Solving the Single-Source Unsplittable Flow Problem with Metaheuristic Approaches

Aleksa Toroman 84/2018

Univerzitet u Beogradu, Matematički fakultet

September 28, 2024

Table of contents

1 Introduction

The Single-Source Unsplittable Flow Problem (SSUFP) is a well-known problem in network flow optimization. Given a graph $G = (V, E)$ with a source vertex s and a collection of sinks $t_1, \ldots, t_k$ associated with integer demands $\rho_1, \ldots, \rho_k$, the goal is to route a single flow path from the source s to each sink t_i such that the total flow across any edge $e \in E$ does not exceed the edge's capacity $c(e)$. Formally, the objective is to minimize the maximum ratio $\frac{f(e)}{c(e)}$ over all edges, where $f(e)$ is the total flow routed across edge e.
Problem Statement:

- **Instance:** A directed graph $G = (V, E)$ with edge capacities $c : E \to Z^+$, a source node s, and k sinks $t_1, \ldots, t_k$, each with a demand ρ_i. Each demand ρ_i must be routed along a single path from s to t_i. Additionally, each vertex can have any number of sinks.
- **Objective:** Route each demand in such a way that the total flow routed across any edge $e \in E$ is bounded by its capacity, $c(e)$.
- **Measure:** Minimize the maximum congestion, defined as $\max_{e \in E} \frac{f(e)}{c(e)}$.

The SSUFP is an NP-hard problem, meaning that there is no known polynomial-time algorithm that can solve all instances optimally. The difficulty of the problem lies in the fact that flows cannot be split across multiple paths for each demand; every demand must be routed along a single path. This constraint introduces significant challenges in finding feasible flow paths, especially as the number of demands or the size of the network increases.

Complexity and Challenges:
Several factors increase the complexity of the problem:

- **Edge Congestion:** Ensuring that no edge exceeds its capacity is challenging when multiple demands compete for the same edges.
- **Path Selection:** Selecting the right paths for each demand is crucial to avoid overloading the network.
- **Scalability:** As the number of demands or the size of the network increases, the number of possible path combinations grows exponentially, making the problem harder to solve.

Known Algorithms:

Despite the complexity, there are approximation algorithms available that provide good solutions in practice. Some notable results include:

- A 2-approximation algorithm for minimizing congestion exists when certain conditions are met, meaning that the algorithm can find a solution with at most twice the optimal congestion.

- The problem is approximable within a factor of 3.23 for directed graphs, which provides a performance guarantee on how far the solution is from the optimal.
- It is also known that no approximation algorithm can achieve a performance better than $\frac{3}{2} - \epsilon$ for any $\epsilon > 0$, demonstrating the inherent difficulty of the problem.

In addition to these approximation algorithms, here we are covering metaheuristic approaches such as Simulated Annealing, Genetic Algorithms, and Variable Neighborhood Search offer heuristic methods to find near-optimal solutions. These algorithms are particularly useful for large instances where exact methods become impractical.

The remainder of this report will explore metaheuristic approaches to solving the SSUFP, focusing on their implementation and the quality of the solutions they provide.

2 Research Methodology

The research for solving the Single-Source Unsplittable Flow Problem (SSUFP) was conducted in several stages, each contributing to the overall understanding and development of a practical solution. The following steps outline the methodology used in this project:

1. Getting familiar with the problem and reading existing literature.
2. Creating a custom graph textual representation.
3. Developing a graph generation library for instance creation.
4. Building core infrastructure classes in Python, including representations for the graph, demands, and utilities.
5. Implementing algorithms, including Brute Force, Simulated Annealing, and Genetic Algorithm.
6. Analyzing and evaluating the results of the implementations.

Each of these steps will be elaborated in the subsequent sections.

2.1 Getting Familiar with the Problem and Literature Review

The initial phase of the project focused on understanding the Single-Source Unsplittable Flow Problem (SSUFP). This involved studying its mathematical formulation, related optimization challenges, and reviewing literature on approximation algorithms. Research on unsplittable flow and NP-hard network problems offered insights into solution techniques and complexities.

