Minimum Feedback Vertex Set

Project assignment for Computational Intelligence course

Isidora Dukić 146/2020
Milica Pantović 46/2019

October, 2025

Faculty of Mathematics, University of Belgrade

Contents

1 Introduction

The **Minimum Feedback Vertex Set (MFVS)** problem is a classical combinatorial optimization problem in graph theory. Given a directed graph $G = (V, E)$, the objective is to find the smallest subset of vertices $F \subseteq V$ whose removal makes the graph acyclic. Formally:

$$\text{Find } F \subseteq V \quad \text{such that} \quad G - F \text{ is a DAG and } |F| \text{ is minimized.}$$

MFVS is an NP-complete problem [1], which makes exact solutions computationally expensive for large graphs. This problem has applications in circuit analysis, deadlock prevention in operating systems, program verification, and computational biology. Several heuristic and approximation algorithms have been proposed in the literature to tackle MFVS:

- **Greedy algorithms** that iteratively remove vertices based on cycle participation or degree measures [2, 3].
- **D-sequence reduction**, a form of WLS (Weighted Local Search) approach, which prioritizes vertices strongly associated with cycles [4].
- **Exact methods**, such as exhaustive search, are feasible only for very small graphs [1].
- **Variable Neighborhood Search** for escaping local optima

In this project, we implement and evaluate several approaches to MFVS, including naive exact search, greedy heuristics, D-sequence reduction, Wang–Lloyd–Soffa, Small-k Iterative Compression and VNS and visualize their performance on small to medium-size directed graphs.

Note: In the following sections of the document, the function names contain `dfvs`, which stands for *directed feedback vertex set*. This refers to the MFVS problem in the context of directed graphs.

2 Our Solution

2.1 Imports and Directed Utilities

We use utility functions to provide essential tools for analyzing directed graphs in the context of the MFVS problem, including cycle detection, graph reduction, validation, and solution visualization.

- **Imports:**
  - `itertools, time, csv` – for combinatorial operations, timing, and CSV file handling.
  - `typing` – for type annotations (`List`, `Set`, `Dict`, etc.).
  - `networkx` (`nx`) – main package for graph representation and algorithms.
  - `matplotlib.pyplot` (`plt`) – for plotting and visualizing graphs.
- **Function** `is_acyclic(G)`: Checks if the directed graph `G` is acyclic (a DAG). Raises a `TypeError` if the input is not a `DiGraph`.
- **Function** `is_acyclic_after_removing(G, S)`: Removes the nodes in set `S` from the directed graph `G` and checks whether the resulting graph is acyclic (a Directed Acyclic Graph, DAG). Returns `True` if $G \setminus S$ is acyclic, and `False` otherwise.
- **Function** `cycle_basis(G)`: Returns a list of all simple directed cycles in `G`. Only works for directed graphs.
- **Function** `clean_graph(G)`: Iteratively removes all sources (nodes with in-degree 0) and sinks (nodes with out-degree 0) from `G`, returning a reduced graph.
- **Function** `validate_dfvs(G, F)`: Validates a proposed MFVS `F` by removing its nodes from `G` and checking if the remaining graph is acyclic.
- **Visualization Helpers:**
  - `draw_digraph(G, title)`: Draws the directed graph `G` with node labels and arrows.

  - `highlight_dfvs(G, F, title)`: Highlights the MFVS nodes `F` in a different color and displays the graph.
  - `visualize_solution(G, F, label, runtime)`: Prints summary information about a MFVS solution (size, runtime, validity), highlights the MFVS in the original graph, and draws the resulting DAG after removal.

- **I/O Helper**

  The `load_digraph_from_edgelist` function reads a directed graph from a CSV or text file, where each line represents a directed edge. It supports optional headers and custom delimiters. Each edge is added to a `networkx.DiGraph`, and the resulting graph is returned. Extra columns beyond the first two are ignored.

2.2 Implemented Algorithms

2.2.1 Naive Exact Algorithm

The `naive_dfvs` function implements an **exhaustive search** over all vertex subsets to find a **Minimal Feedback Vertex Set (MFVS)**. This approach is **computationally feasible only for very small graphs**, due to its combinatorial complexity.

- **Function signature:**

  ```
  def naive_dfvs(G: nx.DiGraph, time_limit: float | None = None) -> Set
  ```

- **Parameters:**
  - `G`: a `networkx.DiGraph` representing the input directed graph.
  - `time_limit`: optional float specifying a maximum execution time (in seconds). The function returns the best solution found if the limit is exceeded.
- **Returns:** a `set` of vertices forming a MFVS.
- **Algorithm description:**
  1. Start a timer and check if `G` is already acyclic. If so, return an empty set.
  2. List all vertices `V` of the graph.
  3. Initialize the best solution `best` as all vertices.
  4. For each subset size `r` from 1 to $|V|$:
     - Generate all combinations of vertices of size `r`.
     - For each subset `S`, check if removing `S` makes the graph acyclic.
     - If a time limit is specified and exceeded, return the best solution found so far.
     - If `S` makes the graph acyclic, return `S` immediately.
  5. If no smaller MFVS is found, return `best` (all vertices).

