CS 106B Section 8 (Week 9) Solutions

1. 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}

2. Breadth-First Search (BFS) – *shorter paths underlined*

Graph 1:

A to B: {A, B}
A to C: {A, B, E, F, C}
A to D: <u>{A, D}</u>
A to E: {A, B, E}
A to F: {A, B, E, F}
A to G: <u>{A, D, G}</u>
A to H: <u>{A, D, G, H}</u>
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: <u>{A, E}</u>
A to F: <u>{A, E, F}</u>
A to G: {A, C, G}

3. Minimum weight paths – *lower weight paths underlined*

A to B: <u>{A, E, F, B}</u>, weight=5

A to C: <u>{A, E, F, B, C}</u>, weight=6

A to D: <u>{A, E, F, B, C, G, D}</u>, weight=12

A to E: {A, E}, weight=1

A to F: {A, E, F}, weight=3

A to G: <u>{A, E, F, B, C, G}</u>, weight=11

CS 106B Section 8 (Week 9) Solutions

4. 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;
    for (Edge* edge : 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;
}
```

5. 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;
}
```

CS 106B Section 8 (Week 9) Solutions

6. 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 : 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;
    }
}
```

7. Dijkstra's Algorithm

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