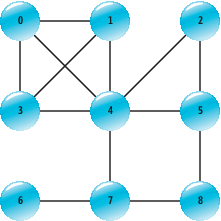
In this month’s column, I’ll present an advanced solution to the graph maximum clique problem. The solution uses what’s called a tabu algorithm, and I’ll discuss how to design and test these algorithms. The idea of the maximum clique problem is to find the largest group of nodes in a graph that are all connected to one another. Take a look at the simple graph in Figure 1. The graph has nine nodes and 13 edges. The graph is unweighted (there are no priorities associated with the edges) and undirected (all edges are bidirectional). Nodes 2, 4 and 5 form a clique of size three. The maximum clique is the node set 0, 1, 3 and 4, which forms a clique of size four.

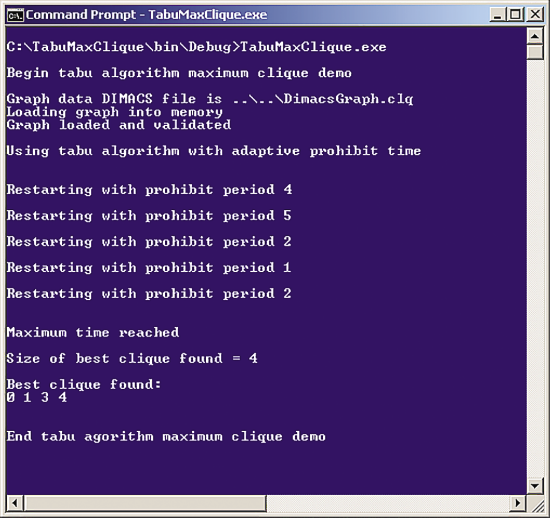
  
**Figure 1 Graph for a Tabu Maximum Clique Algorithm**

The maximum clique problem is interesting for several reasons. Although it’s not apparent from the simple graph in **Figure 1**, the maximum clique problem is one of the most challenging in computer science. It arises in many important practical problems, such as social network communication analysis, where nodes represent people and edges represent messages or relationships. And the problem uses techniques such as greedy algorithms and tabu algorithms, which are important advanced programming techniques. Having a solution to the maximum clique problem in your personal library can be a useful practical tool, and understanding the algorithm used can add new techniques to your skill set.

This is the third column in a series that uses the maximum clique problem to illustrate advanced coding and testing techniques, but this column can be read without direct reference to the previous two. In my October column, “Graph Structures and Maximum Clique” ([msdn.microsoft.com/magazine/hh456397](http://msdn.microsoft.com/magazine/hh456397)), I described how to code an efficient data structure to hold a graph data structure in memory. In my November column, “Greedy Algorithms and Maximum Clique” ([msdn.microsoft.com/magazine/hh547104](http://msdn.microsoft.com/magazine/hh547104)), I described how a relatively simple algorithm can be used to find a solution to the difficult maximum clique problem.

In informal terms, a greedy algorithm is an algorithm that starts with a simple, incomplete solution to a difficult problem and then iteratively looks for the best way to improve the solution. The process is repeated until some stopping condition is reached. How­ever, greedy algorithms typically have a weakness: They’ll repeatedly generate the same solution over and over. Tabu algorithms are designed to deal with this weakness. The word tabu, sometimes spelled taboo, means forbidden. In simple terms, tabu algorithms maintain a list of forbidden data. The processing part of the algorithm isn’t permitted to use tabu data until some prohibit time has passed. Simple forms of tabu algorithms use a fixed prohibit time. Advanced tabu algorithms adapt the prohibit time dynamically while the algorithm executes.

**Figure 2** illustrates the important ideas of a tabu algorithm applied to the maximum clique problem and shows you where I’m headed in this column.

  
**Figure 2 Tabu Maximum Clique Demo Run**

I have a console application that begins by loading into memory the graph corresponding to the one shown in **Figure 1**. The file uses a standard graph file format called DIMACS. Designing and coding an efficient graph data structure is a significant problem in itself. I explained the graph data structure code in my October column.

