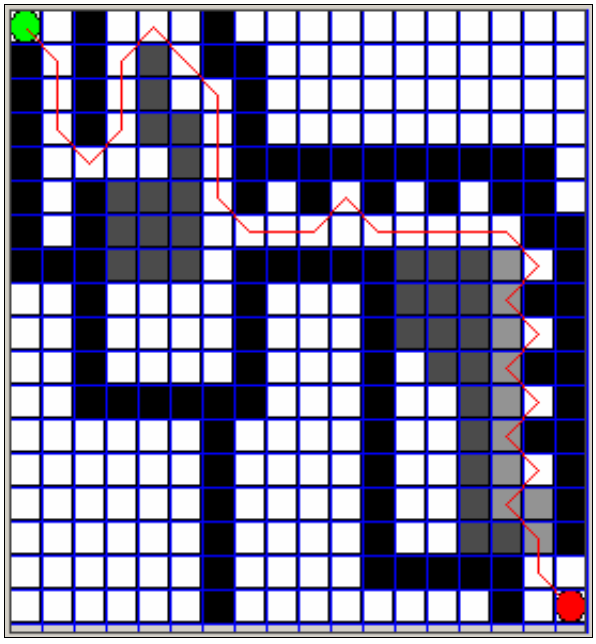
Chapter 1

Pathfinding I



Introduction

Artificial intelligence (AI) is one of the critical components in the modern game development project. With the exception of graphics and sound, there are very few elements that are as vitally important when it comes to establishing engaging gameplay. The AI breathes the life of the development team and the designers into the game, and presents the player with the challenges that keep the game interesting and fun to play. In many ways, artificial intelligence *is* the game. In fact, it is not uncommon for modern game engines to devote as much as 20% or more of their processing time solely to artificial intelligence.

Artificial intelligence can be an awkward subject to define because many game developers hold a different set of ideas about what it means and what exactly it constitutes. So let us begin by trying to establish a working definition and then we will move on to what components it encompasses.

We all have a pretty good understanding about what the term "artificial" means. It is a man-made substitute for something natural (i.e., a simulation). But "intelligence" immediately begs the question, "What do we mean when we say something is intelligent?" Since this is not a Psychology course, we need not delve too deep to arrive at a useful answer. Certainly there are many ways that people intuitively characterize intelligence. We often think of intelligence as a measure of one's ability to acquire knowledge and learn from experience. A more utilitarian definition might focus on the use of reasoning faculties to solve problems.

By combining and simplifying these various concepts we can arrive at a fairly good standard definition: artificial intelligence is the application of simulated reasoning for the purposes of making informed decisions and solving problems. This seems like a fair enough way to characterize AI and it probably sits well with what most of you had in mind when pondering the nature of the terminology.

In the relatively young field of artificial intelligence, the definition we have constructed here is quite applicable. Much research has gone, and continues to go, into creating machines that simulate human intelligence. Some people might place specific methods of solving problems in different AI categories while others bind it all up into a single unified AI concept, but whether taken collectively or not, this is probably a good overall means for understanding AI conceptually.

But an important question to ask is, "is this what we mean when we talk about AI in games?" Well, in a manner of speaking, yes. But for our purposes as game developers, we will simplify even further and define artificial intelligence as the means by which a system approximates the appearance of intelligent decision processes. This concept of the "appearance" of intelligence is a very critical point. It is significantly important because there is obviously a distinct difference between artificial intelligence *being* intelligent and artificial intelligence *appearing* intelligent.

A quick story will illustrate this point. While working on the game *Psi Ops: The Mindgate Conspiracy*™, our team spent a good deal of time making the various enemies behave more intelligently. We programmed them to crouch and duck, roll out and fire, hide behind cover, throw grenades from cover, dodge objects thrown at them, chase a fleeing player, and pull alarms when they needed help. Despite all this effort, the game designers did not think that the enemies were intelligent enough. They wanted them to be "smarter" and exhibit even more complex and intelligent behaviors.

The AI programming team's initial response to this request was simply to double the hit points of the minions. The results of this small modification may surprise you -- the designers were thrilled with the "intelligence enhancement." But in reality, it is clear that the characters were no more intelligent than they were before the hit-point increase. Essentially what we learned was that the enemies were dying too quickly for the designers to fully appreciate their intelligence. While it may not immediately jump out at you, this story illustrates an important point: when it comes to artificial intelligence in games, more often than not, it is all about *end user perception*.

That is, for game development, it is fair to say that if it *looks* smart, it is smart. It is ultimately irrelevant to the player how an AI system makes the decisions it does. They care only that the system seems to produce behaviors that give at least the outward appearance of having been thoughtfully considered. In other words, game AI is essentially a results-oriented concept. How the AI was able to arrive at the decision it did (and we will explore some of the methods for handling decision making in this course) is not remotely as important as the action that took place when the decision was finally made.

This is not a new concept of course. Alan Turing's famous test of machine intelligence is a good example. The Turing Test conceived of locking away a human interrogator in one room while a human and a computer were situated in another room. The means of communication between the two rooms would be text only. The central question was, could the interrogator tell the difference between the human and the computer based on an interactive exchange of questions and dialog? Turing suggested that the measure of the machine's intelligence was its ability to convince the human interrogator that it was interacting with another human.

This is not that far from where we find ourselves today. After all, our goal is to design an AI that makes the player believe that the entities in the game world are behaving the way one would expect an intelligent being to behave.

To be sure, this was not always the case in the game field. Indeed the earliest games used virtually no AI at all. Games like *Space Invaders*, *Centipede*, *Galaga*, and *Donkey Kong* made little effort to convince the player that they were interacting with truly intelligent beings. Hardware limitations resulted in gameplay that relied almost exclusively on pattern-driven events with some varying degree of randomness. Eventually, a good player would recognize and memorize those patterns (e.g., where the next enemy would appear at the top of the screen) and use that knowledge to advance in the game.

Today's games exhibit a considerably more sophisticated set of artificial intelligence than those early titles could provide. As the rendering of more realistic scenery and in-game characters continues the steady march forward, the AI programmer will feel the pressure of having to maintain pace. Physically realistic looking game characters are expected to be paired with realistic looking behavioral traits. However, in modern game systems, graphics and sound have their own dedicated hardware, while AI remains CPU bound. So our job is to build AI systems that can provide that added realism without consuming all of the processing load needed for other important game tasks that aid in player immersion (like realistic physics models for example).

1.1 A Few Guidelines

Before we begin discussion about the different types of artificial intelligence we are going to examine in this course, let us first take a moment to establish a few helpful guidelines that will come in handy when designing artificial intelligence for games.

1.1.1 Love and Kisses

One of the most important things to remember is the "KISS" method. KISS is an acronym that stands for "Keep It Simple Stupid". As games become more advanced and hardware becomes ever faster, the software simulations are becoming more advanced to follow suit. As AI developers, not only do we *not* have the luxury of infinite processing time, but we will soon learn that we do not need the systems to be overly complicated. This is where our second and related acronym comes in: "LOVE". This is short for "Leave Out Virtually Everything". In most cases, the simplicity of the artificial intelligence used in many games would probably shock you. Remember the mantra from earlier: artificial intelligence approximates the *appearance* of intelligent decision processes. In other words, the AI only needs to convince the player that it is doing something smart. To be sure, there is a fine line that the AI developer must always be aware of. While simplicity can be a beautiful thing, we must be careful to make sure that the simplicity of our implementation does not come at the cost of the player experience. The last thing we want is for our game AI to become predictable and boring.

1.1.2 Hard Does Not Equal Fun

It is probably fair to say that most people do not want to play a game where it takes fourteen hours to complete a small level because the puzzles are too hard or they keep dying over and over again. Indeed it is not complicated to code artificial intelligence that is so difficult to beat that the game is no longer any fun to play. Real time strategy (RTS) games can easily fall into this trap. Since an RTS game is deeply mathematical, it is very easy for a developer, who has advance knowledge of all of the inner workings of the simulation, to build an artificial intelligence system that makes optimal use of its resources, buildings, and units all of the time. The problem with this approach is that a human player could never be as efficient as the computer driven artificial intelligence. Apart from the obvious advantage of pure calculation ability that the computer maintains, on a more practical level, at any given time a player can only interact with so many units and can only view a subset of the entire map. This places him at an impossible disadvantage versus the AI opponent. You should always be aware of these types of imbalances which the game interface imposes on the player. Certainly in this case it is obvious that the player will never be able to play as effectively as the artificial intelligence and allowances will need to be made.

1.1.3 Play Fair

Taking a cue from our last rule, as much as possible, you should make the artificial intelligence play by the same rules that the player must abide by. It is supremely frustrating to play a game where the artificial intelligence "cheats". Once again, real time strategy games often exhibit a tendency to cheat in this fashion. For example, in many cases, the game AI knows where the player units are located and often it does not need to pay the same costs for unit production. This is grossly unfair to the player. But RTS games are not the only culprits. In many first person shooters, the artificial intelligence also knows where the player is, and when alerted, relentlessly chases him down, regardless of where he runs. Additionally, enemies in many shooters do not have to worry about ammunition resources. All of this adds up to an unfair advantage and potentially, a very disenchanted player.

1.2 Fundamental Artificial Intelligence

There are many subcategories of artificial intelligence, each of which has its own usage scenario. Some of these types tend towards the complex and as such, remain more in the domain of academic research than in game development (although there is often some crossover).

Classification, for example, is a type of artificial intelligence that typically takes some input data and classifies it as something else. For instance, there may be a system that looks at a small bitmap and determines what letter of the alphabet or number it is. Types of classification systems include neural networks and fuzzy systems. These systems require "training" in order to produce the classifications desired. This training typically involves showing the system examples of each of the things which need to be classified, along with the expected result. The training algorithm then adjusts the internals of the classification system to attempt to produce the desired output on the fly. Here we see the concept of an AI actually learning and getting smarter as a result. These systems can be very difficult to build and perfect and are not widely used in games. However, the idea of simulated "learning" makes these systems very popular in the wider AI research field. We will not be discussing classification much in this course since it tends to be a more esoteric type of AI which is often more complicated than we will need in the typical commercial game.

Life Systems are another type of artificial intelligence that is popular in AI research, but less so in games. Genetic algorithms are a popular example. Life Systems work by creating a set of artificial intelligence systems, letting them perform, and then rating them. The ones with the highest rated performances survive and evolve, while the ones with the lowest rated performances are killed off. Clearly we see a relationship to the study of evolution at work here. What is most interesting about these systems is the concept of emergent behaviors. Rather than scripted sequences that are hardwired into the AI, such systems will often produce completely unexpected behaviors that emerge as a result of the AI adapting and maturing over time. This kind of AI can be a lot of fun to observe and study, but there is a major downside. Emergent behaviors tend to lead to systems that are very idiosyncratic. Sometimes you will get the behavior you expect, and other times you will not. The problem is, when you cannot control the outcomes, scenario design becomes very difficult. While there are some games that make use of this AI (e.g., SimCity™), most games do not. The Adaptable AI seminar here at the Game Institute explores

an interesting way to approach Life Systems by utilizing concepts from biology, evolution and genetic science. It is a great way to follow up the material we will study in this course if you are interested in investigating this area further.

It is worth noting upfront that in this course, we are going to adopt a practical approach to studying AI. Our goal is not to learn a little bit about everything, but rather to zero in on the key areas of study that the typical game AI programmer will need to master if he wants to join a professional programming team. As such, we are going to spend almost all of our time focusing on two of the most fundamental artificial intelligence systems that game developers need to learn and understand: decision making and environment navigation (or "pathfinding").

1.2.1 Decision Making

Decision making is the core component of all artificial intelligence systems. It is a compilation of routines which help the AI entities decide what they want to do next. This system typically determines decisions such as what to build, when to attack, when to look for cover, when to shoot, when to run away, when to get health, etc. Decision making is the chief means by which the artificial intelligence appears intelligent. State machines, decision trees, and squad behaviors will be examples of decision making that we will talk about in this course.

1.2.2 Pathfinding

Pathfinding is the aspect of artificial intelligence systems which assists the AI driven entities with navigating in the game environment. At its simplest, this means moving the entity from one location to the next without running into things. Pathfinding is arguably the most fundamental type of artificial intelligence for games because without it, entities will remain unable to take on any convincing physical presence. Indeed there are very few genres of games where these algorithms will not be needed.

