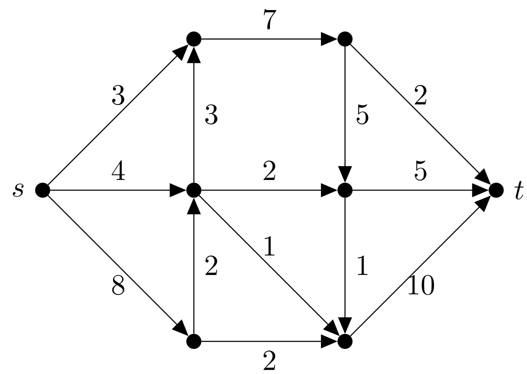
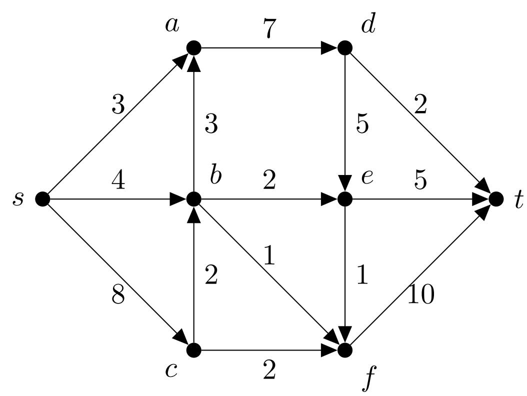
Caleb Logemann
MATH 566 Discrete Optimization
Homework 6

1. Consider the graph below



Find a shortest path and prove optimality using duality (find dual LP and its optimal solution)

First let me redraw the graph with all of the vertices labeled.



From observation it is possible to see that the shortest path is $s \to b \to e \to t$ and it has weight 11.

In order to verify this, the dual of the shortest path linear program can be solved. The dual of the shortest path linear program is shown below

$$\begin{array}{lll} \max & y_t - y_s & \\ \text{s.t.} & y_v - y_u \leq c(uv) & \forall (u,v) \in E \\ & y_s = 0 & \end{array}$$

This linear program can be solved using the following sage script.

```
import operator
M = Matrix([[0, 3, 4, 8, 0, 0, 0, 0],
            [0, 0, 0, 0, 7, 0, 0, 0],
            [0, 3, 0, 0, 0, 2, 1, 0],
            [0, 0, 2, 0, 0, 0, 2, 0],
            [0, 0, 0, 0, 0, 5, 0, 2],
            [0, 0, 0, 0, 0, 0, 1, 5],
            [0, 0, 0, 0, 0, 0, 0,10],
```

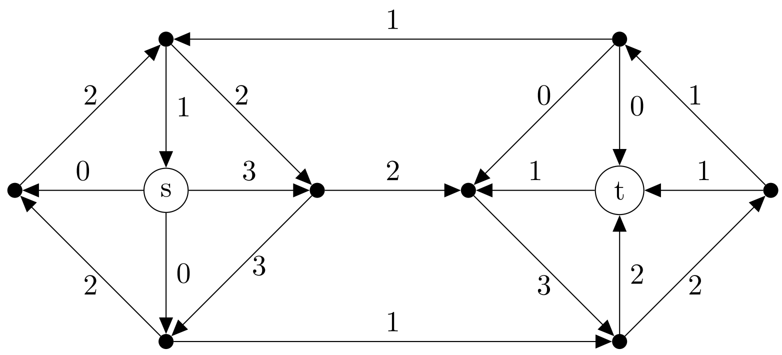
```
                [0, 0, 0, 0, 0, 0, 0, 0]])
G = DiGraph(M, weighted=True)
G.relabel({0:'s', 1:'a', 2:'b', 3:'c', 4:'d', 5:'e', 6:'f', 7:'t'})
s = 's'
t = 't'
milp = MixedIntegerLinearProgram(maximization=True)
y = milp.new_variable(nonnegative=True)
milp.set_objective(y[t] - y[s])
milp.add_constraint(y[s] == 0)
for edge in G.edges():
    milp.add_constraint(y[edge[1]] - y[edge[0]] <= edge[2])


print('Objective Value: {}'.format(milp.solve()))
sol = milp.get_values(y)
sol = sorted(sol.items(), key=operator.itemgetter(0))
for i, v in sol:
    print('y[%s] = %s' % (i, v))
```

The output of this script is as follows

```
Objective Value: 11.0
y[a] = 2.0
y[b] = 4.0
y[c] = 2.0
y[d] = 9.0
y[e] = 6.0
y[f] = 1.0
y[s] = 0.0
y[t] = 11.0
```

This shows that the linear program found a path of length 11 and the results show that $s \to b \to e \to t$ is adding up the weights on their respective edges. In other words $y[b] = 4$ and the weight from $s \to b$ is 4. The weight from $b \to e$ is 2, so $y[e] = 4 + 2 = 6$, and the weight from $e \to t$ is 5, so $y[t] = 6 + 5 = 11$.

