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 re-planning, 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, I'll go down the stairs, out of the front lobby, and around the side of the building," or "I'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, I'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 the 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.

This is a very efficient way of pathfinding. To start with, we plan the abstract route, take the first step of that plan, find a route to complete it, and so on down to the level where we can actually move. After the initial multi-level planning, we only need to plan the next part of the route when we complete a previous section. When I arrive at the bottom of the stairs, on my way to the parking lot (and from there to the London office), I plan my route through the lobby. When I arrive at my car, I then have completed the "get to the car" stage of my next plan up, and I can plan the "drive to the airport" stage.

The plan at each level is typically simple, and we split the pathfinding problem over a long period of time, only doing the next bit when the current bit is complete.

4.6.1 The Hierarchical Pathfinding Graph

To be able to pathfind at higher levels we can still use the A* algorithm and all its optimizations. In order to support hierarchical pathfinding, we need to alter the graph data structure.

Nodes

This is done by grouping locations together to form clusters. The individual locations for a whole room, for example, can be grouped together. There may be 50 navigation

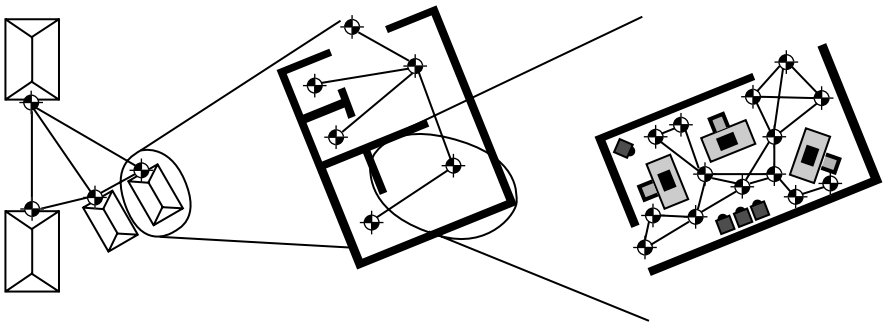

Figure 4.37 Hierarchical nodes

points in the room, but for higher level plans they can be treated as one. This group can be treated as a single node in the pathfinder, as shown in Figure 4.37.

This process can be repeated as many times as needed. The nodes for all the rooms in one building can be combined into a single group, which can then be combined with all the buildings in a complex, and so on. The final product is a hierarchical graph. At each level of the hierarchy, the graph acts just like any other graph you might pathfind on.

To allow pathfinding on this graph, you need to be able to convert a node at the lowest level of the graph (which is derived from the character's position in the game level) to one at a higher level. This is the equivalent of the quantization step in regular graphs. A typical implementation will store a mapping from nodes at one level to groups at a higher level.

Connections

Pathfinding graphs require connections as well as nodes. The connections between higher level nodes need to reflect the ability to move between grouped areas. If any low-level node in one group is connected to any low-level node in another group, then a character can move between the groups, and the two groups should have a connection connected.

Figure 4.38 shows the connections between two nodes based on the connectivity of their constituent nodes at the next level down in the hierarchy.

Connection Costs

The cost of a connection between two groups should reflect the difficulty of travelling between them. This can be specified manually, or it can be calculated from the cost of the low-level connections between those groups.

