Nick Troccoli
CS 106X



# Section #7 Solutions

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

## 1. Graph properties *(graphs)*

**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=4, B=4, C=2, D=3, G=4, H=3, I=2

**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 *(graphs)*

**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 *(graphs)*

Paths that are shorter here than in DFS are 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}

## 4. Minimum weight paths *(graphs)*

Paths with a lower weight here than in problems 2 or 3 are underlined.

**In graph 6:**

```
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. Dijkstra's and A* *(graphs)*

**Dijkstra's:** (graph represented here by {vertex1:minCost,prevVertex, vertex2:minCost,prevVertex, etc.}

```
graph : {A:0,/, B:inf,/, C:inf,/, D:inf,/, E:inf,/, F:inf,/, G:inf,/}
pqueue: {A:0}
remove A, process neighbors B/C/E, update B cost to 4, C to 5, E to 1
graph : {A:0,/, B:4,A, C:5,A, D:inf,/, E:1,A, F:inf,/, G:inf,/}
pqueue: {E:1, B:4, C:5}
remove E, process neighbor F, update F cost to 10
graph : {A:0,/, B:4,A, C:5,A, D:inf,/, E:1,A, F:10,E, G:inf,/}
pqueue: {B:4, C:5, F:10}
remove B, process neighbors C/F, update F cost to 6
graph : {A:0,/, B:4,A, C:5,A, D:inf,/, E:1,A, F:6,B, G:inf,/}
pqueue: {C:5, F:6}
remove C, process neighbors D/G, update D cost to 12, G cost to 8
graph : {A:0,/, B:4,A, C:5,A, D:12,C, E:1,A, F:6,B, G:8,C}
pqueue: {F:6, G:8, D:12}
remove F ... no unprocessed neighbors, no updates.
remove G, process neighbor D, update D cost to 9
graph : {A:0,/, B:4,A, C:5,A, D:9,G, E:1,A, F:6,B, G:8,C}
pqueue: {D:9}
remove D... no unprocessed neighbors, no updates.
```

**Final paths from Dijkstra's:**

```
A to B: {A, B}, cost=4
A to C: {A, C}, cost=5
A to D: {A, C, G, D}, cost=9
A to E: {A, E}, cost=1
A to F: {A, B, F}, cost=6
A to G: {A, C, G}, cost=8
```

**Because of the heuristic, A* would preferentially pursue paths that are in the general direction of the target vertices over paths that are generally in the wrong spatial direction. Yes, A* will <u>always</u> find the same path as Dijkstra's (A* would generally just do it faster).**

## 6. kthLevelFriends *(graphs)*

```
// approach 1: BFS
Set<Vertex*> kthLevelFriendsBFS(BasicGraph& graph, Vertex* v, int k) {
    Set<Vertex*> result;
    HashMap<Vertex*, int> distances;
    Queue<Vertex*> q;

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

    while(!q.isEmpty()){
        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;
}

// approach 2: DFS
void kthLevelDFSHelper(BasicGraph& graph, Vertex* v, Set<Vertex*>& potentialKthFriends,
                       Set<Vertex*>& tooShort, int k) {
    // base case: found a (potential) kth level neighbor, so add it to that set!
    if (k == 0) {
        potentialKthFriends.add(v);
        return;
    }
    // non-base case: a path with fewer than k hops exists to this vertex
    // its extended neighbors might be kth level friends even though this vertex is not one
    if (k > 0) {
        tooShort.add(v);
    }
    // recursive step: explore all of v's neighbors
    for (Vertex* neighbor : graph.getNeighbors(v)) {
        kthLevelDFSHelper(graph, neighbor, potentialKthFriends, tooShort, k - 1);
    }
}

Set<Vertex*> kthLevelFriendsDFS(BasicGraph& graph, Vertex* v, int k) {
    Set<Vertex*> potentialKthFriends;
    Set<Vertex*> tooShort;
    kthLevelDFSHelper(graph, v, potentialKthFriends, tooShort, k);
    //vertices accessible via paths shorter than k hops should be excluded from result
    return potentialKthFriends - tooShort;
}
```

## 7. hasCycle *(graphs)*

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

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;
}
```

## 8. findMinimumVertexCover *(graphs)*

```
void findCoverHelper(BasicGraph& graph, Set<Vertex*>& chosen, Set<Edge*>& coveredEdges,
                     Vector<Vertex*>& allVertices, int index, Set<Vertex*>& best) {
    if (chosen.size() >= best.size()) {
        return;          // base case: current cover too large
    } else if (coveredEdges.size() == graph.getEdgeSet().size()) {
        best = chosen;   // base case: found new smaller cover w/all edges; store it
    } else if (index == graph.getVertexSet().size()) {
        return;          // base case: exhausted all vertices to explore
    } 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
        findCoverHelper(graph, chosen, coveredEdges, allVertices, index + 1, best);
        chosen += allVertices[index];   // choose to include this vertex; explore
```

// continued on next page

```
        // 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;
            }
        }
        findCoverHelper(graph, chosen, coveredEdges, allVertices, index + 1, best);

        chosen -= allVertices[index];   // unchoose
        coveredEdges -= newEdges;
    }
}

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.add(v);
    }
    findCoverHelper(graph, chosen, coveredEdges, allVertices, 0, best);
    return best;
}
```

## 9. tournamentWinners *(graphs)*

```
// version 1: two-level BFS
bool isWinner(BasicGraph& tourney, Vertex* player) {
    Set<Vertex*> oneHop = tourney.getNeighbors(player);
    Set<Vertex*> twoHop;
    for (Vertex* vanquished : oneHop) {
        twoHop += tourney.getNeighbors(vanquished);
    }
    return (oneHop + twoHop + player).size() == tourney.size();
}

Set<Vertex*> tournamentWinners(BasicGraph& tourney) {
    Set<Vertex*> winners;
    for (Vertex* player : tourney) {
        if (isWinnerBFS(tourney, player)) {
            winners.add(player);
        }
    }
    return winners;
}
```

```
// version 2: brute force (substitute for isWinner function call above)
bool isWinnerBruteForce(BasicGraph& tourney, Vertex* player) {
    for (Vertex* other : tourney) {
        if (other == player) { continue; }
        if (!tourney.containsEdge(player, other)) {
            // see if anyone that this player won against beat this other person
            bool success = false;
            for (Vertex* vanquished : tourney.getNeighbors(player)) {
                if (tourney.getNeighbors(vanquished).contains(other)) {
                    success = true;
                    break;
                }
            }
            if (!success) { return false; }
        }
    }
    return true;
}
```