Jagiellonian University
Department of Theoretical Computer Science

Bartosz Procyk

Implementation of maximum flow algorithms

Bachelor Thesis

Supervisor: dr hab. Krzysztof Turowski

June 2025

Contents

Chapter 1

Introduction

The maximum flow problem is a fundamental optimization problem in network theory, concerned with determining the greatest possible flow that can be sent through a directed graph from a designated *source* node to a *sink* node, while respecting the capacity constraints of each edge. This problem models real-world scenarios where a commodity-such as data packets, water, electricity, or railway traffic-must be transported efficiently through a network with limited throughput.

The problem was first formally studied in 1954 by Harris and Ross [1], who framed it as a simplified model for analyzing Soviet railway traffic flows. Shortly after, Ford and Fulkerson [2] introduced the first algorithm to solve this problem, now known as the *Ford-Fulkerson* algorithm.

Over the decades, a variety of algorithmic paradigms have emerged to solve the maximum flow problem more efficiently:

- **Augmenting Path Methods**: This approach iteratively increases flow along paths with available capacity until no further improvements are possible, laying the foundation for subsequent advancements. The *Ford-Fulkerson* algorithm, though simple, lacks a polynomial runtime guarantee. The *Edmonds-Karp* variant [3] improved this by using shortest paths, achieving $O(|V||E|^2)$ time. *Dinic's* algorithm [4] further optimized this approach using *blocking flows*, reducing runtime to $O(|V|^2|E|)$. In 1978 Malhotra, Kumar, Maheshwari [5] optimized the *Dinic*'s algorithm to run in $O(|V|^3)$ time.

- **Preflow-Push Methods**: Goldberg and Tarjan's *push-relabel* algorithm [6] introduced *preflows*, which allow intermediate vertices to temporarily hold excess flow by relaxing the conservation constraint. This method achieves faster computation with $O(|V|^2|E|)$ or $O(|V|^3)$ complexity.

- **Capacity Scaling**: By focusing on high-capacity paths first, scaling techniques improve practical performance. The idea is to start by setting some parameter Δ to a large value, and introduce all edges that have residual capacity at least Δ. After pushing all possible flow through these edges we divide Δ in half. An example algorithm is *Ahuja-Orlin's* algorithm [7], running in $O(|V||E| + |V|^2 \log |f|)$ time.

- **Electrical Flow Methods**: Recent advances use ideas from numerical linear algebra and resistance networks to approximate flows in nearly-linear time [8].

Further breakthroughs have continued to push the theoretical boundaries of the maximum flow problem. In 2013 *Orlin* [9] presented a strongly polynomial algorithm with running time $O(|V||E|)$, which was the fastest known at the time for dense graphs. More recently, Chen et al. [10] introduced an almost-linear time algorithm for maximum flow and minimum-cost flow problems, representing a major advance in both theory and practical applicability for very large-scale networks.

These developments have led to efficient and practical methods for solving maximum flow problems in diverse applications ranging from network design [11] and logistics to data mining [12] and computer vision [13].

Table 1: Maximum Flow Algorithms

Name	**Year**	**Complexity**
Ford-Fulkerson	1956	$O(\|E\| \cdot \|f\|)$
Edmonds-Karp	1972	$O(\|V\|\|E\|^2)$
Dinic	1970	$O(\|V\|^2 E)$
MKM (Malhotra, Kumar, Maheshwari)	**1978**	$O(\|V\|^3)$
Ahuja-Orlin	1987	$O(\|V\|\|E\| + \|V\|^2 \log \|f\|)$
Push-Relabel (FIFO selection)	**1988**	$O(\|V\|^3)$
Boykov-Kolmogorow	**2004**	$O(\|V\|^2\|E\| \cdot \|f\|)$
Orlin	2013	$O(\|V\|\|E\|)$
Chen et al.	2022	$O(\|E\|^{1+o(1)})$

The implemented algorithms highlighted in bold. By $|f|$ we denote the value of the maximum flow.

