Graphs (contd.)

References: "Data Structures and Algorithm Analysis", C. Shaffer, pp. 371–388.

Paths and cycles

Path: A sequence of vertices $v_1, v_2, \ldots, v_n$ with an edge from v_i to v_{i+1} for $1 \leq i < n$ is a path of length $n - 1$.

A path is *simple* if all vertices on the path are distinct.

A *cycle* is a path of length at least 1 that connects v_i to itself.

A cycle is *simple* if the path is simple, except the first and last vertices are the same.

Source : Dr. Calvin & textbook

Graph ADT

```
public interface Graph {
    public class Edge {
        public int from, to;
        public Edge(int i, int j)    { from = i;     to = j;   }
    }


    public void init(int n); // initialize with n vertice

     public int numVertices(); // number of vertices of graph
    public int numEdges(); // number of edges of graph
    public void addEdge(int u, int v, int weight); // add edge
    public Iterator<Edge> getOutgoingEdges(int v); // get out edges from v
    public void delEdge(int u, int v); // delete edge from u to v
    public boolean isEdge(int u, int v); // Is there an edge from u to v
    public int weight(int u, int v); // get weight of edge from u to v

    public void setMark(int u, int val); // mark vertex u with value

    public int getMark(int u); // get value of mark for vertex u

}
```

Graph traversals (search)

- Many applications require visiting nodes or vertices of the graph in some order exactly once.
- A lot of applications for graph search

  -- how connected is a network ?

  -- search for an optimal solution in a combinatorial problem

  --- AI search (e.g. game playing)

Graph traversals

To insure visiting all vertices:

```
public void graphTraverse(Graph g) {
    for (int v=0; v < g.numVertices(); v++) {
        g.setMark(v, UNVISITED);
    }

    for (int v=0; v < g.numVertices(); v++) {
        if (g.getMark(v) == UNVISITED) {
            doTraverse(g, v);
     }
   }
}
```

We will next examine a couple of choices for "doTraverse()".

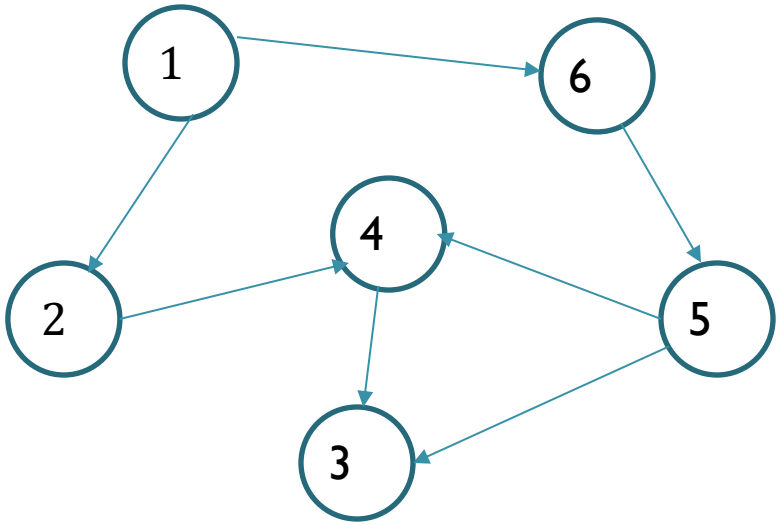
Source : Dr. Calvin & textbook

Depth first search

- similar to backtracking search we did for finding maze path. Examine a path as far as possible and then back track

```
public void dfs(Graph g, int v) {
    preVisit(g, v);  // add to dfslist()
    g.setMark(v,VISITED);
    Iterator<Graph.Edge> outEdgeIter = g.getOutgoingEdges(v);
    while (outEdgeIter.hasNext()) {
        Graph.Edge edge = outEdgeIter.next();
        int w = edge.to;
        if (g.getMark(w) == UNVISITED) {
            dfs(g, w);
        }
    }
    postVisit(g, v);
}
```

