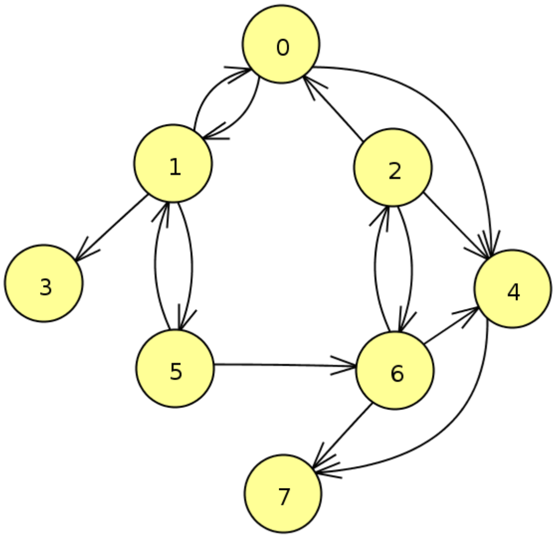
Sabbir Ahmed
Section: 02
HW 6 – Version A
Username: sabbir1

Graphs

1. Complete a Breadth First Search on the graph:

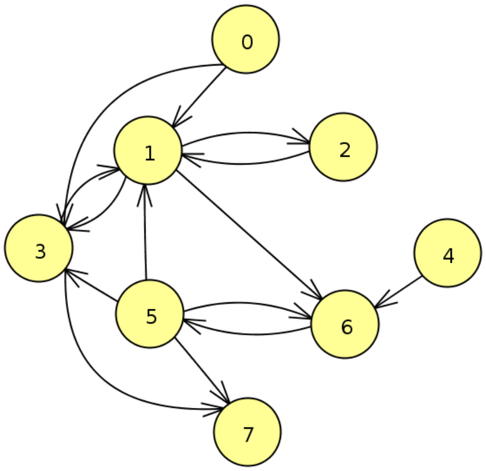


Index	Parent	Visited	BFS Queue
0	2	T	2
1	0	T	0
2	-1	T	4
3	1	T	6
4	2	T	1
5	1	T	7
6	2	T	3
7	4	T	5

Initial:	→ set all values of Visited to false → set Parent as an empty vector → set BFS Queue as an empty queue
Start at 2:	→ enqueue 2 to BFS Queue → assign the 2nd values of Visited to true and Parent to -1 → check neighbors being visited by 2 → find 0, 4, 6 → enqueue the lowest of the neighbors to BFS Queue – 0 → assign the 0th values of Visited to true and Parent to 2 → enqueue the next lowest of the neighbors to BFS Queue – 4 → assign the 4th values of Visited to true and Parent to 2 → enqueue the next lowest of the neighbors to BFS Queue – 6 → assign the 6th values of Visited to true and Parent to 2
Start at 0 (next in BFS Queue):	→ check neighbors being visited by 0 → find 1, 4 → enqueue the lowest of the neighbors to BFS Queue – 1 → assign the 1st values of Visited to true and Parent to 0 → since 4 is already in the BFS Queue and has been visited, ignore it
Start at 4 (next in BFS Queue):	→ check neighbors being visited by 4 → find 7 → enqueue the neighbor to BFS Queue – 7 → assign the 7th values of Visited to true and Parent to 4

Start at 6 (next in BFS Queue):	→ check neighbors being visited by 6 → find 2, 4 → since both 2 and 4 are in the BFS Queue and have been visited, ignore them
Start at 1 (next in BFS Queue):	→ check neighbors being visited by 1 → find 0, 3, 5 → since 4 is already in the BFS Queue and has been visited, ignore it → enqueue the next lowest neighbor to BFS Queue – 3 → assign the 3rd values of Visited to true and Parent to 1 → enqueue the next lowest neighbor to BFS Queue – 5 → assign the 5th values of Visited to true and Parent to 1
Start at 7 (next in BFS Queue):	→ check neighbors being visited by 7 → find 6 → since 4 is already in the BFS Queue and has been visited, ignore it
Start at 3 (next in BFS Queue):	→ check neighbors being visited by 3 → none found, move on
Start at 5 (next in BFS Queue):	→ check neighbors being visited by 5 → find 1, 6 → since both 1 and 6 are in the BFS Queue and have been visited, ignore them

