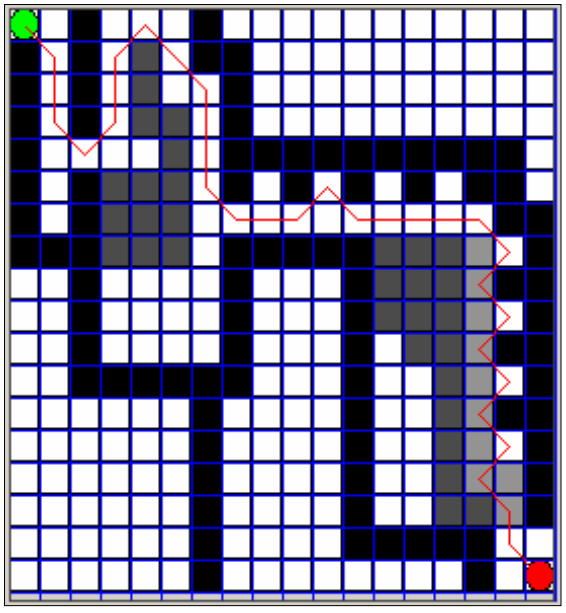
Chapter 2

Pathfinding II



Overview

In the last chapter we learned about some of the important AI subcategories that game developers will encounter in their projects. We mentioned that the two subcategories that we were going to focus on in this course are decision making and pathfinding since they are the two most common and critical AI components in the majority of games. Along the way learned that even these two seemingly very different concepts are somewhat related to one another. At the very least, we know that the goal in both cases is to use algorithms to produce behaviors that appear intelligent to the end user. After all, this was how we defined artificial intelligence. In the case of pathfinding we know that the idea is to use our knowledge of the environment to create algorithms that determine the best way to travel from place to place. From the player's perspective, the end result will be entities that maneuver around obstacles in the game world as they attempt to reach a given destination. Since those destination points will be updated quite frequently in real-time (based on decision making techniques we will learn about later in the course), the illusion of autonomous intelligent entities is fostered and maintained.

So far, we have examined some fundamental types of pathfinding and how it relates to games. In this chapter, we will discuss more advanced pathfinding techniques used in games and see how to apply them. Primarily we will discuss A* and its common advantages and disadvantages as well as how heuristics can be used to produce better results. Additionally, we will discuss ways of simplifying the pathfinding problem with hierarchical pathfinding. We will also discuss methodologies for pathfinding in non-gridded environments such as we find in many 3D games (although our chosen implementation will not come until later in the course, after we have discussed decision making in detail). Finally, we will discuss the chapter demo in detail and the design stratagems employed in its development.

In this chapter we will answer the following questions:

- What is A*?
- What are some of the advantages and disadvantages of A*?
- What are heuristics and how can they be used to assist A*?
- How can A*'s behavior be modified with different heuristics?
- What is hierarchical pathfinding and how can it be used?
- What are some of the methodologies for extending pathfinding systems for use in non-gridded environments?
- What is the Algorithm Design Strategy?
- What is the Grid Design Strategy?

2.1 A*: The New Star in Pathfinding

A* is a more recent development in the arena of pathfinding algorithms. It combines the power of heuristics from Best First Search to limit its search time, with the ability to deal with weighted graphs from Dijkstra's. It is an extremely versatile algorithm which many of today's games use for their core pathfinding needs.

2.1.1 How A* Works

```
bool AStarSearch(Node start, Node goal)
{
	PriorityQueue open;
	List closed;
	Node  n,  child;
	start.parent = NULL;
	open.enqueue(start);
	while(!open.isEmpty())
	{
		n = open.dequeue();
		if (n == goal)
		{
			makePath();
			return true;
		}

		while (n.hasMoreChildren())
		{
			child = n.getNextChild();
			COSTVAL newg = n.g + child.cost;
			if ((open.contains(child) || closed.contains(child))
				&& child.g <= newg)
				continue;
			child.parent = n;
			child.g = newg;
			child.h = GoalEstimate(child);
			child.f = child.g + child.h;
			if(closed.contains(child))
				closed.remove(child)
			if(!open.contains(child))
				open.enqueue(child);
			else
				open.requeue(child);
		}
		closed.add(n);
	}

	return false;
}
```

Above we have a fairly generic implementation of A*. It makes some assumptions about the type of container classes to use, but otherwise it is fundamentally how A* works. After reviewing it, you should see that it is very similar to Dijkstra's and Best First Search combined into one algorithm. Note that it searches like Dijkstra's, but uses the heuristic estimate to limit the search as in Best First Search. A* calculates three values for each node: f, g, and h. The g value is the current true cost to get to the node. The h value is the heuristic estimate value, which is typically the estimated cost from the node to the goal. The f value is the sum of h and g, and is the value by which A* sorts the nodes it will search, with lowest f values being searched first. Let us go over this algorithm in a little more detail.

```
bool AStarSearch(Node start, Node goal)
```

Like the algorithms we discussed in the previous chapter, we get a start node and a goal node, and return whether we found a path or not.

```
	PriorityQueue open;
	List closed;
```

Unlike our previous algorithms, A* has two lists to keep track of. The **open list** is a priority queue just like in Djikstra's and Best First Search. This will allow us to go through our candidate nodes in the order that makes the most sense. The **closed list** is a list of nodes we have already searched, but might need to examine again at a later time.

```
	Node  n,  child;
	start.parent = NULL;

	open.enqueue(start);
```

Like the other algorithms, we will want a current node, and a current child node. We will set the parent of the start node to NULL since we know it is the start, and we prime the open priority queue by adding the start node to it since that is where we begin.

```
	while(!open.isEmpty())
```

Like the other algorithms, we also iterate through the queue until we find the goal or the queue empties. If we do not find the goal before the queue empties, there is no path from the start node to the end node.

```
		n = open.dequeue();
		if (n == goal)
		{
			makePath();
			return true;
		}
```

For each iteration, we will grab a node off the queue. We then check to see if it is the goal node, and if it is, we make the path and return our success.

```
		while (n.hasMoreChildren())
```

We then iterate across all the children of our current node.

```
			child = n.getNextChild();
			COSTVAL newg = n.g + child.cost;
```

For each child, we compute a new actual cost based on the current node's actual cost and the cost to the child.

```
			if ((open.contains(child) || closed.contains(child))
```

```
                        && child.g <= newg)
                        continue;
```

If either the queue or the closed list contains the child, and the child's actual cost is less than the newly computed cost, we skip this child since we already have the shortest path to this child in the queue.

```
                    child.parent = n;
                    child.g = newg;
                    child.h = GoalEstimate(child);
                    child.f = child.g + child.h;
```

If we determine that we need to visit this child, we set the child's parent to the current node, set the computed actual cost to the child, set the child's estimated distance to the goal using our heuristic (just like in Best First Search), and set our total cost value to be the sum of the actual cost to this node plus the estimated cost to the goal.

```
                    if(closed.contains(child))
                          closed.remove(child)
                    if(!open.contains(child))
                          open.enqueue(child);
                    else
                          open.requeue(child);
```

Next, if the closed list contains the child, we need to remove it since we want to visit it again. Also, if the child is not in the open queue already, we add it. If it is, we requeue it so that it gets placed in the right spot in the queue with its new total cost.

```
              closed.add(n);
```

After we visit all of the children of the current node, we place the current node in the closed list so we do not visit it again, unless we find a cheaper way to get there.

To summarize, we start off with two lists, open and closed. The open list is the list of nodes which needs to be searched and the closed list is the list of nodes which have already been searched. If there is a node in the closed list to which a shorter path is found, it is updated and moved to the open list again. The algorithm starts by putting the start node in the open list. Then, while the open list is not empty, the node with the lowest *f* value is checked to see if it is the goal. If it is, we make the path and return. If it is not, we iterate through all of its children. For each child, we determine a new *g* value, and check to see if the child is already in a list, as well as whether its cost is less expensive. If so, we ignore this child. Otherwise, the parent node is set, the *g* value is set to the new *g* value we calculated, the *h* value is set to the heuristic estimate, and the *f* value is calculated. Then, if the child is in the closed list, we remove it from the list. If the child is not in the open list, it is added. If it is in the open list, its position in that list is updated. After iteration through all of the node's children, the node to the closed list is added.

2.1.2 Limitations of A*

A* is a wonderful pathfinding algorithm but it is not without limitations. A* will always find the shortest path to the goal, provided the heuristic estimate from any given child to the goal is never greater than the real distance to the goal. If the estimate is greater than the true distance to the goal, the algorithm will produce results which are not optimal. Moreover, the open and closed lists in A* can be inefficient when dealing with very large graphs. If the methods used to locate nodes in the lists, add nodes to the lists, and remove nodes from the lists are inefficient, the algorithm will be very slow. In addition, the memory requirements to store the nodes in these lists would increase dramatically. A* is a strong contender to keep in mind for your pathfinding needs, but it is always better to use the simplest method when the simplest method works well. Remember our KISS principle from the first chapter!

2.1.3 Making A* More Efficient

A* can be made more efficient so that it becomes a solid choice for our game needs. The first approach to optimizing A* is to improve the storage method for the open and closed lists. Some versions use queues, others use stacks, some use heaps, while still others use hash maps. The important point is to select a container class which can quickly locate nodes, insert nodes, remove nodes, and in the case where the list is not ordered, sort the list. Priority queues are a good choice because they keep the selection of the next best node simple because it is always at the front of the list. Stacks provide easy insertion and removal, and hash maps provide quick node location. They all have their benefits, but we cannot have the best of all worlds. So you should definitely experiment with different containers to find out which one works best in your particular implementation.

Another optimization is to consider the search from a higher level and perform smaller searches for each step along the way. In a dungeon you might go room by room, or on a large outdoor map you might divide it into larger squares and go from region to region. In this case you will begin with a search using the larger regions, and then determine how to get across each region in a subsequent step. This is very much akin to the spatial partitioning techniques you learn about in the Graphics Programming training series here at the Game Institute. Indeed, you should be able to reuse many of the data structures and algorithms (quad-trees, kd-trees, etc.) that you learn about in Graphics Programming Module II in order to accomplish this objective.

Some final optimizations to consider relate to the open and closed lists in particular. There are ways (using recursion) to totally eliminate both the open and closed lists. We will still encounter problems as we did with the Depth First Search, but we do not use the significant amount of memory that the open and closed lists use. Another method is to limit the number of nodes we will store in the open list, dropping the candidates with the highest *f* values if we reach our max open node count. By doing this, we do not need a closed list and this will save some processing time and memory. However, it is not guaranteed to find the best path.

