, `reverse(array)`, that returns a reversed copy of a normal array.

4.2.4 DATA STRUCTURES AND INTERFACES

There are three data structures used in the algorithm: the simple list used to accumulate the final path, the pathfinding list used to hold the open and closed lists, and the graph used to find connections from a node (and their costs).

Simple List

The simple list is not very performance critical, since it is only used at the end of the pathfinding process. It can be implemented as a basic linked list (a `std::list` in C++, for example) or even a resizable array (such as `std::vector` in C++).

Pathfinding List

The open and closed lists in the Dijkstra algorithm (and in A*) are critical data structures that directly affect the performance of the algorithm. Almost all optimization effort in pathfinding goes into their implementation. In particular, there are four operations on the list that are critical:

1. Adding an entry to the list (the `+=` operator);
2. Removing an entry from the list (the `-=` operator);
3. Finding the smallest element (the `smallestElement` method);
4. Finding an entry in the list corresponding to a particular node (the `contains` and `find` methods both do this).

Finding a suitable balance between these four operations is key to building a fast implementation. Unfortunately, the balance is not always identical from game to game.

Because the pathfinding list is most commonly used with A* for pathfinding, a number of its optimizations are specific to that algorithm. We will wait to examine it in more detail until we have looked at A*.

Graph

We have covered the interface presented by the graph in the first section of this chapter.

The `getConnections` method is called low down in the loop and is typically a critical performance element to get right. The most common implementation has a lookup table indexed by a node (where nodes are numbered as consecutive integers). The entry in the lookup table is an array of connection objects. Thus, the `getConnections` method needs to do minimal processing and is efficient.

Some methods of translating a game level into a pathfinding graph do not allow for this simple lookup approach and can therefore lead to much slower pathfinding. Such situations are described in more detail in Section 4.4 on world representation later in the chapter.

The `getNode` and `getCost` methods of the `Connection` class are even more performance critical. In an overwhelming majority of implementations, however, no processing is performed in these methods, and they simply return a stored value in each case. The `Connection` class might therefore look like the following:

```

1  class Connection:
2      cost
3      fromNode
4      toNode
5
6      def getCost(): return cost
7      def getFromNode(): return fromNode
8      def getNode(): return toNode

```

For this reason the Connection class is rarely a performance bottleneck. Of course, these values need to be calculated somewhere. This is usually done when the game level is converted into a graph and is an offline process independent of the pathfinder.

4.2.5 PERFORMANCE OF DIJKSTRA

The practical performance of Dijkstra in both memory and speed depends mostly on the performance of the operations in the pathfinding list data structure.

Ignoring the performance of the data structure for a moment, we can see the theoretical performance of the overall algorithm. The algorithm considers each node in the graph that is closer to the end node. We call this number n . For each of these nodes, it processes the inner loop once for each outgoing connection. We call the average number of outgoing connections per node m . So the algorithm itself is $O(nm)$ in execution speed. The total memory use depends on both the size of the open list and the size of the closed list. When the algorithm terminates there will be n elements in the closed list and no more than nm elements in the open list (in fact, there will typically be fewer than n elements in the open list). So the worst case memory use is $O(nm)$.

To include the data structure times, we note that both the list addition and the find operation (see the section on the pathfinding list data structure, above) are called nm times, while the extraction and `smallestElement` operations are called n times. If the order of the execution time for the addition or find operations is greater than $O(m)$, or if the extraction and `smallestElement` operations are greater than $O(1)$, then the actual execution performance will be worse than $O(nm)$.

In order to speed up the key operations, data structure implementations are often chosen that have worse than $O(nm)$ memory requirements.

When we look in more depth at the list implementations in the next section, we will consider their impact on performance characteristics.

If you look up Dijkstra in a computer science textbook, it may tell you that it is $O(n^2)$. In fact, this is exactly the result above. The worst conceivable performance occurs when the graph is so densely connected that $m = n$. In this case for games, however, there'll be a direct path to the goal anyway, so we can avoid Dijkstra altogether.

4.2.6 WEAKNESSES

The principle problem with Dijkstra is that it searches the entire graph indiscriminately for the shortest possible route. This is useful if we're trying to find the shortest path to every possible node (the problem that Dijkstra was designed for), but wasteful for point-to-point pathfinding.

We can visualize the way the algorithm works by showing the nodes currently on its open and closed lists at various stages through a typical run. This is shown in Figure 4.12.

In each case the boundary of the search is made up of nodes on the open list. This is because the nodes closer to the start (i.e., with lower distance values) have already been processed and placed on the closed list.

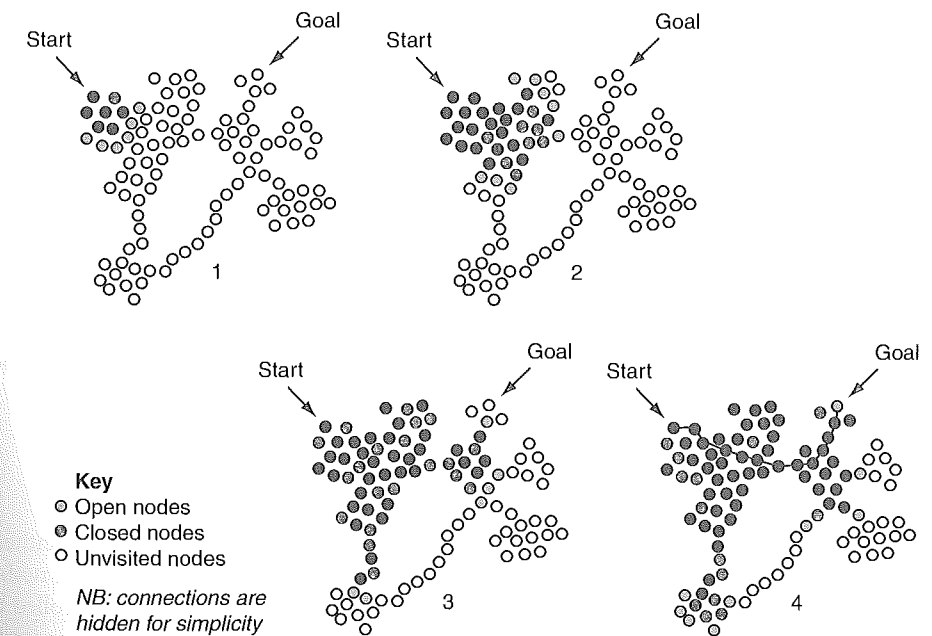


Figure 4.12 Dijkstra in steps

The final part of Figure 4.12 shows the state of the lists when the algorithm terminates. The line shows the best path that has been calculated. Notice that most of the level has still been explored, even well away from the path that is generated.

The number of nodes that were considered, but never made part of the final route, is called the *fill* of the algorithm. In general, you want to consider as few nodes as possible, because each takes time to process.

Sometimes Dijkstra will generate a search pattern with a relatively small amount of fill. This is the exception rather than the rule, however. In the vast majority of cases, Dijkstra suffers from a terrible amount of fill.

Algorithms with big fills, like Dijkstra, are inefficient for point-to-point pathfinding and are rarely used. This brings us to the star of pathfinding algorithms: A*. It can be thought of as a low-fill version of Dijkstra.

4.3 A*

Pathfinding in games is synonymous with the A* algorithm. A* is simple to implement, is very efficient, and has lots of scope for optimization. Every pathfinding system we've come across in the last 10 years has used some variation of A* as its key algorithm, and it has applications well

beyond pathfinding too. In Chapter 5, we will see how A* can be used to plan complex series of actions for characters.

Unlike the Dijkstra algorithm, A* is designed for point-to-point pathfinding and is not used to solve the shortest path problem in graph theory. It can neatly be extended to more complex cases, as we'll see later, but it always returns a single path from source to goal.

4.3.1 THE PROBLEM

The problem is identical to that solved by our Dijkstra pathfinding algorithm. Given a graph (a directed non-negative weighted graph) and two nodes in that graph (start and goal), we would like to generate a path such that the total path cost of that path is minimal among all possible paths from start to goal. Any minimal cost path will do, and the path should consist of a list of connections from the start node to the goal node.

4.3.2 THE ALGORITHM

Informally, the algorithm works in much the same way as Dijkstra does. Rather than always considering the open node with the lowest cost-so-far value, we choose the node that is most likely to lead to the shortest overall path. The notion of "most likely" is controlled by a heuristic. If the heuristic is accurate, then the algorithm will be efficient. If the heuristic is terrible, then it can perform even worse than Dijkstra.

In more detail, A* works in iterations. At each iteration it considers one node of the graph and follows its outgoing connections. The node (again called the "current node") is chosen using a selection algorithm similar to Dijkstra's, but with the significant difference of the heuristic, which we'll return to later.

Processing the Current Node

During an iteration, A* considers each outgoing connection from the current node. For each connection it finds the end node and stores the total cost of the path so far (the "cost-so-far") and the connection it arrived there from, just as before.

In addition, it also stores one more value: the estimate of the total cost for a path from the start node through this node and onto the goal (we'll call this value the estimated-total-cost). This estimate is the sum of two values: the cost-so-far and how far it is from the node to the goal. This estimate is generated by a separate piece of code and isn't part of the algorithm.

These estimates are called the "heuristic value" of the node, and it cannot be negative (since the costs in the graph are non-negative, it doesn't make sense to have a negative estimate). The generation of this heuristic value is a key concern in implementing the A* algorithm, and we'll return to it later in some depth.

Figure 4.13 shows the calculated values for some nodes in a graph. The nodes are labeled with their heuristic values, and the two calculated values (cost-so-far and estimated-total-cost) are shown for the nodes that the algorithm has considered.

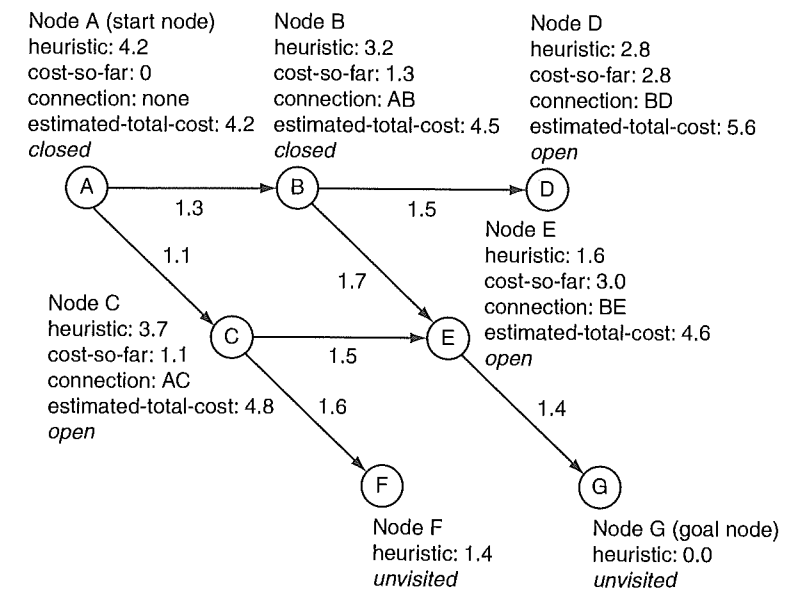


Figure 4.13 A* estimated-total-costs

The Node Lists

As before, the algorithm keeps a list of open nodes that have been visited but not processed and closed nodes that have been processed. Nodes are moved onto the open list as they are found at the end of connections. Nodes are moved onto the closed list as they are processed in their own iteration.