2. Complete a Depth First Search on this graph:

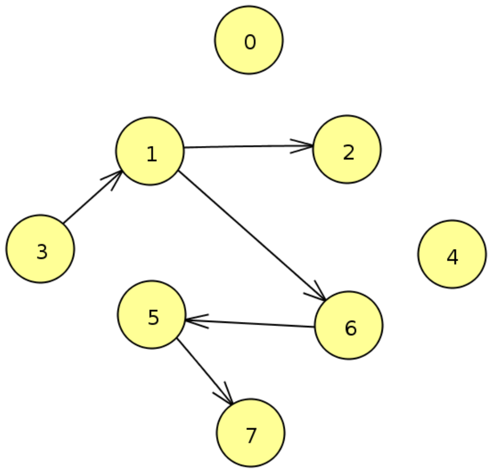


Index	Parent	Visited	DFS Stack
0		F	DFS(3)
1	3	T	DFS(1)
2	1	T	DFS(2)
3	-1	T	DFS(6)
4		F	DFS(5)
5	6	T	DFS(7)
6	1	T	
7	5	T	

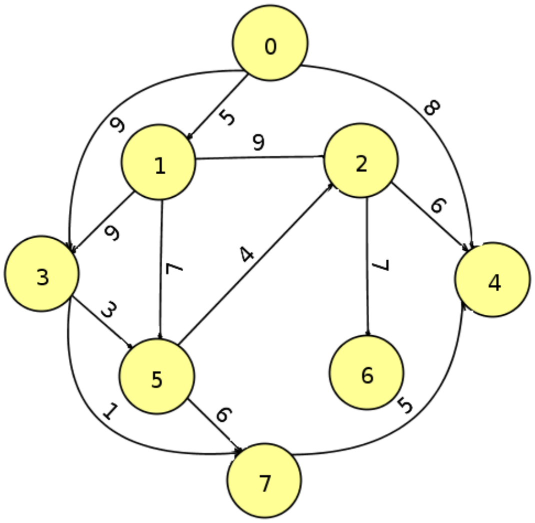
Initial:	→ set all values of Visited to false → set Parent as an empty vector → set DFS Stack as an empty stack
Start at 3:	→ add 3 to the DFS Stack → assign the 3rd values of Visited to true and Parent to -1 → check neighbors being visited by 3 → find 1, 7 → traverse to the lower of the neighbors - 1
Start at 1:	→ add 1 to the DFS Stack with an indent → assign the 1st values of Visited to true and Parent to 3

	→ check neighbors being visited by 1 → find 2, 3, 6 → traverse to the lower of the neighbors - 2
Start at 2:	→ add 2 to the DFS Stack with two indents → assign the 2nd values of Visited to true and Parent to 1 → check neighbors being visited by 2 → found 1 → since 1 has already been visited, search through 2 dies and recurses back to its parent at 1
Start at 1:	→ find the other neighbors 3, 6 → since 3 has already been visited, skip it → traverse to the next neighbor – 6
Start at 6:	→ add 6 to the DFS Stack with two indents → assign the 6th values of Visited to true and Parent to 1 → check neighbors being visited by 6 → found 5 → traverse to 5
Start at 5:	→ add 5 to the DFS Stack with three indents → assign the 5th values of Visited to true and Parent to 6 → check neighbors being visited by 5 → found 1, 3, 7 → since 1 and 3 are visited, skip and traverse to 7
Start at 7:	→ add 7 to the DFS Stack with four indents → assign the 7th values of Visited to true and Parent to 5 → check neighbors being visited by 7 → none found → search through 7 dies, recurse back to its parent at 5
Start at 5:	→ since all its neighbors have been visited, recurse to its parent at 6
Start at 6:	→ since all its neighbors have been visited, recurse to its parent at 1
Start at 1:	→ since all its neighbors have been visited, recurse to its parent at 3
Start at 3	→ depth first search on the graph terminates

Final connected graph after a depth first search:



