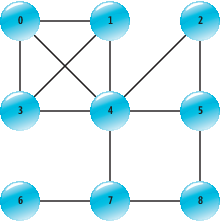
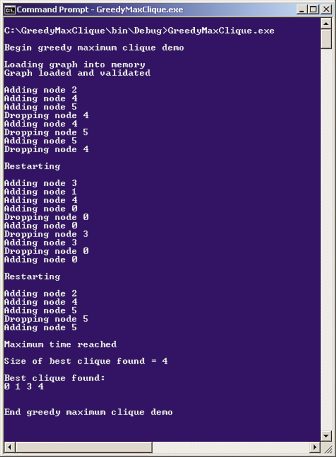
In this month’s column, I’ll present a solution to the graph maximum clique problem. The solution uses what’s called a greedy algorithm, and I’ll discuss how to design and test these algorithms. The idea of the maximum clique problem is to find the largest group of nodes in a graph that are all connected 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 form a clique of size four.

  
**Figure 1 Graph for a Greedy 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 applications such as social network communication analysis, where nodes represent people and edges represent a message or relationship. And the maximum clique problem lends itself well to solution by a greedy algorithm, which is a fundamental technique in computer science.

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. **Figure 2** illustrates the important ideas of the maximum clique problem and shows you where I’m headed in this article.

  
**Figure 2 Greedy Maximum Clique Demo Run**

I have a console application, which begins by loading into memory the graph that corresponds to the one shown in **Figure 1**. Designing and coding an efficient graph data structure is a significant problem in itself. I explained the graph data structure code in my last column (msdn.microsoft.com/magazine/hh456397). The algorithm first selects a single node at random, in this case node 2, and initializes a clique with that node. Next, the algorithm looks for the best node to add to the clique. If you refer to **Figure 1**, you’ll see that there are only two nodes, 4 and 5, that will form a larger clique. I’ll explain what the best node means shortly.

The algorithm selects and adds node 4 to the clique so the clique is now {2, 4}. At this point, there’s only one node in the graph, 5, that will increase the size of the clique, so the algorithm selects node 5 and adds it to the clique. There are now no nodes available that will increase the size of the clique. The algorithm drops the best node from the clique in hopes that a new, different node can be added that will enable progress. Again, I explain what “best” means shortly. The algorithm drops node 4 from the clique. As you can see by looking at the graph, there’s no way to make progress. This situation is common with greedy algorithms, so there must be a way to get out of dead-end solutions.

The algorithm is tracking how long it has been since progress has been made—that is, a clique with a larger size has been found. After a certain amount of time with no progress, the algorithm restarts itself. The clique is cleared. This time the algorithm randomly picks node 3 as the initial node. Using the same iterative process to find the best node to add or drop, the algorithm eventually discovers the maximum clique of {0, 1, 3, 4}. In most problems where greedy algorithms are used, the optimal solution isn’t known, so the algorithm continues until some stopping condition is met. In this case, the algorithm is configured to stop after 20 iterations. At this point, the best clique 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/mag201110TestRun](http://code.msdn.microsoft.com/mag201110TestRun) (this column uses the same code from last month). This article assumes you have intermediate-level programming skill with a C-family language or the VB.NET language. I use C#, but I’ve written the code so that you’ll be able to refactor it to another language without too much difficulty if you wish.

## Overall Program Structure

The overall structure of the program shown in **Figure 2** is presented in **Figure 3**. The program has dependencies on namespaces System, System.Collections.Generic (a clique is represented as type List<int>), System.IO (for loading the source graph from a text file) and System.Collections (the program-defined MyGraph class uses class BitArray). I rename the main class from the Visual Studio-template-generated Program to GreedyMaxCliqueProgram for clarity. I use a class-scope Random object named “random” to initialize and reset a clique to a random node and to break ties when there’s more than one best node to add or drop.

**Figure 3 Overall Program Structure**

The graph is represented as a program-defined MyGraph object. The graph is loaded from an external text file, which uses a standard file format named DIMACS. The key method of MyGraph is AreAdjacent, which returns true if two nodes are connected. I set maxTime to 20 to set an absolute stopping condition for the greedy algorithm. I set targetCliqueSize 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.