Unlike previously, the node from the open list with the smallest estimated-total-cost is selected at each iteration. This is almost always different from the node with the smallest cost-so-far.

This alteration allows the algorithm to examine nodes that are more promising first. If a node has a small estimated-total-cost, then it must have a relatively short cost-so-far and a relatively small estimated distance to go to reach the goal. If the estimates are accurate, then the nodes that are closer to the goal are considered first, narrowing the search into the most profitable area.

Calculating Cost-So-Far for Open and Closed Nodes

As before, we may arrive at an open or closed node during an iteration, and we will have to revise its recorded values.

We calculate the cost-so-far value as normal, and if the new value is lower than the existing value for the node, then we will need to update it. Notice that we do this comparison strictly on

the cost-so-far value (the only reliable value, since it doesn't contain any element of estimate), not the estimated-total-cost.

Unlike Dijkstra, the A* algorithm can find better routes to nodes that are already on the closed list. If a previous estimate was very optimistic, then a node may have been processed thinking it was the best choice when, in fact, it was not.

This causes a knock-on problem. If a dubious node has been processed and put on the closed list, then it means all its connections have been considered. It may be possible that a whole set of nodes have had their cost-so-far values based on the cost-so-far of the dubious node. Updating the value for the dubious node is not enough. All its connections will also have to be checked again to propagate the new value.

In the case of revising a node on the open list, this isn't necessary, since we know that connections from a node on the open list haven't been processed yet.

Fortunately, there is a simple way to force the algorithm to recalculate and propagate the new value. We can remove the node from the closed list and place it back on the open list. It will then wait until it is closed and have its connections reconsidered. Any nodes that rely on its value will also eventually be processed once more.

Figure 4.14 shows the same graph as the previous diagram, but two iterations later. It illustrates the updating of a closed node in a graph. The new route to E, via node C, is faster, and so the record for node E is updated accordingly, and it is placed on the open list. On the next iteration the value for node G is correspondingly revised.

So closed nodes that have their values revised are removed from the closed list and placed on the open list. Open nodes that have their values revised stay on the open list, as before.

Terminating the Algorithm

In many implementations, A* terminates when the goal node is the smallest node on the open list.

But as we have already seen, a node that has the smallest estimated-total-cost value (and will therefore be processed next iteration and put on the closed list) may later need its values revised. We can no longer guarantee, just because the node is the smallest on the open list, that we have the shortest route there. So terminating A* when the goal node is the smallest on the open list will not guarantee that the shortest route has been found.

It is natural, therefore, to ask whether we could run A* a little longer to generate a guaranteed optimal result. We can do this by requiring that the algorithm only terminates when the node in the open list with the smallest cost-so-far (not estimated-total-cost) has a cost-so-far value greater than the cost of the path we found to the goal. Then and only then can we guarantee that no future path will be found that forms a shortcut.

This is effectively the same termination condition we saw in Dijkstra, and it can be shown that imposing this condition will generate the same amount of fill as running the Dijkstra pathfinding algorithm. The nodes may be searched in a different order, and there may be slight differences in the set of nodes on the open list, but the approximate fill level will be the same. In other words, it robs A* of any performance advantage and makes it effectively worthless.

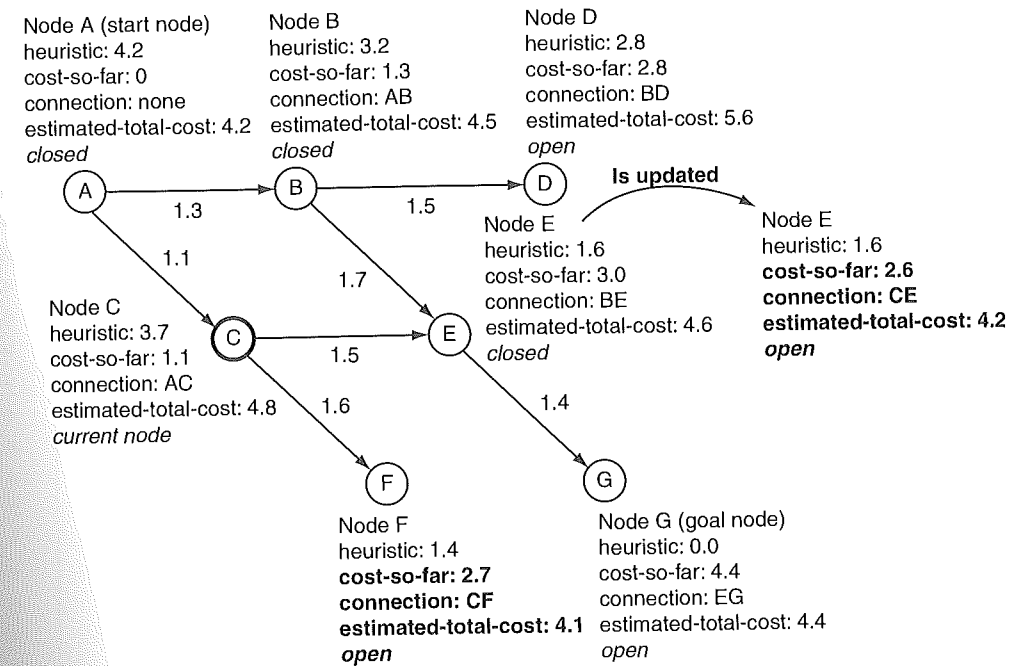


Figure 4.14 Closed node update

A* implementations completely rely on the fact that they can theoretically produce non-optimal results. Fortunately, this can be controlled using the heuristic function. Depending on the choice of heuristic function, we can guarantee optimal results, or we can deliberately allow sub-optimal results to give us faster execution. We'll return to the influence of the heuristic later in this section.

Because A* so often flirts with sub-optimal results, a large number of A* implementations instead terminate when the goal node is first visited without waiting for it to be the smallest on the open list. The performance advantage is not as great as doing the same thing in Dijkstra, but many developers feel that every little bit counts, especially as the algorithm won't necessarily be optimal in any case.

Retrieving the Path

We get the final path in exactly the same way as before: by starting at the goal and accumulating the connections as we move back to the start node. The connections are again reversed to form the correct path.

4.3.3 PSEUDO-CODE

Exactly as before, the pathfinder takes as input a graph (conforming to the interface given in the previous section), a start node, and an end node. It also requires an object that can generate estimates of the cost to reach the goal from any given node. In the code this object is "heuristic." It is described in more detail later in the data structures section.

The function returns an array of connection objects that represents a path from the start node to the end node.

```

1 def pathfindAStar(graph, start, end, heuristic):
2
3     # This structure is used to keep track of the
4     # information we need for each node
5     struct NodeRecord:
6         node
7         connection
8         costSoFar
9         estimatedTotalCost
10
11     # Initialize the record for the start node
12     startRecord = new NodeRecord()
13     startRecord.node = start
14     startRecord.connection = None
15     startRecord.costSoFar = 0
16     startRecord.estimatedTotalCost =
17         heuristic.estimate(start)
18
19     # Initialize the open and closed lists
20     open = PathfindingList()
21     open += startRecord
22     closed = PathfindingList()
23
24     # Iterate through processing each node
25     while length(open) > 0:
26
27         # Find the smallest element in the open list
28         # (using the estimatedTotalCost)
29         current = open.smallestElement()
30
31         # If it is the goal node, then terminate
32         if current.node == goal: break
33
34         # Otherwise get its outgoing connections

```

```

35     connections = graph.getConnections(current)
36
37     # Loop through each connection in turn
38     for connection in connections:
39
40         # Get the cost estimate for the end node
41         endNode = connection.getToNode()
42         endNodeCost = current.costSoFar +
43             connection.getCost()
44
45         # If the node is closed we may have to
46         # skip, or remove it from the closed list.
47         if closed.contains(endNode):
48
49             # Here we find the record in the closed list
50             # corresponding to the endNode.
51             endNodeRecord = closed.find(endNode)
52
53             # If we didn't find a shorter route, skip
54             if endNodeRecord.costSoFar <= endNodeCost:
55                 continue;
56
57             # Otherwise remove it from the closed list
58             closed -= endNodeRecord
59
60             # We can use the node's old cost values
61             # to calculate its heuristic without calling
62             # the possibly expensive heuristic function
63             endNodeHeuristic = endNodeRecord.estimatedTotalCost -
64                 endNodeRecord.costSoFar
65
66             # Skip if the node is open and we've not
67             # found a better route
68             else if open.contains(endNode):
69
70                 # Here we find the record in the open list
71                 # corresponding to the endNode.
72                 endNodeRecord = open.find(endNode)
73
74                 # If our route is no better, then skip
75                 if endNodeRecord.costSoFar <= endNodeCost:
76                     continue;
77
78                 # We can use the node's old cost values

```

```

79     # to calculate its heuristic without calling
80     # the possibly expensive heuristic function
81     endNodeHeuristic = endNodeRecord.cost -
82                       endNodeRecord.costSoFar
83
84     # Otherwise we know we've got an unvisited
85     # node, so make a record for it
86     else:
87         endNodeRecord = new NodeRecord()
88         endNodeRecord.node = endNode
89
90         # We'll need to calculate the heuristic value
91         # using the function, since we don't have an
92         # existing record to use
93         endNodeHeuristic = heuristic.estimate(endNode)
94
95         # We're here if we need to update the node
96         # Update the cost, estimate and connection
97         endNodeRecord.cost = endNodeCost
98         endNodeRecord.connection = connection
99         endNodeRecord.estimatedTotalCost =
100         endNodeCost + endNodeHeuristic
101
102         # And add it to the open list
103         if not open.contains(endNode):
104             open += endNodeRecord
105
106         # We've finished looking at the connections for
107         # the current node, so add it to the closed list
108         # and remove it from the open list
109         open -= current
110         closed += current
111
112         # We're here if we've either found the goal, or
113         # if we've no more nodes to search, find which.
114         if current.node != goal:
115
116             # We've run out of nodes without finding the
117             # goal, so there's no solution
118             return None
119
120         else:
121
122             # Compile the list of connections in the path

```

```

123     path = []
124
125     # Work back along the path, accumulating
126     # connections
127     while current.node != start:
128         path += current.connection
129         current = current.connection.getFromNode()
130
131     # Reverse the path, and return it
132     return reverse(path)

```

Changes from Dijkstra

The algorithm is almost identical to the Dijkstra algorithm. It adds an extra check to see if a closed node needs updating and removing from the closed list. It also adds two lines to calculate the estimated-total-cost of a node using the heuristic function and adds an extra field in the `NodeRecord` structure to hold this information.

A set of calculations can be used to derive the heuristic value from the cost values of an existing node. This is done simply to avoid calling the heuristic function any more than is necessary. If a node has already had its heuristic calculated, then that value will be reused when the node needs updating.

Other than these minor changes, the code is identical.

As for the supporting code, the `smallestElement` method of the pathfinding list data structure should now return the `NodeRecord` with the smallest estimated-total-cost value, not the smallest cost-so-far value as before. Otherwise, the same implementations can be used.

4.3.4 DATA STRUCTURES AND INTERFACES

The graph data structure and the simple path data structure used to accumulate the path are both identical to those used in the Dijkstra algorithm. The pathfinding list data structure has a `smallestElement` method that now considers estimated-total-cost rather than cost-so-far but is otherwise the same.

Finally, we have added a heuristic function that generates estimates of the distance from a given node to the goal.

Pathfinding List

Recall from the discussion on Dijkstra that the four component operations required on the pathfinding list are the following:

1. Adding an entry to the list (the `+=` operator);
2. Removing an entry from the list (the `-=` operator);

3. Finding the smallest element (the `smallestElement` method);
4. Finding an entry in the list corresponding to a particular node (the `contains` and `find` methods both do this).

