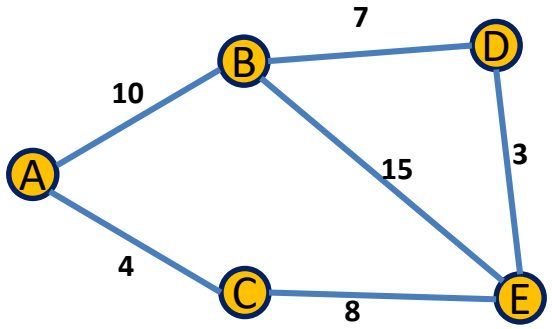
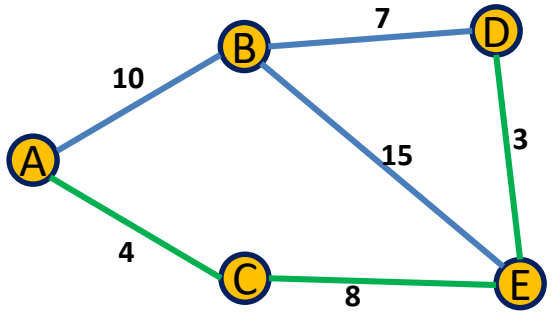
Why Graphs?

For example, given a network of cities and roads connecting them, what is the shortest path between two given cities?





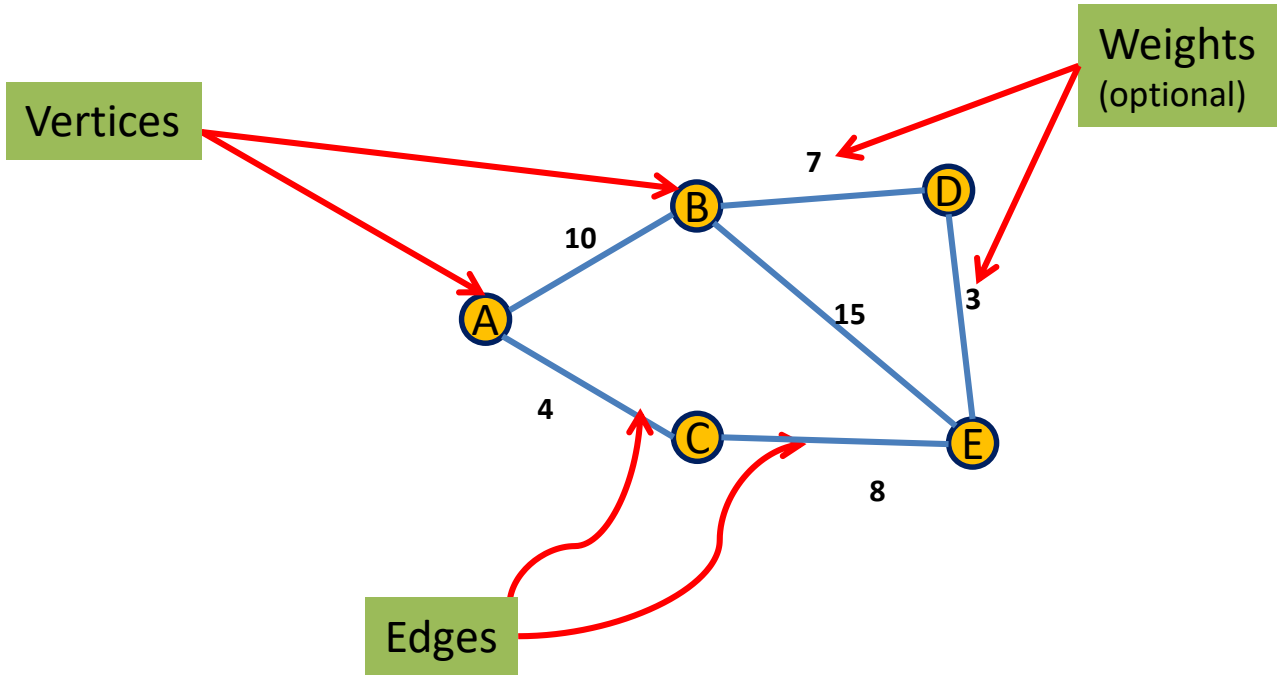
Shortest path between A and D?

This is a very natural way to model and think of the given data.

Today, we'll look at this in detail and formalize some notions about this data structure!

