

Server Allocation using Dinic's Algorithm

Yonatan Viody 13518120
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
yonatanviody@gmail.com

Abstract—Nowadays, the usage of servers are everywhere. For companies that have many customers, the need for servers is crucial and sometimes is not fulfilled because the demand is higher than the resources' availability. We need to find a better way to allocate our servers to give the maximum throughput. In this paper, one method for server allocation will be discussed as a max flow problem which can be solved by using Dinic's algorithm.

Keywords—server allocation; max flow; Dinic's algorithm;

I. INTRODUCTION

The COVID-19 outbreak, that's happening right now, is changing everything. Before COVID-19 appeared, people liked to go hang out, shop, work, and play together. But now, people need to maintain a distance between each other for their safety. To reduce the spreading rate of COVID-19, some countries are doing lockdown and social distancing. Work, study, and some other activities are disturbed by this outbreak. Everything depends on online applications now and that's a fact.

Because of social distancing, people intend to use online applications more often. Some of them are online meeting applications. Online meeting applications can be considered as server intensive applications. That's why online meeting applications require servers, not just two or centralized in one country, but many and distributed in multiple countries for better experiences. For example, when we are using google meet, other people are also using them, but we don't feel any differences. That's because your meeting and other meetings are allocated on different servers. And not just online meetings, online games are also using many servers so there is no significant delay between user actions.

There are many ways on how to allocate servers. Commonly, servers are allocated using 2 approaches, dynamic and static. Both of them are efficient in a particular situation. We will be discussing both server allocation in this paper. Static server allocation is preferred because of many reasons. Here is an example to see the reason why static server allocation is much preferred than dynamic.

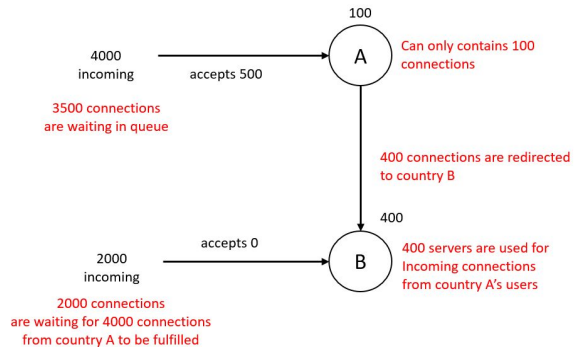


Image 1 Simple dynamic server allocation

There are 2 countries from the example above, which are A and B. Country A has 100 servers and 4000 incoming connections. Country B has 400 servers and 2000 incoming connections. Let's assume that country A's incoming connections happen before country B. So if we allocate servers dynamically, we can say that 100 servers of country A and 400 servers of country B are allocated for 500 incoming connections from country A and we need to wait for 3500 more incoming connections to be fulfilled. You can see that country B's incoming connections are waiting too long for them to be fulfilled. And what if we used a time-shared approach or prioritized the connections that are waiting too long or something else? These approaches are not giving us any optimal solution and are expensive, so why do we even bother to consider them? We can consider using both approaches at the same time, but how to allocate them? Therefore, in this paper, the writer wants to find the numbers of servers needed for static and dynamic allocation to get the best result.

II. THEORETICAL FRAMEWORK

A. Graph

A graph is a representation of discrete objects and their connections. A graph consists of vertices and edges. Vertices represent discrete objects and edges represent their connections.

Graphs can be divided into two types based on the direction orientation on edge, which are:

a. Undirected graph

Graphs, whose edges don't have direction orientation, are undirected graphs.

b. Directed graph

Graphs, whose edges have direction orientation, are directed graphs.

Here is an example of an undirected graph and a directed graph.

