graph

Divyanshu Nauni

October 31, 2021

1 functions:

1. **DFS** Here the dfs of the graph is printed along with the edge type of each edge and each node has a start and end time indicating the time at which they are discovered and finished in the dfs traversal.
   In this example the graph was disjoint as it created 3 components (not strongly component). The time complexity of dfs id O(V+E).

2. **Strongly Connected Component using Tarjens algo** In this function we find the strongly connected components of the input graph using tarjens algo.
   The algorithm takes directed graph as input, and produces a partition of the graph's strongly connected components. Each vertex of the graph appears in exactly one of the strongly connected components.
   Any vertex that is not on a directed cycle forms a strongly connected component all by itself.

   The time complexity of this algorithm is O(V+E).

3. **To check if graph is semi connected or not** To check if the graph is connected or not, what i did to make a reverse graph of the original graph.
   Now for each node, we run dfs and store the nodes reachable from the node in the original graph. Now for the same node run dfs and store the nodes reachable in the reverse graph.
   Now take union of these two and if union is equal to all vertices then continue for next vertex. else if union is not equal to vertices then not semiconnected.

   the time complexity of this algo is O(V(V+E)).

4. **pair shortest path using dijkstra's algo** This function takes argument as source node and destination node using priority queue. if there is no path possible then it prints infinity.
   The time complexity of dijkstra is O(V+ElogV).

5. **compressed graph** this function will print the graph with minimum edges retaining the strongly connected components of the original graph. The compressed graph here does not have multiple edges between components.
   the idea is to to use tarjens algo to find the strongly connected components and once found. we use this to find head of each node i.e., vertices within a component have same head . Now we maintain a 2D array to store info if there is already a edge between a component to another, so that we will not add multiple edges.
   time complexity: first we use tarjens algo which runs in O(V+E) then after that we iterate each vertex and all edges so another O(V+E). here i have not taken the edges withen a component so overall time complexity will be O(V+E).

2 ouput images:

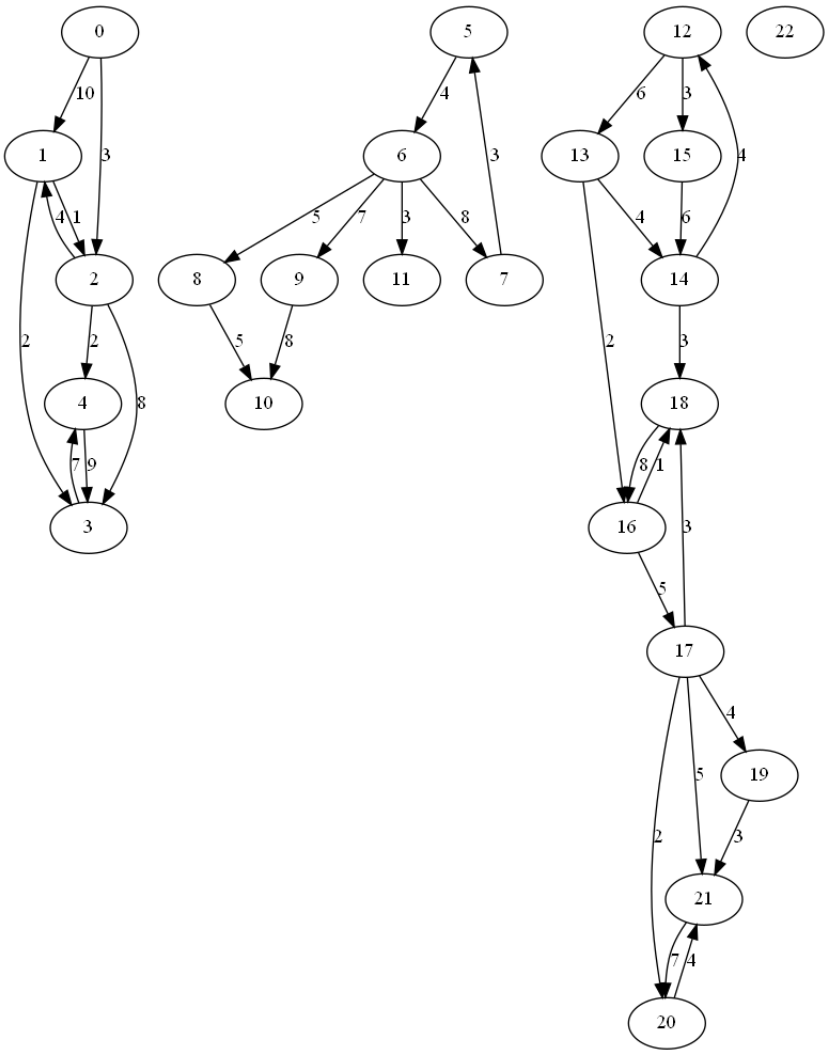

fig:input graph

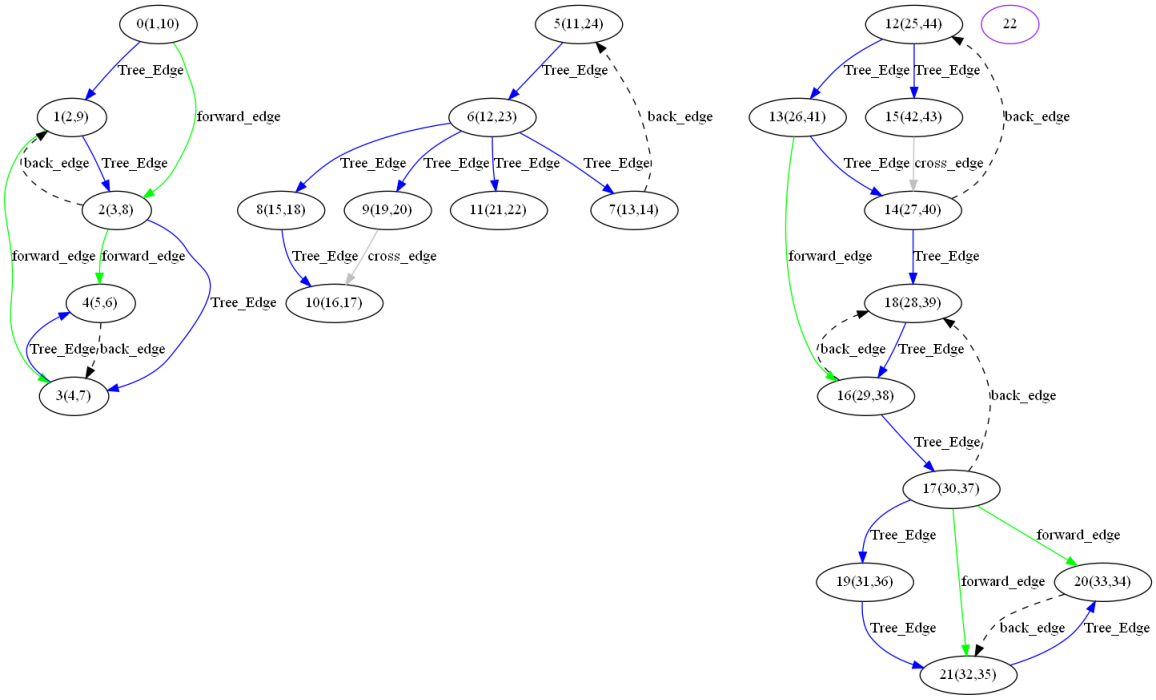

fig:dfs graph

here the edges are colored. the blue edges are tree edges . the black edges are dashed and are back edges. the green edges are forward edges. the grey edges are cross edges.