1.1 Definitions

Definition 1.1. *A directed graph $G = (V, E)$ consists of a set of vertices (also referred to as nodes in this paper) V and set of edges E, whose elements are ordered pairs of vertices.*

Definition 1.2 (from [14])**.** *A flow network $G = (V, E)$ is a directed graph, capacity function $c(e)$, which assigns each edge (u, v) a nonnegative value and two distinct vertices called a source s and a sink t. We assume that for each edge $(u, v) \in E$, we* do not *have $(v, u) \in E$.*

We can work around the above restriction in general case: for each pair of edges $(u, v), (v, u) \in G$ we consider a new vertex w, remove (v, u) from E and add two edges (v, w) and (w, u) with the same capacities as (v, u).

Definition 1.3 (from [14]). *A flow is a function f that assigns each edge a nonnegative value and satisfies:*

$$0 \leq f(u, v) \leq c(u, v)$$

$$\textit{for all } u \in V \setminus \{s, t\}, \quad \sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

The value of a flow is equal to: $|f| = \sum_{v \in V} f(s, v)$, i.e, the output flow of s.

Definition 1.4 (from [14]). *The* maximum flow *problem can therefore be formulated as follows. Given a flow network G with capacity c, source s and sink t, we wish to find the maximum possible value of a flow.*

Definition 1.5 (from [14]). *Let $G = (V, E)$ be a flow network with capacity function $c : V \times V \to \mathbb{R}_{\geq 0}$ and flow function $f : V \times V \to \mathbb{N}$, satisfying the flow constraints. The* residual graph *$G_f = (V, E_f)$ with respect to flow f is defined as follows:*

For each edge $(u, v) \in E$, the residual graph G_f includes:

- *a **forward edge** (u, v) with residual capacity $c_f(u, v) = c(u, v) - f(u, v)$, if $c(u, v) - f(u, v) > 0$,*
- *a **backward edge** (v, u) with residual capacity $c_f(v, u) = f(u, v)$, if $f(u, v) > 0$.*

Thus, the residual graph represents all possible directions in which flow can be augmented: forward along unused capacity and backward to cancel existing flow.

Definition 1.6 (from [14]). *An* augmenting path *is a path P from the source node s to the sink node t in the residual graph G_f, such that every edge on the path has positive residual capacity i.e., $c_f(u, v) > 0$ for each edge (u, v) in P. The minimum residual capacity along this path determines the maximum additional flow that can be pushed through the network.*

Definition 1.7 (from [14]). *An S–T* cut *of a flow network $G = (V, E)$ is a partition of the vertices V into two disjoint sets S and $T = V \setminus S$, such that the source node $s \in S$ and the sink node $t \in T$.*

The capacity *of the cut (S, T) is defined as:*

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

where $c(u, v)$ is the capacity of the edge from u to v.

We can now state the most important theorem in the network flow theory, formulated in 1956 by Ford and Fulkerson:

Theorem 1.1.1 (Max-Flow Min-Cut Theorem [15])**.** *For any flow network, the maximum value of a flow from source s to sink t is equal to the minimum capacity over all $S - T$ cuts in the network.*

When analizing correctness of maximum flow algorithms we often have an assumption that the network after termination does not contain an augmenting path. The following statement proves that in this case the flow is indeed maximal.

Theorem 1.1.2. *If there is no path between s and t in the residual graph G_f then the value of the flow is maximal.*

Proof. Assume that there is no path between s and t in the residual graph G_f. Then we can consider a set S of all nodes reachable from s in G_f. Let $T = V \setminus S$. We know that $t \in T$, and for all $(u, v) \in (S, T)$ we have $f(u, v) = c(u, v)$, because otherwise v would be reachable from s in G_f via a residual edge (i.e., there would be remaining capacity). The total flow across the cut is:

$$|f| = \sum_{u \in S} \sum_{v \in T} f(u, v) = \sum_{u \in S} \sum_{v \in T} c(u, v) = c(S, T),$$

i.e., the value of the flow equals the capacity of the cut.

By the Max-Flow Min-Cut Theorem, since we have found a cut whose capacity equals the current flow, the flow is maximal. □

Chapter 2

Push Relabel Algorithm

2.1 Introduction

The push-relabel algorithm, introduced by Goldberg and Tarjan [6] in the late 1980s, represented a significant conceptual shift in maximum flow computation. Unlike classical augmenting path algorithms such as Ford-Fulkerson or Edmonds-Karp, which focus on iteratively augmenting flow along paths from the source to the sink, the push-relabel method operates by locally redistributing excess flow within the network.

