Section Handout #7 Solutions

If you have any questions about the solutions to the problems in this handout, feel free to reach out to your section leader, Jason, or Chris for more information.

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)

<table>
<thead>
<tr>
<th>Graph 1</th>
<th>Graph 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A to B: {A, B}</td>
<td>A to B: {A, C, B}</td>
</tr>
<tr>
<td>A to C: {A, B, E, F, C}</td>
<td>A to C: {A, C}</td>
</tr>
<tr>
<td>A to D: {A, B, E, D}</td>
<td>A to D: {A, C, D}</td>
</tr>
<tr>
<td>A to E: {A, B, E}</td>
<td>A to E: {A, C, B, F, E}</td>
</tr>
<tr>
<td>A to F: {A, B, E, F}</td>
<td>A to F: {A, C, B, F}</td>
</tr>
<tr>
<td>A to G: {A, B, E, D, G}</td>
<td>A to G: {A, C, G}</td>
</tr>
<tr>
<td>A to H: {A, B, E, D, G, H}</td>
<td>A to I: no path</td>
</tr>
</tbody>
</table>

3. Breadth-First Search (BFS)

*BFS paths that are shorter than the DFS paths are underlined.*

<table>
<thead>
<tr>
<th>Graph 1</th>
<th>Graph 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A to B: {A, B}</td>
<td>A to B: {A, C, B}</td>
</tr>
<tr>
<td>A to C: {A, B, E, F, C}</td>
<td>A to C: {A, C}</td>
</tr>
<tr>
<td>A to D: {A, D}</td>
<td>A to D: {A, C, D}</td>
</tr>
<tr>
<td>A to E: {A, B, E}</td>
<td>A to E: {A, E}</td>
</tr>
<tr>
<td>A to F: {A, B, E, F}</td>
<td>A to F: {A, E, F}</td>
</tr>
<tr>
<td>A to G: {A, D, G}</td>
<td>A to G: {A, C, G}</td>
</tr>
<tr>
<td>A to H: {A, D, G, H}</td>
<td>A to I: no path</td>
</tr>
</tbody>
</table>
4. Minimum Weight Paths
Paths that are lower weight than BFS or DFS are underlined.

<table>
<thead>
<tr>
<th>Paths</th>
<th>Weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>A to B:</td>
<td>{A, E, F, B}</td>
</tr>
<tr>
<td>A to C:</td>
<td>{A, E, F, B, C}</td>
</tr>
<tr>
<td>A to D:</td>
<td>{A, E, F, B, C, G, D}</td>
</tr>
<tr>
<td>A to E:</td>
<td>{A, E}</td>
</tr>
<tr>
<td>A to F:</td>
<td>{A, E, F}</td>
</tr>
<tr>
<td>A to G:</td>
<td>{A, E, F, B, C, G}</td>
</tr>
</tbody>
</table>

5. kth Level Friends
Set<Vertex *> kthLevelFriends(BasicGraph &graph, Vertex *v, int k) {
    Set<Vertex *> result;
    Set<Vertex *> known;
    kthLevelFriendsHelper(graph, v, known, result, k);
    return result;
}
void kthLevelFriendsHelper(BasicGraph &graph, Vertex *v, Set<Vertex *> &known,
Set<Vertex *> &result, int k) {
    if (k == 0) {
        result.add(v);
    } else {
        known += v;
        for (Vertex *friend : graph.getNeighbors(v)) {
            if (!known.contains(friend)) {
                kthLevelFriendsHelper(graph, friend, known, result, k - 1);
            }
        }
    }
}

6. Has Cycle
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.getEdgeSet()) {
    if (isReachable(edge->finish, activelyBeingVisited, previouslyVisited)) {
        return true;
    }
}
activelyBeingVisited -= v;
previouslyBeingVisited += v;
return false;

7. Is Connected

bool isConnected(BasicGraph &graph) {
    for (Vertex *v1 : graph.getVertexSet()) {
        for (Vertex *v2 : graph.getVertexSet()) {
            if (v1 != v2 && !isReachable(graph, v1, v2)) { // isReachable defined in hasCycle
                return false;
            }
        }
    }
    return true;
}

8. Game of Thrones

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

Set<Vertex *> crownTournamentKings(BasicGraph &graph) {
    Set<Vertex *> kings;
    for (Vertex *node : graph.getVertexSet()) {
        if (isKing(node, graph.getVertexSet().size() - 1)) {
            kings += node;
        }
    }
    return kings;
}
9. Minimum Vertex Cover

```cpp
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);
}

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 smaller cover that uses all edges; save it
        best = chosen;
        return;
    } else if (index == graph.getVertexSet().size()) {
        // exhausted all vertices to explore
        return;
    }
    // two recursive calls:
    // 1) choose not to include current vertex
    coverHelper(graph, chosen, coveredEdges, allVertices, index + 1, best);
    // 2) chose to include this vertex
    chosen += allVertices[index];
    // remember which new edges are added here (to unchoose later)
    Set<Edge *> newEdges;
    for (Edge *e : graph.getEdgeSet(allVertices[index]))) {
        if (!coveredEdges.contains(e)) {
            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;
}
```