The combination of just these two AI types can lead to the production of virtually any game scenario you desire. You can create NPCs that range from the simplest of life-forms to emotionally complex entities that exhibit sophisticated reasoning ability. Of course, we know that appearances can be deceiving and that under the hood things may not be nearly as complicated as what the behaviors would indicate. But as we discussed earlier, perception is everything and results are what matters. The decision making and environment navigation systems we develop together in this course will serve as a solid foundation for your AI engine and will provide you with the ability to really express yourself creatively in future projects. If you are also taking the Graphics Programming series here at the Game Institute, then combined with this course, you will have a very impressive set of tools that you can leverage to build tech demos for your next interview or even complex games for your own enjoyment.

While our focus in this course will be on the AI fundamentals, please feel free to drop by the live discussion sessions if you would like to talk about other types of AI that we will not cover in the text.

1.3 Getting Started

Now that we have had a quick overview of the various types of artificial intelligence and some ground rules have been established, we will waste no time in getting started. We will begin our AI studies together with one of the most important areas of artificial intelligence: pathfinding. This topic will serve as a good lead-in for Decision Making because in a sense, pathfinding represents the simplest decision making process of all. The decision centers around the question: how do I get from point A to point B? The decisions themselves are ultimately a choice between directions of travel. That is, if I am 'here' and I want to go 'there', what is the next step I should take? Should I go left, right, up, down, etc.? What step should I then take after that? And so on. At first you might think that this does not really seem to be artificial intelligence. But recall our earlier emphasis on the appearance of intelligence. Even if your game engine simply selected random points in the world at random times and said to an NPC, "go there", from the player's perspective the NPC would *appear* to have some particular purpose as he wandered by. So by providing the means to get from A to B, we have instilled the NPC with some very basic, but still very important game AI capability.

In remainder of this chapter the following questions will be addressed:

- What is pathfinding?
- Why is pathfinding useful or necessary?
- What is a graph?
- What is a weighted graph?
- What is a directed graph?
- What are the traditional types of pathfinding methods?
- How are the traditional types of pathfinding methods implemented?
- Who is E. Dijkstra?
- What is Dijkstra's Algorithm?
- What are some traditional implementations of Dijkstra's Algorithm?

1.4 Introduction to Pathfinding

Pathfinding is a critical component in just about every game on the shelves. Without pathfinding, autonomous entities would not be able to get from place to place. Think of a simple case where you have a large field and a tractor that needs to go from one side to the other. The tractor starts on one side and proceeds in a straight line to the other side. When the tractor reaches the other side, it stops. Did it use a pathfinding algorithm? Of course! It may be very simple and rudimentary, but it found a straight-line path from one side of the field to the other. If the field had ditches to avoid, this particular algorithm would not have been the best choice. How would the tractor get across in that case? That question is one of many we will learn the answer to as we progress in this course.

1.4.1 Graphs and Pathfinding

Pathfinding is ultimately about the traversal of a graph. For our purposes, a **graph** is simply a set of points connected by paths between them (see **Figure 1.1**).

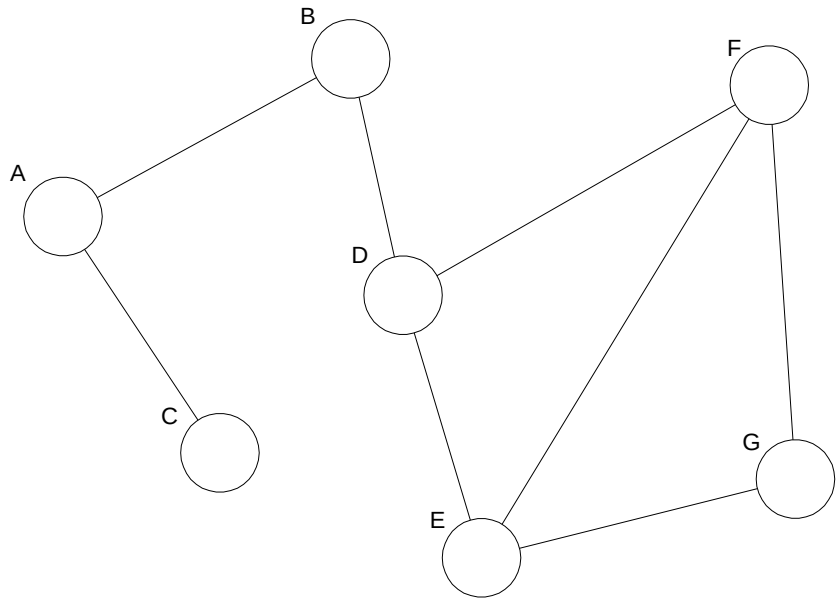


Figure 1.1

A graph can contain any number of points (also called 'nodes') as well as any number of connections between those points. The interesting thing about a graph is that there are actually an infinite number of paths from any point on the graph to any other point on the graph. Pathfinding in the gaming world typically means finding the shortest path. It would not make sense to have an avatar running back and forth between points, or going in circles multiple times before reaching the destination.

Graphs can also have *costs* associated with traveling a particular path between points. A cost is a value that indicates an implied relative expense for choosing one path over another. Depending on how we choose to interpret this value, one path will be deemed more cost-effective to traverse than another, so we will generally want to choose that path over the alternative(s). This type of graph is referred to as a **weighted graph** (see **Figure 1.2**).

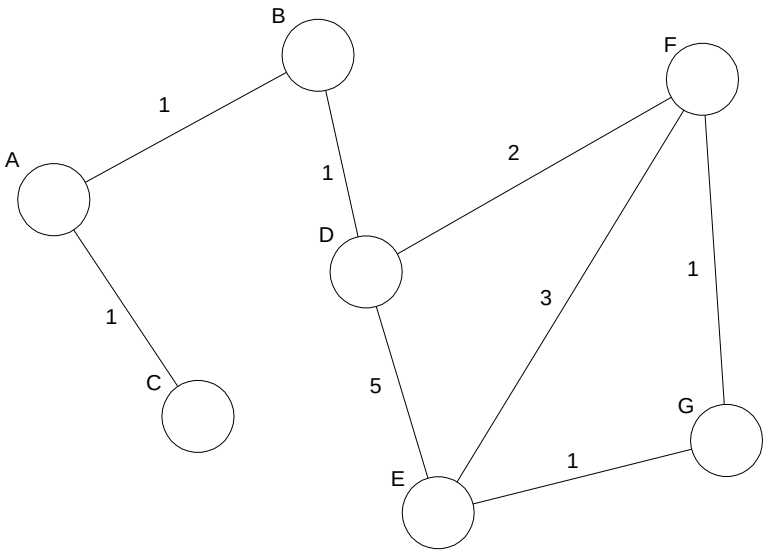


Figure 1.2

Weighted graphs are identical to un-weighted graphs in all respects, except that there are weights/costs associated with the paths between points. This weight might be fixed or it might even be a function.

Lastly, there is the concept of a directional graph. In **directional graphs**, each path between the points can be considered to be one-way (see **Figure 1.3**).

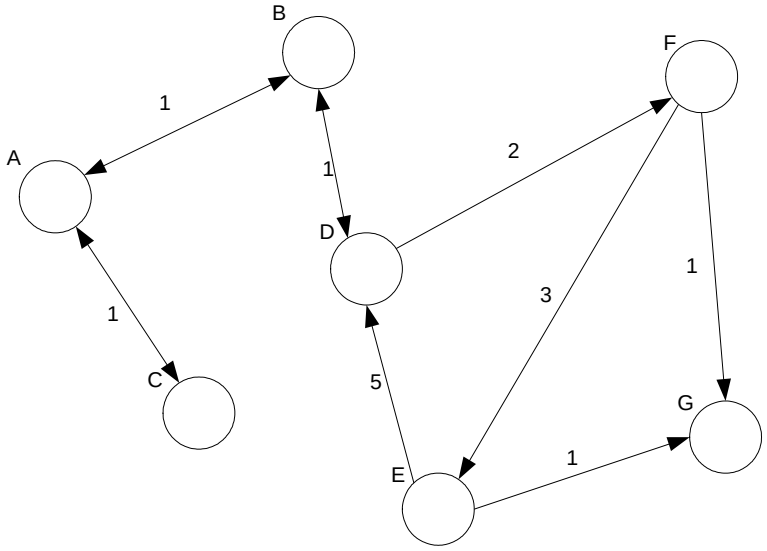


Figure 1.3

Like a weighted graph, a directional graph can contain weights on the paths between its nodes. These types of graphs can be more complex, as the entity cannot always go back from the direction it came.

1.5 Graph Traversals

Pathfinding is simply a shortest distance graph traversal. Let us imagine that we are traversing the un-weighted graph in **Figure 1.1**, and we want to get from A to E. We would want to travel from A to B to D to E because this path is the most direct route. However, in the case of the weighted graph (**Figure 1.2**), we would choose the path, A to B to D to F to G to E, as it is the least expensive path in terms of cost. In the case of the directed graph (**Figure 1.3**), we would choose the path A to B to D to F to E. How we determined that those paths were the least expensive is covered in the next topic. But first we need to redefine our graph.

The graphs we have already seen are a bit contrived, so let us change our design to something more like a map, as this is a fairly common graph layout in games. For now, we will define our graph as a regularly spaced grid of points where one can travel N, NE, E, SE, S, SW, W, or NW (**Figure 1.4**).

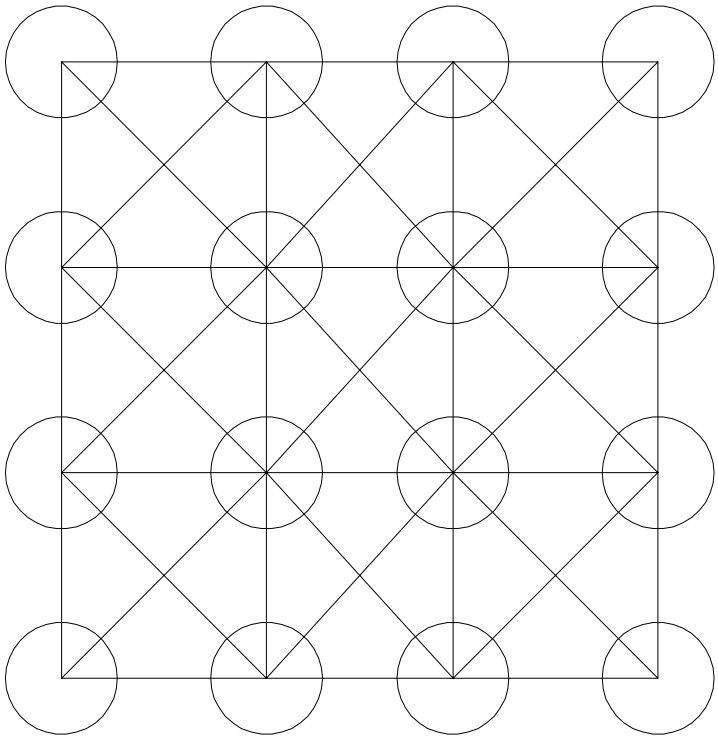


Figure 1.4

A graph such as the one in Figure 1.4 represents a map which can be found in many top-down real time strategy games because such games typically take place over vast expanses of terrain. In most games, terrain geometry is built using a uniform grid of polygons, and this lends itself well to such a representation. Units can move in any of the cardinal directions. To prevent movement to a particular node requires only the removal of the node from the graph, simply marking it as impassable. To make it more expensive to travel to a given node, a cost can be associated either with the node itself or the path to the node.

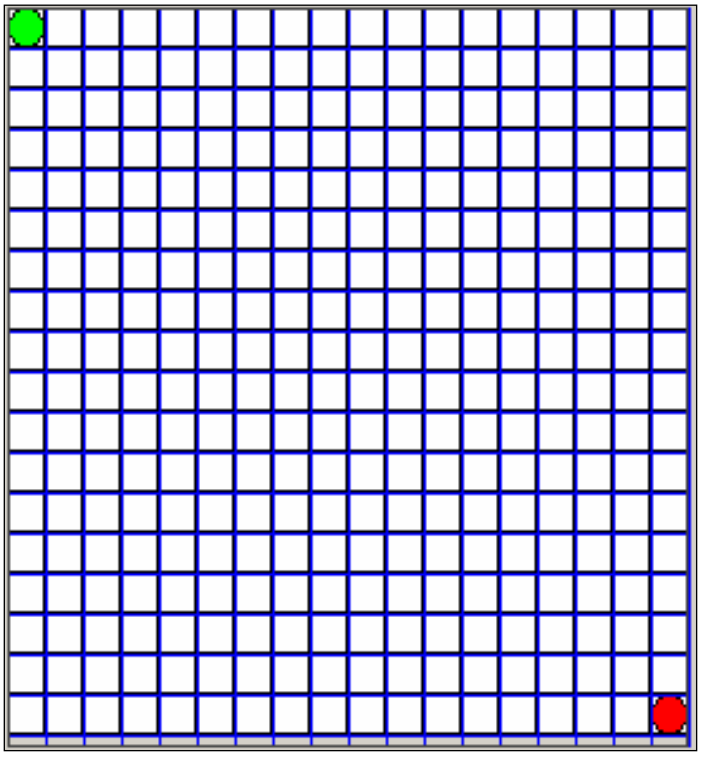


Figure 1.5

Although the graph displayed in **Figure 1.4** is the primary type of graph we will be examining in this course, it is very difficult to read in that form. From now on, we will look at graphs as shown in **Figure 1.5**. The green dot represents where we start (our origin) and the red dot represents where we wish to go (our destination). Valid paths of travel are in all of the cardinal directions. As grid squares become increasingly more expensive to travel through, they will become darker gray. Impassable grid squares will be black.

