Computer Science III (22C:30, 22C:115)
Project 3, Due: 12/13/02, 5 pm

1. Introduction.

In this programming project you will build on the code you developed for Projects 1 and 2 to produce a program that computes a *Hamiltonian cycle* in a given graph. A Hamiltonian cycle in a graph is a path that starts at a node, visits every node of the graph exactly once, and returns to the starting node. For example, a knight's tour is a Hamiltonian cycle in a graph whose nodes are the squares in the chessboard with two nodes connected by an edge if a knight can go from one node to the other in one hop. So the problem of computing a Hamiltonian cycle in a graph is a generalization of the problem of finding an open knight's tour.

2. Backtracking with Pruning.

The main thrust of the project is in making the recursive backtracking procedure that you used for Knight's Tour more sophisticated. Consider the backtracking procedure you used for the Knight's Tour problem and suppose that the procedure has found a path P thus far. Further suppose that v is the last node in P. Now the procedure has to decide which node to visit next after v. The procedure we implemented for the Knight's Tour problem was not very clever about this; it simply picked an *arbitrary* unvisited neighbor w and proceeded to visit w. However, to speed up the procedure there are two things we should attempt to do: (i) not pick any node w from which we have little hope of completing a Hamiltonian cycle and (ii) pick a node w from which the likelihood of completing a Hamiltonian cycle is high. Item (i) is a way of *pruning* away unnecessary search and if done well, will have a tremendous impact on the running time of the procedure. The question then is whether there is a quick way of identifying "bad" nodes w, that is, nodes w that we should try to avoid.

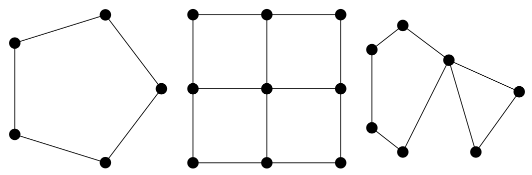
There are some heuristics that will help. First some terminology. Let s be the starting node of the path P we have computed thus far. Let G be the given graph and let H be the graph obtained by deleting the nodes in P from G. For w to be a good candidate the visit next, it must be the case that there is a path in H from w to some neighbor of s that visits every node in H exactly once. This implies several simple conditions that H must necessarily satisfy and if H does not satisfy any of these conditions then visiting w is a waste of time. These conditions are listed below.

(i) **Connectedness of $H - w$:** The graph obtained by deleting w from H must be connected, otherwise there is no hope of finding a path in H from w to a neighbor of s that visits every node in H. So the first test of whether w is any good, is checking the connectedness of the graph obtained by deleting w from H. This check for connectedness is easy to do via `bfs` or `dfs`. Specifically, you should consider adding a function called `isConnected` to the graph class that tests if a given graph is connected and make a call to this function. Recall that a graph is connected is there is a path between any two nodes in the graph.

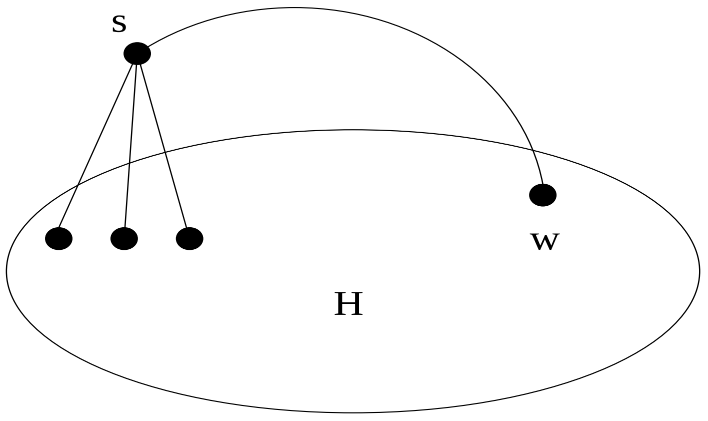
(ii) **The degrees of nodes in H:** If H contains a path from w to some neighbor of s that visits every node exactly once, it must be the case that every node in H has degree at least 2, with the possible exception of w and some neighbor of s. The second test of whether w is any good is this check on the degrees of the nodes in H. If the degree check fails, then there is no point in visiting w. You should consider implementing a boolean function called `degreeCheck` as part of the graph class to check this condition.

(iii) **Biconnectedness of $H + s + (s, w)$:** This is the test that is most difficult to implement, but this is also the test that will probably prune most successfully. The notion of "biconnectedness" is a generalization of the notion of connectedness.

