Min-cut Max-flow algorithm visualizer

Students: John James P. Benitez Noa Hadad

Advisor: Dr. Elad Horev

September 4, 2019

§1. Introduction

A flow network N consists of a weighted directed graph G =(V,E). Each edge of the graph has a non-negative weight, which represents the capacity of the edge - the maximum flow that can be passed across the edge. The flow is from source s ϵ V to sink t ϵ V.
We can define 2 groups of vertices in the graph: V1 which includes vertex s and vertices S={v ϵ V|vertices that are reachable from s by path}, and V2 which includes vertex t and vertices T={v ϵ V|v $\notin$ V1}.
A cut is defined as the set of edges which connects between the 2 groups. The flow across a cut is the total flow of forward edges(from vϵV1 to vϵV2) minus the total flow of backward edges. The capacity of a cut is the total capacity of forward edges . Minimum cut is defined to be the cut with minimum capacity.
cut connect between the 2 "sides" of the network. Thus, the flow of each cut is equal to the other. Because the flow of a cut is less than or equals to its capacity- the capacity of the minimum cut represents the maximum flow.

§2. Abstract

The purpose of this programming project is to visualize the flow algorithms step by step. This project is a learning tool for students to understand the process of finding the maximum flow and it's correlation with the minimum cut.
The application lets the users to draw their own flow network and to choose which algorithm they would like to simulate: Dinitz's, Edmonds-Karp, Ford-Fulkerson . Then the users would see the simulation on their screens and would be given an option to pause, rewind, fast forward and stop the simulation.

§3. The classes behind the application

Edge - Class for keeping every edge's attributes: source vertex, destination vertex, color and capacity.

Vertex - Class for keeping every vertex's attributes: coordinates, place in the vector, character, if to show it or not, color, height, excess.

InputData- Class for keeping the data which the user provides: number of sources, number of vertices, vector for the edges, vector for the vertices, flag which indicates if the input has been changed.
It provides operations for adding/deleting the vertices/edges, getting reference to the vertices/edges and getting/setting the flag.
In the vertices' vector, the places '0' and '1' are reserved for the source and sink, and if there are more sources and sinks, they will be in the successive places.
The class is a singleton because the input needs to be saved in one place. In the MainActivity, the number of its sources and sinks is determined. In the DrawingActivity, edges and vertices are added/erased, and the OutputData's object gets its vertices' and edges' references. The GraphConstructor's object gets its edges' reference.

OutputData- Class for the data, edges and vertices, which the MyView class draws.
It has operation for setting reference to its edges and vertices, and operations for changing the vertices' attributes.
The class is a singleton for having one place to apply for setting/getting the presented data. In the DrawingActivity, its edges get a reference to the InputData's edges and vertices. In the AlgorithmActivity, according to the current step, its edges get a reference to the step's edges, and changes are set to its vertices' attributes. The MyView class draws the graph according to its data.

MyViewDrawingEdmondsKarpDinitz- Inherits from the class *View*. It has the OutputData's object, and it draws the graph according to the object's data.

MyViewPushRelabel- Inherits from class MyViewDrawingEdmondsKarpDinitz. It adds for every vertex - a draw of the height and excess.

GraphConstructor- It converts the InputData's edges' vector to a graph representation as an adjacency list. It creates 2 adjacency lists, one for the initial residual graph and one for the initial graph. It provides a copy of the needed graph to the Graph class and to the ResidualGraph class.
The class is a singleton because every algorithm has a graph and a residual graph. In order to prevent 3 times conversion- the conversion will take place only once when a new object is created.

AbstarctGraph- Abstract class for the 3 graphs: the graph, the residual graph and the level graph. It has an adjacency list which represents the graph, and operations of getting a specific edge, getting the neighbors of a specific vertex, getting vector of the edges' copies, and abstract operation for updating the edge (which is implemented in every successor).

ResidualGraph- Inherits from class AbstractGraph and implements the operation for updating the edge: the capacity of the current edge is reduced by the flow. If the capacity is now '0'- the edge is erased. If the opposite edge exists- the capacity is increased by the flow. If not- new edge is created with the flow as the capacity.

Graph- Inherits from class AbstractGraph and implements the operation for updating the edge: if the current edge exists- the flow's part is increased (using StringTokenizer). If the opposite edge exists- the flow's part is reduced.

LevelGraph- Inherits from class AbstractGraph. It has also attributes for every vertex: degree+, degree- and show (which is a reference to the show's attribute of class Dinitz).
It implements the operation for updating the edge: the capacity of the current edge is reduced by the flow. If it is now '0'- the edge is erased, and the degree- of the source vertex and the degree+ of the destination vertex are reduced by 1. For every degree, if it is now '0', the show flag of the vertex is turned off, and the operations for deleting forward/backward edges are called (these operations get vertex's number, find its connected edges, and send these edges and their capacity to the operation for updating the edge- in order to set their capacity to '0' and erase them).
It has also operation for finding if the graph is empty by checking the degree+ of vertex 't'.

AbstractAlgorithm- The class which implements the algorithm's phases and enables the user to go backward and forward through the algorithm's steps.
It has the residual graph, the graph, vector of the previous steps, number of the current step, flag for reaching the final step, data of the vertices' attributes, flags for stop running and instance of OutputData's object. It has operations for replacing the presented edges, updating the vertices' attributes, getting the next step, getting the previous step, saving the current step, running backward and forward, stop running and restarting. Its abstract operation for saving the current step is implemented by the successors, according to the type of the step.
Its operation for getting the next step is partly implemented, where the current step has been already saved. If it is a new step, the implementation is continuing by the successors.

GraphFunctions- The class consists the functions which are needed in order to implement the algorithm's phases. It has no attributes and all its functions are static. It was established in order to put all the operations on the graphs in one separated class.

'BFS' functions:- The function gets as an input a residual graph, number of vertices, and references for the father's and distance's attributes. It initializes every vetex's father and distance to -1, and generates inner variant for the vertices' colorBFS which is initialized to '0'(WHITE).
It generates a new empty queue. Takes the source vertex 's' (place 0) and sets its distance to '0', its color to '1'(GRAY), and puts it in the queue. It finds the neighbors of 's' and changes their distance to distance[0]+1 , color to '1', father to '0', and sets them in the queue.
It ends when 't' (1) will be added to the queue and the returned value will be distance[1], or when the queue will be empty and the returned value will be '0'.

'greedySearch' function-The function gets as an input a level graph and reference for the father's attribute. Because all the paths in the level graph are from 's' to 't', it gets the first neighbor of 's', set 's' (0) as its father, and continues this way until 't' (1) is reached.