Graphs

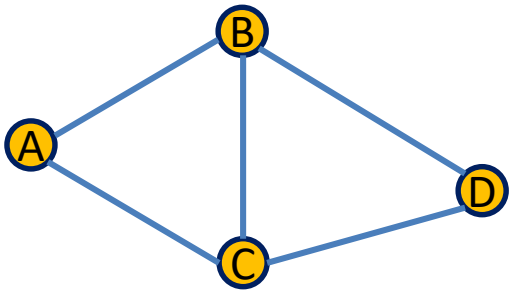
A ***graph*** *G* is defined as *G = (V, E).*
Here, *V* is a set of vertices and *E* is a set of edges. Edges may or may not have weights.



And edge connects two vertices and can be denoted by it's two endpoints, e.g., (A,C). We have, $E \subseteq V \times V$.

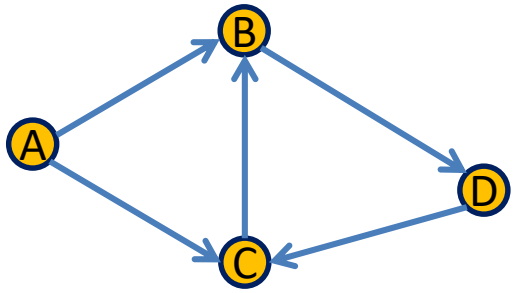
Classifications of graphs

Undirected Graph



V = {A, B, C, D}
E = { (A, B), (A,C),
(B,D), (B,C), (C, D) }

Directed Graph



V = {A, B, C, D}
E = { (A, B), (A,C),
(B,D), (C,B), (D, C) }

Note: E is a set. Therefore, in an undirected graph, we write only one of (A,C) and (C, A) since they are the same edge. This is not the case for a directed graph.

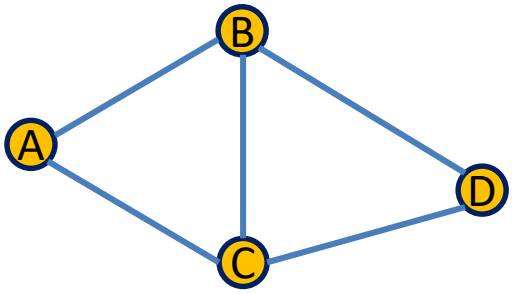
In general, edge sets cannot tell you whether the graph is directed or undirected. Rather, knowing about the graph tells you how to interpret the edge set.

Graph Terminologies

- ***Walk***:
  A walk from *x* to *y* is a sequence of vertices <x, v_0, v_1, …, v_k, y> such that
  - *(x, v_0) ∈ E*
  - (v_k, y) ∈ *E*
  - For all *i = 0 to k-1, (v_i, v_{i+1}) ∈ E*

- ***Path***:
  A path is a walk <x, v_0, v_1, …, v_k, y> where no vertex gets repeated.

- ***Cycle***:
  A cycle is a walk <x, v_0, v_1, …, v_k, y> where no intermediate vertex gets repeated and y=x



Examples:
<A, B, C, D, B, A, C> is a walk.

<A, B, D> is a path.

<A, B, D, C, A> is a cycle.

<B, C, D, A> is **not** a walk. Why?

Representations of Graphs

Suppose there are n vertices in the graph. This gives us the vertex set {1, …, n}. Now we need to store the edge set. Here are two ways of doing this.

Adjacency Matrix representation

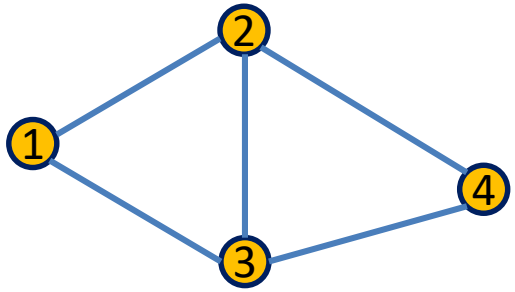
- We store an $n \times n$ size 2D array A (a matrix, essentially).
- $A_{i,j} = 1$ *iff* there is an edge between i and j in the graph.
- Size of the representation is $O(n^2)$.

Adjacency List representation

- We store a linked list for every vertex.
- The linked list for vertex i consists of the neighbours of i in the graph.
- Size of the representation is $O(n + m)$ where m is the number of edges in the graph.

Let us look at examples of these now.

