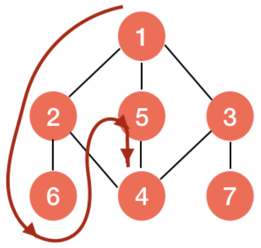
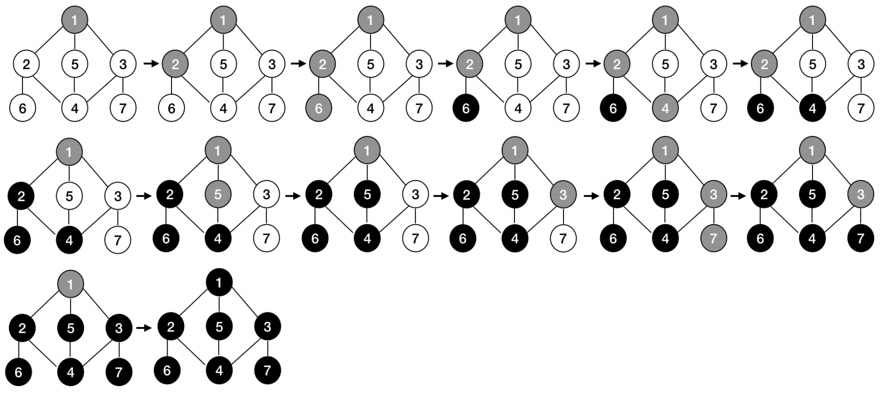
(/add_ques

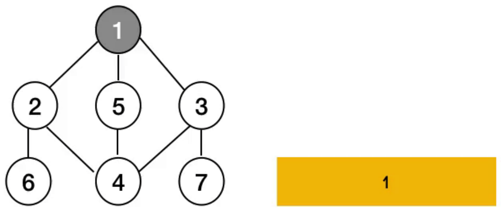
Depth First Search

Depth-first search or DFS is also a searching technique like BFS (/course/algorithms-bfs/). As its name suggests, it first explores the depth of the graph before the breadth i.e., it traverses along the increasing depth and upon reaching the end, it backtracks to the node from which it was started and then do the same with the sibling node.



Similar to the BFS, we also mark the vertices white, gray and black to represent unvisited, discovered and complete respectively.





Code for DFS

Similar to the BFS, we start by making all the nodes white.

```
DFS(G)
  for i in G.V
    i.color = white
```

Now we need to explore each of these vertices deeper. So, we will iterate again and call a function which will visit deeper for each node.

```
for i in G.V
  if i.color == white
    DFS-VISIT(G, i)
```

The DFS-VISIT will traverse over each of the nodes deeply. Now for each of the nodes passed to this function, the adjacent nodes are one level deeper. So, we will again iterate over the adjacent nodes and call DFS-VISIT function again on them.

```
DFS-VISIT(G, i)
  i.color = gray // we just discovered this vertex
  for v in G.Adj[u]
    if v.color == white
      DFS-VISIT(G, v)
  u.color = black
```

```
DFS(G)
  for i in G.V
    i.color = white

  for i in G.V
    if i.color == white
      DFS-VISIT(G, i)
```

```
DFS-VISIT(G, i)
  i.color = gray
  for v in G.Adj[i]
    if v.color == white
      DFS-VISIT(G, v)
  i.color = black
```