2.2 Our Version of the Algorithm

```
MapGridWalker::WALKSTATETYPE AStarMapGridWalker::iterate()
{
	if(!m_open.isEmpty())
	{
		m_n = (AStarMapGridNode*)m_open.dequeue();
		if(m_n->equals(*m_end))
		{
			return REACHEDGOAL;
		}

		int x, y;

		// add all adjacent nodes to this node
		// add the east node...
		x = m_n->m_x+1;
		y = m_n->m_y;
		if(m_n->m_x < (m_grid->getGridSize() - 1))
		{
			visitGridNode(x, y);
		}

		// Check the rest of the directions here
		// see the code for details

		// add the north-east node...
		x = m_n->m_x+1;
		y = m_n->m_y-1;
		if(m_n->m_y > 0 && m_n->m_x < (m_grid->getGridSize() - 1))
		{
			visitGridNode(x, y);
		}

		m_closed.enqueue(m_n);

		return STILLLOOKING;
	}

	return UNABLETOREACHGOAL;
}
```

```
void AStarMapGridWalker::visitGridNode(int x, int y)
{
	int newg;

	// if the node is blocked or has been visited, early out
	if(m_grid->getCost(x, y) == MapGridNode::BLOCKED)
		return;

	// we are visitable
	newg = m_n->m_g + m_grid->getCost(x, y);

	if( (m_open.contains(&m_nodegrid[x][y])
```

```
              || m_closed.contains(&m_nodegrid[x][y]))
              && m_nodegrid[x][y].m_g <= newg)
       {
              // do nothing... we are already in the queue
              // and we have a cheaper way to get there...
       }
       else
       {
              m_nodegrid[x][y].m_parent = m_n;
              m_nodegrid[x][y].m_g = newg;
              m_nodegrid[x][y].m_h = goalEstimate( &m_nodegrid[x][y] );
              m_nodegrid[x][y].m_f = m_nodegrid[x][y].m_g
                                     + m_nodegrid[x][y].m_h;

              if(m_closed.contains(&m_nodegrid[x][y]))
                     m_closed.remove(&m_nodegrid[x][y]); // remove it

              if(!m_open.contains(&m_nodegrid[x][y]))
                     m_open.enqueue(&m_nodegrid[x][y]);
              else
              {
                     // update this item's position in the
                     // queue as its cost has changed
                     // and the queue needs to know about it
                     m_open.remove(&m_nodegrid[x][y]);
                     m_open.enqueue(&m_nodegrid[x][y]);
              }
       }
}
```

Here is the actual implementation from our demo. Similar to Dijkstra's Method, it makes use of the priority queue to keep our nodes sorted in order of cost. We grab the top node off our queue, see if it is traversable, update all its neighbors, and continue on until we find the goal node. Let us take a closer look at this implementation.

```
MapGridWalker::WALKSTATETYPE AStarMapGridWalker::iterate()
```

As with our other implementations, our iterate interface begins inside the outer while loop of our algorithm snippet. It also returns the state of the graph traversal, informing the caller if it has found the goal, is unable to find the goal, or requires more iterations.

```
       if(!m_open.isEmpty())
```

We begin by checking to see if our open queue is empty. If the queue is empty, we cannot find a path from the given start node to the goal node.

```
              m_n = (AStarMapGridNode*)m_open.dequeue();
              if(m_n->equals(*m_end))
              {
                     return REACHEDGOAL;
              }
```

For each iteration, we grab the node with the lowest perceived cost to the goal and make it our current node. If this node is the goal, we return success.

```
            // add all adjacent nodes to this node
            // add the east node...
            x = m_n->m_x+1;
            y = m_n->m_y;
            if(m_n->m_x < (m_grid->getGridSize() - 1))
            {
                  visitGridNode(x, y);
            }
```

We then visit each neighbor of the current node. Again we check to ensure that the node we want to visit is within the bounds of our grid. Also, we call upon visitGridNode to do the work of visiting the neighbor node.

```
void AStarMapGridWalker::visitGridNode(int x, int y)
```

As in previous implementations, visitGridNode does the work of visiting the grid node at the given (x, y) coordinate.

```
      // if the node is blocked or has been visited, early out
      if(m_grid->getCost(x, y) == MapGridNode::BLOCKED)
            return;
```

First, it determines if the grid node to be visited is blocked, and if it is, it returns without visiting the node.

```
      newg = m_n->m_g + m_grid->getCost(x, y);
```

If the grid node is not blocked, we compute the new actual cost to the node via the current node.

```
      if( (m_open.contains(&m_nodegrid[x][y])
            || m_closed.contains(&m_nodegrid[x][y]))
            && m_nodegrid[x][y].m_g <= newg)
```

If the open queue or the closed list contains the node already, and the new cost is higher than the cost already computed for this child node, we skip this node since we already have a path to this child node which is shorter. Otherwise, we will want to visit this node.

```
            m_nodegrid[x][y].m_parent = m_n;
            m_nodegrid[x][y].m_g = newg;
            m_nodegrid[x][y].m_h = goalEstimate( &m_nodegrid[x][y] );
            m_nodegrid[x][y].m_f = m_nodegrid[x][y].m_g
                                 + m_nodegrid[x][y].m_h;
```

Once it is determined that the child node needs visiting, its parent is set to be the current parent so that we know how we got here. We also set its actual cost to be the newly computed actual cost. Next, we estimate the distance to the goal using our goal estimate (exactly like we did in Best First Search).

Lastly, we compute the total cost for this node by summing the actual cost with the estimate to the goal cost.

```
            if(m_closed.contains(&m_nodegrid[x][y]))
                  m_closed.remove(&m_nodegrid[x][y]); // remove it
```

After we have computed our costs, we check to see if the closed list contains this node. If it does, we remove it since we plan to visit it again.

```
            if(!m_open.contains(&m_nodegrid[x][y]))
                  m_open.enqueue(&m_nodegrid[x][y]);
            else
            {
                  // update this item's position in the
                  // queue as its cost has changed
                  // and the queue needs to know about it
                  m_open.remove(&m_nodegrid[x][y]);
                  m_open.enqueue(&m_nodegrid[x][y]);
            }
```

Last, we check to see if the open queue already contains this node. If the queue does not contain the node, we add it. If it does contain the node, we requeue it into its proper position by removing it and adding it back.

```
            m_closed.enqueue(m_n);
```

After we have visited all of the current node's neighbors, we add the current node to the closed list. This will keep us from visiting it again unless we find a shorter path to it.

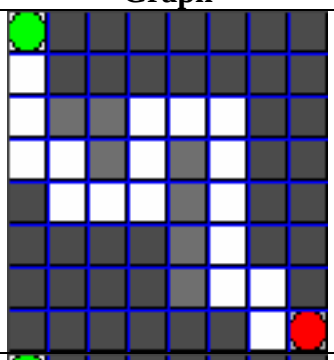
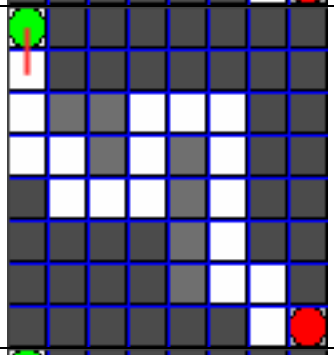
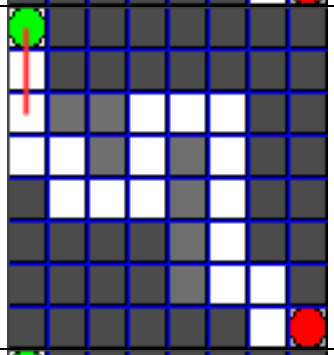
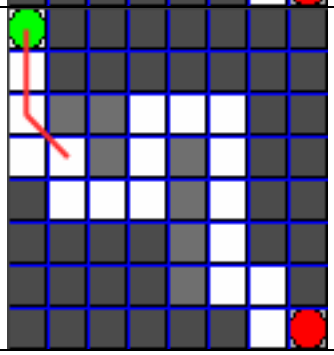
```
            return STILLLOOKING;
```

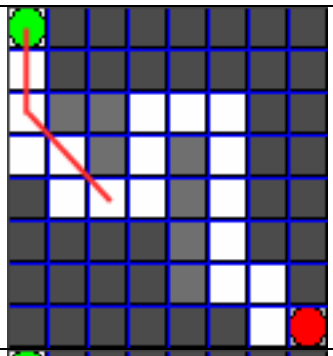
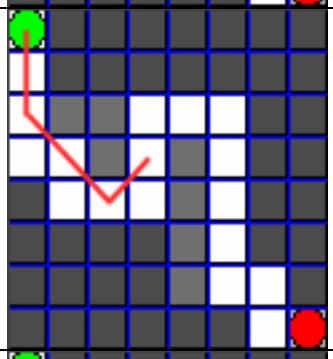
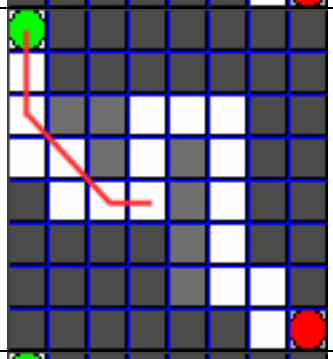
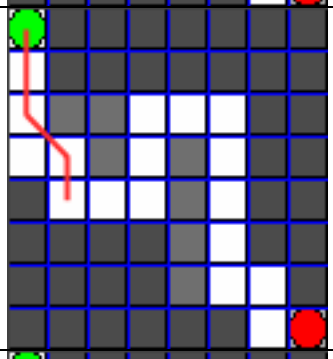
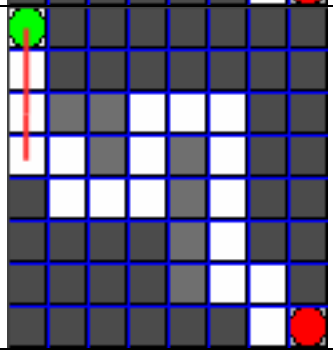
Finally, we return STILLLOOKING to ensure we get more iterations so we can find the goal node.

