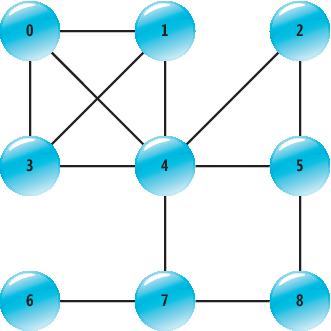
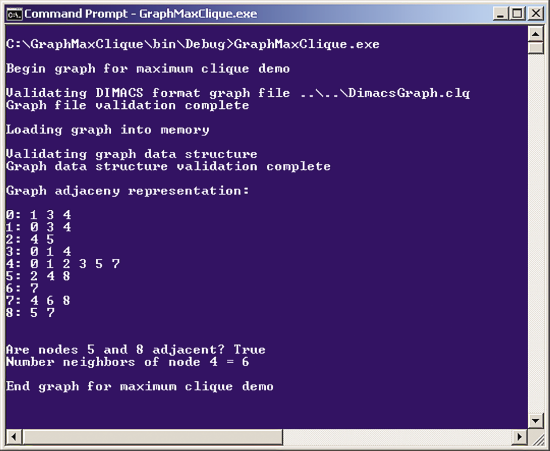
In this month’s column, I present the design, a C# language implementation and testing techniques for a graph data structure that can be used to solve the maximum clique problem. The graph code can also be used for many other problems, as I’ll explain.

So, just what is the maximum clique problem and why might it be relevant to you? A clique is a subset of a graph where every node is connected to every other node. Take a look at the graph representation in **Figure 1**. Nodes 2, 4 and 5 form a clique of size three. The maximum clique problem is to find the clique with the largest size in a graph. The maximum clique for the graph in **Figure 1** is the node set { 0, 1, 3, 4 }, which has size four.

  
**Figure 1 A Graph for the Maximum Clique Problem**

The maximum clique problem is encountered in a wide range of applications, including social network communication analysis, computer network analysis, computer vision and many others. For graphs of even moderate size, it turns out that the maximum clique problem is one of the most challenging and interesting problems in computer science. The techniques used to solve the maximum clique problem—which include tabu search, greedy search, plateau search, real-time parameter adaptation and dynamic solution history—can be used in many other problem scenarios. In short, code that solves the maximum clique problem can be directly useful to you, and the advanced techniques employed in the algorithm can be helpful for solving other difficult programming problems.

A complete solution to the maximum clique problem is too long to present and explain in one article, so I’ll present the solution over several articles. The first step for solving the maximum clique problem is to design, implement and test a data structure that can efficiently store the graph under analysis in memory. The console application in **Figure 2** shows you where I’m headed in this column.

  
**Figure 2 Graph Loading and Validation**

With some WriteLine statements removed, the code that produced the run shown in **Figure 2** is:

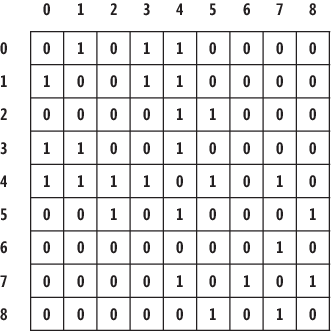
1. **string** graphFile = "..\\..\\DimacsGraph.clq";
2. MyGraph.ValidateGraphFile(graphFile, "DIMACS");
4. MyGraph graph = **new** MyGraph(graphFile, "DIMACS");
6. graph.ValidateGraph();
7. Console.WriteLine(graph.ToString());
9. Console.WriteLine("\nAre nodes 5 and 8 adjacent? " +
10. graph.AreAdjacent(5,8));
11. Console.WriteLine("Number neighbors of node 4 = " +
12. graph.NumberNeighbors(4));

The data for the **Figure 1** graph is stored in an external text file named DimacsClique.clq, which uses a standard format called DIMACS. I’ll explain the DIMACS file format shortly. My demo program begins by validating the source file, then instantiates a graph data structure using the data file. After the graph has been instantiated, I validate the internal representation and display it in a human-friendly image. As you’ll see, an efficient internal representation of a graph is critically important for the maximum clique problem. The demo program finishes by calling a method that determines if two nodes are adjacent, nodes 5 and 8 in this case, and by calling a method that returns the number of neighbors a node has, for node 4 in this case.

