Assignment 3

March 29, 2024

1 DAT600

1.0.1 Ardijan Rexhaj, Daniel Alvsåker, Ove Oftedal

1.1 Task 1:

1.1.1 a) 1.

1 -> [2]

2 -> [2, 3, 4]

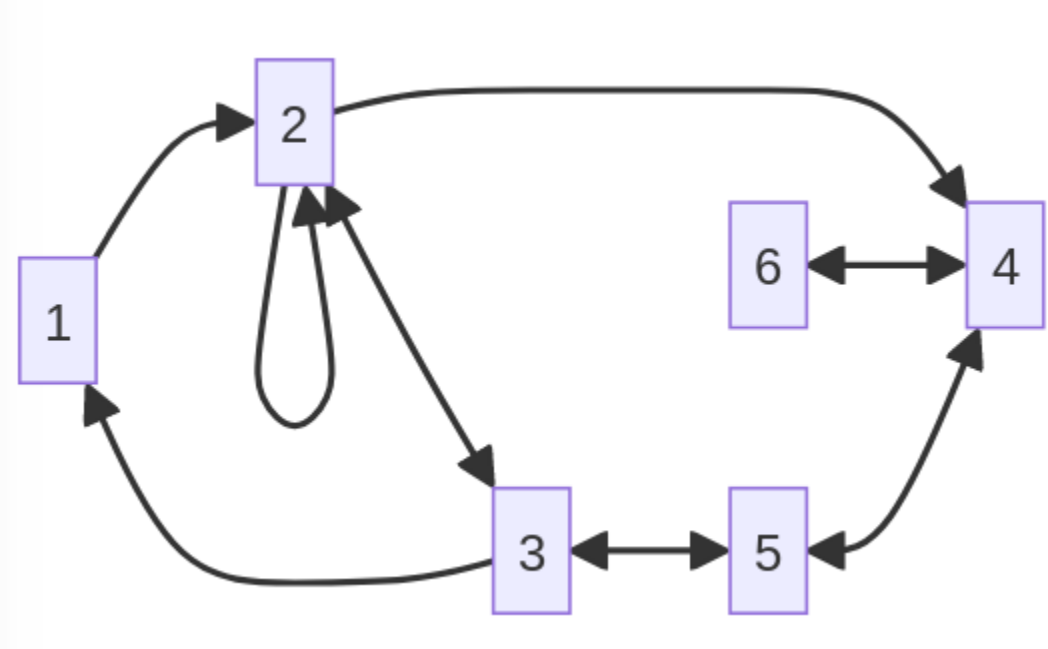
3 -> [1, 2, 5]

4 -> [5, 6]

5 -> [3, 4]

6 -> [4]

1.1.2 a) 2.



1.1.3 a) 3.

A -> [B]

B -> [C,D]

C -> [E,F]

D -> [E,F]

E -> [F,J]

F -> [B,J,G,H]

G -> []

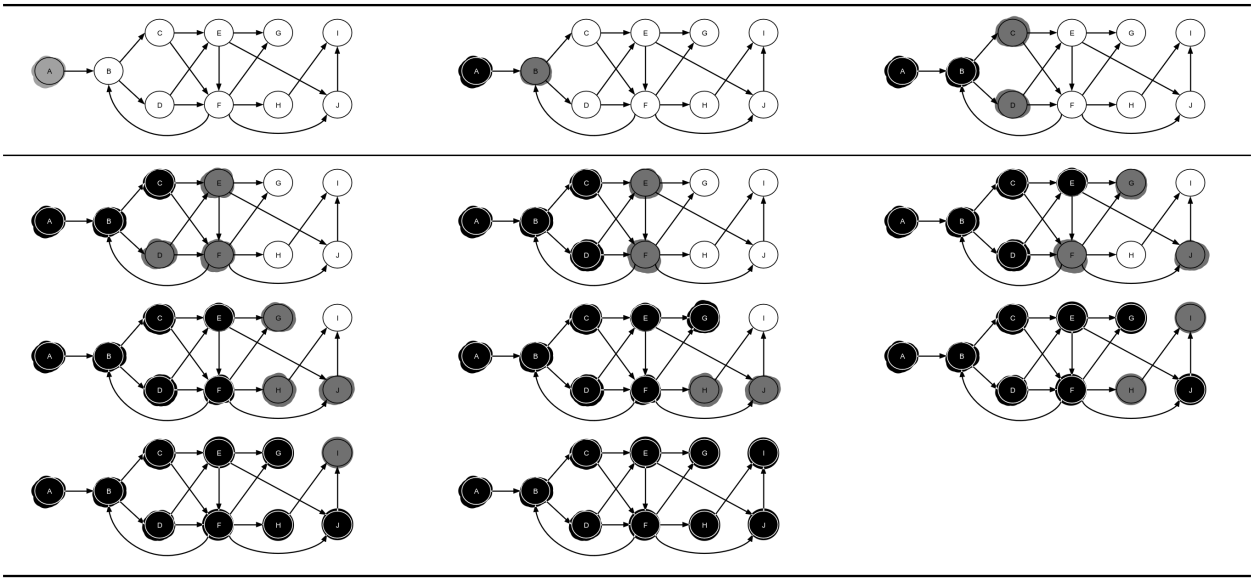
H -> [I]