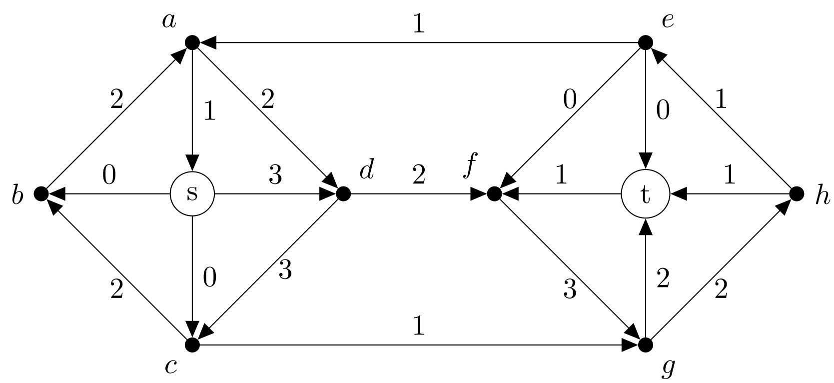
2. Consider the network below with given edge values, forming an integer feasible flow. Find a list of path and cycle flows whose sum is this flow.



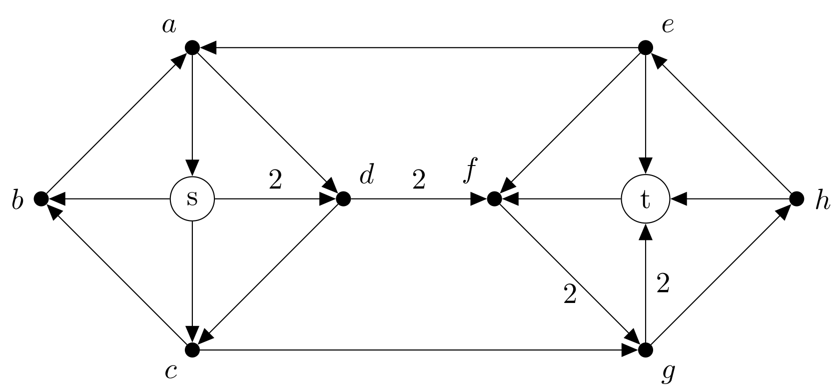
Gallai; Ford and Fulkerson proved a theorem that stated that any flow can be decomposed into s-t paths, $\mathcal{P}$ and circuits $\mathcal{C}$ with weight function $w : \mathcal{P} \cup \mathcal{C} \to \mathbb{R}^+$, such that

$$\begin{aligned} f(e) &= \sum_{e \in P \in \mathcal{P}} (w(P)) + \sum_{e \in C \in \mathcal{C}} (w(C)) \\ value(f) &= \sum_{P \in \mathcal{P}} (w(P)) \\ |\mathcal{P} + \mathcal{C}| &\leq |E(G)| \end{aligned}$$

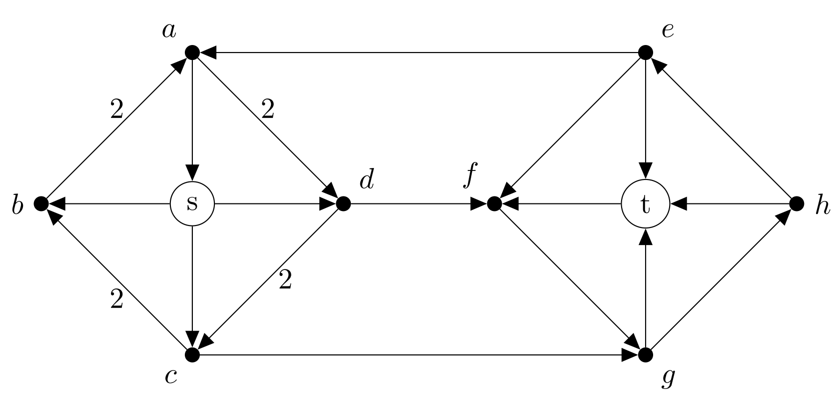
In order to find such a decomposition I will first relabel all of the vertices as follows.