3 Graph Representation and Infrastructure

3.1 Creating a Custom Graph Textual Representation

To represent the input graph for the Single-Source Unsplittable Flow Problem, a custom textual format was designed. This format is easy to parse and process programmatically,

and it captures all necessary details, including vertices, edges, capacities, and demands for each sink.
The structure of the graph representation is as follows:

- The first line contains the number of vertices.
- The second line indicates the number of edges.
- The third line identifies the source vertex.
- The next N lines represent each edge in the format: *source vertex, destination vertex, capacity*, where the graph is directed.
- After the edges, there is a line containing the number of sink vertices.
- The remaining lines correspond to the sink vertices, where each line contains the index of the vertex, the number of sinks that vertex has, and their respective demands.

Example Representation:

```
5
7
1
1 2 7
1 3 3
2 4 6
2 3 5
2 5 4
5 4 2
1 5 5
2
4 1 3
3 2 2 1
```

In this example:

- The graph has 5 vertices and 7 edges.
- Vertex 1 is the source.
- The edges are represented as directed connections between vertices, with specified capacities.
- There are 2 sink vertices, with vertex 3 having two demands (2 and 1), and vertex 4 having one demand (3).

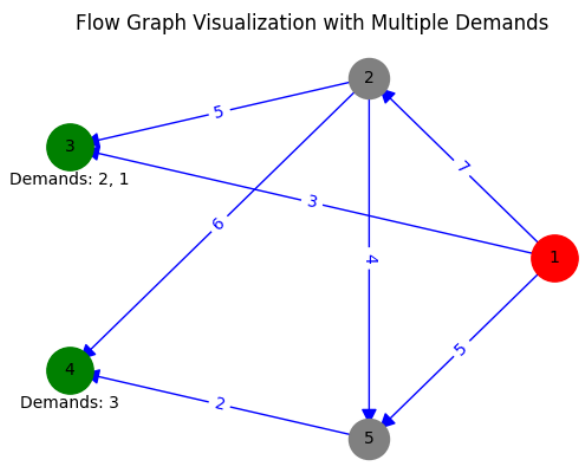


Figure 1: Example graph problem instance

Example Solution:

```
Path for Flow (Source: 1, Sink: 4, Flow: 3): [1, 2, 4]
Path for Flow (Source: 1, Sink: 3, Flow: 2): [1, 3]
Path for Flow (Source: 1, Sink: 3, Flow: 1): [1, 2, 3]
All constraints satisfied.
Maximum Flow-to-Capacity Ratio: 0.67
```

3.2 Graph Generation Library for Instance Creation

In order to effectively test the algorithms, a graph generation library was developed due to the lack of a pre-existing dataset for the Single-Source Unsplittable Flow Problem (SSUFP). Initially, I created a few simple examples that could be manually calculated, serving as a basic test for the algorithms. However, for larger-scale testing and analysis, I implemented a mechanism to generate random graph instances of various sizes.

The graph generation uses the NetworkX library in python to create directed random graphs, with adjustments made to tailor the graphs specifically for the SSUFP problem. The process involves the following key steps:

- Initial Graph Generation: A random directed graph is generated using the Erdős-Rényi model, with an edge probability that varies depending on the graph size. For small graphs, the edge probability is 0.2, for medium graphs it is 0.15, and for large graphs it is 0.1. These probabilities influence the density of edges in the generated graph.
- Graph Adjustments: After generating the random graph, certain adjustments are made:
  - Outgoing edges from non-source vertices to the source are removed to avoid reverse flow.

  - Outgoing edges from sink vertices are removed to ensure sinks do not act as intermediate vertices for other flows.
  - If no path exists between the source and any sink, random "hops" are added between intermediate nodes to create a valid path, ensuring at least one solution exists for each instance.

- Edge Capacities: Capacities for the edges are assigned randomly within a range of 10 to 40. These capacities represent the maximum flow that each edge can carry.
- Demand Generation: Demands for sink vertices are generated based on the type of graph. For balanced graphs, each sink vertex typically has 1 or 2 demands, while for large-demand graphs, each sink can have up to 3 demands. The demand values are random integers, but they are kept small compared to the edge capacities to ensure feasible solutions. Also, each demand cannot be higher than the capacity of any edge in defined graph.
- Graph Sizes:
  - Small Graphs: Small instances contain between 5 and 10 nodes, with 1 to 3 demands.
  - Medium Graphs: Medium instances range from 15 to 25 nodes, with 3 to 6 demands.
  - Large Graphs: Large instances contain between 30 and 50 nodes, with 5 to 10 demands.
- Graph Serialization: After generating and adjusting the graph, it is serialized into a given text format described earlier. The serialization includes all nodes, edges, capacities, and demands. The generated graph is saved as a text file, where each graph's filename includes a hash for uniqueness.

