

Solution to Section #7

Based on handouts by various current and past CS106B/X instructors and TAs.

1. Graph Properties

- Graph 1: directed, unweighted, not connected, cyclic
 - degrees: A=(in 0 out 2), B=(in 2 out 1), C=(in 1 out 1), D=(in 2 out 1), E=(in 2 out 2), F=(in 2 out 1), G=(in 2 out 1), H=(in 2 out 2), I=(in 0 out 2)
- Graph 2: undirected, unweighted, connected, acyclic
 - degrees: A=1, B=3, C=1, D=2, E=2, F=1
- Graph 3: directed, unweighted, not connected, cyclic
 - degrees: A=(in 1 out 2), B=(in 3 out 1), C=(in 0 out 1), D=(in 2 out 1), E=(in 1 out 2)
- Graph 4: undirected, weighted, not connected, cyclic
 - degrees: A=2, B=2, C=2, D=1, E=1
- Graph 5: undirected, unweighted, connected, cyclic
 - degrees: A=3, B=3, C=3, D=3
- Graph 6: directed, weighted, not connected (weakly connected), cyclic
 - degrees: A=(in 2 out 2), B=(in 2 out 3), C=(in 2 out 3), D=(in 2 out 0), E=(in 2 out 2), F=(in 3 out 2), G=(in 1 out 2)

2. Depth-First Search (DFS)

Graph 1

A to B: {A, B}
A to C: {A, B, E, F, C}
A to D: {A, B, E, D}
A to E: {A, B, E}
A to F: {A, B, E, F}
A to G: {A, B, E, D, G}
A to H: {A, B, E, D, G, H}
A to I: no path

Graph 6

A to B: {A, C, B}
A to C: {A, C}
A to D: {A, C, D}
A to E: {A, C, B, F, E}
A to F: {A, C, B, F}
A to G: {A, C, G}

3. Breadth-First Search (BFS)

BFS paths that are shorter than the DFS paths are underlined

Graph 1

A to B: {A, B}
A to C: {A, B, E, F, C}
A to D: {A, D}
A to E: {A, B, E}
A to F: {A, B, E, F}
A to G: {A, D, G}
A to H: {A, D, G, H}
A to I: no path

Graph 6

A to B: {A, C, B}
A to C: {A, C}
A to D: {A, C, D}
A to E: {A, E}
A to F: {A, E, F}
A to G: {A, C, G}

4. kth Level Friends

```

Set<Vertex *> kthLevelFriends(BasicGraph& graph, Vertex *v, int k) {
    Set<string> result;
    HashMap<Vertex*, int> distances;
    Queue<Vertex*> q;

    // initialize BFS
    q.enqueue(v);
    distances[v] = 0;

    while(!q.isEmpty()){
        Vertex *curr = q.dequeue();
        if (distances[curr] == k){
            result += curr;
        }
        if (distances[curr] > k) {
            break;
        }
        for (Vertex* buddy : graph.getNeighbors(curr)) {
            if (!distances.containsKey(buddy)){
                distances[buddy] = distances[curr] + 1;
                q.enqueue(buddy);
            }
        }
    }
    return result;
}

```

5. Cycling

```

bool hasCycle(BasicGraph &graph) {
    Set<Vertex *> previouslyVisited;
    Set<Vertex *> toBeVisited = graph.getVertexSet();
    while (!toBeVisited.isEmpty()) {
        Vertex *front = toBeVisited.first();
        Set<Vertex *> activelyBeingVisited;
        if (isReachable(front, activelyBeingVisited, previouslyVisited)) {
            return true;
        }
        toBeVisited -= previouslyVisited;
    }
    return false;
}

bool isReachable(Vertex *v, Set<Vertex *> &activelyBeingVisited, Set<Vertex
*> &previouslyVisited) {
    if (activelyBeingVisited.contains(v)) {
        return true;
    } else if (previouslyVisited.contains(v)) {
        return false;
    }
    activelyBeingVisited += v;
    for (Edge *e : v->edges) {
        if (isReachable(e->finish, activelyBeingVisited, previouslyVisited)){
            return true;
        }
    }
}