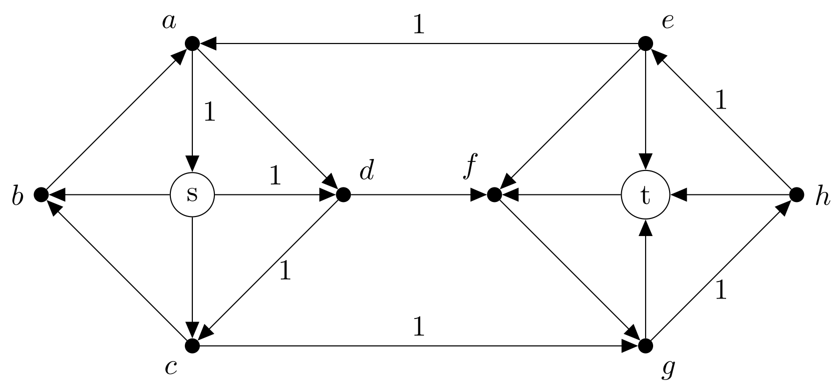
Now one way this flow can be decomposed is by using 1 path and 3 circuits as follows. The one s-t path is $P = s \to d \to f \to g \to t$ with weight $w(P) = 2$.



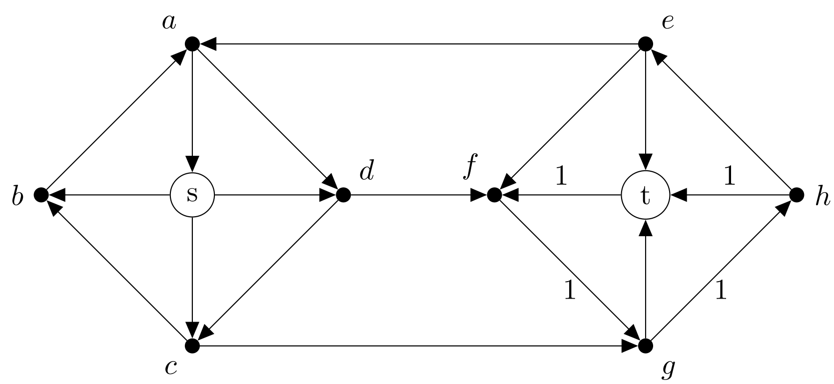
The first circuit is $C_1 = d \to c \to b \to a \to d$ with weight $w(C_1) = 2$.



The second circuit is $C_2 = a \to s \to d \to c \to g \to h \to e \to a$ with weight $w(C_2) = 1$.

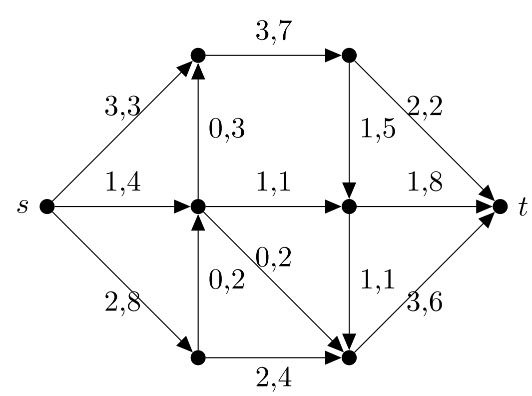


The third and final circuit is $C_3 = f \to g \to h \to t \to f$ with weight $w(C_3) = 1$.

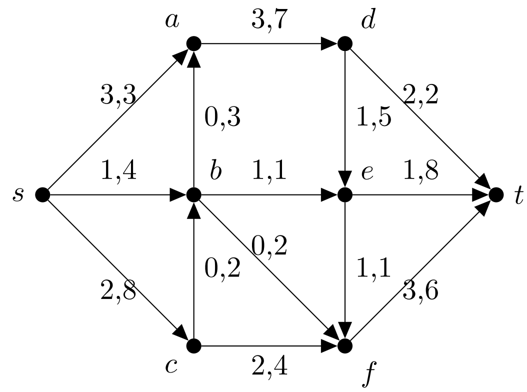


This set of paths and circuits with corresponding weight function satisfies all three of the properties stated in the Theorem. The sum of these paths and circuits results in the original flow. The sum of the weights on the paths is equal to the value of the flow. Also the number of paths and circuits is less than the number of edges of G.