3.3 Building Core Infrastructure Classes in Python

To effectively implement and test various algorithms for solving the Single-Source Unsplittable Flow Problem (SSUFP), several core infrastructure components were developed in Python. These include essential classes to represent the graph, demands, and results, as well as utility functions used by the algorithms. The code was designed to be modular and reusable, supporting various hyperparameter configurations and testing scenarios.

Core Classes:

- **Graph Class**: This class represents the underlying network with nodes and edges, storing relevant information such as capacities and connections between vertices as well as list of demands.
- **Demand Class**: Each demand corresponds to a specific sink vertex and its associated demand value, ensuring that the each demand in graph must be routed unsplittably along a single path.
- **Result Class**: This class holds the results of each algorithm run, tracking information such as the time taken, whether the solution was feasible, and the objective function value.

- **Statistics Class**: For each algorithm run, a separate 'reports.csv' file is generated, containing detailed information about the results. This data is later aggregated to compare the performance of different algorithms and parameter configurations.

Utility Functions: In addition to the core classes, several utility functions were implemented to assist the algorithms:

- **Feasibility Check**: A utility function checks whether a solution is feasible by verifying that the flow on each edge does not exceed its capacity. This operation is of polynomial time complexity.

- **Random Path Finder**: A depth-first search (DFS) based function is used to find random paths between two nodes.

Hyperparameter Support: The infrastructure is flexible enough to allow passing different hyperparameters to various algorithms. Similar to the grid search in libraries like *scikit-learn*, different combinations of hyperparameters can be tested to optimize the performance of the algorithms. Parameters such as cooling rate for Simulated Annealing or population size for the Genetic Algorithm can be modified this way.

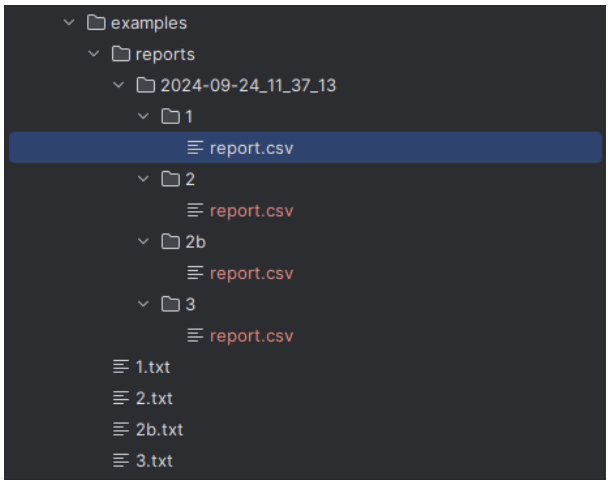
Automation and Result Logging: A root file structure was designed to automatically run different algorithms and store the results in a well-organized format. The results for each algorithm run are saved in CSV files, which include details such as:

- Algorithm used
- Hyperparameters used
- Time taken by the algorithm
- Whether the solution was feasible
- Objective function value (optimization target)
- ...

This mechasnism also allows to store different artifacts such as different kinds of plots in well organized structure during the algorithm execution.

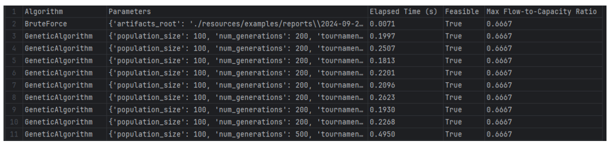
Additionally, a command-line interface allows users to specify which algorithms to run, and where is the root folder of given example instances. The results are automatically exported into an csv file in above mentioned file structure, making it easier to analyze and compare the performance of different algorithms across multiple instances.