A *biconnected graph* is a graph from which at least two nodes have to be deleted to break it up into disconnected pieces. Consider for example the three graphs shown below — the first two are biconnected, the third is not. The graph on the left, the cycle with 5 nodes, is biconnected because removing any one node leaves a path and we have to remove a second node to break this up into pieces. The graph in the middle, the grid graph, is also biconnected and in fact 3 nodes have to be removed to disconnect it. The graph on the right is not biconnected because you can remove just one node, the one that is shared by the two cycles, and we'll end up with two disconnected pieces.



Biconnectedness is important for network designers who want their network to be biconnected so that failure of a single machine in the network is not enough to break up the network into pieces that cannot talk to each other. How is biconnectedness relevant to us? As mentioned before, if w is a good candidate to visit next, it must be the case that there is a path in H from w to some neighbor of s that visits every node in H exactly once. This means that the graph T obtained from H by adding the node s to it (along with all incident edges) and an edge (s, w) contains a Hamiltonian cycle (see figure below). It should be obvious to you that any graph that contains a Hamiltonian cycle is also biconnected. Hence T ought to be biconnected. Therefore if we construct T, test for biconnectedness, and discover that it is not biconnected, then we know for certain that w is *not* a good candidate to visit next.



The graph T

You should consider implementing a function `isBiconnected` to test if a given graph is biconnected. Before committing to visit w, you should test if T is biconnected. If T is not biconnected then visiting w is a waste of time and a portion of the search can be pruned. Details of how to implement `isBiconnected` are given in the following section.

Note that even if H and w satisfy all three of the conditions above w might still be a bad choice. However, if H and w do not satisfy any of the above three conditions we know for sure that w is a bad choice and we can prune it right away.

3. Testing biconnectivity of graphs
It turns out that biconnectivity of a graph can be easily tested by modifying `dfs`. Define a *cutpoint* of a graph as a node whose removal breaks up the graph into disconnected pieces. From the definition of a biconnected graph it follows that a biconnected graph has no cutpoints. So to test for biconnectivity is equivalent to testing for cutpoints. If a graph contains no cutpoints, then it is biconnected; otherwise, if it does contain a cutpoint, it is not biconnected. We now show how to modify `dfs` so that it identifies any cutpoints the graph might contain.

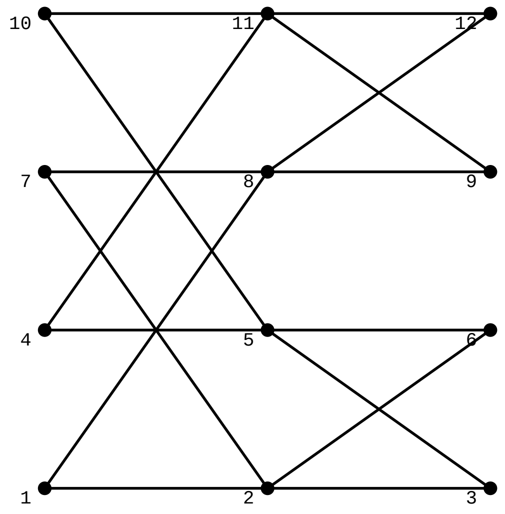
First recall how `dfs` works. At a node v, `dfs` scans the neighbors of v and if it encounters a neighbor u that has not yet been visited then it starts `dfs` at u. The edge (v, u) becomes a *tree edge*, an edge in the dfs tree connecting the parent v with u the child. On the other hand, if `dfs` encounters a neighbor u that has been already visited, it simply skips to the next neighbor. In this case the edge (v, u) is called a *back edge*. The key property of the back edge (v, u) is that u is an ancestor of v in the dfs tree.

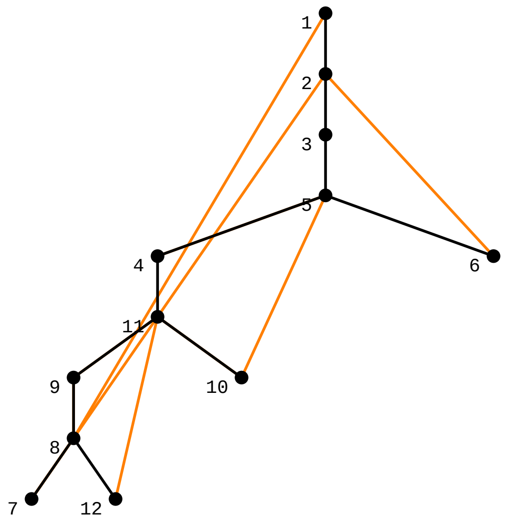
Thus dfs classifies edges into tree edges (edges that connect a node to a child in the dfs tree) and back edges (edges that connect a node to an ancestor in the dfs tree). We can then characterize cutpoints as follows. Suppose that we start the dfs of a graph at a node A and suppose that T is the dfs tree of the graph. Note that T is rooted at A. Then for any node x:

(i) if x is the root A then x is a cutpoint iff x has more than one child in T.