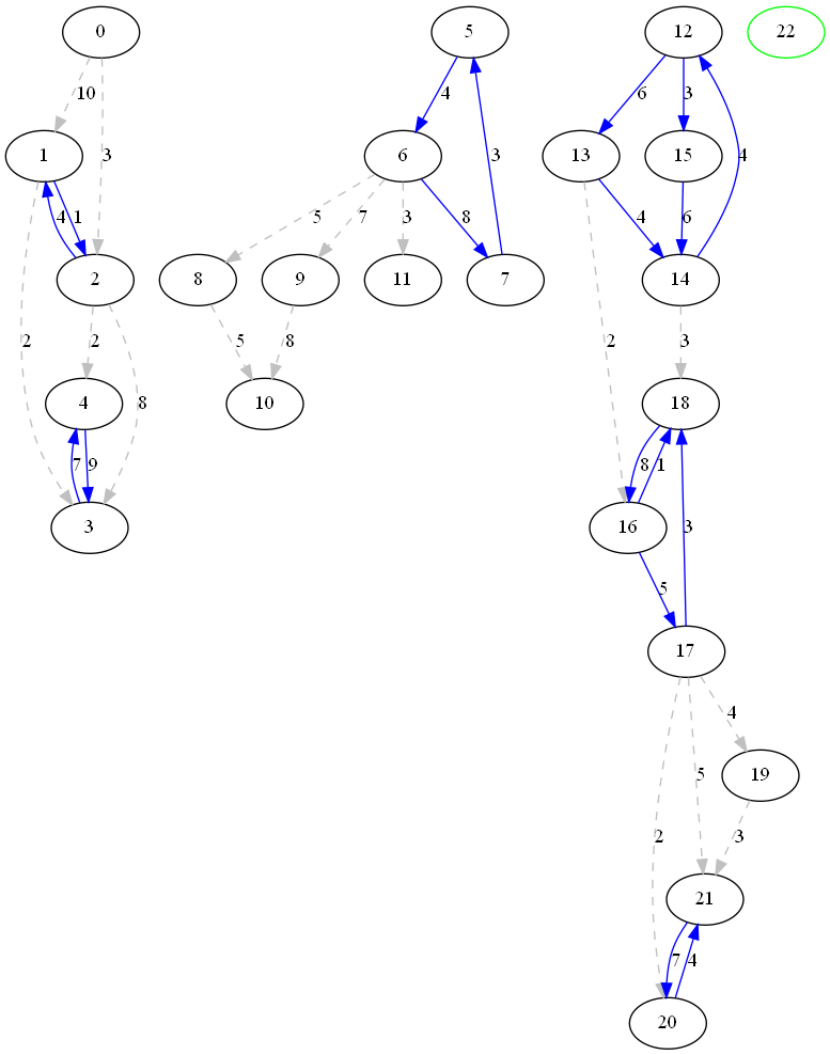

fig:strongly connected graph

the edges are of two types. the edges within a SCC are blue colored and edges across SCC are grey

```
enter operation to perform on the directed graph

1.find all strongly connected components
2.find dfs of the graph         3.find the distance between source and destination
4.check if the graph is semi connected or not   5.to print the minimal strongly connected graph
6.exit
3
enter source and destination
12 21
0       infinity
1       infinity
2       infinity
3       infinity
4       infinity
5       infinity
6       infinity
7       infinity
8       infinity
9       infinity
10      infinity
11      infinity
12      0
13      6
14      9
15      3
16      8
17      13
18      9
19      17
20      15
21      18
22      infinity
the distance between    12      and     21      =       18


continue
Y        or N
```