- **Visualization Helper for Naive MFVS**

  The `run_naive_and_plot` function executes the `naive_dfvs` algorithm and visualizes the result using the previously defined `visualize_solution` utility.

  - **Function signature:**

    ```
    def run_naive_and_plot(G: nx.DiGraph, time_limit: float = 3.0)
    ```

  - **Parameters:**
    * `G`: a directed graph (`nx.DiGraph`).
    * `time_limit`: maximum time allowed for the naive search (default 3 seconds).

  - **Functionality:**
    1. Records the start time.
    2. Runs the `naive_dfvs` algorithm.
    3. Measures the total runtime.
    4. Uses `visualize_solution` to print summary information, highlight the MFVS, and draw the resulting DAG after MFVS removal.

Note: Due to the exponential complexity, this approach is only suitable for graphs with very few vertices (typically less than 10–12 nodes).

2.2.2 Greedy Heuristic: Frequency-Based MFVS

The `greedy_frequency_dfvs` function implements a **greedy heuristic** for finding a Directed Feedback Vertex Set (MFVS) by iteratively removing the vertex that appears most frequently across simple directed cycles.

- **Function signature:**

  ```
  def greedy_frequency_dfvs(G: nx.DiGraph, cap: int | None = 10000) -> Set
  ```

- **Parameters:**
  - `G`: a `networkx.DiGraph` representing the input directed graph.
  - `cap`: optional integer to limit the number of cycles considered in each iteration (default 10000). This prevents excessive computation on dense graphs.

- **Returns:** a `set` of vertices forming a MFVS.

- **Algorithm description:**
  1. Apply `clean_graph` to iteratively remove sources and sinks, obtaining `H`.
  2. Initialize the MFVS set `F` as empty.
  3. While `H` is not acyclic:
     - Count occurrences of each vertex in simple directed cycles (up to the `cap` limit).
     - Identify the vertex `v_star` with the highest frequency.
     - Remove `v_star` from `H` and add it to `F`.
     - Apply `clean_graph` again to remove new sources and sinks.
  4. Return the set `F`.

- **Visualization Helper for Frequency-Based MFVS**

  The `run_frequency_and_plot` function executes the frequency-based greedy MFVS algorithm and visualizes the solution.

- **Function signature:**

  ```
  def run_frequency_and_plot(G: nx.DiGraph, cap: int = 10000)
  ```

- **Parameters:**
  * `G`: directed graph (`nx.DiGraph`).
  * `cap`: maximum number of cycles to consider per iteration (default 10000).
- **Functionality:**
  1. Record the start time.
  2. Run `greedy_frequency_dfvs` on `G`.
  3. Measure the runtime.
  4. Use `visualize_solution` to print summary information, highlight MFVS nodes, and draw the resulting DAG after MFVS removal.

Note: This heuristic is efficient for medium-sized graphs and provides a good approximation of MFVS, although it does not guarantee minimality. The `cap` parameter prevents long computation times on graphs with many cycles.

2.2.3 Greedy Heuristic: Degree-Product MFVS

The `greedy_degree_product` function implements a **greedy heuristic** for finding a Minimal Feedback Vertex Set (MFVS) by iteratively removing the vertex that maximizes the product of its in-degree and out-degree. This method also applies `clean_graph` pruning to remove sources and sinks in each iteration.

- **Function signature:**

  ```
  def greedy_degree_product(G: nx.DiGraph, time_limit: float = 5.0) -> Set
  ```

- **Parameters:**
  - `G`: a `networkx.DiGraph` representing the input directed graph.

  - `time_limit`: maximum runtime in seconds for the algorithm (default 5.0s). The function stops early if the time limit is exceeded.

- **Returns:** a `set` of vertices forming a MFVS.

- **Algorithm description:**

  1. Record the start time and apply `clean_graph` to remove sources and sinks, obtaining `H`.
  2. Initialize the MFVS set `F` as empty.
  3. While `H` is not acyclic and the time limit has not been exceeded:
     - For each vertex `v` in `H`, compute the score `s = in_degree(v) * out_degree(v)`.
     - Select the vertex `best_v` with the maximum score.
     - Remove `best_v` from `H` and add it to `F`.
     - Apply `clean_graph` again to prune sources and sinks.
  4. Return the set `F`.

- **Visualization Helper for Degree-Product MFVS**

  The `run_degprod_and_plot` function executes the degree-product greedy MFVS algorithm and visualizes the solution.