(ii) if x is not the root A, then x is a cutpoint iff x has a child y such that there is back edge from y or any descendent of y to a proper ancestor of x.

The explanation for item (a) is that if the root A has two (or more children) then deleting A will disconnect the children. The explanation for item (b) is that if there are no back edges from y or any descendent to a proper ancestor of x then deleting x disconnects the subtree rooted at y from the rest of the graph.





For example, look at the graph at the bottom of the previous page and its dfs tree above. I have shown the back edges also (in orange) in addition to the tree edges. First note that the root of the dfs tree has only one child and hence it is not a cutpoint. Now let us use the above characterization to check if some other node, say 5, is a cutpoint. 5 has two children 4 and 6 and there is a back edge from the subtree rooted at 4 going all the way up to 1 and there is a back edge from 6 going up to 4. Hence, 5 is not a cutpoint. It is easy to verify, using the same idea, that the graph has no cutpoint and is therefore biconnected.

The question now is simply how to use this characterization to efficiently determine if a node x is a cutpoint. The main idea is to modify `dfs` so that it maintains two additional pieces of information per node. Specifically, it maintains a *dfs number* for each node which is simply a numbering of the nodes that indicates the order in which the nodes were visited. The root of the dfs tree has dfs number 1, the node visited next has dfs number 2, the next node has dfs number 3, and so on. Let us denote the dfs number of a node v by $d[v]$. In addition, it maintains the *low* number for each node defined as

$$\text{low}[v] = \min\{d[v]\} \cup \{d[w] \mid (u, w) \text{ is a back edge for some descendent } u \text{ of } v\}.$$

Thus the low number of a node v is either the dfs number of v or the smallest dfs number of an ancestor to which there is a back edge from some descendent of v. There are two things you now need to figure out:

(i) What is the connection between the low numbers and cutpoints?

(ii) How are the low numbers computed efficiently by dfs.

These and other matters related to the testing of biconnectedness will be discussed in class (Monday 12/2).

4. Testing your project.
I'll make some small graphs available on the course web site indicating whether they are Hamiltonian or not. You can use these to test your project as it develops. A week before the due date (12/6) I'll post four larger graphs, `g1`, `g2`, `g3`, and `g4`. You are expected to run your program on each of these and submit files `g1.out`, `g2.out`, `g3.out`, and `g4.out`. `g1.out` should contain either a Hamiltonian cycle for graph `g1` or a message saying that `g1` has no Hamiltonian cycle. Similarly, `g2.out`, `g3.out`, and `g4.out`

5. Grading your project.
The grading scheme for your project is simple. You get 15 points (out of 75) if your project runs correctly on a small graph (of our choosing). You get 15 addition points for each large graph `g1`, `g2`, `g3`, or `g4` that your program is able to process successfully. The key to doing well is therefore making your program as fast as possible. There are absolutely no other design requirements your project needs to satisfy. This means that you can throw away everything from your current `graph` class implementation that you think might slow it down. For example, the `graph` class need not be templated. The graphs I will provide for testing have positive integer labels for the nodes. This also means that the function `getIndex` can be done away with. More than anything else, your strategies for pruning the search space will affect the running time of the function. You are most welcome to implement any strategies you come up with, besides the three I have listed in Section 2 of this handout.

6. Overall organization.
Submit the following files: `graph.h`, `graph.cxx`, `node.h`, `node.cxx`, `edge.h`, `edge.cxx`, and `driver.cxx`. The `graph` class should contain functions such as `isConnected`, `degreeCheck`, `isBiconnected`, and `HamiltonianCycle`. The file `driver.cxx` should contain code to read and construct the graph, a call to the `HamiltonianCycle` function, and finally some code to print the Hamiltonian cycle (or a message saying no Hamiltonian cycle exists).

In addition to these files that contain your code, you should also have a `README` file that clearly tells us about any known errors your programs have and also lists all required features not implemented.

All of these files should be in a directory that you will have to submit by 5 pm on 12/13/02.