Below is an example of the folder structure and a snippet from the result log:



```
examples
  reports
    2024-09-24_11_37_13
      1
        report.csv
      2
        report.csv
      2b
        report.csv
      3
        report.csv
    1.txt
    2.txt
    2b.txt
    3.txt
```

Figure 2: Screenshot of result file structure and log



Algorithm	Parameters	Elapsed Time (s)	Feasible	Max Flow-to-Capacity Ratio
BruteForce	{'artifacts_root': './resources/examples/reports\\2024-09-2…	0.0071	True	0.6667
GeneticAlgorithm	{'population_size': 100, 'num_generations': 200, 'tournamen…	0.1997	True	0.6667
GeneticAlgorithm	{'population_size': 100, 'num_generations': 200, 'tournamen…	0.2507	True	0.6667
GeneticAlgorithm	{'population_size': 100, 'num_generations': 200, 'tournamen…	0.1813	True	0.6667
GeneticAlgorithm	{'population_size': 100, 'num_generations': 200, 'tournamen…	0.2201	True	0.6667
GeneticAlgorithm	{'population_size': 100, 'num_generations': 200, 'tournamen…	0.2096	True	0.6667
GeneticAlgorithm	{'population_size': 100, 'num_generations': 200, 'tournamen…	0.2623	True	0.6667
GeneticAlgorithm	{'population_size': 100, 'num_generations': 200, 'tournamen…	0.1930	True	0.6667
GeneticAlgorithm	{'population_size': 100, 'num_generations': 200, 'tournamen…	0.2268	True	0.6667
GeneticAlgorithm	{'population_size': 100, 'num_generations': 500, 'tournamen…	0.4950	True	0.6667

Figure 3: Snippet of result log showing algorithm details and performance

4 Algorithm Overview

4.1 Used Algorithms

To solve the Single-Source Unsplittable Flow Problem (SSUFP), several algorithms were implemented, each offering different approaches to finding feasible and near-optimal solutions. The algorithms developed include both exact and metaheuristic techniques:

- **Brute Force**: A straightforward approach that explores all possible paths from the source to each sink and evaluates all combinations of paths to find the best feasible solution. Due to the problem's NP-hard nature, this method is only feasible for small instances.

- **Simulated Annealing**: A metaheuristic algorithm inspired by the annealing process in metallurgy. It starts with an initial solution and explores neighboring solutions, gradually reducing the likelihood of accepting worse solutions as the temperature decreases. This allows for exploration of the solution space while avoiding getting stuck in local optima.

- **Variable Neighborhood Search (VNS)**: This algorithm systematically changes the neighborhood structures during the search. By altering the neighborhood, VNS explores different parts of the solution space, switching between local searches to escape local optima and find better solutions.

- **Genetic Algorithm (GA)**: A population-based optimization algorithm that mimics natural selection. The algorithm generates an initial population of solutions and evolves them using operations like crossover and mutation. Over several generations, the population converges towards a feasible and near-optimal solution.

Each of these algorithms were implemented in Python, leveraging the core infrastructure classes and utility functions described earlier.

In the following sections, we will present the results obtained from each algorithm, how is it implemented, along with a comparison of their effectiveness and quality of solutions.

4.2 Brute Force Algorithm

The Brute Force method retrieves all possible paths from the source to each sink, including multiple demands for the same sink. Using the NetworkX library to find simple paths, this works well for small graphs.
The algorithm then generates all route combinations using the Cartesian product of the paths and checks each for feasibility. For feasible solutions, it calculates the maximum congestion.
While effective for small graphs, this approach becomes impractical for larger graphs due to the exponential growth in path combinations.

Example: In one example graph with 11 vertices and 3 sink nodes, where two of the sink nodes have two demands each, the number of possible paths for each demand is as follows:

```
Number of paths for each demand: [19, 19, 38, 38, 11]
```

This results in a total of 5,734,124 combinations. Out of these, only 3,041 combinations were feasible. The Brute Force algorithm completed in 63.03 seconds for this instance.