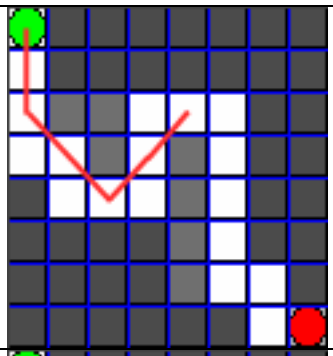
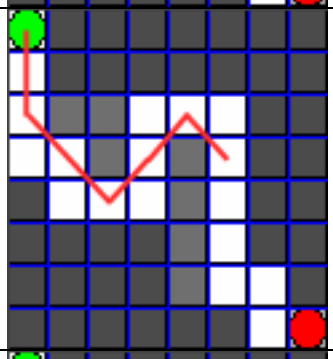
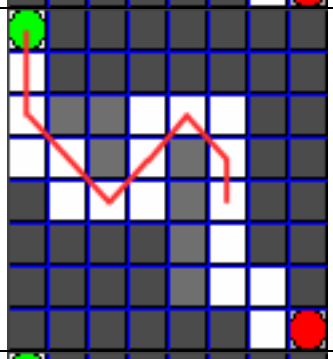
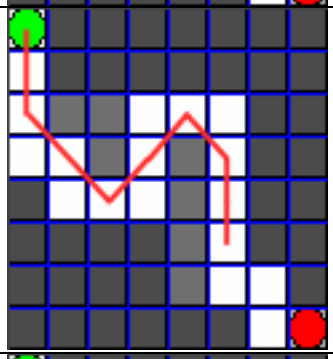
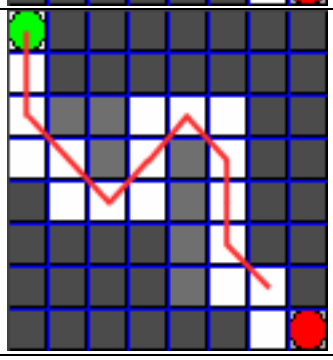
```
      return UNABLETOREACHGOAL;
```

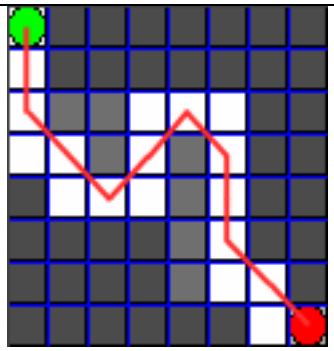
If our open queue was empty, we return UNABLETOREACHGOAL to ensure we cease iterating since we will never find the goal.

Now that we have obtained a deeper understanding of how A* and our implementation of A* works, let us take a moment to walk through our implementation of the algorithm with an example graph from our demo.

Graph	Open List	Closed List	Notes
	**(1,2)** **(2,2)** **(2,1)**	**(1,1)**	Here we have an example map where we want to get from corner to corner. The path is clear-cut, but pathfinding algorithms do not know that. Note that A* will arrive at the goal in 15 iterations rather than spend lots of time searching the expensive areas. By making the path free, you should see how the heuristic will provide for the solution sooner than Dijkstra's would.
	**(1, 3)** **(2, 3)** **(2, 2)** **(2, 1)**	**(1,1)** **(1,2)**	In the first iteration, the choice is simple. (1,2) was on the top of the list because it is free. The heuristic estimates were the same for all three possible nodes using Max(dx, dy). In the second iteration, the heuristic shows that going to (1,3) is cheapest (it is also free).
	**(2, 4)** **(1, 4)** **(2, 3)** **(2, 2)** **(2, 1)**	**(1, 1)** **(1, 2)** **(1, 3)**	(1.3)'s neighbors are put on the open list, and it is placed in the closed list. We see from our estimate that, regardless of which node we choose, the estimate is the same, but the cost is zero for (2, 4) and (1, 4). Since (2, 4) was added last it goes on top in this implementation.
	**(3, 5)** **(2, 5)** **(1, 4)** **(3, 3)** **(3, 4)** **(1, 5)**	**(1, 1)** **(1, 2)** **(1, 3)** **(2, 4)**	As we search (2, 4)'s neighbors, we see that either (3, 5) or (2, 5) is our best bet (as they are free). (3, 5) becomes the clear winner as it is added after (2, 5). Each step has still taken us closer to the goal so our heuristic has not done much for us yet.

	**(4, 4)** **(4, 5)** **(2, 5)** **(1, 4)** **(3, 3)** **(3, 4)** **(4, 6)** **(2, 6)** **(3, 6)** **(1, 5)**	**(1, 1)** **(1, 2)** **(1, 3)** **(2, 4)** **(3, 5)**	As we search (3, 5)'s neighbors, we see that we again have a tie between (4, 4) and (4, 5). (4, 4) gets added on top because it was searched last. Notice our open list is getting very long now.
	**(4, 5), (2, 5), (1, 4), (5, 3), (4, 3), (5, 5), (5, 4), (3, 3), (3, 4), (4, 6), (2, 6), (3, 6), (1, 5)**	**(1, 1)** **(1, 2)** **(1, 3)** **(2, 4)** **(3, 5)** **(4, 4)**	Here we see that our heuristic is going to make us backtrack a bit. We are now potentially headed upward, and "away" from our goal. We see we have nodes in our open list which do not go away from our goal so we search them first.
	**(2, 5), (1, 4), (5, 3), (4, 3), (5, 6), (5, 5), (5, 4), (3, 3), (3, 4), (4, 6), (2, 6), (3, 6), (1, 5)**	**(1, 1)** **(1, 2)** **(1, 3)** **(2, 4)** **(3, 5)** **(4, 4)** **(4, 5)**	We backtrack a bit and see that (4, 5) did not have any better ways to go. Again, we have some nodes in our open list with better heuristic values so we try them first.
	**(1, 4), (5, 3), (4, 3), (5, 6), (5, 5), (5, 4), (3, 3), (3, 4), (1, 6), (4, 6), (2, 6), (3, 6), (1, 5)**	**(1, 1)** **(1, 2)** **(1, 3)** **(2, 4)** **(3, 5)** **(4, 4)** **(4, 5)** **(2, 5)**	We still find no better paths here. There is one more node to check in our open list before we can get back to where we were before, so we check it next.
	**(5, 3), (4, 3), (5, 6), (5, 5), (5, 4), (3, 3), (3, 4), (1, 6), (4, 6), (2, 6), (3, 6), (1, 5)**	**(1, 1)** **(1, 2)** **(1, 3)** **(2, 4)** **(3, 5)** **(4, 4)** **(4, 5)** **(2, 5)** **(1, 4)**	There is still nothing better here. We go back to (5, 3) in the next iteration in order to see if we can get moving towards the goal again.

	**(6, 4), (6, 3) (4, 3), (5, 6), (5, 5), (5, 4), (3, 3), (3, 4), (4, 2), (6, 2), (5, 2), (1, 6), (4, 6), (2, 6), (3, 6), (1, 5)**	**(1, 1), (1, 2), (1, 3), (2, 4), (3, 5), (4, 4), (4, 5), (2, 5), (1, 4), (5, 3)**	Here we are headed back along our path again. We have (6, 4) and (6, 3) which we can try. (6, 4) was added last so we try it next.
	**(6, 5), (6, 3) (4, 3), (5, 6), (5, 5), (5, 4), (3, 3), (3, 4), (7, 3), (7, 5), (7, 4), (4, 2), (6, 2), (5, 2), (1, 6), (4, 6), (2, 6), (3, 6), (1, 5)**	**(1, 1), (1, 2), (1, 3), (2, 4), (3, 5), (4, 4), (4, 5), (2, 5), (1, 4), (5, 3), (6, 4)**	Our heuristic helps by declaring that (6, 5) is a better choice than (6, 3) as it is closer to the goal. We search there next.
	**(6, 6), (6, 3) (4, 3), (5, 6), (5, 5), (5, 4), (3, 3), (3, 4), (7, 6), (7, 3), (7, 5), (7, 4), (4, 2), (6, 2), (5, 2), (1, 6), (4, 6), (2, 6), (3, 6), (1, 5)**	**(1, 1), (1, 2), (1, 3), (2, 4), (3, 5), (4, 4), (4, 5), (2, 5), (1, 4), (5, 3), (6, 4), (6, 5)**	Now we are moving nicely towards the goal. We see that our next best step is (6, 6) so we move in that direction.
	**(7, 7), (6, 7), (6, 3), (4, 3), (5, 7), (5, 6), (5, 5), (5, 4), (3, 3), (3, 4), (7, 6), (7, 3), (7, 5), (7, 4), (4, 2), (6, 2), (5, 2), (1, 6), (4, 6), (2, 6), (3, 6), (1, 5)**	**(1, 1), (1, 2), (1, 3), (2, 4), (3, 5), (4, 4), (4, 5), (2, 5), (1, 4), (5, 3), (6, 4), (6, 5), (6, 6)**	We now have a choice between (7, 7) and (6, 7). (7, 7) is chosen as it was added last via the search. Remember that Max(dx, dy) will not pick a node that is both closer in x and y; only the one that is closer of the two. This can result in a lot of ties as you can see from this example.
	**(8, 8), (7, 8), (6, 7), (6, 3), (4, 3), (5, 7), (5, 6), (5, 5), (5, 4), (3, 3), (3, 4), (8, 6), (8, 7), (6, 8) (7, 6), (7, 3), (7, 5), (7, 4), (4, 2), (6, 2), (5, 2), (1, 6),**	**(1, 1), (1, 2), (1, 3), (2, 4), (3, 5), (4, 4), (4, 5), (2, 5), (1, 4), (5, 3), (6, 4), (6, 5), (6, 6), (7, 7)**	We are nearing the end. We see that (8, 8) is closest to the goal (and is the goal), so we go there next. The next iteration ends the search.

	(4, 6), (2, 6), (3, 6), (1, 5)		
	**(8, 8), (7, 8), (6, 7), (6, 3), (4, 3), (5, 7), (5, 6), (5, 5), (5, 4), (3, 3), (3, 4), (8, 6), (8, 7), (6, 8) (7, 6), (7, 3), (7, 5), (7, 4), (4, 2), (6, 2), (5, 2), (1, 6), (4, 6), (2, 6), (3, 6), (1, 5)**	**(1, 1), (1, 2), (1, 3), (2, 4), (3, 5), (4, 4), (4, 5), (2, 5), (1, 4), (5, 3), (6, 4), (6, 5), (6, 6), (7, 7)**	**Our implementation does not remove the last node from the open list after we arrive at the goal. We simply quit and make the path. Notice how many nodes are still in the open list. Even with a small grid, the list becomes very large very quickly. An efficient container class for this list is highly recommended for optimal use on larger graphs.**

2.3 Heuristics: A Few Examples

Let us take a moment to discuss heuristic estimates with A* and the multitude of ways they can be used.