I’ll walk you through the code that generated the **Figure 2** output line by line. The complete source code for the demo program is available at [code.msdn.microsoft.com/mag201110TestRun](http://code.msdn.microsoft.com/mag201110TestRun). The code is written in C#, but you should be able to follow me if you have intermediate-level programming skills in any modern high-level language. The graph code presented here lays the foundation for solving the maximum clique problem in upcoming articles and should be a useful addition to your developer, tester and software project management toolkits.

## A Bit Matrix

There are several common ways to represent an unweighted (graph edges aren’t assigned priorities of some sort), undirected (edges don’t have a direction from one node to another) graph in memory. For the maximum clique problem, representing a graph using a bit matrix provides an excellent combination of space and performance efficiencies. **Figure 3** shows a bit matrix that corresponds to the example graph. Even though we’re dealing with an undirected graph, it’s common to call the vertical indices the from-nodes and the horizontal indices the to-nodes. A value of 1 means there’s an edge between the corresponding nodes; a 0 value indicates no edge between nodes. Notice that the matrix is symmetric and that we assume nodes aren’t adjacent to themselves.

  
**Figure 3 A Bit Matrix Graph Representation**

The primary advantage of a bit matrix over alternative designs is that it allows fast adjacency lookups, which often dominate the runtime of many graph algorithms, including the maximum clique problem. If implemented crudely, the primary disadvantage of a bit matrix is memory usage. For example, if the 9x9 matrix in **Figure 3** was implemented as a two-dimensional array of 4 byte integers or Booleans, the matrix would require 9 \* 9 \* 4 = 324 bytes. But because each value in a bit matrix can be only 0 or 1, we can use the bits of an integer to store up to 32 values per integer. In this example, if we imagine that the low-order bit is on the right, the first row can be stored as a single 32-bit integer 00000000-00000000-00000000-10110000, which has the decimal value of 128 + 32 + 16 = 176. So if each row of the matrix is stored as a single integer where the bits of the integer are used to represent the presence or absence of an edge between nodes, the 9x9 matrix would require only 36 bytes.

In older programming languages, you’d have to implement a bit matrix from scratch using low-level bit operators such as left-shift, bitwise-or and so on. But the Microsoft .NET Framework System.Collections namespace has a BitArray type that makes implementing a program-defined BitMatrix type easy. A BitMatrix class can be defined as shown in **Figure 4**.

**Figure 4 A BitMatrix Class**

1. **private** **class** BitMatrix
2. {
3. **private** BitArray[] data;
4. **public** **readonly** **int** Dim;
6. **public** BitMatrix(**int** n)
7. {
8. **this**.data = **new** BitArray[n];
9. **for** (**int** i = 0; i < data.Length; ++i) {
10. **this**.data[i] = **new** BitArray(n);
11. }
12. **this**.Dim = n;
13. }
14. **public** **bool** GetValue(**int** row, **int** col)
15. {
16. **return** data[row][col];
17. }
18. **public** **void** SetValue(**int** row, **int** col, **bool** value)
19. {
20. data[row][col] = value;
21. }
22. **public** **override** **string** ToString()
23. {
24. **string** s = "";
25. **for** (**int** i = 0; i < data.Length; ++i) {
26. **for** (**int** j = 0; j < data[i].Length; ++j) {
27. **if** (data[i][j] == **true**)
28. s += "1 ";
29. **else**
30. s += "0 ";
31. }
32. s += Environment.NewLine;
33. }
34. **return** s;
35. }
36. }

The BitMatrix class represents a square matrix and is essentially an array of BitArray array objects. I declare the BitMatrix class with private scope because I intend to embed it within a graph class definition rather than use it as a standalone class. The BitMatrix constructor accepts a parameter n that’s the dimension of an nxnmatrix, allocates a column of size n of BitArray array objects and then instantiates each BitArray using size n.  Because there’s no bit type in the .NET Framework, the values in a BitArray—and therefore in the BitMatrix class—are exposed as type bool, as you can see in the SetValue method. Notice that to keep my code short, I’ve removed normal error checking.

