**Graphs**

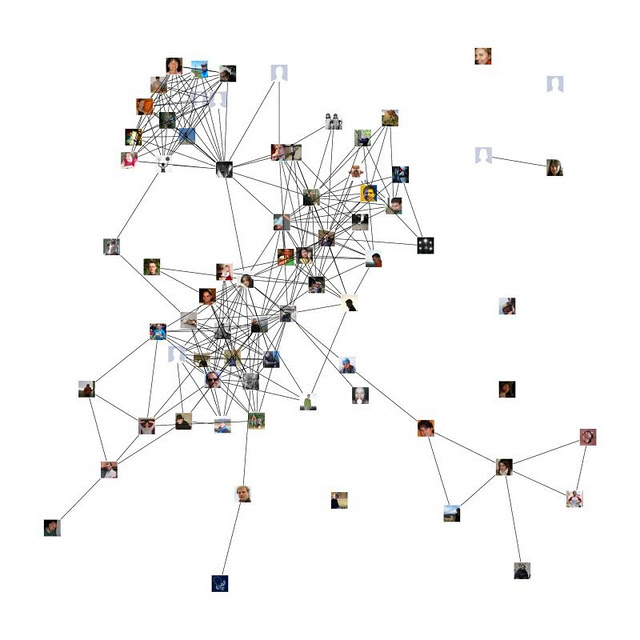
**1.0 Introduction to Graphs**

* The plotting of data on X-Y axis is known as Charts in CS not Graphs
* Graphs in CS are a combination of vertices that are connected to each other. Unlike Trees nodes in Graphs do not have a top and bottom
* Traversal of graph: is the movements or path required to move within a given graph
* A visial example of a directed graph is presented below:

Diagram

Description automatically generated

* Graphs are more realistic than Trees or Heaps in the sense of usage. A great example of Graphs is the “friends” layout of Facebook profiles, Look at the image below:
  + Each person represents a vertex and Each person connected to other people
  + The below graph is an Undirected graph



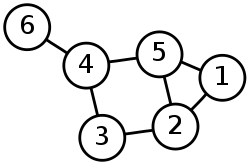
**2.0 Types of Graphs**

* There are 3 types of graphs in CS. Directed, Undirected, and Weighted (Directed or Undirected)
* **The Directed Graph** 
  + This is a graph where there is direction of orders in it. The edges of these graphs have arrows representing the direction of movements
  + Every edge has the same exact weight. Therefore, the optimum path is the path with least number of steps
  + Below is a representation of a directed graph
  + The traversal of the graph below is:
    - 1. Start at A
    - 2. Go to B
    - 3. Go to C
    - 4. Go to E
    - 5. Go to D or F
      * If went to D Then we Can go to B
      * If we went to F then we end the graph path
    - The transversality of this graph dictates only a specific path must be followed. We **cannot** go from F back to E back to C back to B and back to A

