Section Solution

Discussion Problem 1 Solution: People You May Know

We use a brute force double **foreach** loop to gain access to all of your friends' friends. We maintain a **peopleYouAlreadyKnow** set (yourself, all of your friends) so that we don't accidentally include a friend in the return value.

The solution here makes the reasonable assumption that each user is uniquely identified by the address of his or her **user** record.

```
static Set<user *> getFriendsOfFriends(user *loggedinuser) {
   Set<user *> peopleYouMayKnow;
   Set<user *> peopleYouAlreadyKnow = loggedinuser->friends;
   peopleYouAlreadyKnow += loggedinuser;
   foreach (user *fr in loggedinuser->friends) {
      foreach (user *friendOfFriend in fr->friends) {
         if (!peopleYouAlreadyKnow.contains(friendOfFriend)) {
            peopleYouMayKnow += friendOfFriend;
         }
      }
   }

   return peopleYouMayKnow;
}
```

Note that there's no real need to check to see if a friend of a friend has already been added to the **peopleYouMayKnow** set. There's no harm in adding the same item multiple times, as the set discards all duplicates.

Discussion Problem 2 Solution: Detecting Cycles

This is a variation on the depth-first traversal example presented in the reader. The trick is to maintain a list of **Node ***s actively being explored, and if we ever trip over the same node twice during a depth-first exploration, then we have a cycle and need to report that back.

The one feature of this particular solution is that it properly handles those graphs that aren't fully connected, but instead come as two or more disconnected components.

```
static bool isReachable(Node *node,
                        Set<Node *>& activelyBeingVisited,
                        Set<Node *>& previouslyVisited) {
   if (activelyBeingVisited.contains(node)) return true;
   if (previouslyVisited.contains(node)) return false;
   activelyBeingVisited += node;
   foreach (Arc *arc in node->arcs) {
      if (isReachable(arc->finish, activelyBeingVisited, previouslyVisited)) {
         return true;
      }
   }

   activelyBeingVisited -= node;
   previouslyVisited += node;
   return false;
}

static bool containsCycle(SimpleGraph& graph) {
   Set<Node *> previouslyVisited;
   Set<Node *> toBeVisited = graph.nodes;
   while (!toBeVisited.isEmpty()) {
      Node *front = toBeVisited.first();
      Set<Node *> activelyBeingVisited;
      if (isReachable(front, activelyBeingVisited, previouslyVisited)) {
         return true;
      }
      toBeVisited -= previouslyVisited;
   }

   return false;
}
```

Discussion Problem 3 Solution: Minimum Vertex Cover

```
static void computeMinimumVertexCover(Set<Node *>& coveringNodes,
                                      Set<Arc *>& coveredArcs,
                                      int numArcs,
                                      Vector<Node *>& allNodes,
                                      int start,
                                      Set<Node *>& bestCover) {

   if (coveringNodes.size() >= bestCover.size()) return;
   if (coveredArcs.size() == numArcs) {
      bestCover = coveringNodes;
      return;
   }

   if (start == allNodes.size()) return;
   computeMinimumVertexCover(coveringNodes, coveredArcs, numArcs,
                             allNodes, start + 1, bestCover);
   coveringNodes += allNodes[start];
   Set<Arc *> newlyCoveredArcs;
   foreach (Arc *arc in allNodes[start]->arcs) {
      if (!coveredArcs.contains(arc)) {
         newlyCoveredArcs += arc;
         coveredArcs += arc;
      }
   }

   computeMinimumVertexCover(coveringNodes, coveredArcs, numArcs,
                             allNodes, start + 1, bestCover);
```

```
   coveringNodes -= allNodes[start];
   coveredArcs -= newlyCoveredArcs;
}

static Set<Node *> computeMinimumVertexCover(SimpleGraph& graph) {
   Set<Node *> bestCover = graph.nodes; // upper bound on optimal solution
   Set<Node *> coveringNodes;
   Set<Arc *> coveredArcs;

   Vector<Node *> allNodes;
   foreach (Node *node in graph.nodes) allNodes += node;

   computeMinimumVertexCover(coveringNodes,
                             coveredArcs,
                             graph.arcs.size(),
                             allNodes,
                             0,
                             bestCover);
   return bestCover;
}
```

Lab Problem 1 Solution: Tournament Kings

The solution is a brute force examination of all of the nodes to see whether or not they satisfy a certain property. The following captures the core of what you needed to write:

```
static bool isKing(Node *winner, int numOpponents) {
   Set<Node *> beaten;
   foreach (Arc *arc1 in winner->arcs) {
      Node *loser = arc1->finish;
      beaten += loser;
      foreach (Arc *arc2 in loser->arcs) {
         Node *loserToLoser = arc2->finish;
         beaten += loserToLoser;
      }
   }

   return beaten.size() == numOpponents;
}

Set<Node *> crownTournamentKings(SimpleGraph& graph) {
   Set<Node *> kings;
   foreach (Node *node in graph.nodes) {
      if (isKing(node, graph.nodes.size() - 1)) {
         kings += node;
      }
   }

   return kings;
}
```