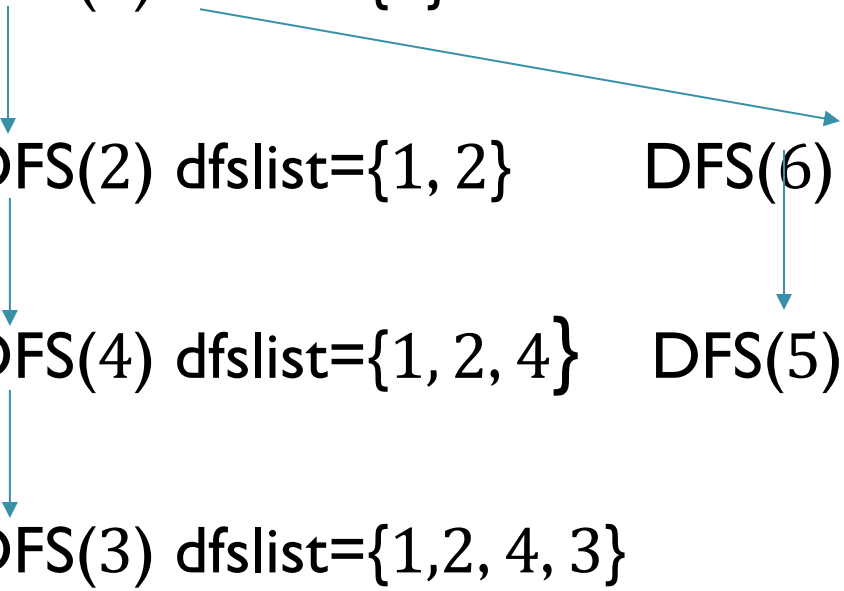
Graph search example



Graph search example – DFS tree

- Start vertex : 1; dfslist ={1}

DFS(1) dfslist={1}



DFS(2) dfslist={1, 2}

DFS(6) dfslist={1, 2, 4, 3, 6}

DFS(4) dfslist={1, 2, 4}

DFS(5) dfslist={1,2, 4, 3, 6, 5}

DFS(3) dfslist={1,2, 4, 3}

Breadth-first search

Like DFS, but replace stack with a queue.

Visit vertex's neighbors before continuing deeper in the tree.

Each vertex has initial mark 0 (unvisited). As explore graph, set mark of a vertex to one greater than mark of vertex that precedes it in BFS.

Mark indicates the level of a vertex in the BFS tree

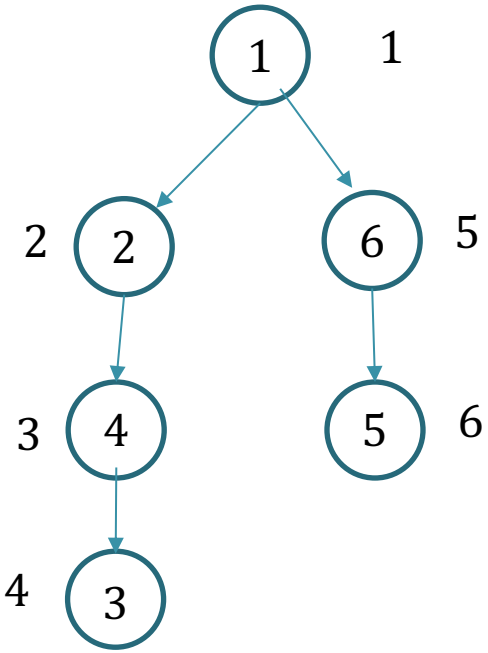
Source : Dr. Calvin & textbook

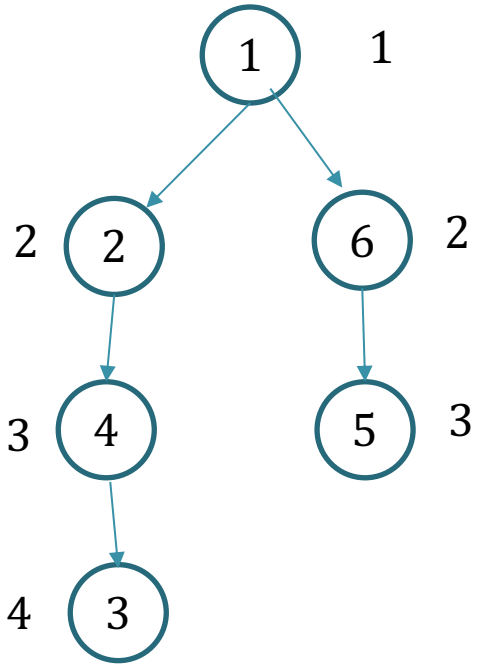
```
public void bfs(Graph g, int start) {
    Queue<Integer> vertexQueue = new LinkedList<Integer>();
    vertexQueue.add(start);
    g.setMark(start,1);
    while (!vertexQueue.isEmpty()) {
        int v = vertexQueue.remove(); // remove from head of the queue
        preVisit(g, v);
        Iterator<Graph.Edge> outEdgeIter = g.getOutgoingEdges(v);
        while (outEdgeIter.hasNext()) {
            Graph.Edge edge = outEdgeIter.next();
            int w = edge.to;
            if (g.getMark(w) == 0) {
                g.setMark(w, g.getMark(v)+1);
                vertexQueue.add(w);
            }
        }
        postVisit(g, v);
    }
}
```