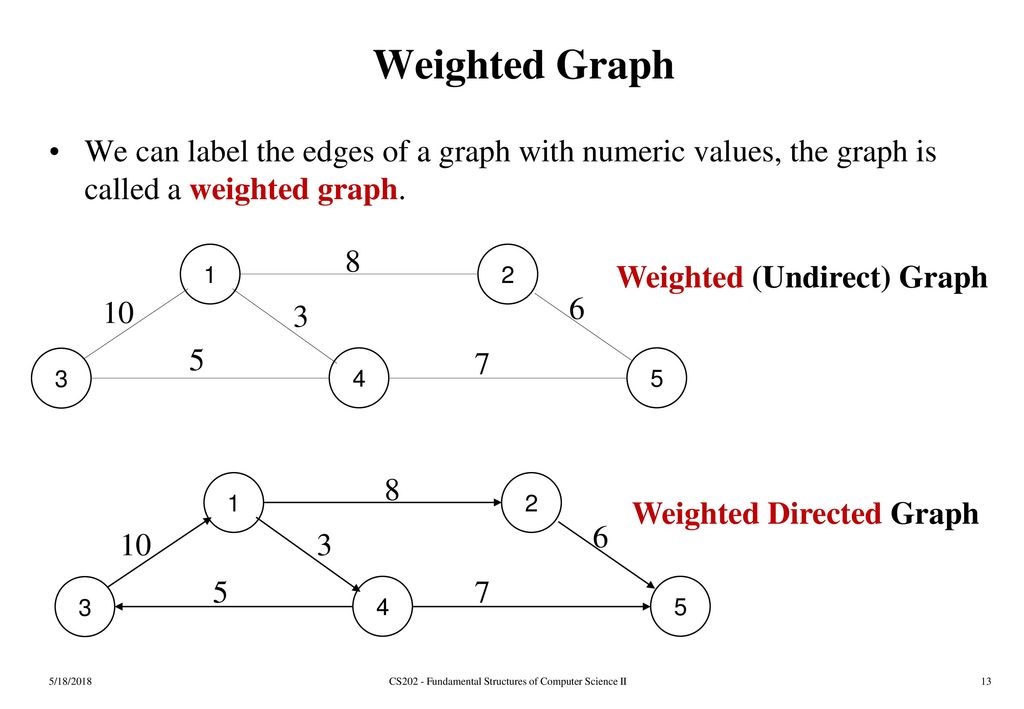
Diagram

Description automatically generated

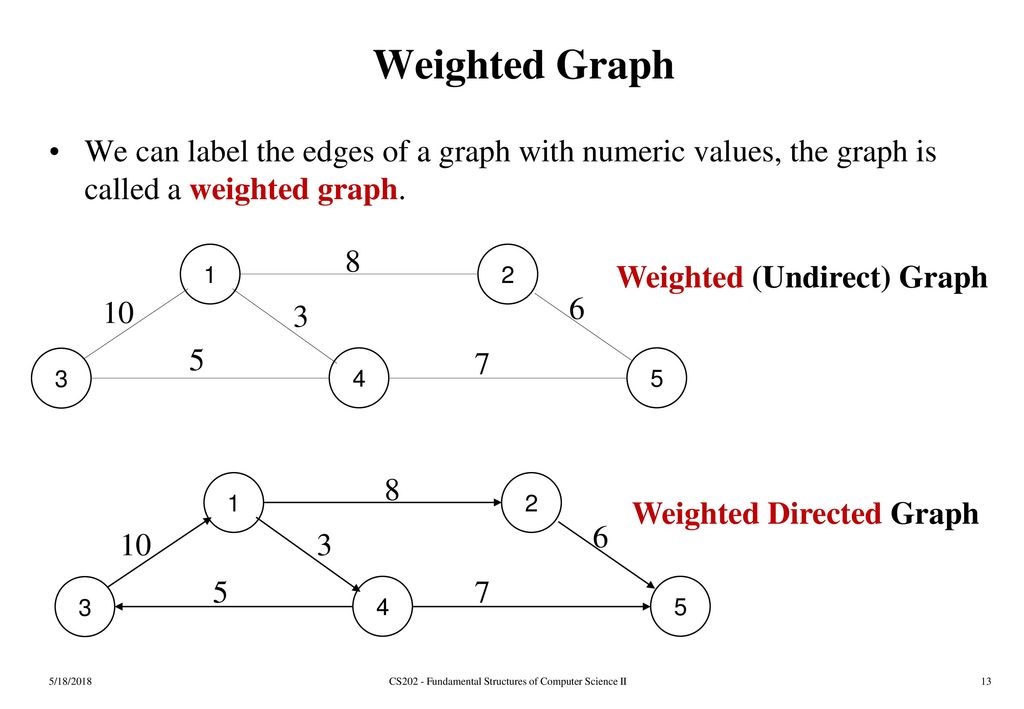
* **The Undirected Graph** 
  + This is a graph where there is no direction of orders in it. The edges of these graphs have no arrows . Therefore, the movements between them are much freer
  + Every edge has the same exact weight. Therefore, the optimum path is the path with least number of steps
  + Below is an example of using an undirected graph
  + The traversal of the graph below is:
    - 1. You can start from anywhere. Say you start from 6
    - 2. You only can go to 4 from 6, because there is a single connection between them
    - 3. You can go to either 5 or 3
      * If you decide to go to 3 then you can go to 2 which can go to 5 or 1
      * If you decide to 5 then you can go to 1 or 2
    - 4. There is no specific direction. Therefore, the transversality of this graph allows to go backwards from 1 to 5 to4 to 6 or any other direction as long they are linked by the edges



* **The Weighted Graph** 
  + This is a graph means that the layout of the graph has different weights
  + The weighted graph may be directed or undirected
  + The weighted graph helps while traversing through a graph to decide on which route to take. You tend to take the route with least amount of weight. The weight represents the resources
  + Every edge has a weight unique to itself. Therefore, the optimum path is not always the path with least number of steps
  + The weight is displayed on the edges that connected the data points (vertices)
* **The Weighted Graph (Directed)**
  + In addition to having specific directions to move within the graph those directions are weighted
  + For Example: To move from 1 to 5. There are two paths
    - The first path is: 1 -> 2 -> 5
      * The weight of this path is 8 + 6 = 14
    - The second path is: 1+4+5
      * The wight of this path is 3 + 7 = 10
    - The second path weights less. Therefore, less resources would be used to get the data of 5 from 1