Representations of Graphs

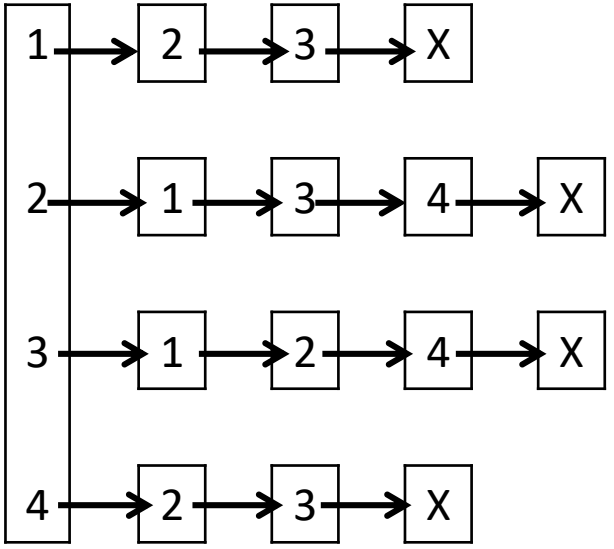


n = 4, m = 5

Adjacency matrix:

	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	1
4	0	1	1	0

Adjacency list:



Implementations of Graph Data Structures

Adjacency matrix: Use a *n* x *n* matrix. $A_{i, j}$ = weight of edge from node *i* to node *j*.

Adjacency list: We can use vectors from C++ STL!

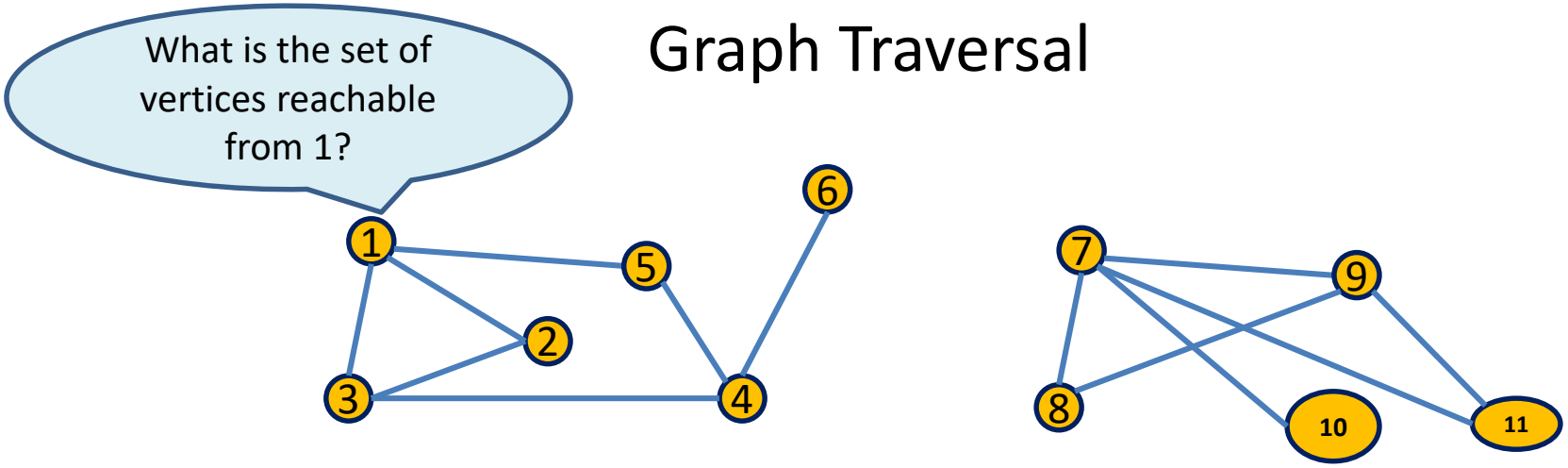
```
#include <bits/stdc++.h>
using namespace std;

// Define the adjacency list as an array of vectors!
vector<int> adj[n];

// Function to add an edge between nodes u and v in an undirected
graph.
void addEdge(vector<int> adj[], int u, int v)
{
   // Add v to adjacency list of u
   adj[u].push_back(v);
   // Add u to adjacency list of v
   adj[v].push_back(u);
}
```

There are also other implementations of Adjacency Lists.
An array of linked lists is a basic implementation, but it requires careful manipulation of pointers and good knowledge of the workings linked lists and structures.

Graph Traversal



A vertex ***y*** is said to be reachable from vertex ***x*** if there exists a path from ***x*** to ***y***.

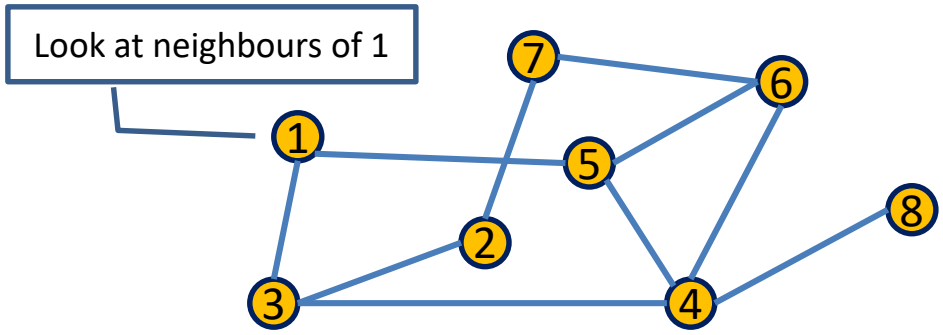
Is there a path between 7 and 11? No!