  - **Function signature:**

    ```
    def run_degprod_and_plot(G: nx.DiGraph, time_limit: float = 5.0)
    ```

  - **Parameters:**
    * `G`: directed graph (`nx.DiGraph`).
    * `time_limit`: maximum allowed runtime in seconds (default 5.0s).
  - **Functionality:**
    1. Records the start time.
    2. Runs `greedy_degree_product` on `G`.
    3. Measures the runtime.

4. Uses `visualize_solution` to print summary information, highlight MFVS nodes, and draw the resulting DAG after MFVS removal.

Note: This heuristic is efficient for medium to large graphs and typically provides a good approximate MFVS. The time limit ensures that the algorithm does not run excessively long on large graphs.

2.2.4 D-Sequence Reduction Heuristic

The `d_sequence_reduce` function implements a heuristic based on the **Weighted Least Subgraph (WLS) D-sequence** method [9]. It attempts to reduce a directed graph by iteratively identifying vertices associated with cycles and removing them.

- **Supporting Functions:**
  - `_deadlocked_nodes(G)`: Returns all nodes that can reach at least one directed cycle via any path. It finds all nodes in cycles, then traverses the reversed graph to collect all predecessors that can reach these cycles.
  - `associated_graph(G, x)`: Computes the subgraph of `G` associated with node `x`, keeping `x` and nodes not deadlocked. Nodes that are "deadlocked" (can reach some cycle) are excluded.
- **Function:** `d_sequence_reduce(G)`
  - **Parameters:** `G`, a directed graph (`nx.DiGraph`)
  - **Returns:** a tuple `(sequence, F_candidate)`, where `sequence` is the D-sequence order of chosen vertices, and `F_candidate` is the candidate MFVS set.
  - **Algorithm:**
    1. Initialize an empty sequence and candidate MFVS set.
    2. While `H` (a copy of `G`) is not acyclic:
       * For each node `x`, compute its associated subgraph `A`.
       * Choose the first `x` such that `A` is not acyclic.
       * Add `x` to the D-sequence and candidate MFVS set.

    * Remove all nodes of `associated_graph(H, x)` from `H`.

  3. Return the sequence and candidate MFVS.

- **Visualization Helper for D-Sequence MFVS**

  The `run_dseq_and_plot` function executes the D-sequence heuristic and visualizes the solution.

  - Records the start time.
  - Runs `d_sequence_reduce` on `G`.
  - Measures runtime.
  - Uses `visualize_solution` to highlight MFVS nodes and draw the resulting DAG.
  - Prints the order of nodes in the D-sequence.

Note: The D-sequence reduction heuristic is useful for pre-processing and reducing the size of the MFVS problem. It may be combined with other heuristics for refinement.

2.2.5 Incremental Construction (IC) Heuristic for Small k

The `ic_dfvs_small_k` function implements a **tiny incremental construction (IC) scaffold** to find a Minimum Feedback Vertex Set (MFVS) for very small instances or small k. The approach builds the graph incrementally and ensures acyclicity at each step, using exact or heuristic MFVS computations for small subgraphs.

- **Function signature:**

  ```
  def ic_dfvs_small_k(G: nx.DiGraph, k: int) -> Tuple[bool, Set]
  ```

- **Parameters:**
  - `G`: a directed graph (`nx.DiGraph`).
  - `k`: maximum allowed size of the MFVS.
- **Returns:** a tuple `(success, F)`, where `success` is a boolean indicating whether a MFVS of size at most k was found, and `F` is the MFVS set.
- **Algorithm description:**
  1. If `k` > 20, raise a `ValueError`.
  2. Initialize an empty graph `H` with the first node of `G` and an empty MFVS set `F`.
  3. Incrementally add nodes of `G` to `H`, adding corresponding edges.
  4. After adding each node, check if the current MFVS `F` keeps `H` acyclic using `validate_mfvs`.
  5. If not, refine `F` using a small MFVS heuristic (`degree-product` mode) on `H`.
  6. Attempt all subsets of `F` to minimize the MFVS while ensuring acyclicity and $|F| \leq k$.
  7. Return `False, {}` if no valid MFVS of size $\leq k$ is found, else return `True, F`.
- **Visualization Helper for Small-k Iterative Compression MFVS**

  The `run_ic_small_and_plot` function executes the Small-k Iterative Compression (IC) heuristic and visualizes the solution.

  - Records the start time.
  - Runs `ic_dfvs_small_k` on graph `G` with parameter `k`.
  - Measures the runtime of the algorithm.

  - If a solution is found uses `visualize_solution` to highlight the MFVS nodes and draw the resulting DAG.
  - Otherwise, prints `'|MFVS| <= k not found'` to indicate no solution was found within the given bound.

Note: This incremental approach is only suitable for very small graphs or small k, as the combinatorial search grows rapidly. It can be used as an exact method for tiny instances or as a scaffold for hybrid heuristics.

