

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.

```
Set<user *> getFriendsOfFriends(user *loggedinuser) {
    Set<user *> peopleYouMayKnow;
    Set<user *> peopleYouAlreadyKnow = loggedinuser->friends;
    peopleYouAlreadyKnow.add(loggedinuser);
    foreach (user *fr in loggedinuser->friends) {
        foreach (user *friendOfFriend in fr->friends) {
            if (!peopleYouAlreadyKnow.contains(friendOfFriend)) {
                peopleYouMayKnow.add(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 any and all duplicates.

Discussion Problem 2 Solution: Detecting Cycles

This is a variation on the depth-first traversal example in the reader. The trick is to maintain a list of **nodeT** *s actively being explored, and if we ever trip over such a node during our exploration, then we know 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.

```
bool componentReachableFromContainsCycle(nodeT *node,
                                         Set<nodeT *>& activelyBeingVisited,
                                         Set<nodeT *>& previouslyVisited) {
    if (activelyBeingVisited.contains(node)) return true;
    if (previouslyVisited.contains(node)) return false;
    activelyBeingVisited.add(node);
    foreach (arcT *arc in node->arcs) {
        if (componentReachableFromContainsCycle(arc->finish,
                                                activelyBeingVisited,
                                                previouslyVisited)) {
            return true;
        }
    }
}
```

```

    }
}

activelyBeingVisited.remove(node);
previouslyVisited.add(node);
return false;
}

bool containsCycle(graphT& graph) {
    Set<nodeT *> previouslyVisited;
    Set<nodeT *> toBeVisited = graph.nodes;
    while (toBeVisited.size() > 0) {
        Set<nodeT *>::Iterator iter = toBeVisited.iterator();
        nodeT *node = iter.next();
        Set<nodeT *> activelyBeingVisited;
        if (componentReachableFromContainsCycle(node,
                                                activelyBeingVisited,
                                                previouslyVisited)) {
            return true;
        }

        toBeVisited.subtract(previouslyVisited);
    }

    return false;
}

```

Discussion Problem 3 Solution: Minimum Vertex Cover

The following solution makes use of the **Set<T>::Iterator**. It's clean and clever and one of the better ways to implement the solution.

```

void computeMinimumVertexCover(Set<nodeT *>& coveringNodes,
                               Set<arcT *>& coveredArcs,
                               int numArcs,
                               Set<nodeT *>::Iterator iter, // intentional copy
                               Set<nodeT *>& bestCover) {

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

    if (!iter.hasNext()) return;
    nodeT *node = iter.next();
    computeMinimumVertexCover(coveringNodes, coveredArcs,
                              numArcs, iter, bestCover);

    coveringNodes.add(node);
    Set<arcT *> newlyCoveredArcs;
    foreach (arcT *arc in node->arcs) {
        if (!coveredArcs.contains(arc)) {
            newlyCoveredArcs.add(arc);
            coveredArcs.add(arc);
        }
    }

    computeMinimumVertexCover(coveringNodes, coveredArcs,
                              numArcs, iter, bestCover);
    coveringNodes.remove(node);
    coveredArcs.subtract(newlyCoveredArcs);
}

```

```

}

Set<nodeT *> computeMinimumVertexCover(graphT& graph) {
    Set<nodeT *> bestCover = graph.nodes; // upper bound on optimal solution
    Set<nodeT *> coveringNodes;
    Set<arcT *> coveredArcs;
    Set<nodeT *>::Iterator iter = graph.nodes.iterator();
    computeMinimumVertexCover(coveringNodes, coveredArcs,
                              graph.arcs.size(), iter, bestCover);
    return bestCover;
}

```

The above solution has the distinct disadvantage of using a construct—the **Iterator**—that we haven’t really focused on, because under most circumstances we’ve used the **foreach** construct instead.

Another approach—relatively amateur but nonetheless clear and correct—is to transfer all of the **nodeT** *s to a **Vector** and recur on that instead. We have more experience with that type of recursion, so this seems like the fair and more reasonable solution to expect from you.

```

Set<nodeT *> computeMinimumVertexCover(graphT& graph) {
    Set<nodeT *> bestCover = graph.nodes; // upper bound on optimal solution
    Set<nodeT *> coveringNodes;
    Set<arcT *> coveredArcs;

    Vector<nodeT *> allNodes;
    foreach (nodeT *node in graph.nodes) {
        allNodes.add(node);
    }

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

    return bestCover;
}

void computeMinimumVertexCover(Set<nodeT *>& coveringNodes,
                               Set<arcT *>& coveredArcs,
                               int numArcs,
                               Vector<nodeT *>& allNodes,
                               int start,
                               Set<nodeT *>& 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.add(allNodes[start]);
Set<arcT *> newlyCoveredArcs;
foreach (arcT *arc in allNodes[start]->arcs) {
    if (!coveredArcs.contains(arc)) {
        newlyCoveredArcs.add(arc);
        coveredArcs.add(arc);
    }
}

computeMinimumVertexCover(coveringNodes,
                           coveredArcs,
                           numArcs,
                           allNodes,
                           start + 1,
                           bestCover);
coveringNodes.remove(allNodes[start]);
coveredArcs.subtract(newlyCoveredArcs);
}

```

The above is really just the same exact algorithm, reframed to run over a **Vector** instead of an **Iterator**.

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 core of what you needed to write is captured by the following:

```

bool isKing(nodeT *winner, int numOpponents) {
    Set<nodeT *> beaten;
    foreach (arcT *arc1 in winner->arcs) {
        nodeT *loser = arc1->finish;
        beaten.add(loser);
        foreach (arcT *arc2 in loser->arcs) {
            nodeT *loserToLoser = arc2->finish;
            beaten.add(loserToLoser);
        }
    }

    return beaten.size() == numOpponents;
}

Set<nodeT *> crownTournamentKings(graphT& graph) {
    Set<nodeT *> kings;
    foreach (nodeT *node in graph.nodes) {
        if (isKing(node, graph.nodes.size() - 1)) {
            kings.add(node);
        }
    }

    return kings;
}

```