The greatest advantage of A* is its heuristic estimate, which is used to modify and optimize its search. At its heart, A* is simply a Breadth First Search without the heuristic. But with the heuristic, you can control how A* performs its search, and which nodes it chooses to search first. For example, the heuristic can be as simple as a cost estimate to the goal, or it can be the cost estimate to the goal coupled with performance penalties or bonuses for traveling on a specific type of terrain. You can use the state of the object (velocity, momentum, accelerations) to determine the quality or suitability of a node. With the appropriate heuristic, A* may even be used to solve the little puzzle game where you push the numbers around until they are in sequence. A* has even been used with performance heuristics to determine the shortest yet safest way to do load balancing on multi-server networked systems. The power of the heuristic is not to be underestimated.

2.4 A Simple Real Time Strategy Game Design

In order to examine how A*'s behavior can be modified by the heuristic, let us consider a very simple real time strategy game design. To start, some units and some basic terrain types will be defined. We will then define a heuristic which will determine a cost multiplier for the standard heuristic estimate (such as max(dx, dy)) and define how each unit is effected by a given terrain type.

2.4.1 Terrain Types

Jungle

Jungle terrain consists of varied elevations mixed with dense tropical vegetation. It is very demanding and nearly impossible to traverse other than by foot.

Forest

Forest terrain consists of regions of land lightly-to-densely covered with coniferous and deciduous trees, as well as various types of underbrush. While not as demanding as jungle terrain, larger vehicles are unable to traverse this terrain due to tree spacing and underbrush.

Plains

Plains terrain consists of flat land to gently rolling hills covered with rowed crops and grasses. This terrain is fairly forgiving and is easily traversed by most modes of transportation.

Desert

Desert terrain can be anything from steppes to rolling dunes. This terrain is typically not restrictive to vehicles but may be a bit more costly for those on foot due to heat.

Foothills

Foothill terrain consists of rolling hills through small canyons. The terrain is typically very rocky with uneven ground making travel difficult for most, and nearly impossible for large vehicles.

Mountains

Mountain terrain consists of very steep slopes and rocky uneven ground. This terrain is considered impassible to all but those on foot and, even so, is still very difficult to traverse.

Roadway

Roadways are dirt, gravel, or paved surfaces which are wide enough for large vehicles to travel along, though possibly only one lane at a time. They provide for very easy travel, though they may not go in the direction desired.

Trail

Trails are dirt paths which are sufficiently cleared for those on foot or small vehicles to use for traveling more quickly. They are typically winding in nature which may make them unsuitable for exclusive use.

Swamp

Swamp terrain consists of wetlands and fairly dense undergrowth. The ground itself is soft which mires vehicles, especially heavy ones. Travel through this terrain is slow but possible.

Water

Water terrain is any type of body of water, whether a river, lake, or ocean. Streams and small bodies of water contained within a terrain are not included as they are typically easily fordable or avoidable without much added cost.

2.4.2 Units

For now, let us define four separate types of units, defined by their primary mode of motion. We will have Infantry, Wheeled Vehicles, Tracked Vehicles, and Hovercraft. We then have two more specific unit types for each generic type of unit, with exception of the Hovercraft type.

Infantry

Infantry are soldiers traveling on foot. Being on foot gives them excellent maneuverability. They can traverse all types of terrain aside from water with little impediment, although trails, roadways and other non-varied terrain are preferred. Light infantry carries light backpacks and arms, thereby making them capable of moving around easily. Heavy infantry carries large backpacks, shoulder mounted rocket launchers, or other similar encumbrances. Due to such heavy equipment, they are incapable of traversing jungles, mountains, and swampy terrain.

Wheeled Vehicles

Wheeled vehicles are vehicles that use wheels on axels. Being mostly lighter and smaller vehicles, they are capable of traversing terrains such as lightly forested areas, plains, deserts, foothills, and using normal roadways, as well as general footpaths and trails. Our two types of units are Jeeps and Armored Personnel Carriers. Jeeps are the smaller and lighter of the two and are more capable of traversing dense terrain. APC's, on the other hand, are heavier and larger vehicles, rendering them incapable of using trails, and limiting their ability to function in the varied terrain of foothills.

Tracked Vehicles

Tracked vehicles are vehicles that maneuver by use of linked treads running over many sets of wheels. This gives them tremendous amounts of traction and surface area allowing heavier chassis. This also limits their maneuverability, thereby restricting them from swamps, jungles, and the use of trails. The two tracked vehicles we have defined are tanks and mobile base units. Tanks are capable of traversing forested areas, plains, deserts, and foothills, as well as using roadways. Mobile base units are very large in size rendering them incapable of traversing forested areas or foothills.

Hovercraft

Hovercraft are vehicles that move around by riding a cushion of air produced by a large fan or turbine blowing down toward the ground. The air is trapped by a skirt around the vehicle which makes close contact with the ground and forms a seal. This seal keeps the air trapped under the vehicle, thereby allowing it to move about. If the seal is broken, the vehicle is immobilized until the seal can be reformed, which limits its movement to areas without dense vegetation or uneven terrain.

Below we have a table showing which units are capable of traversing which terrain types. A check means the unit is capable of traversing the terrain (although possibly at great cost). An x means the vehicle is incapable of traversing the terrain at any cost.

Unit\Terrain	Jungle	Forest	Plains	Desert	Foothills	Mountains	Roadway	Trail	Swamp	Water
Light Infantry	☑	☑	☑	☑	☑	☑	☑	☑	☑	☒
Heavy Infantry	☒	☑	☑	☑	☑	☒	☑	☑	☒	☒
Jeep	☒	☑	☑	☑	☑	☒	☑	☑	☒	☒
APC	☒	☑	☑	☑	☒	☒	☑	☒	☒	☒
Tank	☒	☑	☑	☑	☑	☒	☑	☒	☒	☒
Mobile Base	☒	☒	☑	☑	☒	☒	☑	☒	☒	☒
Hovercraft	☒	☒	☑	☑	☒	☒	☑	☒	☑	☑

2.4.3 Terrain Type vs. Unit Type Weighting Heuristic

In the example real time strategy game we are designing, we defined seven different types of units and ten different types of terrain. We will now define a Terrain Type versus Unit Type weighting heuristic. For each unit, we will store the cost weight of traversing each type of terrain. This cost weight will be used as a multiplier of the heuristic estimate. Any single node cost over a limit will be considered blocked and impassible terrain. If we wanted to be even more advanced, we could specify a unit task and the task would choose the type of terrain best suited for the task. For instance, if a unit were in reconnaissance mode, it would prefer terrain types that are more suited to stealth.

Below we have a table of proposed unit type terrain modifiers. This is completely arbitrary but should give you an idea of how it would work. Ideally, you would adjust these numbers to make the units move in the fashion desired.

Unit\Terrain	Jungle	Forest	Plains	Desert	Foothills	Mountains	Roadway	Trail	Swamp	Water
Light Infantry	3.0	1.5	1.2	1.8	1.5	3.0	1.0	1.0	1.5	100.0
Heavy Infantry	100.0	2.0	1.3	2.0	2.0	100.0	1.0	1.0	100.0	100.0
Jeep	100.0	1.5	1.1	1.2	1.5	100.0	1.0	1.0	100.0	100.0
APC	100.0	1.8	1.1	1.2	100.0	100.0	1.0	100.0	100.0	100.0
Tank	100.0	2.0	1.0	1.1	1.3	100.0	1.0	100.0	100.0	100.0
Mobile Base	100.0	100.0	1.2	1.2	100.0	100.0	1.0	100.0	100.0	100.0
Hovercraft	100.0	100.0	1.3	1.3	100.0	100.0	1.0	100.0	1.2	1.1

Using our newly defined set of weights, we can define our heuristic estimate function. We will use whichever heuristic estimate function we choose (Max, Manhattan, Euclidean), and multiply it by our weight for this unit on this terrain. This gives us the function:

$$h(n) = h'(n) \cdot W_{u,t},$$

where $h'(n)$ is our duly appointed heuristic estimate, and W is the matrix of weights for each unit on each terrain. Bear in mind that modifying only the heuristic will not make the system work as if by magic. We will need to apply similar weights to the actual costs of the nodes for the terrain and for the given units, or else the heuristic estimates will be significantly overestimated.

2.4.4 Defining the Map

At this point, if you were going to start assembling your game, you have a set of units, a set of terrains, weights for each unit for each terrain type, and a heuristic to use those weights. You now need to define your map. The best choice in this case is probably going to be a grid-oriented set of points that allow travel in all of the cardinal directions as well as diagonally. A 2D bitmap texture might serve you nicely here, where each texel equals a node. Start by setting the weights for travel between each node to be 1 for all nodes, and let the heuristic decide the nodes on which to travel. You will have to apply the weights to

the nodes themselves as we traverse them because the actual cost to the node is what drives A* to search for a short path. You will then assign a terrain type to each node, perhaps using a paint program if a texture is your map method of choice. Storing the weights can be done using a text file (e.g., an .ini file) or you can hard-code the values directly into your application. Finally, you need to create several units and instruct them to wander from place to place. Once done, you will have the beginnings of a real time strategy game of your very own! The artificial intelligence required to make our units perform intelligently is a topic to be addressed later in this course. But for now, you should be able to put together something simple, resembling the features discussed here.

2.5 Simplifying the Search: Hierarchical Pathfinding

Hierarchical pathfinding is an approach to pathfinding which attempts to reduce the number of nodes the pathfinding algorithm has to consider when building a path. The concept is simple in theory, and not much more difficult in practice. Start by breaking down the gaming area into sub-areas. For each sub-area, break it up into further sub-areas. Repeat this process until it does not make sense to break up the sub-areas any further. The algorithm is then modified to find its way from the source to the goal via the lower resolution sets, then via the higher resolution sets included in the path found across the lower resolution sets. This does not always provide the absolute best path, but it helps reduce a larger problem into several smaller and more manageable ones. Let us look at a few examples to see how this works.

2.5.1 A Map of the US

Figure 2.1

Let us say we have a map of the United States, and we want to find a path from one city to another. We will break up the United States first into states and then into counties. We first find the state our starting and ending cities are in, and find our way from state to state. We then determine our path across each of the states we know need to be crossed, one at a time, going from county to county.

Suppose we wanted to take a trip from Chicago to Las Vegas. Figure 2.1 shows the need to go from Illinois, through Iowa, through Nebraska, through Colorado, through Utah, and into Nevada. It would be terribly inefficient to calculate a path using all of the roads of all of the states. We begin by determining the best way to get to Iowa first, and from there to Nebraska, and from there to Colorado, and from there to Utah, and finally to Nevada. You can see how this limits the scope of our search, and also allows us to spread the processing of the search across many game cycles, as we do not need to worry about getting across the next state until we have already traversed the prior state.