The algorithm begins by selecting a single node at random and initializing a clique with that node. The algorithm then iteratively tries to find and add the best node that will increase the size of the clique. I’ll explain what best node means later. If the algorithm can’t find a node to add, it attempts to find the best node to drop from the clique. Behind the scenes, the algorithm remembers the last time each node was moved into the current solution clique or moved out from the clique. The algorithm uses this information to prohibit adding or dropping recently used nodes for a certain amount of time. This is the tabu part of the algorithm.

The algorithm restarts itself every so often when no progress has been made in finding a better (larger) clique. The algorithm silently stores the solutions (cliques) it has previously seen. The algorithm uses that solution history information to dynamically adapt the prohibit time. If the algorithm keeps encountering the same solutions, it increases the prohibit time to discourage recently used nodes from being used. If the algorithm doesn’t see the same solutions, it decreases the prohibit time so there’s a larger pool of nodes from which to choose. This is the adaptive part of the tabu algorithm.

In most situations where a greedy algorithm is used, the optimal solution is not known, so one or more stopping conditions must be specified. When the algorithm hits a stopping condition, the best solution found is displayed.

In the sections that follow, I’ll walk you through the code that produced the screenshot in **Figure 2**. The complete source code is available at [code.msdn.microsoft.com/mag201112TestRun](http://code.msdn.microsoft.com/mag201112TestRun). This column assumes you have intermediate-level programming skills with a C-family language or the Visual Basic .NET language. I use C#, but I’ve written the code so that you’ll be able to refactor to another language such as F# or Python without too much difficulty if you want.

## Overall Program Structure

The overall structure of the program shown in **Figure 2** is presented in **Figure 3**. I decided to place all code in a single console application using static methods for simplicity, but you might want to modularize parts of the code into class libraries and use an object-oriented approach. The program is less complicated than you might suspect in looking at **Figure 3** because most of the methods are short helper methods.

**Figure 3 Overall Program Structure**

1. **using** System;
2. **using** System.Collections.Generic; // List<int>
3. **using** System.Text; // StringBuilder
4. **using** System.IO; // FileStream
5. **using** System.Collections; // Hashtable
6. **namespace** TabuMaxClique
7. {
8. **class** TabuMaxCliqueProgram
9. {
10. **static** Random random = **null**;
11. **static** **void** Main(**string**[] args) { ... }
12. **static** List<**int**> FindMaxClique(MyGraph graph, **int** maxTime,
13. **int** targetCliqueSize) { ...  }
14. **static** List<**int**> MakePossibleAdd(MyGraph graph,
15. List<**int**> clique) { ... }
16. **static** **bool** FormsALargerClique(MyGraph graph,
17. List<**int**> clique, **int** node) { ... }
18. **static** **int** GetNodeToAdd(MyGraph graph,
19. List<**int**> allowedAdd, List<**int**> possibleAdd) { ... }
20. **static** **int** GetNodeToDrop(MyGraph graph, List<**int**> clique,
21. List<**int**> oneMissing) { ... }
22. **static** List<**int**> MakeOneMissing(MyGraph graph,
23. List<**int**> clique) { ... }
24. **static** List<**int**> SelectAllowedNodes(List<**int**> listOfNodes,
25. **int** time, **int** prohibitPeriod, **int**[] lastMoved) { ... }
26. **static** **int** UpdateProhibitPeriod(MyGraph graph, List<**int**> clique,
27. **int** bestSize, Hashtable history, **int** time, **int** prohibitPeriod,
28. **ref** **int** timeProhibitChanged) { ... }
29. **static** **string** ListAsString(List<**int**> list) { ... }
30. **private** **class** CliqueInfo
31. {
32. // ...
33. }
34. }
35. **public** **class** MyGraph
36. {
37. // ...
38. **private** **class** BitMatrix
39. {
40. // ...
41. }
42. }
43. } // ns

The program has two high-level classes, and each of those classes has a helper subclass. Class TabuMaxCliqueProgram contains the Main method and all algorithm logic, along with subclass CliqueInfo, which is used to maintain a history of previously seen clique solutions. Class MyGraph encapsulates an efficient graph representation and contains subclass BitMatrix, which is used to store node adjacency information in a condensed form. A class scope Random object named random is used to initialize the clique to a random node and to break ties when there are multiple best nodes to add or drop.