Of these operations, numbers 3 and 4 are typically the most fruitful for optimization (although optimizing these often requires changes to numbers 1 and 2 in turn). We'll look at a particular optimization for number 4, which uses a non-list structure, later in this section.

A naive implementation of number 3, finding the smallest element in the list, involves looking at every node in the open list, every time through the algorithm, to find the lowest total path estimate.

There are lots of ways to speed this up, and all of them involve changing the way the list is structured so that the best node can be found quickly. This kind of specialized list data structure is usually called a *priority queue*. It minimizes the time it takes to find the best node.

In this book we won't cover each possible priority queue implementation in depth. Priority queues are a common data structure detailed in any good algorithms text.

Priority Queues

The simplest approach is to require that the open list be sorted. This means that we can get the best node immediately because it is the first one in the list.

But making sure the list is sorted takes time. We could sort it each time we need it, but this would take a very long time. A more efficient way is to make sure that when we add things to the open list, they are in the right place. Previously, we have appended new nodes to the list with no regard for order, a very fast process. Inserting the new node in its correct sorted position in the list takes longer.

This is a common trade-off when designing data structures: if you make it fast to add an item, it may be costly to get it back, and if you optimize retrieval, then adding may take time.

If the open list is already sorted, then adding a new item involves finding the correct insertion point in the list for the new item. In our implementation so far, we have used a linked list. To find the insertion point in a linked list we need to go through each item in the list until we find one with a higher total path estimate than ours. This is faster than searching for the best node, but still isn't too efficient.

If we use an array rather than a linked list, we can use a binary search to find the insertion point. This is faster, and for a very large list (and the open list is often huge) it provides a massive speed up.

Adding to a sorted list is faster than removing from an unsorted list. If we added nodes about as often as we removed them, then it would be better to have a sorted list. Unfortunately, A* adds many more nodes than it retrieves to the open list. It rarely removes nodes from the closed list at all.

Priority Heaps

Priority heaps are an array-based data structure which represents a tree of elements. Each item in the tree can have up to two children, both of which must have higher values.

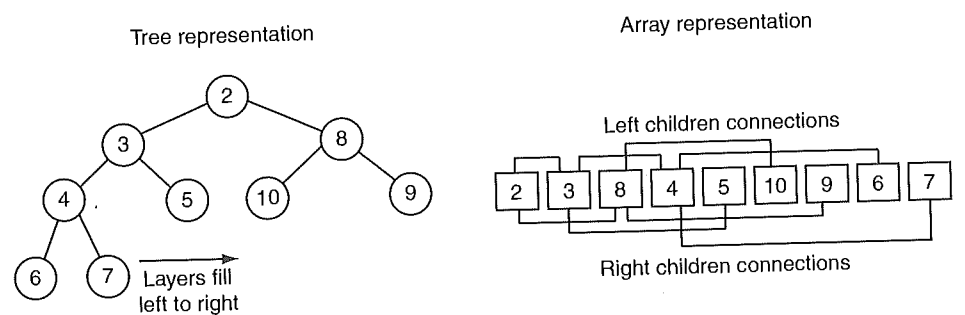


Figure 4.15 Priority heap

The tree is balanced, so that no branch is more than one level deeper than any other. In addition, it fills up each level from the left to the right. This is shown in Figure 4.15.

This structure is useful because it allows the tree to be mapped to a simple array in memory: the left and right children of a node are found in the array at position $2i$ and $2i + 1$, respectively, where i is the position of the parent node in the array. See Figure 4.15 for an example, where the tree connections are overlaid onto the array representation.

With this ultra-compact representation of the heap, the well-known sorting algorithm heap-sort can be applied, which takes advantage of the tree structure to keep nodes in order. Finding the smallest element takes constant time (it is always the first element: the head of the tree). Removing the smallest element, or adding any new element, takes $O(\log n)$, where n is the number of elements in the list.

The priority heap is a well-known data structure commonly used for scheduling problems and is the heart of an operating system's process manager.

Bucketed Priority Queues

Bucketed priority queues are more complex data structures that have partially sorted data. The partial sorting is designed to give a blend of performance across different operations, so adding items doesn't take too long and removing them is still fast.

The eponymous buckets are small lists that contain unsorted items within a specified range of values. The buckets themselves are sorted, but the contents of the buckets aren't.

To add to this kind of priority queue, you search through the buckets to find the one your node fits in. You then add it to the start of the bucket's list. This is illustrated in Figure 4.16.

The buckets can be arranged in a simple list, as a priority queue themselves, or as a fixed array. In the latter case, the range of possible values must be fairly small (total path costs often lie in a reasonably small range). Then the buckets can be arranged with fixed intervals: the first bucket might contain values from 0 to 10, the second from 10 to 20, and so on. In this case the data structure doesn't need to search for the correct bucket. It can go directly there, speeding up node adding even more.

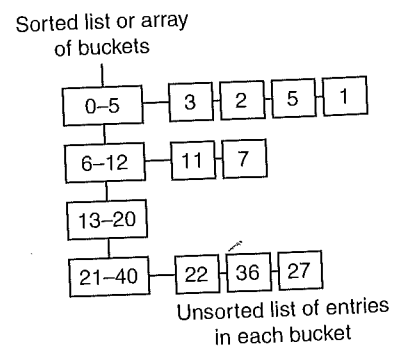


Figure 4.16 Bucketed priority queues

To find the node with the lowest score, you go to the first non-empty bucket and search its contents for the best node.

By changing the number of buckets, you can get just the right blend of adding and removal time. Tweaking the parameters is time consuming, however, and is rarely needed. For very large graphs, such as those representing levels in massively multi-player online games, the speed up can be worth the programming effort. In most cases it is not.

There are still more complex implementations, such as “multi-level buckets,” which have sorted lists of buckets containing lists of buckets containing unsorted items (and so on). We built a pathfinding system that used a multi-level bucket list, but it was more an act of hubris than a programming necessity, and we wouldn’t do it again!

Implementations

In our experience there is little to choose from between priority heaps and bucketed queues in many applications. We’ve built production implementations using both approaches. For very large pathfinding problems (with millions of nodes in the graph), bucketed priority queues can be written that are kinder to the processor’s memory cache and are therefore much faster. For indoor levels with a few thousand or tens of thousands of nodes, the simplicity of a priority heap is often sufficient.

In the source code on the website the A* implementation uses a simple priority queue implementation for its pathfinding lists, for simplicity’s sake.

Heuristic Function

The heuristic is often talked about as a function, and it can be implemented as a function. Throughout this book, we’ve preferred to show it in pseudo-code as an object. The heuristic object we used in the algorithm has a simple interface:

```

1 class Heuristic:
2     # Generates an estimated cost to reach the goal
3     # from the given node
4     def estimate(node)

```

A Heuristic for Any Goal

Because it is inconvenient to produce a different heuristic function for each possible goal in the world, the heuristic is often parameterized by the goal node. In that way a general heuristic implementation can be written that estimates the distance between any two nodes in the graph. The interface might look something like:

```

1 class Heuristic:
2
3     # Stores the goal node that this heuristic is
4     # estimating for
5     goalNode
6
7     # Constructor, takes a goal node for estimating.
8     def Heuristic(goal): goalNode = goal
9
10    # Generates an estimated cost to reach the
11    # stored goal from the given node
12    def estimate(node)

```

which can then be used to call the pathfinder in code such as

```

1 pathfindAStar(graph, start, end, new Heuristic(end))

```

Heuristic Speed

The heuristic is called at the lowest point in the loop. Because it is making an estimate, it might involve some algorithmic process. If the process is complex, the time spent evaluating heuristics might quickly dominate the pathfinding algorithm.

While some situations may allow you to build a lookup table of heuristic values, in most cases the number of combinations is huge so this isn’t practical.

It is essential that you run a profiler on the pathfinding system and look for ways to optimize the heuristic. We’ve seen situations where developers tried to squeeze extra speed from the pathfinding algorithm when over 80% of the execution time was spent evaluating heuristics.



LIBRARY

4.3.5 IMPLEMENTATION NOTES

The design of the A* algorithm we've looked at so far is the most general. It can work with any kind of cost value, with any kind of data type for nodes, and with graphs that have a huge range of sizes.

This generality comes at a price. There are better implementations of A* for most game pathfinding tasks. In particular, if we can assume that there is only a relatively small number (up to a 100,000, say, to fit in around 2Mb of memory) of nodes in the graph, and that these nodes can be numbered using sequential integers, then we can speed up our implementation significantly.

We call this node array A* (although you should be aware that we've made this name up; strictly, the algorithm is still just A*), and it is described in detail below.

Depending on the structure of the cost values returned and the assumptions that can be made about the graph, even more efficient implementations can be created. Most of these are outside the scope of this book (we could easily fill the whole book with just pathfinding variations), but the most important are given in a brief introduction at the end of this chapter.

The general A* implementation is still useful, however. In some cases you may need a variable number of nodes (if your game's level is being paged into memory in sections, for example), or there just isn't enough memory available for more complex implementations. We've used the general A* implementation on several occasions where more efficient implementations were not suitable.

4.3.6 ALGORITHM PERFORMANCE

Once again, the biggest factor in determining the performance of A* is the performance of its key data structures: the pathfinding list, the graph, and the heuristic.

Once again, ignoring these, we can look simply at the algorithm (this is equivalent to assuming that all data structure operations take constant time).

The number of iterations that A* performs is given by the number of nodes whose total estimated-path-cost is less than that of the goal. We'll call this number l , different from n in the performance analysis of Dijkstra. In general, l should be less than n . The inner loop of A* has the same complexity as Dijkstra, so the total speed of the algorithm is $O(lm)$, where m is the average number of outgoing connections from each node, as before. Similarly for memory usage, A* ends with $O(lm)$ entries in its open list, which is the peak memory usage for the algorithm.

In addition to Dijkstra's performance concerns of the pathfinding list and the graph, we add the heuristic function. The heuristic function is called very low in the loop, in the order of $O(lm)$ times. Often, the heuristic function requires some processing and can dominate the execution load of the algorithm. It is rare, however, for its implementation to directly depend on the size of the pathfinding problem. Although it may be time-consuming, the heuristic will most commonly have $O(1)$ execution time and memory and so will not have an effect on the order of the performance of the algorithm. This is an example of when the algorithm's order doesn't necessarily tell you very much about the real performance of the code.

4.3.7 NODE ARRAY A*

Node array A* is our name for a common implementation of the A* algorithm that is faster in many cases than the general A*. In the implementation we looked at so far, data are held for each node in the open or closed lists, and these data are held as a NodeRecord instance. Records are created when a node is first considered and then moved between the open and closed lists, as required.

There is a key step in the algorithm where the lists are searched for a node record corresponding to a particular node.

Keeping a Node Array

We can make a trade-off by increasing memory use to improve execution speed. To do this, we create an array of all the node records for every node in the whole graph before the algorithm begins. This node array will include records for nodes that will never be considered (hence the waste of memory), as well as for those that would have been created anyway.

If nodes are numbered using sequential integers, we don't need to search for a node in the two lists at all. We can simply use the node number to look up its record in the array (this is the logic of using node integers that we mentioned at the start of the chapter).

Checking if a Node Is in Open or Closed

We need to find the node data in order to check if we've found a better route to a node or if we need to add the node to one of the two lists.

Our original algorithm checked through each list, open and closed, to see if the node was already there. This is a very slow process, especially if there are many nodes in each list. It would be useful if we could look at a node and immediately discover what list, if any, it was in.