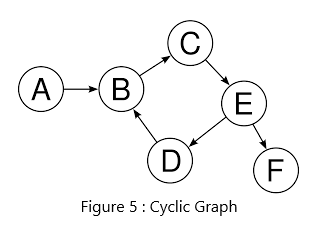


* **The Weighted Graph (Undirected)**
  + The graph have only edges connecting them without any direction expect weights on each edge
  + Also, we can go in any direction we want because this is an Undirected graph
  + For Example: To move from 1 to 3. There are many paths. However, the three easiest to visualize are:
    - The first path is: 1 -> 3
      * The weight of this path is 10
    - The second path is: 1 -> 2 -> 5 -> 4 -> 3
      * The wight of this path is 8+6+7+5 = 26
    - The third path is: 1 -> 4 -> 3
      * The wight of this path is 3 +5 = 8
    - The third path weights less. Therefore, less resources would be used to get the data of 5 from 3. Even though the third path requires an additional step it weights less

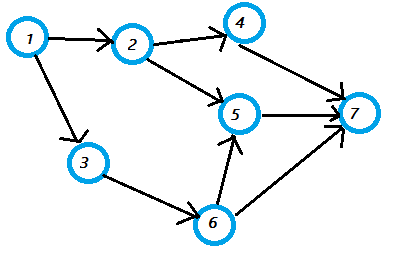


**3.0 Some of The Terminologies in CS Graph Theory**

* The **Vertex**: A **vertex** is a single data point in the graph. Do not call it node. In graph theory these single data points are known as **vertices**
  + The plural of **Vertex** is **Vertices**
* The **Set**: an unordered linear representation of all the **vertices** of a graph. It holds every **vertex** of a graph without a specific order. Sets are of curly brackets
* An **Edge** : Is the line that connects two **vertices**. You must call it an edge instead of chain or link
* **Edge Set**: It is an unordered combination of **Edges**. There is no order for them. It is purpose is to view the relationships of each two vertices in the graph. A group of items that have no importance or order to them. So {A,B,C} is equivalent to {B,C,A} and {C,B,A} etc. The order doesn't matter. They are just a group of items that share something in common.
* **Adjacency**: The idea of two **vertices** connected to each other via **a single edge**
* **Cyclic** **Graph**: A type of a graph that must be a directed graph and have a cycle in it**. A cyclic graph** is a graph containing at least one graph cycle.
  + The cycle represents the possibility of being stuck in an infinite loop. Therefore, Cyclic graphs are better avoided. Thus, Acyclic graph are preferred to use in CS
  + The figure below represents a cycle in a graph
    - The graph has a cycle between C->E->D->B



* **Acyclic Graph**: A type of a graph that must be a directed graph and can not a cycle in it. **Acyclic graph** can not contain any graph cycle.
  + The cycle represents the possibility of being stuck in an infinite loop. Therefore, Acyclic graph are preferred to use in CS
  + The figure below represents **Acyclic graph**