With some WriteLine statements and try-catch code removed, the Main method is:

1. Console.WriteLine("\nBegin tabu algorithm maximum clique demo\n");
2. **string** graphFile = "..\\..\\DimacsGraph.clq";
3. MyGraph graph = **new** MyGraph(graphFile, "DIMACS");
4. **int** maxTime = 50;
5. **int** targetCliqueSize = graph.NumberNodes;
6. List<**int**> maxClique = FindMaxClique(graph, maxTime, targetCliqueSize);
7. Console.WriteLine("\nSize of best clique found = " + maxClique.Count);
8. Console.WriteLine(ListAsString(maxClique));

The graph is represented as a program-defined MyGraph object. The graph is loaded from an external text file that uses the DIMACS file format. The key method of MyGraph is AreAdjacent, which returns true if two nodes are connected. I set maxTime to 50 iterations to establish an absolute stopping condition for the greedy algorithm. This is artificially small. In a real maximum clique problem, the max time is typically in the range of 1,000 to 100,000. I set targetClique size to establish a second stopping condition; if a clique is found that has the same number of nodes as there are in the graph, the maximum clique must have been found. Most of the work is done by the FindMaxClique method, which uses a greedy, adaptive tabu algorithm to search for the largest possible clique. The helper method ListAsString simply creates a string representation of a List<int> object.

## The FindMaxClique Method

The FindMaxClique method calls several helper methods, which I’ll describe shortly. In high-level pseudocode, the FindMaxClique algorithm is given in **Figure 4**. The pseudocode leaves out several important details for clarity but presents the main points of the algorithm.

**Figure 4 Adaptive Tabu Maximum Clique Algorithm**

initialize clique with a single node

get list of candidate nodes to add

loop until stop condition met

  if there are nodes which can be added

    get list of allowed nodes

    if there are allowed nodes to add

      find best node to add and add to clique

      mark added node as recently used

      record current clique

 else

   get list of allowed nodes to drop

   if there are allowed nodes which can be dropped

     find best node to drop and drop from clique

     mark dropped node as recently used

     record current clique

 else

   select a random node to drop and drop from clique

   mark dropped node as recently used

   record current clique

 if lack of progress

   restart algorithm

  update list of candidate nodes

  update prohibit time

end loop

return largest clique found

The FindMaxClique method definition begins:

static List<int> FindMaxClique(MyGraph graph, int maxTime,

  int targetCliqueSize)

{

  List<int> clique = new List<int>();

  random = new Random(1);

  int time = 0;

  int timeBestClique = 0;

  int timeRestart = 0;

  int nodeToAdd = -1;

  int nodeToDrop = -1;

...

The local clique object is the current solution clique. The random object is instantiated with an arbitrary seed value of 1. The time variable is the counter for the algorithm’s main processing loop. The timeBestClique and timeRestart variables are used to determine if there has been a lack of progress so the algorithm can restart itself. Next:

1. **int** prohibitPeriod = 1;
2. **int** timeProhibitChanged = 0;
3. **int**[] lastMoved = **new** **int**[graph.NumberNodes];
4. **for** (**int** i = 0; i < lastMoved.Length; ++i) {
5. lastMoved[i] = **int**.MinValue;
6. }
7. Hashtable history = **new** Hashtable();
8. ...