Graph search example - BFS

- Start vertex : 1; Q = {1}; mark(1) = 1
- Remove 1 from Q, add 2,6 to Q; Q = {2,6} mark(2)= mark(6) = mark(1) + 1 = 2
- Remove 2 , add 4 to Q; Q = {6, 4} mark(4)= mark(2) + 1 = 3
- Remove 6 , add 5 to Q; Q = {4, 5} mark(5)= mark(6) + 1 = 3
- Remove 4 , add 3 to Q ; Q = {5,3} mark(3)= mark(4) + 1 = 4
- Remove 5 , no vertex to add; Q = {3}
- Remove 3, no vertex to add; Q = {}

DFS and BFS trees





Time complexities

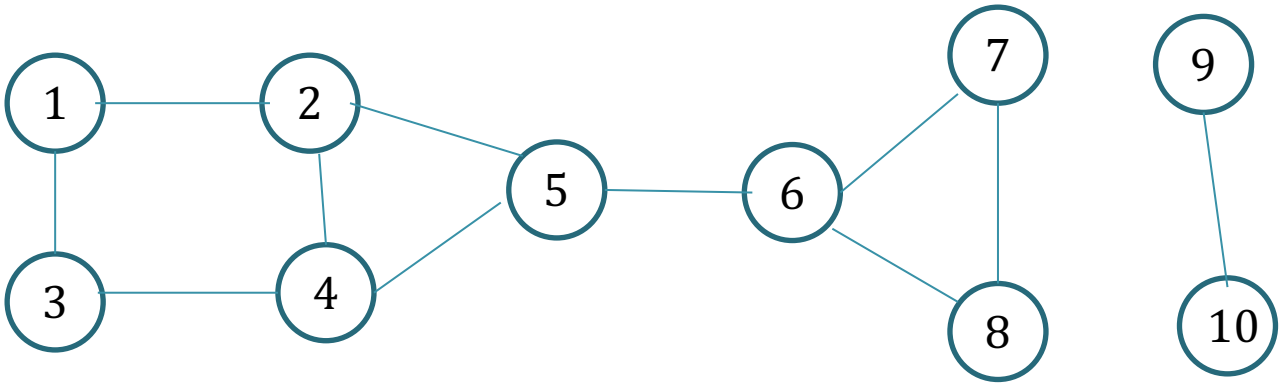
- Both DFS and BFS take O(n+m) time provided the operations hasNext() and next() of edge iterator and preVisit() and postVisit() operations take constant time.

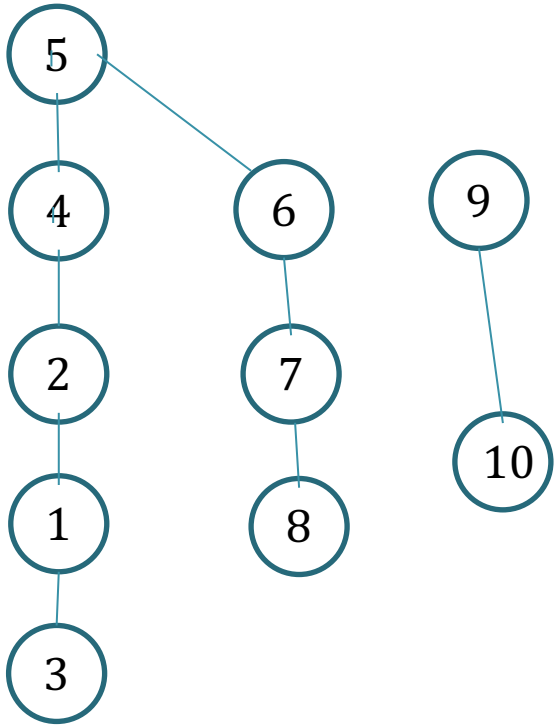
Here n is number of nodes and m is number of edges.

Source : Dr. Calvin & textbook

Undirected graph connectivity

- A graph is connected if there is a path between every pair of vertices in G.
- A connected component is a maximal subgraph of G which is connected.
- Can use BFS and DFS to find connected components of a G in O(n+m) time.
- A spanning tree of a connected graph G is a subgraph of G which is connected and has minimum number of edges. It has $n - 1$ edges where n = |V|
- BFS and DFS provide spanning trees.
- A spanning forest is a collection of spanning trees of a graph that contains all the vertices.





2 connected components –
DFS spanning forest

Similar BFS spanning forest exists. Note BFS gives path from u to v in G with smallest number of edges.