I -> []

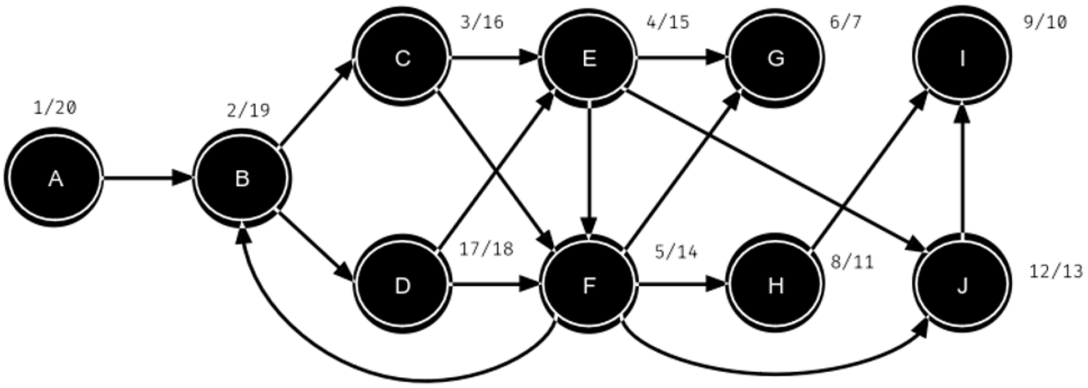
J -> [I]

1.1.4 b)

Breadth first search:

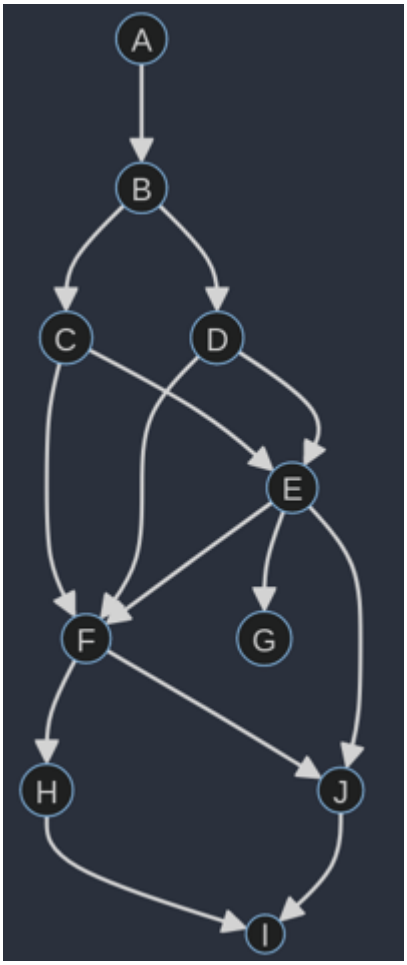


Depth first search:



1.1.5 c)

In order to make the graph on figure 1 into a directed acyclic graph, we can remove the (F,B) edge.