The prohibit period is initialized to 1. This means that—initially, at least—after a node has been used by the algorithm, it can’t be used for one time iteration. If I had used a value of 0, the effect would have been to have no prohibit time at all. Most tabu algorithms use a fixed value for the prohibit time, but the problem with that approach is that research has shown the best value for a fixed prohibit time varies from problem to problem. The adaptive approach I present updates the prohibit time while the algorithm runs, based on the history of previous solutions. I call this technique adaptive tabu, but some research papers call the technique reactive or dynamic.

The lastMoved array is used to determine if a node is allowed or prohibited at any given time. The index of the array represents a node, and the corresponding array value is the time the node was last moved (added or dropped). By using the lastMoved array, I eliminate the need to explicitly create a list of allowed nodes (or, equivalently, prohibited nodes). I initialize all cells of lastMoved to int.MinValue so I can later determine if a node has never been used.

The history Hashtable object holds a collection of previously seen solution cliques. Each item in the hash table is a CliqueInfo object, which I’ll describe in detail later. I could’ve used a generic Dictionary object instead of the non-generic Hashtable object. FindMaxClique continues:

1. **int** randomNode = random.Next(0, graph.NumberNodes);
2. clique.Add(randomNode);
3. List<**int**> bestClique = **new** List<**int**>();
4. bestClique.AddRange(clique);
5. **int** bestSize = bestClique.Count;
6. timeBestClique = time;
7. ...

I initialize the current solution clique to a random node from the graph. I instantiate a bestClique list to keep track of the best (largest) clique solution found by the algorithm. Next comes:

1. List<**int**> possibleAdd = MakePossibleAdd(graph, clique);
2. List<**int**> oneMissing = MakeOneMissing(graph, clique);
3. ...

The MakePossibleAdd method constructs a list of all nodes that can be added to the current clique to increase the size of the clique by 1. In other words, the method creates a list of nodes that aren’t in the clique but are connected to every node in the clique. This possibleAdd list could have a count of 0 or could contain prohibited nodes. MakePossibleAdd calls helper method FormsALargerClique.

The MakeOneMissing method constructs a list of all nodes that are connected to all but exactly one of the nodes in the current clique. It’s not obvious, but it turns out that maintaining such a list creates a clever way to determine which node in the current clique is the best node to drop, as I’ll explain later. The main processing loop begins:

1. **while** (time < maxTime && bestSize < targetCliqueSize)
2. {
3. ++time;
4. **bool** cliqueChanged = **false**;
5. ...

The main processing loop will terminate when the maximum number of iterations is reached or if a best solution with the target clique size is found. I use the cliqueChanged variable to control the branching logic between adding and dropping nodes, as you’ll see. The logic to add an allowed node is shown in**Figure 5**.

**Figure 5 The Logic to Add an Allowed Node**

1. **if** (possibleAdd.Count > 0) {
2. List<int> allowedAdd = SelectAllowedNodes(possibleAdd, time,
3. prohibitPeriod, lastMoved);
4. **if** (allowedAdd.Count > 0) {
5. nodeToAdd = GetNodeToAdd(graph, allowedAdd, possibleAdd);
6. clique.Add(nodeToAdd);
7. lastMoved[nodeToAdd] = time;
8. clique.Sort();  cliqueChanged = **true**;
9. **if** (clique.Count > bestSize) {
10. bestSize = clique.Count;
11. bestClique.Clear();
12. bestClique.AddRange(clique);
13. timeBestClique = time;
14. }
15. }
16. }

The algorithm checks to see if there are any nodes that will increase the size of the current clique. If so, method SelectAllowedNodes creates a list of nodes that are allowed—that is, not tabu—from the list of nodes that can be added.

The key line in SelectAllowedNodes is:

1. **if** (time > lastMoved[currNode] + prohibitPeriod)
2. result.Add(currNode); // Allowed

The currNode is one of the nodes in the possibleAdd list. The logic checks to see if enough time has elaspsed since the node was last used so that the node is past the prohibit period. If so, the node is added to the list of allowedAdd nodes.