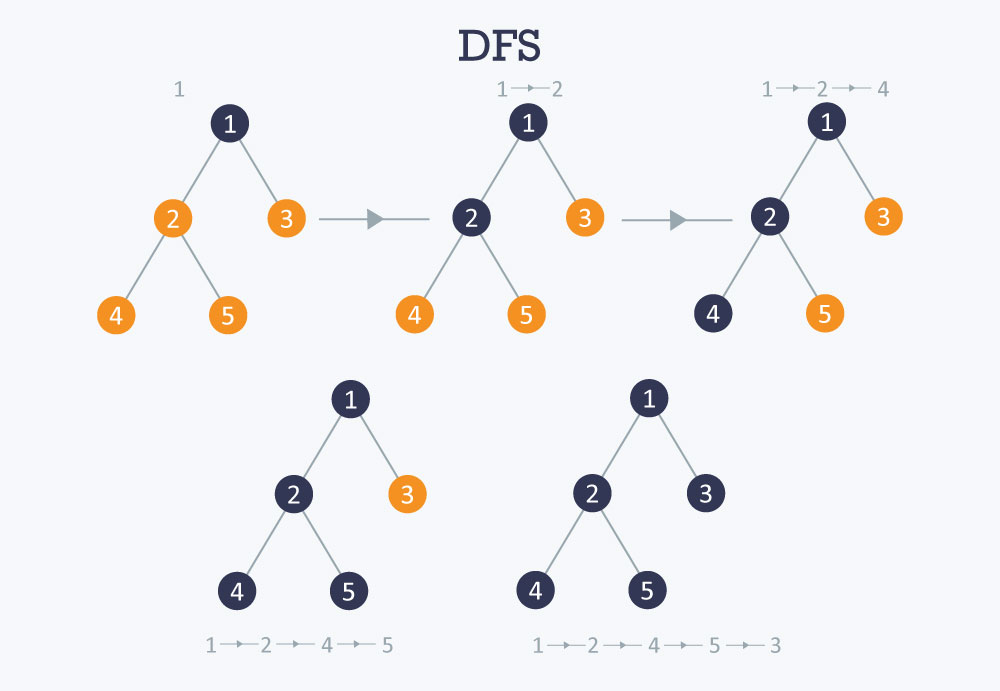
* **Disconnected Graph**: Is a graph that has vertices that are not connected to any other vertices
  + The figure below represents **a disconnected graph**. This may be looked at as two groups of friends on Facebook that have no common friends yet

Diagram

Description automatically generated

**4.0 Graphs Applications – Depth First Search**

* Depth-first search is an algorithm for traversing or searching tree or graph data structures. It is a way to find every vertex within the graph in an efficient manner. It can
  + Avoid cycles
  + Avoids repeating vertices (When we repeat different combinations of the same node)
* Using a stack (DFS) or a queue (BFS) made sure we can visit every single vertex and not get stuck in a cycle
* The way DFS work: imagine the graph you have is a maze and you want to find the way exit that maze
* The general steps are:
  + The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
  + The basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node
  + Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.
* Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a Boolean visited array.
* In the example below: We were able to use DFS to visit every vertex only once, and avoid cycles (there was no cycles there). Then the results of [1,2,4,3,5] represent the resultant stack of order of visitation to satisfy the DFS conditions of visiting each node only once. You do not necessarily need to start from node 1.
* Steps:
  + 1. Start by selecting any vertex. We selected the first vertex. **The list of moving around the graph (current stack)** = [1] & **The output list** = [1]
  + 2. After adding the vertex you selected to your stack. Search for all unselected vertices that are adjacent (connected by an edge) to the first vertex you selected. In our case we could have selected either vertices 2 or 3. But we selected 2. There is no importance of which vertex you select. **The list of moving around the graph (current stack)** = [1,2] & **The output list** = [1,2]
  + 3. We keep going down and selected the unselected vertices that are connected to the last vertex we select in this case we selected vertex 4 after selecting 2. **The list of moving around the graph (current stack)** = [1,2,4] & **The output list** = [1,2,4]
  + 5. After adding the vertex 4. We reached the special case of having 4 not connected to any other unselect vertex that is not connected to it via an edge. Therefore, we pop it back from **the list of moving around the graph (current)** but not **The output list**. **The list of moving around the graph (current)** = [1,2] & **The output list** = [1,2,4]
  + 6. Then we look back at the vertex before the vertex that we just removed. Then we look for any other untouched vertices and add them. In our case the vertex 5 is connected to the vertex 2. **The list of moving around the graph (current stack)** = [1,2,5] & **The output list** = [1,2,4,5].
  + 7. Again, we reached the special case of having the vertex 5 not connected via to any other vertex that has yet to be selected. Therefore, we pop it back. Remove the vertex 5 **The list of moving around the graph (current)** but not **the output list**. **The list of moving around the** -> **The list of moving around the graph (current)** = [1,2] & **The output list** = [1,2,4,5].
  + 8. Now we have reached a special case again, because vertex 2 is not connected to any unselected vertex via an edge . Therefore, we pop it from **the list of moving around the graph (current stack)** but not **the output list**. **The list of moving around the** -> **The list of moving around the graph (current stack)** = [1] & **The output list** = [1,2,4,5].
  + 9. Now we are at the vertex 1 and we see that it is connected to the unselected vertex 3 via an edge therefore we go there and add it. **The list of moving around the graph (current stack)** = [1,3] & **The output list** = [1,2,4,5,3].
* **The output list** = [1,2,4,5,3]. Is the output of running the DFS algorithm on the graph. The DFS was able to find every single vertices in an efficient matter
* Instead of using a queue like BFS. In DFS we use a stack, which is the main difference between them. Therefore, we are always working with most recent item added to the stack