This approach introduces the notion of a *preflow* and employs *distance labels* to guide the flow towards the sink, allowing for more flexible and efficient manipulation of intermediate states. Unlike augmenting path algorithms, the Push-Relabel method temporarily violates flow conservation but never exceeds edge capacities. The push-relabel framework brought new insights into how flow could be pushed in a network without requiring strictly augmenting paths at every step. It enabled highly efficient implementations and performance improvements in a variety of settings.

While the algorithm maintains strong polynomial time guarantees, its practical performance surpasses that of all previous methods [16]. This efficiency has made push-relabel a popular choice not only in classical network optimization problems but also in modern applications such as network design, and data mining, where large-scale graphs with complex structures arise.

2.2 Definitions

Definition 2.1 (from [6]). *A preflow $f : E \to \mathbb{R}_+$ is a function satisfying a weaker constraints than a flow function:*

$$0 \leq f(u,v) \leq c(u,v)$$

$$\forall u \in V \setminus \{s,t\}, \quad \sum_{v \in V} f(v,u) \geq \sum_{v \in V} f(u,v)$$

That is, the total flow into any vertex $v \neq s$ is at least as great as the total flow out of v.

Definition 2.2 (from [6])**.** *We call* excess flow *$excess(u)$ the difference in incoming and outgoing flow*

$$excess(u) = \sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v)$$

If the vertex has a positive excess flow it is called active.

Definition 2.3 (from [6])**.** *Let $G = (V, E)$ be a flow network with source s and sink t. A* labeling *or* distance function *is a function $d : V \to \mathbb{N} \cup \{\infty\}$ that assigns an integer (or ∞) to each vertex. A labeling d is called* valid *with respect to flow f if it satisfies the following conditions:*

- $d(s) = |V|$,
- $d(t) = 0$,
- *for every edge $(u, v) \in E$ such that $c_f(u, v) > 0$, it holds that $d(u) \leq d(v) + 1$.*

The value $d(v)$ serves as an estimate of the shortest distance from vertex v to the sink t in the residual graph. If $d(v) < |V|$, then $d(v)$ is a lower bound on the shortest-path distance from v to t in the residual graph G_f.

2.3 Algorithm

The algorithm begins by initializing a *preflow*: all outgoing edges from the source node s are saturated, pushing their full capacity to adjacent vertices. These neighbors then accumulate *excess flow* and become *active*. A *labeling function* is maintained, initialized to $d(s) = |V|$, $d(t) = 0$, and $d(v) = 0$ for all other vertices. This label provides a heuristic estimate of the shortest path from a vertex to the sink and determines directions for pushing flow.

While there exists an active vertex v, the algorithm performs a DISCHARGE operation. This procedure attempts to push excess flow from v to its neighbors along *admissible edges*, i.e., residual edges (v, u) such that $c(v, u) - f(v, u) > 0$ and $d(v) = d(u)+1$. For each admissible edge, the PUSH operation transfers flow from v to u, decreasing $excess(v)$ and increasing $excess(u)$. If u accumulates positive excess and is not s or t, it is added to the queue of active vertices.

If no admissible edge exists for v , a RELABEL operation is invoked. This increases $d(v)$ to one more than the minimum label among its neighbors connected by residual edges, thereby enabling future pushes. The DISCHARGE operation is repeated until the excess of v becomes zero or its label reaches $|V|$, at which point it can no longer participate in the flow propagation.

This process repeats until no active vertices remain. Since a vertex is active if it has positive excess, then at this point of the algorithm each vertex satisfies the flow conservation $\sum_{v\in V} f(v,u) - \sum_{v\in V} f(u,v) = 0$ and thus the preflow function is a valid flow.