The topological sort produces the following dependency sequence:

`[A,B,D,C,E,F,J,H,I,G]`

1.1.6 d)

An algorithm which converts a directed graph into an directed acyclic graph can be done through the following code. This algorithm works by first detecting a cycle through the recursion stack, which simply keeps track of how we recursed depth first to see if any edges point back to a node on that stack, if it does then we remove the edge and check the other nodes.

[2]:
```rust
use std::collections::HashMap;

fn print_graph(graph: HashMap<usize, Vec<usize>>){
    for i in 0..graph.len(){
        println!("vertex: {}, connections: {:?}", i , graph.get(&i).unwrap())
    }
}

fn add_edge(a: usize, b: usize, graph: &mut HashMap<usize, Vec<usize>>) {
    graph.entry(b).or_default();
    graph.entry(a).and_modify(|x| x.push(b)).or_insert(vec![b]);
}

fn find_and_remove_cycles_recursive(
    node: usize,
    visited: &mut Vec<bool>,
    recursion_stack: &mut Vec<bool>,
    graph: &mut HashMap<usize, Vec<usize>>,
) -> bool {
    if recursion_stack[node] {
        return true;
    }

    if visited[node] {
        return false;
    }

    visited[node] = true;

    recursion_stack[node] = true;

    let connected_nodes = graph.get(&node).unwrap().clone();

    for cn in connected_nodes {
        if find_and_remove_cycles_recursive(cn, visited, recursion_stack,␣
 ↪graph) {
            // if a connected node returns true here it means the edge
            // is producing a cycle we can thus remove the connection
            graph.entry(node).and_modify(|x| x.retain(|y| *y != cn));
        }
    }
```

```
    recursion_stack[node] = false;

    false
}

fn find_and_remove_cycles(graph: &mut HashMap<usize, Vec<usize>>){
    let mut visited = vec![false; graph.len()];
    let mut recursion_stack = vec![false; graph.len()];

    for i in 0..graph.len() {
        find_and_remove_cycles_recursive(i, &mut visited, &mut recursion_stack,␣
 ↪graph);
    }
}



let mut graph: HashMap<usize, Vec<usize>> = HashMap::new();

add_edge(0, 1, &mut graph);
add_edge(0, 2, &mut graph);
add_edge(1, 2, &mut graph);
add_edge(2, 0, &mut graph); // Cycle
add_edge(2, 1, &mut graph); // Cycle

find_and_remove_cycles(&mut graph);

println!("Graph Test");
print_graph(graph);

let mut graph: HashMap<usize, Vec<usize>> = HashMap::new();

//Figure 1 but letters are replaced with numbers
add_edge(0, 1, &mut graph);
add_edge(1, 2, &mut graph);
add_edge(1, 3, &mut graph);
add_edge(2, 4, &mut graph);
add_edge(2, 5, &mut graph);
add_edge(3, 4, &mut graph);
add_edge(3, 5, &mut graph);
add_edge(4, 5, &mut graph);
add_edge(4, 6, &mut graph);
add_edge(4, 9, &mut graph);
add_edge(5, 6, &mut graph);
add_edge(5, 7, &mut graph);
add_edge(5, 9, &mut graph);
```

```
add_edge(5, 1, &mut graph); // Cycle, F->B
add_edge(7, 8, &mut graph);
add_edge(9, 8, &mut graph);
add_edge(8, 2, &mut graph);
add_edge(8, 2, &mut graph); // Cycle, I->C
add_edge(2, 0, &mut graph); // Cycle, C->A

find_and_remove_cycles(&mut graph);
println!("Graph from fig 1");
print_graph(graph);
```

```
Graph Test
```

1.2 Task 2:

1.2.1 a)

In this case it is possible to connect all the edges with a cost of just 26, this algorithm centers around making a locally optimal decision based on a starting point which in this case was the cheapest edge. From there we continually expand the network with the cheapest node possible until all nodes are connected.