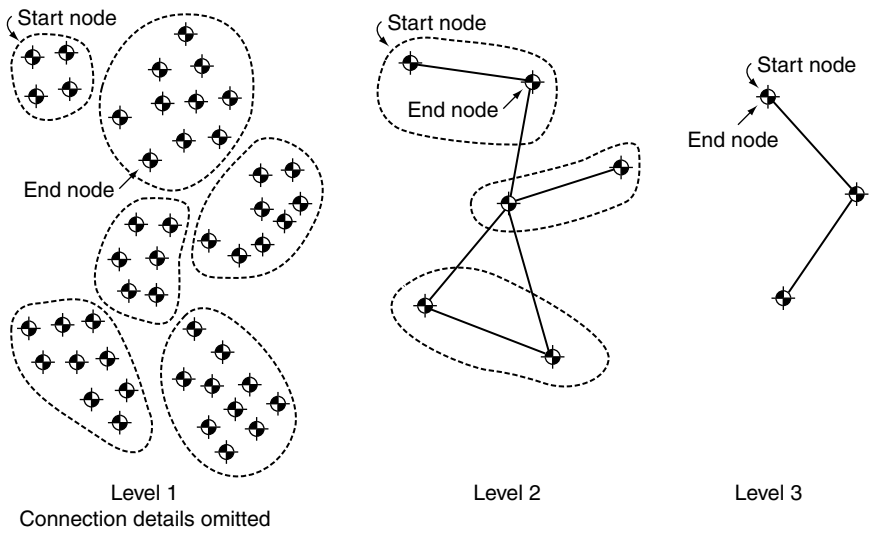

Figure 4.38 A hierarchical graph

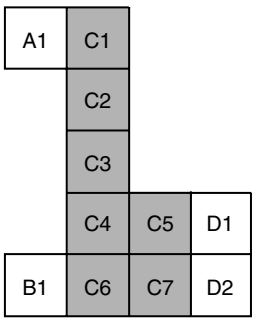

Figure 4.39 A tile-based representation of a level with groups marked

This is a complex calculation, however. Figure 4.39 shows that the cost of moving from group C to group D depends on whether you entered group C from group A (a cost of 1) or from group B (a cost of 4).

In general, the grouping should be chosen to minimize this problem, but it cannot be resolved easily.

There are three heuristics that are commonly used, straight or blended, to calculate the connection cost between groups.

Minimum Distance

The first is minimum distance. This heuristic says that the cost of moving between two groups is the cost of the cheapest link between any nodes in those groups. This makes sense because the pathfinder will try to find the shortest route between two locations. In the example above, the cost of moving from C to D would be 1. Note that if you entered C from either A or B, it would take more than 1 move to get to D. The value of 1 is almost certainly too low, but this may be an important property depending on how accurate you want your final path to be.

Maximin Distance

The second is the "maximin" distance. For each incoming link, the minimum distance to any suitable outgoing link is calculated. This calculation is usually done with a pathfinder. The largest of these values is then added to the cost of the outgoing link and used as the cost between groups.

In the example, to calculate the cost of moving from C to D, two costs are calculated: the minimum cost from C1 to C5 (4) and the minimum cost from C6 to C7 (1). The largest of these (C1 to C5) is then added to the cost of moving from C5 to D1 (1). This leaves a final cost from C to D of 5. To get from C to D from anywhere other than C1, this value will be too high. Just like the previous heuristic, this may be what you need.

Average Minimum Distance

A value in the middle of these extremes is sometimes better. The "average minimum" distance is a good general choice. This can be calculated in the same way as the maximin distance, but the values are averaged, rather than simply selecting the largest. In our example, to get from C to D coming from B (i.e., via C6 and C7), the cost is 2, and when coming from A (via C2 to C5) it is 5. So the average cost of moving from C to D is $\frac{31}{2}$.

Summary of the Heuristics

The minimum distance heuristic is very optimistic. It assumes that there will never be any cost to moving around the nodes within a group. The maximin distance heuristic is pessimistic. It finds one of the largest possible costs and always uses that. The average minimum distance heuristic is pragmatic. It gives the average cost you'll pay over lots of different pathfinding requests.

The approach you choose isn't only dictated by accuracy; each heuristic has an effect on the kinds of paths returned by a hierarchical pathfinder. We'll return to why this is so after we look in detail at the hierarchical pathfinding algorithm.

4.6.2 Pathfinding on the Hierarchical Graph

Pathfinding on a hierarchical graph uses the normal A* algorithm. It applies the A* algorithm several times, starting at a high level of the hierarchy and working down. The results at high levels are used to limit the work it needs to do at lower levels.

Algorithm

Because a hierarchical graph may have many different levels, the first task is to find which level to begin on. We want as high a level as possible, so we do the minimum amount of work. However, we also don't want to be solving trivial problems either.

The initial level should be the first in which the start and goal locations are not at the same node. Any lower and we would be doing unnecessary work; any higher and the solution would be trivial, since the goal and start nodes are identical.

In Figure 4.38 the pathfinding should take place initially at level 2, because level 3 has the start and end locations at the same node.

Once a plan is found at the start level, then the initial stages of the plan need to be refined. We refine the initial stages because those are the most important for moving the character. We won't initially need to know the fine detail of the end of the plan; we can work that out nearer the time.