Algorithm 1 Push$(v, e = (v,w))$

```
1: pushable ← min{$c(v,w) - f(v,w)$, excess[$v$]}
2: excess[$v$] ← excess[$v$] − pushable
3: excess[$w$] ← excess[$w$] + pushable
4: $f(e) \leftarrow f(e)$ + pushable
5: if excess[$w$] > 0 and $w \neq s$ and $w \neq t$ then
6:     queue.push($w$)
7: end if
```

Algorithm 2 Relabel(v)

```
1: min_label ← $\infty$
2: for each edge $(v,w) \in E$ do
3:     if $c(v,w) - f(v,w) > 0$ then
4:         min_label ← min {min_label, d($w$)}
5:     end if
6: end for
7: if min_label < $\infty$ then
8:     d($v$) ← min_label + 1
9: end if
```

Algorithm 3 Discharge(v)

```
1: while excess($v$) > 0 do
2:     while nextEdge($v$) < deg($v$) do ▷ Index of last saturated outgoing edge
3:         $u$ ← getIthNeighbor($v$, nextEdge($v$))
4:         if $c(v,u) - f(v,u) > 0$ and d($v$) = d($u$) + 1 then
5:             Push($v$, $(v,u)$)
6:             if excess[$v$] = 0 then
7:                 return
8:             end if
9:         end if
10:        nextEdge($v$) ← nextEdge($v$) + 1
11:    end while
12:    Relabel($v$)
13:    nextEdge($v$) ← 0
14:    if d($v$) ≥ $|V|$ then
15:        break
16:    end if
17: end while
```

2.4 Correctness and Complexity

We start by stating a few simple lemmas taken from [6]:

Lemma 2.1 (Lemma 3.3 in [6]). *If f is a preflow and d is any valid labeling of nodes, then the sink t is not reachable from the source s in the residual graph G_f. In other words an augmenting path doesn't exist.*

Proof. The label function satisfies $d(u) \leq d(v) + 1$ if the residual capacity $c_f(u, v) > 0$. An augmenting path from s to t will have a length of at most $|V| - 1$, and each edge can decrease the label by at most by one, which is impossible when the first label $d(s) = |V|$ and the last label is $d(t) = 0$. □

Lemma 2.2 (Lemma 3.5 in [6]). *If f is a preflow and v is a vertex with positive excess, then the source s is reachable from v in the residual graph G_f.*

Proof. Let S be the set of vertices reachable from v in the residual graph G_f, and suppose the source $s \notin S$. Let $\bar{S} = V \setminus S$. By the choice of S, for every edge (u, w) with $u \in \bar{S}$ and $w \in S$, there is no residual capacity from u to w. That is, $f(u, w) \leq 0$ for all such edges. Now consider the total excess in S:

$$\sum_{w \in S} excess(w) = \sum_{\substack{u \in V \\ w \in S}} f(u, w) = \sum_{\substack{u \in S \\ w \in S}} f(u, w) + \sum_{\substack{u \in \bar{S} \\ w \in S}} f(u, w).$$

The first term, $\sum_{u,w \in S} f(u, w)$, is zero by antisymmetry of flow: every internal flow is canceled by its reverse. The second term satisfies: $\sum_{\substack{u \in \bar{S} \\ w \in S}} f(u, w) \leq 0$, by the construction of S. Combining these two we get $\sum_{w \in S} excess(w) \leq 0$.

But since f is a preflow, all nodes except the source have non-negative excess, and $v \in S$ has $excess(v) > 0$, which contradicts $\sum_{w \in S} excess(w) \leq 0$. Therefore, the assumption $s \notin S$ must be false, and so s is reachable from v in the residual graph. □

Lemma 2.3 (Lemma 3.6 in [6]). *For any vertex v, the label $d(v)$ never decreases. Each relabeling operation applied to v increases $d(v)$ by at least 1.*

Proof. The Relabel operation is called on a node u only when for all neighbours v such that $c_f(u, v) > 0$ and $d(u) \neq d(v) + 1$. But from the definition $d(u) \leq d(v) + 1$, so it must be that $d(u) \leq min\{d(v) \colon (u, v) \in E\}$. Thus when Relabel sets $d(u) = min\{d(v) \colon (u, v) \in E\} + 1$ it must increase the value by at least 1. □