1.5.1 Non-Look-Ahead Iterative Traversals

With this new graph to navigate, and an easier way to look at it, let us briefly talk about some of the methods that might be used to get from origin to destination, but which typically have pitfalls. These methods all have one thing in common; they do not look ahead to find a good path to the goal. They will make their decision based solely on their current position and the position/direction of the goal. The concept itself is simple: take one step at a time towards the goal. The system becomes more complicated when obstacles in the environment must be navigated around. So first, let us examine the most common methods of avoiding obstacles in non-look-ahead iterative traversals.

1.5.1.1 Random Backstepping

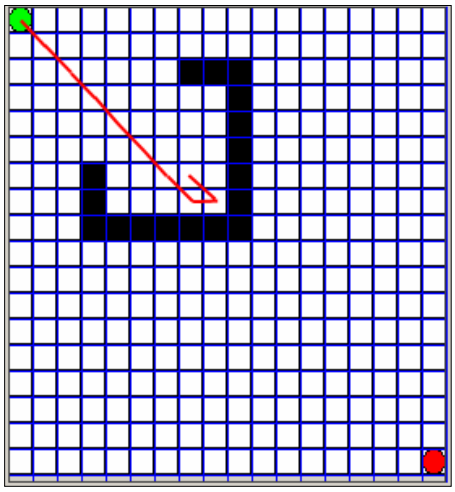


Figure 1.6

The simplest method is to take one step at a time in the direction of the goal. If an obstacle is encountered, try to step around it. If the obstacle is too large/long (i.e. 3 or more squares long centered on the current location), take a step back in a random direction, and try again. This method encounters serious problems if a cul-de-sac is encountered as it only takes a single step back (see **Figure 1.6**).

1.5.1.2 Obstacle Tracing

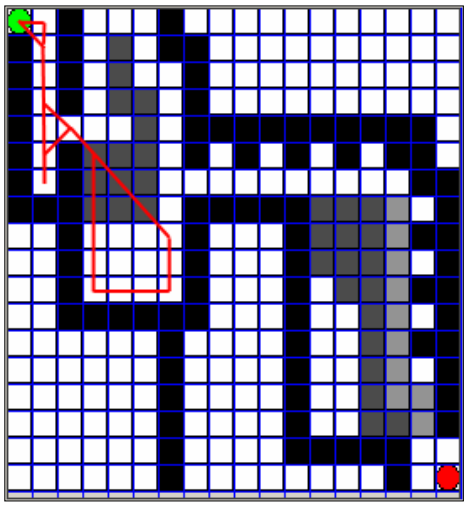


Figure 1.7

Another method is to move one step at a time in the direction of the goal, and if an obstacle is encountered, trace around it to the right. This method encounters problems in complicated graphs, as it can get caught in a cycle where it repeats (see **Figure 1.7**). To prevent this method from entering infinite loops, a common solution is to detect if the path taken traces across the path again. However, this does not make the method any more successful. Another method is to trace to the right until the path is crossed, then trace to the left until the path is crossed. In the case of this graph, tracing to the left would have succeeded.

1.5.2 Look-Ahead Iterative Traversals

Now that we have seen some of the methods that can be used to traverse a graph without looking ahead, let us take a look at some methods that plan the entire path before taking a single step.

1.5.2.1 Breadth First Search

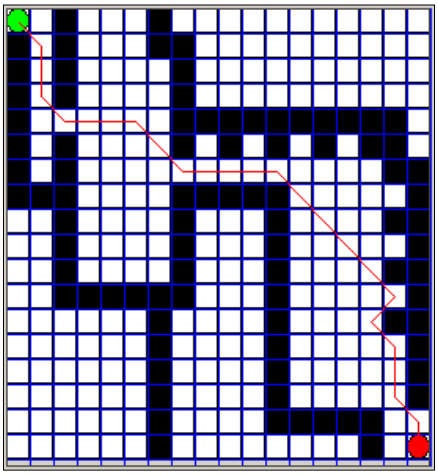


Figure 1.8

One of the most fundamental graph traversal methods is Breadth First Search. This method finds the shortest path in an un-weighted graph by iteratively searching the neighbors of the start position until it reaches the end position (see **Figure 1.8**). This is a robust method which will always find the shortest path, but it can require much CPU time doing it.

1.5.2.2 Best First Search

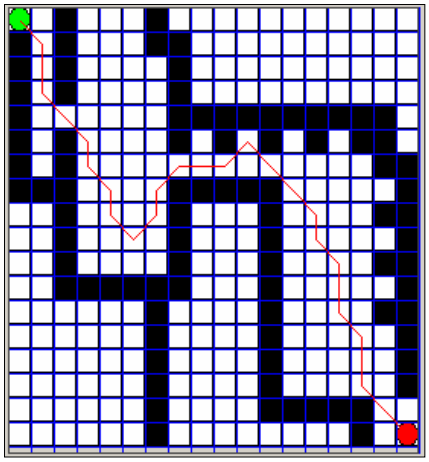


Figure 1.9

Another method which is very similar to the Breadth First Search is the Best First Search. This method iteratively searches all the neighbors of the start node of an un-weighted graph, but it chooses the neighbor with the perceived best chance of having a path first (see **Figure 1.9**). This method will always find a path if there is a path to be found, but it may not be the shortest. It sacrifices the shortest path for the speed in which it finds a path using a heuristic.

1.5.2.3 Dijkstra's Method

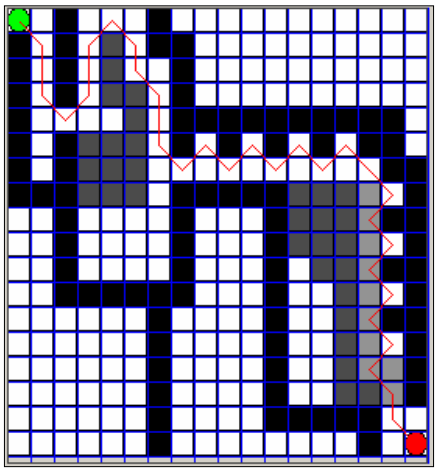


Figure 1.10

Another method, created by E. Dijkstra and now called Dijkstra's method, is a very robust method of traversing graphs. This method finds the shortest path of a weighted graph by keeping track of the cost to every node (see **Figure 1.10**). This is a useful method but it is not the fastest method when dealing with large graphs.

1.5.2.4 A* Method

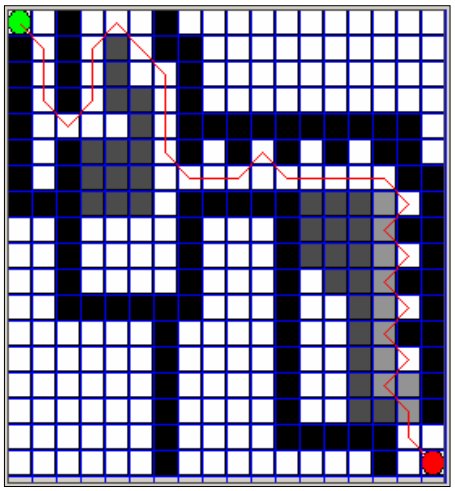


Figure 1.11

One of the most efficient pathfinding methods available is known as A* (A-star). This method is a very robust weighted graph traversal that makes use of heuristics to find the goal in a timely manner (see **Figure 1.11**). This method is very powerful as it allows extra knowledge about the graph to be leveraged in the heuristic. This method will be the center of discussion in the next chapter.

1.5.3 Look-Ahead Recursive Traversals

Some graph traversal methods are most easily implemented via recursion. The most popular of these methods is the Depth First Search traversal. Instead of searching all the neighbors as in other methods, it searches deep into the graph first. This method can have problems if the depth of the search is not limited. Many times this is handled by limiting the depth by guessing the distance to the goal, via a heuristic, and increasing the depth until the goal is found. This can be a very time consuming traversal, and dangerous in large graphs, due to recursive depth.

1.6 Non-Look-Ahead Iterative Methods, In Depth

We have discussed performing pathfinding one step at a time and mentioned some methods for dealing with navigating around obstacles when they are encountered. Now let us take a closer look at these obstacle avoidance strategies to better understand them.

1.6.1 Random Backstepping

The Random Backstepping (or Random Bounce) method is simple in its execution; it moves a step at a time towards the goal. If it runs into an obstacle, however, it chooses a random direction in which to move and tries moving toward the goal again. Though simple and elegant, this method will fail to get out of deep cul-de-sacs.

The Algorithm

```
bool RandomBounce(Node start, Node goal)
{
      Node n = start;
      Node next;
      while(true)
      {
            next = n.getNodeInClosestDirectionToGoal(goal);
            if (next == goal)
                  return true;
            while (next.blocked)
                  next = n.getRandomNeighbor();
            n = next;
      }

      return false;
}
```

Listing 1.1

The method outlined in **Listing 1.1** is straightforward. View it in its entirety, and then read on for more detailed discussion.

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

Let us start with the declaration itself. We will provide a start node and the goal node at which we are attempting to arrive. When we are done, we will return true if we arrived at the goal node, and false if we fail.

```
      Node n = start;
      Node next;
```

First, some locals are defined to keep track of the starting node and the next node to which we plan to go. The node is initialized to be the starting node which was passed in.

```
      while(true)
```

This method will run until we find a solution. Presumably, if we fail at finding a solution, we will give up after some number of iterations rather than cycling forever as this particular loop does.

```
            next = n.getNodeInClosestDirectionToGoal(goal);
```

The method *getNodeInClosestDirectionToGoal*() is graph specific, but it will always return the best neighbor node to this node which will put us closer to the goal.

```
            if (next == goal)
                  return true;
```

If the next node returned is the goal node, then we successfully made it to the goal.

```
            while (next.blocked)
                  next = n.getRandomNeighbor();
```

Here is where the obstacle avoidance is applied. If the next best node that leads us towards the goal is blocked, a randomly selected neighbor to this node will be selected and tested. This is done until a neighbor node is found which is not blocked. Presumably, we would exit if it was discovered that all of our neighbor nodes are blocked.

```
      n = next;
```

After a valid next node is returned, it will be set to our current node and iteration continues.

The idea is to take a step at a time towards the goal, and if the step we wish to take is blocked, pick a random direction and go in that direction instead. This process is continued until we reach our goal. This is very different from any of the algorithms that we are going to implement in this course, as it does not keep track of the path it took; it just knows where it is, and where it wants to go. It also never gives up in its search for the goal. An enhancement to this algorithm might be to add some kind of maximum iteration count which is checked periodically so that it does not continue to fail forever. In many cases this is adequate for some games, even if it may be a little boring.

1.6.2 Obstacle Tracing

Obstacle Tracing is exactly like the Random Bounce method in its means for getting to the goal. The difference is that it attempts to trace around an encountered obstacle until it can head toward its goal again. A more robust method would change the direction in which it traces, or wait until it crosses a line from the start to the goal again before attempting to approach the goal.

The Algorithm

```
bool Trace(Node start, Node goal)
{
	Node n = start;
	Node next;
	while(true)
	{
		next = n.getNodeInClosestDirectionToGoal();
		if (next == goal)
			return true;
		while (next.blocked)
			next = n.getLeftNeighbor(next);
		n = next;
	}

	return false;
}
```

Listing 1.2

The method outlined in **Listing 1.2** is identical to the random bounce method with the exception of what it does when its desired node is blocked. Review the code in its entirety and read on for more in-depth discussion.

