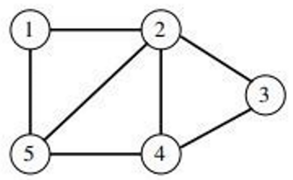
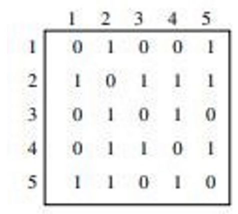
Graph(unweighted)

17 July 2021 02:11

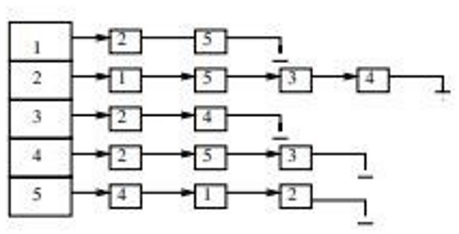
Graph is defined as G(E,V) where edges and vertices are used to define.
Two different ways of defining graphs in code:

1. Adjacency Matrix
2. Adjacency List





$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$



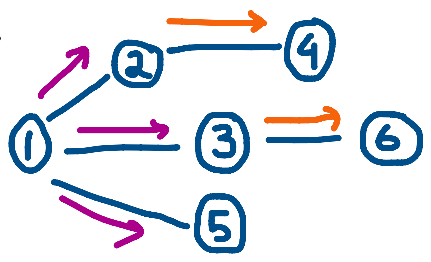
BFS:

Type of traversal where each of the adjacent edges to a node is visited layer by layer and with the help of a queue, runs until the queue is empty. Two Boolean arrays are required :

Processed

Visited

And an integer array called Parent to keep record of the traversal relation.

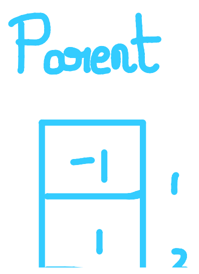


Initially Queue is only filled by start element. $\longrightarrow$ $\boxed{1}$

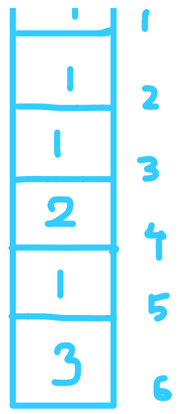
After level 1 is processed the adjacent edges are inserted into the queue
And the starting edge is marked processed and removed from queue. $\longrightarrow$ $\boxed{2}\boxed{3}\boxed{5}$

After level 2 is done then the adjacent and unprocessed edges of the edges in queue after level 1 are inserted in the queue. $\longrightarrow$ $\boxed{4}\boxed{6}$

```
bool processed[MAXV+1];
/* which vertices have been processed */
bool discovered[MAXV+1];
/* which vertices have been found */
int parent[MAXV+1];
/* discovery relation */
```



```
/* which vertices have been found */
int parent[MAXV+1];
/* discovery relation */

initialize_search(graph *g)
{
        int i; /* counter */
        for (i=1; i<=g->nvertices; i++) {
                processed[i] = discovered[i] = FALSE;
                parent[i] = -1;
        }
}

bfs(graph *g, int start){
        queue q;      /* queue of vertices to visit */
        int v;      /* current vertex */
        int y;      /* successor vertex */
        edgenode *p;    /* temporary pointer */

        init_queue(&q);
        enqueue(&q,start);
        discovered[start] = TRUE;

        while (empty_queue(&q) == FALSE) {
                v = dequeue(&q);
                process_vertex_early(v);
                processed[v] = TRUE;
                //before going into adjacent edges

                p = g->edges[v];
                while (p != NULL) {
                        y = p->y;
                        if ((processed[y] == FALSE) || g->directed)
                                process_edge(v,y);
                        if (discovered[y] == FALSE) {
                                enqueue(&q,y);
                                discovered[y] = TRUE;
                                parent[y] = v;
                        }
                        p = p->next;
                }
                process_vertex_late(v);
        }
}
```



Applications of BFS:

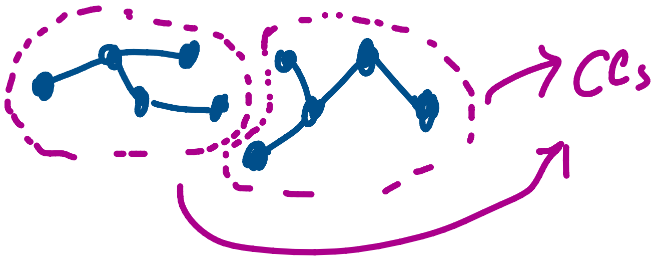
- **Find path:**

A function to find the path between two nodes in a recursive bottom up(LIFO) approach using the parent array produced by BFS. We print the stack to get the path from end to start.

```
find_path(stack *s,int start, int end, int parent[]){
    if(end == start || end == -1)
        return 0;
    else{
        push(&s,end);
        return find_path(&s, start, parent[end], parent)
    }
}
```

- **Connected Components:**

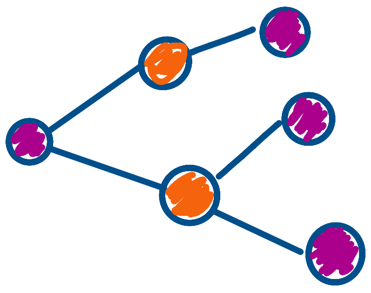
A connected component is a subgraph which contains nodes which are connected minimum by 1 edge, but is disconnected from the other part of the graph. We modify the BFS by defining process_edge_early() to print the current vertex;



```
connected_components(graph *g){
    int c;  //component counter
    int i;  //counter
    initialize_search(g);
    c=0;
    for(i=1;i<=g->nvertices;i++){
        c++;
        if(!discovered[i]){
            print(c);
            bfs(g,i);
        }
    }
}
```

- **<u>Two coloring or Bipartite Graph:</u>**

  A graph whose child edges are of opposite color of the parent edges and only two colors are possible. An array called color is required.