Lemma 2.4 (Lemma 3.7 in [6]). *At any time during the execution of the algorithm for any vertex $v \in V$ the following is true:*

$$d(v) \leq 2|V| - 1.$$

Proof. A vertex v is only relabeled when it has $excess(v) > 0$. By Lemma 2.2, there exists a path in the residual graph G_f with at most $|V| - 1$ edges. Each vertex in the path can lower the label function by one at each step. So traversing the path from v to s decreases the label by at most $|V| - 1$, and ends up with $d(s) = |V|$. Therefore, $d(v) \leq 2|V| - 1$. □

Termination

If the algorithm terminates and all distance labels are finite, all vertices in $V \setminus \{s, t\}$ must have zero excess, because there are no active vertices. Therefore f must be a flow. By Lemma 2.1 there is no augmenting path between s and t. From Theorem 1.1.2 stated in the introduction this means the flow is maximal.

Complexity

To find the complexity of the algorithm we will distinguish three types of operations and count the number of times each operation is performed.

1. **Relabel operations**
2. **Saturating pushes** — pushes that saturate an edge, i.e., reduce its residual capacity to zero
3. **Non-saturating pushes**

Theorem 2.4.1 (Lemma 3.8 in [6])**.** *The number of relabel operations is* $O(|V|^2)$.

Proof. We know from Lemma 2.3 and Lemma 2.4 that each relabel operation increases label by at least one and each vertex has a label at most $2|V|-1$. Then the number of relabel operations is bounded by $(2|V| - 1)|V| = O(|V|^2)$. □

Theorem 2.4.2 (Lemma 3.9 in [6])**.** *The number of saturated pushes is* $O(|V||E|)$.

Proof. For each edge (v, u) after a saturated push has been performed, for it to happen again we must first have a push from (u, v). For a push to be possible $d(u)$ must grow by at least 2. Since $d(u)$ is at most $2|V|-1$, for each edge there will be $O(|V|)$ saturating pushes, bringing the total to $O(|V||E|)$. □

Theorem 2.4.3. *A pass over the queue is defined as a list of operations until all nodes which were on the queue have been removed. The first pass starts at the beginning of the algoritm. The number of passes over the queue is* $O(|V|^2)$.

Proof. Let $\Delta = \max\{d(v)\colon v \text{ is active}\}$. Consider the effect of a single pass over the queue on Δ. If no distance label changes during the pass, each vertex has its excess pushed to lower-labeled vertices, so Δ decreases during the pass. If Δ is not changed by the pass, then at least one vertex's label must increase by at least 1. Notice that if Δ increases, some vertex label must increase by at least as much as Δ increases. From Theorem 2.4.1 we can conclude that the total number of label increases is at most $2|V|^2$, so the number of passes in which Δ stays the same or increases is at most $2|V|^2$.

Since $\Delta = 0$ both at the beginning and end of the algorithm, the number of passes in which Δ decreases is also at most $O(|V|^2)$. Therefore, the total number of passes is $O(|V|^2)$. □

Theorem 2.4.4 (Corollary 4.4 in [6]). *The number of non-saturated pushes is* $O(|V|^3)$.

Proof. We notice that each non-saturating push is followed by a removal from the queue. Then since by Theorem 2.4.3 there is at most $|V|^2$ passes over the queue each node will be removed from the queue this many times. Summing over all nodes we get $O(|V|^3)$. □

Combining these results, the push-relabel algorithm with a FIFO queue runs in $O(|V|^3)$ time in the worst case.

Chapter 3

Malhotra-Kumar-Maheshwari Algorithm

3.1 Introduction

The MKM (Malhotra, Kumar, Maheshwari) [5] algorithm is an enhancement of Dinic's algorithm within the family of maximum flow methods based on layered networks and blocking flows. Like Dinic's approach, the MKM algorithm iteratively constructs a layered network from the residual graph and computes a blocking flow to push the maximum possible amount of flow through the network before reconstructing the layers.

Dinic's original blocking flow algorithm, running in $O(|V|^2|E|)$ time, employs depth-first search (DFS) to find augmenting paths within the layered network. Each DFS traversal saturates at least one edge, and since edges are not revisited excessively, the total complexity for finding a blocking flow is bounded by $O(|V||E|)$ per phase. With the amount of phases bounded by the number of nodes V, the overall complexity reaches $O(|V|^2|E|)$.