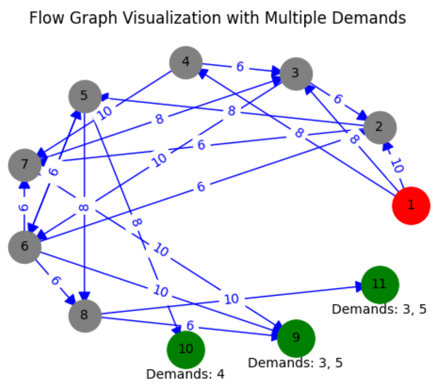


Figure 4: Example graph solved using brute force algorithm

- Total combinations: 5,734,124
- Feasible combinations: 3,041
- Time taken: 63.03 seconds

While Brute Force provides an exact solution, its time complexity grows exponentially, making it impractical for graphs with many paths and multiple demands. This highlights the need for more scalable, approximate methods like Simulated Annealing, VNS and Genetic Algorithms for larger instances.

4.3 Common Stoppage Criteria for Heuristic Algorithms

All heuristic algorithms in this project are controlled by two main criteria: maximum execution time and the no_improvement_threshold. These criteria ensure that algorithms stop either after a set time or when no further improvements are made.

- **Maximum Time**: Each algorithm runs for a set time limit, ranging from 20 seconds for small graphs to 2 minutes for larger ones, to balance runtime and exploration.
- **No_Improvement_Threshold**: This defines how many iterations can occur without improvement. For Simulated Annealing, this threshold is higher (100-150 iterations), while for Genetic Algorithms, it's lower (50-100 iterations).

The parameters adjust with graph size:

- **Small Graphs**: Lower limits (20-40 seconds and 50 iterations).
- **Medium Graphs**: Moderate limits (40-90 seconds and around 100 iterations).
- **Large Graphs**: Higher limits (up to 2 minutes and 150 iterations).

These limits can be adjusted for more exploration if computational resources allow but are kept within reasonable bounds to ensure efficient testing and meaningful results.

4.4 Generating Neighbor Solution

In Simulated Annealing, Variable Neighborhood Search (VNS), and Genetic Algorithms, generating a neighbor solution is key to exploring new options. This function goal is to make randomized slight adjustments in current solution and can be implement in various ways based on problem and solution representation. In our case, this function works in following way:

- It randomly selects demand in available list of demands.
- It randomly selects two nodes in the previously selected demand flow path (including the source and sink).
- A new path between these two nodes is generated using randomized depth-first search (DFS).
- The new path replaces the selected portion of the original flow path.

To optimize this process:

- Cycles are avoided by excluding nodes that would form a loop.
- The current path is excluded from consideration to ensure a true modification.

This method is used in Simulated Annealing for exploring new solutions, in VNS to switch between neighborhoods, and in Genetic Algorithms during mutation to introduce variety.

4.5 Goal Function

In this problem, there are two key objectives:

- Find a feasible solution where the flow on every edge does not exceed its capacity.
- Minimize the congestion, measured by:

$$\max_{e \in E} \frac{f(e)}{c(e)},$$

where $f(e)$ is the total flow on edge e and $c(e)$ is its capacity.

To achieve this, the goal function follows these rules:

- Feasible solutions are always prioritized over non-feasible ones.
- Among feasible solutions, the one with the lowest max_flow_to_capacity ratio is considered the best.
- Non-feasible solutions are compared based on the total overflow across all edges, with smaller total overflow being better. This helps guide the algorithm toward feasible solutions.

Goal Function Pseudocode:

```
function calculate_score():
    total_penalty = 0
    feasible = True

    edge_flows = calculate_edge_flows()
    capacities = edges

    for each edge in edge_flows:
        flow = edge_flows[edge]
        capacity = capacities[edge]

        if flow > capacity:
            overflow = flow - capacity
            total_penalty += overflow
            feasible = False

    if feasible:
        max_flow_to_capacity_ratio = calculate_max_flow_to_capacity_ratio()
        return 1 - max_flow_to_capacity_ratio
    else:
        return -(1 + total_penalty)
```

This function ensures that feasible solutions with lower congestion are favored, while non-feasible solutions are penalized based on how much they overflow. This strategy helps navigate the search space toward finding feasible, optimal solutions.

4.6 Simulated Annealing

The Simulated Annealing algorithm was implemented to explore the solution space for the Single-Source Unsplittable Flow Problem (SSUFP). The goal is to find feasible solutions while minimizing the congestion, using the previously defined goal function.