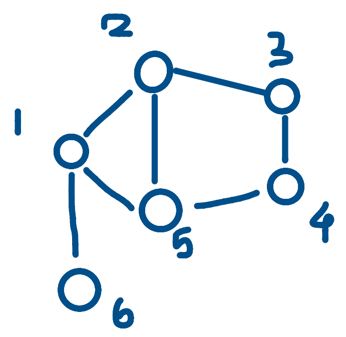
```
two_coloring(graph *g){
    int i;  //counter
    int color[g->nvertices+1];
    for(i=1;i<=g->nvertices;i++)
        color[i]=uncolored;

    bipartite = true;
    initialize_search(g);
    for(i=1;i<=g->nvertices;i++){
        if(discovered[i]){
            color[i] = white;
            bfs(g,i);
        }
    }
}

process_edge(int x, int y){
    if(color[x]!=color[y])
        color[y] = complement(color[x]);
    else{
        print("not possible");
        bipartite = false;
    }
}

complement(int x){
    if(color[x]==white) return black;
    if(color[x]==black) return white;
    return uncolored;
}
```

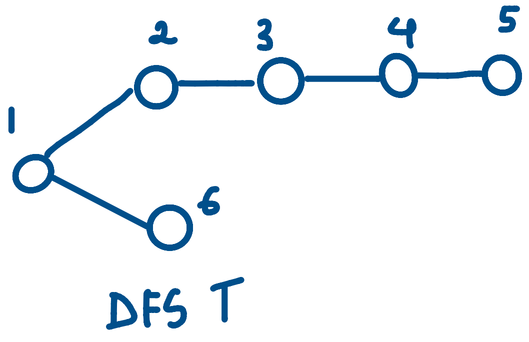
DFS:

Type of traversal where the first encountered node is traversed depth wise while it has no descendant left. It is a recursive stack based LIFO implementation. It results in a dfs tree and the arrays required are:

Processed
Visited

And an integer array called Parent to keep record of the traversal relation.



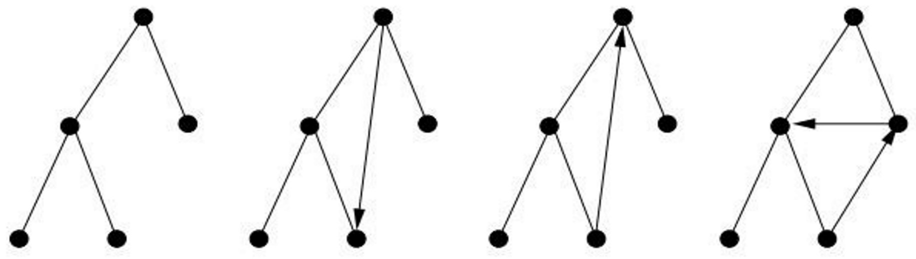


```
visited[g->nvertices]
processed[g->nvertices]
entry_time[g->nvertices]

dfs(graph *g, int start){
    edgenode *temp;
    int data;

    entry_time[start] = time;
    time = time+1;
    visited[start] = true;
    process_vertex_early(start);

    temp = g->edges[start];
    while(temp!=NULL){
        data = temp->data;
        if(!visited[start]){
            parent[data] = start;
            dfs(g,data);
        }
        else if(!processed[data] || g->directed)
            process_edge(start,data);
        if(finished) return;
        temp = temp->next;
        }

        time = time+1;
        exit_time[start] = time;
        process_vertex_late(start);
        processed[start] = time;
}
```

Applications of DFS:

- **Finding cycles and classifying edges:**

You can use the parent array and the time arrays in the dfs to classify different edges. If there's a back edge then there's a cycle.



```
classify_edge(int x, int y){
        if(parent[y] == x) return TREE;
        if(discovered[y] && !processed[y]) return BACK;
        if(processed[y] && entry_time[y] > entry_time[x]) return FRONT;
        if(processed[y] && entry_time[x] > entry_time[y]) return CROSS;
}

/*also can be used to find out a cycle*/
process_edge(int x, int y){
        if (parent[x] != y) {
                /* found back edge! */
                printf("Cycle from %d to %d:",y,x);
                find_path(y,x,parent);
                printf("\n\n");
                finished = TRUE;
        }
}
```

- **Topological sort in a DAG:**

Topological sorting is the most important operation on directed acyclic graphs (DAGs). It orders the vertices on a line such that all directed edges go from left to right. Such an ordering cannot exist if the graph contains a directed cycle, because there is no way you can keep going right on a line and still return back to where you started from!
Each DAG has at least one topological sort. The importance of topological sorting is that it gives us an ordering to process each vertex before any of its successors. Suppose the edges represented precedence constraints, such that edge (x,y) means job x must be done before job y. Then, any topological sort defines a legal schedule. Indeed, there can be many such orderings for a given DAG.

```
/* Stack used to store each edge to a DFS tree bottom up in order to get the Topo Sort
vertices left to right */

process_vertex_edge(int v){
    push(&s, v);
}

topological_sort(){
    int i;
    stack *s;

    initialise_stack(&s);
    initialise_search(g);

    for(i=1;i<=g->nvertices;i++){
        if(!discovered[i])
            dfs(g,i);
    }

    print(&s);
}
```