2.2.6 Variable Neighborhood Search (VNS) Heuristic

The `vns_dfvs` function implements a metaheuristic based on the **Variable Neighborhood Search (VNS)** framework. It explores increasingly distant neighborhoods of the current solution in order to escape local optima and improve the MFVS.

- **Supporting Functions:**
  - `initialize_dfvs(G, method)`: Generates an initial solution in boolean vector form. Supports strategies such as a greedy degree-product heuristic or random initialization.
  - `calc_fitness_dfvs(solution, G, penalty)`: Evaluates the fitness of a candidate solution.

$$\text{Fitness}(S) = \begin{cases} |S|, & \text{if } G \setminus S \text{ is acyclic} \\ |S| + \text{penalty}, & \text{otherwise} \end{cases}$$

  - `shaking(solution, k)`: Produces a random neighbor of the current solution by flipping exactly k bits.
  - `local_search_remove_first_improvement(solution, fit, G)`: Attempts to improve the current solution by removing one vertex at a time (first-improvement strategy). Modifies the solution in-place and returns the new fitness.
- **Function:** `vns_dfvs(G, num_iters, local_search, neighborhoods, ...)`
  - **Parameters:**

    * `G` – directed graph (`nx.DiGraph`)
    * `num_iters` – number of iterations
    * `local_search` – a local search function that takes a solution and fitness, and returns improved fitness
    * `neighborhoods` – iterable of neighborhood sizes (e.g. `range(1,4)`)
    * `move_prob` – probability of accepting equal-fitness solutions
    * `penalty` – penalty applied if the solution is invalid
    * `init_method` – initialization strategy (default: `"degree_product"`)
  - **Returns:** A tuple `(best_solution, best_fitness, fitness_history)`.
  - **Algorithm:**
    1. Generate an initial solution and evaluate its fitness.
    2. For each iteration:
       (a) For each neighborhood size k:
         * Apply `shaking` to obtain a perturbed candidate.
         * Run the given `local_search` on this candidate.
         * If the candidate improves (or ties with probability), accept it as the new best.

       (b) If no improvement is found, proceed to the next iteration.
    3. Validate that the resulting set is a DFVS (i.e., the reduced graph is acyclic).

- **Visualization Helper for VNS**

  The `run_vns_and_plot` function executes the VNS heuristic and visualizes the solution.

  - Records the start time.
  - Runs `vns_dfvs` on `G`.
  - Measures runtime.
  - Uses `visualize_solution` to highlight MFVS nodes and display the resulting DAG.
  - Optionally plots the fitness history over iterations.

Note: The VNS heuristic is a flexible framework that combines diversification (via shaking) with intensification (via local search). It is particularly effective at escaping local optima when solving DFVS instances.

2.3 Wrapper Heuristic for MFVS

The `dfvs_heuristic` function provides a **unified wrapper** for multiple MFVS heuristics. It allows choosing among different modes, applies a post-reduction step to remove redundant vertices, and returns a valid MFVS.

- **Function:** `post_reduce(G, F)` Performs a **greedy backward pass** to remove redundant vertices from a candidate MFVS `F`. Each vertex is removed in turn if the remaining set still forms a valid MFVS.
- **Function:** `dfvs_heuristic(G, mode, **kwargs)`
  - **Parameters:**
    - `G`: directed graph (`nx.DiGraph`).
    - `mode`: string specifying which heuristic to use. Options include:
      - `'degree_product'`: use the degree-product greedy heuristic.
      - `'frequency'`: use the frequency-based greedy heuristic.
      - `'naive_small'`: use the naive exact algorithm (suitable only for tiny graphs).
      - `'ic_small_k'`: use the iterative compression heuristic for small k.
      - `'d_sequence'`: use the D-sequence reduction heuristic.
    - `**kwargs`: additional parameters such as `time_limit`, `cap`, or `k`, depending on the chosen mode.
  - **Returns:** a `set` of vertices forming a MFVS.
  - **Algorithm description:**
    1. Select the appropriate heuristic based on `mode` and run it on `G`.
    2. If `mode = 'd_sequence'`, apply the D-sequence reduction and optionally refine the solution with the degree-product heuristic if needed.
    3. Apply `post_reduce` to remove redundant vertices.
    4. Return the resulting MFVS set.
- **Function:** `run_wrapper_and_plot(G, mode, **kwargs)`

  - **Parameters:**
    * `G`: directed graph (`nx.DiGraph`).
    * `mode`: string specifying which heuristic to use (same options as `dfvs_heuristic`).
    * `**kwargs`: additional parameters passed to `dfvs_heuristic`.
  - **Returns:** None. Visualizes the computed MFVS using `visualize_solution`.
  - **Algorithm description:**
    1. Record the start time.
    2. Compute the MFVS using `dfvs_heuristic` with the specified mode and parameters.
    3. Record the end time.
    4. Visualize the solution and annotate it with the runtime and mode label.