Parameter Grid:

```
sa_param_grid = {
    'initial_temp': [1000, 5000],
    'cooling_rate': [0.99, 0.95]
}
```

Algorithm Overview:

- The algorithm starts with an initial solution where, for each demand, a random path is generated from the source to the sink. If a sink has multiple demands, the method is called separately for each.
- At each iteration, a neighbor solution is generated by altering the current flow paths as described earlier.
- The goal function is used to evaluate each solution, and a solution is accepted based on its score. If the new solution is worse, an acceptance probability is calculated based on the current temperature, allowing worse solutions to be accepted with some likelihood to avoid local optima.
- The temperature gradually decreases according to the cooling rate, reducing the likelihood of accepting worse solutions as the algorithm progresses.

This standard approach to Simulated Annealing leverages the flexibility of the neighbor generation function and the goal function to guide the search toward feasible and optimal solutions.

4.7 Variable Neighborhood Search (VNS)

Variable Neighborhood Search (VNS) is a metaheuristic algorithm that systematically explores different neighborhoods of the solution space. In this implementation, neighborhoods are defined based on the number of demands whose paths will be modified during the shaking process. The algorithm also alternates between small, incremental changes and more significant rerouting of paths.

Parameter Grid:

```
vns_param_grid = {
    'k_min': [0, 1, 2],
    'k_max': [3, 5],
    'move_prob': [0.1, 0.2],
    'reroute_entire_path_prob': [0.3, 0.5]
}
```

Shaking Process:

- Each neighborhood defines how many demands to modify. The number of demands increases as the search moves through larger neighborhoods.
- For each demand selected in the shaking process, there is a probability (between 0.3 and 0.5 based on given params configuration) of performing a full rerouting from the source to the sink, which replaces the entire path with a new one. Otherwise, a small modification is made to the path using the neighbor generation method.

This dual approach allows the algorithm to explore both small and large modifications, balancing local exploration with more significant changes when necessary.

Local Search: After shaking, the local search is applied to refine the solution. The goal is to iteratively improve the solution by making small changes to individual demands. The process works as follows:

- A copy of the current solution is made.
- A randomized order of demands is generated to explore possible improvements.
- For each demand, a neighboring solution is generated by modifying the flow path for that demand.
- The score of the modified solution is calculated using the goal function. If the new score is better, the current solution is replaced with the modified one, and the search continues.
- This process repeats until no further improvements can be made, ensuring the local optimum is reached.

This combination of shaking and local search allows VNS to efficiently escape local optima by dynamically adjusting the neighborhood size and exploring a variety of solution modifications.

4.8 Genetic Algorithm (GA)

The Genetic Algorithm (GA) was implemented to solve the Single-Source Unsplittable Flow Problem (SSUFP) by mimicking the process of natural selection. This involves creating an initial population, selecting the best solutions, and applying crossover and mutation to evolve new generations.

Parameter Grid:

```
ga_param_grid = {
    'population_size': [100, 200],
    'num_generations': [200, 500],
    'tournament_size': [5, 7],
    'elitism_size': [10, 20],
    'mutation_prob': [0.1, 0.2]
}
```

Algorithm Overview:

- **Initial Population:** The initial population is generated in the same way as the initial solution for Simulated Annealing, by finding random paths for each demand. This process is repeated based on the size of the population.
- **Selection:** Tournament-based selection is used to choose parents for crossover. In each tournament, a subset of individuals is selected at random, and the one with the best fitness is chosen as a parent.
- **Crossover:** Uniform crossover is applied to combine two parents and create two children. For each demand, there is a 50% chance of taking the demand path from one parent or the other. The unselected demand path goes to the other child, ensuring both children are formed by mixing parts of their parents' paths.
- **Mutation:** Each demand path in the child solution is mutated with a given probability (mutation_prob). The mutation is performed using the previously described *generate_neighbor* method, which modifies part of the path while maintaining feasibility.

```
function mutate(mutation_prob):
    for each demand path in solution:
        if random() < mutation_prob:
            mutate the demand path using generate_neighbor
```

The fitness of each solution is determined by the goal function described earlier. After several generations, the best solutions evolve toward feasible and optimal paths, balancing exploration and exploitation through crossover and mutation.

4.9 Results Analysis