Single source shortest path problem

- **Given** : A directed graph G = (V,E) with weights on edges (w(e) for e ∈ E) and a source vertex s ϵ V
- **Required:** Find the shortest path length D[v] from s to v for all v ϵ V
- Example : shortest route between cities

  Length of path = sum of weights of edges in the path
- Negative weights with cycles can cause shortest path between vertices to have infinite number of edges (not converge)
- Focus on non-negative weights on edges

Dijkstra's shortest path alg.

- **Cut** – Partition of V into disjoint subsets V_1 and V_2 (V_2 = V- V_1)

  Cut-set – {(u,v) ∈ E | u ∈ V_1 and v ∈ V_2 }

- **Greedy approach idea** :

(a) Have a cut where V_1 is set of vertices for which we know shortest path length from s; initially V_1 = {s}

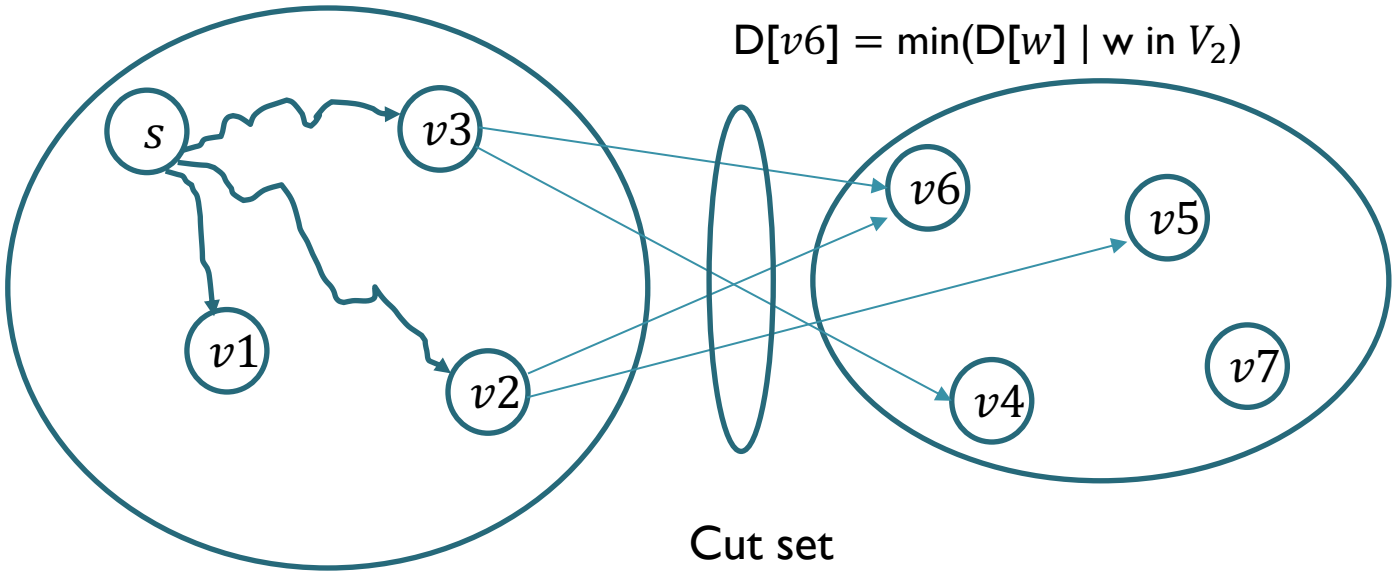
(b) For all v ∈ V_2 , we know shortest path length from s to v with vertices in path (except v) restricted to V_1 (at most one edge in cut-set). Let it be D[v]

(c) At each step we can move one vertex from V_2 to V_1 if V_2 is not empty. Why ?

Let D[u] = $\min_{v \in V_2} D[v]$.

Cannot have a better path from s to u that includes a vertex from V_2

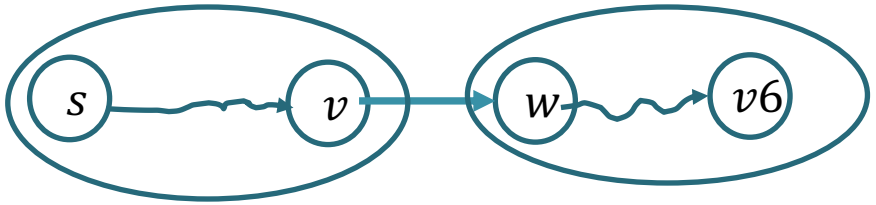
→ D[u] is shortest path from s to u



We know shortest path from s to every vertex in V_1

For very vertex in V_2 , we know shortest of all paths from s with intermediate vertices only in V_1

Can you find a shorter path from s to $v6$ that goes through a vertex say w in V_2 ?