fig:distance from 12 to 21

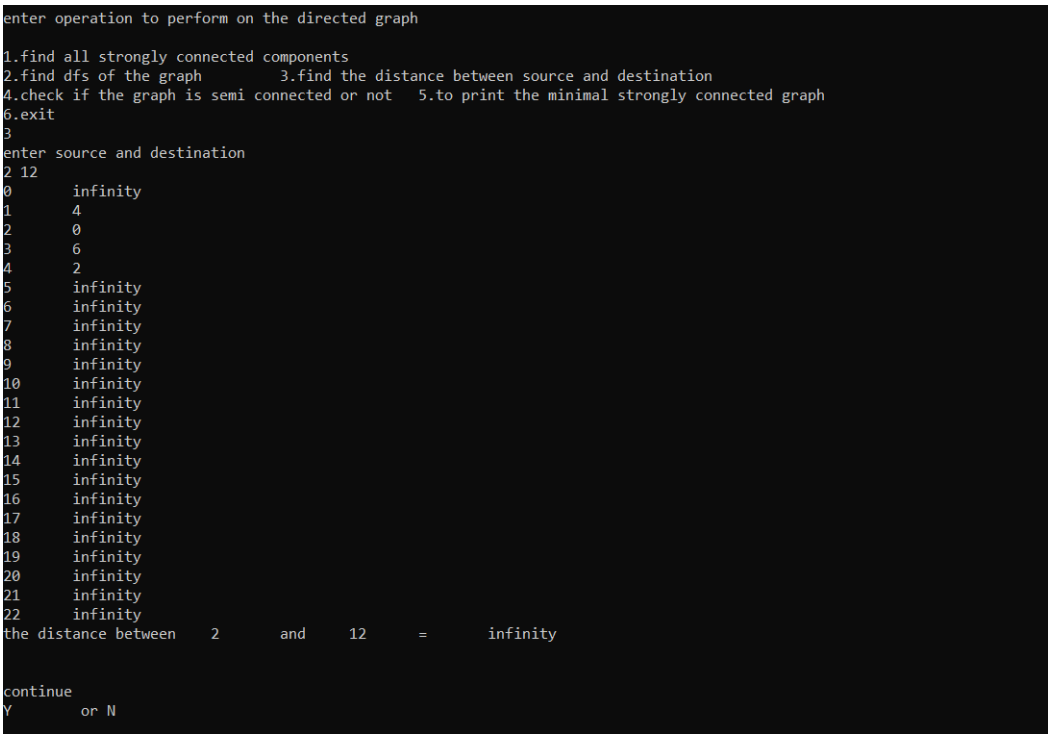

fig: distance from 2 to 12

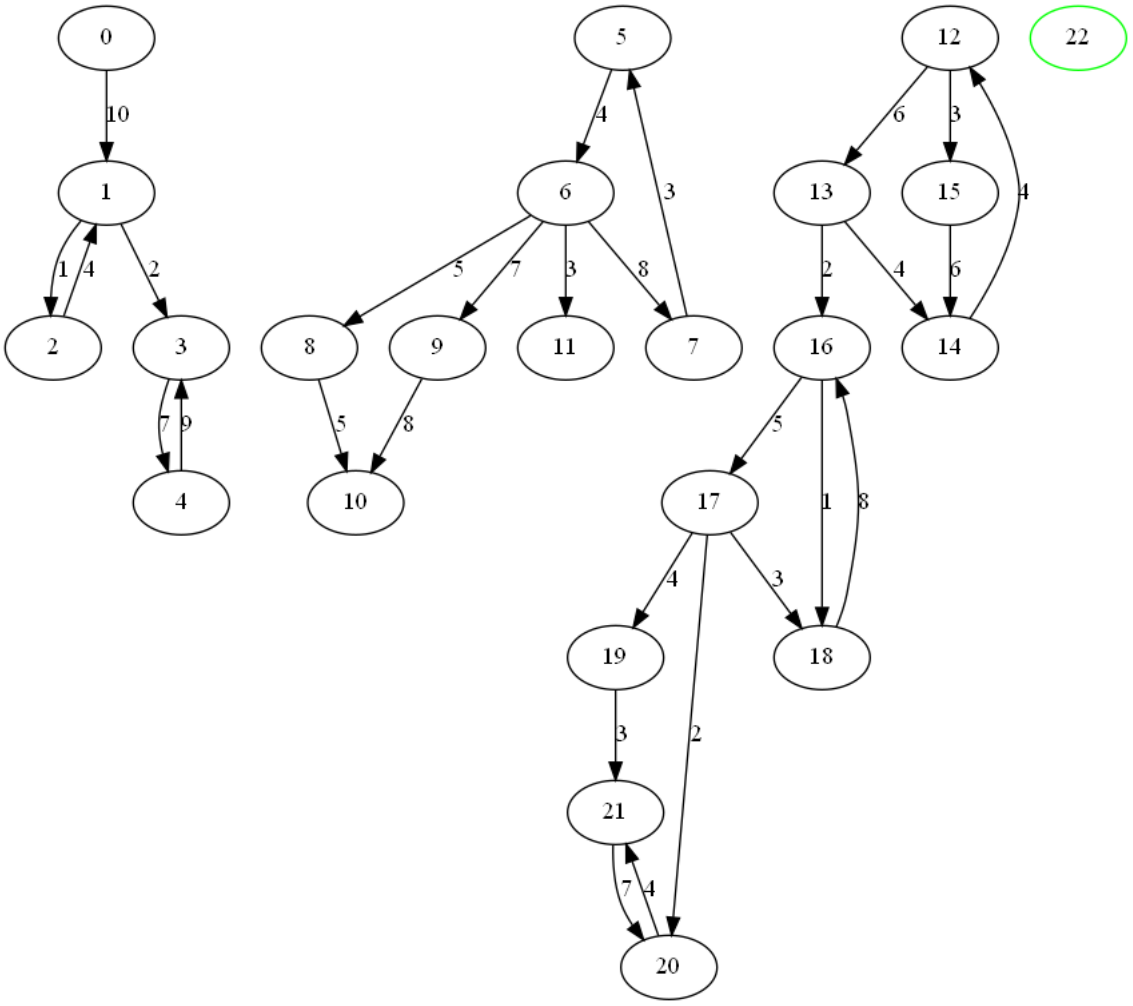

fig: compressed graph