Now, if there are any nodes that are allowed and will increase the size of the clique, method GetNodeToAdd determines the best node to add to the clique. The best node is the node from the allowedAdd list that’s most connected to nodes in the possibleAdd list. The idea is that you want to add a node that will give you the most chances of finding a node to add in the next iteration of the algorithm. There could be multiple nodes in the allowedAdd list that are tied as most connected to the nodes in possibleAdd. If so, one of the tied nodes is selected at random. After adding the best node, the current clique solution is sorted for reasons that I’ll explain later. The code to drop a node is:

1. **if** (cliqueChanged == **false**) {
2. **if** (clique.Count > 0) {
3. List<**int**> allowedInClique = SelectAllowedNodes(clique, time,
4. prohibitPeriod, lastMoved);
5. **if** (allowedInClique.Count > 0) {
6. nodeToDrop = GetNodeToDrop(graph, clique, oneMissing);
7. clique.Remove(nodeToDrop);
8. lastMoved[nodeToDrop] = time;
9. clique.Sort();  cliqueChanged = **true**;
10. }
11. }
12. }
13. ...

If the algorithm can’t add a node, it attempts to drop an allowed node in the hope that backtracking will yield a solution that will allow a new node to be added in the next iteration. The list of nodes in the current clique is filtered by the SelectAllowedNodes method to get nodes that are not tabu. The GetNodeToDrop picks the best of these nodes to drop using the list of oneMissing nodes.

The idea is to drop the allowed node in the current clique, which will result in the largest increase in the list of possibleAdd nodes. One way to do this is to test each node in the allowed list by actually removing it from the current clique and then computing the size of the resulting possibleAdd list. But there’s a much more efficient approach that uses a list of nodes that are connected to all but exactly one of the nodes in the current clique. Using the oneMissing list of nodes, the list can be used as follows. Scan through each node in the allowed clique nodes and count how many nodes in the oneMissing list are notconnected to the clique node. The node in the allowed clique list that’s least connected to the nodes in the oneMissing list is the best node to drop. After dropping this least-connected node, all the nodes in the oneMissing list that weren’t connected to the dropped node will now be fully connected to the remaining nodes in the clique and therefore become new possibleAdd nodes.

At this point in the logic, it might not have been possible to add an allowed node or drop an allowed node. To unstick itself, the algorithm drops a random node from the current clique:

1. **if** (cliqueChanged == **false**) {
2. **if** (clique.Count > 0) {
3. nodeToDrop = clique[random.Next(0, clique.Count)];
4. clique.Remove(nodeToDrop);
5. lastMoved[nodeToDrop] = time;
6. clique.Sort();
7. cliqueChanged = **true**;
8. }
9. }
10. ...

Next, the algorithm checks to see if there has been a lack of progress and, if so, restarts itself:

1. **int** restart = 100 \* bestSize;
2. **if** (time - timeBestClique > restart && time - timeRestart > restart) {
3. timeRestart = time;
4. prohibitPeriod = 1;
5. timeProhibitChanged = time;
6. history.Clear();
7. ...

The restart variable is the number of iterations to allow where there’s no improvement or since the last restart. Here I use a value of 100 times the size of the current best solution. The value to use for the restart interval isn’t well understood and you might want to try alternatives. (I used a dummy value of 2 to produce more restarts for the screenshot in **Figure 2**). The value I use has worked well for me in practice, however. If there is a restart, the algorithm resets the prohibit time and clears the history hash table holding solution cliques that have been seen. Note that the algorithm hasn’t yet updated the history hash table. The restart code doesn’t, however, reset the lastVisited array, which stores information about when nodes were last added to or dropped from the solution clique. Next comes:

1. **int** seedNode = -1;
2. List<**int**> temp = **new** List<**int**>();
3. **for** (**int** i = 0; i < lastMoved.Length; ++i) {
4. **if** (lastMoved[i] == **int**.MinValue) temp.Add(i);
5. }
6. **if** (temp.Count > 0)
7. seedNode = temp[random.Next(0, temp.Count)];
8. **else**
9. seedNode = random.Next(0, graph.NumberNodes);
10. ...