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

Let us start with the declaration itself. Just like Random Bounce, a start node and the goal node at which we are trying to arrive will be given. We will return true if we arrived at the goal node, and false if we fail.

```
	Node n = start;
	Node next;
```

Some local variables are defined to keep track of the current node, and the next node we plan to go to. The start node which was passed in will be initialized as our starting node.

```
	while(true)
```

Just as with RandomBounce, this method will run until a solution is found. Presumably, we will give up after some number of iterations rather than continuing forever as this loop does.

```
		next = n.getNodeInClosestDirectionToGoal(goal);
```

Just as with the Random Bounce method, *getNodeInClosestDirectionToGoal*() is graph specific, but it will always return the best neighbor node to this node, which will put us closer to the goal.

```
		if (next == goal)
			return true;
```

If the next node returned is the goal node, then we successfully made it to the goal.

```
            while (next.blocked)
                  next = n.getLeftNeighbor(next);
```

This is where the Obstacle Tracing method differs from the Random Bounce method. Rather than grabbing a random neighbor, the neighbor of this node which will take us to the left of the node passed in will be selected. The method *getLeftNeighbor* is graph specific, but it will always return the node which will take you to the left of the input node. We will keep looking to the left until we get a node that is not blocked. Presumably, we would exit if we found ourselves in a condition where there was no way out.

```
      n = next;
```

After a valid next node is returned, it is set to our current node and iteration is continued.

We could make this routine a little more robust by checking to see where we started tracing and trace all the way around until we returned to where we started. If we did return to our start position, we can try to trace the other way, or just give up. We could also calculate the line between our start point and our end point, and if we passed that line twice while tracing, we could give up or try something other than tracing (maybe resorting to a little random bouncing).

1.7 Look-Ahead Iterative Methods, In Depth

We mentioned a variety of look-ahead iterative methods which plan the entire route from the starting point to the goal in advance. This ensures the path chosen will be effective, and in most cases, the shortest. Let us look at these methods in more detail, as well as an implementation example for each.

1.7.1 A Note on Implementation Examples

Both implementation examples we are going to discuss are taken from the code provided in the course projects.

```
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:
      MapGrid *m_grid;
};
```

Listing 1.3

In order to make the chapter demo display the path as it was being discovered, objects derived from *MapGridWalker* will do their traversals one step at a time during the *iterate*() method. After they iterate each step of the traversal, *drawState*() is called to draw the current state of the traversal. For the sake of brevity, we will only discuss the contents of the *iterate*() method and any heuristic functions that apply to the pathfinding algorithm. Of course, in most games, you would want to find the entire path in one pass rather than iterating repeatedly. Let us discuss a few of the elements of this class in more detail.

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

The WALKSTATE enumeration will provide information on the progress of the algorithm in its search for the goal. STILLLOOKING represents that the algorithm is still searching for the goal, and has not encountered any problems as yet. REACHEDGOAL means the algorithm has reached the goal and a path has been created. UNABLETOREACHGOAL is returned when the algorithm cannot build a path from the start and goal nodes given.

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

The class supports a default constructor as well as a constructor, both of which take the grid upon which it will walk as a parameter. If the default constructor is used, the grid must be set separately.

```
      virtual ~MapGridWalker();
```

The class has a virtual destructor so that derived classes can properly clean up their resources if delete is called on a MapGridWalker pointer.

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

The drawState method allows the class to draw its current progress into the given device context within the bounds given. This allows the UI to visualize the progress of the algorithm.

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

The iterate method is the primary interface to the class. This method will perform one iteration of the graph traversal and return its state.

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

This method resets the algorithm so it can start again.

```
	virtual bool weightedGraphSupported() { return false; };
	virtual bool heuristicsSupported() { return false; }
	virtual stringvec heuristicTypesSupported()
		{ stringvec empty; return empty; }
```

These methods inform us if the given class instantiation can support weighted graphs or heuristics. Additionally, the interface for which heuristics are supported are given as strings for the UI.

```
	virtual std::string getClassDescription() = 0;
```

This method returns a description of the class for the UI.

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

These accessors provide access to the map grid which the walker is traversing.

1.7.2 Breadth First Search

The Breadth First Search algorithm is a simple traversal of the graph, where each neighbor is visited before its siblings are. This method does not care about weighted graphs, as it finds the shortest path in steps from start to finish. The largest problem with this algorithm is encountered with large graphs -- this traversal can take a *very* long time.

```
bool BreadthFirstSearch(Node start, Node goal)
{
	Queue open;
	Node  n,  child;
	start.parent = NULL;

	open.enqueue(start);
```

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

            while (n.hasMoreChildren())
            {
                  child = n.getNextChild();
                  if (child.visited())
                        continue;
                  child.parent = n;
                  open.enqueue(child);
            }
      }
      return false;
}
```

Listing 1.4

Take a moment and examine the algorithm in **Listing 1.4**. Notice how it ends in the event that it fails to find a path, unlike the non-look-ahead methods. As mentioned before, this method, as well as all of the other methods we will discuss henceforth, builds the entire path before it takes a single step. Let us look at the method in more detail.

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

First, the method expects a starting node and goal node. It will return true if it finds a path to the goal, and false if it does not. This algorithm will find the entire path before returning. Our implementation of this algorithm (which we will address later) takes place on the inside of the while loop so that we can inspect each iteration.

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

A queue is needed to hold the nodes which we plan to visit, as well as a couple of nodes to keep track of where we are currently, and which child we are about to visit. We make sure to set the parent pointer of the starting node to NULL since this is where we started.

```
      open.enqueue(start);
```

Our queue is primed by adding the start node to it. This is the first node which we will visit since we are starting at this node.

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

We will iterate through every node in the queue until we find the goal, at which point we will abort the loop. If the queue becomes empty and the goal was not found, we cannot reach the goal node from the start node.

```
            n = open.dequeue();
```

Once inside the loop, the next node from the queue is returned and set as our current node.

```
            n.setVisited(true);
```

We will mark this child as visited so that we do not visit it again. Remember, a neighbor of this node has this node as a neighbor. Thus, it will try to visit this node unless it is specified that it has already been visited before.

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

If the current node is the goal node, we will make the path and return success.

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

Next, we will iterate across all the children of the current node.

```
                  child = n.getNextChild();
```

The current child is set as the next child of the current node.

```
                  if (child.visited())
                        continue;
```

If this child has been visited already, we will skip it.

```
                  child.parent = n;
                  open.enqueue(child);
```

This child's parent is set as the current node so that we know how we reached it. Then, the child node will be added to the queue to be visited later.

To summarize, first a queue is built and our starting position placed onto it. Iteration occurs until our queue is empty or until a path is found. During each step of the iteration, a node is removed from our queue, marked as visited, and tested to see if it is our goal. Then, each of our children is added to the queue if they have not been visited before. This last step is of utmost importance. If the children are not tested for prior visitation, we will never leave the first node, as each neighbor of the first node also has the first node as its neighbor. So basically they would go about adding each other to the queue ad

infinitum! Also, by using a queue, we are guaranteeing that each node we add to the queue will be checked in the order they were visited, thereby enforcing the breadth first traversal. Using a stack would make it depth first (with some other modifications, as we will see later). As each node is added to the queue, we also ensure that we set the child's parent to be the node we grabbed from the queue. This allows a path to be built and also shows how we arrived to our current location.

```
MapGridWalker::WALKSTATETYPE BreadthFirstSearchMapGridWalker::iterate()
{
	if(m_open.size() > 0)
	{
		m_n = (MapGridNode*)m_open.front();
		m_open.pop();
		m_n->setVisited(true);
		if(m_n->equals(*m_end))
			return REACHEDGOAL; // we found our path...

		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);

		// The other directional checks go here,
		// but that would take a tremendous amount of space

		// 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);

		return STILLLOOKING;
	}

	return UNABLETOREACHGOAL; // no path could be found
}
```

Listing 1.5

```
void BreadthFirstSearchMapGridWalker::visitGridNode(int x, int y)
{
	// if the node is blocked or has been visited, early out
	if(m_grid->getCost(x, y) == MapGridNode::BLOCKED ||
		m_nodegrid[x][y].getVisited())
		return;

	// we are visitable
	m_open.push(&m_nodegrid[x][y]);
	m_nodegrid[x][y].m_parent = m_n;
}
```

Listing 1.6

Listing 1.5 and **Listing 1.6** contain the important parts of the implementation of the breadth first search as found in our demo. Let us go over this implementation.

```
MapGridWalker::WALKSTATETYPE BreadthFirstSearchMapGridWalker::iterate()
```

The iterate() method begins on the inside of the while loop from our algorithm snippet. It will return the state of the current iteration in order to inform the application whether the algorithm is still looking for a path, has found a path, or has failed to find a path.

```
      if(m_open.size() > 0)
```

First we will check to see if the queue is empty. If it is, there is no valid path from the start node to the goal node.

```
            m_n = (MapGridNode*)m_open.front();
            m_open.pop();
            m_n->setVisited(true);
```

The next node from the queue is returned and set as visited. The setVisited method on the node simply sets a Boolean flag.

```
            if(m_n->equals(*m_end))
                  return REACHEDGOAL; // we found our path...
```

If the new node returned from the queue is the goal node, successful status is returned. This means a path to the goal node has been found.

```
            // 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);
```

Next we check each of our neighbors. In the demo code, a grid represents our graph, so we do some border checking on the grid to ensure we have not overstepped the edge, and if this is true, we visit the node.

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

The method visitGridNode visits the grid node specified at x and y.

```
      if(m_grid->getCost(x, y) == MapGridNode::BLOCKED ||
            m_nodegrid[x][y].getVisited())
            return;
```

First, check to see if the node is blocked or visited. If the node is either blocked or visited, it returns without visiting the node.

```
      m_open.push(&m_nodegrid[x][y]);
      m_nodegrid[x][y].m_parent = m_n;
```

If the node can be visited, it is added to the queue, and the parent of the visited node is set to be the current node in order to track how we arrived there.

```
            return STILLLOOKING;
```

After all of the neighbor nodes are visited, we return STILLLOOKING to indicate further iteration is required to find the goal.

To summarize, the queue is checked first to determine whether it is empty. If it is, we cannot reach the goal from our current location. Otherwise, we remove the first node in line from our queue, and test to see if we are at the goal. If so, we return and iteration stops. If the goal has not been reached, we add each of our neighbor nodes, provided that they exist (we live on a 2D grid and therefore edges occur), they are accessible, and we have not visited them yet. We then return STILLLOOKING so that iteration will continue in the next time-slice. The visitGridNode() method wraps up the parts of the algorithm that take care of all the things which need to happen when a node is visited. It checks to see if the node is blocked or visited, and if so, aborts early. If the node is not blocked or visited, the method pushes the node onto the queue, and sets the parent so we can see how we arrived.

1.7.3 Best First Search

The Best First Search is an optimized Breadth First Search in that is uses a heuristic to choose which nodes to traverse next instead of just traversing them in a sequential order. This method also has the same disregard for weighted graphs, as it only cares about the number of nodes needed to traverse from start to finish. This is a good method, as it is much faster than the Breadth First Search, but it might not always find the shortest path to the goal. Path length will be primarily dependent on the appropriateness of the heuristic chosen.

```
bool BestFirstSearch(Node start, Node goal)
{
      PriorityQueue open;
      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();
                  if (child.visited())
                        continue;
                  child.parent = n;
                  child.setCost = findCost(child, goal);
                  open.enqueue(child);
            }
      }

      return false;
}
```

Listing 1.7

At a first glance, you are probably wondering how this method (**Listing 1.7**) is any different than the one we just discussed. The magic is in the queue type we use. The best first search uses a *priority queue* that is keyed on the perceived cost to the goal. This allows the method to start traversing in a direction towards the goal before it would start investigating nodes that take us away from the goal. Aside from the use of a priority queue, the only other difference is the *cost heuristic*. Let us take a moment to discuss a few common heuristics.

1.7.3.1 Max (dx, dy)

The Max(dx, dy) method uses the maximum of the x distance and the y distance to the goal. Often, this heuristic underestimates the distance to the goal. If the goal is directly above, below, left of, or right of the node (in a grid environment such as ours), the estimate is reasonably accurate. If the node position is diagonal to the goal, the estimate becomes less accurate.

1.7.3.2 Euclidean Distance

The Euclidean distance method uses the standard Euclidean formula to determine the length of the vector from the node to the goal. This formula is: $d = \sqrt{(x_g - x_n)^2 + (y_g - y_n)^2}$ An important point to remember when dealing with square roots is that, not only are they expensive, they require floating point precision. If your costs are integers, you will lose precision and your estimate will be more inaccurate.

1.7.3.3 Manhattan (dx + dy)

The Manhattan (dx + dy) method uses the x distance added to the y distance to the goal. Often this method overestimates the distance to the goal. Like the Max(dx, dy) method, if the goal is directly above, below, left of, or right of the node (in a grid environment such as ours), the estimate is reasonably accurate. If the node position is diagonal to the goal, the estimate becomes less accurate.