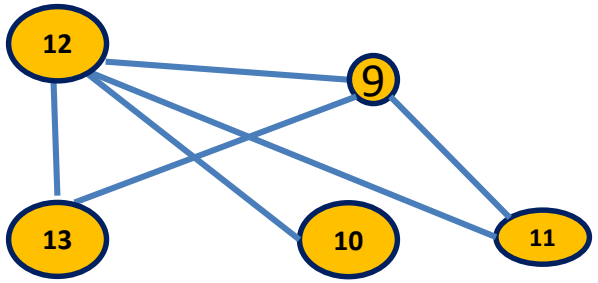
Question: Given a vertex ***x***, can we visit all vertices reachable from ***x***?

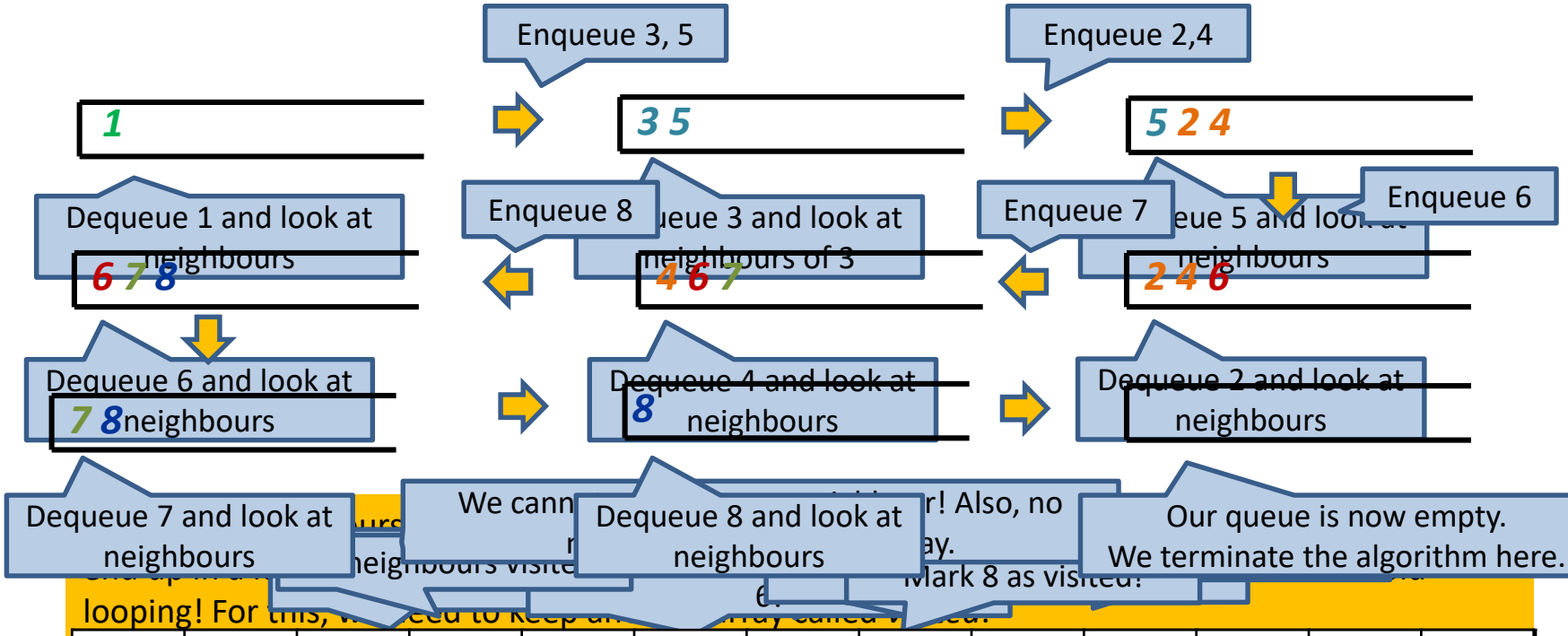
Graph Traversal at ***x***: Visit all vertices reachable from ***x***.

How do we perform graph traversal?