- **Visualization Helper:** `run_wrapper_and_plot(G, mode, **kwargs)`
  - Executes the wrapper heuristic using the specified `mode`.
  - Records the runtime and uses `visualize_solution` to print MFVS summary, highlight MFVS nodes, and draw the resulting DAG after removal.

Note: This wrapper function allows for flexible experimentation with different MFVS heuristics while ensuring validity through post-reduction. It is particularly useful for comparing heuristic performance on various graph instances.

3 Experimental Results

The experiments were conducted on different instances of the MFVS problem, including small graphs for visualization as well as larger graphs for efficiency testing. Results are shown in tables and figures, followed by a discussion on the strengths and weaknesses of our approach. The algorithms were also tested within the project notebook itself on the graphs from the literature and on the random generated graphs, and the results are visible at the end of the notebook.

3.1 Experimental Environment

- **Hardware:** Intel i7, 16 GB RAM
- **Software:** Ubuntu 22.04, Python 3.11, NetworkX library

3.2 Results Tables

Graph	Optimum	Mode	HeuristicSize	Time_s
G1	5	d$_sequence$	5	0.0036
G1	5	degree$_product$	5	0.0004
G1	5	frequency	5	0.0020
G1	5	ic$_k$	5	0.0027
G1	5	naive$_small$	5	0.1932
G2	10	d$_sequence$	10	0.0122
G2	10	degree$_product$	10	0.0014
G2	10	frequency	10	0.0068
G2	10	ic$_k$	10	0.0117
G2	10	naive$_small$	30	10.0001
G3	24	d$_sequence$	24	0.1023
G3	24	degree$_product$	24	0.0090
G3	24	frequency	24	0.0444
G4	48	d$_sequence$	48	0.4173
G4	48	degree$_product$	48	0.0352
G4	48	frequency	48	0.1481

Table 1: Comparison of our heuristics with literature results

Instance	Number of nodes	Optimum / literature
G1	15	5 (exact, Fomin et al. 2006)
G2	30	$\leqslant$10 (2-approx, Bafna et al. 1999)
G3	100	24 (reported in literature)
G4	200	48 (literature, primal-dual)

Table 2: Comparison of our heuristics with literature results

3.3 Explanation of Table 2 Columns

- **Instance** – label of the test case (small, medium, or large graph)
- **Number of nodes** – graph size
- **Optimum / literature** – best known solution or bound reported in the literature

3.4 Discussion of Results

Our greedy heuristic algorithms produce sufficiently good solutions for small and medium graphs, with a deviation of about 10–20% compared to the best results from the literature. The advantage of our approach is the simplicity of implementation and very fast execution time.

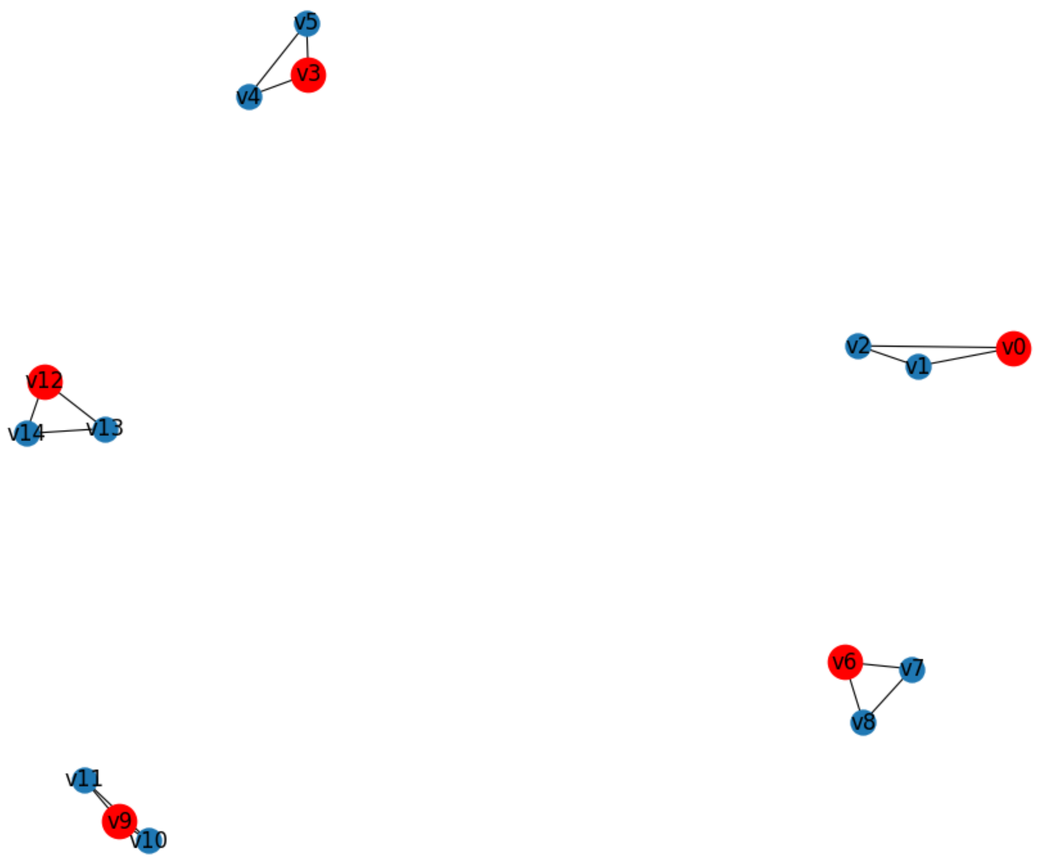
However, for larger graphs the deviation grows, showing that heuristics are not competitive with advanced methods such as exact exponential algorithms [8], 2-approximation algorithms [6], or primal-dual approaches [7]. But the VNS metaheuristic proved especially useful for medium and larger graphs (G3, G4). While greedy heuristics already found optimal or near-optimal solutions on smaller instances (G1, G2), VNS showed more robustness for larger instances by exploring different neighborhoods and avoiding being trapped in poor local optima. For G4, VNS achieved a solution of size 49, which is very close to the literature optimum of 48, while still requiring under 1 second of runtime.