## The FindMaxClique Method

All of the work is done by the method FindMaxClique, which uses a greedy algorithm to search for the largest clique possible. FindMaxClique calls several helper methods, which I’ll describe in detail. I structure the program using static methods, but you may want to refactor the code I present here to a fully object-oriented approach. The FindMaxClique method iterates until one of the two stopping conditions is met and then returns the best clique found. The program definition includes an embedded MyGraph class, which was the subject of last month’s article.

In high-level pseudo-code, the FindMaxClique algorithm is:

The FindMaxClique method definition begins:

The local clique object is the current clique. I instantiate the Random object using an arbitrary seed value of 1. The time variable is a loop counter variable; because it’s discrete, a good alternative name would be “tick.” I track the time when the current best clique was found and the time of the last restart to control when restarts will happen. I assign dummy values of -1 to variables for the add-node and drop-node:

The Random.Next(x,y) method returns a value greater than or equal to x and strictly less than y. By setting x = 0 and y = NumberNodes, I get a random node from 0 to 8 inclusive for the example graph:

The bestClique list is used to track a copy of the largest clique found during the algorithm’s search. I use the handy AddRange method to copy the items in the current clique into bestClique. The bestSize variable is used for convenience. The timeBestClique is used to determine if a restart of the algorithm will occur:

The MakePossibleAdd helper method examines the current clique and constructs a list of all nodes that can be added to the clique to increase the clique’s size by one. This list is the source of candidates for the best node to add to the clique. The MakeOneMissing helper method is a bit tricky. The method constructs a list of all nodes that are connected to all but one node in the current clique. As you’ll see, this oneMissing list is used to determine the best node to drop from a clique. Now I begin the main processing loop:

As I explained earlier, greedy algorithms typically need one or more stopping conditions. Here the loop terminates if the maximum number of iterations is exceeded or the size of the current clique meets a target size. The cliqueChanged variable is used to control the branching logic between adding nodes and dropping nodes. I could’ve omitted this variable and used an if..else control structure instead of separateif..then statements, but in this case, the use of a branching control variable leads to code that’s easier to read and modify, in my opinion:

I check to make sure there’s at least one node that can be added to the clique and then call helper method GetNodeToAdd. This method scans the list of nodes in the possibleAdd list and returns the best node to add (the oft-promised explanation of “best” will be presented when I describe GetNodeToAdd). This node is added to the current clique. At this point I sort the clique, because later in the algorithm I need to search the clique, and if the clique is sorted, I can use a quick binary search instead of a linear search. The new clique is checked to see if it’s better (larger) than the current best clique.

Next comes:

If the size of the current clique can’t be increased, the algorithm attempts to drop a node from the clique. The helper method GetNodeToDrop selects the best node to drop from the clique:

At this point, the algorithm checks to see if there’s been a lack of progress. The restart variable determines how long to wait before restarting. Here I use a value of twice the size of the current best clique. In research on the maximum clique value, the optimal value for restart is still an open question. The value I use here is based on experiments I’ve performed with several benchmark graph problems. A restart occurs if there’s been lack of progress in finding a new best solution or if it’s been a relatively long time since the last restart. If a restart is made, the algorithm empties the current clique and then selects a random node from all nodes in the graph. Notice that this is a crude approach; if you refer back to the demo run in  **Figure 2**, you’ll see that the last restart picked an initial node that had already been used as the first seed node:

The FindMaxClique method computes from scratch the possibleAdd and oneMissing lists based on the new clique. When the main processing loop terminates, the method returns the best clique found.

## Adding the Best Node

There are two steps to determining the best node to add to the current clique. First, a list of all nodes that will increase the size of the current clique is needed. Second, some way to determine which of these nodes is the best node is needed:

The MakePossibleAdd method scans through each node in the graph. If the node being examined is connected to all nodes in the current clique as determined by helper FormsALargerClique, then the node being examined is a possible “add” node and joins the result list. Notice the result could be an empty list but will never be null:

The FormsALargerClique method compares one node against all nodes in the current clique. If the candidate node isn’t adjacent to even one of the nodes in the clique, then the candidate node can’t be added to the current clique. Notice that because AreAdjacent returns false when a node is compared against itself, nodes in the current clique won’t be added to the list of possibleAdd nodes.

The main idea behind determining the best node to add is to select the one node from the list of possibleAdd nodes that’s most connected to all the other nodes in the possibleAdd set. A node that’s highly connected yields the largest possible new list of nodes to add on the next iteration of the algorithm.