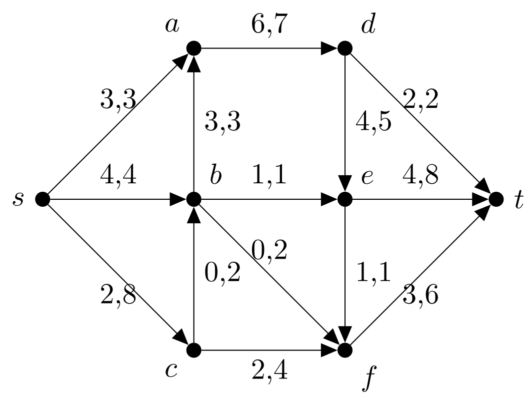
3. Consider the network below with given capacity and flow values. (The edge label f, u means flow-value f and capacity u.) Find augmenting paths and augment the flow to a maximum flow. Provide the list of residual graphs AND augmenting paths. In other words, run Ford-Fulkerson algorithm.



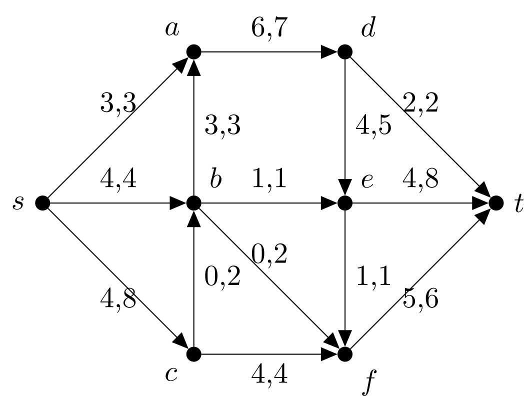
I will show the augmenting paths and the flow on the original graph however I have attached the residual graphs on a separate sheet of paper. First let me relabel the vertices.



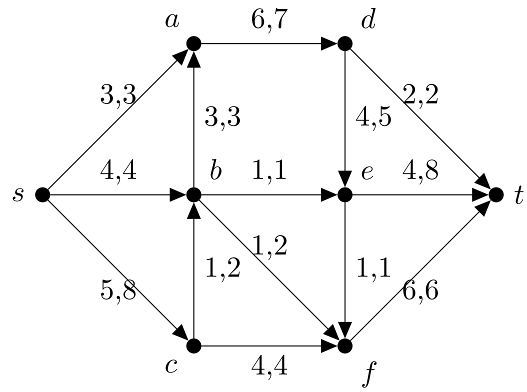
The initial augmenting path will be $P = s \to b \to a \to d \to e \to t$. The minimum capacity over this path is $\gamma = 3$. Augmenting on this path gives the following flow.



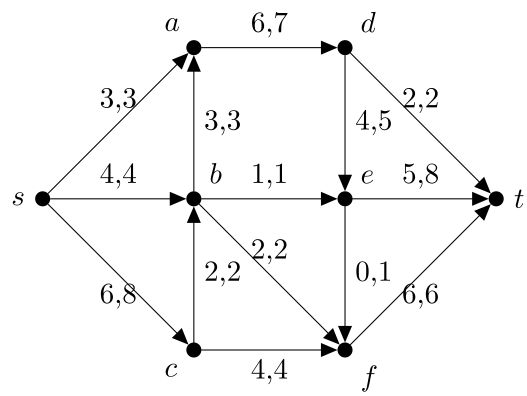
The second augmenting path I will use is $P = s \to c \to f \to t$ with a minimum capacity of $\gamma = 2$. The new flow will be



The third augmenting path is $P = s \to c \to b \to f \to t$ with minimum capacity, $\gamma = 1$. Augmenting along this path gives



The fourth augmenting path is $P = s \to c \to b \to f \to e \to t$ with minimum capacity, $\gamma = 1$. Note that this is decreasing the flow on $f \to e$.