The first stage in the high-level plan is considered (occasionally, it can be useful to consider the first few; this is a heuristic that needs to be tried for different game worlds). This small section will be refined by planning at a slightly lower level in the hierarchy.

The start point is the same, but if we kept the end point the same we'd be planning through the whole graph at this level, so our previous planning would be wasted. So the end point is set at the end of the first move in the high-level plan.

For example, if we are planning through a set of rooms, the first level we consider might give us a plan that takes us from where our character is at in the lobby to the guardroom and from there to its goal in the armory. At the next level we are interested in maneuvering around obstacles in the room, so we keep the start location the same (where the character currently is), but set the end location to be the doorway between the lobby and the guardroom. At this level we will ignore anything we may have to do in the guardroom and beyond.

This process of lowering the level and resetting the end location is repeated until we reach the lowest level of the graph. Now we have a plan in detail for what the character needs to do immediately. We can be confident that, even though we have only looked in detail at the first few steps, making those moves will still help us complete our goal in a sensible way.

Pseudo-Code

The algorithm for hierarchical pathfinding has the following form:

```
def hierarchicalPathfind(graph, start, end, heuristic):

  # Check if we have no path to find
  if start == end: return None

  # Set up our initial pair of nodes
  startNode = start
  endNode = end
  levelOfNodes = 0

  # Descend through levels of the graph
  currentLevel = graph.getLevels()-1
  while currentLevel >= 0:

    # Find the start and end nodes at this level
    startNode = graph.getNodeAtLevel(0, start,
                                     currentLevel)
    endNode = graph.getNodeAtLevel(levelOfNodes,
                                   endNode, currentLevel)
    levelOfNodes = currentLevel

    # Are the start and end node the same?
    if startNode == endNode:

      # Skip this level
      continue

    # Otherwise we can perform the plan
    graph.setLevel(currentLevel)
    path = pathfindAStar(graph, startNode, endNode, heuristic)

    # Now take the first move of this plan and use it
    # for the next run through
    endNode = path[0].getToNode()

  # The last path we considered would have been the
  # one at level zero: we return it.
  return path
```

Data Structures and Interfaces

We have made some additions to our graph data structure. Its interface now looks like the following:

```
class HierarchicalGraph (Graph):

  # ... Inherits getConnections from graph ...

  # Returns the number of levels in the graph
  def getLevels()

  # Sets the graph so all future calls to getConnections
  # are treated as requests at the given level
  def setLevel(level)

  # Converts the node at the input level into a node
  # at the output level.
  def getNodeAtLevel(inputLevel, node, outputLevel)
```

The `setLevel` method switches the graph into a particular level. All calls to `getConnections` then act as if the graph was just a simple, non-hierarchical graph at that level. The A* function has no way of telling that it is working with a hierarchical graph; it doesn't need to.

The `getNodeAtLevel` method converts nodes between different levels of the hierarchy. When increasing the level of a node, we can simply find which higher level node it is mapped to. When decreasing the level of a node, however, one node might map to any number of nodes at the next level down.

This is just the same process as localizing a node into a position for the game. There are any number of positions in the node, but we select one in localization. The same thing needs to happen in the `getNodeAtLevel` method. We need to select a single node that can be representative of the higher level node. This is usually a node near the center, or it could be the node that covers the greatest area or the most connected node (an indicator that it is a significant one for route planning).

Personally, I have used a fixed node at a lower level, generated by finding the node closest to the center of all those mapped to the same higher level node. This is a fast, geometric, pre-processing step that doesn't need human intervention. This node is then stored with the higher level node, and it can be returned when needed without additional processing. This has worked well and produced no problems for me, but you may prefer to try different methods or manual specification by the level designer.

Performance

The A* algorithm has the same performance characteristics as before, since we are using it unchanged.

The hierarchical pathfinder function is $O(1)$ in memory and $O(p)$ in time, where p is the number of levels in the graph. Overall, the function is $O(plm)$ in time. Obviously, this appears to be slower than the basic pathfinding algorithm.

And it may be. It is possible to have a hierarchical graph that is so poorly structured that the overall performance is lower. In general, however, there are p stages of the $O(lm)$ A* algorithm, but in each case the number of iterations (l) should be much smaller than a raw A* call, and the practical performance will be significantly higher.

