

Section 7 (Week 8) – SOLUTION

Problem and solution authors include Marty Stepp

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 1),
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. 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. BFS (*shorter paths 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. minimum weight paths (*lower weight paths underlined*)

A to B: {A, E, F, B}, weight=5
 A to C: {A, E, F, B, C}, weight=6
 A to D: {A, E, F, B, C, G, D}, weight=12
 A to E: {A, E}, weight=1
 A to F: {A, E, F}, weight=3
 A to G: {A, E, F, B, C, G}, weight=11

5. kthLevelFriends

```
Set<Vertex*> kthLevelFriends(BasicGraph& graph, Vertex* v, int k) {
    Set<Vertex*> result;
    Set<Vertex*> known;
    kthLevelHelper(graph, v, known, result, k);
    return result;
}

void kthLevelHelper(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)) {
                kthLevelHelper(graph, friend, known, result, k - 1);
            }
        }
    }
}
```

6. isReachable

DFS solution:

```
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;
    foreach (Edge* edge in graph.getEdgeSet(v1)) {
        Vertex* neighbor = edge->finish;
        if (!visited.contains(neighbor)
            && isReachable(graph, neighbor, v2, visited)) {
            return true;
        }
    }
    return false;
}
```

BFS 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. isConnected

```
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. findMinimumVertexCover

```

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 or not 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 (so that we can un-choose later)
        Set<Edge*> newEdges;
        for (Edge* e in 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;
    }
}

```