**5.0 Graphs Applications – Breadth First Search BFS**

* Breadth-first search is an algorithm for traversing or searching tree or graph data structures. It is a way to find every vertex within the graph in an efficient manner. It can avoid getting stuck in graph cycles.
* Using a stack (DFS) or a queue (BFS) made sure we can visit every single vertex and not get stuck in a cycle
* BFS is an **exhausted** search process, where we start from one location and try to capture every vertex as we go through the graph without popping
* Instead of using a stack like DFS. In BFS we use a queue, which is the main difference between them. Therefore, we are always working with oldest item in the queue
* The example below covers step by step the implementation of BFS on a graph.
  + This algorithm continues going until there is not more vertices to be added to the queue and the output list is empty

|  |  |
| --- | --- |
| 1. Start with laying out the full graph layout with an empty queue | bfs1 |
| 2. Select any vertex  -In our case we selected vertex 1  -Then, add it to the visited list and to the queue  - visited list = [1]  - queue = [1] | bfs2 |
| 3. Pop oldest vertex from the queue and add it into the output/print list  - visited list = [1]  - queue = []  - output/print = [1] | bfs3 |
| 4. Add all the untouched vertices that are adjacent to the (most recently added vertex to the output/print from the queue) to the queue. Then, add them to the visited list as well.  - visited list = [1,2,3]  - queue = [2,3]  - output/print = [1] |  |
| 5. Pop the last added vertex to queue from the queue and add it into the output/print list. Also, by doing so remove it from the queue  - Then, add all the untouched vertices that are adjacent to the (most recently added vertex to the output/print from the queue) to the queue. Then, add them to the visited list as well.  - visited list = [1,2,3,4,5]  - queue = [3,4,5]  - output/print = [1,2] |  |
| 6. Pop the last added vertex to queue from the queue and add it into the output/print list. Also, by doing so remove it from the queue  - Then, add all the untouched vertices that are adjacent to the (most recently added vertex to the output/print from the queue) to the queue. Then, add them to the visited list as well.  - visited list = [1,2,3,4,5]  - queue = [4,5]  - output/print = [1,2,3]  7. Add all the untouched vertices that are adjacent to the most recently added vertex to the output/print from the queue. Then, add them to the visited list as well. In this case the vertex 3 was the most recently added vertex from the queue to the output/print list. However, it is connected to only vertex 5. Which is already has been added to the queue. Therefore, continue on adjusting the queue |  |
| 8. Pop the last added vertex to queue from the queue and add it into the output/print list. Also, by doing so remove it from the queue  - Then, add all the untouched vertices that are adjacent to the most recently added vertex to the output/print from the queue. Then, add them to the visited list as well  - visited list = [1,2,3,4,5]  - queue = [5]  - output/print = [1,2,3,4] | bfs8 |
| 9. Add all the untouched vertices that are adjacent to the (most recently added vertex to the output/print from the queue) to the queue. Then, add them to the visited list as well.  - visited list = [1,2,3,4,5,6]  - queue = [5,6]  - output/print = [1,2,3,4] |  |
| 10. Pop the last added vertex to queue from the queue and add it into the output/print list. Also, by doing so remove it from the queue  - Then, add all the untouched vertices that are adjacent to the most recently added vertex to the output/print from the queue. Then, add them to the visited list as well -> in this case there was no more vertices to add  - visited list = [1,2,3,4,5,6]  - queue = [6]  - output/print = [1,2,3,4,5] |  |
| 11. Pop the last added vertex to queue from the queue and add it into the output/print list. Also, by doing so remove it from the queue  - Then, add all the untouched vertices that are adjacent to the most recently added vertex to the output/print from the queue. Then, add them to the visited list as well -> in this case there was no more vertices  - visited list = [1,2,3,4,5,6]  - queue = []  - output/print = [1,2,3,4,5,6] |  |

**6.0 Graphs Traversal Algorithms Run Time – First Depth Search FDS and Breadth First Search BFS**

* The run time of the algorithms is O(V+E). V stands for vertex and E stands for edge.
* The run time for both algorithms is O(V+E). is because to fully cover any graph using any of these algorithms, we must go over every single vertex and every edge that connects any two vertices
* We can not have O(n). because each vertex can have different number of edges. Therefore, there is no solid relationship between the number of vertices and the number of edges each vertex has