To find out which list a node is in, we add a new value to the node record. This value tells us which of the three categories the node is in: unvisited, open, or closed. This makes the search step very fast indeed (in fact, there is no search, and we can go straight to the information we need).

The new NodeRecord structure looks like the following:

```

1 # This structure is used to keep track of the
2 # information we need for each node
3 struct NodeRecord:
4     node
5     connection
6     costSoFar
7     estimatedTotalCost
8     category

```

where the category member is OPEN, CLOSED, or UNVISITED.

The Closed List Is Irrelevant

Because we've created all the nodes in advance, and they are located in an array, we no longer need to keep a closed list at all. The only time the closed list is used is to check if a node is contained within it and, if so, to retrieve the node record. Because we have the node records immediately available, we can find the record. With the record, we can look at the category value and see if it has been closed.

The Open List Implementation

We can't get rid of the open list in the same way because we still need to be able to retrieve the element with the lowest score. We can use the array for times when we need to retrieve a node record, from either open or closed lists, but we'll need a separate data structure to hold the priority queue of nodes.

Because we no longer need to hold a complete node record in the priority queue, it can be simplified. Often, the priority queue simply needs to contain the node numbers, whose records can be immediately looked up from the node array.

Alternatively, the priority queue can be intertwined with the node array records by making the node records part of a linked list:

```

1 # This structure is used to keep track of the
2 # information we need for each node
3 struct NodeRecord:
4     node
5     connection
6     costSoFar
7     estimatedTotalCost
8     category
9     nextRecordInList

```

Although the array will not change order, each element of the array has a link to the next record in a linked list. The sequence of nodes in this linked list jumps around the array and can be used as a priority queue to retrieve the best node on the open list.

Although we've seen implementations that add other elements to the record to support full bucketed priority queues, our experience is that this general approach leads to wasted memory (most nodes aren't in the list, after all), unnecessary code complexity (the code to maintain the priority queue can look very ugly), and cache problems (jumping around memory should be avoided when possible).

The node array pathfinding implementation on the website uses the separate priority queue approach. We'd recommend you do the same, unless you have a good reason to do otherwise.



LIBRARY

A Variation for Large Graphs

Creating all the nodes in advance is a waste of space if you aren't going to consider most of them. For small graphs on a PC, the memory waste is often worth it for the speed up. For large graphs, or for consoles with limited memory, it can be problematic.

In C, or other languages with pointers, we can blend the two approaches to create an array of pointers to node records, rather than an array of records themselves, setting all the pointers to NULL initially.

In the A* algorithm, we create the nodes when they are needed, as before, and set the appropriate pointer in the array. When we come to find what list a node is in, we can see if it has been created by checking if its pointer is NULL (if it is, then it hasn't been created and, by deduction, must be unvisited), if it is there, and if it is in either the closed or open list.

This approach requires less memory than allocating all the nodes in advance, but may still take up too much memory for very large graphs.

4.3.8 CHOOSING A HEURISTIC

The more accurate the heuristic, the less fill A* will experience, and the faster it will run. If you can get a perfect heuristic (one that always returns the exact minimum path distance between two nodes), A* will go straight to the correct answer: the algorithm becomes $O(p)$, where p is the number of steps in the path.

Unfortunately, to work out the exact distance between two nodes, you typically have to find the shortest route between them. This would mean solving the pathfinding problem—which is what we're trying to do in the first place! In only a few cases will a practical heuristic be accurate.

For non-perfect heuristics, A* behaves slightly differently depending on whether the heuristic is too low or too high.

Underestimating Heuristics

If the heuristic is too low, so that it underestimates the actual path length, A* takes longer to run. The estimated-total-cost will be biased toward the cost-so-far (because the heuristic value is smaller than reality). So A* will prefer to examine nodes closer to the start node, rather than those closer to the goal. This will increase the time it takes to find the route through to the goal.

If the heuristic underestimates in all possible cases, then the result that A* produces will be the best path possible. It will be the exact same path that the Dijkstra algorithm would generate. This avoids the problem we discussed earlier with sub-optimal paths.

If the heuristic ever overestimates, however, this guarantee is lost.

In applications where accuracy is more important than performance, it is important to ensure that the heuristic is underestimating. When you read articles about path planning in commercial and academic problems, accuracy is often very important, and so underestimating heuristics abound. This bias in the literature to underestimating heuristics often influences game developers.

In practice, try to resist dismissing overestimating heuristics outright. A game isn't about optimum accuracy; it's about believability.

Overestimating Heuristics

If the heuristic is too high, so that it overestimates the actual path length, A* may not return the best path. A* will tend to generate a path with fewer nodes in it, even if the connections between nodes are more costly.

The estimated-total-cost value will be biased toward the heuristic. The A* algorithm will pay proportionally less attention to the cost-so-far and will tend to favor nodes that have less distance to go. This will move the focus of the search toward the goal faster, but with the prospect of missing the best routes to get there.

This means that the total length of the path may be greater than that of the best path. Fortunately, it doesn't mean you'll suddenly get very poor paths. It can be shown that if the heuristic overestimates by at most x (i.e., x is the greatest overestimate for any node in the graph), then the final path will be no more than x too long.

An overestimating heuristic is sometimes called an "inadmissible heuristic." This doesn't mean you can't use it; it refers to the fact that the A* algorithm no longer returns the shortest path.

Overestimates can make A* faster if they are almost perfect, because they home in on the goal more quickly. If they are only slightly overestimating, they will tend to produce paths that are often identical to the best path, so the quality of results is not a major issue.

But the margin for error is small. As a heuristic overestimates more, it rapidly makes A* perform worse. Unless your heuristic is consistently close to perfect, it can be more efficient to underestimate, and you get the added advantage of getting the correct answer.

Let's look at some common heuristics used in games.

Euclidean Distance

Imagine that the cost values in our pathfinding problem refer to distances in the game level. The connection cost is generated by the distance between the representative points of two regions. This is a common case, especially in first-person shooter (FPS) games where each route through the level is equally possible for each character.

In this case (and in others that are variations on the pure distance approach), a common heuristic is Euclidean distance. It is guaranteed to be underestimating.

Euclidean distance is "as the crow flies" distance. It is measured directly between two points in space, through walls and obstructions.

Figure 4.17 shows Euclidean distances measured in an indoor level. The cost of a connection between two nodes is given by the distance between the representative points of each region. The estimate is given by the distance to the representative point of the goal node, even if there is no direct connection.

Euclidean distance is always either accurate or an underestimate. Traveling around walls or obstructions can only add extra distance. If there are no such obstructions, then the heuristic is accurate. Otherwise, it is an underestimate.

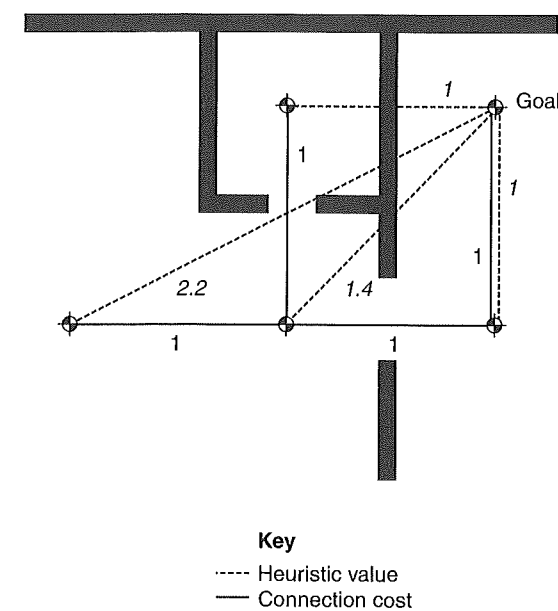


Figure 4.17 Euclidean distance heuristic

In outdoor settings, with few constraints on movement, Euclidean distance can be very accurate and provide fast pathfinding. In indoor environments, such as that shown in Figure 4.17, it can be a dramatic underestimate, causing less than optimal pathfinding.

Figure 4.18 shows the fill visualized for a pathfinding task through both tile-based indoor and outdoor levels. With the Euclidean distance heuristic, the fill for the indoor level is dramatic, and performance is poor. The outdoor level has minimal fill, and performance is good.

Cluster Heuristic

The cluster heuristic works by grouping nodes together in clusters. The nodes in a cluster represent some region of the level that is highly interconnected. Clustering can be done automatically using graph clustering algorithms that are beyond the scope of this book. Often, clustering is manual, however, or a by-product of the level design (portal-based game engines lend themselves well to having clusters for each room).

A lookup table is then prepared that gives the smallest path length between each pair of clusters. This is an offline processing step that requires running a lot of pathfinding trials between all pairs of clusters and accumulating their results. A sufficiently small set of clusters is selected so that this can be done in a reasonable time frame and stored in a reasonable amount of memory.

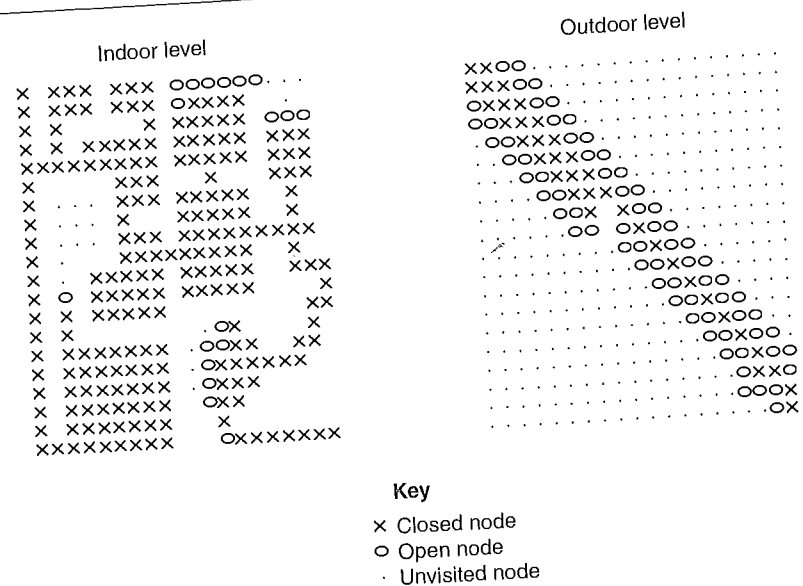


Figure 4.18 Euclidean distance fill characteristics

When the heuristic is called in the game, if the start and goal nodes are in the same cluster, then Euclidean distance (or some other fallback) is used to provide a result. Otherwise, the estimate is looked up in the table. This is shown in Figure 4.19 for a graph where each connection has the same cost in both directions.

The cluster heuristic often dramatically improves pathfinding performance in indoor areas over Euclidean distance, because it takes into account the convoluted routes that link seemingly nearby locations (the distance through a wall may be tiny, but the route to get between the rooms may involve lots of corridors and intermediate areas).

It has one caveat, however. Because all nodes in a cluster are given the same heuristic value, the A* algorithm cannot easily find the best route through a cluster. Visualized in terms of fill, a cluster will tend to be almost completely filled before the algorithm moves on to the next cluster.

If cluster sizes are small, then this is not a problem, and the accuracy of the heuristic can be excellent. On the other hand, the lookup table will be large (and the pre-processing time will be huge).

If cluster sizes are too large, then there will be marginal performance gain, and a simpler heuristic would be a better choice.

We've seen various modifications to the cluster heuristic to provide better estimates within a cluster, including some that include several Euclidean distance calculations for each estimate. There are opportunities for performance gain here, but as yet there are no accepted techniques for reliable improvement. It seems to be a case of experimenting in the context of your game's particular level design.