```
MapGridWalker::WALKSTATETYPE BestFirstSearchMapGridWalker::iterate()
{
	if(!m_open.isEmpty())
	{
		m_n = m_open.dequeue();
		m_n->setVisited(true);
		if(m_n->equals(*m_end))
		{
			// we found our path...
			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);

		//
		// The other directional checks go here,
		// but that would take a tremendous amount of space
		//

		// 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);
		}

		return STILLLOOKING;
	}

	return UNABLETOREACHGOAL; // no path could be found
}
```

Listing 1.8

```
void BestFirstSearchMapGridWalker::visitGridNode(int x, int y)
{
	// if the node is blocked or has been visited, early out
	if(m_grid->getCost(x, y) == MapGridNode::BLOCKED ||
		m_nodegrid[x][y].getVisited())
		return;

	// we are visitable
	m_nodegrid[x][y].m_parent = m_n;
	m_nodegrid[x][y].m_cost = goalEstimate(&m_nodegrid[x][y]);
	m_open.enqueue(&m_nodegrid[x][y]);
}
```

Listing 1.9

The above implementation is nearly identical to the prior algorithm with the exception of m_open being a priority queue keyed on the heuristic goal estimate. The node's cost is calculated only if it is added to the queue, in which case it is set via the goalEstimate() function. This function implements one of the heuristic methods we discussed above. Let us walk through the code and discuss it in more detail.

```
MapGridWalker::WALKSTATETYPE BestFirstSearchMapGridWalker::iterate()
```

Like all of our implementations, the iterate method does the work, and returns information telling us whether it needs to be called again because it is still searching, whether it found the goal, or whether it cannot find the goal.

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

Just like the Breadth First Search, the first thing to check for is an empty queue. If it is empty and we have not found the goal, we cannot get to the goal from the start position.

```
            m_n = m_open.dequeue();
            m_n->setVisited(true);
```

Next we take the first item off the priority queue and use that as our current node. We also mark it as visited so we do not try to visit it again.

```
            if(m_n->equals(*m_end))
            {
                  // we found our path...
                  return REACHEDGOAL;
            }
```

Next we determine if our current node is, in fact, the goal. If it is, we return that we have reached our goal.

```
            // 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 all of our neighbors, just as we did in the Breadth First Search method. Again, we have a visitGridNode method that does the work of visiting the node for us.

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

As before, it takes an (x, y) coordinate of the node it is to visit on our grid.

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

```
            m_nodegrid[x][y].getVisited())
            return;
```

It checks to see if the node is blocked or already visited, and if either condition is true, it returns without visiting the node.

```
      // we are visitable
      m_nodegrid[x][y].m_parent = m_n;
      m_nodegrid[x][y].m_cost = goalEstimate(&m_nodegrid[x][y]);
      m_open.enqueue(&m_nodegrid[x][y]);
```

If this node can be visited, it sets the parent of the child node as the current node, determines the cost of this node per our heuristic estimate as discussed above, and adds the node to the priority queue. The priority queue automatically sorts the node into its proper place in the queue.

```
            return STILLLOOKING;
```

After we visit all of our neighbor nodes, we return that we need more iteration to find the goal.

1.8 Edsger W. Dijkstra and his Algorithm

Edsger W. Dijkstra was born in 1930 in The Netherlands. He was one of the first to think of programming as a science in itself and actually called himself a programmer by profession in 1957. The Dutch government did not recognize programming as a real profession, however, so he had to re-file his taxes as "theoretical physicist." He won the Turing Award from the Association for Computing Machinery in 1972, and was appointed to the Schlumberger Centennial Chair in Computer Science at the University of Texas in 1984. He also is responsible for developing the prized "shortest-path" algorithm that has been integral to many computer games.

E. Dijkstra's shortest path algorithm is so useful and well-known, that it has simply been dubbed the "shortest path algorithm." It is so popular that if you were to mention pathfinding to most programmers, they would assume you were speaking of this particular algorithm. Interestingly enough, E. Dijkstra's algorithm varies a bit depending on where you look it up.

1.8.1 Three Common Versions of Dijkstra's

Let us analyze three common versions of the Dijkstra's shortest path algorithm in a little more detail. First the algorithm will be shown, and then an example will be walked through for each of the versions.

1.8.1.1 Version One

```
procedure dijkstra(w, a, z, L)
  L(a) := 0
  for all vertices x ≠ a do
     L(x) := ∞
  T := set of all vertices
  // T is the set of vertices whose shortest distance
  // from a has not been found
  while z ∈ T do
    begin
    choose v ∈ T with minimum L(v)
    T := T – {v}
    for each x ∈ T adjacent to v do
      L(x) := min{L(x), L(v) + w(v, x)}
    end
end dijkstra
```

Listing 1.10

Listing 1.10 shows a version of Dijkstra's algorithm where it finds the shortest path from *a* to *z*. In this algorithm, *w* denotes the set of weights where *w*(*i*, *j*) is the weight of the edge between point *i* and *j*, *L*(v) denotes the current minimum length from *a* to *v*. This particular algorithm does not track the actual path from *a* to *z*, just the length of the path. The algorithm works by first initializing L for all vertices, except *a*, to a very large value. It then chooses a vertex with the shortest length, and removes it from the set of all vertices. Then for each adjacent vertex, it calculates the new minimum distance.

1.8.1.2 Version One Example

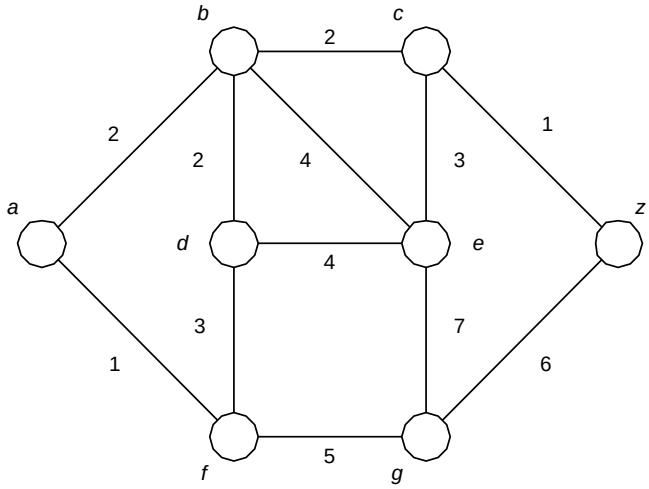


Figure 1.12

For the walkthrough of this algorithm, let us use this simple graph (Fig 1.12) as our example. The vertices are marked *a* through *z*, and the cost for a given edge is labeled nearest the edge center.

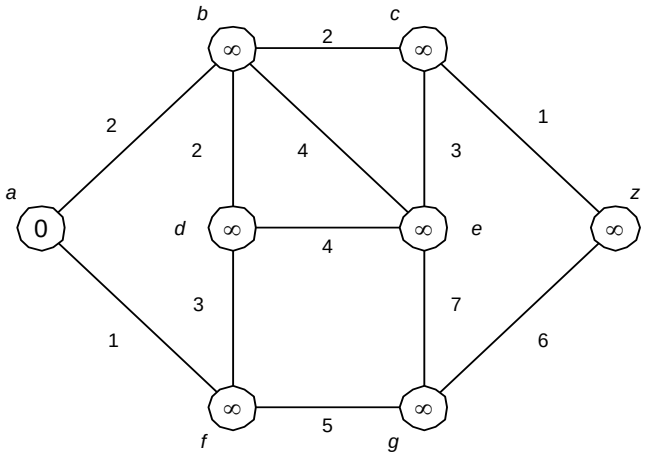


Figure 1.13

When the algorithm begins, it initializes the lengths from *a* to all the other vertices to a very large value. It also places each of the vertices in a list.

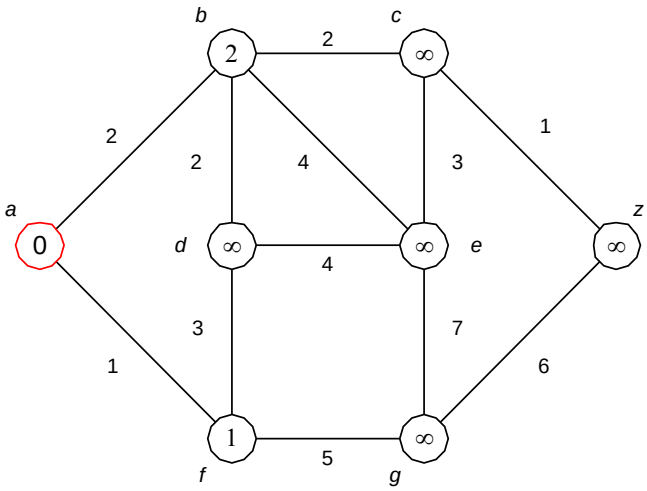


Figure 1.14

In the first iteration, the algorithm naturally selects the a vertex, as it was initialized to 0 during initialization and is lowest. It is removed from the vertex list, and all of the vertices adjacent to a (b and f) have their L values calculated.

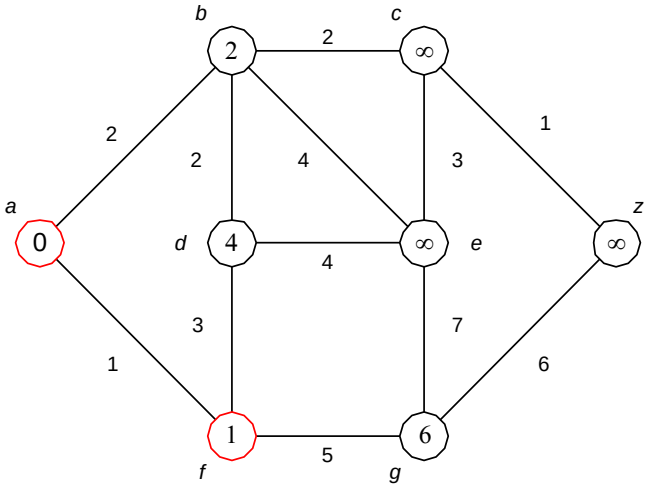


Figure 1.15

In the second iteration, the algorithm chooses vertex *f* as it has the lowest cost, and it is removed from the vertex list. The adjacent vertices (*d* and *g*) have their *L* values calculated, and the algorithm moves on.

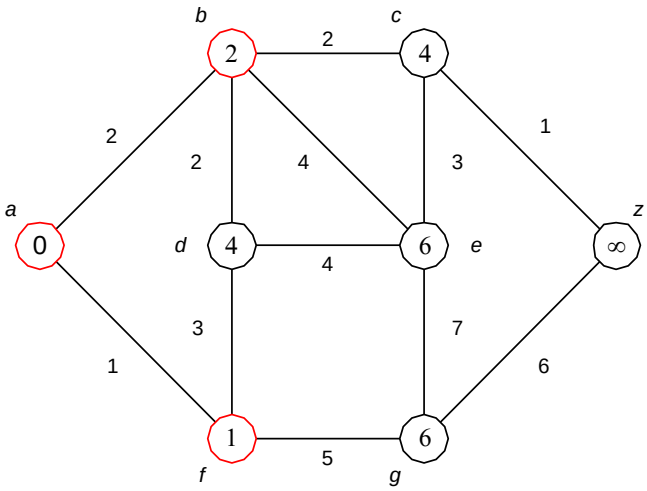


Figure 1.16

In the third iteration, the algorithm chooses vertex *b*, as it has the lowest cost, and it is removed from the vertex list. The adjacent vertices (*d*, *e*, and *c*) then have their *L* values calculated. The *z* vertex is now the only vertex that has not had an *L* value calculated.

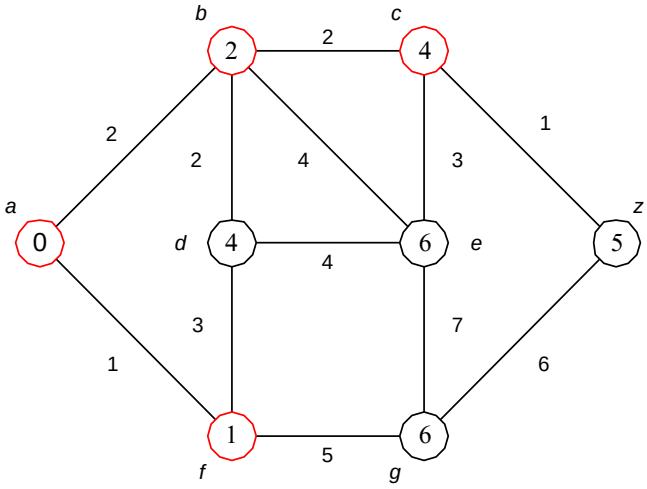


Figure 1.17

In the fourth iteration, the algorithm chooses the *c* vertex, and it is removed from the vertex list. The adjacent vertices (*z* and *e*) have their *L* values calculated, and we now have all of the vertices in the graph with an *L* value. The algorithm would next pick *d*, and then finally *z*. When *z* is removed from the vertex list, the algorithm stops and it is seen that the shortest path from *a* to *z* is 5 units long. Of course, without going back and looking, we have no way of knowing that path to take is *a*-*b*-*c*-*z*, so it would be a good idea to keep track of this. We do that in our algorithm, as you will see later.