The MKM algorithm improves this stage of the algorithm by introducing a more efficient method to compute blocking flows in $O(|V|^2)$ time per phase, significantly reducing the bottleneck in Dinic's method. This is achieved through a careful combinatorial approach that systematically identifies minimal cuts within the layered network and pushes flow accordingly, rather than relying solely on augmenting path searches.

By improving the blocking flow computation, the MKM algorithm achieves an overall worst-case time complexity of $O(|V|^3)$, making it particularly advantageous in dense graphs or scenarios where multiple blocking flow computations are required.

3.2 Definitions

Definition 3.1. *A* blocking flow *is a flow such that every path from the source s to the sink t contains at least one* saturated *edge, i.e., an edge with zero residual capacity. Formally, for any s-t path, there exists an edge (u, v) on the path for which*

$$c_f(u, v) = c(u, v) - f(u, v) = 0.$$

Definition 3.2. *A* layered network *is a subgraph of the residual graph $G_f = (V, E_f)$ constructed as follows:*

Each vertex v is assigned a level value $level[v]$, which is the length of the shortest path from the source s to v, using only residual edges with positive capacity.

The layered network includes only those residual edges $(v, u) \in E_f$ such that

$$level[u] = level[v] + 1,$$

and the residual capacity $c_f(v, u) > 0$.

Definition 3.3. *A* flow potential *$p(v)$ is defined as*

$$p(v) = \min \left\{ \sum_{(v,w)\in E} (c(v, w) - f(v, w)), \sum_{(u,v)\in E} (c(u, v) - f(u, v)) \right\} \quad \text{for } v \neq s, t$$

$$p(s) = \sum_{(s,w)\in E} (c(s, w) - f(s, w))$$

$$p(t) = \sum_{(u,t)\in E} (c(u, t) - f(u, t))$$

Intuitively, the flow potential represents the amount of flow that can be pushed through a node.

Definition 3.4. *A node v is called a* reference node *if it has the minimal potential among all nodes:*

$$p(v) = \min_{u\in V} p(u)$$

Lemma 3.1. *Let r be the reference node in a layered network $G = (V, E)$ with flow f. Then the flow f can be augmented by $p(r)$ to obtain a new flow f' such that*

$$p(r) = 0.$$

Proof. Let the reference node r lie in layer i of the layered network. We now push $p(r)$ flow into r from nodes in layer $i-1$, and then push the same amount of flow out of r to nodes in layer $i+1$. Since the incoming capacity is at least $p(r)$, there exists enough capacity to route $p(r)$ units into r. Similarly, the outgoing capacity from r is at least $p(r)$, so we can send this flow out of r.

Next, observe that all nodes in layer $i+1$ have potential at least $p(r)$ (by the definition of the reference node). Therefore, any node w in layer $i+1$ that receives flow from r has enough flow potential to forward the received flow toward the sink in subsequent augmentation steps. The same holds for nodes in layer $i-1$: they have potential at least $p(r)$, so they are able to supply $p(r)$ flow to r. □

3.3 Per-Stage Algorithm

The algorithm is based on a previously stated lemma and proceeds iteratively. At each iteration, flow is redistributed through the network based on the potential of the reference node.

At the beginning of each iteration, a reference node is found in $O(|V|)$ time. If its potential is zero then we remove it from the graph and find new reference node. From that node, an amount of flow equal to its potential is pushed in both directions:

1. **Towards the sink:** The nodes are processed from the reference node to the sink in *topological order*. At each node, incoming flow is pushed through outgoing edges and edges are saturated one by one. At most one outgoing edge may receive additional flow without being saturated.

2. **Towards the source:** The same process is applied in the reverse direction.

Saturated edges are removed from the network, since no further flow can pass through them in later iterations. Nodes are removed if either all of their incoming or all of their outgoing edges have been deleted. Deleting a node implies deleting all of its incident edges. Below is the pseudocode for the described procedure.