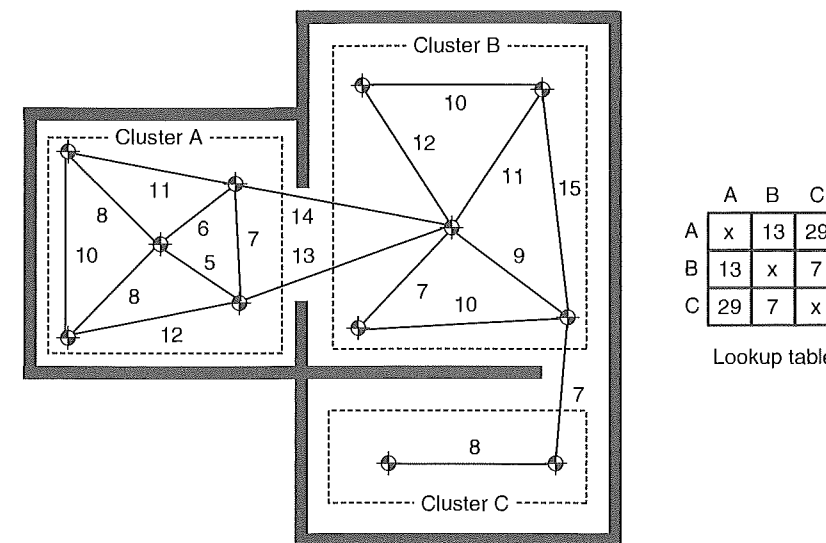


Figure 4.19 The cluster heuristic

Clustering is intimately related to hierarchical pathfinding, explained in Section 4.6, which also clusters sets of locations together. Some of the calculations we'll meet there for distance between clusters can be adapted to calculate the heuristics between clusters.

Even without such optimizations, the cluster heuristic is worth trying for labyrinthine indoor levels.

Fill Patterns in A*

Figure 4.20 shows the fill patterns of a tile-based indoor level using A* with different heuristics.

The first example uses a cluster heuristic tailored specifically to this level. The second example used a Euclidean distance, and the final example has a zero heuristic which always returns 0 (the most dramatic underestimate possible). The fill increases in each example; the cluster heuristic has very little fill, whereas the zero heuristic fills most of the level.

This is a good example of the knowledge vs. search trade-off we looked at in Chapter 2.

If the heuristic is more complex and more tailored to the specifics of the game level, then the A* algorithm needs to search less. It provides a good deal of knowledge about the problem. The ultimate extension of this is a heuristic with ultimate knowledge: completely accurate estimates. As we have seen, this would produce optimum A* performance with no search.

On the other hand, the Euclidean distance provides a little knowledge. It knows that the cost of moving between two points depends on their distance apart. This little bit of knowledge goes a long way, but still requires more searching than the perfect heuristic.

The zero heuristic has no knowledge, and it requires lots of search.

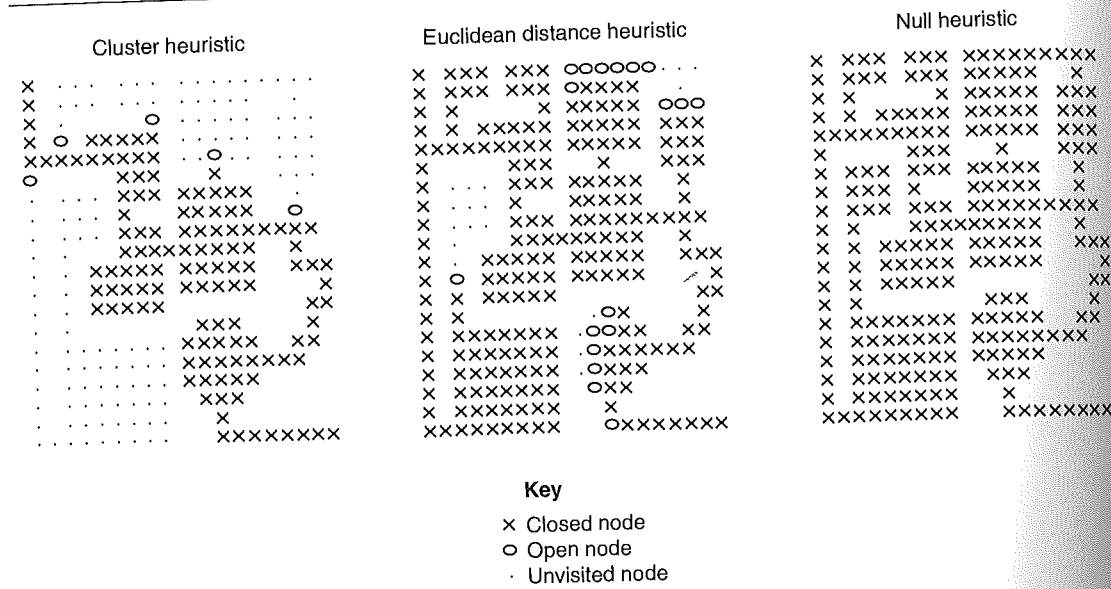


Figure 4.20 Fill patterns indoors

In our indoor example, where there are large obstructions, the Euclidean distance is not the best indicator of the actual distance. In outdoor maps, it is far more accurate. Figure 4.21 shows the zero and Euclidean heuristics applied to an outdoor map, where there are fewer obstructions. Now the Euclidean heuristic is more accurate, and the fill is correspondingly lower.

In this case Euclidean distance is a very good heuristic, and we have no need to try to produce a better one. In fact, cluster heuristics don't tend to improve performance (and can dramatically reduce it) in open outdoor levels.

Quality of Heuristics

Producing a heuristic is far more of an art than a science. Its significance is massively underestimated by AI developers. In our experience, many developers drop in a simple Euclidean distance heuristic without thought and hope for the best.

The only surefire way to get a decent heuristic is to visualize the fill of your algorithm. This can be in-game or using output statistics that you can later examine. We've found to our cost that tweaks to the heuristic we thought would be beneficial have often produced inferior results.

There has been some research done into automatically generating heuristics based on examining the structure of the graph and its connections. This may lead in time to automated heuristic algorithms that can produce better than Euclidean performance and may support graphs with non-distance-based costs. It is an interesting line of attack, but the results have yet to prove compelling.

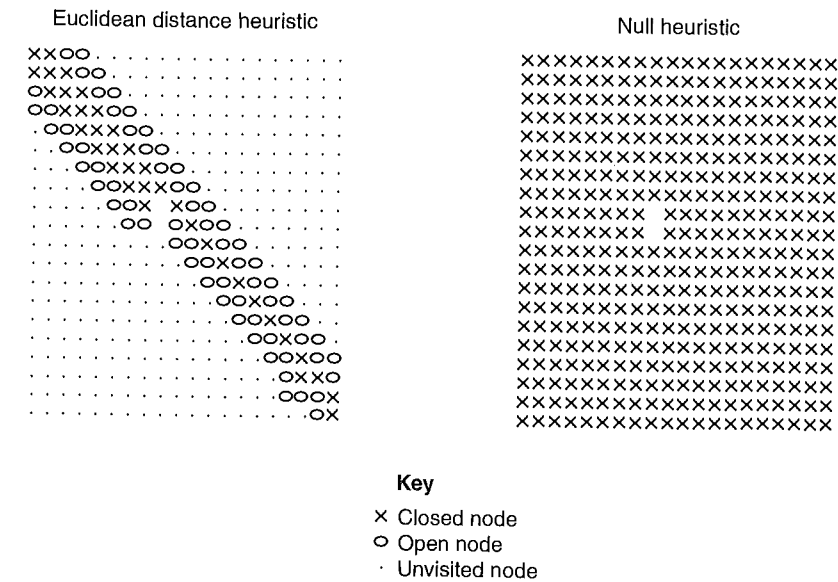


Figure 4.21 Fill patterns outdoors

Most developers aim for heuristics that are close, but err on the side of underestimating. The simplest, and most common, heuristic is Euclidean distance, and it will continue to be so for some time.

Dijkstra Is A*

It is worth noticing that the Dijkstra algorithm is a subset of the A* algorithm. In A* we calculate the estimated-total-cost of a node by adding the heuristic value to the cost-so-far. A* then chooses a node to process based on this value.

If the heuristic always returns 0, then the estimated-total-cost will always be equal to the cost-so-far. When A* chooses the node with the smallest estimated-total-cost, it is choosing the node with the smallest cost-so-far. This is identical to Dijkstra. A* with a zero heuristic is the pathfinding version of Dijkstra.

4.4 WORLD REPRESENTATIONS

So far we've assumed that pathfinding takes place on a graph made up of nodes and connections with costs. This is the world that the pathfinding algorithm knows about, but games aren't made up of nodes and connections.

To squeeze your game level into the pathfinder you need to do some translation—from the geometry of the map and the movement capabilities of your characters to the nodes and connections of the graph and the cost function that values them.

For each pathfinding world representation, we will divide the game level into linked regions that correspond to nodes and connections. The different ways this can be achieved are called *division schemes*. Each division scheme has three important properties we'll consider in turn: quantization/localization, generation, and validity.

You might also be interested in Chapter 11, Tools and Content Creation, which looks at how the pathfinding data are created by the level designer or by an automatic process. In a complete game, the choice of world representation will have as much to do with your toolchain as technical implementation issues.

Quantization and Localization

Because the pathfinding graph will be simpler than the actual game level, some mechanism is needed to convert locations in the game into nodes in the graph. When a character decides it wants to reach a switch, for example, it needs to be able to convert its own position and the position of the switch into graph nodes. This process is called *quantization*.

Similarly, if a character is moving along a path generated by the pathfinder, it needs to convert nodes in the plan back into game world locations so that it can move correctly. This is called *localization*.

Generation

There are many ways of dividing up a continuous space into regions and connections for pathfinding. There are a handful of standard methods used regularly. Each works either manually (the division being done by hand) or algorithmically.

Ideally, of course, we'd like to use techniques that can be run automatically. On the other hand, manual techniques often give the best results, as they can be tuned for each particular game level.

The most common division scheme used for manual techniques is the Dirichlet domain. The most common algorithmic methods are tile graphs, points of visibility, and navigation meshes. Of these, navigation meshes and points of visibility are often augmented so that they automatically generate graphs with some user supervision.

Validity

If a plan tells a character to move along a connection from node A to node B, then the character should be able to carry out that movement. This means that wherever the character is in node A, it should be able to get to any point in node B. If the quantization regions around A and B don't allow this, then the pathfinder may have created a useless plan.

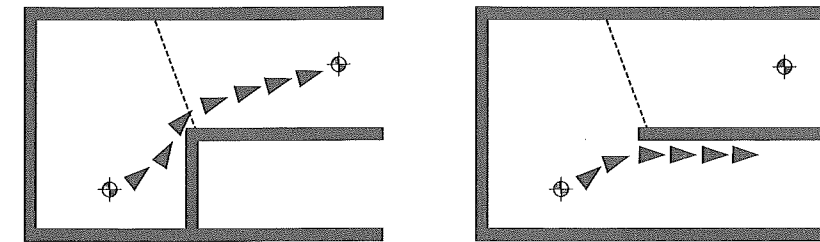


Figure 4.22 Two poor quantizations show that a path may not be viable

A division scheme is valid if all points in two connected regions can be reached from each other. In practice, most division schemes don't enforce validity. There can be different levels of validity, as Figure 4.22 demonstrates.