2.5.2 A Dungeon

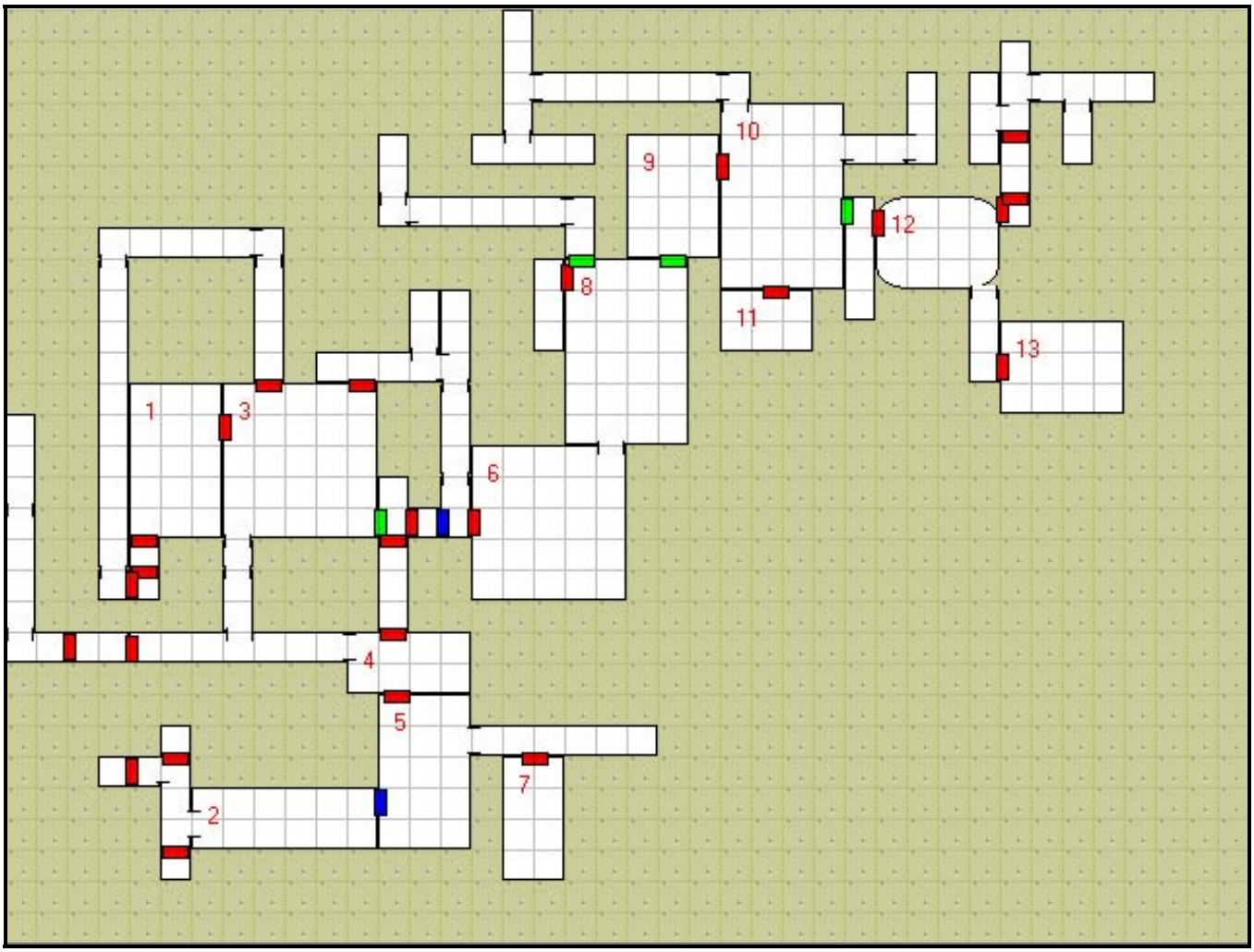


Figure 2.2

Another example is a dungeon with multiple rooms and levels. The dungeon could be broken up into levels first, and then broken down further into rooms on each level. If we wanted to get from one room in one level to another room in another level, we would first find out which levels we would need to get through. Then we would determine which rooms we needed to pass through for each of those levels. Finally, we find out how to get across each of those rooms. Take, for example, Figure 2.2. This dungeon has many rooms. If each room were to have various obstacles and pillars around which we needed to navigate, you might think that it would make sense to figure out how to exit the room before worrying about how to get all the way across the dungeon. But before you can concentrate on finding a path from your current location to the door, you first have to figure out which door is the most appropriate one to start from. In some implementations, you might decide to simply let the pathfinder work its way back to the correct door and then use only the game engine's collision system to navigate from the current position to the selected door. In most cases however, a pathfinder is be used even for this task (albeit in conjunction with the collision engine). Regardless of how you implement the close distance navigation, the larger point here is the breakdown of navigation tasks from low resolution to high resolution.

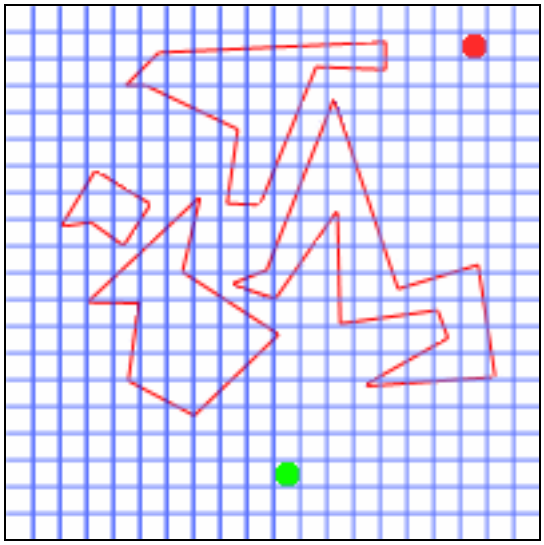
2.5.3 A Real Time Strategy Map

For a real time strategy game map, we could divide the map into sixteen squares. We would then divide each of the sixteen squares into sixteen squares again, and repeat this breakdown until we had squares of small enough size that we could quickly traverse them. We would then traverse the largest squares to determine which ones needed to be crossed, and then determine which smaller squares would need to be crossed from that point, and so on. This method is very much like the quad-tree method you will study in the Graphics Programming course series here at the Game Institute, so you might wish to apply some of that knowledge to tackling this problem.

2.6 Pathfinding on Non-Gridded Maps

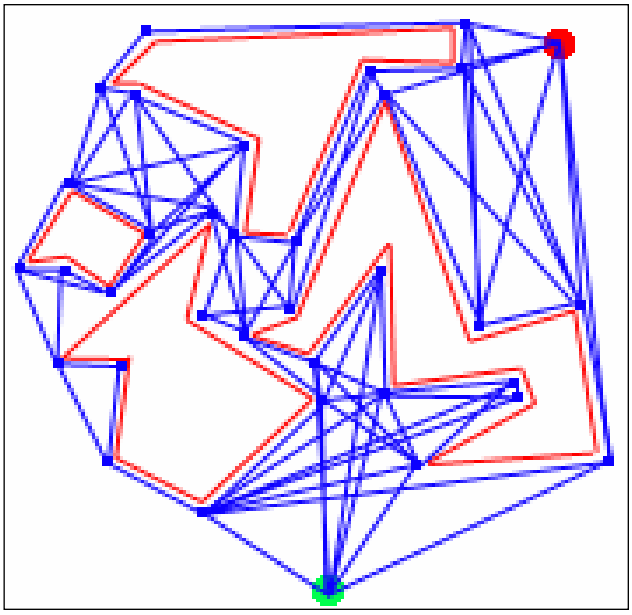
In the world of gaming, it cannot always be assumed that we are dealing with maps that are based on grid systems. Yet we still must find our way around the maps. While there are many ways to accomplish this, let us cover some of the most common methodologies for traveling from one place to another in worlds where there are no pre-defined natural grids. Usually in non-gridded environments, next step closer techniques are utilized to move around locally while dodging around and fighting, and the higher level pathfinding is only used when trying to go longer distances. In these cases, it is typical to "acquire" the larger pathfinding graph by getting to the closest node, and then pathfinding to the closest node to your destination. Once you acquire the closest point to the goal on your larger pathfinding graph, you fall back to next step closer techniques to access the actual goal position.

2.6.1 Superimposed Grids



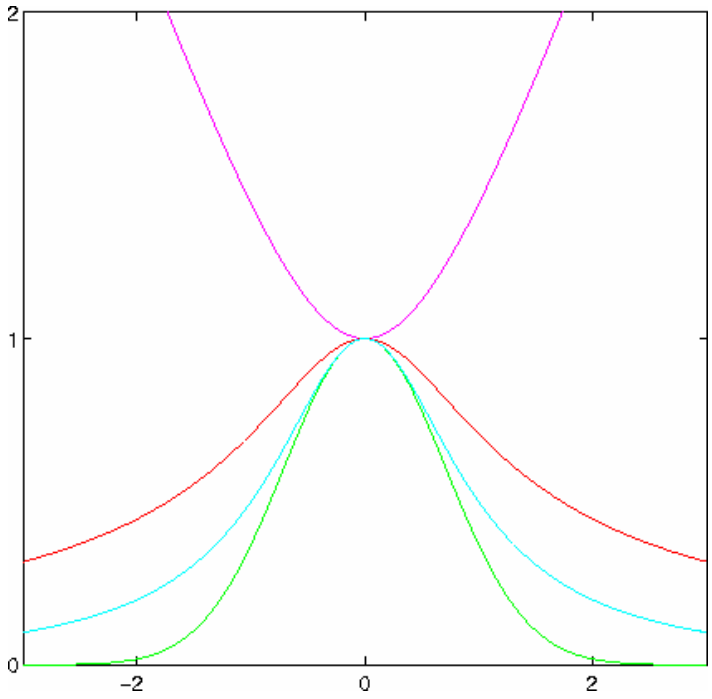
One solution is to make the non-gridded world a gridded world. To do this, a grid is superimposed over the gaming area, which is where our pathfinding will be done. For multi-level systems, we can create grids for each level, and define entry and exit points to move from one level to another.

