Project Report of COMP6651: Analyzing and Implementing Minimum-Cost Flow Algorithms on Randomized Source-Sink Networks

Shanmukha Venkata Naga Sai Tummala
Department of Computer Science
Concordia University, 40289721
shanmukha.tummala@outlook.com

Mohamed Mohamed
Department of Computer Science
Concordia University, 40266483
Mohamed.rasmy.fathy@gmail.com

Parsa Kamalipour
Department of Computer Science
Concordia University, 40310734
parsa.kamalipour@mail.concordia.ca

Naveen Rayapudi
Department of Computer Science
Concordia University, 40291526
rayapudinaveen777@gmail.com

Abstract

This project examines the minimum-cost flow problem in source-sink networks, an extension of the maximum-flow problem. Using adaptations of the Ford-Fulkerson method, we implement and evaluate four algorithms: Successive Shortest Path, Capacity Scaling, combined approach, and Additionally, we include the Primal-Dual Algorithm for comparison. Randomly generated directed Euclidean graphs with specific properties are used as test cases. For each graph, we identify the largest connected component and determine source-sink pairs algorithmically. Performance metrics, such as flow, cost, and path characteristics, are analyzed for eight predefined graph configurations and additional custom scenarios. This work aims to assess the efficiency and effectiveness of these algorithms under different conditions, providing insights into their computational performance and practical applications.

1 Introduction

1.1 Introduction to minimum-cost flow problem

The minimum-cost flow (MCF) problem is an important topic in network optimization, with applications in areas such as transportation, telecommunications, scheduling, and resource management. The goal of this problem is to find the cheapest way to transport a given amount of flow from supply nodes to demand nodes in a directed network. Each connection (arc) in the network has a capacity limit and a cost per unit of flow, and the objective is to minimize the total cost while meeting all demands [1].

Researchers have developed many algorithms to solve the MCF problem over the years. These methods include basic techniques like cycle-canceling and shortest-path algorithms, as well as advanced methods such as cost-scaling and network simplex algorithms. The performance of these algorithms can vary depending on the size, density, and structure of the network. For example, network simplex algorithms often work better on smaller networks, while cost-scaling algorithms are faster on large and sparse networks due to their better efficiency for such cases [2].

In this project, we study four algorithms to solve the MCF problem: Successive Shortest Path, Capacity Scaling, a hybrid method, and the Primal-Dual Algorithm. We test these algorithms on different types of networks, including both randomly generated and real-world examples. The aim is

to understand how these methods perform under various conditions and provide practical insights into choosing the best algorithm for different types of problems.

1.2 Problem Description

The minimum-cost flow (MCF) problem is a fundamental challenge in network optimization. It involves determining the most cost-effective way to transport a specified amount of flow from supply nodes to demand nodes in a directed network. Each arc in the network has two key attributes:

- **Capacity (u_{ij})**: The maximum amount of flow that can pass through the arc.
- **Cost (c_{ij})**: The cost incurred per unit of flow on the arc.

Each node in the network is assigned a supply value (b_i), where:

- $b_i > 0$: The node is a supply node with a surplus of b_i.
- $b_i < 0$: The node is a demand node requiring $|b_i|$.
- $b_i = 0$: The node is a transshipment node with no supply or demand.

The objective is to find a flow assignment (x_{ij}) for all arcs (i, j) such that:

1. **Capacity Constraints:** The flow on each arc does not exceed its capacity:
$$0 \leq x_{ij} \leq u_{ij}, \quad \forall (i,j) \in A.$$
2. **Flow Conservation:** For every node, the total incoming flow plus supply equals the total outgoing flow:
$$\sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = b_i, \quad \forall i \in V.$$

The goal is to minimize the total cost of the flow:

$$\text{Total Cost} = \sum_{(i,j)\in A} c_{ij} \cdot x_{ij}.$$

The goal of the MCF problem is to minimize this total cost while satisfying all capacity and flow conservation constraints. [3]

This problem is a generalization of the maximum-flow problem, which focuses solely on maximizing flow without considering costs. As such, it has numerous practical applications, including transportation logistics, telecommunications network design, and resource allocation in supply chains. Solving this problem requires efficient algorithms that can handle both small-scale and large-scale network instances.

1.3 Motivation, Challenges, and Purpose of This Project

The minimum-cost flow (MCF) problem is important in network optimization, with applications in areas like transportation, telecommunications, and logistics. Solving this problem in real-world scenarios is difficult because networks are often large and complex. This project is motivated by the need to study and evaluate how different algorithms perform in terms of speed, scalability, and accuracy.

A major challenge is balancing computational speed with solution accuracy. Algorithms must handle large data and meet constraints on capacity, cost, and flow conservation. Their performance often depends on network features like size and density, making it essential to study when specific algorithms work best.

The purpose of this project is to implement and analyze four prominent algorithms for solving the MCF problem: the Successive Shortest Path, Capacity Scaling, Primal-Dual, and a hybrid approach. By testing these algorithms on randomly generated, this project aims to provide a comprehensive comparison of their efficiency, scalability, and practical applicability based on the Minimization results of each of the algorithms. Furthermore, the insights gained from this study will contribute to the broader understanding of network optimization methods and offer guidance for selecting appropriate algorithms for specific applications.

2 Related Work

The minimum-cost flow (MCF) problem has been extensively studied in the fields of operations research and computer science. Numerous algorithms have been developed to solve this problem, ranging from combinatorial methods to primal-dual approaches and cost-scaling techniques. Each algorithm offers unique trade-offs in terms of computational complexity, scalability, and suitability for various types of networks.

One of the earliest methods for solving the MCF problem is the network simplex algorithm, introduced by Dantzig [4]. This approach is widely used in practice due to its efficiency on small and medium-sized networks. Edmonds and Karp [5] later proposed capacity-scaling techniques that improve computational efficiency for larger networks. The successive shortest path algorithm, developed independently by Jewell [6] and Iri [7], is another key method that incrementally builds an optimal solution by augmenting flow along shortest paths.

Goldberg and Tarjan [8] introduced the cost-scaling algorithm, which has become one of the most efficient solutions for the MCF problem in both theory and practice. Their approach leverages the concept of ϵ-optimality to refine solutions iteratively. The primal-dual algorithm, initially proposed by Ford and Fulkerson [9], remains a foundational method, especially for problems requiring dual feasibility alongside primal optimization.