1.8.1.3 Version Two

```
Given the arrays distance, path, weight and included, initialize included[source]
to true and included[j] to false for all other j.
Initialize the distance array via the rule
  if j = source
    distance[j] = 0
  else if weight[source][j] != 0
    distance[j] = edge[source][j]
  else if j is not connected to source by a direct edge
    distance[j] = Infinity
for all j
Initialize the path array via the rule
  if edge[source][j] != 0
    path[j] = source
  else
    path[j] = Undefined
Do
  Find the node J that has the minimal distance
    among those nodes not yet included
  Mark J as now included
  For each R not yet included
    If there is an edge from J to R
      If distance[j] + edge[J][R] < distance[R]
        distance[R] = distance[J] + edge[J][R]
          path[R] = J
While all nodes are not included
```

Listing 1.11

The algorithm in Listing 2.2 utilizes 3 arrays to do its work. It is similar to the former version in that the distance array is the *L* value, but it differs in that it tracks the actual path to the goal as well. It also uses an array to mark vertices that have been chosen rather than removing them from the list. The algorithm runs to completion when all nodes have been included. We could shorten the algorithm easily by changing the while loop to "while the goal node is not included" since once the goal node is included, we have the shortest path.

1.8.1.4 Version Two Example

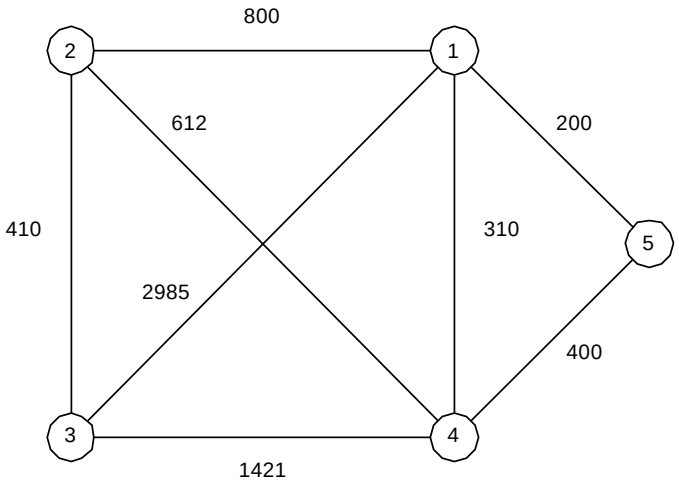


Figure 1.18

Let us use the graph in Figure 1.18 for this version of the algorithm. We will walk through the iteration of the algorithm and examine the contents of the various arrays along the way. The walk-through for this graph will use our algorithm starting at vertex 1.

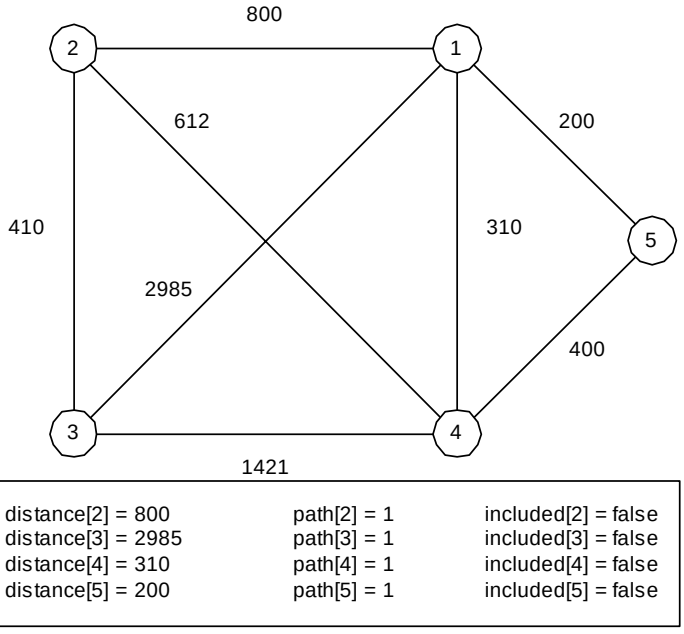
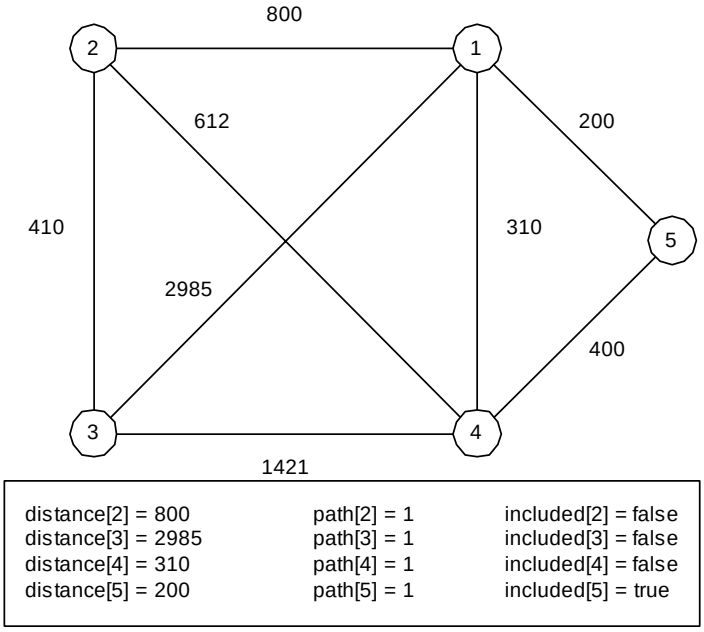


Figure 1.19

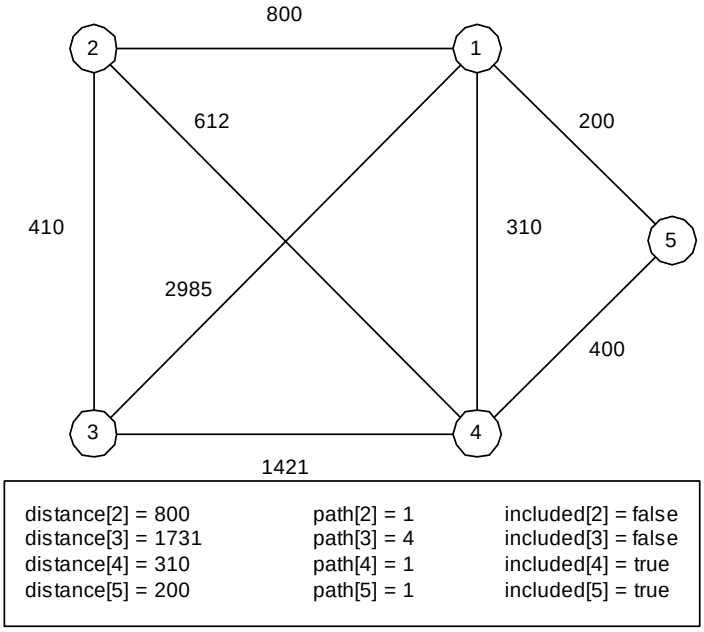
First we initialize our distance, path, and included arrays. All of the path array locations are set to 1 (where we started) and none of the other vertices are marked as included.



distance[2] = 800	path[2] = 1	included[2] = false
distance[3] = 2985	path[3] = 1	included[3] = false
distance[4] = 310	path[4] = 1	included[4] = false
distance[5] = 200	path[5] = 1	included[5] = true

Figure 1.20

In the first iteration, we find the vertex with the smallest distance, which is vertex 5. We then mark that vertex as included, and check to see if the distances from vertex 1 to vertex 5's neighbors are smaller than the one already stored, which they are not.



distance[2] = 800	path[2] = 1	included[2] = false
distance[3] = 1731	path[3] = 4	included[3] = false
distance[4] = 310	path[4] = 1	included[4] = true
distance[5] = 200	path[5] = 1	included[5] = true

Figure 1.21

In the second iteration, we see that vertex 4 has the shortest distance and is not included. We mark it as included, and then update our distances. This is because the distance from vertex 1 to vertex 4 to vertex 3 is shorter than the distance from vertex 1 directly to vertex 3. We also update the path to vertex 3 to indicate that travel through vertex 4 from the source is the shortest path.

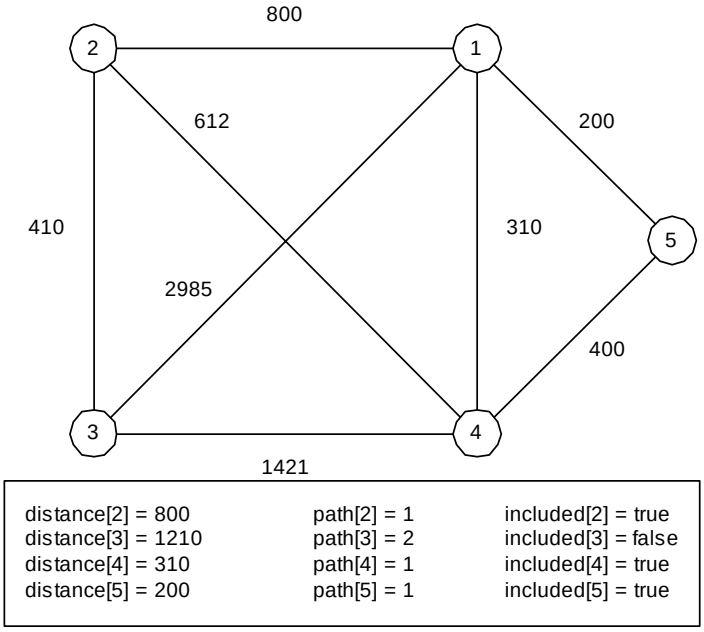


Figure 1.22

In the third iteration, we see that vertex 2 has the shortest distance, so we mark it included. We also see that by traveling through vertex 2 to vertex 3, it is shorter than traveling through vertex 4, so we update the distance for vertex 3 as well as the path.

In the fourth iteration, all that is left is vertex 3, so we mark it as included. Nothing changes in terms of path or distance, so we now have the shortest distance as well as the path to all vertices from vertex 1.

1.8.1.5 Version Three

```
void Dijkstra( Table T )
{
      Vertex V, W;

      while( true )
      {
            V = Smallest Unknown Distance Vertex;
            if( V == Not A Vertex )
                  break;

            T[ V ] .Known = true;
            for Each W Adjacent To V
                  if( !T[ V ].Known )
                  {
                        // Update W.
                        decrease (  T[ W ].Dist To
                                          T[ V ].Dist + C ( V, W );
                        T[ W ].Path = V;
                  }
      }
}
```

Listing 1.12

This algorithm is very similar to the previous algorithm. It maintains a list of vertices that are known, the distance to each vertex, as well as the path to each vertex. The biggest difference is more of an architectural change. Rather than keeping data in arrays, a table is used to manage the weights, and Vertex structures are used to store the related path data. This method also finds the shortest path to each vertex in the graph.

1.8.1.6 Version Three Example

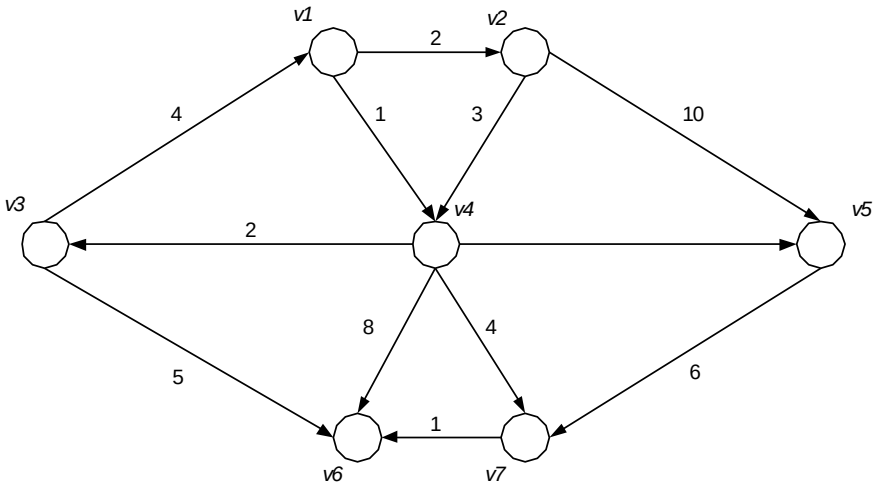


Figure 1.23

For this last example, we will take a look at how things change when using a directed graph rather than a non-directed graph. The graph above is a directed graph where travel is only allowed in the direction of the arrows. Let us traverse this graph starting at v_1.