Breadth First Search – A Graph Traversal Algorithm







1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	1	1	1	1	1	1	0	0	0	0	0

Details on Breadth First Search

- BFS from *1* visited *all vertices* reachable from *1*. Also, it visited *only reachable vertices* from *1*.
- Before enqueing, we check if the vertex is visited. And after we enqueue, we mark it visited. So a vertex enters the queue at most once. This is essential to ensure termination.
- We look at all and only unvisited neighbours while processing a vertex.
- BFS from *1* visits vertices in increasing order of ***distance.***

Distance between ***x*** and ***y*** is the number of edges in the shortest path from ***x*** to ***y***.

```
BFS(G, x)              //BFS on graph G, starting at vertex
  Create empty queue Q.
  Create array visited of length n=|V| and initialize to 0.
  Enqueue(x,Q);        visited[x] = 1;        //BFS on graph G, starting at vertex x.
  while( Q is not empty)
  {
        v ←DeQueue(Q);
        For each neighbor w of v
        {
              if(visited[w] == 0)
              {       Enqueue(w,Q);
                      visited[w] = 1;
              }
        }
  }
```

Breadth First Search - Implementation

```
// Define the visited array initialized to 0.
vector<int> visited(n, 0);
deque<int> Q;

// Function to perform BFS starting from x
void BFS(int x)
{
        // Enqueue x and mark it visited!
        Q.push_back(x);
        visited[x]=1;
        while(!(Q.empty())) //While the queue is not empty
        {
                // Dequeue operation.
                int v = Q.pop_front();
                num_v = adj[v].size();
                for(int i=0; i<num_v; i++)
                {
                        int w = adj[v][i];
                        if(visited[w] == 0)
                        {
                                Q.push_back(w);
                                visited[w] = 1;
                        }
                }
        }
}
```

*Check if **w** is visited.*

*Enqueue **w** and mark it visited.*

*After **BFS(x)** terminates, for all vertices **y**, visited[**y**] = 1 iff **y** is reachable from **x**.*

Breadth First Search - Applications

- Computing Distances: Given a source vertex ***x***, compute the distance of all vertices from ***x***.

- Checking for cycles in a graph: Given an undirected graph G, report whether there exists a cycle in the graph or not. (Note: won't work for directed graphs)

- Checking for bipartite graph: Given a graph, check whether it is bipartite or not? A graph is said to be bipartite if there is a partition of the vertex set V into two sets V_1 and V_2 such that if two vertices are adjacent, either both are in V_1 or both are in V_2.

- Reachability: Given a graph G and vertices ***x*** and ***y***, determine if there exists a path from ***x*** to ***y***.

- For more applications, visit http://www.geeksforgeeks.org/applications-of-breadth-first-traversal/

Breadth First Search – Time and Space Complexity

```
BFS(G, x)              //BFS on graph G, starting at vertex x.
  Create empty queue Q.
  Create array visited of length n=|V| and initialize to 0.
  Enqueue(x,Q);    visited[x] = 1;        //BFS on graph G, starting at vertex x.
  while( Q is not empty)
  {
        v ←DeQueue(Q);
        For each neighbor w of v
        {
              if(visited[w] == 0)
              {       Enqueue(w,Q);
                      visited[w] = 1;
              }
        }
  }
```

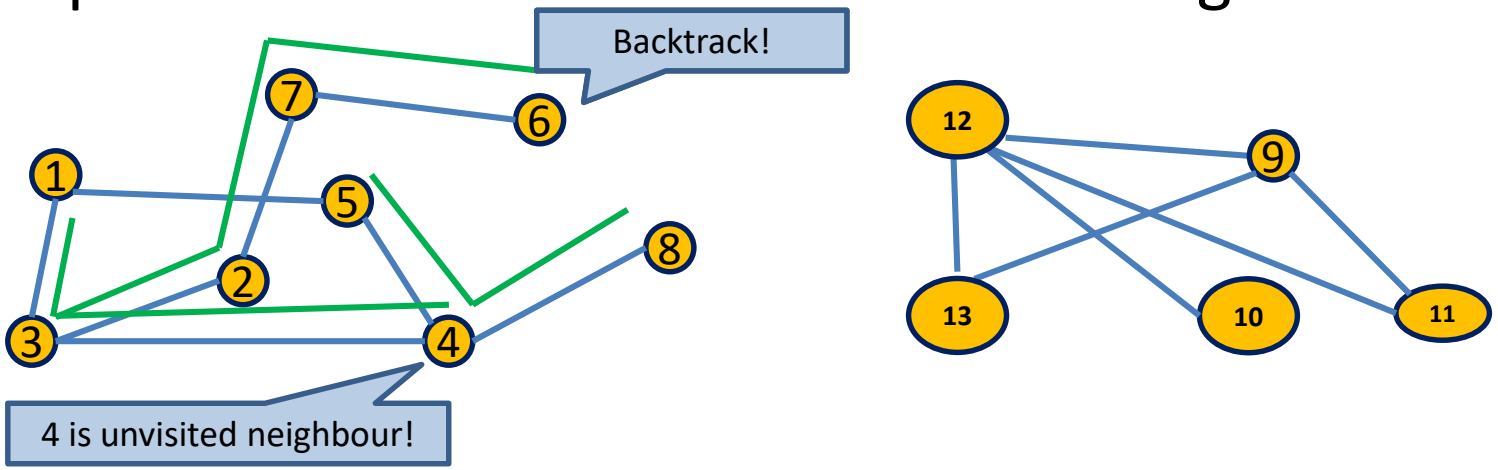
O(n)

