**Chapter 3-5 – Practice Exercises**

**Exercise 3.3**

1. Table:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Node | A | B | C | D | E | F | G | H |
| pre | 1 | 15 | 2 | 3 | 11 | 4 | 5 | 7 |
| post | 14 | 16 | 13 | 10 | 12 | 9 | 6 | 8 |

1. **Source**: (A, B) and **Sink**: (G, H)
2. B, A, C, E, D, F, H, G
3. Any topological ordering of the graph will be of the form [AB]C[DE]F[GH], with the ordering of the pairs in brackets arbitrary (for example, ABCEDFHG is valid). Each bracketed pair can be organized in 2 different ways, so there are 2 · 2 · 2 = 8 different topological orderings for this graph.

**Exercise 3.4**

1. Answers:
2. {C,D,F,J}, {G,H,I}, {A}, {E}, {B}
3. {D,F,G,H,I}, {C}, {A,B,E}
4. Answers**:**
5. **Source:** {E},{B} and **Sink:** {C,D,F,J}
6. **Source:** {A,B,E} and **Sink:** {D,F,G,H,I}
7. See answer key
8. Answers:
9. Two edges must be added to make the entire graph strongly connected: one from any vertex in {C,D, F, J} to B, and one from any vertex in {C,D, F, J} to E.
10. One edge must be added to make the entire graph strongly connected: from any vertex in {D, F, G,H, I} to any vertex in {A,B,E}.

**Exercise 3.5**

* Initialize empty adjacency list for V.
  + Takes O(n) time.
* For each *u* of V, scan the list of its neighbors.
* For each neighbor *v* of *u* in *G*, add *u* as a neighbor of *v* in the GR adjacency list.
  + Takes O(n + m) time to check each vertex and add each edge to GR

**Exercise 3.15**

**Part A**

* Solve this using the SCC algorithm.
* If graph *G* is a single strongly connected component, the mayor’s claim is TRUE.
* This is because in a SCC, there’s a path from every vertex to every other vertex in the same SCC.
* If the graph has a single SCC then every intersection has a route to every other intersection.
* This would take O(n + m) time.

**Part B**

* This claim requires town hall to lie in a sink SCC.
  + If it lies in a sink SCC *S* then from the town hall, we can reach every other intersection in *S* and from every other intersection in *S* we can get to townhall.
  + If *S* is not a sink SCC, then it has edges coming out, making it possible that there are intersections that can be reached from town hall, but cannot get back to town hall.
* Can also be solved using the SCC algorithm.
* Check if there are edges coming out of the SCC town hall is in by examining the DAG of the metagraph of the SCC.
  + Or check each vertex which lies in the same SCC as the town hall to see if they have outgoing edges to vertices not in town hall’s SCC.
* This would take O(n + m) time for both the SCC run and scanning individual SCCs, making the overall runtime O(n + m).

**Exercise 4.14**

**Algorithm:**

* Run Dijkstra’s algorithm from v0 to all vertices in V.
* Reverse the original graph and run Dijkstra’s algorithm again from v0 to get the distances of the shortest path from all other vertices to v0 in the original graph.
* Find the shortest distance from *u* to *w* using a path through v0 will be the sum of the two distances, *u* to v0 in the reverse graph, and v0 to *w* in the original graph. Any path from *u* to v0 to *w* is recovered via the prev[] arrays from the algorithm.

**Correctness:**

* The graph is strongly connected so a path exists between every pair of vertices. Dijkstra’s algorithm will find the shortest path from a starting vertex to every other vertex. Reversing the directed graph reverses the direction of the path between any pair of vertices.
* So the shortest path form *u* to *w* which includes v0 is the combination of the shortest path from *u* to v0 in the reversed graph plus the shortest path from v0 to *w* in the original graph.

**Runtime:**

* Both runs of Dijkstra’s algorithm takes O((m + n)log n), or O(m log n) since the graph is strongly connected. Building the reverse graph takes O(n + m) time. Calculating the shortest distance for each pair of vertices takes O(n2) time. Making our overall runtime O(n2 + (m log n)).

**Exercise 5.1**

1. **MST:** A,E,F,B,G,C,D,H; **Cost:** 19
2. 2 – A,B,E,F and C,D,G,H
3. Table:

|  |  |
| --- | --- |
| Edge Included | Cut |
| AE | {A} & {B,C,D,E,F,G,H} |
| EF | {A,E} & {B,C,D,F,G,H} |
| BE | {A,E,F} & {B,C,D,G,H} |
| FG | {A,B,E,F} & {C,D,G,H} |
| GH | {A,B,E,F,G} & {C,D,H} |
| CG | {A,B,E,F,G,H} & {C,D} |
| GD | {A,B,C,E,F,G,H} & {D} |

**Exercise 5.2**

1. Table:

|  |  |  |
| --- | --- | --- |
| Vertex Included | Edges Included | Cost |
| A |  | 0 |
| B | AB | 1 |
| C | BC | 3 |
| G | CG | 5 |
| D | GD | 6 |
| F | GF | 7 |
| H | GH | 8 |
| E | AE | 12 |

1. Table:

|  |  |
| --- | --- |
| Union | Pi vertex values |
| Start | [ A, B, C, D, E, F, G, H ] |
| (A,B) | [ B, B, C, D, E, F, G, H ] |
| (F,G) | [ B, B, C, D, E, G, G, H ] |
| (D,G) | [ B, B, C, G, E, G, G, H ] |
| (G,H) | [ B, B, C, G, E, G, G, G ] |
| (C,G) | [ B, B, G, G, E, G, G, G ] |
| (B,C) | [ B, G, G, G, E, G, G, G ] |
| (A,E) | [ G, G, G, G, G, G, G, G ] |

**Exercise 5.9**

1. **False:** Consider a graph where a vertex is adjacent to a single tree.
2. **True:** Consider the order in which edges would be processed by Kruskal’s algorithm; Cycle property from HW5
3. **True:** Minimum weight is a candidate for at least one possible MST
4. **True:** Assured by the cut property