'findPath' function- The function gets any successor of *AbstractGraph* and a reference for the father's attribute. It goes over the fathers until (-1) is

reached (the father of 's'). It adds the vertices in the path to a string (place 0 converts to 's' , 1 to 't' , i>1=>i-2+'A') ,marks the edges in the path, and finds the minimum capacity. It returns a Path's object with the flow and the path.

'updateGraph' function- The function gets the passing flow, reference for the father's attribute, and the graphs which need to be updated. Every edge in the path is sent to each graph's 'updateEdge' method.

'pushFromSource' function- The function gets a residual graph, vertices' size, and references for the height's and color's attributes.
For every vertex which is connected to 's', it checks if the height is '0' (there wasn't any push to the vertex) and sets the color of 's' and the founded vertex to 1 (to change their color on the view).
It returns a Push's object with the values: the capacity of the edge, sender and receiver. if there is no need to push- returns object with values of '0'.

'push' function- The function gets a residual graph, vertices' size and references for the height's, excess's and color's attributes.
It finds a vertex with excess, searches for a neighbor with its height minus 1, sets the color of both of them to 1, and returns a Push object with the vertices and the flow (the minimum between the excess and the capacity of the edge).

'findRelabeledVertices' function- The function gets a residual graph, vertices' size, and references for the heightToUpdate's and color's attributes.
For every vertex which has positive excess, it finds the minimum height of all the neighbors. If the vertex's height<min+1, it sets the color to 2, and adds him to the returned vector.

'findMinCut' function- The function gets a graph, vertices' size, a reference for the distance's attribute. For all the edges in the graph, it checks which edge is blocking- one side of the edge is reachable (distance!=-1), but the other is not. It marks these edges.

Push- Keeps the data of the pushing functions: sender, receiver and flow.

Path- Keeps the data of the 'findPath' function: path and flow.

EdmondsKarp-The next operation is implemented by using the variable 'whichPhase' that indicates to which of these 4 phases to call:

1) findingImprovementPathPhase
   - Calls to 'BFS' function.
     If it returns '0', calls to 'findingNoPathPhase'.
     Otherwise:
     - Calls to 'findPath' function.
     - Updates the affected fields.
     - Sets 2 to 'whichPhase'.
2) updatingResidualGraphPhase
   - Calls to 'updateGraph' function.
   - Updates the affected fields.
   - Sets 1 to 'whichPhase'.
3) findingNoPathPhase (updating the user that there are no more paths)
   - Updates the affected fields.
   - Sets 4 to 'whichPhase'.
4) endingOfAlgoPhase
   - Calls to 'minCut' function.
   - Updates the affected fields.
   - Turns on the end flag.

Dinitz- It has also level graph and data if to show the vertices.
The next operation is implemented by using variable 'whichPhase' that indicates to which of these 6 phases to call:

1) findingNewLevelPhase
   - Calls to 'BFS' function.
     If it returns '0', calls to 'findingNoNewLevelPhase'.
     Otherwise:
     - Updates the affected fields.
     - Sets 2 to 'whichPhase'.
2) buildingLevelGraphPhase
   - Constructs a level graph from the results of the 'BFS'.
   - Updates the affected fields.
   - Sets 3 to 'whichPhase'.
3) findingPathPhase
   - Calls the method 'isEmpty' of the level graph.
     If it is-calls to 'findingNewLevelPhase'
     Otherwise:
     - Calls 'findPath' function.
     - Updates the affected fields.
     - Sets 4 to 'whichPhase'.

4) updatingGraphPhase
- Calls 'updateGraph' function.
- Updates the affected fields.
- Sets 3 to 'whichPhase'.

5) findingNoNewLevelPhase
- Updates the affected fields.
- Sets 5 to 'whichPhase'.

6) endingOfAlgoPhase
- Calls 'minCut' function.
- Updates the affected fields.
- Turns on the end flag.

PushRelabel- It has also the sender and receiver vertices , flag that indicates if to call the 'pushFromSource' function, and data about the vertices: the height, excess and color.
The next operation is implemented by using variable 'whichPhase' that indicates to which of these 6 phases to call:
1) findingPushPhase
- If 'pushFromSourceFlag' is on, it calls the function 'pushFromSource'. If object with '0' values is returned, the 'pushFromSourceFlag' is set to 'off'.
- If 'pushFromSourceFlag' is off, it calls the 'push' function.
- Updates the affected fields.
- Sets 2 to 'whichPhase'

2) updatingPushPhase
- Calls the 'updateEdge' method of the residual graph and the graph.
- Updates the affected fields.
- Sets 3 to 'whichPhase'.

3) findingRelabeledVerticesPhase
- Checks if all the vertices are not active- if they are, calls to 'noMorePushPhase' .
  Otherwise-
  - Calls the function 'findRelabeledVertices'
  - Updates the affected fields
  - Sets 4 to 'whichPhase'.

4) updatingRelabelPhase

- Changes the height presented in the view.
- Updates the affected fields.
- Sets 1 to 'whichPhase'.

5) noMorePushPhase

- Updates the affected fields.
- Sets 5 to 'whichPhase'.

6) endingOfAlgoPhase

- Calls 'BFS' function.
- Calls 'minCut' function.
- Updates the affected fields.
- Turns on the end flag.

Every one of the algorithms is a singleton, because we need only one object of them.