Using the BitMatrix could look like:

1. BitMatrix matrix = **new** BitMatrix(9);
2. matrix.SetValue(5, 8, **true**);
3. matrix.SetValue(8, 5, **true**);
4. **bool** connected = matrix.GetValue(2, 6);

The first line creates a 9x9 BitMatrix object initially set to all false (or zeros) to represent an unweighted, undirected graph with nine nodes. The second line sets row 5, column 8 to true/1 to indicate there’s an edge between node 5 and node 8. The third line sets row 8, column 5 to true/1 so that the graph edge representation is consistent. The fourth line fetches the value at row 2, column 6, a value indicating whether or not there’s an edge between nodes 2 and 6, which would be false/0. Notice that determining whether or not two nodes are adjacent is just a quick array lookup.

## A Graph Class

With a BitMatrix class in hand, it’s easy to  define an efficient graph class suitable for the maximum clique problem and many other graph-related problems. The structure of a graph class is presented in **Figure 5**. The graph class has dependencies on namespaces System, System.IO and System.Collections. For the example program, I placed the graph class directly inside the console app, but you may want to place the code in a class library.

**Figure 5 A Graph Class Definition**

1. **public** **class** MyGraph
2. {
3. **private** BitMatrix data;
4. **private** **int** numberNodes;
5. **private** **int** numberEdges;
6. **private** **int**[] numberNeighbors;
8. **public** MyGraph(**string** graphFile, **string** fileFormat)
9. {
10. **if** (fileFormat.ToUpper() == "DIMACS")
11. LoadDimacsFormatGraph(graphFile);
12. **else**
13. **throw** **new** Exception("Format " + fileFormat + " not supported");
14. }
16. **private** **void** LoadDimacsFormatGraph(**string** graphFile)
17. {
18. // Code here
19. }
21. **public** **int** NumberNodes
22. {
23. get { **return** **this**.numberNodes; }
24. }
26. **public** **int** NumberEdges
27. {
28. get { **return** **this**.numberEdges; }
29. }
31. **public** **int** NumberNeighbors(**int** node)
32. {
33. **return** **this**.numberNeighbors[node];
34. }
36. **public** **bool** AreAdjacent(**int** nodeA, **int** nodeB)
37. {
38. **if** (**this**.data.GetValue(nodeA, nodeB) == **true**)
39. **return** **true**;
40. **else**
41. **return** **false**;
42. }
44. **public** **override** **string** ToString()
45. {
46. // Code here
47. }
49. **public** **static** **void** ValidateGraphFile(**string** graphFile, **string** fileFormat)
50. {
51. **if** (fileFormat.ToUpper() == "DIMACS")
52. ValidateDimacsGraphFile(graphFile);
53. **else**
54. **throw** **new** Exception("Format " + fileFormat + " not supported");
55. }
57. **public** **static** **void** ValidateDimacsGraphFile(**string** graphFile)
58. {
59. // Code here
60. }
62. **public** **void** ValidateGraph()
63. {
64. // Code here
65. }
67. // -------------------------------------------------------------------
68. **private** **class** BitMatrix
69. {
70. // Code here
71. }
72. // -------------------------------------------------------------------
74. } // Class MyGraph

The graph class definition starts with:

1. **public** **class** MyGraph
2. {
3. **private** BitMatrix data;
4. **private** **int** numberNodes;
5. **private** **int** numberEdges;
6. **private** **int**[] numberNeighbors;
7. ...