Next comes:

The GetNodeToAdd method assumes that the possibleAdd list has at least one node. If there’s exactly one node, then that must be the best node:

There could be several nodes in the possibleAdd list that are tied with others as being the most connected to the other nodes in the list. For example, suppose the graph under analysis is the graph shown in **Figure 1** and the current clique has just node 0. The possibleAdd nodes are {1, 3, 4}. Node 1 is connected to two of the nodes in possibleAdd—and, in fact, so are nodes 3 and 4. So a preliminary scan is made to determine the maximum number of connections available:

Once the maximum number of connections is known, the algorithm rescans the possibleAdd list and adds each of the nodes in possibleAdd that have the maximum number of connections to a list of candidate nodes. Note that the double-scan could be eliminated by using auxiliary data stores to keep track of the number of connections each possibleAdd node has:

At this point, the candidates list has one or more best nodes to add to the clique. Notice that candidates must have a count of at least one because the possibleAdd list is assumed to have a count of at least one. The algorithm randomly selects one of the candidate nodes and returns it. I could’ve put in a check to see if the size of the candidates list is exactly one, and if so, return the single node in candidates.

## Dropping the Best Node

Determining the best node to drop from the current clique is a bit subtle. The idea is to drop the one node in the current clique that will result in the largest increase in the list of possibleAdd nodes.

One way to do this is to test each node in the current clique 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. If there is such a oneMissing list of nodes, the list can be used as follows: Scan through each node in the current clique and count how many nodes in the oneMissing list are not connected to the clique node. The node in the current clique 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, therefore becoming new possibleAdd nodes.

The MakeOneMissing method is presented in **Figure 4**. The method scans through each node in the graph. The idea is to count how many nodes in the clique are connected to the current node being examined. If the total count of connected nodes is exactly one less than the size of the current clique, then the node being examined is a oneMissing node. The method short-circuits if the current node being examined has fewer than the necessary number of neighbors. The method must filter out nodes that are already in the clique because they’re connected to all but one node in their own clique (that is, themselves). Because the current clique is sorted (after each add or drop), a binary search can be used instead of a slower linear search (see **Figure 4**).

**Figure 4 Make List of oneMissing Nodes**

The GetNodeToDrop method begins:

The method assumes that the current clique has at least one node. If there’s only one node in the current clique, then it’s the only node that can be dropped. The method determines the largest number of nodes in the oneMissing list that aren’t connected to any node in the current clique because there could be more than one node in the clique that’s maximally disconnected to the oneMissing list:

Once the maximum number of disconnections is known, the method rescans the clique to find those nodes that are most disconnected and adds each to a list of candidates. As with the GetNodeToAdd method, GetNodeToDrop could avoid a rescan by maintaining auxiliary data stores:

Method GetNodeToDrop finishes by returning a randomly selected node from the list of best nodes to drop:

## Final Thoughts

How effective are greedy algorithms? In spite of their relative simplicity, greedy algorithms have been shown to be quite effective in many problem scenarios. However, effectiveness can be defined using several criteria, including speed and quality of solution. I ran this algorithm against several extremely difficult benchmark graph problems maintained by the DIMACS research organization and the algorithm was quite effective, running quickly and producing maximum cliques that were generally within 5 percent of the best known solutions.

Testing greedy algorithms can be tricky. In addition to all the usual testing techniques such as unit testing, boundary-condition testing and so on—and because greedy algorithms have solutions that grow over time—an effective testing strategy is to write helper methods that iteratively check the state of the data structures used by the algorithm. For example, while testing the maximum clique algorithm presented here, I wrote methods that verify there were no duplicate nodes in the current clique, verify no node in the current clique is in the current possibleAdd or current oneMissing lists and so on.

The greedy maximum clique algorithm I’ve presented here can be modified to produce better results in several ways. One technique, common to most greedy algorithms, is to add a so-called tabu feature. Tabu algorithms maintain a list of recently used data and possibly a list of recently seen solutions. Data in the tabu list isn’t used to construct new solutions. Additionally, greedy algorithms can often employ a strategy called plateau search. When a greedy algorithm is unable to improve its current solution, a plateau search produces a new solution without going backward, such as dropping a node in the maximum clique problem. I’ll present these interesting and useful tabu and plateau techniques in a future column.