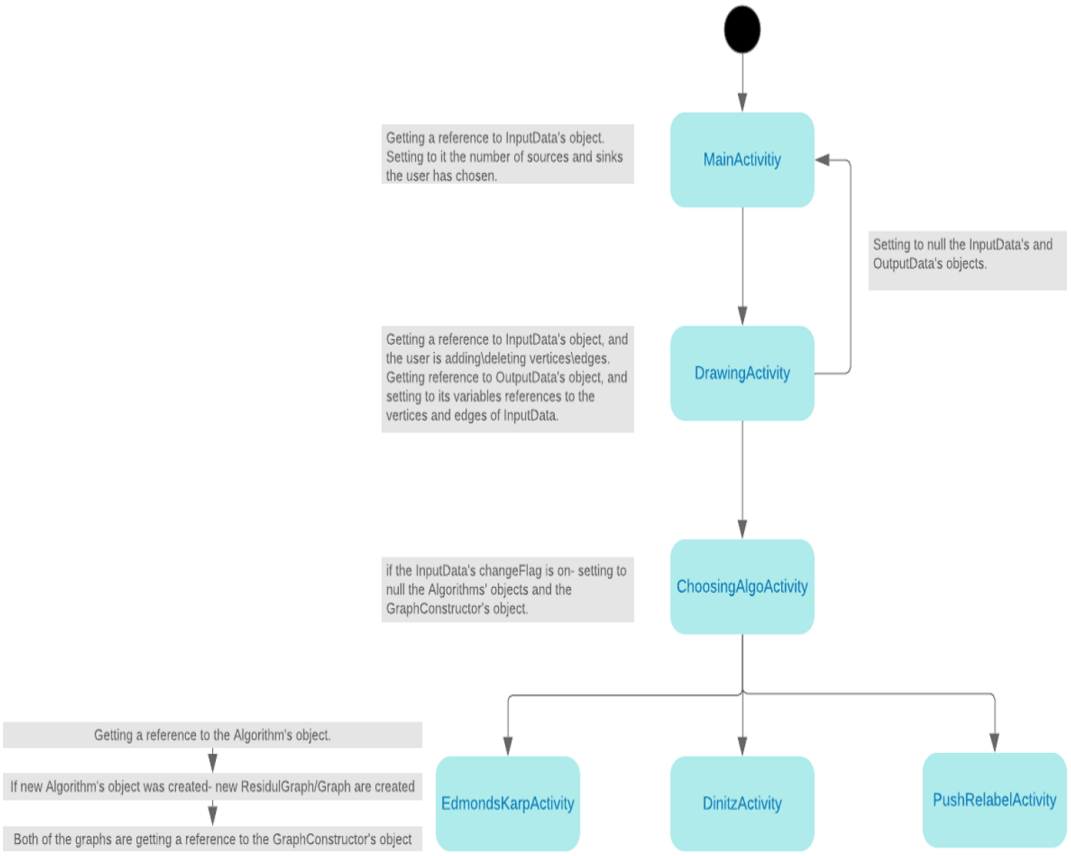
EdmondsKarpStep- Keeps the data which needs to be presented for every step in Edmonds-Karp algorithm- edges of the residual graph, edges of the graph, which graph to present, string of the operation.

DinitzStep- Inherits from EdmondsKarpStep, and also keeping the edges of the level graph and data of the vertices need to be shown where the level graph is presented.

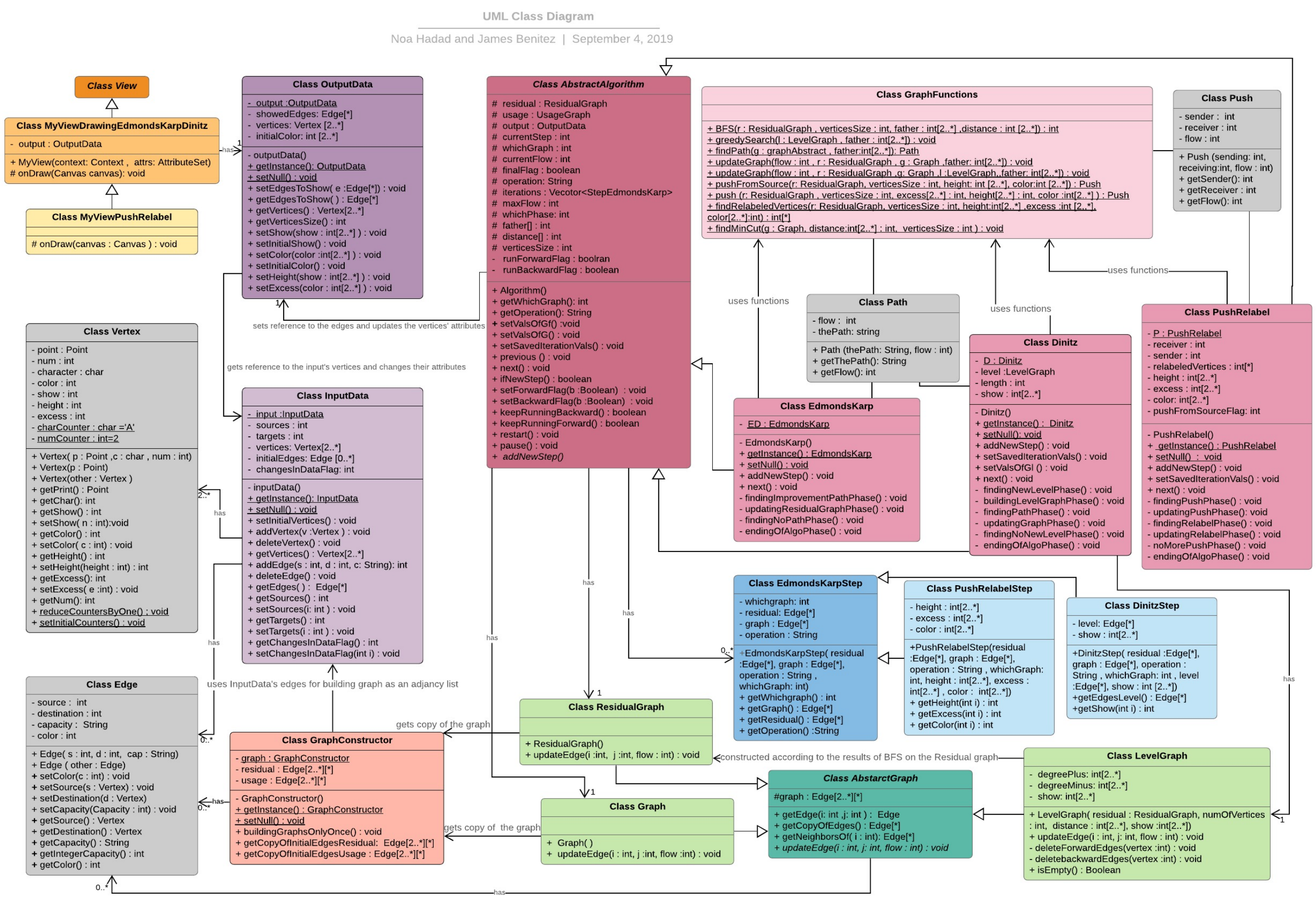
PushRelabelStep- Inherits from EdmondsKarpStep, and keeping data of the height, excess and color of the vertices.