I name the class MyGraph. It’s somewhat tempting to try to define an all-purpose graph class, but there are so many variations of graphs that it’s a better idea to define different graph classes for different kinds of problems. The graph class I define here is aimed at solving the maximum clique and related problems, so I could’ve named the class something like MaxCliqueGraph. The class has four data fields. The first is a BitMatrix object as described in the previous section. The numberNodes and numberEdges fields hold the number of nodes (nine in the example) and the number of undirected edges (13 in the example) in the graph.

When solving many graph problems, it’s necessary to know how many neighbors a node has—that is, how many nodes are connected to a given node. For the example graph in **Figure 1**, node 5 has three neighbors. The number of neighbors a node has is also called the degree of the node. For a given node, this value could be computed on the fly when needed by counting the number of true/1 values in the node’s data row. A much faster approach is to count and store the number of neighbors for each node once in the graph constructor and then do an array lookup when needed. So, for the example graph after instantiation, array numberNeighbors would have nine cells with values [3,3,2,3,6,3,1,3,2], indicating node 0 has three neighbors, node 1 has three neighbors, node 2 has two neighbors and so on.

The graph class constructor is:

1. **public** MyGraph(**string** graphFile, **string** fileFormat)
2. {
3. **if** (fileFormat.ToUpper() == "DIMACS")
4. LoadDimacsFormatGraph(graphFile);
5. **else**
6. **throw** **new** Exception("Format " + fileFormat + " not supported");
7. }

The constructor accepts a text file that holds graph data and a string that indicates the specific format of the data file. Here, I immediately transfer control to a helper method LoadDimacsFormatGraph. This design allows the graph class to be easily extended to accommodate multiple data file formats. If you’re a fan of enumeration types, the file format parameter can be implemented using an enumeration.

The heart of the MyGraph class is the LoadDimacsFormatGraph method, which reads a source data file and stores the graph representation. There are many more or less standard graph file formats. The one I use here is called the DIMACS format. The acronym DIMACS stands for Discrete Mathematics and Theoretical Computer Science. DIMACS is a collaborative association headed by Rutgers University.

The example program shown in **Figure 2** uses a file named DimacsGraph.clq, which is listed in **Figure6**. Lines beginning with c are comment lines. There’s a single line beginning with p that has the string “edge,” followed by the number of nodes, followed by the number of edges. Lines beginning with e define edges. Notice that the DIMACS file format is blank-space delimited and 1-based, and that each edge is stored only once.

**Figure 6 DIMACS Format Data File**

c DimacsGraph.clq

c number nodes, edges: 9, 13

p edge 9 13

e 1 2

e 1 4

e 1 5

e 2 4

e 2 5

e 3 5

e 3 6

e 4 5

e 5 6

e 5 8

e 6 9

e 7 8

e 8 9

The load method begins:

1. **private** **void** LoadDimacsFormatGraph(**string** graphFile)
2. {
3. FileStream ifs = **new** FileStream(graphFile, FileMode.Open);
4. StreamReader sr = **new** StreamReader(ifs);
5. **string** line = "";
6. **string**[] tokens = **null**;
7. ...

When reading text files, I prefer using the FileStream and StreamReader classes, but you may want to use one of the many .NET alternatives. Next:

1. line = sr.ReadLine();
2. line = line.Trim();
3. **while** (line != **null** && line.StartsWith("p") == **false**) {
4. line = sr.ReadLine();
5. line = line.Trim();
6. }
7. ...

I perform a priming read and then advance to the p line in the data file. Because text files can easily acquire spurious whitespace characters over time, I use the Trim method to help avoid problems. Continuing:

1. tokens = line.Split(' ');
2. **int** numNodes = **int**.Parse(tokens[2]);
3. **int** numEdges = **int**.Parse(tokens[3]);
4. sr.Close(); ifs.Close();
5. **this**.data = **new** BitMatrix(numNodes);
6. ...