For large graphs (tens of thousands of nodes, for example) it is not uncommon to see two orders of magnitude improvement in running speed, using several levels in the hierarchy. I've used this technique to allow characters to pathfind on a graph with a hundred million nodes in real time (with the AI getting 10% of the processor time).

4.6.3 Hierarchical Pathfinding on Exclusions

A character can only follow the plan generated by the previous algorithm for a short time. When it reaches the end of the lowest level plan, it will need to plan its next section in more detail.

When its plan runs out, the algorithm is called again, and the next section is returned. If you use a pathfinding algorithm that stores plans (see Section 4.7, Other Ideas in Pathfinding), the higher level plans won't need to be rebuilt from scratch (although that is rarely a costly process).

In some applications, however, you might prefer to get the whole detailed plan up front. In this case hierarchical pathfinding can still be used to make the planning more efficient.

The same algorithm is followed, but the start and end locations are never moved. Without further modification, this would lead to a massive waste of effort, as we are performing a complete plan at each level.

To avoid this, at each lower level, the only nodes that the pathfinder can consider are those that are within a group node that is part of the higher level plan.

For example, in Figure 4.40 the first high-level plan is shown. When the low-level plan is made (from the same start and end locations), all the shaded nodes are ignored. They are not even considered by the pathfinder. This dramatically reduces the size of the search, but it can also miss the best route, as shown.

It is not as efficient as the standard hierarchical pathfinding algorithm, but it can still be a very powerful technique.

4.6.4 Strange Effects of Hierarchies on Pathfinding

It is important to realize that hierarchical pathfinding gives an approximate solution. Just like any other heuristic, it can perform well in certain circumstances and poorly in others. It may be that high-level pathfinding finds a route that can be a shortcut at a lower level. This shortcut will never be found, because the high-level route is "locked in" and can't be reconsidered.

The source of this approximation is the link costs we calculated when we turned the lowest level graph into a hierarchical graph. Because no single value can accurately

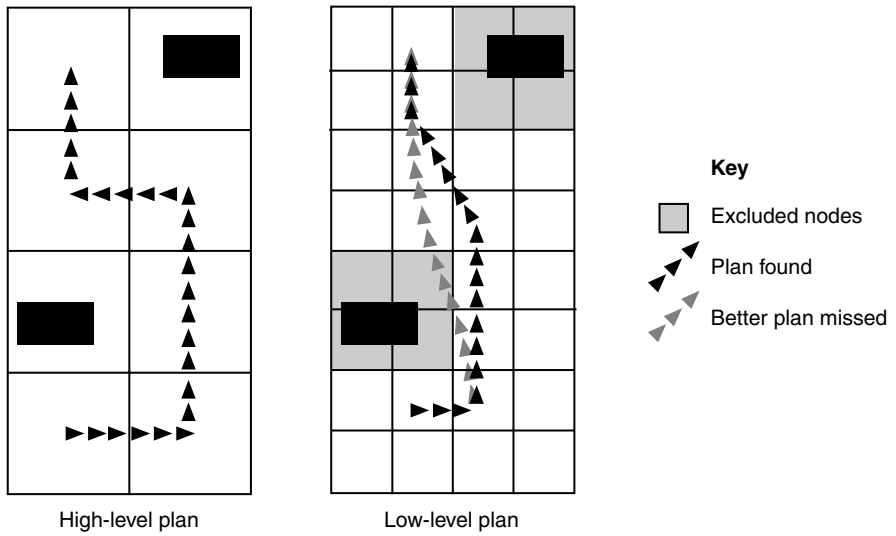

Figure 4.40 Switching off nodes as the hierarchy is descended

represent all the possible routes through a group of nodes, they will always be wrong some of the time.

Figures 4.41 and 4.42 show cases in which each method of calculating the link cost produces the wrong path.

In Figure 4.41 we see that because the minimum cost is 1 for all connections between rooms, the path planner chooses the route with the smallest number of rooms, rather than the much more direct route. The minimum cost method works well for situations where each room is roughly the same size.

We see in Figure 4.42 that the obvious, direct route is not used because the connection has a very large maximin value. The maximin algorithm works better when every route has to pass through many rooms.

In the same example, using the average minimum method does not help, since there is only one route between rooms. The direct route is still not used. The average minimum method often performs better than the maximin method, except in cases where most of the rooms are long and thin with entrances at each end (networks of corridors, for example) or when there are few connections between rooms.