Recent studies have focused on experimental evaluations and efficient implementations of these algorithms. Kovács [3] compared modern implementations of cost-scaling, successive shortest path, and network simplex methods, highlighting their performance on large-scale networks. The LEMON library [10] provides open-source implementations of these algorithms, offering a practical framework for testing and deployment.

This study builds on prior work by implementing and comparing four MCF algorithms—Successive Shortest Path, Capacity Scaling, Primal-Dual, and a hybrid approach. It aims to evaluate their performance across different network scenarios, contributing to the understanding of algorithmic trade-offs in practical applications.

3 Implementation Details

3.1 Algorithm 1: Successive Shortest Paths

Brief description of the working of Algorithm 1 (Successive Shortest Paths):

This algorithm uses a greedy approach. It first checks if there is any augmenting path in the residual network, and if there is one, it selects the augmenting path that has the minimum cost. It then uses the capacity of that augmenting path to augment the flow in the network. After that, it computes the updated residual graph and updates the required total flow d that is yet to be pushed. This process repeats until d becomes 0 or there are no more augmenting paths.

Pseudo code for Algorithm 1:

Algorithm CHECKIFPATHEXISTS

```
Inputs: s (source node), t (sink node), residualGraph, visited
Output: True or False
if s = t then
    return True
end if
visited[s] ← 1
for i = 0 to n − 1 do
    if residualGraph[s][i] ≥ 1 and visited[i] = 0 then
        if CHECKIFPATHEXISTS(i, t, residualGraph, visited) then
            return True
        end if
    end if
end for
return False
```

Algorithm SuccessiveShortestPaths

Inputs: *graphFileName, s* (source node), *t* (sink node), *d* (required total flow)
adjacencyMatrix ← CREATEADJACENCYMATRIXFROMFILE(*graphFileName*)
cap ← CREATECAPACITYMATRIXFROMFILE(*graphFileName*)
unitCost ← CREATEUNITCOSTMATRIXFROMFILE(*graphFileName*)
Initialize *flow* matrix to all zeros
residualGraph ← COMPUTERESIDUALCAPACITY(*residualGraph*, *adjacencyMatrix*, *flow*, *cap*)
if $d = 0$ **then**
    return result with all fields set to 0
end if
while $d > 0$ **and** ISAUGMENTINGPATHEXISTS(s, t, *residualGraph*) **do**
    Initialize *minCostPath* ← $\emptyset$
    minCostPath ← FINDMINIMUMCOSTPATH(s, t, *unitCost*, *residualGraph*)
    maxFlowThatCanBePushed ← FINDMAXFLOWTHATCANBEPUSHED(*minCostPath*, *residualGraph*)
    if *maxFlowThatCanBePushed* $> d$ **then**
        maxFlowThatCanBePushed ← d
    end if
    AUGMENTFLOW(*maxFlowThatCanBePushed*, *adjacencyMatrix*, *flow*, *minCostPath*)
    COMPUTERESIDUALCAPACITY(*residualGraph*, *adjacencyMatrix*, *flow*, *cap*)
    $d \leftarrow d -$ *maxFlowThatCanBePushed*
end while
avgLengthOfAugmentingPath ← *sumOfLengthsOfAugmentingPaths* / *numOfAugmentingPaths*
longestAcyclicPath ← FINDLENGTHOFLONGESTACYCLICPATH(*adjacencyMatrix*, s, t)
meanProportionalLength ← *avgLengthOfAugmentingPath* / *longestAcyclicPath*
minCost ← FINDCOST(*flow*, *unitCost*)
flowValue ← GETFLOWVALUE(s, *flow*)
Result: <*minCost, flowValue, numOfAugmentingPaths, avgLengthOfAugmentingPath, meanProportionalLength*>
return *result*

Algorithm COMPUTEMINCOSTPATHSFROMSOURCE

Inputs: s (source node), *unitCost*, *residualGraph*, *parent*
Initialize *minCost*[0...n-1] ← ∞
minCost[s] ← 0
for $i = 1$ to $n - 1$ **do**
    for $j = 0$ to $n - 1$ **do**
        for $k = 0$ to $n - 1$ **do**
            if $j \neq k$ **and** *unitCost*[j][k] $\neq 0$ **then**
                if *residualGraph*[j][k] ≥ 1 **then**
                    if *minCost*[k] - *minCost*[j] $>$ *unitCost*$[j][k]$ **then**
                        minCost[k] ← *minCost*$[j]$ + *unitCost*$[j][k]$
                        parent[k] ← j
                    end if
                end if
            end if
        end for
    end for
end for

Algorithm ISAUGMENTINGPATHEXISTS

Inputs: s (source node), t (sink node), *residualGraph*
Output: *True* or *False*
Initialize *visited*[0...n-1] with all entries 0
return CHECKIFPATHEXISTS(s, t, *residualGraph*, *visited*)

Algorithm FINDMINIMUMCOSTPATH

Inputs: s (source node), t (sink node), *unitCost*, *residualGraph*
Initialize *parent*[0...n-1] $\leftarrow -1$
COMPUTEMINCOSTPATHSFROMSOURCE(s, *unitCost*, *residualGraph*, *parent*)
Initialize *minCostPath* $\leftarrow \emptyset$
$k \leftarrow t$
while $k \neq -1$ **do**
    Add k *to the front of minCostPath*
    $k \leftarrow$ *parent*$[k]$
end while
return *minCostPath*

Algorithm AUGMENTFLOW

Inputs: *maxFlowThatCanBePushed*, *adjacencyMatrix*, *flow*, *minCostPath*
Output: Updates the *flow* matrix by augmenting the flow
for each edge (u, v) **in** *minCostPath* **do**
    if *(u, v)* exists in the original network **then**
        flow[u][v] $\leftarrow$ *flow*$[u][v]$ + *maxFlowThatCanBePushed*
    else if *(v, u)* exists in the original network **then**
        flow[v][u] $\leftarrow$ *flow*$[v][u]$ − *maxFlowThatCanBePushed*
    end if
end for

3.2 Algorithm 2: Capacity Scaling

Brief description of the working of Algorithm 2 (Capacity Scaling):

This algorithm uses a scaling factor, which is initialized to the maximum capacity of an edge in the source-sink network. The scaling factor is reduced by 2 when there is no augmenting path with residual capacity greater than or equal to the scaling factor. Because of this, we can reduce the total number of times we augment the flow, improving the runtime of the algorithm. When there is an augmenting path in the network, it selects the augmenting path that has the shortest distance from source to sink among all the augmenting paths and uses the capacity of that augmenting path to augment the flow in the network. After that, it computes the updated residual graph and updates the required total flow d, that is yet to be pushed.