3. Complete a Dijkstra's Shortest Path algorithm on this graph:



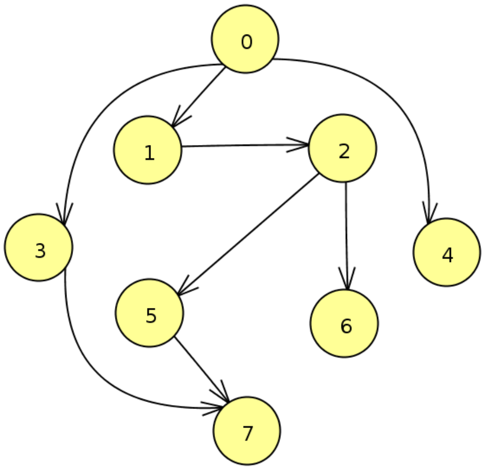
Vertex	Known	Cost	Parent	Path
0	T	5	1	1 0
1	T	0	-1	1
2	T	9	1	1 2
3	T	9	1	1 3
4	T	13	0	1 0 4
5	T	7	1	1 5
6	T	16	2	1 2 6
7	T	10	3	1 3 7

Initial:	→ set all values of Known to false and Path to -1 → set Cost as an empty vector
Start at 1:	→ assign the 1st values of Known to true and Cost to 0 → check neighbors being visited by 1 → find 0, 2, 3, 5 → traverse to the lower of the neighbors - 0
Start at 0:	→ assign the 0th values of Cost to 5 and Path to 1 → traverse to the next lower of the neighbors of 1 - 2
Start at 2:	→ assign the 2nd values of Cost to 9 and Path to 1 → traverse to the next lower of the neighbors of 1 - 3
Start at 3:	→ assign the 3rd values of Cost to 9 and Path to 1 → traverse to the last of the neighbors of 1 - 6
Start at 5:	→ assign the 5th values of Cost to 7 and Path to 1 → traversal of the neighbors of 1 is done
Start at 1:	→ find the least cost of the neighbors → find 0 with a cost of 5 → traverse back to 0
Start at 0:	→ assign the 0th value of Known to true → check neighbors being visited by 0 → find 1, 3, 4 → since 1 has already been visited and its cost 0 is smaller than the new cost of 5, don't update and move on → since 3 has a cost of 9 which is smaller than the new cost of 5 + 9, don't update and move on → traverse to the last of its neighbors - 4
Start at 4:	→ assign the 4th values of Cost to 8 + 5 and Path to 0 → traversal of the neighbors of 0 is done
Start at 1:	→ find the least cost of the neighbors → find 5 with a cost of 7

	→ traverse back to 5
Start at 5:	→ assign the 5th value of Known to true → check neighbors being visited by 5 → find 1, 2, 3, 7 → since 1 has already been visited and its cost 0 is smaller than the new cost of 7, don't update and move on → since 2 has a cost of 9 which is smaller than the new cost of 9 + 7, don't update and move on → since 3 has a cost of 9 which is smaller than the new cost of 9 + 3, don't update and move on → traverse to the last of its neighbors - 7
Start at 7	→ assign the 7th values of Cost to 7 + 6 and Path to 5 → traversal of the neighbors of 5 is done
Start at 1:	→ find the least cost of the neighbors → find 2 and 3 with costs of 9 → traverse back to 2 because of the value of its key being lower
Start at 2:	→ assign the 2nd value of Known to true → check neighbors being visited by 2 → find 1, 4, 5, 6 → since 1 has already been visited and its cost 0 is smaller than the new cost of 9, don't update and move on → since 4 has a cost of 13 which is smaller than the new cost of 9 + 6, don't update and move on → since 5 has a cost of 7 which is smaller than the new cost of 9 + 4, don't update and move on → traverse to the last of its neighbors – 6
Start at 6	→ assign the 6th values of Cost to 9 + 7 and Path to 2 → traversal of the neighbors of 2 is done
Start at 3:	→ assign the 3rd value of Known to true → check neighbors being visited by 3 → find 0, 1, 5, 7 → since 0 has already been visited and its cost 5 is smaller than the new cost of 9 + 5, don't update and move on → since 1 has already been visited and its cost 0 is smaller than the new cost of 9, don't update and move on → since 5 has a cost of 7 which is smaller than the new cost of 9 + 3, don't update and move on → since 7 has a cost of 13 which is greater than the new cost of 9 + 1, update its cost to 9 + 1 and Parent to 3 → traversal of the neighbors of 3 is done
Start at 7	→ since the next lowest cost is assigned to 7 with a 10 → assign the 7th value of Known to true → check neighbors being visited by 7 → find 3, 4, 5 → since 3 has already been visited and its cost 9 is smaller than the new cost of 1 + 9, don't update and move on