I use the String.Split method to parse the p line. At this point, tokens[0] holds the string literal “p”, tokens[1] holds “edge”, tokens[2] holds “9” and tokens[3] holds “13”. I use the int.Parse method (I could’ve used Convert.ToInt32) to convert the number of nodes and edges into int values that I store in local variables numNodes and numEdges. I could’ve stored these values into class fields this.numberNodes and this.numberEdges at this time. Now that I’ve determined the number of nodes and the number of edges, I close the data file and instantiate the BitMatrix data field.

Now I’m ready to read edge data from the data file:

1. ifs = **new** FileStream(graphFile, FileMode.Open);
2. sr = **new** StreamReader(ifs);
3. **while** ((line = sr.ReadLine()) != **null**) {
4. line = line.Trim();
5. **if** (line.StartsWith("e") == **true**) {
6. tokens = line.Split(' ');
7. **int** nodeA = **int**.Parse(tokens[1]) - 1;
8. **int** nodeB = **int**.Parse(tokens[2]) - 1;
9. data.SetValue(nodeA, nodeB, **true**);
10. data.SetValue(nodeB, nodeA, **true**);
11. }
12. }
13. sr.Close(); ifs.Close();
14. ...

I reopen the file and start reading from the beginning. Technically—because of the presence of the p line before any e lines—there’s no need to use two reads of a DIMACS format file. However, for other file formats that don’t explicitly store the number of edges, you may want to perform a double scan like the one used here. When the code encounters an e line such as “e 3 6”, I parse the e line, convert the two nodes to type int and subtract 1 to change the representation from 1-based to 0-based. I use the SetValue method to create symmetric entries in the BitMatrix. Note that because the BitMatrix is symmetric, I could've stored just the upper or lower triangular portion to reduce memory.

Next, I take care of the numberNeighbors array:

1. **this**.numberNeighbors = **new** **int**[numNodes];
2. **for** (**int** row = 0; row < numNodes; ++row) {
3. **int** count = 0;
4. **for** (**int** col = 0; col < numNodes; ++col) {
5. **if** (data.GetValue(row, col) == **true**) ++count;
6. }
7. numberNeighbors[row] = count;
8. }
9. ...

For each node, I walk across its corresponding row and count the number of true/1 values that gives the number of edges and therefore the number of neighbors the node has. The LoadDimacsFormatGraph method finishes with:

1. ...
2. **this**.numberNodes = numNodes;
3. **this**.numberEdges = numEdges;
4. **return**;
5. }

After transferring the number of nodes and the number of edges from local variables to class field variables, I use an explicit return for readability to exit the method.

The rest of the MyGraph class is easy. I expose the private numberNodes and numberEdges class fields as read-only values using the C# class Property mechanism:

1. **public** **int** NumberNodes  {
2. get { **return** **this**.numberNodes; }
3. }
5. **public** **int** NumberEdges  {
6. get { **return** **this**.numberEdges; }
7. }

I prefer using the explicit Property syntax, but you can use auto-implemented property syntax if you’re using .NET 3.0 or greater. I expose the number of neighbors a node has through a method:

1. **public** **int** NumberNeighbors(**int** node) {
2. **return** **this**.numberNeighbors[node];
3. }

When working with graphs, it’s difficult to know when to perform standard error checking and when to omit checking. Here, I don’t check to see if the node parameter is in the range 0 .. this.numberNodes-1, leaving me open to an array index out of range exception. I usually add error-checks during development, then after development I remove those checks that I feel can safely be omitted in order to improve performance. Because of my data structure design with a BitMatrix class, writing a method to determine if two nodes are adjacent is easy:

1. **public** **bool** AreAdjacent(**int** nodeA, **int** nodeB)
2. {
3. **if** (**this**.data.GetValue(nodeA, nodeB) == **true**)
4. **return** **true**;
5. **else**
6. **return** **false**;
7. }

Recall that the BitMatrix is symmetric, so I can check either GetValue(nodeA, nodeB) or GetValue(nodeB, nodeA). As I mentioned earlier, checking node adjacency dominates the runtime of many graph algorithms. When using a bit matrix, checking node adjacency is quick because the check is just an array lookup plus a little bit-manipulation overhead handled by the BitArray class.