Pseudo code for Algorithm 2:

Algorithm CHECKIFPATHEXISTS

Inputs: s (source node), t (sink node), *scalingFactor*, *residualGraph*, *visited*
Output: *True* or *False*
if $s = t$ **then**
    return *True*
end if
visited[s] $\leftarrow 1$
for $i = 0$ to $n - 1$ **do**
    if *residualGraph*[s][i] $\geq$ *scalingFactor* **and** *visited*[i] = 0 **then**
        if CHECKIFPATHEXISTS(i, t, *scalingFactor*, *residualGraph*, *visited*) **then**
            return *True*
        end if
    end if
end for
return *False*

Algorithm Capacity Scaling

Inputs: *graphFileName, s* (source node), *t* (sink node), *d* (required total flow)
adjacencyMatrix ← CREATEADJACENCYMATRIXFROMFILE(*graphFileName*)
cap ← CREATECAPACITYMATRIXFROMFILE(*graphFileName*)
unitCost ← CREATEUNITCOSTMATRIXFROMFILE(*graphFileName*)
Initialize *flow* matrix to all zeros
residualGraph ← COMPUTERESIDUALCAPACITY(*adjacencyMatrix*, *flow*, *cap*)
if $d = 0$ **then**
    return result with all fields set to 0
end if
scalingFactor ← GETMAXCAPACITY(*cap*)
while *scalingFactor* ≥ 1 **do**
    while $d > 0$ **and** ISAUGMENTINGPATHEXISTS(s, t, *scalingFactor*, *residualGraph*) **do**
        shortestPath ← FINDSHORTESTPATH(s, t, *scalingFactor*, *residualGraph*)
        maxFlowThatCanBePushed ← FINDMAXFLOWTHATCANBEPUSHED(*shortestPath*, *residualGraph*)
        if *maxFlowThatCanBePushed* $> d$ **then**
            maxFlowThatCanBePushed ← d
        end if
        AUGMENTFLOW(*maxFlowThatCanBePushed*, *adjacencyMatrix*, *flow*, *shortestPath*)
        COMPUTERESIDUALCAPACITY(*residualGraph*, *adjacencyMatrix*, *flow*, *cap*)
        $d \leftarrow d -$ *maxFlowThatCanBePushed*
    end while
    scalingFactor ← *scalingFactor*/2
end while
avgLengthOfAugmentingPath ← *sumOfLengthsOfAugmentingPaths* / *numOfAugmentingPaths*
longestAcyclicPath ← FINDLENGTHOFLONGESTACYCLICPATH(*adjacencyMatrix*, s, t)
meanProportionalLength ← *avgLengthOfAugmentingPath* / *longestAcyclicPath*
minCost ← FINDCOST(*flow*, *unitCost*)
flowValue ← GETFLOWVALUE(s, *flow*)
Result: <*minCost, flowValue, numOfAugmentingPaths, avgLengthOfAugmentingPath, meanProportionalLength*>
return *result*

Algorithm ISAUGMENTINGPATHEXISTS

Inputs: s (source node), t (sink node), *scalingFactor*, *residualGraph*
Output: *True* or *False*
Initialize *visited*[0...n-1] with all entries 0
return CHECKIFPATHEXISTS(s, t, *scalingFactor*, *residualGraph*, *visited*)

Algorithm FINDSHORTESTPATH

Inputs: s (source node), t (sink node), *scalingFactor*, *residualGraph*
Initialize *parent*[0...n-1] ← -1
COMPUTESHORTESTPATHSFROMSOURCE(s, *scalingFactor*, *residualGraph*, *parent*)
Initialize *shortestPath* ← $\emptyset$
$k \leftarrow t$
while $k \neq -1$ **do**
    Add k *to the front of shortestPath*
    $k \leftarrow$ *parent*$[k]$
end while
return *shortestPath*

Algorithm COMPUTESHORTESTPATHSFROMSOURCE

Inputs: s (source node), *scalingFactor*, *residualGraph*, *parent*
Initialize *shortestDistance*[0...n-1] $\leftarrow \infty$
shortestDistance[s] $\leftarrow 0$
for $i = 1$ to $n - 1$ **do**
  for $j = 0$ to $n - 1$ **do**
    for $k = 0$ to $n - 1$ **do**
      if $j \neq k$ **and** *unitCost*[j][k] $\neq 0$ **then**
        if *residualGraph*[j][k] $\geq$ *scalingFactor* **then**
          if *shortestDistance*[k] - *shortestDistance*[j] > 1 **then**
            shortestDistance[k] $\leftarrow$ *shortestDistance*$[j] + 1$
            parent[k] $\leftarrow j$
          end if
        end if
      end if
    end for
  end for
end for

Algorithm AUGMENTFLOW

Inputs: *maxFlowThatCanBePushed*, *adjacencyMatrix*, *flow*, *shortestPath*
Output: Updates the *flow* matrix by augmenting the flow
for each edge (u, v) **in** *shortestPath* **do**
  if *(u, v)* exists in the original network **then**
    flow[u][v] $\leftarrow$ *flow*$[u][v]$ + *maxFlowThatCanBePushed*
  else if *(v, u)* exists in the original network **then**
    flow[v][u] $\leftarrow$ *flow*$[v][u]$ − *maxFlowThatCanBePushed*
  end if
end for

3.3 Algorithm 3: Successive Shortest Paths with Scaling Factor (SC)

Brief description of the working of Algorithm 3 (Successive Shortest Paths SC):