The algorithm attempts to reseed the solution clique with a node that has never been used before. If there are several unused nodes, one is selected at random. If there are no unused nodes, a random node is selected. Instead of using a random node, an unexplored alternative is to select the node that was least recently moved. The restart code finishes with the following:

1. ...
2. clique.Clear();
3. clique.Add(seedNode);
4. }

The main processing loop and FindMaxClique wrap up like so:

1. ...
2. possibleAdd = MakePossibleAdd(graph, clique);
3. oneMissing = MakeOneMissing(graph, clique);
4. prohibitPeriod = UpdateProhibitPeriod(graph, clique, bestSize,
5. history, time, prohibitPeriod, **ref** timeProhibitChanged);
6. } // Main processing loop
7. **return** bestClique;
8. }

The possibleAdd and oneMissing lists are regenerated. An alternative is to maintain auxiliary data structures and update these two lists instead of recreating them from scratch. The UpdateProhibitPeriod method is the key component of the adaptive part of the tabu algorithm and I’ll describe it shortly.

## Remembering Previous Solutions

Method UpdateProhibitPeriod uses a hash table of previously seen solutions to dynamically increase or decrease the prohibit time. Recall that a solution is a clique that’s a List<int> of nodes that are all connected to one another. But I also need to store the time when a solution was last seen. Therefore, I encapsulate a clique solution and the last time it was seen in a CliqueInfo class as listed in **Figure 6**.

**Figure 6 The CliqueInfo Class**

1. **private** **class** CliqueInfo
2. {
3. **private** List<**int**> clique;
4. **private** **int** lastSeen;
5. **public** CliqueInfo(List<**int**> clique, **int** lastSeen)
6. {
7. **this**.clique = **new** List<**int**>();
8. **this**.clique.AddRange(clique);
9. **this**.lastSeen = lastSeen;
10. }
11. **public** **int** LastSeen
12. {
13. get { **return** **this**.lastSeen; }
14. set { **this**.lastSeen = value; }
15. }
16. **public** **override** **int** GetHashCode()
17. {
18. StringBuilder sb = **new** StringBuilder();
19. **for** (**int** i = 0; i < clique.Count; ++i) {
20. sb.Append(clique[i]);
21. sb.Append(" ");
22. }
23. **string** s = sb.ToString();
24. **return** s.GetHashCode();
25. }
26. **public** **override** **string** ToString()
27. {
28. **string** s = "";
29. **for** (**int** i = 0; i < clique.Count; ++i)
30. s += clique[i] + " ";
31. **return** s;
32. }
33. }

Because a CliqueInfo item will be added to the solution history Hashtable object, I need a custom GetHashCode hash function. Writing custom hash functions is surprisingly tricky, and a thorough discussion of all the issues involved would require an entire column. In this situation, I use the simplest—but not the most efficient—approach. I create a string representation of the nodes in the clique field, separated by spaces, and then use the built-in String.GetHashCode. For example, if a clique contained nodes {3 5 8}, I generate "3 5 8 " (with a trailing space) and then generate a hash code from that string. Recall that cliques are always maintained in a sorted order, so it wouldn’t be possible to have one clique {3 5 8} and another clique {8 3 5}. I place spaces between the nodes so that clique {3 5 8} is distinguished from clique {3 58}.

## Updating the Prohibit Period

Method UpdateProhibitPeriod adaptively increases or decreases the prohibit time based on previously seen clique solutions. The method begins:

1. **static** **int** UpdateProhibitPeriod(MyGraph graph, List<**int**> clique,
2. **int** bestSize, Hashtable history, **int** time, **int** prohibitPeriod,
3. **ref** **int** timeProhibitChanged)
4. {
5. **int** result = prohibitPeriod;
6. CliqueInfo cliqueInfo = **new** CliqueInfo(clique, time);
7. . . .