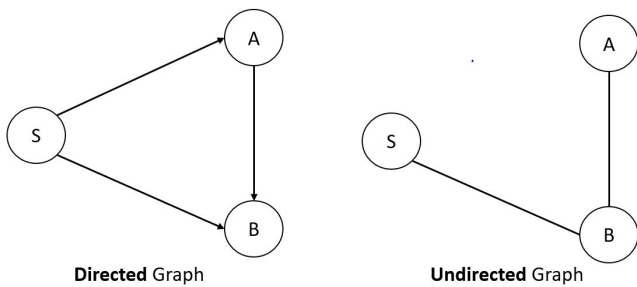


Image 2 Example of directed and undirected graph

B. Breadth-First Search Algorithm

Breadth-first search (BFS) algorithm is an algorithm for traversing graph data structures. BFS algorithm visits all adjacent vertices from the current vertex first and repeatedly visits the new adjacent vertices from the old adjacent vertices after all the old adjacent vertices are visited, until it arrives at the goal vertex or visited all vertices. BFS algorithm is implemented using a queue because the first vertex that arrives is the first vertex that is served by the algorithm. Commonly, the BFS algorithm steps from vertex s are as follows:

1. Visit vertex s
2. Visit all adjacent vertices from vertex s
3. Visit unvisited vertices that are adjacent from vertices before
4. Repeat until all vertices are visited

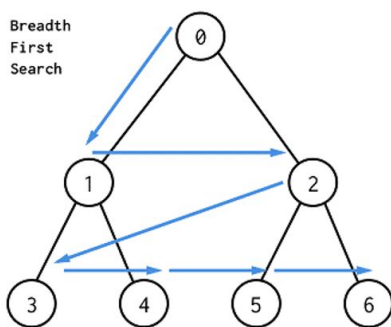


Image 3 Example of BFS (Source: <https://www.freelancinggig.com/blog/wp-content/uploads/2019/02/BFS-and-DFS-Algorithms.png>)

C. Depth-First Search Algorithm

Depth-first search (DFS) algorithm is also an algorithm for traversing graph data structures. DFS algorithm explores the most recently discovered vertex and leaves some discovered vertices behind. DFS algorithm is implemented using a stack, because the last discovered vertex is the first vertex that is served by the algorithm. Commonly, the DFS algorithm steps from vertex s are as follows:

1. Visit vertex s
2. Visit vertex a that is adjacent from vertex s
3. Repeat step 2 for vertex a and so on.
4. If there is no unvisited vertex, backtrack to the latest vertex and repeat step 2.
5. Stop until all vertices are visited.

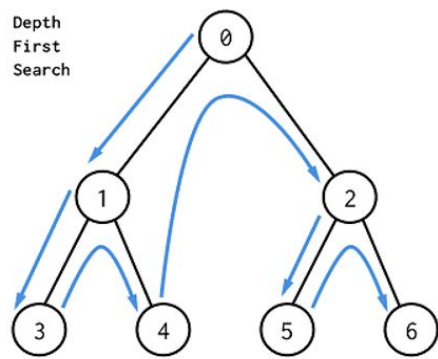


Image 4 Example of DFS (Source: <https://www.freelancinggig.com/blog/wp-content/uploads/2019/02/BFS-and-DFS-Algorithms.png>)

D. Max-flow Problem

Max-flow or maximum flow problem is a problem where we wish to compute the greatest rate at which we can transfer flow from a source to a sink. This problem can be solved by some algorithms, including Dinic's algorithm. The idea of this problem is that an edge is like a pipe that has a capacity and a flow that can't be higher than its capacity. For example, we have a flow network graph given below.

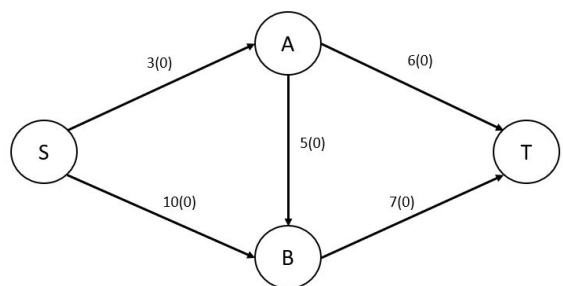


Image 5 Example of a flow network

From the flow network, we can see numbers that represent the capacity and the flow on a certain edge. Initially, there is no flow in all edges. We want to find the maximum rate of flow that can be sent from the source and received by the sink. We can see that the answer is 10, shown in the graph below.