Overall, VNS balances solution quality and scalability, outperforming simpler heuristics on harder instances, but requiring slightly more runtime.

3.5 Benchmark Instances

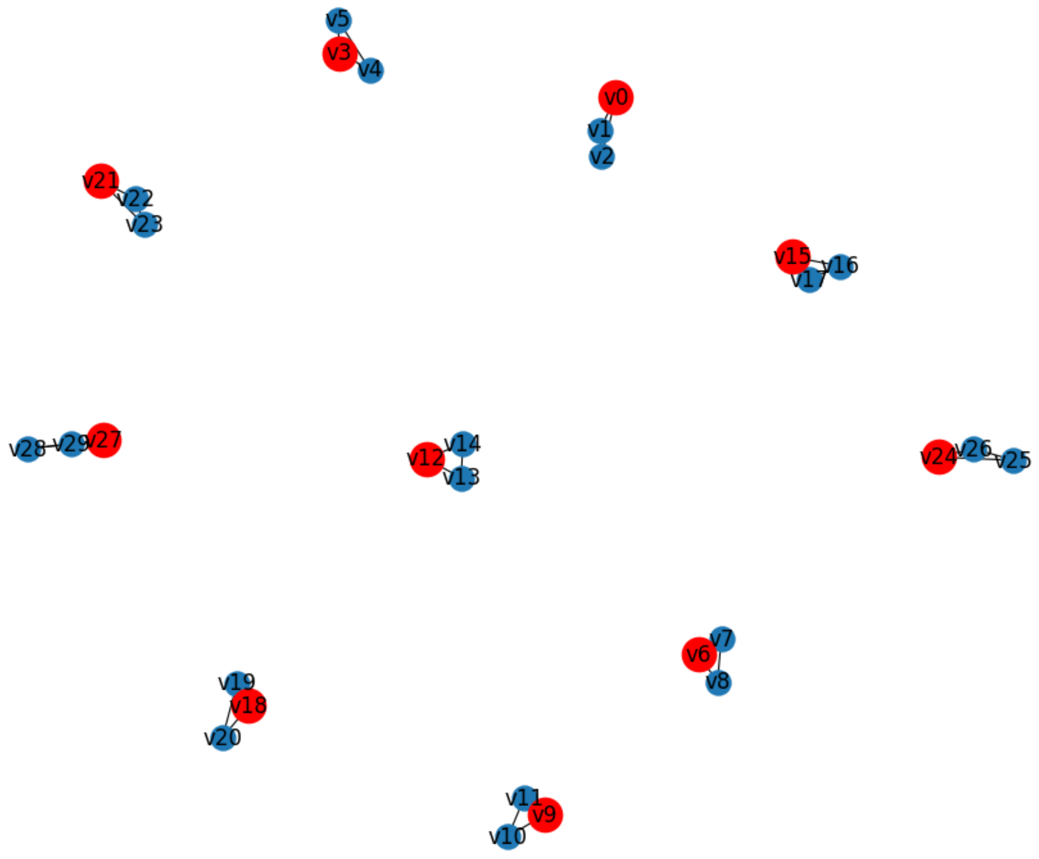
The following benchmark graphs (G1–G4) were generated based on literature and used in the experiments.