[3]:
```
const A: usize = 0;
const B: usize = 1;
const C: usize = 2;
const D: usize = 3;
const E: usize = 4;
const F: usize = 5;
const G: usize = 6;
const H: usize = 7;


fn find_cheapest_network(mut edges: Vec<(usize, usize, u32)>, vertecies:␣
 ↪Vec<usize>) {
    // sort them by cost
    edges.sort_by_key(|&(_, _, cost)| cost);

    let cheapest_edge = edges[0];
    let mut connected_vertecies = vec![cheapest_edge.0, cheapest_edge.1];
    let mut network = vec![cheapest_edge];

    while connected_vertecies.len() != vertecies.len() {
        let mut filtered_edges: Vec<(usize, usize, u32)> = edges
            .iter()
            .filter(|&&(u, v, _)| {
                (connected_vertecies.contains(&u) || connected_vertecies.
 ↪contains(&v)) // edges must have a vertex in the network
```

```
                        && !(connected_vertecies.contains(&u) &&␣
↪connected_vertecies.contains(&v)) // both vertecies cannot be in the network
                        // && ( // Limit the number of edges vertex D can have.
                        //     network.iter().filter(|&(x, y,_)| (u==D ||␣
↪v==D) && x == D || y == D ).count()
                        // )<3
            })
            .cloned()
            .collect::<Vec<_>>();
        filtered_edges.sort_by_key(|&(_, _, cost)| cost);

        let cheapest_connection_to_new_vertex = filtered_edges[0];
        network.push(cheapest_connection_to_new_vertex);

        if !connected_vertecies.contains(&cheapest_connection_to_new_vertex.0) {
            connected_vertecies.push(cheapest_connection_to_new_vertex.0)
        }
        if !connected_vertecies.contains(&cheapest_connection_to_new_vertex.1) {
            connected_vertecies.push(cheapest_connection_to_new_vertex.1)
        }
    }

    let cost: u32 = network.iter().map(|&(_, _, cost)| cost).sum();

    println!("Network: {:?}", network);
    println!("Cost: {:?}", cost);
    println!("Connected Vertices: {:?}", connected_vertecies);
}


let edges = vec![
    (A, B, 5),
    (A, D, 1),
    (B, D, 4),
    (B, H, 8),
    (C, D, 2),
    (C, G, 6),
    (D, E, 2),
    (D, F, 4),
    (E, H, 8),
    (F, G, 9),
    (F, H, 7),
];

let vertices = vec![A, B, C, D, E, F, G, H];

find_cheapest_network(edges, vertices);
```

```
Network: [(0, 3, 1), (2, 3, 2), (3, 4, 2), (1, 3, 4), (3, 5, 4), (2, 6, 6), (5,
7, 7)]
Cost: 26
Connected Vertices: [0, 3, 2, 4, 1, 5, 6, 7]
vertex: 0, connections: [1, 2]
Network: [(0, 3, 1), (2, 3, 2), (3, 4, 2), (1, 3, 4), (3, 5, 4), (2, 6, 6), (5,
7, 7)]
Cost: 26
Connected Vertices: [0, 3, 2, 4, 1, 5, 6, 7]
```

1.2.2 b)

If we restrict the vertex D to only have up to 3 nodes, we cannot achieve a cost below or equal to 30.

The code is the same as above, but now we add a condition to limit the number of edges for node D, it then becomes impossible to achieve the cost limit of 30.

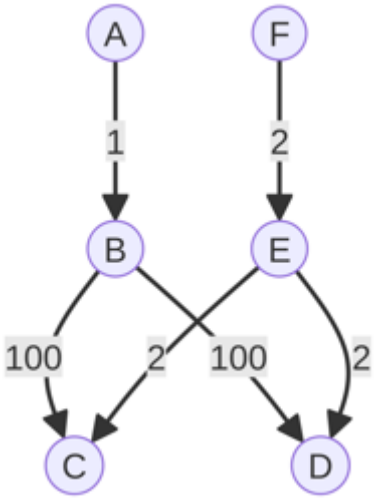
Extra condition, commented out in the code block above:

```
&& (
    network.iter().filter(|&&(x, y,_)| (u==D || v==D) && x == D || y == D ).count()
)<3
```

Output from running the code with condition:

```
Network: [(0, 3, 1), (2, 3, 2), (3, 4, 2), (0, 1, 5), (2, 6, 6), (1, 7, 8), (5, 7, 7)]
Cost: 31
Connected Vertices: [0, 3, 2, 4, 1, 6, 7, 5]
```

It is not possible to achieve a cost cap of 30 if D is limited to 3 edges. The algorithm is sensitive to the starting position, if the cheapest edge is not connected to an optimal starting vertex it might not discover an optimal solution. A graph where this algorithm would not preform well in would be the following graph (ignore the arrows, it should be undirected):



However this issue can be solved by calculating for all possible starting positions, or at least optimized by scoring each vertex based on how many edges it connects and the average cost to those edges with a N'th connection distance.