This algorithm uses a scaling factor, which is initialized to the maximum capacity of an edge in the source-sink network. The scaling factor is reduced by 2 when there is no augmenting path with residual capacity greater than or equal to the scaling factor. Because of this, we can reduce the total number of times we augment the flow, improving the runtime of the algorithm. When there is an augmenting path in the network, it selects the augmenting path with the minimum cost among all the augmenting paths. It then uses the capacity of that augmenting path to augment the flow in the network. After that, it computes the updated residual graph and updates the required total flow d that is yet to be pushed. [11]

Pseudo code for Algorithm 3:

Algorithm ISAUGMENTINGPATHEXISTS

Inputs: s (source node), t (sink node), *scalingFactor*, *residualGraph*
Output: *True* or *False*
Initialize *visited*[0...n-1] with all entries 0
return CHECKIFPATHEXISTS(s, t, *scalingFactor*, *residualGraph*, *visited*)

Algorithm SuccessiveShortestPathsSC

Inputs: *graphFileName, s* (source node), *t* (sink node), *d* (required total flow)
adjacencyMatrix ← CREATEADJACENCYMATRIXFROMFILE(*graphFileName*)
cap ← CREATECAPACITYMATRIXFROMFILE(*graphFileName*)
unitCost ← CREATEUNITCOSTMATRIXFROMFILE(*graphFileName*)
Initialize *flow* matrix to all zeros
residualGraph ← COMPUTERESIDUALCAPACITY(*residualGraph*, *adjacencyMatrix*, *flow*, *cap*)
if $d = 0$ **then**
    return result with all fields set to 0
end if
scalingFactor ← GETMAXCAPACITY(*cap*)
while *scalingFactor* ≥ 1 **do**
    while $d > 0$ **and** ISAUGMENTINGPATHEXISTS(s, t, *scalingFactor*, *residualGraph*) **do**
        Initialize *minCostPath* ← $\emptyset$
        maxFlowThatCanBePushed ← FINDMAXFLOWTHATCANBEPUSHED(*minCostPath*, *residualGraph*)
        if *maxFlowThatCanBePushed* $> d$ **then**
            maxFlowThatCanBePushed ← d
        end if
        AUGMENTFLOW(*maxFlowThatCanBePushed*, *adjacencyMatrix*, *flow*, *minCostPath*)
        COMPUTERESIDUALCAPACITY(*residualGraph*, *adjacencyMatrix*, *flow*, *cap*)
        $d \leftarrow d -$ *maxFlowThatCanBePushed*
    end while
    scalingFactor ← *scalingFactor*$/2$
end while
avgLengthOfAugmentingPath ← *sumOfLengthsOfAugmentingPaths* / *numOfAugmentingPaths*
longestAcyclicPath ← FINDLENGTHOFLONGESTACYCLICPATH(*adjacencyMatrix*, s, t)
meanProportionalLength ← *avgLengthOfAugmentingPath* / *longestAcyclicPath*
minCost ← FINDCOST(*flow*, *unitCost*)
flowValue ← GETFLOWVALUE(s, *flow*)
return *result*

Algorithm CHECKIFPATHEXISTS

Inputs: s (source node), t (sink node), *scalingFactor*, *residualGraph*, *visited*
Output: *True* or *False*
if $s = t$ **then**
    return *True*
end if
visited[s] ← 1
for $i = 0$ to $n - 1$ **do**
    if *residualGraph*[s][i] $\geq$ *scalingFactor* **and** *visited*[i] = 0 **then**
        if CHECKIFPATHEXISTS(i, t, *scalingFactor*, *residualGraph*, *visited*) **then**
            return *True*
        end if
    end if
end for
return *False*

Algorithm AUGMENTFLOW

Inputs: *maxFlowThatCanBePushed*, *adjacencyMatrix*, *flow*, *minCostPath*
Output: Updates the *flow* matrix by augmenting the flow
for each edge (u, v) **in** *minCostPath* **do**
    if *(u, v)* exists in the original network **then**
        flow[u][v] $\leftarrow$ *flow*$[u][v]$ + *maxFlowThatCanBePushed*
    else if *(v, u)* exists in the original network **then**
        flow[v][u] $\leftarrow$ *flow*$[v][u]$ − *maxFlowThatCanBePushed*
    end if
end for

3.4 Algorithm 4: Primal-Dual Algorithm

Description of the Primal-Dual Algorithm:

The Primal-Dual Algorithm is a method particularly effective for solving network flow problems such as the Minimum Cost Flow problem. The goal is to determine the minimum cost flow in a directed graph. This algorithm operates by iteratively adjusting the flow through a network while maintaining dual feasibility through node potentials and reduced costs. The approach ensures that each augmentation step is both feasible and cost-efficient, ultimately leading to an optimal flow configuration that satisfies the specified demand with minimal total cost. [12, 13]

Key Aspects:

- **Primal Aspect:** Focuses on adjusting flows along paths from the source to the sink to meet demand while respecting edge capacities.
- **Dual Aspect:** Maintains node potentials to ensure that the reduced costs of edges remain non-negative, facilitating efficient path selection.

Steps:

1. Compute initial node potentials using the Bellman-Ford Algorithm. Although typically used to handle potential negative edge costs, this step does not check for negative costs since the graph does not include any.
2. Use a modified Dijkstra's Algorithm with reduced costs to identify the shortest augmenting path from the source to the sink.
3. Increase the flow along the identified path by the minimum residual capacity available on that path.
4. Adjust node potentials (`UpdatePotentials`) based on the latest distances (reduced costs) to maintain dual feasibility.
5. Repeat the path finding and augmentation steps until the demand is met or no further augmenting paths are available.