I code a simple ToString method for the MyGraph class:

1. **public** **override** **string** ToString()
2. {
3. **string** s = "";
4. **for** (**int** i = 0; i < **this**.data.Dim; ++i) {
5. s += i + ": ";
6. **for** (**int** j = 0; j < **this**.data.Dim; ++j) {
7. **if** (**this**.data.GetValue(i, j) == **true**)
8. s += j + " ";
9. }
10. s += Environment.NewLine;
11. }
12. **return** s;
13. }

In the maximum clique scenario, performance isn’t a big issue, so for the ToString method, I use simple string concatenation rather than the more efficient StringBuilder class. Here, I use i to index into the BitMatrix rows and j to index into the columns. I terminate the string with an Environment.NewLine rather than “\n” to make the MyGraph class more portable.

## Validating the Graph

If you refer back to **Figure 2**, you’ll notice that I perform two important types of graph validation: validating the graph data file before the graph object is instantiated and validating the internal graph representation after instantiation.

A complete discussion of graph validation and testing would require an entire article, so I’ll just present an overview. You can obtain and view the complete validation code from code.msdn.microsoft.com/mag201110TestRun.

I perform data file validation using a static method ValidateGraphFile as shown in **Figure 5**. As with the MyGraph constructor, ValidateGraphFile immediately calls a helper method ValidateDimacsGraphFile to do the actual work. The file validation code iterates through the file to see if every line is in valid DIMACS form:

1. **if** (line.StartsWith("c") == **false** &&
2. line.StartsWith("p") == **false** &&
3. line.StartsWith("e") == **false**)
4. **throw** **new** Exception("Unknown line type: " + line);

The method also checks the format of non-comment lines by attempting to parse. For example, for the single p line:

1. **try** {
2. **if** (line.StartsWith("p")) {
3. tokens = line.Split(' ');
4. **int** numNodes = **int**.Parse(tokens[2]);
5. **int** numEdges = **int**.Parse(tokens[3]);
6. }
7. **catch** {
8. **throw** **new** Exception("Error parsing line = " + line);
9. }

The method uses similar logic to test e lines. This pattern holds in general when validating graph data files: check for valid lines and attempt to parse data lines.

Once instantiated, I validate the internal graph representation using a method ValidateGraph. It turns out that a complete check of the graph data structure is surprisingly complex, so in practice it’s common to check for just those errors that are most likely to occur. A common error in graph data files is a missing data line that creates an asymmetric BitMatrix data store. It can be checked with code like this:

1. **for** (**int** i = 0; i < **this**.data.Dim; ++i) {
2. **for** (**int** j = 0; j < **this**.data.Dim; ++j) {
3. **if** (**this**.data.GetValue(i, j) != **this**.data.GetValue(j, i))
4. **throw** **new** Exception("Not symmetric at " + i + " and " + j);
5. }
6. }

Other errors to check for include the presence of a true/1 on the bit matrix main diagonal, a bit matrix consisting of either all false/0 or true/1, and the sum of the values in the numberNeighbors array field not equaling the total number of true/1 values in the bit matrix.

## Stay Tuned for More Details

This article presented a graph data structure type that can be used for solving many graph-related problems including the maximum clique problem. The essential feature of the graph data structure is the use of a program-defined bit matrix that’s efficient in terms of memory usage and that allows quick node-adjacency lookups. The bit matrix field of the graph data structure is implemented using the .NET BitArray class, which takes care of all the low-level bit manipulation operations. In the next Test Run column, I’ll describe the maximum clique problem in more detail and show you a greedy algorithm solution that uses the graph structure described here.