O(1)

O(deg(v))

We keep a queue and a visited array. This takes O(n) space at most.
Total space requirement for BFS is therefore O(n).

Depth First Search - Another Traversal algorithm



Need visited array again!

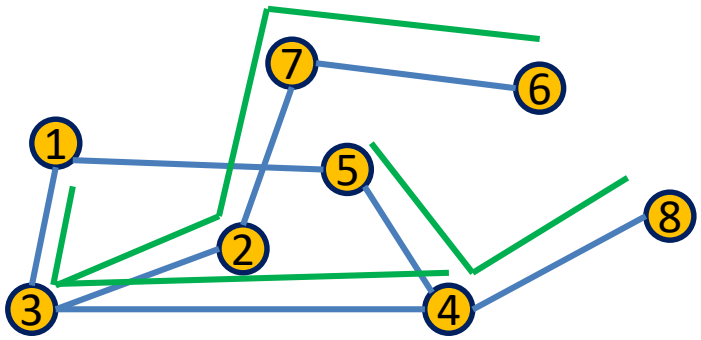
- The simple idea is : Keep going as "deep" into the graph as possible. This is done by an unvisited neighbour and visiting it, if it hasn't been visited.
- Once all neighbours of the current vertex ***x*** have been visited, the control goes to the vertex ***y*** that called the procedure on ***x***.
- Visited array needed to avoid looping!
- A naturally Recursive procedure due to backtracking.

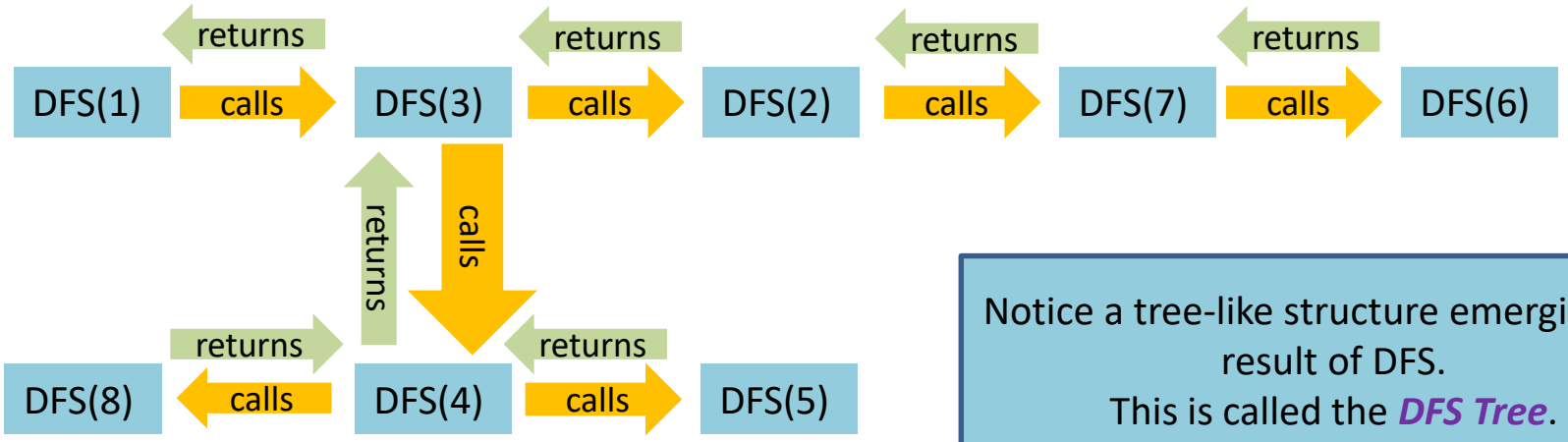
Mark 4 a | Mark 5 as visited! | Mark 8 as visited!

1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	1	1	1	1	1	1	0	0	0	0	0

Details on Depth First Search

```
DFS(x)
{
   visited[x] = 1;        // Mark x as visited.
   For each neighbor w of v
   {
            if(visited[w] == 0)
            {
                     DFS(w);
            }
   }
}
```





Notice a tree-like structure emerging as a result of DFS.
This is called the ***DFS Tree***.