§4. Going backward and forward through the activities

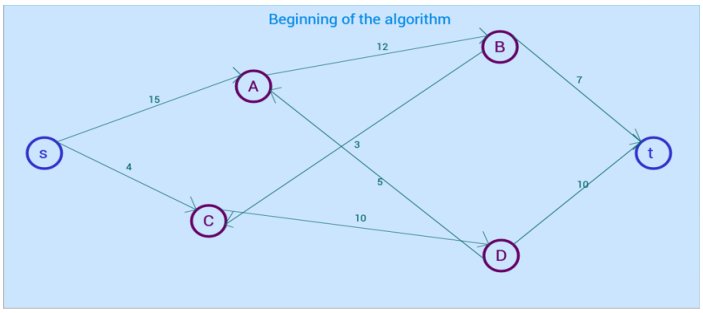
In order to enable the user to create new graph or to change the graph he has drawn, when the user goes backward and forward the activities, some of the singleton's objects are set to null, like how it is presented in the diagram:

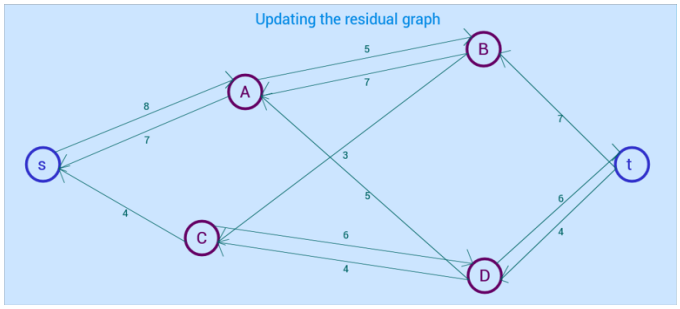


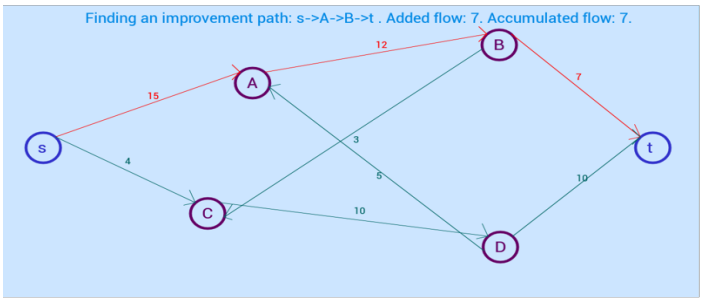
§5. Class diagram

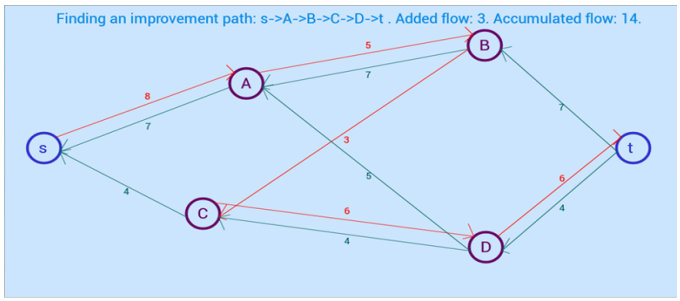


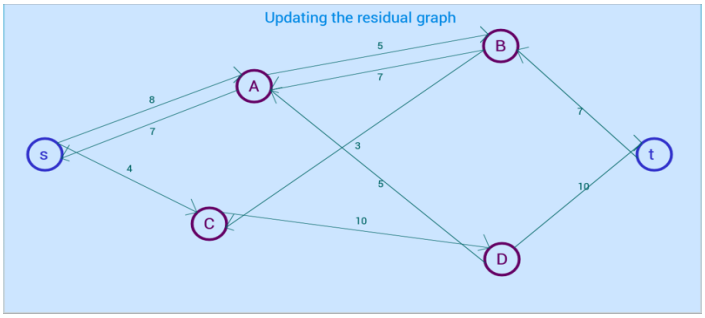
§6. Edmonds-Karp's simulation