Pseudo code:

Algorithm PrimalDualAlgo

Inputs: G (directed graph), s (source vertex), t (sink vertex), *demand* (required flow)
Output: *AlgoResult* (result object containing total cost, total flow, path count, mean length, and mean proportional length)
INITIALIZEPOTENTIALS(G, s)
longestAcyclicPath $\leftarrow$ FINDLONGESTACYCLICPATH(G, s, t)
totalFlow $\leftarrow 0$
totalCost $\leftarrow 0.0$
pathCount $\leftarrow 0$
cumulativePathLength $\leftarrow 0$
while *totalFlow* $<$ *demand* **do**
    pathResult $\leftarrow$ DIJKSTRA(G, s, t)
    amount $\leftarrow \min$(*pathResult.minCapacity*, *demand - totalFlow*)
    AUGMENTFLOW(*pathResult*, *amount*, s, t)
    totalFlow $\leftarrow$ *totalFlow* $+$ *amount*
    totalCost $\leftarrow$ *totalCost* $+$ (*amount* $\times$ *pathResult.pathCost*)
    pathCount $\leftarrow$ *pathCount* $+ 1$
    cumulativePathLength $\leftarrow$ *cumulativePathLength* $+$ (LENGTH(*pathResult.path*) $- 1$)
    UPDATEPOTENTIALS(*pathResult.distances*)
end while
if *pathCount* > 0 **then**
    meanLength $\leftarrow$ *cumulativePathLength*/*pathCount*
else
    meanLength $\leftarrow 0.0$
end if
if *longestAcyclicPath* > 0 **then**
    meanProportionalLength $\leftarrow$ *meanLength*/*longestAcyclicPath*
else
    meanProportionalLength $\leftarrow 0.0$
end if
return ALGORESULT(*totalCost*, *totalFlow*, *pathCount*, *meanLength*, *meanProportionalLength*)

Algorithm INITIALIZEPOTENTIALS

Inputs: G (directed graph), s (source vertex)
Output: Initializes potentials using Bellman-Ford without checking for negative costs
distances $\leftarrow \infty$, *predecessors* $\leftarrow$ Null
distances[s] $\leftarrow 0$
for $i = 1$ **to** $|V| - 1$ **do**
    for each edge (u, v) **in** $G.E$ **do**
        if REMAININGCAPACITY((u, v)) > 0 **then**
            weight $\leftarrow$ COST((u, v))
            if *distances*[u] + *weight* $<$ *distances*[v] **then**
                distances[v] $\leftarrow$ *distances*[u] $+$ *weight*
                predecessors[v] $\leftarrow u$
            end if
        end if
    end for
end for
for each vertex v **in** V **do**
    nodePotentials[v] $\leftarrow$ *distances*[v]
end for
return *true*

Algorithm DIJKSTRA

Inputs: G (directed graph), s (source vertex), t (sink vertex)
Output: *PathResult* (distances, predecessors, minCapacity, pathCost, path)
distances $\leftarrow \infty$, *predecessors* $\leftarrow$ Null
distances[s] $\leftarrow 0$
queue $\leftarrow$ PRIORITYQUEUE(s, 0)
while *queue*.ISEMPTY() = *false* **do**
    $u \leftarrow$ EXTRACTMIN(*queue*)
    for each edge (u, v) **in** $G.E$ **do**
        if REMAININGCAPACITY((u, v)) > 0 **then**
            reducedCost $\leftarrow$ COST((u, v)) + *nodePotentials*[u] - *nodePotentials*[v]
            if *reducedCost* $<$ *distances*[v] $-$ *distances*[u] **then**
                distances[v] $\leftarrow$ *distances*[u] $+$ *reducedCost*
                predecessors[v] $\leftarrow u$
                queue.enqueue(v, *distances*[v])
            end if
        end if
    end for
end while
path $\leftarrow$ EMPTYLIST
$v \leftarrow t$, *minCapacity* $\leftarrow \infty$, *pathCost* $\leftarrow 0.0$
while $v \neq s$ **do**
    $e \leftarrow$ *predecessors*[v]
    minCapacity $\leftarrow \min($*minCapacity*, REMAININGCAPACITY((e, v))$)$
    pathCost $\leftarrow$ *pathCost* $+$ COST((e, v))
    path.append(v)
    $v \leftarrow e$
end while
path.append(s)
REVERSE(*path*)
result $\leftarrow$ PATHRESULT
result.distances $\leftarrow$ *distances*
result.predecessors $\leftarrow$ *predecessors*
result.minCapacity $\leftarrow$ *minCapacity*
result.pathCost $\leftarrow$ *pathCost*
result.path $\leftarrow$ *path*
return *result*

Algorithm UPDATEPOTENTIALS

Inputs: *distances* (the distance of all vertices from the source)
for each vertex v **in** G **do**
    if *distances*[v] $< \infty$ **then**
        nodePotentials[v] $\leftarrow$ *nodePotentials*[v] $+$ *distances*[v]
    end if
end for

4 Implementation correctness

To ensure the correctness of our implementation, we developed a dedicated testing class named `Test`, which validates the results of all four algorithms. We used three different source-sink graphs with 5, 11, and 26 edges, respectively. Since these graphs are relatively small, we manually computed the expected values for minimum cost and flow for all four algorithms and stored them in a 2-D array, `expectedFlowAndMinCost`.

The graphs were saved in three separate files: `graph-test-1.txt`, `graph-test-2.txt`, and `graph-test-3.txt`. The `Test` class executes each algorithm on these graph files and compares the

output with the precomputed expected values. If the outputs match, the test passes; otherwise, it fails. The results of each test are clearly displayed on the console.

In addition to this structured testing approach, we ran the algorithms multiple times to ensure consistency and reliability. No issues were observed during these runs. We also compared metrics such as the number of augmenting paths and the minimum cost across the algorithms to detect any unexpected behavior. All comparisons confirmed the correctness of our implementation, with all algorithms performing as expected.

5 Results

5.1 Tables

Table 1: Simulation 1: Graph Characteristics

Graph	n	r	upperCap	upperCost	fMax	nodesInLCC	maxOutDegree	maxInDegree	avgDegree
1	100	0.2	8	5	6	95	10	11	4.305
2	200	0.2	8	5	9	200	18	17	8.815
3	100	0.3	8	5	10	100	19	21	9.690
4	200	0.3	8	5	56	200	32	34	18.095
5	100	0.2	64	20	45	99	10	11	4.273
6	200	0.2	64	20	74	200	17	21	8.290
7	100	0.3	64	20	38	100	19	19	9.450
8	200	0.3	64	20	326	200	29	30	17.335