Depth First Search - Applications

- Computing Strongly Connected Components: A directed graph is strongly connected if there exists a path from every vertex to every other vertex. Trivial Algo: Perform DFS **n** times. Efficient Algo: Single DFS.

- Checking for Biconnected Graph: A graph is biconnected if removal of any vertex does not affect connectivity of the other vertices. Used to test if network is robust to failure of certain nodes. A single DFS traversal algorithm.

- Topological Ordering: Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge (***x***, ***y***), vertex ***x*** comes before y in the ordering.

- Plus, almost everything that BFS can do!

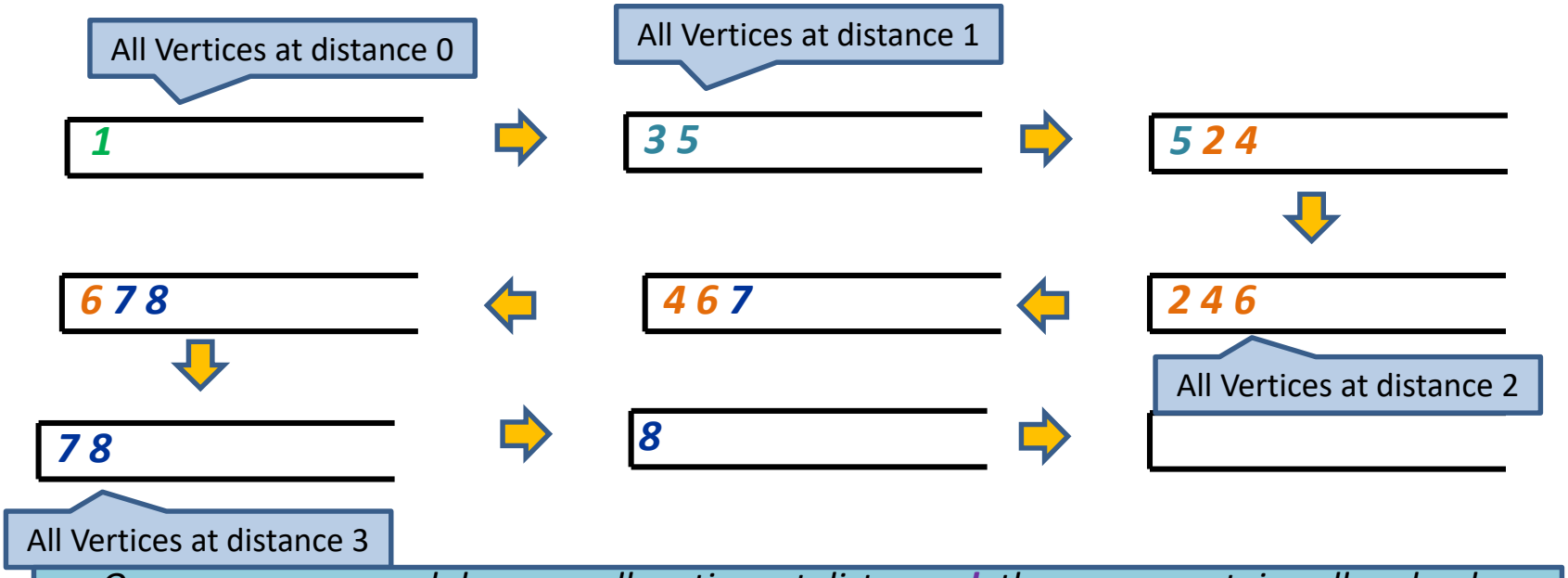
- For more applications, visit http://www.geeksforgeeks.org/applications-of-depth-first-search/

*In general, **DFS** is more powerful and more intuitive algorithm, than **BFS**!*

DFS and BFS

Some Applications.

Breadth First Search – A Graph Traversal Algorithm



*Once we process and dequeue all vertices at distance **d**, the queue contains all and only vertices at distance **d+1**.*

1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	1	1	1	1	1	1	0	0	0	0	0

Using BFS to compute distances

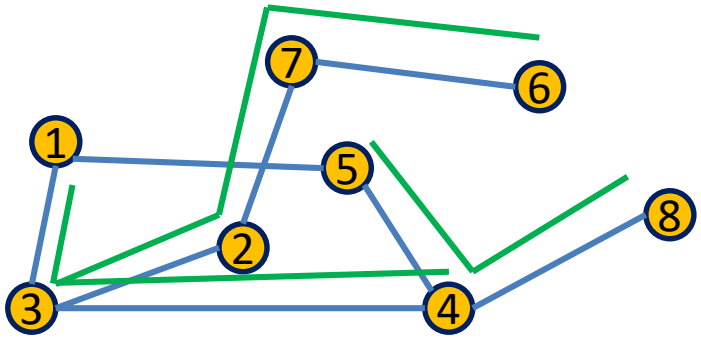
- BFS from *1* visits vertices in increasing order of ***distance.***