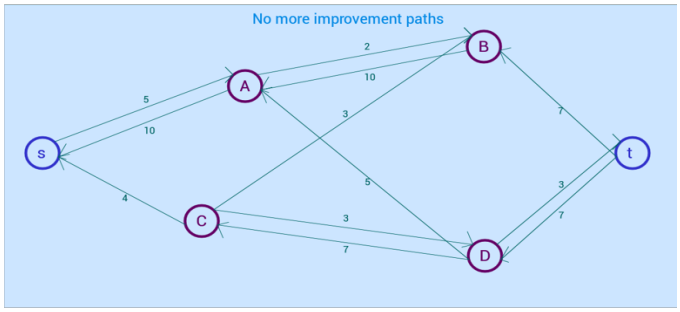


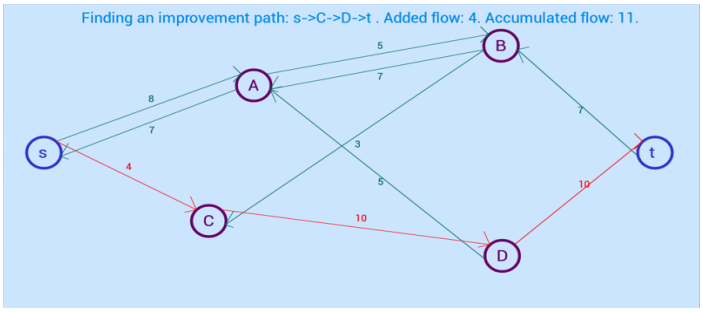


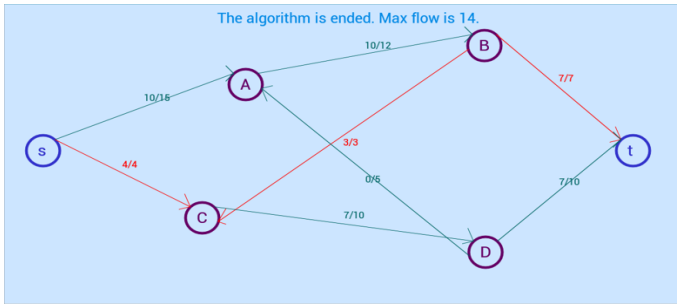




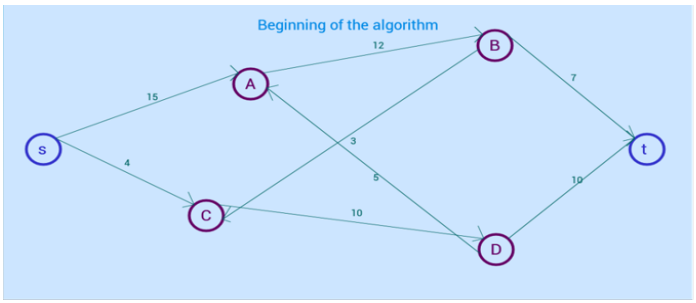


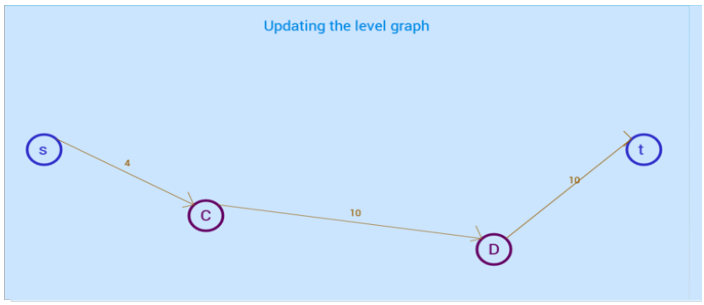


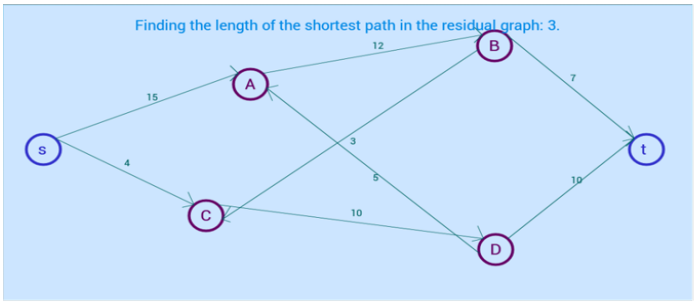


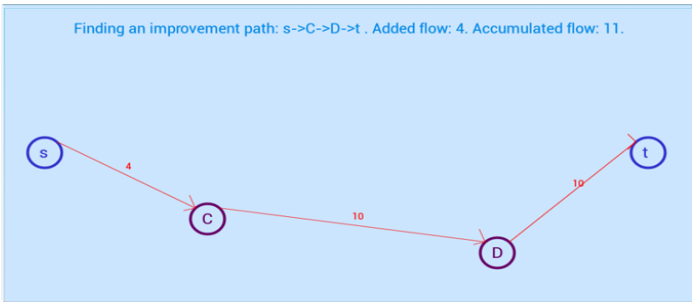


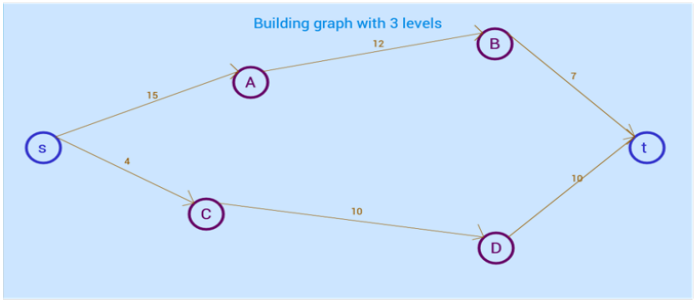
§7. Dinitz's simulation

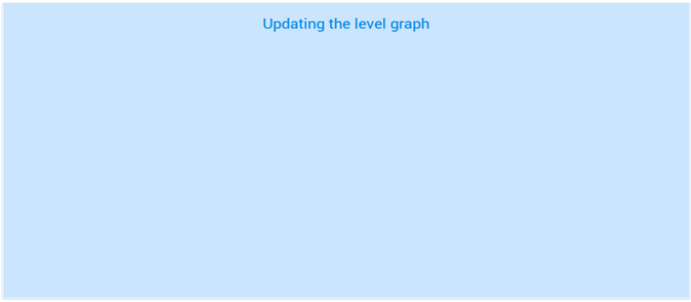


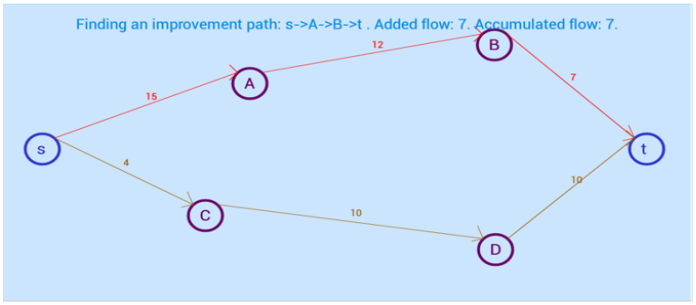


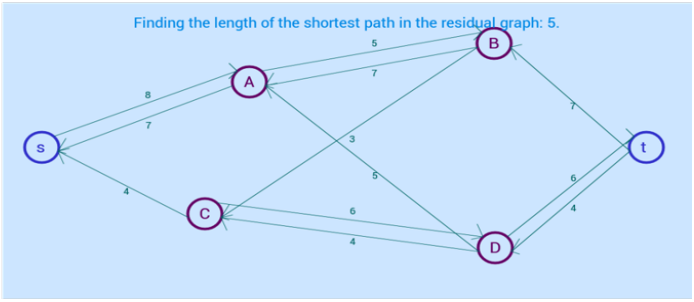