Table 2: Simulation 1: Algorithm Performance Metrics

Algorithm	Graph	Flow (f)	Min Cost (MC)	Paths	Mean Length (ML)	Mean Proportional Length (MPL)
SSP	1	5	203.0	4	15.000	0.300
CS	1	5	217.0	3	14.333	0.287
SSPCS	1	5	203.0	3	15.000	0.300
PD	1	5	201.0	4	17.750	0.222
SSP	2	8	102.0	9	10.111	0.060
CS	2	8	166.0	6	10.167	0.061
SSPCS	2	8	122.0	6	10.667	0.063
PD	2	8	101.0	5	6.000	0.033
SSP	3	9	86.0	16	8.500	0.104
CS	3	9	122.0	9	8.556	0.104
SSPCS	3	9	89.0	9	9.111	0.111
PD	3	9	86.0	7	6.571	0.069
SSP	4	53	582.0	54	7.111	0.037
CS	4	53	1297.0	20	7.600	0.039
SSPCS	4	53	1217.0	21	8.524	0.044
PD	4	53	571.0	39	8.308	0.044
SSP	5	42	3732.0	66	8.439	0.196
CS	5	42	4918.0	24	8.375	0.195
SSPCS	5	42	4129.0	25	9.560	0.222
PD	5	42	3664.0	13	13.769	0.181
SSP	6	70	3587.0	79	9.114	0.055
CS	6	70	6724.0	28	8.321	0.050
SSPCS	6	70	3802.0	30	10.100	0.060
PD	6	70	3577.0	14	13.929	0.083
SSP	7	36	1332.0	83	9.048	0.113
CS	7	36	2087.0	31	8.129	0.102
SSPCS	7	36	1367.0	33	9.970	0.125
PD	7	36	1332.0	3	7.333	0.088
SSP	8	309	8849.0	141	8.879	0.048
CS	8	309	16749.0	43	7.302	0.039
SSPCS	8	309	10253.0	47	9.362	0.051
PD	8	309	8673.0	84	11.655	0.076

Table 3: Simulation 2, Set 1: Graph Characteristics

Graph	n	r	upperCap	upperCost	fMax	nodesInLCC	maxOutDegree	maxInDegree	avgDegree
9	100	0.9	8	5	139	100	50	49	38.080
10	200	0.9	8	5	357	200	98	100	79.745
11	100	0.9	64	20	1148	100	52	52	39.480
12	200	0.9	64	20	2616	200	100	100	77.175

Table 4: Simulation 2, Set 1: Algorithm Performance Metrics

Algorithm	Graph	Flow (f)	Min Cost (MC)	Paths	Mean Length (ML)	Mean Proportional Length (MPL)
SSP	9	132	938.0	67	3.478	0.036
CS	9	132	1277.0	34	2.971	0.031
SSPCS	9	132	1094.0	39	3.462	0.036
PD	9	132	899.0	80	4.600	0.047
SSP	10	339	2187.0	233	3.210	0.016
CS	10	339	2773.0	124	2.903	0.015
SSPCS	10	339	2443.0	130	3.315	0.017
PD	10	339	2078.0	206	4.796	0.024
SSP	11	1090	24791.0	334	3.437	0.035
CS	11	1090	32541.0	170	2.900	0.030
SSPCS	11	1090	26854.0	172	3.430	0.035
PD	11	1090	24017.0	173	7.538	0.076
SSP	12	2485	53223.0	560	3.646	0.019
CS	12	2485	73211.0	261	2.904	0.015
SSPCS	12	2485	61673.0	280	3.668	0.019
PD	12	2485	50584.0	545	9.268	0.047

Table 5: Simulation 2, Set 2: Graph Characteristics

Graph	n	r	upperCap	upperCost	fMax	nodesInLCC	maxOutDegree	maxInDegree	avgDegree
13	100	0.1	8	5	2	8	2	2	1.125
14	200	0.1	8	5	5	108	9	7	2.981
15	100	0.1	64	20	9	15	4	3	1.467
16	200	0.1	64	20	3	93	6	9	2.720

Table 6: Simulation 2, Set 2: Algorithm Performance Metrics

Algorithm	Graph	Flow (f)	Min Cost (MC)	Paths	Mean Length (ML)	Mean Proportional Length (MPL)
SSP	13	1	15.0	1	6.000	0.857
CS	13	1	15.0	1	6.000	0.857
SSPCS	13	1	15.0	1	6.000	0.857
PD	13	1	15.0	1	6.000	0.750
SSP	14	4	204.0	5	17.400	0.378
CS	14	4	244.0	2	13.500	0.293
SSPCS	14	4	212.0	2	13.500	0.293
PD	14	4	204.0	4	20.750	0.648
SSP	15	8	708.0	7	15.000	1.250
CS	15	8	904.0	3	12.000	1.000
SSPCS	15	8	776.0	3	12.000	1.000
PD	15	8	708.0	2	9.000	0.818
SSP	16	2	438.0	8	16.750	0.356
CS	16	2	510.0	4	15.750	0.335
SSPCS	16	2	438.0	4	16.250	0.346
PD	16	2	438.0	1	29.000	0.537

Table 7: Simulation 2, Set 3: Graph Characteristics

Graph	n	r	upperCap	upperCost	fMax	nodesInLCC	maxOutDegree	maxInDegree	avgDegree
17	100	0.2	8	1	7	96	14	12	5.083
18	200	0.3	8	1	35	200	40	37	18.740
19	100	0.2	64	1	8	98	9	9	4.194
20	200	0.3	64	1	246	200	36	34	17.865

Table 8: Simulation 2, Set 3: Algorithm Performance Metrics

Algorithm	Graph	Flow (f)	Min Cost (MC)	Paths	Mean Length (ML)	Mean Proportional Length (MPL)
SSP	17	6	51.0	4	8.750	0.133
CS	17	6	54.0	2	9.000	0.136
SSPCS	17	6	54.0	2	9.000	0.136
PD	17	6	51.0	4	8.750	0.162
SSP	18	33	194.0	27	6.296	0.034
CS	18	33	226.0	12	6.750	0.037
SSPCS	18	33	226.0	12	6.750	0.037
PD	18	33	190.0	23	6.435	0.041
SSP	19	7	70.0	30	6.667	0.106
CS	19	7	70.0	13	7.000	0.111
SSPCS	19	7	70.0	13	7.000	0.111
PD	19	7	70.0	3	10.000	0.182
SSP	20	233	1256.0	57	6.070	0.034
CS	20	233	1294.0	24	6.375	0.036
SSPCS	20	233	1294.0	24	6.375	0.036
PD	20	233	1238.0	27	6.333	0.035