2.6.2 Visibility Points / Waypoint Networks



Visibility points are a common way of determining where obstacles are in order to avoid them. The idea is to place points around the obstacles, and draw lines from each point to every other point such that the lines do not cross through any obstacles. These points are then used by the pathfinding algorithm to determine where you can walk. These systems are also referred to as **waypoint networks**. The finer the network of points, the finer the movements your entities will have. Coarse networks result in jagged paths and zig-zagging. We will look at waypoint networks in more detail a little later in the course as they will be our method of choice for 3D world navigation.

2.6.3 Radial Basis

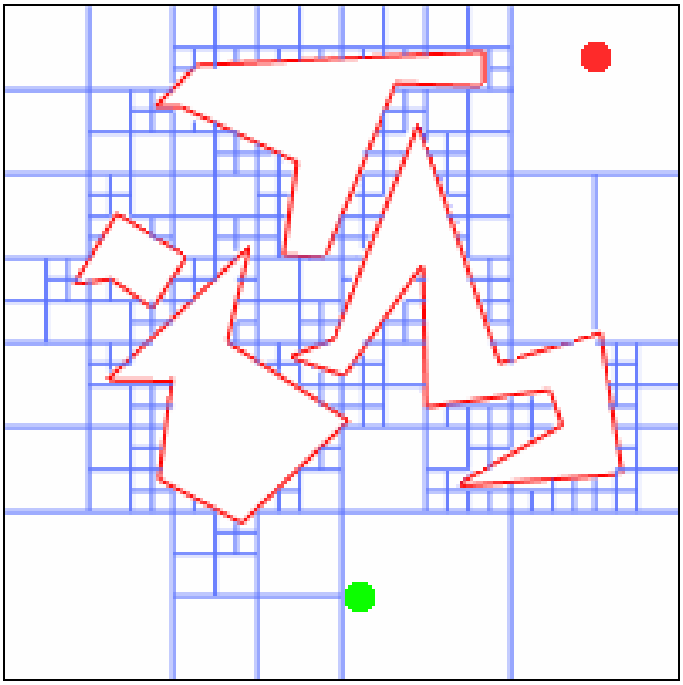


Radial Basis functions are functions that look similar to a normal distribution curve. Centering one of these functions on each of the obstacles allows the pathfinding algorithm to determine distances to obstacles and incur higher cost as it gets closer to an obstacle. The algorithm can then travel in the direction that induces the least amount of cost. These types of systems tend to be more expensive since the radial basis function contains an e^x (exponential) expression.

2.6.4 Cost Fields

Similar to the radial basis method, this method surrounds obstacles with cost fields. Cost fields are typically implemented using continuous functions, and the pathfinding method simply uses gradient descent or the Newton-Rhapson method to find the lowest cost and travel in that direction. The problem with this method is that it can get caught in local minima, requiring some sort of agitation method to get back out. Moreover, like the Radial Basis method, this method can be computationally expensive.

2.6.5 Quad-Trees



This method is a combination of hierarchical pathfinding and the grid method. The area is cut into quads, and then each of those quads is cut into quads. The largest quad that can be formed without crossing a boundary of an obstacle is then created and stored. Recursion occurs to some depth which limits the number of nodes, but also provides fine pathfinding near obstacles. The centers and corners of the squares are used as route points. For more information about creating quad-trees, please consult the Graphics Programming Module II course available here at the Game Institute.

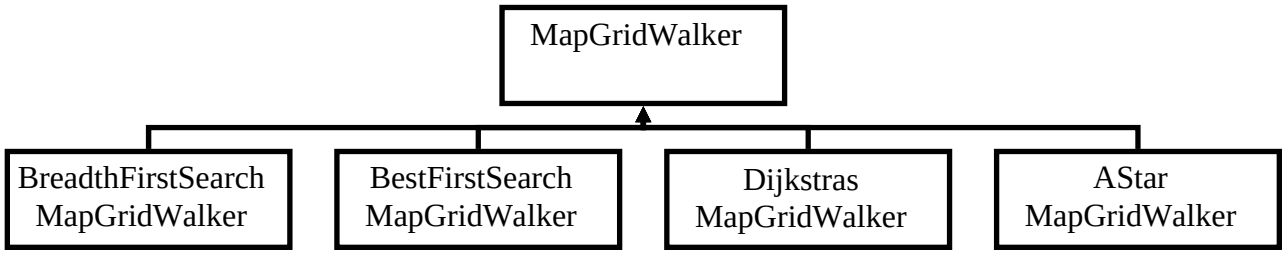
2.6.6 Mesh-Based Navigation

For 3D worlds consisting of polygonal data, graphs can be built from the mesh data itself. This is a tricky method, and requires a fair amount of work on the part of both the artist/level designer and the programmer, but it allows the use of world geometry as your pathfinding graph. The idea is to define the polygons in your world that are 'floor-walkable', and build an adjacency list for each one. In this way, it can be determined which floor polygon can be traversed to reach another traversable floor polygon. Walls and other obstacle polygons will not be included in the adjacency list. Additionally, as other moving entities cross the polygons, the larger polygons can be dynamically tesselated and the ones they are standing on removed so that they can be circumnavigated. Once the entity moves off the polygon, it can be re-added into the adjacency lists and remerged if all of its pieces are available again. This is a fairly complex system, and will not be demonstrated in this course. Our preference for 3D world navigation in this course will be waypoint networks and we will discuss those techniques a little later in our studies.

2.7 Algorithm Design Strategy

In order to allow the application to show each step of the pathfinding algorithm's decision making process, the algorithms need to be designed so that an iteration function is repeatedly called which updates the display between steps, rather than finding the entire path in a loop. This method is useful because it also allows the update rate of the algorithm to be changed dynamically, thereby allowing quicker or slower playback of the graph search. Of course, your actual game implementation may not have this one step at a time design requirement, but it is easy enough to modify the approach to generate either the complete path in one pass or to do *n* iterations before returning.

2.7.1 Class Hierarchy



2.7.2 MapGridWalker Interface

The class hierarchy is designed around the base class MapGridWalker. This class provides the basic interface common to all MapGridWalker types. It tells us if weighted graphs are supported, whether or not heuristics are supported, what types of heuristics are supported, and so on. In our demo, there are two classes which support weighted graphs, and two classes which support heuristics. This architecture provides for the ability to add heuristics, as well as add new types of MapGridWalkers, simply and easily.

```
typedef std::vector<std::string> stringvec;

class MapGridWalker
{
public:

	typedef enum WALKSTATE
		{ STILLLOOKING, REACHEDGOAL, UNABLETOREACHGOAL } WALKSTATETYPE;

	MapGridWalker();
	MapGridWalker(MapGrid* grid) { m_grid = grid; }
	virtual ~MapGridWalker();
```

```
	virtual void drawState(CDC* dc, CRect gridBounds) = 0;
	virtual WALKSTATETYPE iterate() = 0;
	virtual void reset() = 0;
	virtual bool weightedGraphSupported() { return false; };
	virtual bool heuristicsSupported() { return false; }
	virtual stringvec heuristicTypesSupported()
		{ stringvec empty; return empty; }

	virtual std::string getClassDescription() = 0;

	void setMapGrid(MapGrid *grid) { m_grid = grid; }
	MapGrid *getMapGrid() { return m_grid; }

protected:
	virtual void visitGridNode(int x, int y) = 0;

	MapGrid *m_grid;

};
```

The MapGridWalker interface is fairly straightforward. Let us go over it a bit at a time.

```
	typedef enum WALKSTATE
		{ STILLLOOKING, REACHEDGOAL, UNABLETOREACHGOAL } WALKSTATETYPE;
```

MapGridWalker defines the WALKSTATE enumeration, which provides the application with an understanding of the walker's progress. STILLLOOKING informs the application whether it must call iterate again to keep looking for the goal. REACHEDGOAL informs the application that the goal has been reached and iterate need not be called again. UNABLETOREACHGOAL informs the application that the walker has failed to find a path to the goal and further calls to iterate will not make any progress.

```
	MapGridWalker();
	MapGridWalker(MapGrid* grid) { m_grid = grid; }
```

MapGridWalker supports a default constructor as well as a constructor which supplies a MapGrid upon which to walk. Accessors to set the MapGrid are also provided so the default constructor can be used.

```
	virtual void drawState(CDC* dc, CRect gridBounds) = 0;
```

The virtual drawState() function allows the specific implementation of the MapGridWalker to draw its current state into a Windows Device Context (CDC) using the bounds of the window provided in the rect gridBounds. This allows walker specific drawing to be done in an object oriented fashion.

```
	virtual WALKSTATETYPE iterate() = 0;
```

The virtual iterate() method makes one iteration of the walker's algorithm before returning a value corresponding to its current state (using the enum).

```
	virtual void reset() = 0;
```

The virtual reset() method resets the walker's state to the start node and reinitializes its map grid to mark all nodes as not visited.

```
	virtual bool weightedGraphSupported() { return false; };
```

The virtual weightedGraphSupported() method defaults to false, but the specific implementation may return true if the walker supports navigation of weighted graphs.

```
	virtual bool heuristicsSupported() { return false; }
```

The virtual heuristicsSupported() method also defaults to false, but may be overloaded in the specific class to return true if the walker supports the use of heuristics.

```
	virtual stringvec heuristicTypesSupported()
```

The heuristicTypesSupported() method returns an empty vector of strings in the default implementation, but may be overloaded by a walker that supports heuristics to provide a vector of strings that contain the descriptions of the heuristics supported. This vector is used by the application to populate the heuristics dropdown.

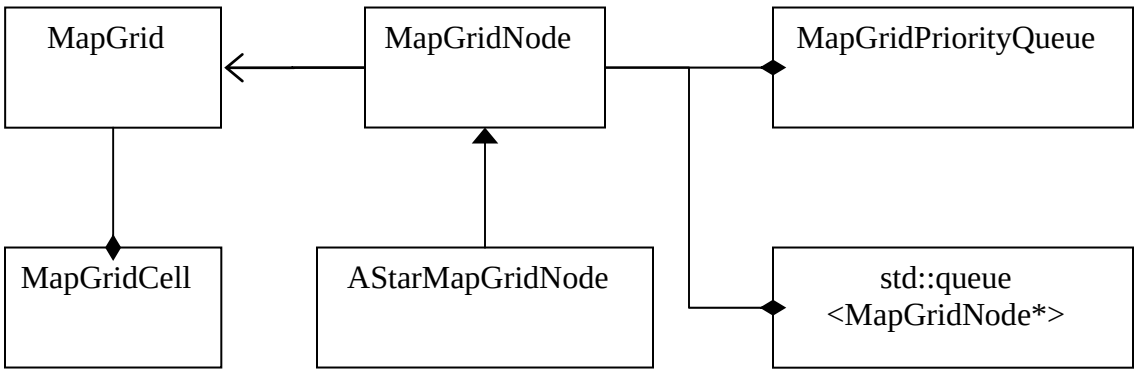
```
	void setMapGrid(MapGrid *grid) { m_grid = grid; }
	MapGrid *getMapGrid() { return m_grid; }
```

There are some accessor methods to obtain the specific walker class's name (to populate the pathfinding method dropdown), and set or get the MapGrid object which the walker will navigate.

```
	virtual void visitGridNode(int x, int y) = 0;
```

The virtual method visitGridNode allows derived MapGridWalkers to visit the grid node at the given coordinate in its own specific fashion.