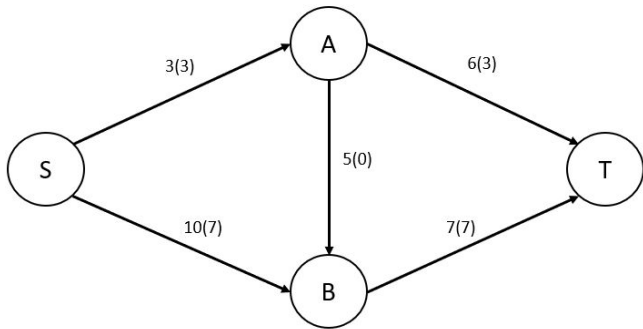


Image 6 Maximum flow in the flow network

E. Level Graph

A level graph is a graph where the vertices are marked by a certain number that indicates its level. The idea is that a level 3 vertex can only be accessed from level 2 vertices. This way, we can use a level graph to show us the augmenting paths from the source to the sink in the flow network. Level graphs can be constructed using BFS algorithm because a vertex's level is the shortest distance from the source vertex to the vertex, which can be computed using BFS.

F. Blocking Flow

A blocking flow is a condition when every path from source to sink contains a saturated edge or more. Saturated edges are edges in a flow network graph whose flow values are equal to their capacities.

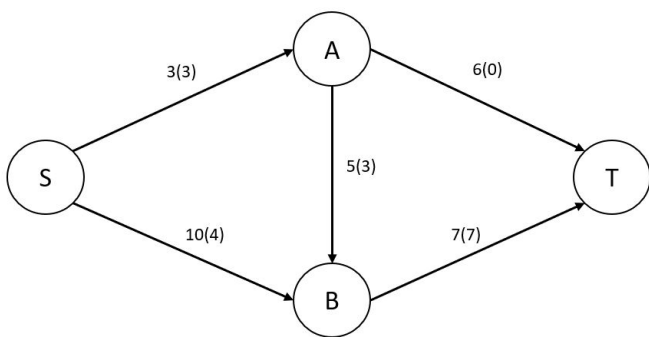


Image 7 A flow network graph

For example, given a flow network graph as above, we can see that there are 2 saturated edges, which are edge s-a and b-t, and 3 paths from source to sink, which are s-a-t, s-b-t,

and s-a-b-t, that have a saturated edge or more. Therefore we can say that a blocking flow has occurred.

G. Dinic's Algorithm

Dinic's algorithm is one of the algorithms that are used to solve max-flow problems. The general approach of Dinic's algorithm is for a flow network graph G with n vertices and m edges, repeatedly finds a blocking flow and effectively increases flow along all paths from the source to the sink (augmenting paths) for the corresponding level graph simultaneously. Dinic's algorithm steps are as follows:

1. Construct a level graph from the source to the sink.
2. Check if an augmenting path exists or not. If not, stop.
3. Compute a blocking flow by sending flow using our level graph.
4. Repeat from step 1 until there is no augmenting path left.

Dinic's algorithm time complexity is $O(EV^2)$, where $O(VE)$ is the time needed to compute a blocking flow and the constructions of the level graph could reach E-1 times.

H. Server Allocation

As discussed before, there are 2 approaches for server allocation based on the time of allocation, static and dynamic. Static server allocation happens before the runtime, where a certain number of servers has already been assigned to an area. Dynamic server allocation happens at the runtime, where a certain number of servers can be allocated for an area if there is a request. There is another approach for server allocation, the hybrid between static and dynamic, where we allocated a certain number of servers to an area, but spared some of the servers for dynamic server allocation.

III. PROBLEM ANALYSIS

A. Decomposition

Server allocation can be represented as a flow network graph. We can represent each country as a vertex. Each country has incoming connections and servers. Incoming connections from a country are represented by the edge from the source to the country. The maximum servers that can be allocated from a country are represented by the edge from the country to the sink. A country can use other countries' servers if they allow the country to use their servers. This can be represented by the edge from one country to another. We will need to add 2 additional nodes, the source (as World) and the sink (as Main Server).