1.2.3 c)

Currently the cost is 26 without the limitation of the number of edges that D can have and swapping the cost of edges will certainly produce a network that is below the original cost.

Generally it is possible to improve the network cost iff:

There exists an edge which we do not use, which is cheaper than our most expensive edge.

In our case an optimal swap would be, (5,7,7) which is the edge E-H with a cost of 7. This can be swapped for the cheapest edge which we do not use, which has a cost of 5 between A-B, thus achieving a cost of 24.

Changes made to edges:

```
    let edges = vec![
        // (A, B, 5),
        (A, B, 7),
        (A, D, 1),
        (B, D, 4),
        (B, H, 8),
        (C, D, 2),
        (C, G, 6),
        (D, E, 2),
        (D, F, 4),
        (E, H, 8),
        (F, G, 9),
        (F, H, 5),
        // (F, H, 7),
    ];
```

Output

```
Network: [(0, 3, 1), (2, 3, 2), (3, 4, 2), (1, 3, 4), (3, 5, 4), (5, 7, 5), (2, 6, 6)]
Cost: 24
Connected Vertices: [0, 3, 2, 4, 1, 5, 7, 6]
```

1.3 Task 2:

1.3.1 a and b)

[5]:
```rust
use std::{
    collections::{HashMap, HashSet},
};

fn add_edge(a: usize, b: usize, graph: &mut HashMap<usize, Vec<usize>>) {
    graph.entry(b).or_default();
    graph.entry(a).and_modify(|x| x.push(b)).or_insert(vec![b]);
```

```rust
}

fn dfs_topological_sort(
    graph: &HashMap<usize, Vec<usize>>,
    node: usize,
    visited: &mut HashSet<usize>,
    stack: &mut Vec<usize>,
) {
    visited.insert(node);

    if let Some(adjacent_nodes) = graph.get(&node) {
        for &adj_node in adjacent_nodes {
            if !visited.contains(&adj_node) {
                dfs_topological_sort(graph, adj_node, visited, stack);
            }
        }
    }

    stack.push(node);
}

fn topological_sort(graph: &HashMap<usize, Vec<usize>>) -> Vec<usize> {
    let mut visited = HashSet::new();
    let mut stack = Vec::new();

    for &node in graph.keys() {
        if !visited.contains(&node) {
            dfs_topological_sort(graph, node, &mut visited, &mut stack);
        }
    }

    stack.reverse();
    stack
}

fn find_cycles(graph: &HashMap<usize, Vec<usize>>) -> Vec<HashSet<usize>> {
    let mut visited = HashSet::new();
    let mut rec_stack = HashSet::new();
    let mut all_cycles = Vec::new();
    let mut path_stack = Vec::new();

    let mut nodes: Vec<usize> = graph.keys().copied().collect();
    nodes.sort_unstable();

    for node in nodes {
        if !visited.contains(&node) {
            find_cycles_recursive(
```

```
                node,
                &mut visited,
                &mut rec_stack,
                graph,
                &mut path_stack,
                &mut all_cycles,
            );
        }
    }
    let all_cycles: Vec<HashSet<usize>> = all_cycles
        .into_iter()
        .map(|x| x.into_iter().collect())
        .collect();

    let mut filtered_cycles: Vec<HashSet<usize>> = all_cycles
        .clone()
        .into_iter()
        .filter(|cycle| {
            !all_cycles
                .iter()
                .any(|other_cycle| other_cycle != cycle && other_cycle.
 ↪is_superset(cycle))
        })
        .collect();

    let non_cyclic_nodes: HashSet<usize> = graph
        .keys()
        .copied()
        .collect::<HashSet<usize>>()
        .difference(&filtered_cycles.iter().flatten().copied().collect())
        .copied()
        .collect();
    if !non_cyclic_nodes.is_empty() {
        filtered_cycles.push(non_cyclic_nodes);
    }

    filtered_cycles
}

fn find_cycles_recursive(
    node: usize,
    visited: &mut HashSet<usize>,
    recursive_stack: &mut HashSet<usize>,
    graph: &HashMap<usize, Vec<usize>>,
    path_stack: &mut Vec<usize>,
    all_cycles: &mut Vec<Vec<usize>>,
) {
```

```
    if recursive_stack.contains(&node) {
        if let Some(pos) = path_stack.iter().rposition(|&x| x == node) {
            all_cycles.push(path_stack[pos..].to_vec());
        }
        return;
    }
    // println!("{:?}", path_stack);

    if !visited.insert(node) {
        return;
    }

    recursive_stack.insert(node);
    path_stack.push(node);

    if let Some(neighbors) = graph.get(&node) {
        for &next in neighbors {
            find_cycles_recursive(
                next,
                visited,
                recursive_stack,
                graph,
                path_stack,
                all_cycles,
            );
        }
    }

    recursive_stack.remove(&node);
    path_stack.pop();
}



const A: usize = 0;
const B: usize = 1;
const C: usize = 2;
const D: usize = 3;
const E: usize = 4;
const F: usize = 5;
const G: usize = 6;


let mut graph: HashMap<usize, Vec<usize>> = HashMap::new();

add_edge(A, B, &mut graph);
add_edge(A, D, &mut graph);
```

```
add_edge(D, B, &mut graph);
add_edge(B, A, &mut graph);
add_edge(B, C, &mut graph);
add_edge(D, C, &mut graph);
add_edge(C, E, &mut graph);
add_edge(C, F, &mut graph);
add_edge(F, E, &mut graph);
add_edge(E, G, &mut graph);
add_edge(G, F, &mut graph);

let result_groups = find_cycles(&graph);
println!("Grouped: {:?}", result_groups);

let result_sort = topological_sort(&graph);
println!("Topological sort: {:?}", result_sort);
{
    let champions = result_groups.iter().find(|cycle| {cycle.
 ↪contains(&result_sort[0])});
    println!("Champions are: {:?}", champions);
};
```