The failure of each of these methods doesn't indicate that there is another, better, method that we haven't found yet; all possible methods will be wrong in some cases. Whatever method you use, it is important to understand what effects the wrongness has. One of the scenarios above, or a blend of them, is likely to provide the optimum trade-off for your game, but finding it is a matter of tweaking and testing.

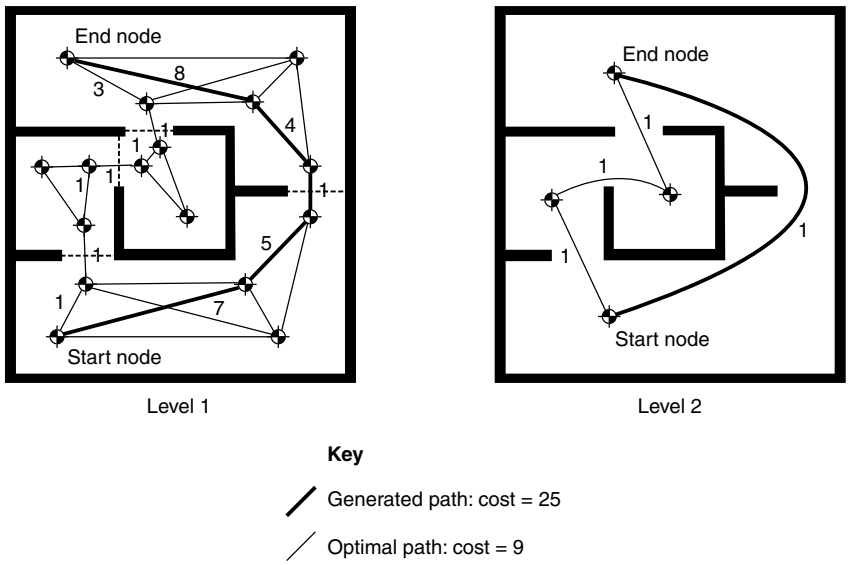

Figure 4.41 Pathological example of the minimum method

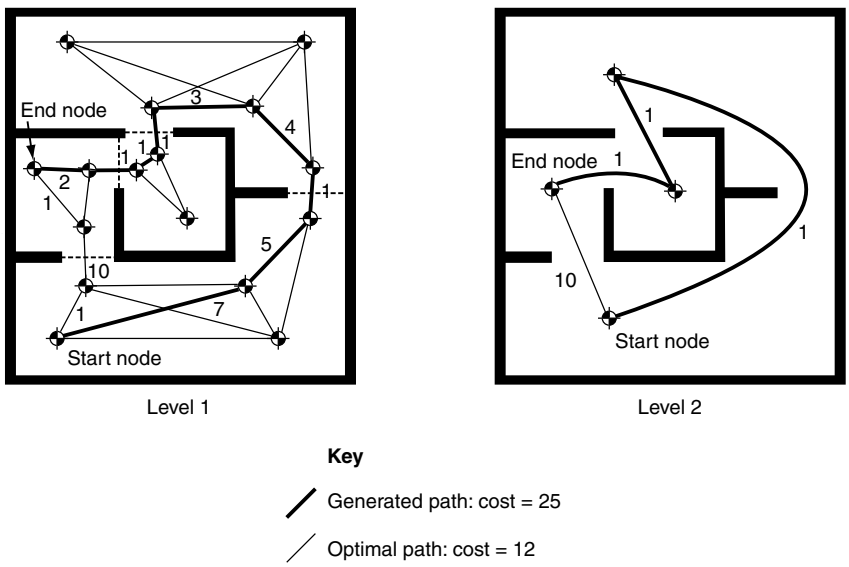

Figure 4.42 Pathological example of the maximin method

4.6.5 Instanced Geometry

In a single-player game or a level-based multi-player game, all the detail for a level is typically unique. If multiple copies of the same geometry are used, then they are usually tweaked to be slightly different. The pathfinding graph is unique for the whole level, and it doesn't make sense to use the same subsection of the graph for more than one area in the level.