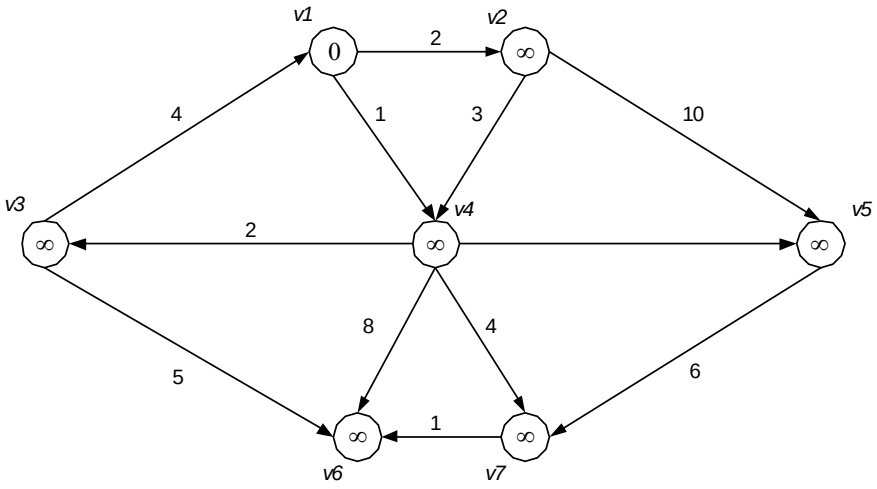


Figure 1.24

First we initialize all of the nodes to unknown, and the distances to infinity. We also set the parent vertex, for each vertex, to 0.

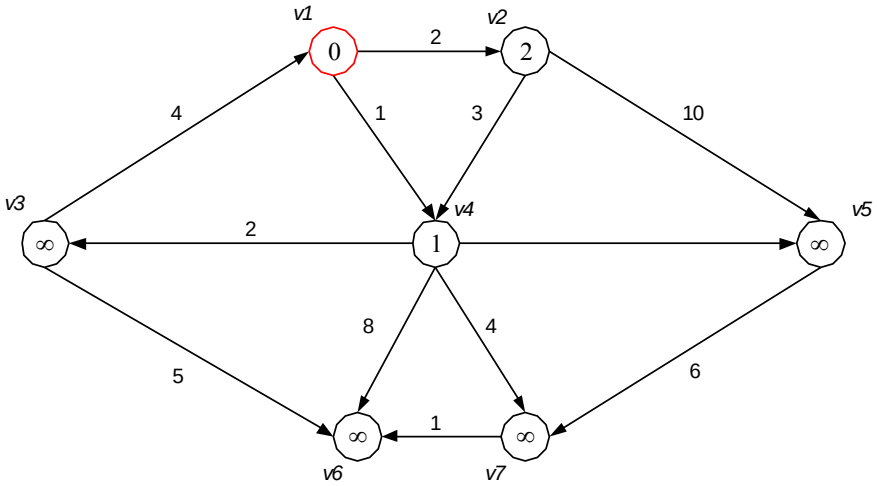


Figure 1.25

In the first iteration, we mark the starting vertex as known, and update the distance members of v_2, and v_4. We also set both parents to v_1.

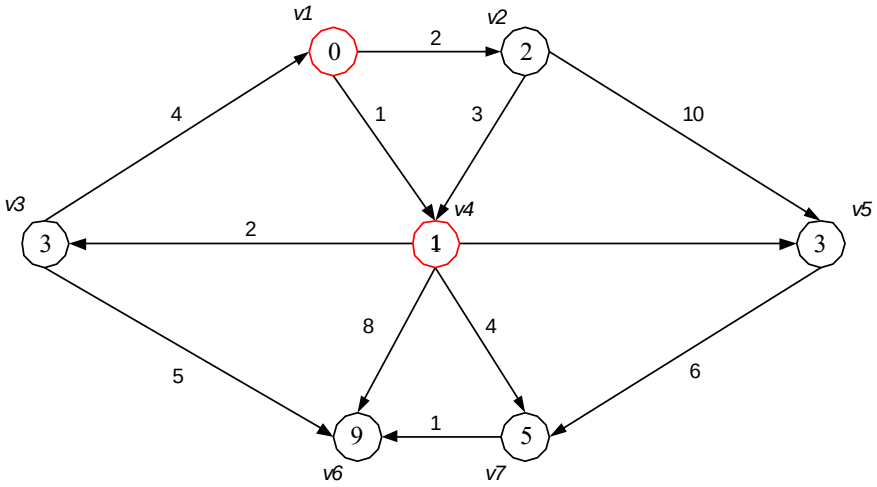


Figure 1.26

In the second iteration, we mark v_4 as known, as it has the shortest distance so far, and update all of its neighbor's distances. For those neighbor vertices it does set the distance for, v_4 makes itself their parent vertex as well.

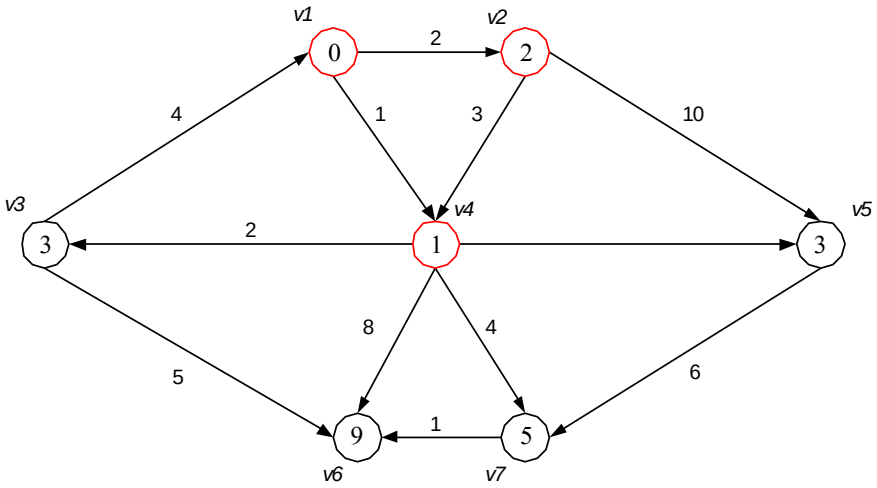


Figure 1.27

In the third iteration, we mark v_2 as known, and update all of its neighbor's distances. In this case, there are no neighbors that need to be updated.

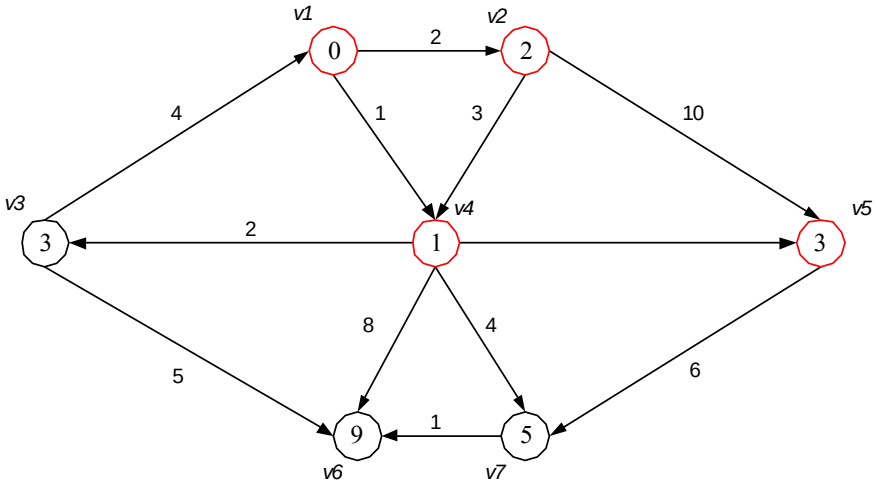


Figure 1.28

In the fourth iteration, we mark v5 as known, and try to update neighbors. Again, no updates are needed.

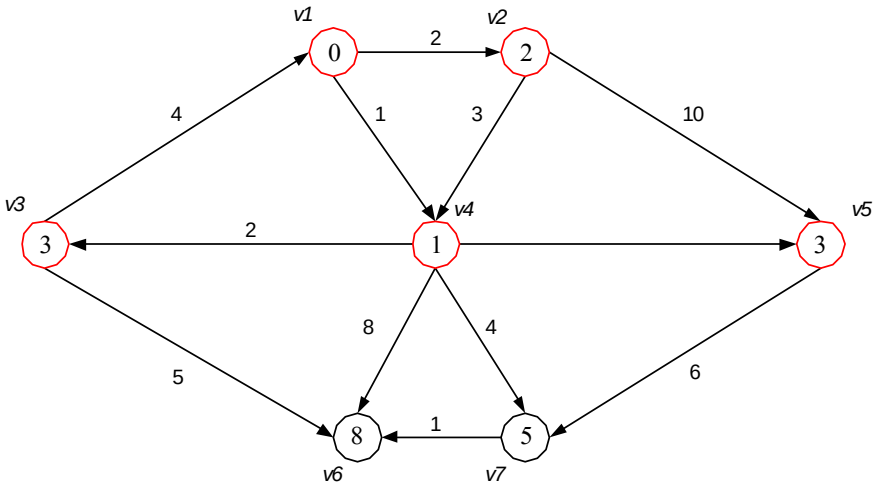


Figure 1.29

In the fifth iteration, we mark v_3 as known, and update neighbors. This time, we actually find a shorter route to v_6, and update its distance and make v_3 its parent vertex.

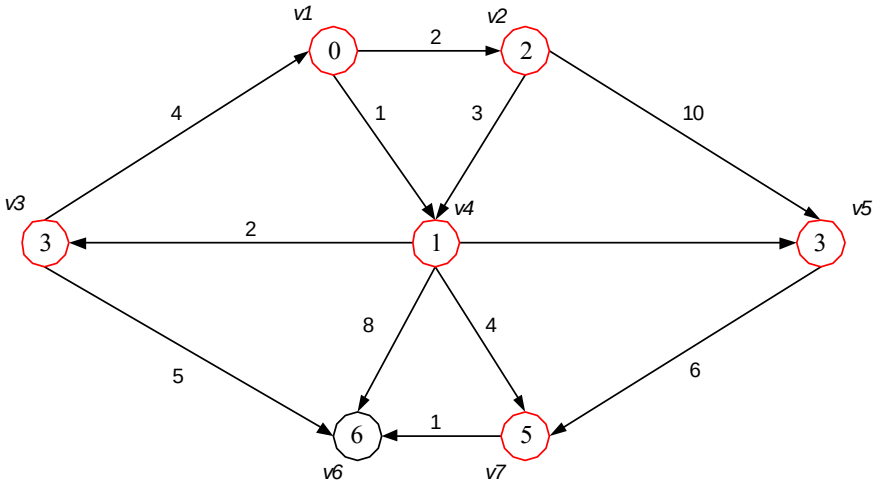


Figure 1.30

In the sixth iteration, we mark v_7 as known, and update its neighbors. Again we find a shorter path to v_6, so it is updated and v_7 is made its parent vertex. The last iteration, we mark v_6 as known, and no updates are needed. Now we are done.

1.8.2 Our Version of the Algorithm

```
bool DijkstraSearch(Node start, Node goal)
{
      PriorityQueue open;
      Node  n,  child;

      start.parent = NULL;
      start.cost = 0;

      open.enqueue(start);
      while(!open.isEmpty())
      {
            n = open.dequeue();
            n.setVisited(true);
            if (n == goal)
            {
                  makePath();
                  return true;
            }

            while (n.hasMoreChildren())
            {
                  child = n.getNextChild();
                  COSTVAL newcost = n.cost + cost(n, child);
                  if (child.visited())
                        continue;
                  if (open.contains(child) && child.cost <= newcost)
                        continue;
                  child.parent = n;
                  child.cost = newcost;
                  if (!open.contains(child))
                        open.enqueue(child);
                  else
                        open.reenqueue(child);
            }
      }

      return false;
}
```

Listing 1.13

Our version of the algorithm is very much like the last two versions we studied. That is, we will keep track of the shortest distance we have found thus far at each node and also keep track of which nodes we have visited. The biggest difference is how we pick which node to next traverse through. We use a priority queue to sort our unvisited nodes in order of their cost. We then grab the top one off of the queue and do our traversal. Let us discuss this particular version in more detail since it is the version we will be using in our demo.