In the first part of the figure, the issue isn't too bad. An "avoid walls" algorithm (see Chapter 3) would easily cope with the problem. In the second figure with the same algorithm, it is terminal. Using a division scheme that gave the second graph would not be sensible. Using the first scheme will cause fewer problems. Unfortunately, the dividing line is difficult to predict, and an easily handled invalidity is only a small change away from being pathological.

It is important to understand the validity properties of graphs created by each division scheme; at the very least it has a major impact on the types of character movement algorithm that can be used.

So, let's look at the major division schemes used in games.

4.4.1 TILE GRAPHS

Tile-based levels, in the form of two-dimensional (2D) isometric graphics, have almost disappeared from mainstream games. The tile is far from dead, however. Although strictly not made up of tiles, a large number of games use grids in which they place their three-dimensional (3D) models. Underlying the graphics is still a regular grid.

This grid can be simply turned into a tile-based graph. Many real-time strategy (RTS) games still use tile-based graphs extensively, and many outdoor games use graphs based on height and terrain data.

Tile-based levels split the whole world into regular, usually square, regions (although hexagonal regions are occasionally seen in turn-based war simulation games).

Division Scheme

Nodes in the pathfinder's graph represent tiles in the game world. Each tile in the game world normally has an obvious set of neighbors (the eight surrounding tiles in a rectangular grid, for example). The connections between nodes connect to their immediate neighbors.

Quantization and Localization

We can determine which tile any point in the world is within, and this is often a fast process. In the case of a square grid, we can simply use a character's x and z coordinates to determine the square it is contained in. For example,

```
1 tileX = floor(x / tileSize)
2 tileZ = floor(z / tileSize)
```

where `floor()` is a function that returns the highest valued integer less than or equal to its argument, and `tileX` and `tileZ` identify the tile within the regular grid of tiles.

Similarly, for localization we can use a representative point in the tile (often the center of the tile) to convert a node back into a game location.

Generation

Tile-based graphs are generated automatically. In fact, because they are so regular (always having the same possible connections and being simple to quantize), they can be generated at runtime. An implementation of a tile-based graph doesn't need to store the connections for each node in advance. It can generate them as they are requested by the pathfinder.

Most games allow tiles to be blocked. In this case the graph will not return connections to blocked tiles, and the pathfinder will not try to move through them.

For tile-based grids representing outdoor height fields (a rectangular grid of height values), the costs often depend on gradient. The height field data are used to calculate a connection cost based both on distance and on gradient. Each sample in the height field represents the center point of a tile in the graph, and costs can be calculated based on distance and the change in elevation between the two points. In this way it will cost less to go downhill than uphill.

Validity

In many games that use tile-based layouts, a tile will be either completely blocked or completely empty. In this case, if the only tiles that are connected are empty, then the graph will be guaranteed to be valid.

When a graph node is only partially blocked, then the graph may not be valid, depending on the shape of the blockage. Figure 4.23 shows two cases: one in which a partial blockage does not make the graph invalid, and another in which it does.

Usefulness

While tile-based levels are one of the easiest to convert to a graph representation, there are often a vast number of tiles in the game. A small RTS level can have many hundreds of thousands of tiles. This means that the pathfinder has to work hard to plan sensible paths.

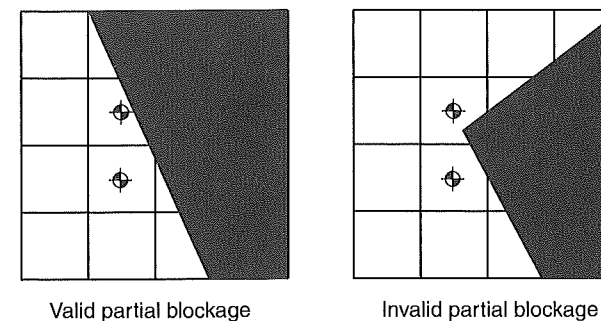


Figure 4.23 Tile-based graph with partially blocked validity

When the plans returned by the pathfinder are drawn on the graph (using localization for each node in the plan), they can appear blocky and irregular. Characters following the plan will look strange. This is illustrated in Figure 4.24.

While this is a problem with all division schemes, it is most noticeable for tile-based graphs (see Section 4.4.7 on path smoothing for an approach to solving this problem).

4.4.2 DIRICHLET DOMAINS

A Dirichlet domain, also referred to as a Voronoi polygon in two dimensions, is a region around one of a finite set of source points whose interior consists of everywhere that is closer to that source point than any other.

Division Scheme

Pathfinding nodes have an associated point in space called the *characteristic point*, and the quantization takes place by mapping all locations in the point's Dirichlet domain to the node. To determine the node for a location in the game, we find the characteristic point that is closest.

The set of characteristic points is usually specified by a level designer as part of the level data.

You can think of Dirichlet domains as being cones originating from the source point. If you view them from the top, as in Figure 4.25, the area of each cone that you see is the area that "belongs" to that source point. This is often a useful visualization for troubleshooting.

The basic idea has been extended to use different falloff functions for each node, so some nodes have a larger "pull" than others in the quantization step. This is sometimes called a *weighted Dirichlet domain*: each point has an associated weight value that controls the size of its region. Changing the weight is equivalent to changing the slope on the cone; squatter cones end up with larger regions. But care needs to be taken. Once you change the slope, you can get strange effects.

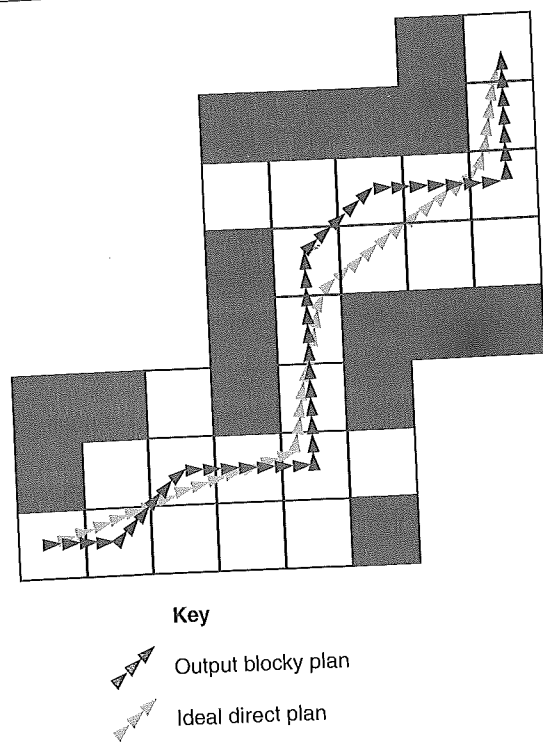


Figure 4.24 Tile-based plan is blocky

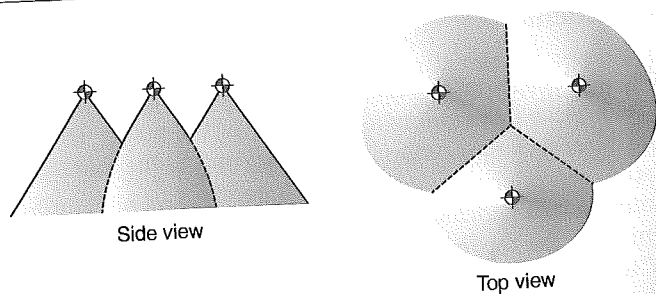


Figure 4.25 Dirichlet domains as cones

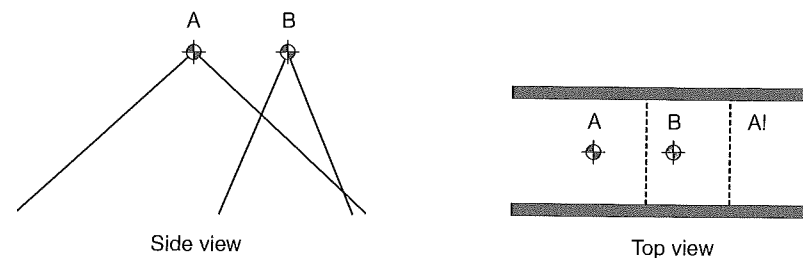


Figure 4.26 Problem domains with variable falloff

Figure 4.26 shows the Dirichlet domains in a passageway. You can see that the end of the passageway belongs to the wrong source point: the fat cone has peaked back out. This can make it difficult to debug pathfinding problems.

If you are manually assigning weighted Dirichlet domains, it's a good idea to have them displayed to check for overlapping problems.

Connections are placed between bordering domains. The pattern of connections can be found using a mathematical structure that has deep connections to Voronoi diagrams, called a *Delaunay triangulation*. The edges of the Delaunay triangulation are the connections in the graph, and the vertices are the characteristic points of the domains. Creating a Delaunay triangulation of a set of points is beyond the scope of this book. There are many websites dedicated to the algorithms for constructing Delaunay triangulations.

Most developers don't bother with a mathematically correct algorithm, however. They either make the artist specify connections as part of their level design, or they ray cast between points to check for connections (see the points of visibility method below). Even if you use the Delaunay triangulation method, you will need to check that domains that touch can actually be moved between, as there might be a wall in the way, for example.

Quantization and Localization

Positions are quantized by finding the characteristic point that is closest.

Searching through all points to find the closest is a time-consuming process (an $O(n)$ process, where n is the number of domains). Typically, we will use some kind of spatial partitioning algorithm (quad-tree, octree, binary space partition, or multi-resolution map) to allow us to consider only those points that are nearby.

The localization of a node is given by the position of the characteristic point that forms the domain (i.e., the tip of the cone in the example above).

Validity

Dirichlet domains can form intricate shapes. There is no way to guarantee that traveling from a point in one domain to a point in a connected domain will not pass through a third domain. This

third domain might be impassable and might have been discounted by the pathfinder. In this case, following the path will lead to a problem. Strictly, therefore, Dirichlet domains produce invalid graphs.

In practice, however, the placement of nodes is often based on the structure of obstacles. Obstacles are not normally given their own domains, and so the invalidity of the graph is rarely exposed.

To make sure, you can provide some kind of backup mechanism (like an avoid walls steering behavior) to solve the issue and avoid your characters happily running headfirst into walls.

Usefulness

Dirichlet domains are very widely used. They have the advantage of being very easy to program (automatic generation of connections aside) and easy to change. It is possible to rapidly change the structure of the pathfinding graph in a level editing program without having to change any level geometry.

4.4.3 POINTS OF VISIBILITY

It can be shown that the optimal path through any 2D environment will always have inflection points (i.e., points on the path where the direction changes) at convex vertices in the environment. If the character that is moving has some radius, these inflection points are replaced by arcs of a circle at a distance away from the vertex. This is illustrated in Figure 4.27.

In three dimensions, the same thing applies, but inflection points are located at either convex polygon edges or vertices.

In either case, we can approximate these inflection points by choosing a characteristic point that is shifted out from the vertices a short distance. This will not give us the curves, but it will give us believable paths. These new characteristic points can be calculated from the geometry by extending out the geometry a little way and calculating where the edges of the new geometry are.

Division Scheme

Since these inflection points naturally occur in the shortest path, we can use them as nodes in the pathfinding graph.

Working on the actual level geometry will provide us with far too many inflection points. A simplified version is needed so that we can find inflection points where the large-scale geometry changes. It may be possible to take these points from collision geometry, or they may need to be generated specially.

These inflection points can then be used as the node locations to build a graph.