For massively multi-player games, the whole world can consist of a single level. There is no way to have the same detailed, unique modelling on this scale. Most MMOGs use one large definition for the topology of the landscape (typically, a height field grid that can be represented to the pathfinding system as a tile-based graph). Onto this landscape buildings are placed, either as a whole or as entrances to a separate mini-level representing the inside of the building. Tombs, castles, caves, or spaceships can all be implemented in this way. I've used the technique to model bridges connecting islands in a squad-based game, for example. For simplicity, I'll refer to them all as a building in this section.

These placed buildings are sometimes unique (for special areas with significance to the game content). In most cases, however, they are generic. There may be 20 farmhouse designs, for example, but there may be hundreds of farmhouses across the world. In the same way that the game wouldn't store many copies of the geometry for the farmhouse, it shouldn't store many copies of the pathfinding graph.

We would like to be able to instantiate the pathfinding graph so that it can be reused for every copy of the building.

Algorithm

For each type of building in the game, we have a separate pathfinding graph. The pathfinding graph contains some special connections labelled as "exits" from the building. These connections leave from nodes that we'll call "exit nodes." They are not connected to any other node in the building's graph.

For each instance of a building in the game, we keep a record of its type and which nodes in the main pathfinding graph (i.e., the graph for the whole world) each exit is attached to. Similarly, we store a list of nodes in the main graph that should have connections into each exit node in the building graph. This provides a record of how the building's pathfinding graph is wired into the rest of the world.

Instance Graph

The building instance presents a graph to be used by the pathfinder. Let's call this the instance graph. Whenever it is asked for a set of connections from a node, it refers to the corresponding building type graph and returns the results.

To avoid the pathfinder getting confused about which building instance it is in, the instance makes sure that the nodes are changed so that they are unique to each instance.

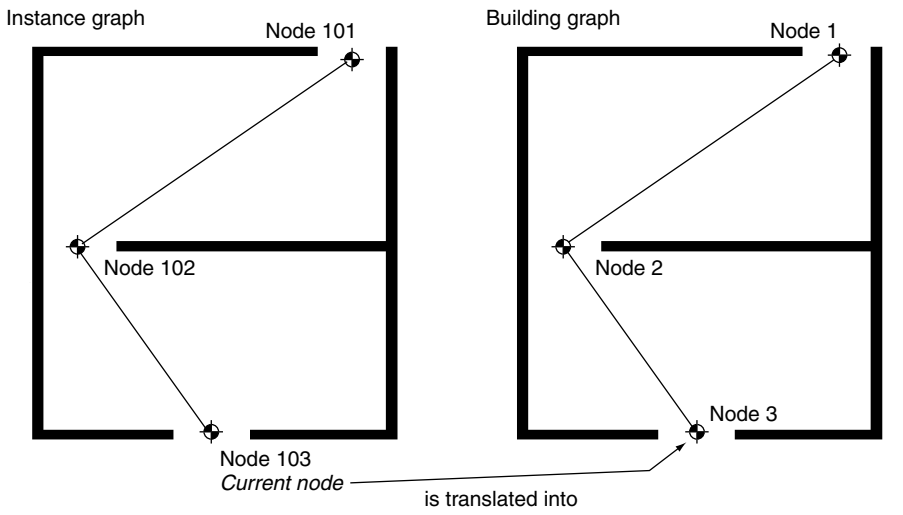

Figure 4.43 The delegation in progress

The instance graph is simply acting as a translator. When asked for connections from a node, it translates the requested node into a node value understood by the building graph. It then delegates the connection request to the building graph, as shown in Figure 4.43. Finally, it translates the results so that the node values are all instance-specific again and returns the result to the pathfinder.

For exit nodes, the process adds an additional stage. The building graph is called in the normal way, and its results are translated. If the node is an exit node, then the instance graph adds the exit connections, with destinations set to the appropriate nodes in the main pathfinding graph.

Because it is difficult to tell the distance between nodes in different buildings, the connection costs of exit connections are often assumed to be zero. This is equivalent to saying that the source and destination nodes of the connection are at the same point in space.

World Graph

To support entrance into the building instance, a similar process needs to occur in the main pathfinding graph. Each node requested has its normal set of connections (the eight adjacent neighbors in a tile-based graph, for example). It may also have connections into a building instance. If so, the world graph adds the appropriate connection to the list. The destination node of this connection is looked up in the instance definition, and its value is in instance graph format. This is illustrated in Figure 4.44.