The answers we seek are the amount of static and dynamic allocation, and how we allocate the servers statically. The amount of static and dynamic allocation can be shown by

simply print out the output, but we need a visualization to show how we allocate the servers statically

B. Planning

We will use python programming language to solve this problem. To implement Dinic's algorithm, we will need 2 components, which are level graph constructor and flow transmitter. We want to construct a level graph using BFS algorithm and send a flow using DFS algorithm based on the level graph. For level graph construction, we just want to get the shortest augmenting paths, so we will stop until it reaches the sink. For flow sending, we will just DFS each augmenting path and make a reverse path to provide a possibility to retract flow in an edge.

For our visualization, we can use matplotlib and networkx library. What we want is to visualize a flow network, which can be represented using a weighted directed graph.

IV. IMPLEMENTATION

A. Graph Representation

For the implementation, the graph representation, that is used, is a list of nodes that contain adjacent nodes' names and their edges' capacities given below. The adjacent data is represented using a dictionary with names as keys and capacities as values.

```
class Node:
    def __init__(self, name):
        self.name = name
        self.adj = {}

    def addEdge(self, name, cap):
        self.adj[name] = cap
```

B. Level Graph Constructor

For constructing the level graph, we can use BFS algorithm given below. We will assign the level based on the depth from the source node and stop the BFS until it reaches our destination node. If a node is already given a level value or its edge capacity is 0, we don't want to visit it at all.

```
# BFS for setup level
def bfs(source, dest, nodes):
    q = Queue()
    q.put(source)
    level = {source: 0}
```

```
visited = {}
while (not q.empty()):
    name = q.get()
    if (name == dest):
        break
    if (name not in visited):
        visited[name] = True
        for x, cap in
nodes[name].adj.items():
            if (x not in level and cap >
0):
                level[x] = level[name] +
1
                q.put(x)
return (name == dest, level)
```

C. Flow Transmitter

For sending the flow, we can use DFS algorithm given below. We will do DFS based on our level graph because the level graph represents our augmenting paths in our flow network. By sending a flow, we can retract it, so we need to construct reverse edges with the sent flow value as their capacities.

```
# DFS for sending flow
def dfsFlow(source, dest, level, depth,
flow, nodes):
    if (source == dest):
        return flow
    sum = 0
    for name, cap in
nodes[source].adj.items():
        if (name in level and level[name] ==
depth+1):
            t_flow = min(cap, flow)
            t_flow = dfsFlow(name, dest,
level, depth+1, t_flow, nodes)
            nodes[source].adj[name] -= t_flow
            if (source not in
nodes[name].adj):
                nodes[name].adj[source] =
t_flow
            else:
                nodes[name].adj[source] +=
```

```
t_flow
    sum += t_flow
return sum
```

D. Dinic's Algorithm

Here is the implementation of our Dinic's algorithm. We want to construct a level graph and see if it's possible to send a flow. If it's possible, we just send flow simultaneously using DFS. After there is no possible flow transfer left, we return the max flow value that can be received by our destination.

```
# Dinic's Algorithm
def dinic(source, dest, nodes):
    possible, level = bfs(source, dest, nodes)
    flow = 0
    while (possible):
        temp = dfsFlow(source, dest, level, 0, sys.maxsize, nodes)
        if (temp <= 0):
            break
        flow += temp
        possible, level = bfs(source, dest, nodes)
    return flow
```

E. Main Program

Our main program will be the input handler and construct the flow network graph based on the input. Don't forget to add two additional nodes (source and sink) to the graph. After the graph is ready, we can call our Dinic's algorithm implementation and print out the result. And lastly, we visualize the problem and the solution graph using our visualizer as images. Full implementation can be seen in the GitHub's link.