Table 9: Simulation 2, Set 4: Graph Characteristics

Graph	n	r	upperCap	upperCost	fMax	nodesInLCC	maxOutDegree	maxInDegree	avgDegree
21	100	0.9	8	1	113	100	52	55	38.680
22	200	0.9	8	1	332	200	104	102	79.020
23	100	0.9	64	1	1063	100	51	52	39.710
24	200	0.9	64	1	2566	200	103	103	79.135

Table 10: Simulation 2, Set 4: Algorithm Performance Metrics

Algorithm	Graph	Flow (f)	Min Cost (MC)	Paths	Mean Length (ML)	Mean Proportional Length (MPL)
SSP	21	107	290.0	43	2.791	0.028
CS	21	107	322.0	26	2.846	0.029
SSPCS	21	107	322.0	26	2.846	0.029
PD	21	107	290.0	51	3.137	0.032
SSP	22	315	860.0	188	2.782	0.014
CS	22	315	937.0	114	2.860	0.014
SSPCS	22	315	937.0	114	2.860	0.014
PD	22	315	860.0	153	3.497	0.018
SSP	23	1009	2843.0	257	2.813	0.029
CS	23	1009	2867.0	150	2.867	0.029
SSPCS	23	1009	2867.0	150	2.867	0.029
PD	23	1009	2843.0	83	3.566	0.036
SSP	24	2437	6669.0	429	2.818	0.014
CS	24	2437	6876.0	244	2.857	0.014
SSPCS	24	2437	6913.0	242	2.860	0.014
PD	24	2437	6669.0	200	3.460	0.017

Table 11: Simulation 2, Set 5: Graph Characteristics

Graph	n	r	upperCap	upperCost	fMax	nodesInLCC	maxOutDegree	maxInDegree	avgDegree
25	100	0.5	1	5	16	100	33	33	18.290
26	200	0.5	1	5	27	200	67	70	39.310
27	100	0.5	1	20	7	100	37	40	20.180
28	200	0.5	1	20	16	200	77	75	41.950

Table 12: Simulation 2, Set 5: Algorithm Performance Metrics

Algorithm	Graph	Flow (f)	Min Cost (MC)	Paths	Mean Length (ML)	Mean Proportional Length (MPL)
SSP	25	15	116.0	15	3.867	0.042
CS	25	15	142.0	15	3.400	0.037
SSPCS	25	15	116.0	15	3.867	0.042
PD	25	15	111.0	15	5.400	0.057
SSP	26	25	166.0	40	3.750	0.019
CS	26	25	244.0	40	3.325	0.017
SSPCS	26	25	166.0	40	3.750	0.019
PD	26	25	164.0	25	4.680	0.024
SSP	27	6	102.0	46	3.913	0.042
CS	27	6	222.0	46	3.391	0.036
SSPCS	27	6	102.0	46	3.913	0.042
PD	27	6	102.0	6	5.000	0.052
SSP	28	15	257.0	61	4.148	0.021
CS	28	15	479.0	61	3.311	0.017
SSPCS	28	15	257.0	61	4.148	0.021
PD	28	15	253.0	15	7.267	0.037

5.2 Simulation Results and Observations

The performance of the four algorithms was evaluated on various graph datasets generated with different configurations. Below, we summarize the key observations and results obtained across different simulation sets.

5.2.1 Input Set 1

Description: We chose the value of r to be very high ($r = 0.9$) while keeping the other parameters the same as in Simulations 1. A high r implies that every node is connected to a higher number of nodes, resulting in higher degrees for all nodes. This configuration increases the number of paths between the source and target, often leading to shorter path lengths compared to Simulations 1.

Observation 1: In Simulations 1, where $r = 0.2$ or $r = 0.3$, the minimum cost of Algorithm 2 was approximately 30% higher than the second worst algorithm. However, in Simulations 2, the minimum cost of Algorithm 2 is only 10% higher than the second worst algorithm. Although Algorithm 2 remains the worst-performing algorithm, it shows comparatively better performance with a very high r. This improvement is due to the increased availability of shorter paths with lower costs, as the edge costs are uniformly distributed between 1 and the upperCost.

Observation 2: In Simulations 1, where $r = 0.2$ or $r = 0.3$, Algorithm 1's minimum cost was closer to the optimal minimum cost achieved by the Primal-Dual algorithm (Algorithm 4). However, in this input set with $r = 0.9$, the performance of Algorithm 1 is not as close to the optimal. This discrepancy can be attributed to Algorithm 1's dependency on minimum cost paths and its greedy strategy, which does not perform as effectively in graphs with very high degrees.

Other Observations:

- Algorithm 3 continues to perform better than Algorithm 2 but does not surpass Algorithm 1.
- The mean length of augmenting paths in Algorithm 4 is significantly higher than that of the other algorithms.

5.2.2 Input Set 2

Description: For this input set, we chose a very low value of r ($r = 0.1$), while keeping the other parameters the same. A low r results in a reduced number of edges, creating very sparse graphs.

Observations: In this scenario, all the algorithms exhibit almost identical behavior. Due to the sparsity of the graphs, there are fewer paths from the source to the sink, increasing the likelihood that all algorithms identify the same augmenting paths.

5.2.3 Input Set 3

Description: In this input set, we set the upperCap to 1 while keeping all other parameters the same as in Simulations 1. This configuration ensures that every edge has a unit cost of 1.

Observation 1: Algorithms 2 and 3 exhibit identical behavior in this scenario. Since upperCap $= 1$, every edge has a unit cost of 1, making the shortest path equivalent to the minimum cost path. Additionally, as both algorithms use the same scaling factor, they consistently select the same augmenting paths. As a result, the metrics such as minimum cost, number of augmenting paths, mean length (ML), and mean proportional length (MPL) are identical for both Algorithms 2 and 3.

Observation 2: Algorithm 1 achieves the minimum cost value equivalent to the optimal cost in most instances. This outcome is attributed to both PD (Algorithm 4) and SSP (Algorithm 1) dealing with minimum cost paths. However, the greedy nature of SSP typically poses a drawback. In this case, since upperCap $= 1$, the impact of the greedy approach is minimized, leading to competitive performance.