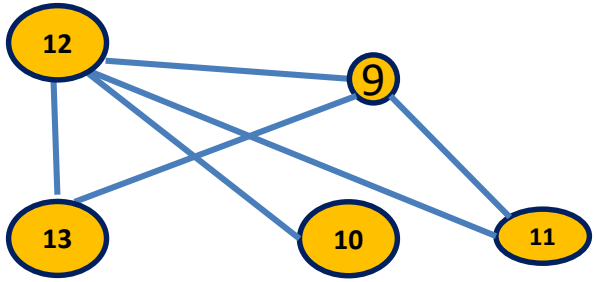
```
BFS(G, x)              //BFS on graph G, starting at vertex x.
  Create empty queue Q.
  Create array distance of length n=|V| and initialize to ∞.
  Enqueue(x,Q);  distance(x) = 0;
  while( Q is not empty)
  {
        v ←DeQueue(Q);
        For each neighbor w of v
        {
              if (distance(w) == ∞)
              {
                    distance(w) = distance(v) + 1;
                    Enqueue(w,Q);
              }
        }
  }
```

Idea: We don't need visited array because unvisited nodes have distance ∞. So a distance array suffices!

*This is essentially a **BFS** so time complexity is still **O(m + n).***

Depth First Search - Another Traversal algorithm

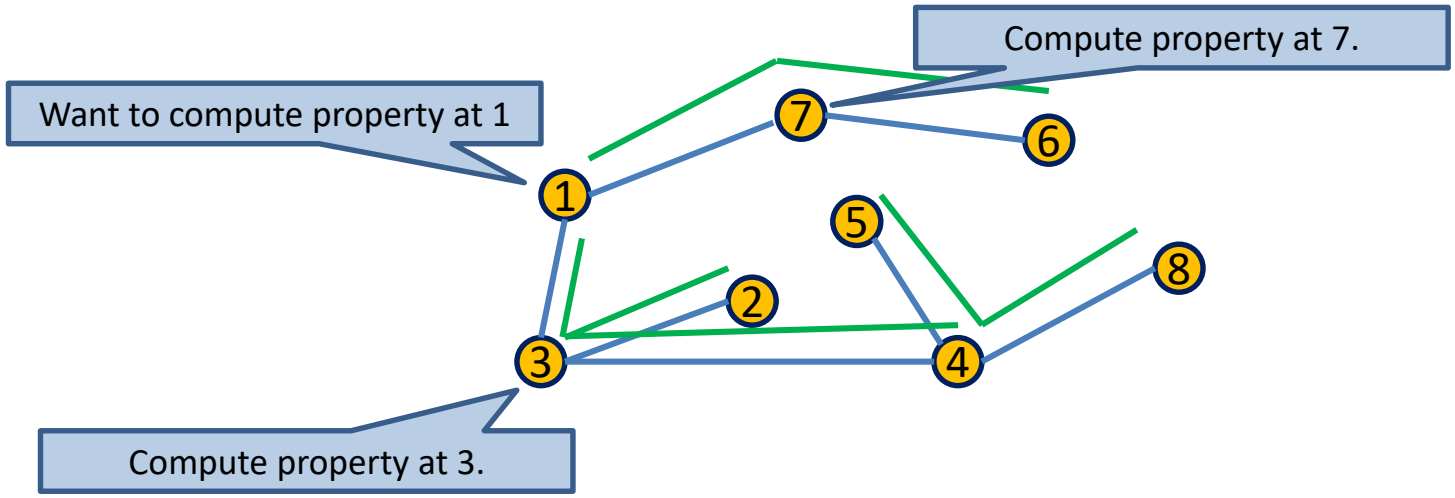




- Keep going as "deep" into the graph as possible. This is done by an unvisited neighbour and visiting it, if it hasn't been visited.
- Once all neighbours of the current vertex ***x*** have been visited, the control goes to the vertex ***y*** that called the procedure on ***x***.
- Visited array needed to avoid looping!
- A naturally Recursive procedure due to backtracking.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
1	1	1	1	1	1	1	1	0	0	0	0	0

Depth First Search - Another Traversal algorithm



Combine properties at 3 and 7 to compute property at 1. This is essentially done using recursion! DFS is naturally recursive and therefore is a suitable choice in this case.

Some recursive properties:

- Height (or depth) of a node in a tree.
- DFN numbering.
- Size of subtree.
- Reachability.
- …. and many more.