V. TESTING

A. Test Cases

- Test Case 1

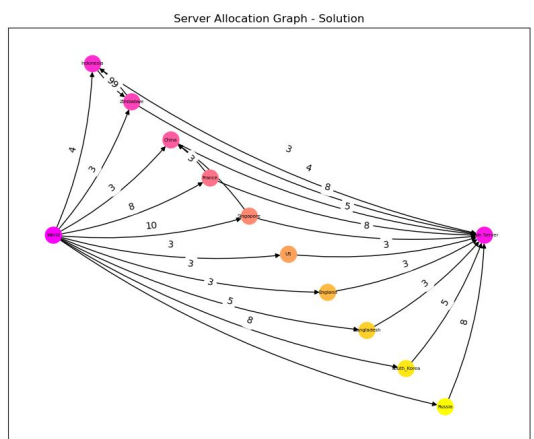
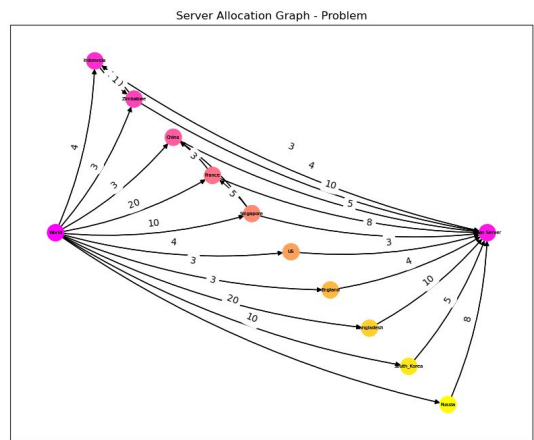
```
10
Indonesia 3 4
Zimbabwe 4 3
China 10 3
France 5 20
```

```
Singapore 8 10
US 3 4
England 4 3
Bangladesh 10 3
South Korea 5 20
Russia 8 10
5
Zimbabwe Indonesia 1
Singapore France 5
France China 3
Singapore China 4
Indonesia Singapore 100
```

- Expected :
- 50 static allocation
 - 10 dynamic allocation

Result :

```
The number of servers allocated statically: 50
The number of servers allocated dynamically: 10
```



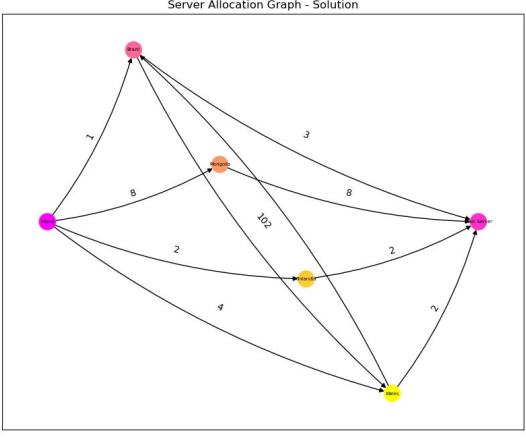
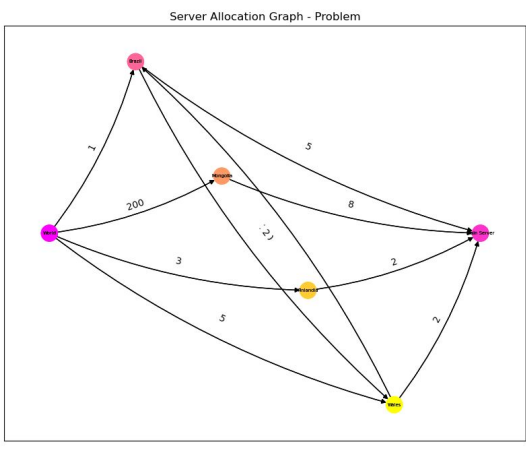
- Test Case 2

```
3
Brazil 5 1
```

Mongolia 8 200
 Wales 2 5
 2
 Brazil Wales 100
 Wales Brazil 2

Expected :
 ○ 15 static allocation
 ○ 2 dynamic allocation

Result :
 The number of servers allocated statically: 15
 The number of servers allocated dynamically: 2



