Section Handout

Discussion Problem 1: People You May Know

Because Facebook is interested in growing out its social graph as much as possible, most users are presented with a list of other users who they think you might be friends with even though that friendship isn't officially recorded. That list is drawn from the set of Facebook users who are strictly two degrees away from you—that is, the list of your friends' friends that aren't already friends with you.

Assume that the following node definition is used to represent a Facebook user:

```
struct user {
   int userID;          // unique
   string name;         // not necessarily unique
   Set<user *> friends; // assume friendship is symmetric
};
```

Write a function called **`getFriendsOfFriends`**, which given the address of your node in the social graph, returns as a **`Set`** the collection of nodes representing those on Facebook who are two degrees away from you.

```
static Set<user *> getFriendsOfFriends(user *loggedinuser);
```

Discussion Problem 2: Detecting Cycles

Given access to a graph (in the form of an exposed **`SimpleGraph`**), write a predicate called **`containsCycles`**, which returns **`true`** if there are any cycles whatsoever in the graph, and **`false`** otherwise. The ability to detect cycles, and the ability to confirm that the addition of an edge doesn't introduce cycles, is important for some applications (i.e. Excel needs to confirm that no two cell formulas mutually depend—directly or eventually—on each other, and C++ compilers sometimes elect to check that no two header files mutually **`#include`** one an other.)

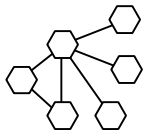
For this problem (and the next one), use the exposed **`Node`**, **`Arc`**, and **`SimpleGraph`** record definitions defined in Chapter 18.

```
static bool containsCycles(SimpleGraph& graph);
```

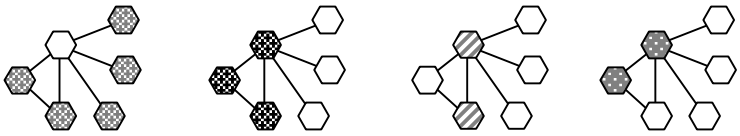
Discussion Problem 3: Minimum Vertex Cover

A **vertex cover** is a subset of an undirected graph's vertices such that each and every arc in the graph is incident to at least one vertex in the subset. A **minimum vertex cover** is a vertex cover of the smallest possible size.

Consider the following graph:



Each of the following illustrates some vertex cover (shaded, textured nodes are included in the vertex cover, and hollow ones are excluded):



Each is technically a vertex cover, because in each, each arc touches one if not two vertices in the cover. The two vertex covers on the right are each minimum vertex covers, because there's no single vertex that's attached to all arcs.

Write the **`computeMinimumVertexCover`** function, which accepts a reference to an undirected **`SimpleGraph`** and returns a **`Set<Node *>`** identifying some minimum vertex cover. Understand that because the graph is undirected, that means for every arc that leads from **`n1`** to **`n2`**, there will be an arc that leads from **`n2`** to **`n1`**. If there are two or more minimum vertex covers, then you can return any one of them. The implementation of this function should consider every possible vertex subset, keeping track of the smallest one that covers the entire graph. For reasons just beyond the scope of the class, there is an overwhelming amount of evidence suggesting that there aren't any better algorithms than the one that compares all node subsets to one another.

```
static Set<Node *> computeMinimumVertexCover(SimpleGraph& graph);
```

Lab Problem 1: Tournaments and Kings

A **complete** graph is one where every single graph node is connected to every other node, both in the forward and backwards directions. That is, for distinct nodes **m** and **n**, there exists an arc connecting **m** to **n**, and another connecting **n** to **m**.

Tournament graphs are the directed graphs that come from a complete graph when we impose a direction on each and every arc. Equivalently, a tournament graph results when one of the two edges bridging **m** and **n**—for all choices of **m** and **n**—is removed, leaving a clear directionality on the connection between any two nodes. Below, on the left is a complete graph on five nodes, and on the right is a tournament on five nodes.

Informally, a tournament is a visual summary of who prevailed over whom in an exhaustive competition of one-on-one matches, where every single person eventually competes—exactly once—against everyone else. The tournament on the above right states that player 1 beat players 2, 3, and 4 (but not 5), that player 2 lost to everybody, and so forth.

A **tournament king** is a node in a tournament representing someone who, for every other player, either directly prevailed over that player, or prevailed over someone who prevailed over that player. In other words, a node is a king if one can travel from it to every other node via a path of at most 2 arcs. So, in the above graph, player 1 is a king, because player 1 prevailed over players 2, 3, and 4. And while player 1 lost to player 5—player 1's only loss—player 1 triumphed over player 3, who managed to defeat player 5. As it is, players 1, 3, and 5 are all kings, but players 2 and 4 are not.

The test framework for this problem consists of two **`.cpp`** files: **`tournament-graph-kings.cpp`** and **`tournament-graph-test.cpp`**. The code provided already reads in all graphs from provided data files, so you don't need to worry about that. All you need to do is write a function called **`crownTournamentKings`** that, given a tournament **`SimpleGraph`** reference, returns a set of all **`Node *`**s identifying tournament kings. Simple definitions for **`Node`**, **`Arc`**, and **`SimpleGraph`** are presented in **`graphtypes.h`**. You should update the **`tournament-graph-kings.cpp`** file with your implementation. You shouldn't need to make any changes to **`graphtypes.h`**, **`tournament-graph-test.cpp`**, or **`tournament-graph-kings.h`**.