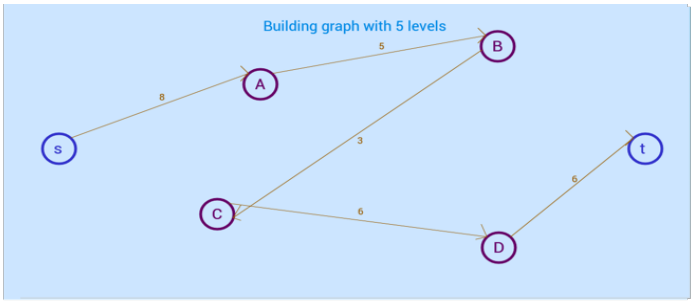


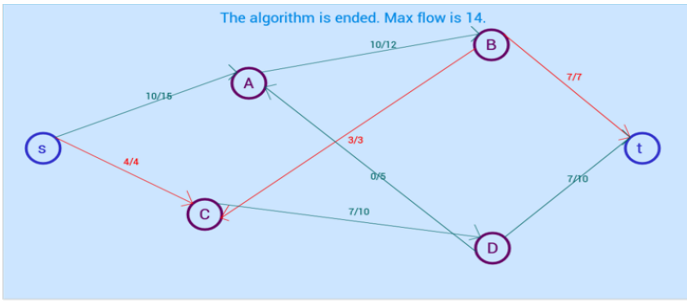


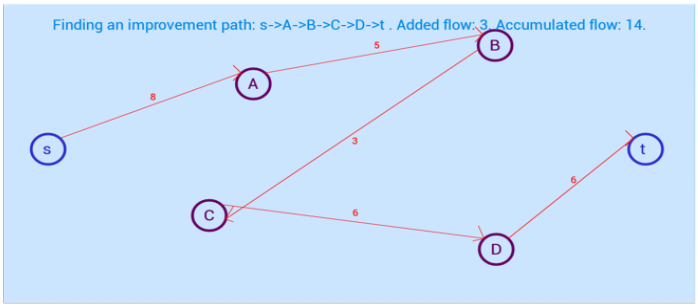


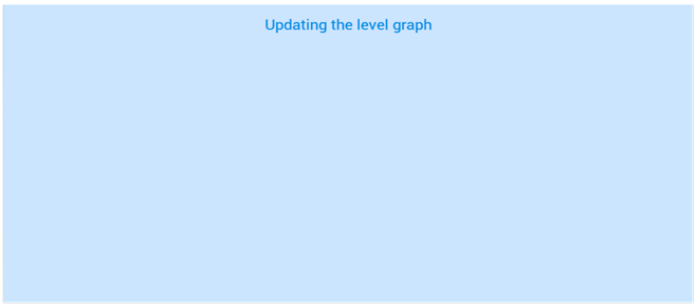


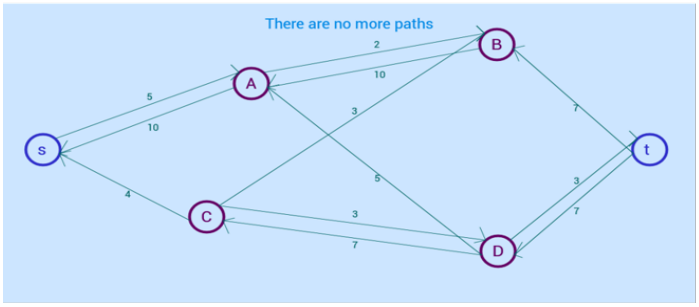



Building graph with 5 levels
B
5
A
8
s
3
t
6
C
6
D

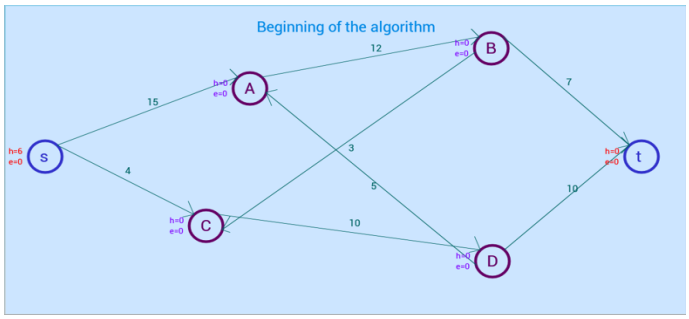

The algorithm is ended. Max flow is 14.
10/12
B
A
7/7
10/15
s
3/3
t
4/4
0/5
7/10
C
7/10
D

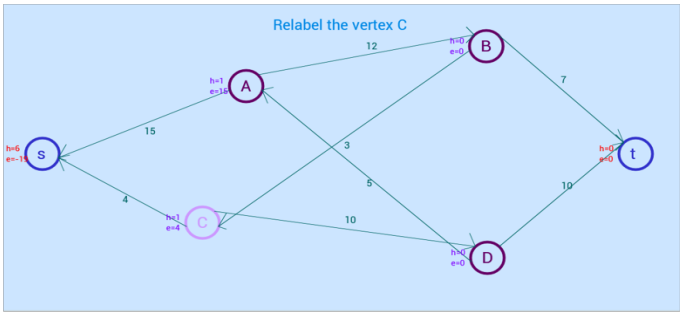

Finding an improvement path: s->A->B->C->D->t . Added flow: 3. Accumulated flow: 14.
B
5
A
8
s
3
t
6
C
6
D