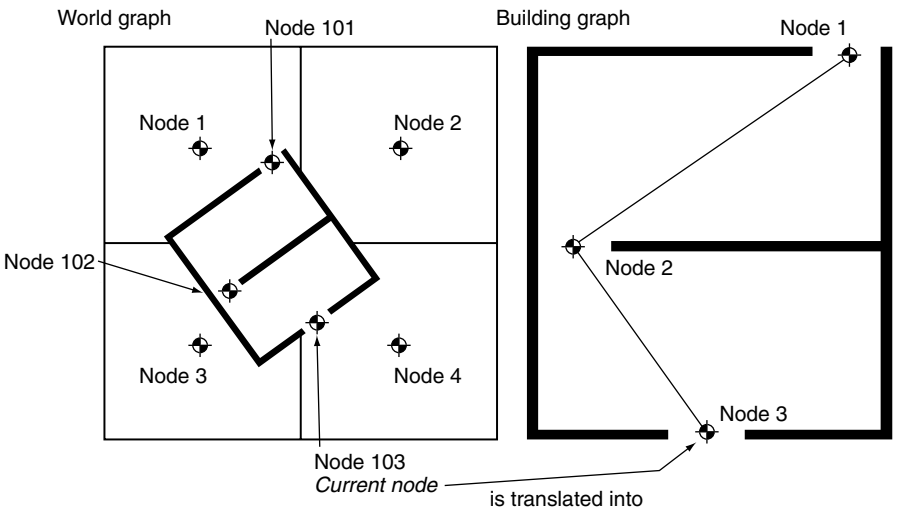

Figure 4.44 An instance in the world graph

The pathfinder, as we've implemented it in this chapter, can only handle one graph at a time. The world graph manages all the instance graphs to make it appear as if it is generating the whole graph. When asked for the connections from a node, it first works out which building instance the node value is from or if it is from the main pathfinding graph. If the node is taken from a building, it delegates to that building to process the `getConnections` request and returns the result unchanged. If the node is not taken from a building instance, it delegates to the main pathfinding graph, but this time adds connections for any entrance nodes into a building.

If you are building a pathfinder from scratch to use in a game where you need instancing, it is possible to include the instancing directly in the pathfinding algorithm, so it makes calls to both the top-level graph and the instanced graphs. This approach makes it much more difficult to later incorporate other optimizations such as hierarchical pathfinding, or node array A*, however, so we'll stick with the basic pathfinding implementation here.

Pseudo-Code

To implement instanced geometry we need two new implicit graphs: one for building instances and one for the main pathfinding graph.

I've added the data used to store the building instance with the instance graph class, since the same data is needed for each. The instance graph implementation therefore has the following form:

```
class InstanceGraph (Graph):

  # Holds the building graph to delegate to
  buildingGraph

  # Holds data for exit nodes
  struct ExitNodeAssignment:
    fromNode
    toWorldNode

  # Holds a hash of exit node assignments for
  # connections to the outside world
  exitNodes

  # Stores the offset for the nodes values used in
  # this instance.
  nodeOffset

  def getConnections(fromNode):

    # Translate the node into building graph values+
    buildingFromNode = fromNode - nodeOffset

    # Delegate to the building graph
    connections =
      buildingGraph.getConnections(buildingFromNode)

    # Translate the returned connections into instance
    # node values
    for connection in connections:
      connection.toNode += nodeOffset

    # Add connections for each exit from this node
    for exitAssignment in exitNodes[fromNode]:
      connection = new Connection
      connection.fromNode = fromNode
      connection.toNode = exitAssignment.toWorldNode
      connection.cost = 0
      connections += connection
```

```
40
41      return connections
```

The main pathfinding graph has the following structure:

```
1   class MainGraph (Graph):
2
3     # Holds the graph for the rest of the world
4     worldGraph
5
6     # Holds data for a building instance
7     struct EntranceNodeAssignment:
8       fromNode
9       toInstanceNode
10      instanceGraph
11
12    # Holds entrance node assignments. This data structure
13    # can act as a hash, and is described below.
14    buildingInstances
15
16    # Holds a record of
17
18    def getConnections(fromNode):
19
20      # Check if the fromNode is in the range of any
21      # building instances
22      building = buildingInstances.getBuilding(fromNode)
23
24      # If we have a building, then delegate to the building
25      if building:
26        return building.getConnections(fromNode)
27
28      # Otherwise, delegate to the world graph
29      else:
30        connections = worldGraph.getConnections(fromNode)
31
32        # Add connections for each entrance from this node.
33        for building in buildingInstances[fromNode]:
34           connection = new Connection
35           connection.fromNode = fromNode
36           connection.toNode = building.toInstanceNode
37           connection.cost = 0
38           connections += connection
```

```
39
40      return connections
```