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

Like the other algorithms, this one expects a start node and a goal node, and returns if it was capable of finding a path.

```
      PriorityQueue open;
      Node  n,  child;
```

As in Best First Search, a priority queue is used to keep track of the nodes we need to visit, and we will have a current node as well as the current child we are visiting of the current node.

```
      start.parent = NULL;
      start.cost = 0;
```

We will start out by setting the parent of our starting node to NULL to denote it is indeed the start. We also set the cost to 0.

```
      open.enqueue(start);
```

Next we will initialize the queue by adding our start node to it since we will want to visit it first.

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

While the queue is not empty, we will iterate through all the children of each node in the queue. If the queue empties before we find the goal, there is no path from the start node to the goal.

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

For each iteration, we will grab a node off the queue and make it our current node. We also mark this node as visited so we do not visit it again. If this node is the goal node, we found the path, so we make it and return success.

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

We then iterate across each of the current nodes' children.

```
                  child = n.getNextChild();
                  COSTVAL newcost = n.cost + cost(n, child);
```

For each child, we compute the cost from this node to the child and add it to the cost which this node has stored as the computed cost from the start node to it. This allows us to keep track of the total cost it takes to get from the start node to every other node as we visit it.

```
                  if (child.visited())
                        continue;
                  if (open.contains(child) && child.cost <= newcost)
                        continue;
```

Here is where the algorithm starts to differ from the other algorithms we have discussed so far. Like the other algorithms, if we have visited this child node, we do not visit it again. But if we have not visited this child, but we have already determined that we need to visit it, we check to see if the cost we've computed previously for this child is less than the cost we just computed. This allows us to update the cost to this particular child if we found a shorter path to this child. If we have a cost for this child computed already and it is shorter than the cost we just found, we ignore this path to the child node since we have a better one already.

```
                  child.parent = n;
                  child.cost = newcost;
                  if (!open.contains(child))
                        open.enqueue(child);
                  else
                        open.reenqueue(child);
```

If we determine that we want to visit this child, we set its parent to be our current node, set its cost to be our computed cost, and if the queue does not already contain the child, we add it. If the queue does contain the node, we inform the queue that it needs to reinsert the child into its proper position now that its cost has changed.

1.8.3 The Implementation of Our Version

```
MapGridWalker::WALKSTATETYPE DijkstrasMapGridWalker::iterate()
{
      if(!m_open.isEmpty())
      {
            m_n = m_open.dequeue();
            m_n->setVisited(true);
            if(m_n->equals(*m_end))
            {
                  // we found our path...
                  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);
            }
```

```
            //
            // All other directions here, but that takes up too much space
            //

            // 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);
            }

            return STILLLOOKING;
      }

      return UNABLETOREACHGOAL;
}
```

Listing 1.14

```
void DijkstrasMapGridWalker::visitGridNode(int x, int y)
{
      int newcost;
      bool inqueue;

      if(m_grid->getCost(x, y) == MapGridNode::BLOCKED ||
            m_nodegrid[x][y].getVisited())
            return;

      newcost = m_n->m_cost + m_grid->getCost(x, y);

      inqueue = m_open.contains(&m_nodegrid[x][y]);

      if(inqueue && m_nodegrid[x][y].m_cost <= newcost)
      {
            // 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_cost = newcost;

            if(!inqueue)
            {
                  m_open.enqueue(&m_nodegrid[x][y]);
            }
            else
            {
                  m_open.remove(&m_nodegrid[x][y]);
                  m_open.enqueue(&m_nodegrid[x][y]);
            }
      }
}
```

Listing 1.15

Here is the actual implementation from our demo. Similar to the algorithms we discussed already, 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 of its neighbors, and continue on until we find the goal node. Let us go over our implementation of the algorithm in more detail.

```
MapGridWalker::WALKSTATETYPE DijkstrasMapGridWalker::iterate()
```

As in all our implementations, the iterate method starts inside the while loop of our algorithm snippet. It returns a status of needing more iteration because it is still looking, whether it found the goal, or if it cannot find the goal.

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

We begin by checking to see if the queue is empty. If it is, we cannot find a path from the start to the goal. Otherwise we begin another iteration.

```
            m_n = m_open.dequeue();
            m_n->setVisited(true);
```

We grab the next node off the queue and make it our current node. We also mark that node as visited so that we do not visit it again.

```
            if(m_n->equals(*m_end))
            {
                  // we found our path...
                  return REACHEDGOAL;
            }
```

If the current node is, in fact, the goal node, we have reached our goal and 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 check each of the current node's neighbors. Again we make sure to stay within the bounds of our grid and let the visitGridNode method do the work.

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

This method takes an (x, y) coordinate and visits the corresponding grid node.

```
      if(m_grid->getCost(x, y) == MapGridNode::BLOCKED ||
            m_nodegrid[x][y].getVisited())
```

```
            return;
```

First it checks to see if the node in question is blocked or already visited. If it is, it returns and does not visit the node.

```
      newcost = m_n->m_cost + m_grid->getCost(x, y);

      inqueue = m_open.contains(&m_nodegrid[x][y]);
```

Next it computes the cost to this child node via the current node. Also, we check to see if the node in question is already in our queue.

```
      if(inqueue && m_nodegrid[x][y].m_cost <= newcost)
      {
            // do nothing... we are already in the queue
            // and we have a cheaper way to get there...
      }
```

If we are already in the queue, and the new cost we computed is greater than the cost the child node already has, we ignore the node since we already have a cheaper way to get there.

```
            m_nodegrid[x][y].m_parent = m_n;
            m_nodegrid[x][y].m_cost = newcost;
```

If we determine we have a cheaper way to get to the child node, we set its parent to the current node, and its cost to the cost we computed for it.

```
            if(!inqueue)
            {
                  m_open.enqueue(&m_nodegrid[x][y]);
            }
            else
            {
                  m_open.remove(&m_nodegrid[x][y]);
                  m_open.enqueue(&m_nodegrid[x][y]);
            }
```

If the child node is not in the queue, we simply add it. If it is in the queue, we remove it and add it again so it can be put in its proper place.

```
            return STILLLOOKING;
```

After we visit all of the neighbor nodes of the current node, we return STILLLOOKING to indicate that we need more iterations to find the goal.

1.9 Look-Ahead Recursive Methods

As discussed earlier, there are some look-ahead pathfinding methods that are most easily implemented with recursion. The prime example we discussed is the Depth First Search. Let us take a look at this algorithm, and discuss it in detail.

1.9.1 Depth First Search

The Depth First Search algorithm is a simple traversal for weighted or non-weighted graphs, in which siblings are visited before neighbors. The method has a few caveats. The Depth First Search method is recursive in nature, and unless the depth to which it searches is constrained, it will search to an infinite depth in an attempt to find its goal. This method also has a tendency to wrap around unless we constrain it to moving towards the goal, if at all possible.

```
bool DepthFirstSearch(Node node, Node goal, int depth, int length)
{
      int d;

      if (node == goal)
      {
            makePath();
            return true;
      }

      if (depth < MAXDEPTH)
      {
            while (node.hasMoreChildren())
            {
                  child = node.getNextChild();
                  d = node.dist + node.getCost(child);
                  if (!isTowardsGoal(node, child, goal))
                        continue;
                  if (child.visited() || d > child.cost)
                        continue;
                  child.parent = node;
                  child.visited = true;
                  child.cost = d;
                  if (DepthFirstSearch(child,  goal, depth+1,  child.cost))
                        return true;
                  child.visited = false;
            }
      }

      return false;
}
```

Listing 1.16

After looking over the algorithm in **Listing 1.16**, you should notice it is recursive in nature rather than iterative. Use of recursion allows us to leverage the call stack rather than maintaining our own stack. We might have implemented this method using iteration, and it would have looked much like the others except for its use of a stack rather than a queue. However, using recursion for this method is much more elegant. Let us go over this algorithm in a little more detail.

```
bool DepthFirstSearch(Node node, Node goal, int depth, int length)
```

The recursive method DepthFirstSearch takes a node to search from, a goal to get to, the depth to search to, and the current cost. Each call to this method will change the node, depth, and length parameters while the goal will remain the same.

```
      if (node == goal)
      {
            makePath();
            return true;
      }
```

If the node passed in is the goal, we make the path and return our success. The true return value will trigger a full recursive unroll to get us out and back to the initial caller of the method.

```
      if (depth < MAXDEPTH)
```

If we have not exceeded our depth, we search further, otherwise we will return false to say we did not find the goal.

```
            while (node.hasMoreChildren())
```

If we have not exceeded our depth, we will iterate across all the passed in node's children.

```
                  child = node.getNextChild();
                  d = node.dist + node.getCost(child);
```

For each child, we will compute the distance to this child by adding our passed in node's pre-computed cost to the cost of getting to the child node.

```
                  if (!isTowardsGoal(node, child, gloal))
                        continue;
```

Here we do a little trickery to keep the algorithm from doing loops. We check to see if the child helps us to get towards the goal. The implementation of isTowardsGoal is graph specific, but it will return true if the passed-in node is closer to the goal and false if it is not. If the child does not take us closer to the goal, we do not traverse it since it might take us on a crazy, winding path.

```
                  if (child.visited() || d > child.cost)
                        continue;
```

If the child has been visited already or the child's cost is cheaper than the computed cost, we also skip this child.

```
                    child.parent = node;
                    child.visited = true;
                    child.cost = d;
```

Next we set the child's parent to be the node passed in, mark the child as visited, and set its cost to be the cost we computed.

```
                    if (DepthFirstSearch(child,  goal, depth+1,  child.cost))
                          return true;
```

We then recursively call the method again using the child node as the node to pass in, increment the depth, and pass in the child's cost as the length. If this returns true, we found the goal and return immediately. This will unroll the recursive stack back to the initial caller.

```
                    child.visited = false;
```

Here is another tricky bit. After the recursive call, we mark the child as unvisited again, since we might need to go through it via another depth traversal.

To summarize, we start by calling DepthFirstSearch() and pass the start node, the end node, a depth of 1, and a length of 0. The algorithm would first check the node passed in to see if it is the goal. If so, we make the path and return all the way out of the recursive stack. Otherwise, if our depth is less than the max depth we wish to search to, we iterate through each child. If the child is in the direction of the goal, the child has not been visited, and the cost to the child is less than the child's current remembered cost, we recursively call DepthFirstSearch on that child, incrementing depth and passing the cost to the child. It is important to be sure that the cost to the child is better than the last cost, and that we are moving in the direction of the goal. Otherwise, the algorithm will create curly, winding paths that lead nowhere. Also, it is important to mark nodes as visited as we traverse into the graph, and unmark them on our way back out. This is so that we do not visit the same node more than once on the way into the graph, but we are sure to try them again if a search to a given depth fails. An added improvement that could be made is to iteratively increase the MAX_DEPTH value to enable searching deeper into the graph until we find a goal. One might also attempt to calculate a beginning MAX_DEPTH using a heuristic goal estimate and implement the iterative deepening from that starting point so as to reduce the number of deepening iterations.

Conclusion

In this chapter we have discussed pathfinding at its most basic. We talked about graphs, what they are, and why they are important in pathfinding. We also examined single step path traversals, as well as iterative and recursive methods of pathfinding. These latter methods determine optimal paths through the graph to the goal. Finally, we looked at some specific implementations for some of the common algorithms used in pathfinding. In the next chapter we will expand our understanding of pathfinding by looking at more complex pathfinding methods such as A* and hierarchical pathfinding.

One thing you hopefully recognized is that even the simplest pathfinder requires decision making; even when that decision was as simple as "we hit a barrier, so try moving in some random direction to get around it". As we progressed, we saw that the means for improving the efficiency of the search and the ability to circumnavigate obstacles involved more complex decision making criteria (such as the various heuristics we mentioned). Again, while this may not be the pure Decision Making AI that we will learn about later on, you can probably understand why some programmers tend to lump everything together into a single catch-all AI category (which we made efforts to define at the outset) while others might consider this just a branch on a larger tree. In a sense, they are both right. After all, the job of the pathfinder is to make an entity move from point A to point B in a manner such that, on screen, it looks like the entity "figured out" how to get there in the shortest or quickest way possible. To the player, the entity certainly looks like it knows what it is doing and is therefore exhibiting some manner of intelligence. According to our original definition of artificial intelligence, this certainly fits the bill. Keep these thoughts in mind as you work your way through the rest of the course.