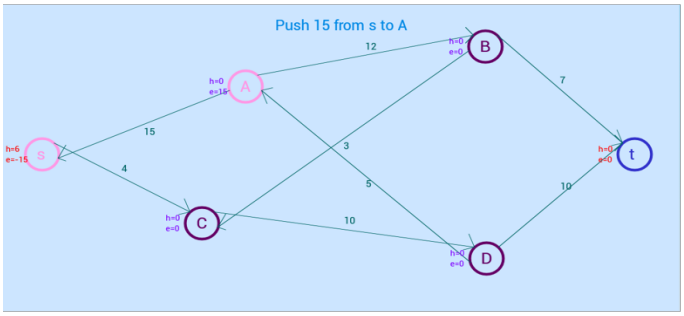

Updating the level graph

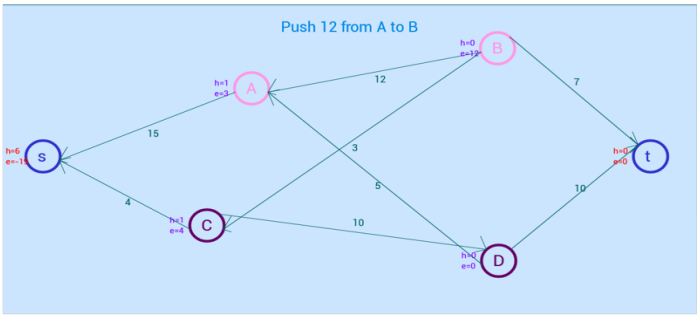

There are no more paths
2
B
A
10
5
7
10
3
s
t
5
3
4
C
3
7
7
D

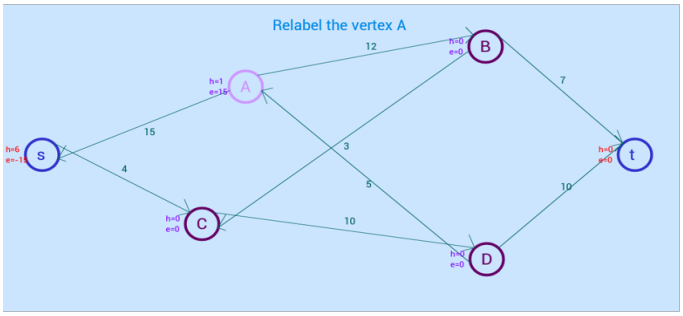
§8. Push-Relabel's simulation

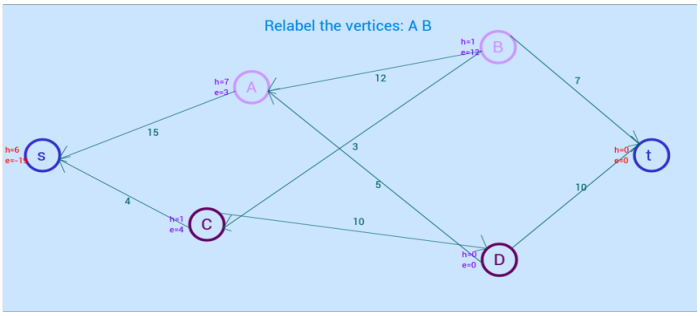


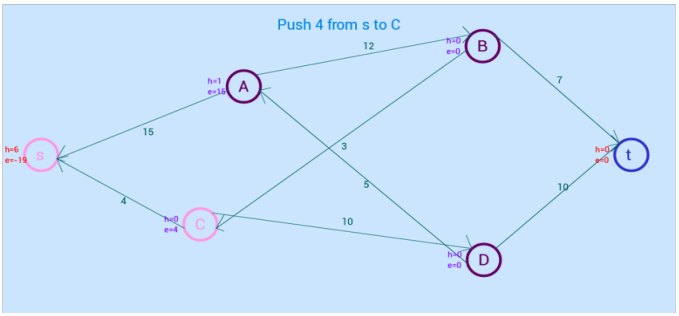


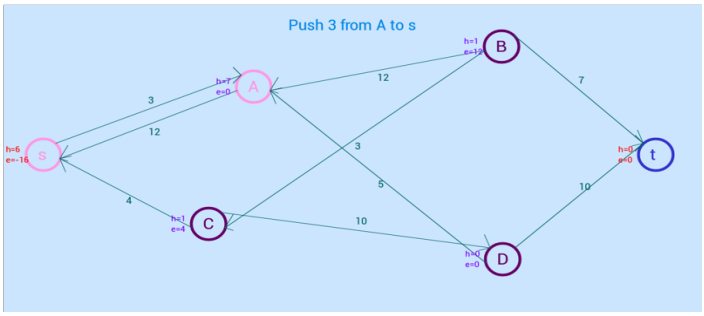


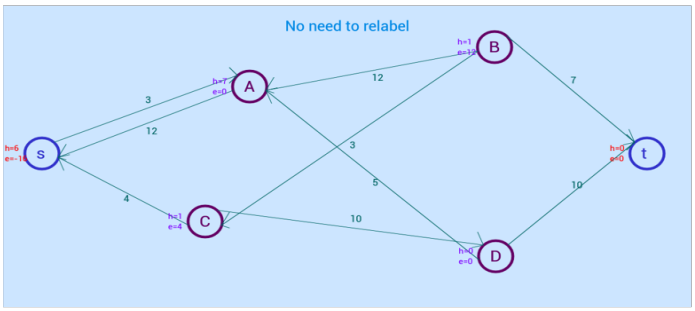


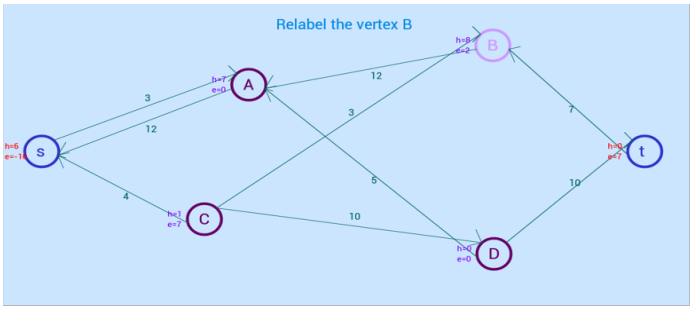


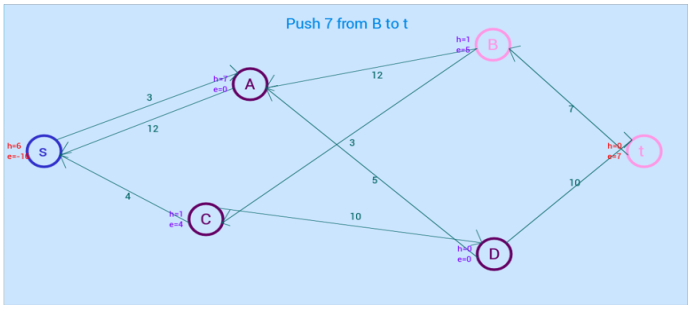


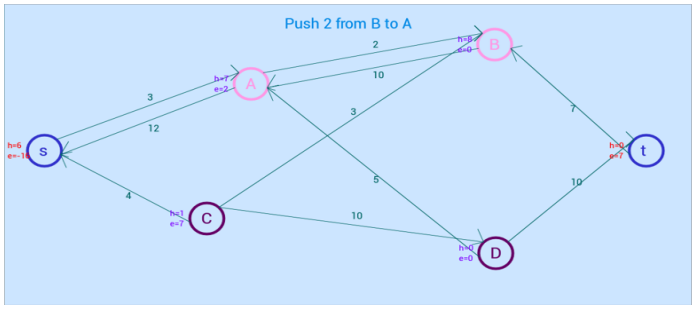


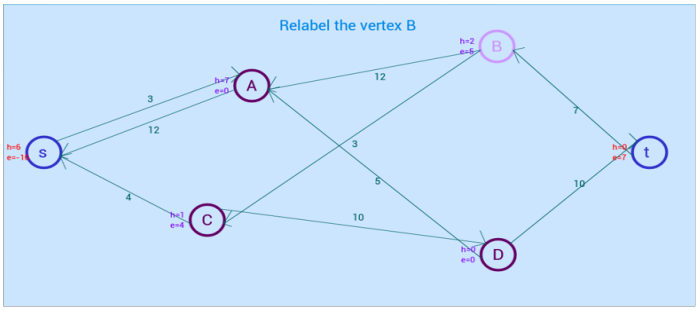


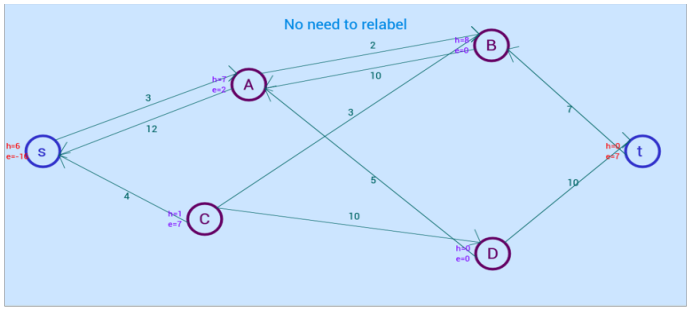

No need to relabel
s
A
B
C
D
t
3
12
12
4
3
5
7
10
10

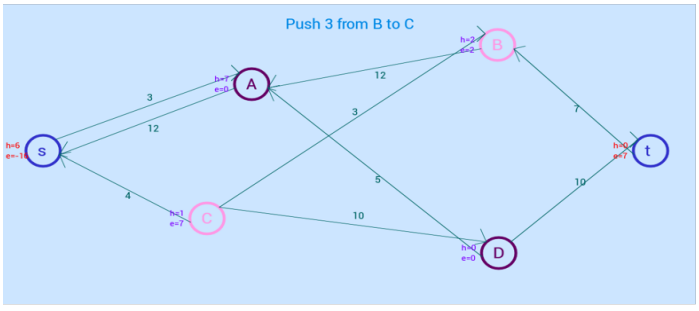

Relabel the vertex B
s
A
B
C
D
t
3
12
12
4
3
5
7
10
10

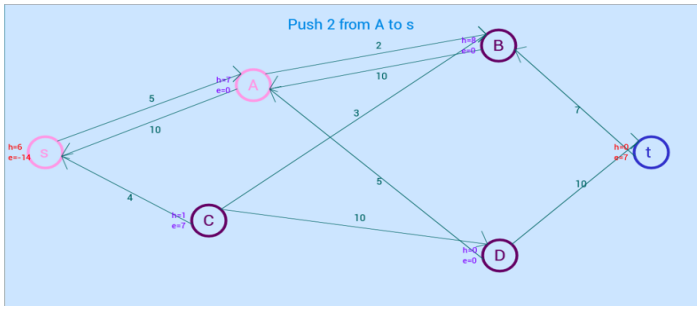