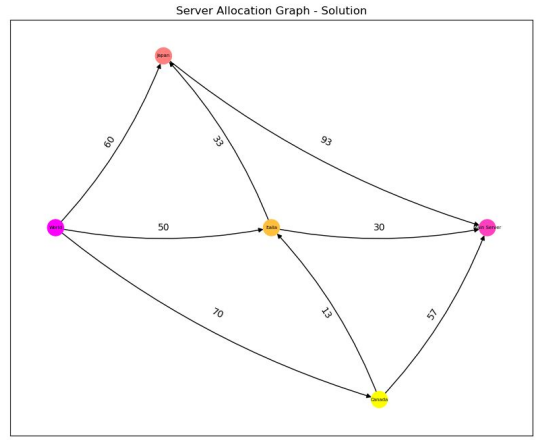
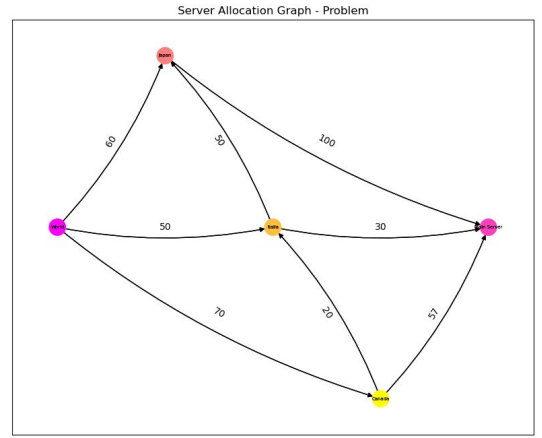
• Test Case 3

3
 Japan 100 60
 Italia 30 50
 Canada 57 70
 2
 Italia Japan 50
 Canada Italia 20

Expected :

- 180 static allocation
- 7 dynamic allocation

Result :
 The number of servers allocated statically: 180
 The number of servers allocated dynamically: 7



B. Analysis

From the test results, we can see that the implemented algorithm is indeed correct and able to solve this problem with any given number of countries and borrow relations. But, there are still weaknesses in this program as follows:

1. The visualization is not the best because of overlapping labels. We could fix this using another visualization library that has better label placement.
2. For a case where 2 countries allow each other to borrow servers (in test case 2), the implemented algorithm will still give us the correct static and dynamic allocation, but the flow network (from visualization) is wrong because Dinic's algorithm is disoriented when a cycle, created from 2 edges, exists.

VI. CONCLUSION

Server allocation problem can be represented as a max-flow problem and can be solved optimally using Dinic's algorithm as proven above. This shows that some real-life problems can be represented as computer science problems and can be solved using the correct algorithm. Even though the problem has been solved, there is still room for improvement, indicated from the test analysis. Further research is needed to improve Dinic's algorithm when a cycle, created from 2 edges, exists, to always give the correct solution in the form of network flow.

SOURCE CODE AT GITHUB

<https://github.com/haverzard/CFE/tree/master/ServerAllocation>

VIDEO LINK AT YOUTUBE

<https://youtu.be/T1gKSVeGNek>

ACKNOWLEDGMENT

The writer would like to thank Mr. Rinaldi Munir, Ms. Masayu Leylia Khodra, and Ms. Nur Ulfa Maulidevi for their guidance and patience in teaching algorithm strategies course. Their lectures brought a lot of new insights about algorithms and it has been an honor to learn from them.

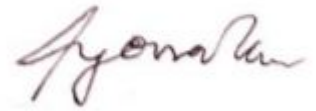
REFERENCES

- [1] How Stuff Works, "How to Host a Web Conference", accessed from <https://money.howstuffworks.com/business-communications/how-to-host-a-web-conference.htm> on April 20th, 2020.
- [2] James A. Storer. An Introduction to Data Structures and Algorithms. Berlin: Springer Science & Business Media, 2001, pp. 314.
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. London: MIT Press, 2009, pp. 586-612, 708-730.
- [4] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. Operating System Concepts, Tenth Edition. New Jersey: Wiley, 2018, pp. 200-244.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 2 Mei 2020



Yonatan Viody 13518120