	→ since 4 has a cost of 13 is smaller than the new cost of 10 + 13, don’t update and move on → since 5 has already been visited and its cost 7 is smaller than the new cost of 10 + 7, don’t update and move on → traversal of the neighbors of 4 is done
Start at 4:	→ since the next lowest cost is assigned to 4 with a 13 → assign the 4th value of Known to true → check neighbors being visited by 4 → find 0, 2, 7 → since 0 has already been visited and its cost 5 is smaller than the new cost of 8 + 5, don’t update and move on → since 2 has already been visited and its cost 9 is smaller than the new cost of 6 + 9, don’t update and move on → since 7 has a cost of 10 which is smaller than the new cost of 5 + 13, don’t update and move on → traversal of the neighbors of 4 is done
Start at 6:	→ since the last of the nodes → assign the 6th value of Known to true → since the only neighbor of 6 is 2 which has been visited, its cost cannot be updated → Dijkstra’s Shortest Path algorithm terminates

4. Complete a Topological sort on this graph:

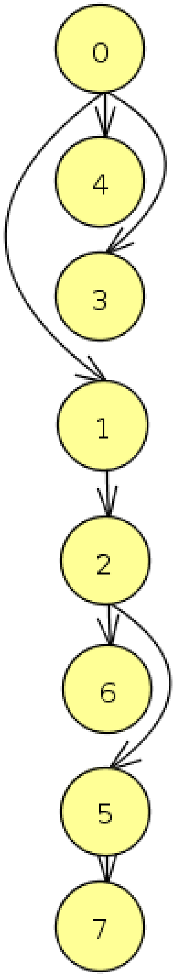


Vertex	Path	Topological order
0	DFS(0)	7
1	DFS(1)	5
2	DFS(2)	6
3	DFS(5)	2
4	DFS(7)	1
5	DFS(6)	3
6	DFS(3)	4
7	DFS(4)	0

Initial:	→ set Path and Topological order as empty vectors → topological sort begins with the vertex with the lowest value
Start at 0:	→ assign the 0th value of Path to DFS(0) → start depth first search from 0 → look for neighbors visited by 0 → find 1, 3, 4 → traverse to the vertex with the lowest value – 1
Start at 1:	→ assign the 1st value of Path to DFS(1) with an indent → look for neighbors visited by 1

	→ find 2 → traverse to 2
Start at 2:	→ assign the 1st value of Path to DFS(2) with two indents → look for neighbors visited by 2 → find 5, 6 → traverse to the vertex with the lowest value – 5
Start at 5	→ assign the 5th value of Path to DFS(5) with three indents → look for neighbors visited by 5 → find 7 → traverse to 7
Start at 7	→ assign the 7th value of Path to DFS(7) with four indents → look for neighbors visited by 7 → none found → add 7 to Topological order → recurse back to its parent at 5
Start at 5	→ look for other neighbors visited by 5 → none found → add 5 to Topological order → recurse back to its parent at 2
Start at 2	→ look for other neighbors visited by 5 → found 6 → traverse to 6
Start at 6	→ assign the 6th value of Path to DFS(6) with three indents → look for neighbors visited by 6 → none found → add 6 to Topological order → recurse back to its parent at 2
Start at 2	→ look for other neighbors visited by 2 → none found → add 2 to Topological order → recurse back to its parent at 1
Start at 1	→ look for other neighbors visited by 1 → none found → add 1 to Topological order → recurse back to its parent at 0
Start at 0	→ look for other neighbors visited by 0 → found 3, 4 → traverse to the vertex with the lowest value – 3
Start at 3	→ assign the 3rd value of Path to DFS(3) with an indent → look for neighbors visited by 3 → found 7 → since 7 has already been visited and added, move on → add 3 to Topological order → recurse back to its parent at 0
Start at 0	→ look for other neighbors visited by 0 → found 4 → traverse to its last neighbor - 4

Start at 4	→ assign the 4th value of Path to DFS(4) with an indent → look for neighbors visited by 4 → none found → add 4 to Topological order → recurse back to its parent at 0
Start at 0	→ no other neighbors left → add 0 to Topological order
Topological order vector	→ vertices currently are in order: 7, 5, 6, 2, 1, 3, 4, 0 → flip the vector so the vertices are: 0, 4, 3, 1, 2, 6, 5, 7 → reconnect the vertices to verify topological order of the vertices



Final topological sorted graph