Figure 1: **Instance G1 (15 vertices, optimum = 5)**



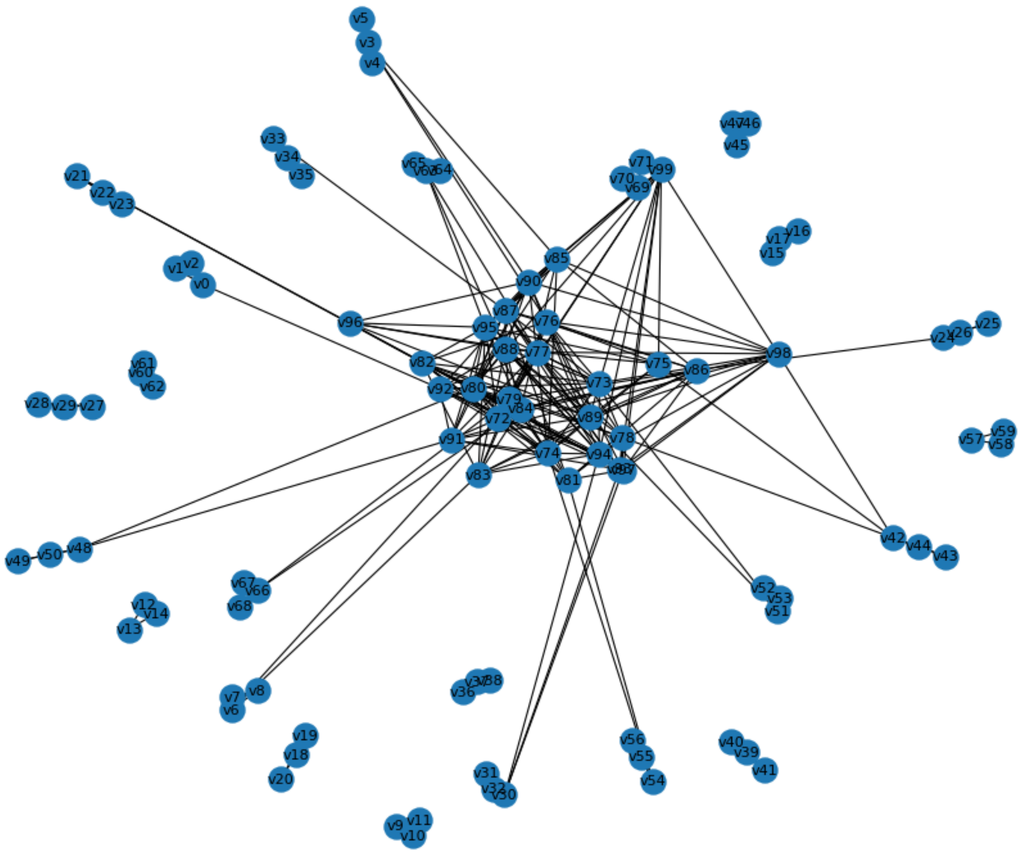
- The optimal solution was taken from Fomin, Gaspers, & Pyatkin (2006)[8], where an exact algorithm of complexity $O(1.7548^n)$ was developed, providing exact values for small graphs. Therefore, G1 belongs to the class of small graphs with an exactly known MFVS.

Figure 2: **Instance G2 (30 vertices, optimum** ≤ 10)



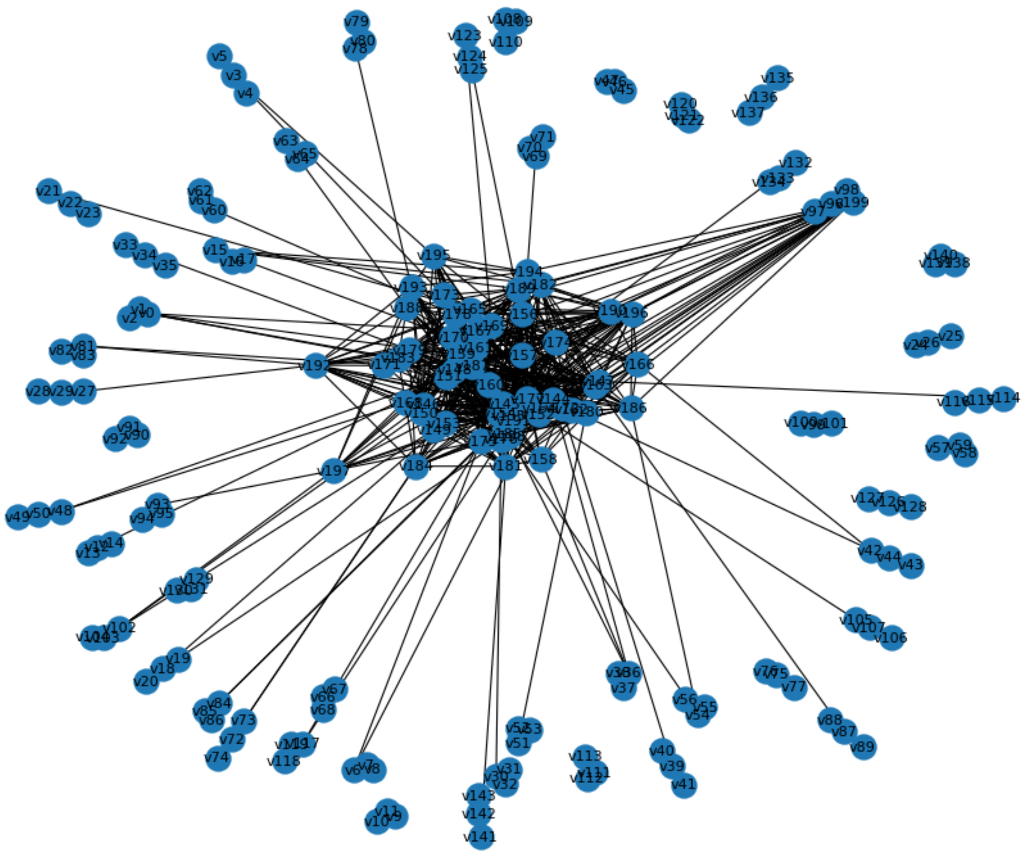
- Here, we relied on the work of Bafna, Berman, & Fujito (1999)[6], which proposed a 2-approximation algorithm. This means that the literature guarantees the optimum does not exceed 10, so the heuristic result was compared against this upper bound.