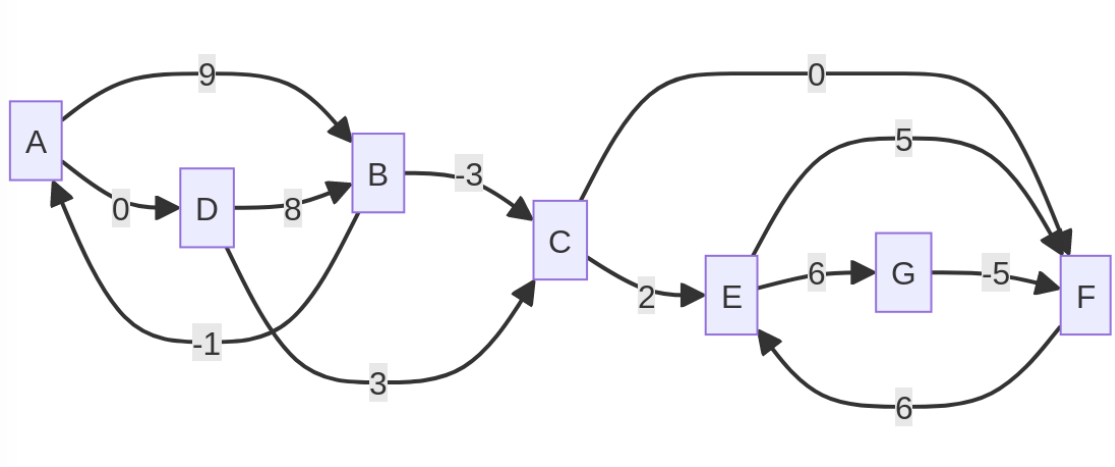
```
Grouped: [{0, 3, 1}, {4, 6, 5}, {2}]
Topological sort: [3, 1, 2, 4, 6, 5, 0]
Champions are: Some({0, 3, 1})
```

As for the running time of the algorithm the main component for finding groups is the find_cycles function, which needs to visit all vertices at least once. This function calls a recursive function which visits all nodes in addition it must filter all edges which in a worst case scenario is E^2 since it must compare all edges to current edges, and again add any missing edges. Thus the running time is $O(V + E^2 + E)$.