Length of this path $\geq$ D[w] $\geq$ D[$v6$] hence not TRUE !

So D[$v6$] is shortest path from s to $v6$. We can add $v6$ to V_1

Updating D[v]'s

- Once we add a new vertex, set V_1 has changed. This means D[v] must be updated for remaining vertices in V_2

  Do we need to consider all vertices in V_2 ?

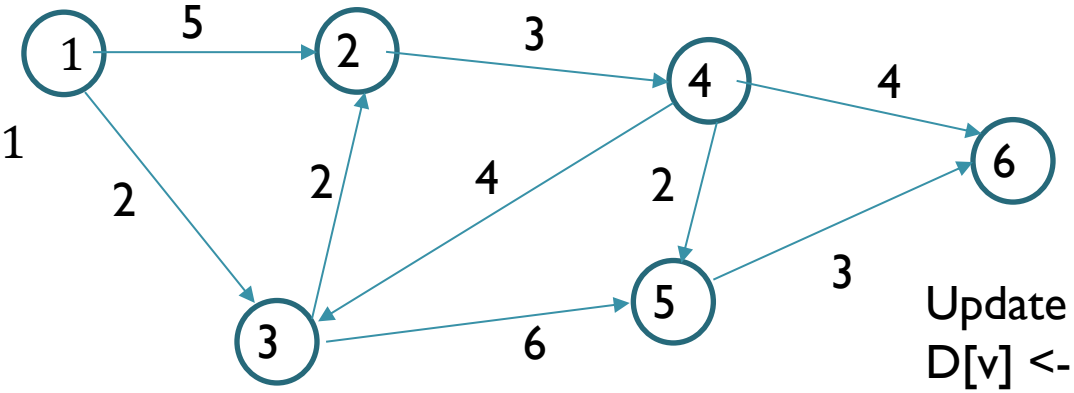
No. Only vertices adjacent to newly added vertex (e.g. $v6$)

- For each adjacent vertex w of $v6$,

update D[w] = min(D[w], D[$v6$] + w(($v6$,w))

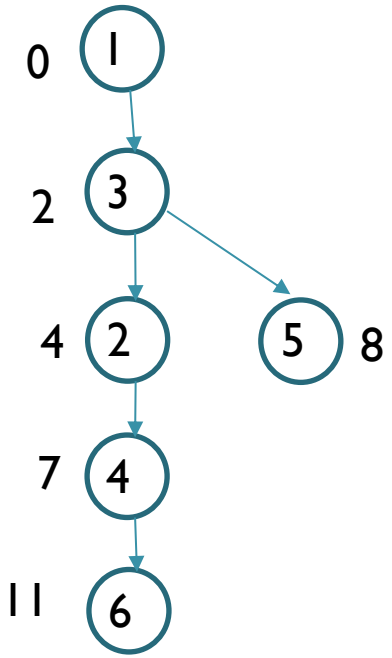
Dijkstra's algorithm example

Source = 1



Update :
D[v] <- min(D[v], D[u] + w((u,v)

D[v]	1	2	3	4	5	6
Add 1; update 2,5,6	0(*)	5	2	∞	∞	∞
Add 3 update 2,5	0(*)	4	2(*)	∞	8	∞
Add 2 update 4	0(*)	4 (*)	2(*)	7	8	∞
Add 4 update 6	0(*)	4(*)	2 (*)	7(*)	8	11
Add 5; no updates	0(*)	4(*)	2 (*)	7 (*)	8(*)	11



Add v6

```
Dijkstra(G, w, s)
1 for each v ∈ V do d[v] ← +∞
2 d[s] ← 0
3 S ← ∅
4 Q ← V
5 while Q ≠ ∅
6     u ← ExtractMin(Q)
7     S ← S ∪ {u}
8     for each   v ∈ Adj[u]
9         if d[v] > d[u] + w(u, v) then d[v] ← d[u] + w(u, v)
```

The total time required by Dijkstra's algorithm is $T = O(|V|) + |V| \cdot T_{ExtractMin} + |E| \cdot T_{Update}$. For the unordered array priority queue, this becomes $O(|V|)+|V|\cdot O(|V|)+|E|\cdot O(1) = O(|V|^2)$. For the binary heap, we get $T = O(|V|)+|V|\cdot O(\lg |V|)+|E|\cdot O(\lg |V|) = O(|E| \lg(|V|))$

Source : Dr. Calvin & textbook