2.8 Grid Design Strategy



The MapGrid is simply a two-dimensional array of MapGridCell objects, which is a nested class of MapGrid. Each of these cells has a cost associated with it for moving into them. The MapGrid class also keeps track of the start and end indices into the grid for the walkers to use in their search. The walkers themselves build a PriorityQueue or a STL queue which contains MapGridNode objects. MapGridNode objects keep track of which MapGridCell they represent by storing the row and column index into the MapGrid. MapGridNode objects also keep track of state information for the walker class such as current traversal cost, visited state, and so forth. Walker specific MapGridNodes can easily be subclassed from MapGridNode as in the instance of the AStarMapGridNode, since A* requires more cost state variables to be stored than the other methods discussed. These special-case MapGridNodes may also be used for other special-case walkers to allow for extensibility. The MapGridNode is segregated from the MapGrid itself so that the grid can be discarded and replaced with another type of map environment. Note however, that the walkers themselves must be rewritten to use different map types; hence, their derivation from MapGridWalker, which is used to walk MapGrids.

2.8.1 MapGrid Interface

```
class MapGrid
{
public:
	class GridCell
	{
	public:
		GridCell();
		GridCell(int cost);
		GridCell(const GridCell& copy);
		GridCell &operator=(const GridCell& rhs);

		inline int getCost() const { return m_cost; }
		void setCost(const int cost) { m_cost = cost; }
	private:
		int m_cost;
	};

	MapGrid(int gridsize);
	virtual ~MapGrid();

	int  getGridSize() const { return m_gridsize; }
	int  getCost(int x, int y) const;
	void setCost(int x, int y, const int cost);

	void setStart(int  x, int y)        { m_startx = x; m_starty = y; }
	void getStart(int &x, int &y) const { x = m_startx; y = m_starty; }
	void setEnd  (int  x, int y)        { m_endx = x; m_endy = y; }
	void getEnd  (int &x, int &y) const { x = m_endx; y = m_endy; }

private:
	GridCell **m_grid;
	int m_gridsize;
	int m_startx, m_starty, m_endx, m_endy;
};
```

The MapGrid class is fairly simple. It contains a nested class GridCell, which contains only the cost needed to enter that cell during a traversal. The MapGrid itself consists of a two-dimensional array of GridCell objects, and the row and column indices to the start and end nodes. The MapGrid class has various accessors for setting and getting the cost of a given GridCell, the start node, and the end node. Let us talk about it in a little more depth.

```
            GridCell();
            GridCell(int cost);
            GridCell(const GridCell& copy);
```

The contained class GridCell has a default constructor which initializes the m_cost variable to 1, as well as a constructor which assigns the cost passed in. There is also a copy constructor for deep copies.

```
            GridCell &operator=(const GridCell& rhs);
```

The assignment operator helps prevent copy constructs and is also useful for general assignment.

```
            inline int getCost() const { return m_cost; }
            void setCost(int cost) { m_cost = cost; }
```

Accessors provide access to the m_cost variable of the GridCell.

```
      private:
            int m_cost;
```

The cost value is encapsulated and requires the accessors to gain access.

```
      MapGrid(int gridsize);
      virtual ~MapGrid();
```

The MapGrid itself provides only a constructor, which expects a grid size used to construct a two dimensional array of GridCell objects. The destructor is virtual in order to allow potential inheritance and proper polymorphic destruction.

```
      int  getGridSize() const { return m_gridsize; }
      int  getCost(int x, int y) const;
      void setCost(int x, int y, const int cost);

      void setStart(int  x, int y)        { m_startx = x; m_starty = y; }
      void getStart(int &x, int &y) const { x = m_startx; y = m_starty; }
      void setEnd  (int  x, int y)        { m_endx = x; m_endy = y; }
      void getEnd  (int &x, int &y) const { x = m_endx; y = m_endy; }
```

The class provides various accessors to obtain the data contained within the class. The size of the grid, the cost of a given node, the start position, and the goal position can all be obtained and modified via these accessors.

```
private:
      GridCell **m_grid;
```

```
        int m_gridsize;
        int m_startx, m_starty, m_endx, m_endy;
```

The class owns a two dimensional array of GridCell objects which it allocates during construction. It is aware of the size of this array, and knows about the start and end positions for the pathfinding system.

2.8.2 MapGridNode Class

```
class MapGridNode
{
public:
        // constructors
        MapGridNode()
              { m_cost = m_x = m_y = 0; m_parent = NULL; m_visited = false;}
        MapGridNode(int x, int y, MapGridNode *parent,
              bool visited, int cost)
              { m_x = x; m_y = y; m_parent = parent;
              m_visited = visited; m_cost = cost; }
        MapGridNode(const MapGridNode &copy);

        // destructor
        virtual ~MapGridNode() { m_parent = NULL; }

        virtual MapGridNode &operator=(const MapGridNode &rhs);
        virtual bool operator==(const MapGridNode &rhs);
        virtual bool operator<(const MapGridNode &rhs);
        virtual bool operator>(const MapGridNode &rhs);

        // accessors
        void setParent(MapGridNode* parent) { m_parent = parent;}
        void setVisited(bool visited) { m_visited = visited; }
        bool getVisited() const { return m_visited; }
        virtual void setCost(int cost);
        virtual int getCost() const;

        // helpers
        bool equals(const MapGridNode &rhs) const
        { return ((m_x == rhs.m_x) && (m_y == rhs.m_y)); }

        // members
        int m_x, m_y; // the coord of the grid cell
        int m_cost;
        bool m_visited;
        const static int BLOCKED;
        MapGridNode *m_parent;
};
```

The MapGridNode class is the interface via which the walkers navigate the MapGrid object. These classes contain the intermediate state information needed by the walker classes to properly navigate the MapGrid. These classes are created and placed into Queue objects which the walkers use to decide which nodes are to be traversed next, and also store the current traversal costs. The MapGridNode base class

provides indices into the MapGrid for their location, a cost value (which may be interpreted by the specific walker class however it needs), a visited flag, and a pointer to the parent node. The pointer to the parent node is extremely important as this node is the only way we can know how the walker class got to this node. By traversing these pointers in a linked list fashion, we are able to find our way back from the ending node to the starting node to build our path. Notice that get and setCost() are virtual, allowing subclasses (such as AStarMapGridNode) to provide their own cost metrics. Let us take a closer look at this class.

```
MapGridNode()
      { m_cost = m_x = m_y = 0; m_parent = NULL; m_visited = false;}
MapGridNode(int x, int y, MapGridNode *parent,
      bool visited, int cost)
      { m_x = x; m_y = y; m_parent = parent;
      m_visited = visited; m_cost = cost; }
MapGridNode(const MapGridNode &copy);
```

The MapGridNode class provides a default constructor which initializes all of the values to null defaults. There is also a constructor which takes all the parameters necessary to build the node in place and a copy constructor for deep copies.

```
virtual ~MapGridNode() { m_parent = NULL; }
```

The destructor is virtual to allow for correct polymorphic destruction of derived classes.

```
virtual MapGridNode &operator=(const MapGridNode &rhs);
```

An assignment operator is provided to help prevent overuse of copy construction, as well as for general assignment usage.

```
virtual bool operator==(const MapGridNode &rhs);
virtual bool operator<(const MapGridNode &rhs);
virtual bool operator>(const MapGridNode &rhs);
```

Various comparators are defined for allowing comparisons to be made. STL requires the < operator to use this object in sorted containers. The other operators are for convenience.

```
void setParent(MapGridNode* parent) { m_parent = parent;}
void setVisited(bool visited) { m_visited = visited; }
bool getVisited() const { return m_visited; }
virtual void setCost(int cost);
virtual int getCost() const;
```

Various accessors are provided to obtain and manipulate the class data such as current parent, visited state, and cost metrics. The cost metrics are virtual to allow derived classes to have their own cost metrics.

```
bool equals(const MapGridNode &rhs) const
{ return ((m_x == rhs.m_x) && (m_y == rhs.m_y)); }
```

The equals method helps derived classes perform default comparisons while adding their own in derived comparator operators.

```
	int m_x, m_y; // the coord of the grid cell
	int m_cost;
	bool m_visited;
	const static int BLOCKED;
	MapGridNode *m_parent;
```

The node's data consists of its coordinate in the MapGrid, its cost, its visited status, a constant to represent blocked cost, and a pointer to its parent for completed path traversal.

2.8.3 MapGridPriorityQueue Class

```
class MapGridPriorityQueue
{
public:
	MapGridPriorityQueue();
	~MapGridPriorityQueue()
		{ makeEmpty(); delete m_head->m_node;
		delete m_tail->m_node; delete m_head; delete m_tail; }

	void makeEmpty();
	bool isEmpty() { return m_size == 0; }
	void enqueue( MapGridNode *node );
	MapGridNode* dequeue();
	void remove(MapGridNode *node);
	bool contains(MapGridNode *node) const;

private:
	class QueueNode
	{
	public:
		QueueNode() { m_node = NULL; m_next = m_back = NULL; }
		QueueNode(MapGridNode *node)
		{ m_node = node; m_next = m_back = NULL; }

		MapGridNode *m_node;
		QueueNode *m_next;
		QueueNode *m_back;
	};

	unsigned int m_size;
	QueueNode *m_head;
	QueueNode *m_tail;
};
```

The MapGridPriorityQueue class is a linked list of QueueNode objects, which contains a pointer to a MapGridNode. It keeps the list sorted in order of the MapGridNode's cost via the getCost() method of MapGridNode, and keeps the nodes with the cheapest cost on the top of the list. It is critically important

to note that this class does not re-sort the list if a node pointed to by this list has its cost changed. In these instances, the node must be removed from the list, and reinserted.

The MapGridPriorityQueue class provides a few methods which are important to note.