Push 7 from B to t
s
A
B
C
D
t
3
12
12
4
3
5
7
10
10

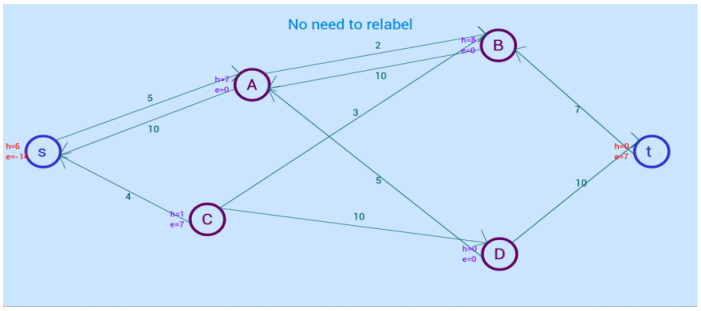

Push 2 from B to A
s
A
B
C
D
t
2
10
3
12
4
3
5
7
10
10

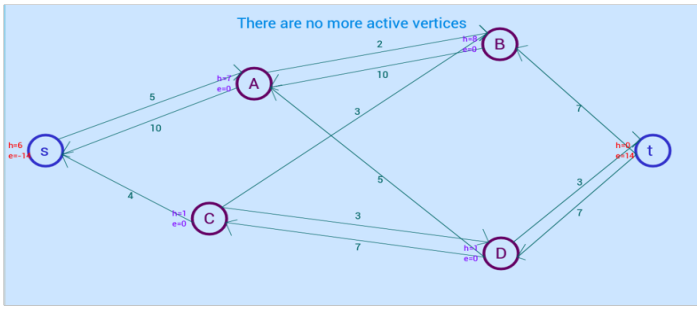

Relabel the vertex B
s
A
B
C
D
t
3
12
12
4
3
5
7
10
10

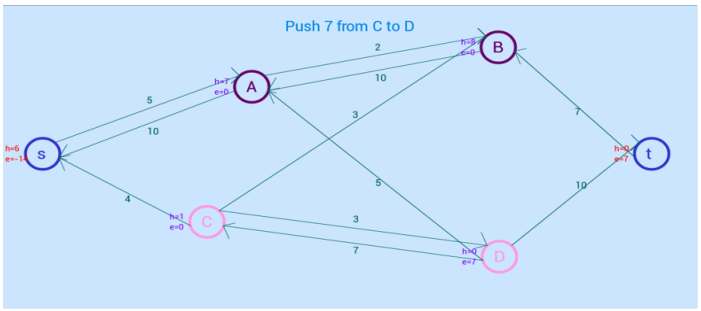

No need to relabel
s
A
B
C
D
t
2
10
3
12
4
3
5
7
10
10

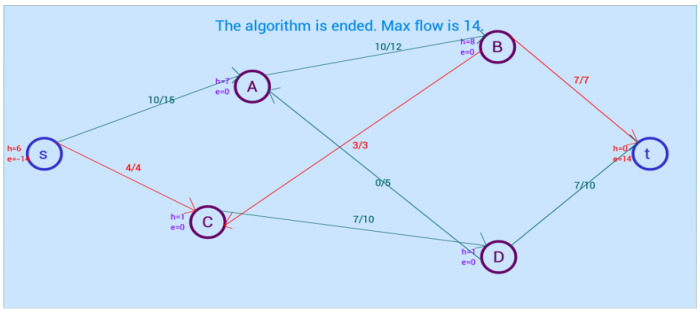

Push 3 from B to C
s
A
B
C
D
t
3
12
12
4
3
5
7
10
10

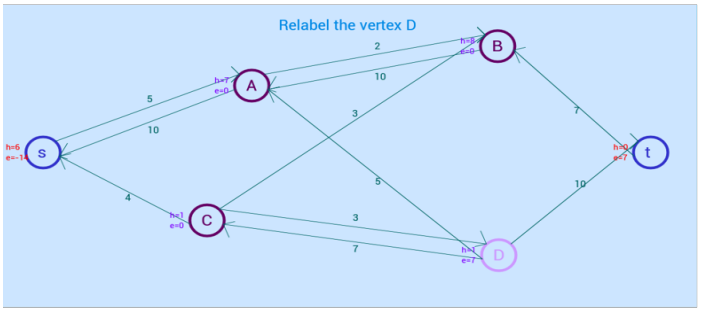

Push 2 from A to s
s
A
B
C
D
t
5
10
2
10
3
4
5
7
10
10

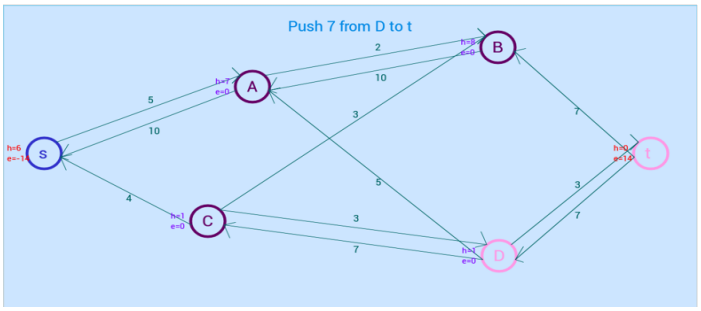

No need to relabel
s
A
B
C
D
t


There are no more active vertices
s
A
B
C
D
t


Push 7 from C to D
s
A
B
C
D
t


The algorithm is ended. Max flow is 14.
s
A
B
C
D
t


Relabel the vertex D
s
A
B
C
D
t


Push 7 from D to t
s
A
B
C
D
t