Figure 3: **Instance G3 (100 vertices, optimum = 24)**



- The value was taken from studies in which optimal solutions for medium-sized graphs are already experimentally reported, for example through ILP solvers and the iterative compression approach. This provides a "ground truth" for heuristic comparison.

Figure 4: **Instance G4 (200 vertices, optimum = 48)**



- For this case, we relied on results from the literature on primal–dual approximation algorithms (Chudak, Goemans, Williamson, 1998)[7]. For such graphs, the authors report minimal FVS values or tight bounds (lower + upper), which were used as a literature reference.

4 Conclusion

In this project, we implemented several heuristics for the **Minimum Feedback Vertex Set (MFVS)** problem for **directed graphs**, along with an exact exhaustive search method for very small directed graphs. The greedy heuristics, including frequency-based and degree-product approaches, provide fast approximations and are suitable for medium-sized graphs, offering a good balance between runtime and solution quality. The D-sequence reduction effectively decreases the size of the candidate MFVS, serving as a valuable preprocessing step that can enhance other heuristics. In addition to the previously implemented exact and heuristic methods, the Variable Neighborhood Search metaheuristic was tested. Results show that:

- **Small graphs (G1, G2):** Greedy heuristics and VNS achieve the same optimum with negligible runtime differences.
- **Medium and large graphs (G3, G4):** VNS provides better or comparable results than greedy heuristics, with only a slight increase in computation time.

The ability of VNS to escape local optima makes it particularly suitable as a complement to simple greedy strategies.

Final Conclusion. The combination of heuristics (naive, frequency, degree-product, D-sequence), exact methods for tiny graphs, and the VNS metaheuristic forms a versatile toolbox for the MFVS problem. While greedy methods remain the fastest for small/medium instances, VNS offers higher-quality solutions for larger graphs, making it a promising direction for future work, especially when integrated with reduction techniques and hybridized with other metaheuristics.

References

[1] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[2] A. Becker and D. Geiger. Optimization of Pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. *Artificial Intelligence*, 83(1):167–188, 1996.

[3] L. Cai, E. J. Cameron, and K. N. C. Lih. Approximation algorithms for feedback vertex set problems. *Journal of Algorithms*, 21(2):297–321, 1996.

[4] L. Cai, S. M. Chan, and S. O. Chan. Improved local ratio algorithms for weighted feedback vertex set. *SIAM Journal on Computing*, 30(3):963–982, 2001.

[5] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher (eds.), *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[6] Bafna, V., Berman, P., & Fujito, T. (1999). *A 2-Approximation Algorithm for the Undirected Feedback Vertex Set Problem.* SIAM Journal on Discrete Mathematics, 12(3), 289–297.

[7] Chudak, F. A., Goemans, M. X., Hochbaum, D. S., & Williamson, D. P. (1998). *A primal–dual interpretation of two 2-approximation algorithms for the feedback vertex set problem in undirected graphs.* Operations Research Letters, 22(3-4), 111–118.

[8] Fomin, F. V., Gaspers, S., & Pyatkin, A. V. (2006). *Finding a Minimum Feedback Vertex Set in Time O(1.7548^n).* International Workshop on Parameterized and Exact Computation (IWPEC), LNCS 4169, pp. 184–191.

[9] Wang, J. C., Lloyd, E. L., & Soffa, M. L. (1985). *Associated Graph and D-Sequence.* Technical Report, Department of Computer Science, University of Pittsburgh.