C **Python** **Java**

(/add_ques

```
#include <stdio.h>
#include <stdlib.h>
enum color{White, Gray, Black};

/*
  Node for linked list of adjacent elements.
  This will contain a pointer for next node.
  It will not contain the real element but the index of
  element of the array containing all the vertices V.
*/
typedef struct list_node {
  int index_of_item;
  struct list_node *next;
}list_node;

/*
  Node to store the real element.
  Contain data and pointer to the
  first element (head) of the adjacency list.
*/
typedef struct node {
  int data;
  enum color colr;
  list_node *head;
}node;

/*
  Graph will contain number of vertices and
  an array containing all the nodes (V).
*/
typedef struct graph{
  int number_of_vertices;
  node heads[]; // array of nodes to store the list of first nodes of each adjacency list
}graph;

node* new_node(int data) {
  node *z;
  z = malloc(sizeof(node));
  z->data = data;
  z->head = NULL;
  z->colr = White;

  return z;
}

list_node* new_list_node(int item_index) {
  list_node *z;
  z = malloc(sizeof(list_node));
  z->index_of_item = item_index;
  z->next = NULL;

  return z;
}

// make a new graph
graph* new_graph(int number_of_vertices) {
  //number_of_vertices*sizeof(node) is the size of the array heads
  graph *g = malloc(sizeof(graph) + (number_of_vertices*sizeof(node)));
  g->number_of_vertices = number_of_vertices;

  //making elements of all head null i.e.,
  //their data -1 and next null
  int i;
  for(i=0; i<number_of_vertices; i++) {
    node *z = new_node(-1); //*z is pointer of node. z stores address of node
    g->heads[i] = *z; //*z is the value at the address z
  }

  return g;
}

// function to add new node to graph
void add_node_to_graph(graph *g, int data) {
  // creating a new node;
  node *z = new_node(data);
  //this node will be added into the heads array of the graph g
  int i;
```

```
    for(i=0; i<g->number_of_vertices; i++) {
      // we will add node when the data in the node is -1
      if (g->heads[i].data < 0) {
        g->heads[i] = *z; //*z is the value at the address z
        break; //node is added
      }
    }
  }

  // function to check of the node is in the head array of graph or not
  int in_graph_head_list(graph *g, int data) {
    int i;
    for(i=0; i<g->number_of_vertices; i++) {
      if(g->heads[i].data == data)
        return 1;
    }
    return 0;
  }

  // function to add edge
  void add_edge(graph *g, int source, int dest) {
    //if source or edge is not in the graph, add it
    if(!in_graph_head_list(g, source)) {
      add_node_to_graph(g, source);
    }
    if(!in_graph_head_list(g, dest)) {
      add_node_to_graph(g, dest);
    }

    int i,j;
    // iterating over heads array to find the source node
    for(i=0; i<g->number_of_vertices; i++) {
      if(g->heads[i].data == source) { //source node found

        int dest_index; //index of destination element in array heads
        // iterating over heads array to find node containg destination element
        for(j=0; j<g->number_of_vertices; j++) {
          if(g->heads[j].data == dest) { //destination found
            dest_index = j;
            break;
          }
        }

        list_node *n = new_list_node(dest_index); // new adjacency list node with destination index
        if (g->heads[i].head == NULL) { // no head, first element in adjaceny list
          g->heads[i].head = n;
        }
        else { // there is head which is pointer by the node in the head array
          list_node *temp;
          temp = g->heads[i].head;

          // iterating over adjaceny list to insert new list_node at last
          while(temp->next != NULL) {
            temp = temp->next;
          }
          temp->next = n;
        }
        break;
      }
    }
  }

  void print_graph(graph *g) {
    int i;
    for(i=0; i<g->number_of_vertices; i++) {
      list_node *temp;
      temp = g->heads[i].head;
      printf("%d\t",g->heads[i].data);
      while(temp != NULL) {
        printf("%d\t",g->heads[temp->index_of_item].data);
        temp = temp->next;
      }
      printf("\n");
    }
  }

  void dfs_visit(graph *g, node *i) {
    i->colr = Gray;
    list_node *temp;
    temp = i->head;
    while(temp != NULL) {
      if (g->heads[temp->index_of_item].colr == White) {
        dfs_visit(g, &g->heads[temp->index_of_item]);
      }
      temp = temp->next;
    }
    i->colr = Black;
    printf("%d\n",i->data);
}

void dfs(graph *g) {
  int i;
  for(i=0; i<g->number_of_vertices; i++) {
    g->heads[i].colr = White;
  }

  for(i=0; i<g->number_of_vertices; i++) {
    if (g->heads[i].colr == White) {
      dfs_visit(g, &g->heads[i]);
    }
  }
}

int main() {
  graph *g = new_graph(7);
  add_edge(g, 1, 2);
  add_edge(g, 1, 5);
  add_edge(g, 1, 3);
  add_edge(g, 2, 6);
  add_edge(g, 2, 4);
  add_edge(g, 5, 4);
  add_edge(g, 3, 4);
  add_edge(g, 3, 7);
  dfs(g);
  return 0;
}
```

As BFS uses a queue (https://www.codesdope.com/blog/article/queue-in-c/), the DFS is using a stack (https://www.codesdope.com/blog/article/stacks-in-c/) for its implementation. The recursive calls of the DFS-VISIT are stored in a stack.

Analysis of DFS

Both loops in the DFS function is running in $O(|V|)$ time. In DFS-VISIT, the loop is executed once for each edge in the adjacency list. Also DFS-VISIT is only called if the vertex is white, so the loop will be executed only once for each vertex i.e., $O(|E|)$. Thus, the total time taken will be $O(|E|+|V|)$.

Graph is a vast topic and we have just went through an introduction with these chapters. But you can now learn any other algorithm related to graphs easily whenever you need them because you have gone through all the basics. We will also keep including new articles on BlogsDope (https://www.codesdope.com/blog/) to cover more and more topics. Even you can share your knowledge by contributing articles to BlogsDope (https://www.codesdope.com/blog/submit-article/).

“ For infrastructure technology, C will be hard to displace. ”

- Dennis Ritchie

PREV (/course/algorithms-bfs/) (/course/algorithms-minimum-spanning-tree/) NEXT