To work out how these points are connected, rays are cast between them, and a connection is made if the ray doesn't collide with any other geometry. This is almost equivalent to saying that one point can be seen from the other. For this reason it is called a "points of visibility"

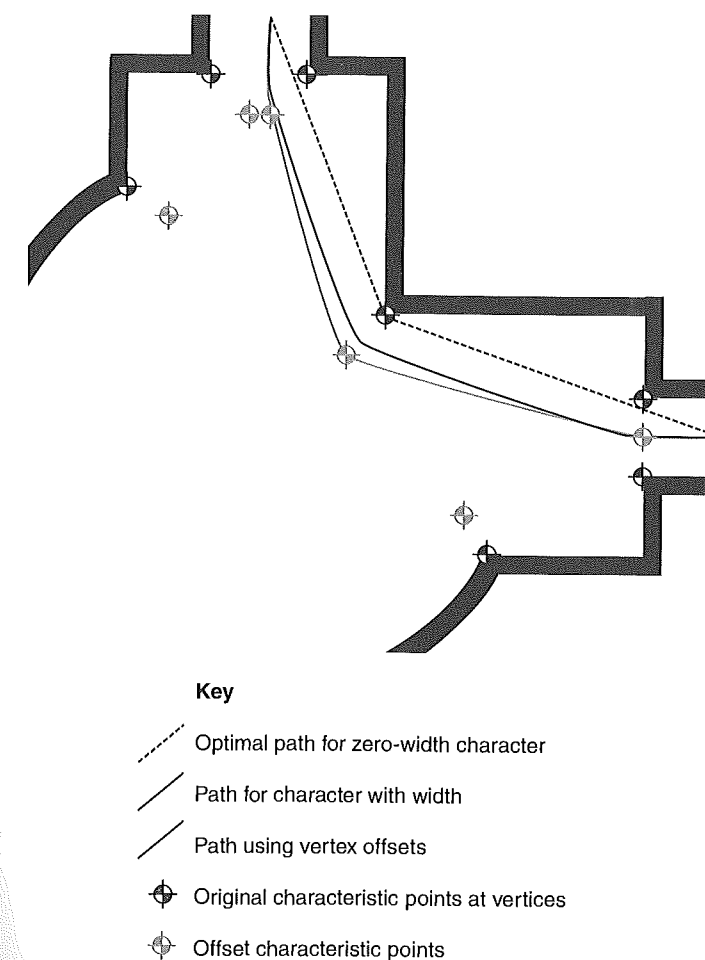
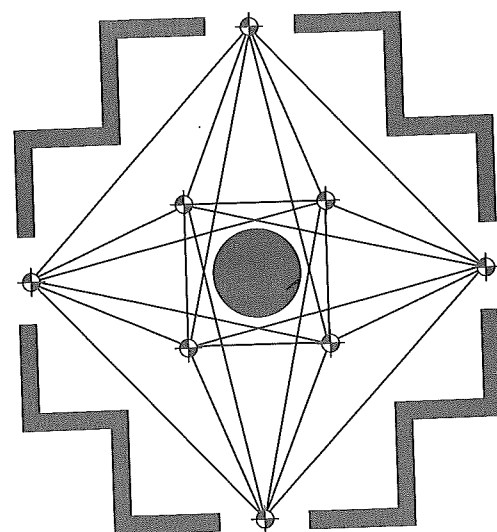


Figure 4.27 Path with inflections at vertices

approach. In many cases the resultant graph is huge. A complex cavern, for example, may have many hundreds of inflection points, each of which may be able to see most of the others. This is shown in Figure 4.28.

Quantization, Localization, and Validity

Points of visibility are usually taken to represent the centers of Dirichlet domains for the purpose of quantization.



Key
 / Connection between nodes

Figure 4.28 Points of visibility graph bloat

In addition, if Dirichlet domains are used for quantization, points quantized to two connected nodes may not be able to reach each other. As we saw in Dirichlet domains above, this means that the graph is strictly invalid.

Usefulness

Despite its major shortcomings, a points of visibility approach is a relatively popular method for automatic graph generation.

However, we think the results are not worth the effort. In our experience a lot of fiddling and clearing up by hand is needed, which defeats the object. We'd recommend looking at navigation meshes instead.

Some AI developers will passionately disagree, however, and swear by points of visibility.

4.4.4 NAVIGATION MESHES

Tile-based graphs, Dirichlet domans, and points of visibility are all useful division schemes to have in your toolbox, but the majority of modern games use navigation meshes (often abbreviated to "navmesh") for pathfinding.

The navigation mesh approach to pathfinding takes advantage of the fact that the level designer already needs to specify the way the level is connected, the regions it has, and whether there is any AI in the game or not. The level itself is made up of polygons connected to other polygons. We can use this graphical structure as the basis of a pathfinding representation.

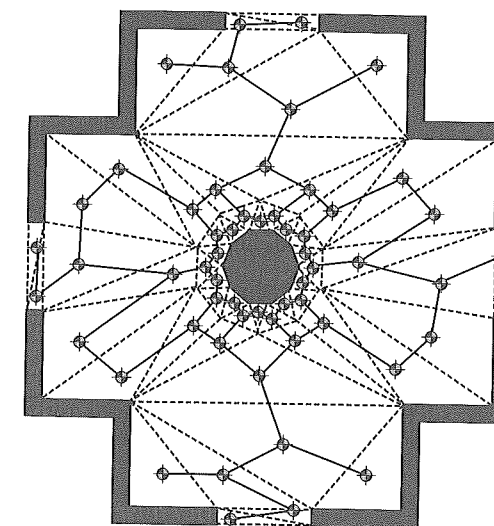
Division Scheme

Many games use floor polygons, as defined by artists, as regions. Each polygon acts as a node in the graph, as shown in Figure 4.29.

The graph is based on the mesh geometry of the level and therefore is often called a navigation mesh, or just "navmesh."

Nodes are connected if their corresponding polygons share an edge. Floor polygons are typically triangles, but may be quads. Nodes therefore have either three or four connections.

Creating a navigation mesh usually involves the artist labeling particular polygons as floor in their modeling package. They may need to do this anyway to specify sound effects or grip characteristics. Navigation meshes require less artist intervention than other approaches, with the exception of tile-based graphs.



Key
 - - - Edge of a floor polygon
 / Connection between nodes

Figure 4.29 Polygonal mesh graph

Quantization and Localization

A position is localized into the floor polygon that contains it. We could search a large number of polygons to find the right one, or we could use a coherence assumption.

Coherence refers to the fact that, if we know which location a character was in at the previous frame, it is likely to be in the same node or an immediate neighbor on the next frame. We can check these nodes first.

This approach is useful in lots of division schemes, but is particularly crucial when dealing with navigation maps.

The only wrinkle occurs when a character is not touching the floor. We can simply find the first polygon below it and quantize it to that. Unfortunately, it is possible for the character to be placed in a completely inappropriate node as it falls or jumps. In Figure 4.30, for example, the character is quantized to the bottom of the room, even though it is actually using the walkways above. This may then cause the character to replan its route as if it were in the bottom of the room; not the desired effect.

Localization can choose any point in the polygon, but normally uses the geometric center (the average position of its vertices). This works fine for triangles. For quads or polygons with more sides, the polygon must be convex for this to work. Geometric primitives used in graphics engines have this requirement anyway. So if we are using the same primitives used for rendering, we are safe.

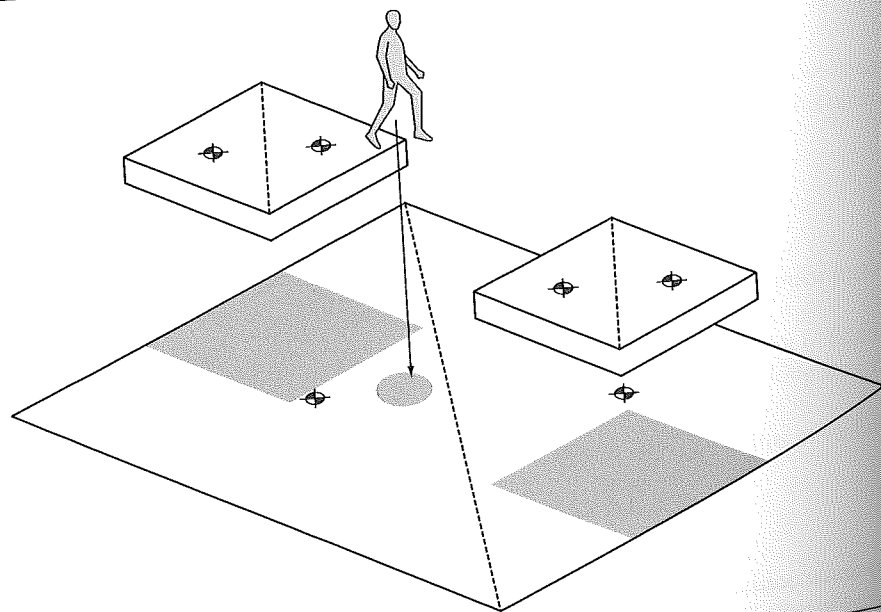


Figure 4.30 Quantization into a gap

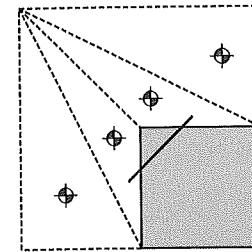


Figure 4.31 Non-interpolation of the navigation mesh

Validity

The regions generated by navigation meshes can be problematic. We have assumed that any point in one region can move directly to any point in a connected region, but this may not be the case. See Figure 4.31 for an example that contains a pair of triangles with areas where traveling directly between them causes a collision.

Because floor polygons are created by the level designer, this situation can be mitigated. The geometry naturally created by most sensible level designers does not suffer from major problems.

Usefulness

Using this approach also requires additional processing to take into account the agent's geometry. Since not all locations in a floor polygon may be occupied by a character (some are too close to the wall), some trimming is required; this may affect the connections generated by finding shared edges. This problem is especially evident at convex areas such as doorways.

Despite this, it is an overwhelmingly popular approach. Games such as *Jak and Daxter* [Naughty Dog, Inc., 2001] and hundreds of others use this approach, as does the PathEngine middleware solution.

For the occasional game that needs it, this approach offers the additional benefit of allowing characters to plan routes up walls, across ceilings, or for any other kind of geometry. This might be useful if characters stick to walls, for example. It is much more difficult to achieve the same result with other world representations.

Edges as Nodes

Floor polygons can also be converted into a pathfinding graph by assigning nodes to the edges between polygons and using connections across the face of each polygon. Figure 4.32 illustrates this.

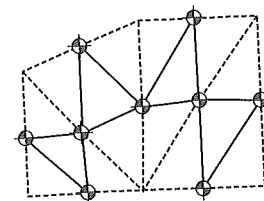


Figure 4.32 Portal representation of a navigation mesh

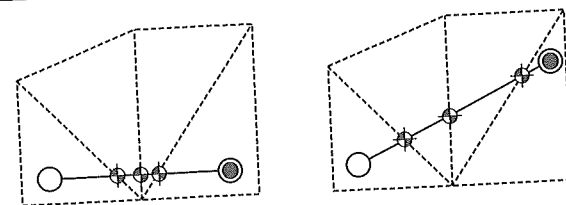


Figure 4.33 Different node positions for different directions

This approach is also commonly used in association with portal-based rendering, where nodes are assigned to portals and where connections link all portals within the line of sight of one another. Portal rendering is a graphics technique where the geometry of the whole level is split into chunks, linked by portals, a 2D polygonal interface between the regions. By separating the level into chunks, it is easier to test which chunks need to be drawn, reducing the rendering time. Full details are beyond the scope of this book, but should be covered in any good modern text on game engine design.