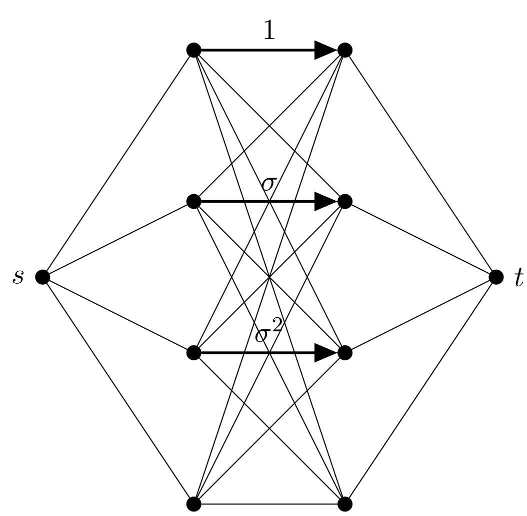


This is last augmenting path, and so this flow is optimal.

4. Let (G, u, s, t) be a network, and let $\delta^+(X)$ and $\delta^+(Y)$ be minimum s-t-cuts in (G, u). Show that $\delta^+(X \cap Y)$ and $\delta^+(X \cup Y)$ are also minimum s-t-cuts in (G, u, s, t).

   Proof. Let (G, u, s, t) be a network with edges E and vertices V. Let $X, Y \subset V$ such that $\delta^+(X)$ and $\delta^+(Y)$ are minimum s-t cuts. Consider $X \cap Y$ $\square$

5. Show that in case of irrational capacities, the Ford-Fulkerson algorithm may not terminate at all. Hint: See the Korte book (in particular exercises on page 199.). It contains the following network:



Where $\sigma = \frac{\sqrt{5}-1}{2}$. Note that σ satisfies $\sigma^n = \sigma^{n+1} + \sigma^{n+2}$. All other capacities are 1.

In order to show that the Ford-Fulkerson algorithm may not terminate it must be shown that there is an infinite sequence of augmenting paths.

6. Red-Blue meta algorithm for MST. Let G be a graph and w be a weight assignment to $E(G)$. Assume that all weights are distinct. Start with all edges being uncolored. Apply the following rules as long as possible.

   - if $e \in E$ is in a cycle C where e is the heaviest edge, color e red
   - if there is a cut where $e \in E$ is the lightest edge, color e blue.

   Claim is that blue edges form a minimum spanning tree.

   - Show that red edge cannot be in MST.
   - Show that blue edge must be in MST.
   - Show that blue edges form a tree
   - Show that every edge gets colored.
   - Show that no edge satisfies both red and blue criteria. (i.e. every edge has one color).

7. Implement Edmonds-Karp algorithm and run it on the network from question three. Print the sequence of augmenting paths used by your implementation. Print the flow and its value.

   I implemented the Edmonds-Karp algorithm in the following function.

```
def edmondsKarp(G, s, t):
    # Find mazimal flow on G from vertex s to vertex t
    # G weighted digraph - weights represent capacities
    # s - starting/source vertex
    # t - ending/target vertex

    # create residual graph as copy of original graph
    RG = G.copy()
    for e in G.edges():
        RG.add_edge(e[1], e[0], 0)

    path = shortestPath(RG, s, t)
    while path != None:
        path.reverse()
        print path
        min_capacity = min({e[2] for e in path})
        # augment flow
        for edge in path:
            RG.add_edge(edge[0], edge[1], edge[2] - min_capacity)
            RG.add_edge(edge[1], edge[0], RG.edge_label(edge[1], edge
                ↪ [0]) + min_capacity)
        path = shortestPath(RG, s, t)

    # uses dictionary to store flow
    # if e is edge in G, then f[e] is flow on e
    # intialize all to have 0 flow
    flow = dict()
```