Data Structures and Interfaces

In the instance graph class, we access the exit nodes as a hash, indexed by node number and returning a list of exit node assignments. This process is called every time the graph is asked for connections, so it needs to be implemented in an efficient a manner as possible. The building instances structure in the world graph class is used in exactly the same way, with the same efficiency requirements.

The building instances structure also has a `getBuilding` method in the pseudo-code above. This method takes a node and returns a building instance from the list if the node is part of the instance graph. If the node is part of the main pathfinding graph, then the method returns a null value. This method is also highly speed critical. Because a range of node values are used by each building, however, it can't be easily implemented as a hash table. A good solution is to perform a binary search on the `nodeOffsets` of the buildings. A further speed up can be made using coherence, taking advantage of the fact that if the pathfinder requests a node in a building instance, it is likely to follow it with requests to other nodes in the same building.

Implementation Nodes

The translation process between instance node values and building node values assumes that nodes are numeric values. This is the most common implementation of nodes. However, they can be implemented as an opaque data instead. In this case the translation operations (adding and subtracting `nodeOffset` in the pseudo-code) would be replaced by some other operation on the node data type.

The main pathfinding graph for a tile-based world is usually implicit. Rather than delegating from a new implicit graph implementation to another implicit implementation, it is probably better to combine the two. The `getConnections` method compiles the connections to each neighboring tile, as well as checks for building entrances. The implementationon the CD follows this pattern.

LIBRARY

Performance

Both the instance graph and the world graph need to perform a hash lookup for entrance or exit connections. This check takes place at the lowest part of the pathfinding loop and therefore is speed critical. For a well-balanced hash, the speed of hash lookup approaches O(1).

The world graph also needs to look up a building instance from a node value. In the case where nodes are numeric, this cannot be performed using a reasonably

sized hash table. A binary search implementation is $O(\log_2 n)$ in time, where n is the number of buildings in the world. Judicial use of caching can reduce this to almost $O(1)$ in practice, although pathological graph structures can theoretically foil any caching scheme and give the $O(\log_2 n)$ worst case.

Both algorithms are $O(1)$ in memory, requiring only temporary storage.

Weaknesses

This approach introduces a fair amount of complexity low down in the pathfinding loop. The performance of the pathfinder is extremely sensitive to inefficiencies in the graph data structure. I've seen a halving of execution speed by using this method. It is not worth the extra time if the game level is small enough to create a single master graph.

For massive worlds with instanced buildings, however, this may not be an option, and instanced graphs are the only way to go. I would personally not consider using instanced graphs in a production environment unless the pathfinding system was hierarchical (if the graph is big enough to need instanced buildings, it is big enough to need hierarchical pathfinding). In this case each building instance can be treated as a single node higher up the hierarchy. When using a hierarchical pathfinding algorithm, moving to instanced geometry usually produces a negligible drop in pathfinding speed.

Setting Node Offsets

In order for this code to work, we need to make sure that every building instance has a unique set of node values. The node values should not only be unique within instances of the same building type, but also between different building types. If node values are numeric, this can be simply accomplished by assigning the first building instance a `nodeOffset` equal to the number of nodes in the main pathfinding graph. Thereafter, subsequent building instances have offsets which differ from the previous building by the number of nodes in the previous building's graph.

For example, let's say we have a pathfinding graph of 10,000 nodes and three building instances. The first and third buildings are instances of a type with 100 nodes in the graph. The second building has 200 nodes in its graph. Then the node offset values for the building instances would be 10,000; 10,200; and 10,300.

4.7 Other Ideas in Pathfinding

There are many variations on the A* algorithm that have been developed for specific applications. It would take a book this size to describe them all. This section takes a whirlwind tour of some of the most interesting. There are pointers to more information, including algorithm specifications, in the references at the end of the book.