1.4 Task 4

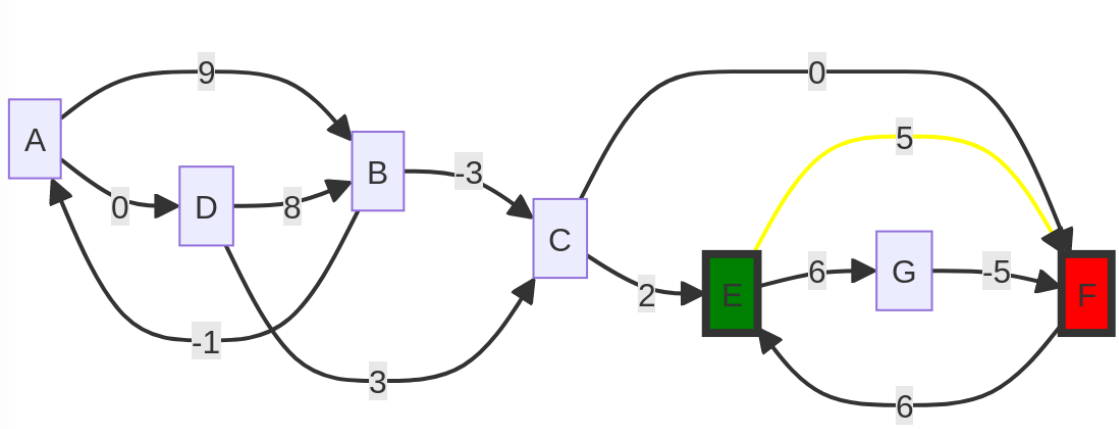
1.4.1 a)

Given the graph:

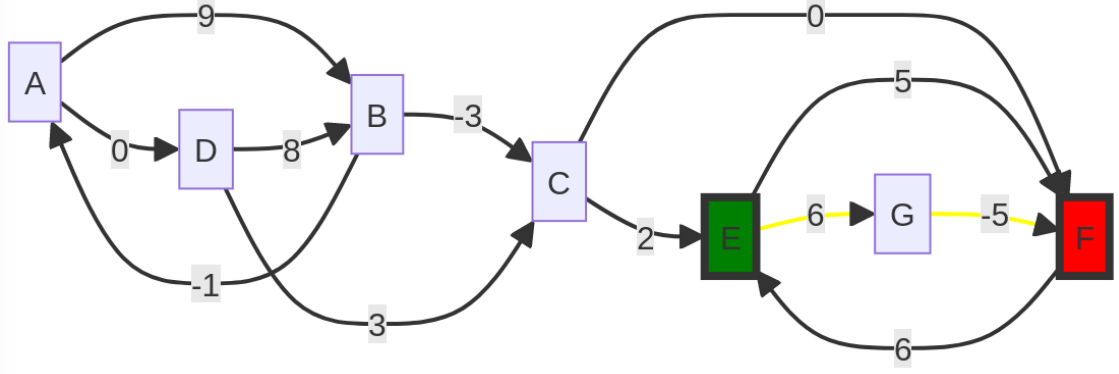


Choosing vertex E, and when finding the shortest path to F Dijkstra's algorithm will want to go directly to it with a cost of 5, and would accept this answer as the shortest one. The path with the least cost though is via G first, then to F, where the cost would then be $6 - 5 = 1$

The route Dijkstra's algorithm would choose:



Optimal Route:



1.4.2 b)

For the algorithm to work on negative edges, it would need to retroactively look through all paths instead of assuming it has gotten to the most efficient one immediatly. Usually as soon as the algorithm has found the target node, it will stop searching. This needs to be changed for it to work. One solution could be to go through all paths and then compare which costs the least, but this would affect the running time dramatically. Changing the algorithm to a dynamic programming algorithm instead of a greedy algorithm could also be used, remembering previously calculated routes, which would reduce the runtime.

1.5 Task 5

1.5.1 a)