The method will return a prohibit time that could possibly be the same as the current prohibit time. A CliqueInfo object holding the current clique and current time are instantiated, as shown here:

1. **if** (history.Contains(cliqueInfo.GetHashCode())) {
2. CliqueInfo ci = (CliqueInfo)history[cliqueInfo.GetHashCode
3. **int** intervalSinceLastVisit = time - ci.LastSeen;
4. ci.LastSeen = time;
5. **if** (intervalSinceLastVisit < 2 \* graph.NumberNodes - 1) {
6. timeProhibitChanged = time;
7. **if** (prohibitPeriod + 1 < 2 \* bestSize) **return** prohibitPeriod + 1;
8. **else** **return** 2 \* bestSize;
9. }
10. }
11. **else** history.Add(cliqueInfo.GetHashCode(), cliqueInfo);
12. ...

The code checks to see if the current clique solution, in the form of a CliqueInfo object, has been seen before—that is, is the clique in the history hash table? If the current clique has been seen before, the algorithm computes how long it has been since the clique was seen. If this interval is short enough, the prohibit time is incremented (subject to an upper limit). The idea is that because it hasn’t been very long since the current clique was seen, the algorithm is generating duplicate solutions. If the prohibit time is increased, there will be more tabu nodes and therefore fewer allowed nodes, reducing the chances of generating duplicate clique solutions. If the current clique solution hasn’t been seen before, it’s added to the history hash table.

The “short enough” interval is twice the number of nodes in the graph, less one. This is used to determine when to increment the prohibit time. The best value to use here is another open question in maximum clique research. The “upper limit” for the prohibit time is twice the size of the current best known solution. The best upper-limit value is, as you can probably guess by now, another open research question.

At this point, either the current clique hasn’t been seen before, or the clique has been seen before but the interval required to increment the prohibit time wasn’t small enough. So the algorithm now checks to see if the prohibit period can be decremented, which will decrease the number of tabu nodes and increase the number of allowed nodes, which in turn gives the algorithm more nodes to add or drop:

1. ...
2. **if** (time - timeProhibitChanged > 10 \* bestSize) {
3. timeProhibitChanged = time;
4. **if** (prohibitPeriod - 1 > 1)
5. **return** prohibitPeriod - 1;
6. **else**
7. **return** 1;
8. }
9. **else** {
10. **return** result; // no change
11. }
12. } // UpdateProhibitTime

Rather than explicitly checking to see if it has been a “relatively long” time since the current clique was seen, the algorithm indirectly checks how long it has been since the current clique was seen by examining the time when the prohibit period was last changed. Once again, the best value for “relatively long” isn’t clearly understood. I use a value 10 times the size of the current best solution. Notice that the prohibit time can’t drop below 1.

## More Research

Research results on the maximum clique problem suggest that a greedy approach with an adaptive tabu feature gives the best overall results when both performance and solution quality are taken into account. DIMACS is a research organization that created a set of difficult graph clique benchmark problems. I ran the code presented here against one particularly difficult DIMACS problem (C2000.9) and the code quickly (in less than two seconds) found a clique with size 76 that’s within 1.5 percent of the size of the best-known solution of 77.

At several points in this column, I’ve mentioned research results on the maximum clique problem. If you’re interested in this research, I recommend you search the Web for academic papers written by the following: R. Battiti et al., W. Pullan et al. and K. Katayama et al. Several papers by these three authors and their colleagues were my primary references.

A promising unexplored area for the maximum clique algorithm presented here is to incorporate some form of what is called plateau search. Recall that the maximum clique algorithm first attempts to add a non-tabu node to the current clique solution. Then, if that isn’t possible, the algorithm drops a node from the current clique solution. The idea of plateau search is to find one node in the current clique solution that can be swapped with a node not in the clique. Although this doesn’t increase the size of the current clique, it doesn’t decrease the size of the clique, either.