In the navigation mesh, the edges of every floor polygon act like a portal and therefore have their own node. We don't need to do the line-of-sight tests. By definition, each edge of a convex floor polygon can be seen from every other edge.

Some articles we've come across suggest that the nodes on the edges of floor polygons are placed dynamically in the best position as the pathfinder does its work. Depending on the direction that the character is moving, the nodes should be at a different location. This is shown in Figure 4.33.

This is a kind of continuous pathfinding, and we'll look at the algorithm for continuous pathfinding later in the chapter. In our opinion, however, this approach is overkill. It is better to work with the faster fixed graph. If the resulting path looks too crinkled, then a path smoothing step (which we'll cover in Section 4.4.7) is perfectly sufficient.

Both the polygon-as-node and the edge-as-node representations are known as navigation meshes. Often, one or the other approach is assumed, so it is worth making sure that whatever source you are using makes it clear which version they are talking about.

4.4.5 NON-TRANSLATIONAL PROBLEMS

There is nothing in the above discussion about regions and connections that requires us to be dealing with positions only.

In some tile-based games, where agents cannot turn quickly, tiles are created for each location and orientation, so an agent with a large turning circle can only move to a tile with a slightly different orientation in one step.

In Figure 4.34 an agent cannot turn without moving and can only turn by 90° at a time. Nodes A1, A2, A3, and A4 all correspond to the same location. They represent different orientations, however, and they have different sets of connections. The quantization of an agent's state into a graph node needs to take account of both its position and its orientation.

The result from planning on this graph will be a sequence of translations and rotations. A plan on the graph in Figure 4.34 is shown in Figure 4.35.

4.4.6 COST FUNCTIONS

In the simplest cases, where we are interested in finding the shortest path, the cost of a connection can represent distance. The higher the cost, the larger the distance between nodes.

If we are interested in finding the quickest path to move along, we could use costs that depend on time. This isn't the same thing as distance; it is quicker to run 10 feet than to climb a 10-foot ladder.

We can add all sorts of other concerns to the costs on a graph. In an RTS, for example, we could make certain connections more costly if they were exposed to enemy fire or if they wandered too near to dangerous terrain. The final path would then be the one with the lowest danger.

Often, the cost function is a blend of many different concerns, and there can be different cost functions for different characters in a game. A reconnaissance squad, for example, may be interested in visibility and speed. A heavy artillery weapon would be more interested in terrain difficulty. This is called *tactical pathfinding*, and we'll look at it in depth in Chapter 6.

4.4.7 PATH SMOOTHING

A path that travels from node to node through a graph can appear erratic. Sensible node placing can give rise to very odd looking paths. Figure 4.36 shows a section of a level with nodes placed in a reasonable manner. The path shown constantly switches direction; a character following the path will not look intelligent.

Some world representations are more prone to rough paths than others. Portal representations with points of visibility connections can give rise to very smooth paths, while tile-based graphs tend to be highly erratic. The final appearance also depends on how characters act on the path. If they are using some kind of path following steering behavior (see Chapter 3), then the path will

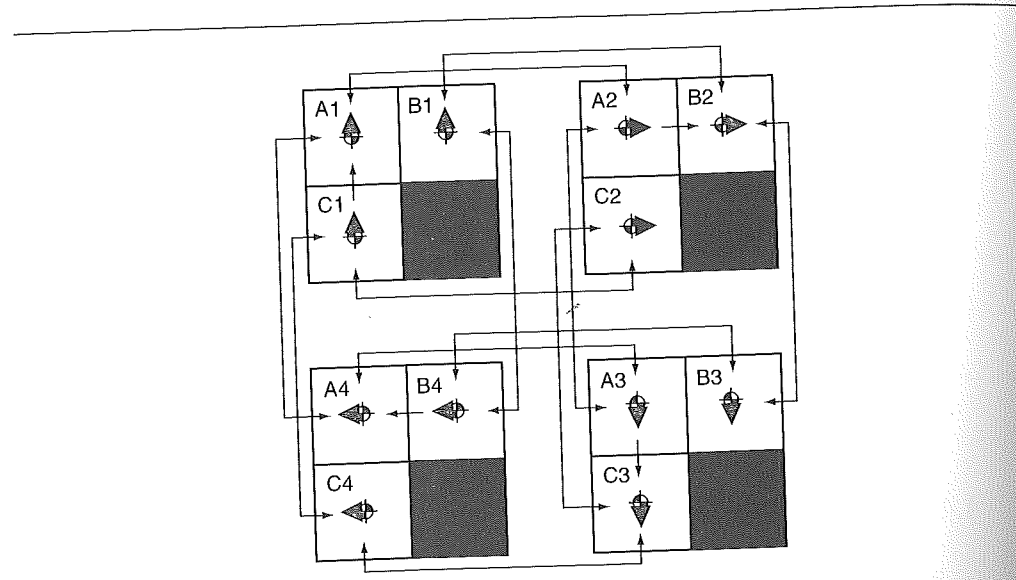


Figure 4.34 A non-translational tile-based world

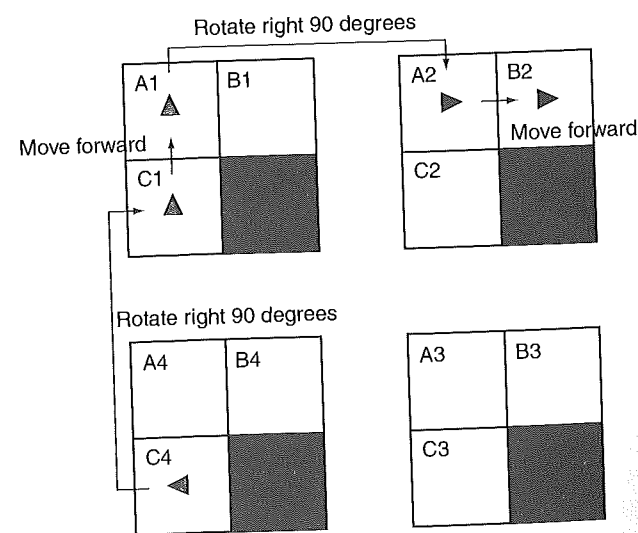


Figure 4.35 Plan on a non-translational tile graph

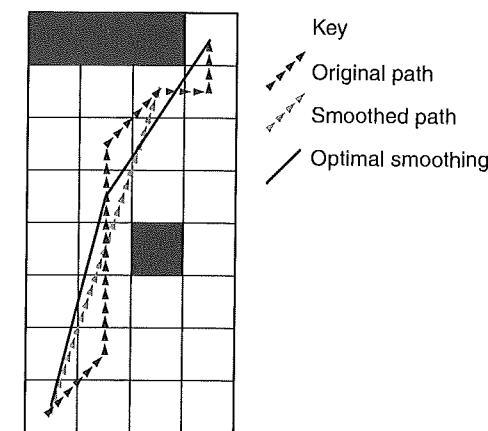


Figure 4.36 Smoothed path with optimal smoothing indicated

be gently smoothed by the steering. It is worth testing your game before assuming the path will need smoothing.

For some games, path smoothing is essential to get the AI looking smart. The path smoothing algorithm is relatively simple to implement but involves queries to the level geometry. Therefore, it can be somewhat time-consuming.

The Algorithm

We will assume in this algorithm that there is a clear route between any two adjacent nodes in the input path. In other words, we are assuming that the division scheme is valid.

First, we create a new empty path. This is the output path. We add the start node to it. The output path will start and end at the same nodes as the input path.

Starting at the third node in the input path, a ray is cast to each node in turn from the last node in the output path. We start at the third node because we are assuming that there is a clear line (a passed ray cast) between the first and second nodes.

When a ray fails to get through, the previous node in the input path is added to the output path. Ray casting starts again from the next node in the input path. When the end node is reached, it is added to the output path. The output path is used as the path to follow.

Figure 4.36 illustrates a path that has been smoothed with this algorithm.

Although this algorithm produces a smooth path, it doesn't search all possible smoothed paths to find the best one. The figure shows the smoothest possible path in our example; it cannot be generated by this algorithm. To generate the smoothest path, we'd need another search among all possible smoothed paths. This is rarely, if ever, necessary.

Pseudo-Code

The path smoothing algorithm takes an input path made up of nodes and returns a smoothed output path:

```

1 def smoothPath(inputPath):
2
3     # If the path is only two nodes long, then
4     # we can't smooth it, so return
5     if len(inputPath) == 2: return inputPath
6
7     # Compile an output path
8     outputPath = [inputPath[0]]
9
10    # Keep track of where we are in the input path
11    # We start at 2, because we assume two adjacent
12    # nodes will pass the ray cast
13    inputIndex = 2
14
15    # Loop until we find the last item in the input
16    while inputIndex < len(inputPath)-1:
17
18        # Do the ray cast
19        if not rayClear(outputPath[len(outputPath)-1],
20                        inputPath[inputIndex]):
21
22            # The ray test failed, add the last node that
23            # passed to the output list
24            outputPath += inputPath[inputIndex-1]
25
26        # Consider the next node
27        inputIndex ++
28
29    # We've reached the end of the input path, add the
30    # end node to the output and return it
31    outputPath += inputPath[len(inputPath)-1]
32    return outputPath

```

Data Structures and Interfaces

The pseudo-code works with paths that are a list of nodes. The pathfinding algorithms so far have returned a path as a list of connections. Although we could take this kind of path as input, the output path cannot be made up of connections. The smoothing algorithm links nodes that are

line of sight, but are unlikely to have any connections between them (if they were connected in the graph, the pathfinder would have found the smoothed route directly, unless their connections had dramatically large costs).

Performance

The path smoothing algorithm is $O(1)$ in memory, requiring only temporary storage. It is $O(n)$ in time, where n is the number of nodes in the path.

The majority of the time spent in this algorithm is spent carrying out ray casting checks.

4.5 IMPROVING ON A*

With a good heuristic, A* is a very efficient algorithm. Even simple implementations can plan across many tens of thousands of nodes in a frame. Even better performance can be achieved using additional optimizations, such as those we considered in the previous sections.

Many game environments are huge and contain hundreds of thousands or even millions of locations. Massively multi-player online games (MMOGs) may be hundreds of times larger still. While it is possible to run an A* algorithm on an environment of this size, it will be extremely slow and take a huge amount of memory. The results are also less than practical. If a character is trying to move between cities in an MMOG, then a route that tells it how to avoid a small boulder in the road five miles away is overkill. This problem can be better solved using hierarchical pathfinding.

Often, many different plans need to be made in quick succession: a whole army may need to plan its routes through a battlefield, for example. Other techniques, such as dynamic pathfinding, can increase the speed of replanning, and a number of A* variations dramatically reduce the amount of memory required to find a path, at the cost of some performance.

The remainder of this chapter will look at some of these issues in detail and will try to give a flavor for the range of different A* variations that are possible.

4.6 HIERARCHICAL PATHFINDING

Hierarchical pathfinding plans a route in much the same way as a person would. We plan an overview route first and then refine it as needed. The high-level overview route might be "To get to the rear parking lot, we'll go down the stairs, out of the front lobby, and around the side of the building," or "We'll go through the office, out the fire door, and down the fire escape." For a longer route, the high-level plan would be even more abstract: "To get to the London office, we'll go to the airport, catch a flight, and get a cab from the airport."

Each stage of the path will consist of another route plan. To get to the airport, for example, we need to know the route. The first stage of this route might be to get to the car. This, in turn, might require a plan to get to the rear parking lot, which in turn will require a plan to maneuver around the desks and get out of the office.