5.2.4 Input Set 4

Description: In this input set, we set $r = 0.9$ and upperCost $= 1$. This configuration ensures a dense graph with a uniform edge cost.

Observation 1: Algorithms 2 and 3 exhibit the same behavior, as observed in the previous input set, since upperCap $= 1$ in this set as well. However, a notable difference is that Algorithm 2, which generally performs the worst, achieves a minimum cost value very close to the optimal. This can be explained by the equivalence of shortest paths and minimum cost paths in this scenario. The only factor causing Algorithm 2's performance to lag is the use of the scaling factor, which slightly increases its minimum cost.

5.2.5 Input Set 5

Description: In this input set, we set upperCap $= 1$, ensuring all edges have uniform capacity.

Observation 1: Since upperCap $= 1$, the scaling factor has no effect, resulting in Algorithm 3 behaving exactly the same as Algorithm 1. Consequently, both algorithms yield identical values for minimum cost and the number of augmenting paths.

Observation 2: Algorithms 1, 3, and 4 produce nearly identical minimum cost values. However, Algorithm 2 performs significantly worse than the others. This disparity arises from Algorithm 2's reliance solely on shortest paths, disregarding minimum cost paths, which leads to suboptimal performance.

5.3 Conclusion and Common Observations

Across all input sets, we observed distinct patterns in the performance of the algorithms. These patterns can be summarized as follows:

- The number of augmenting paths in Algorithm 1 is consistently greater than or equal to that in Algorithm 3. This is because Algorithm 3 uses a scaling factor, which reduces the number of augmenting paths. If minimizing the number of augmenting paths is a priority and we are not overly concerned with achieving the most accurate minimum cost, Algorithm 3 is a suitable choice.
- The Primal-Dual algorithm (Algorithm 4) consistently delivers the optimal minimum cost, outperforming the other algorithms in this metric. If achieving the best minimum cost is essential, Algorithm 4 should be the preferred choice.
- Algorithm 2 performs the worst in terms of minimum cost across almost all input sets. However, it may still be a viable option in scenarios where the output is influenced primarily by the distance between the source and target, and the minimum cost paths are less critical.

In such cases, Algorithm 2 can be chosen for its reduced number of augmenting paths, provided a lower accuracy is acceptable.

These observations provide clear insights into the trade-offs among the algorithms, enabling informed decisions based on the specific requirements of the application.

6 Conclusion

This study compares four algorithms for solving the minimum-cost flow problem using different graph settings. The results highlight the strengths and weaknesses of each algorithm and help in understanding which one is better for specific situations.

Algorithm 1 (Successive Shortest Path) works well in finding minimum cost paths, especially in graphs with fewer edges. However, because it uses a greedy method, it does not perform as well in very dense graphs. Algorithm 3, which uses a scaling factor, needs fewer augmenting paths compared to Algorithm 1 but is slightly less accurate in finding the minimum cost. This makes Algorithm 3 a good option when reducing the number of augmenting paths is more important than getting the exact minimum cost.

Algorithm 2 performs worse than the others in minimizing cost in most input sets. However, it works better in graphs with high edge density (r) and uniform edge costs. Since Algorithm 2 focuses only on shortest paths, it may be useful in cases where the number of augmenting paths is more important than minimizing cost.

The Primal-Dual algorithm (Algorithm 4) is the best in finding the lowest cost in all graph settings. It performs well in both dense and sparse graphs, making it the best choice for applications where minimizing cost is the main goal.

The results show that the number of augmenting paths depends on the algorithm and graph properties. Algorithms that use scaling factors, like Algorithm 3, need fewer paths, while greedy algorithms, like Algorithm 1, use more paths. This shows the importance of choosing an algorithm based on the specific needs of the problem.

In summary, this study provides useful insights into the performance of these four algorithms. For applications where achieving the lowest cost is important, the Primal-Dual algorithm is the best choice. When efficiency or reducing augmenting paths is more important, Algorithms 1 and 3 are good options. Algorithm 2, while not optimal in cost, can work well in special cases where shortest paths are enough. These results help in deciding which algorithm to use for different real-world problems related to the minimum-cost flow problem.

7 Team Work Distribution

TODO

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. Network flows: Theory, algorithms, and applications. 1993.

[2] P. T. Sokkalingam, Ravindra K. Ahuja, and James B. Orlin. New polynomial-time cycle-canceling algorithms for minimum-cost flows. *Networks*, 36:53–63, 2000.

[3] Péter Kovács. Minimum-cost flow algorithms: an experimental evaluation. *Optimization Methods and Software*, 30(1):94–127, 2015.

[4] GEORGE B. DANTZIG. *Linear Programming and Extensions*. Princeton University Press, 1991.

[5] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM (JACM)*, 19:248 – 264, 1972.

[6] William S. Jewell. New methods in mathematical programming—optimal flow through networks with gains. *Operations Research*, 10(4):476–499, 1962.

[7] Masao Iri. A new method for solving transportation network problems. *Journal of Operations Research*, 3:27–31, 1960.

[8] Andrew V. Goldberg and Robert Endre Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, 15:430–466, 1990.

[9] G. B. Dantzig, L. R. Ford, and D. R. Fulkerson. A primal-dual algorithm for linear programs. In H. W. Kuhn and A. W. Tucker, editors, *Linear Inequalities and Related Systems*, pages 171–181. Princeton University Press, Princeton, NJ, 1956.

[10] Zoltán Király and Péter Kovács. Lemon: Library for efficient modeling and optimization in networks. `http://lemon.cs.elte.hu`, 2012.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[12] TopCoder. Minimum Cost Flow Part Two: Algorithms. `https://www.topcoder.com/thrive/articles/Minimum%20Cost%20Flow%20Part%20Two:%20Algorithms`. Accessed: 2024-12-09.

[13] C. Seshadhri. Minimum cost flow - primal-dual method. Lecture slides, available at: `https://users.soe.ucsc.edu/~sesh/Teaching/2021/CSE202/Slides/lec7-mincostflow-primaldual.pdf`. Accessed: 2024-12-09.