The first is the enqueue( ) method. This method inserts a MapGridNode into the list and orders it by its cost.

The second method is the contains( ) method. This method searches the list for a MapGridNode and returns true if the node is in the list.

The next method is the remove( ) method. This method removes a MapGridNode from the list if it is in the list.

Also, there are the isEmpty( ) and makeEmpty( ) methods. These methods determine if the list is empty and empty the list, respectively. Let us take a look at this class in more depth.

```
        MapGridPriorityQueue();
```

The MapGridPriorityQueue provides only the default constructor which initializes the list as empty.

```
        ~MapGridPriorityQueue()
              { makeEmpty(); delete m_head->m_node;
              delete m_tail->m_node; delete m_head; delete m_tail; }
```

The destructor empties the queue, and frees the head and tail nodes.

```
        void makeEmpty();
```

The makeEmpty method empties the queue and properly frees the memory as necessary.

```
        bool isEmpty() { return m_size == 0; }
```

The isEmpty method returns if the queue is empty.

```
        void enqueue( MapGridNode *node );
```

The enqueue method adds the node to the queue and places it in sorted order based on its cost.

```
        MapGridNode* dequeue();
```

The dequeue method removes and returns the node which has the lowest cost in the queue.

```
        void remove(MapGridNode *node);
```

The remove method simply removes the node from the queue.

```
        bool contains(MapGridNode *node) const;
```

The contains method searches the queue for the node and returns if the node is contained in the queue.

```
            QueueNode() { m_node = NULL; m_next = m_back = NULL; }
            QueueNode(MapGridNode *node)
            { m_node = node; m_next = m_back = NULL; }
```

The contained class QueueNode is the actual data which the MapGridPriorityQueue contains. It has a default constructor which assigns its data to null, and a copy constructor which copies the node to which it points, but it does not place it in the queue.

```
            MapGridNode *m_node;
            QueueNode *m_next;
            QueueNode *m_back;
```

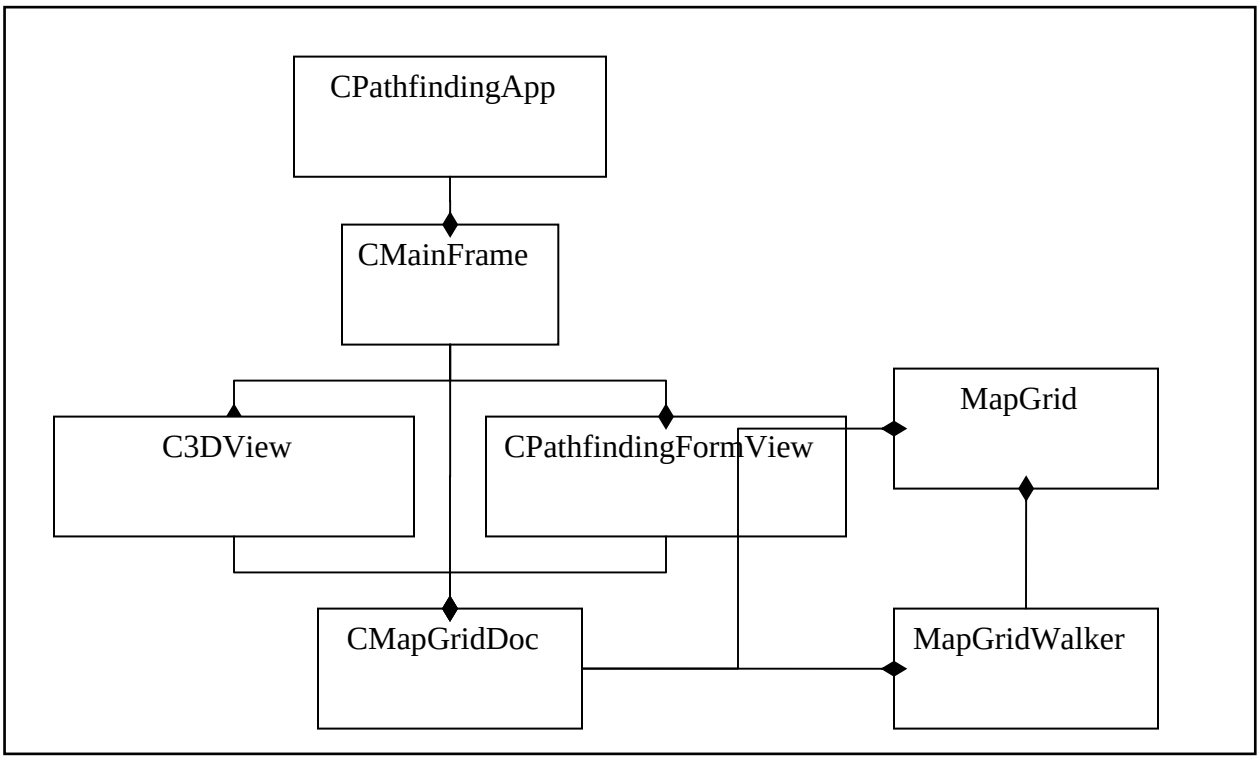
The QueueNode contains a pointer to the node it is holding, as well as pointers to the next queue node and the previous queue node.

```
        unsigned int m_size;
        QueueNode *m_head;
        QueueNode *m_tail;
```

The MapGridPriorityQueue stores a size, a head pointer to the front of the queue, and a tail pointer to the back of the queue.

You will see all of these members in action when you explore the source code to the pathfinding lab project.

2.9 MFC Document/View Architecture and Our Demo



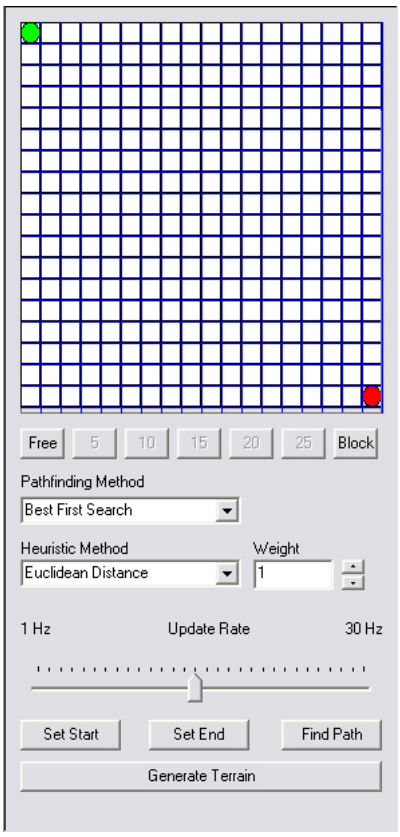
MFC has an interesting architecture called Document/View. The core idea is that you have a single **Document** object containing all information relating to a single instance of the application's data (such as a Word document, Excel spreadsheet, or in our case, a MapGrid) and any number of **View** classes meant to display that data in some fashion.

In our application we have a CPathfindingApp which is the controlling entity. It contains a CMainFrame which is the window frame itself that contains the CPathfindingFormView, the C3DView, and the CMapGridDoc. It associates the view and the document via a DocumentTemplate object.

> **Note:** We will not discuss in too much detail how MFC does its work in this fashion, as it is outside the scope of this course. However, those of you who have taken the C++ Programming courses here at Game Institute are in a good position to begin your investigations into MFC. It is a large API, but not difficult to learn, given your current level of Windows programming experience. Perhaps exposure to MFC here in this course will inspire you to look further. It is a powerful system that allows you to quickly assemble all sorts of useful Windows applications.

Every View has a pointer to its Document so that it can get data from the document to display itself. In our case the document contains a MapGrid and one of each of the MapGridWalker objects. We then let the view update the MapGrid and instruct the MapGridWalker to do its iteration via the CMapGridDoc. The 3DView and CPathfindingFormView can then be updated to display the correct results based on what information is in the CMapGridDoc.

2.9.1 The Form View Panel



Our panel is designed so that we can click on the grid and make our map, select a pathfinding method, any heuristics that might be appropriate, the update rate, the start, the finish, and make it go.

We capture mouse clicks via the CPathfindingFormView (which can be done using MSVC's class wizard), and do a hit test against the grid. Whichever grid square we click is the one we modify (provided the user clicked one). The buttons for the grid costs are enabled and disabled based upon which pathfinding method is selected in the Pathfinding Method dropdown. When the selection changes, we check to see if the method supports weighted graphs and enable or disable the grid costs buttons as appropriate. The Heuristic Method drop down is also disabled or enabled and populated based on the selected method. We check if the method selected supports heuristics and, if so, we enable the drop down and populate the list from the specific walker. The Weight textbox and spinner are only enabled if heuristics are enabled.

The update rate scroller updates the frequency of the timer which drives the iterate calls when the app is in Find Path mode.

The set start and end buttons enable setting the start and end points of the grid when selected.

The Find Path button starts the app searching for a path using the currently selected method and heuristic (if any).

Lastly, the Generate Terrain button takes the current grid, and generates a small terrain for the 3DView. It uses A* to find the shortest path to the goal and make a sandy road along the path. We will not discuss in detail how it does all of that, as it is outside the scope of this course. If you would like to know more about the 3D aspects of this course, it is suggested that you explore the Graphics Programming courses offered here at the Game Institute (Module I, Chapter Seven in this case).

2.10 Conclusion

This brings us to the end of our core pathfinding discussions. In this chapter we were able to introduce one of the most powerful algorithms available to us: A*. We talked about how it works and how we can improve it if we find performance becoming a concern. We also talked about a number of hierarchical pathfinding methods that can prove to be useful when dealing with large maps or non-gridded worlds. It is well worth your time to try implementing some of the hierarchical methods we discussed using the source code you obtained during your studies in the Graphics Programming series. There is much there that can be applied; both with respect to spatial partitioning and to rendering on the whole. You are encouraged to start bringing some of those tools to bear as you work your way through this course. For example, try to get your animated characters (Graphics Programming Module II) to traverse various gridded scenes (which you can create easily in GILES™). Or perhaps try your hand at implementing something using the RTS design we presented earlier in the lesson. Of course, before you try any of these projects, make sure that you understand the demonstration that accompanies this chapter.

In the next chapter we will begin to transition between pathfinding methods and decision making. You will see that during the development of a behavioral system called *flocking*, we will begin to blur the line between these disciplines and bring elements of ideas from both camps to bear. While our flocking system will not directly implement graph-oriented pathfinding as we have done in these last two chapters (although it will take advantage of it later in the course), it will still provide a form of environment navigation, mostly within a more localized area. More on this in the next lesson…