```

```

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

```

6. Is it Reachable?

```

bool isReachable(BasicGraph& graph, Vertex* v1, Vertex* v2) {
    Set<Vertex*> visited;
    return isReachable(graph, v1, v2, visited);
}

bool isReachable(BasicGraph& graph, Vertex* v1, Vertex* v2, Set<Vertex*>
visited) {
    if (v1 == v2) {
        return true;
    }
    visited += v1;
    for (Edge* edge : graph.getEdgeSet(v1)) {
        Vertex* neighbor = edge->finish;
        if (!visited.contains(neighbor) && isReachable(graph, neighbor, v2,
visited)) {
            return true;
        }
    }
    return false;
}

```

An alternate BFS (instead of DFS) solution:

```

bool isReachable(BasicGraph& graph, Vertex* v1, Vertex* v2) {
    Queue<Vertex*> toExplore;
    Set<Vertex*> visited;
    visited += v1;
    toExplore.enqueue(v1);
    while (!toExplore.isEmpty()) {
        Vertex* next = toExplore.dequeue();
        if (next == v2) {
            return true;
        }
        for (Vertex* neighbor : graph.getNeighbors(next)) {
            if (!visited.contains(neighbor)) {
                visited += neighbor;
                toExplore.enqueue(neighbor);
            }
        }
    }
    return false;
}

```

7. Is it Connected?

```

bool isConnected(BasicGraph& graph) {
    for (Vertex* v1 : graph.getVertexSet()) {
        for (Vertex* v2 : graph.getVertexSet()) {

```

```

        if (v1 != v2 && !isReachable(graph, v1, v2)) {
            return false;
        }
    }
}
return true;
}

```

8. Crowning the Champion

```

Set<Vertex *> crownTournamentChampions(BasicGraph &graph) {
    Set<Vertex *> champions;
    for (Vertex *node : graph.getVertexSet()) {
        if (isChampion(node, graph.getVertexSet().size() - 1)) {
            champions += node;
        }
    }
    return champions;
}

bool isChampion(Vertex *winner, int numOpponents) {
    Set<Vertex *> beaten;
    for (Edge *e1 : winner->edges) {
        Vertex *loser = e1->finish;
        beaten += loser;
        for (Edge *e2 : loser->edges) {
            Vertex *loserToLoser = e2->finish;
            beaten += loserToLoser;
        }
    }
    return beaten.size() == numOpponents;
}

```

9. Minimum Vertex Cover

```

Set<Vertex*> findMinimumVertexCover(BasicGraph& graph) {
    Set<Vertex*> best = graph.getVertexSet(); // worst case solution
    Set<Vertex*> chosen;
    Set<Edge*> coveredEdges;
    Vector<Vertex*> allVertices;

    for (Vertex* v : graph.getVertexSet()) {
        allVertices += v;
    }
    coverHelper(graph, chosen, coveredEdges, allVertices, 0, best);
    return best;
}

void coverHelper(BasicGraph& graph, Set<Vertex*>& chosen, Set<Edge*>&
coveredEdges, Vector<Vertex*>& allVertices, int index, Set<Vertex*>& best) {
    if (chosen.size() >= best.size()) {
        // base case: current cover too large
        return;
    } else if (coveredEdges.size() == graph.getEdgeSet().size()) {
        // base case: found a new smaller cover that uses all edges;
        // remember it
        best = chosen;
    }
}

```

```
        return;
    } else if (index == graph.getVertexSet().size()) {
        // base case: exhausted all vertices to explore
        return;
    } else {
        // recursive case: explore whether to include the current vertex
        // (the one at index) in the current vertex cover
        // choose not to include this vertex; explore
        coverHelper(graph, chosen, coveredEdges, allVertices, index + 1,
                    best);
        // choose to include this vertex; explore
        chosen += allVertices[index];
        // remember which new edges are added here (to un-choose later)
        Set<Edge*> newEdges;
        for (Edge* e : graph.getEdgeSet(allVertices[index])) {
            if (!coveredEdges.contains(e)) {
                // must add this edge and its inverse (A -> B and B -> A)
                Edge* inverse = graph.getEdge(e->finish, e->start);
                newEdges += e, inverse;
                coveredEdges += e, inverse;
            }
        }
        coverHelper(graph, chosen, coveredEdges, allVertices, index + 1,
                    best);
        // unchoose
        chosen -= allVertices[index];
        coveredEdges -= newEdges;
    }
}
```