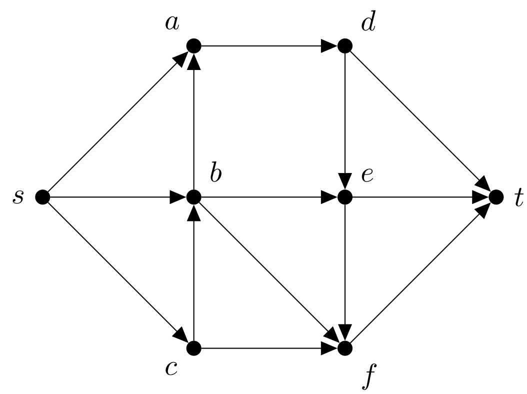
```
    for edge in G.edges():
        flow[edge] = RG.edge_label(edge[1], edge[0])

    return flow

def shortestPath(RG, source, target):
    # G is a graph
    # find the shortest path, P, from s to t or return None
    # shortest path in terms of least number of edges

    path = None
    # remove edges with 0 weight
    G = RG.copy()
    for edge in RG.edges():
        if edge[2] == 0:
            G.delete_edge(edge)
    tree = breadthFirstSearch(G, source)
    if tree.neighbors_in(target):
        path = []
        current_vertex = target
        while tree.neighbors_in(current_vertex):
            edge = tree.incoming_edges(current_vertex)[0]
            path.append(edge)
            current_vertex = edge[0]
    return path
```

This algorithm using a breadth first search which is implemented in the following function.

```
import Queue
def breadthFirstSearch(G, s):
    # G is a graph
    # s is the starting vertex
    # create empty tree
    T = DiGraph([G.vertices(),[]])
    R = {s}

    # create queue to hold nodes
    q = Queue.Queue()

    #distanceDict[s] = 0
    q.put(s)

    while not q.empty():
        currentVertex = q.get()
        # iterate over edges incident to currentVertex
        # if G is directed only includes edges going out from
            ↪ currentVertex
        # Don't use neighbors function different for directed and
            ↪ undirected graphs
        for e in G.edges_incident(currentVertex):
            adjacentVertex = e[1]
```

```
            # if we haven't reached adjacentVertex yet
            if adjacentVertex not in R:
                q.put(adjacentVertex)
                R.add(adjacentVertex)
                T.add_edge(e)
    return T
```

In order to run this algorithm on the graph from problem 3, I first relabeled the vertices in this graph. The graph was relabeled as shown below.




```
load('breadthFirstSearch.sage')
load('edmondsKarp.sage')
M = Matrix([[0, 3, 4, 8, 0, 0, 0, 0],
            [0, 0, 0, 0, 7, 0, 0, 0],
            [0, 3, 0, 0, 0, 1, 2, 0],
            [0, 0, 2, 0, 0, 0, 4, 0],
            [0, 0, 0, 0, 0, 5, 0, 2],
            [0, 0, 0, 0, 0, 0, 1, 8],
            [0, 0, 0, 0, 0, 0, 0, 6],
            [0, 0, 0, 0, 0, 0, 0, 0]])
G = DiGraph(M, weighted=True)
G.relabel({0:'s', 1:'a', 2:'b', 3:'c', 4:'d', 5:'e', 6:'f', 7:'t'})
s = 's'
t = 't'

flow = edmondsKarp(G, s, t)
print flow
```

This is the output of this script. Each list is the augmenting path. Each tuple is an edge in the augmenting path, with first entry the starting vertex, the second entry the ending vertex, and the third entry the available flow. The dictionary shows the flow on each edge in the form edge:flow.

```
[('s', 'a', 3), ('a', 'd', 7), ('d', 't', 2)]
[('s', 'b', 4), ('b', 'e', 1), ('e', 't', 8)]
[('s', 'b', 3), ('b', 'f', 2), ('f', 't', 6)]
[('s', 'c', 8), ('c', 'f', 4), ('f', 't', 4)]
[('s', 'a', 1), ('a', 'd', 5), ('d', 'e', 5), ('e', 't', 7)]
[('s', 'b', 1), ('b', 'a', 3), ('a', 'd', 4), ('d', 'e', 4), ('e', 't', 6)]
```

```
[('s', 'c', 4), ('c', 'b', 2), ('b', 'a', 2), ('a', 'd', 3),
  ('d', 'e', 3), ('e', 't', 5)]
{
  ('b', 'f', 2): 2,
  ('c', 'b', 2): 2,
  ('b', 'a', 3): 3,
  ('f', 't', 6): 6,
  ('s', 'b', 4): 4,
  ('e', 'f', 1): 0,
  ('a', 'd', 7): 6,
  ('s', 'c', 8): 6,
  ('d', 'e', 5): 4,
  ('s', 'a', 3): 3,
  ('b', 'e', 1): 1,
  ('c', 'f', 4): 4,
  ('d', 't', 2): 2,
  ('e', 't', 8): 5
  }
}
```

This flow can also be shown on the graph as follows.