Algorithm 4 MKM Flow Algorithm (Main Loop)

```
1: Initialize total flow $F \leftarrow 0$
2: while true do
3:     level ← BuildLayeredNetwork($s$)    ▷ Build layered network using BFS
4:     if sink $t$ is not reachable from source $s$ then
5:         break
6:     end if
7:     p ← ComputePotential()
8:     while Graph is non empty do
9:         $u \leftarrow \arg\min_v$ potential($v$)    ▷ node with minimal non-zero potential
10:        if p($u$) = 0 then
11:            DeleteNode($u$)
12:        else
13:            PushForward($u$, p($u$))
14:            PushBackward($u$, p($u$))
15:            $F \leftarrow F + \mathrm{p}(u)$
16:            DeleteNode($u$)
17:        end if
18:    end while
19: end while
20: return $F$
```

3.3.1 Complexity

In every iteration, either all incoming or all outgoing edges of the reference node become saturated and the node can be deleted from the graph. Therefore, the number of iterations is at most $|V|$.

To achieve optimal performance a structure for a graph which allows for $O(1)$ edge lookup and deletion must be used. During the i-th iteration the total work done is $O(|V| + E_i)$ where E_i be the number of edges deleted. Since a maximum of traversed edge remains per node after pushing flow (all other have been saturated and deleted from the network), and each edge is deleted at most once, we have: $\sum E_i \leq |E|$.

Summing over all iterations, the total complexity becomes:

$$O\left(\sum(|V| + E_i)\right) = O(|V|^2 + |E|) = O(|V|^2).$$

Algorithm 5 BuildLayeredNetwork(s)

```
1: Set level[$v$] $\leftarrow \infty$ for all $v \in V$
2: Initialize queue $q$
3: $q$.push($s$); level[$s$] $\leftarrow 0$
4: while $q$ is not empty do
5:     $u \leftarrow q$.front(); $q$.pop()
6:     for each $(u, w) \in E$ do
7:         if level[$w$] $= \infty$ and $f(u, w) < c(u, w)$ then
8:             level[$w$] $\leftarrow$ level[$u$] $+ 1$
9:             $q$.push($w$)
10:        end if
11:    end for
12: end while
13: return level
```

Algorithm 6 PushForward(u, $flow$)

```
1: Initialize queue $q$
2: Initialize array to_push[$v$] $\leftarrow 0 \quad \forall v \in V$
3: UpdatePotential($u$)
4: $q$.push($u$)
5: while $q$ is not empty do
6:     $v \leftarrow q$.front()
7:     $q$.pop()
8:     if to_push[$v$] $= 0$ then
9:         continue
10:    end if
11:    for each edge $e = (v, w)$ with level[$v$] $+ 1 =$ level[$w$] and $c_f(e) > 0$ do
12:        pushable $\leftarrow \min\{c_f(e), \text{to\_push}[v]\}$
13:        $f(e) \leftarrow f(e) +$ pushable
14:        UpdatePotential($v$)
15:        UpdatePotential($w$)
16:        if $w \neq t$ then
17:            $q$.push($w$)
18:        end if
19:        if $c_f(e) = 0$ then                          ▷ edge saturated
20:            DeleteEdge($e$)
21:        end if
22:    end for
23: end while
```

3.4 Correctness and Complexity

We prove the correctness of the algorithm by showing that it terminates with a valid maximum flow and analyzing its time complexity.

Correctness

Assume the algorithm terminates. Then, by construction, there is no path from the source s to the sink t in the residual graph. From Theorem 1.1.2 this means that the flow is maximal.

Lemma 3.2. *The number of iterations of the main loop is at most*

$$|V| - 1.$$

Proof. The above observation is based on the fact that each iteration increases the distance between s and t in the residual graph G_f. Then since the maximum distance is $|V|-1$ this gives an upper bound on the number of iterations. Below we present a sketch of the proof. For a complete proof refer to [17].

Let $level_i[v]$ denote the *level* value during i-th iteration. First we will prove that $level_{i+1}[v] \geq level_i[v]$, meaning the distances from s to each node are non-decreasing. Notice that after one iteration the only new edges (u, v) that might appear in the residual graph G_f had their reverse edge (v, u) in G_f in previous iteration (because only edges and its reverses on the paths from s to t in the layered graph had modified flow values). But then $level_i[u] = level_i[v] + 1$ and such edge cannot decrease the distance from s to u.

Similarly we can prove that $level_{i+1}[t] > level_i[t]$. We know already that $level_{i+1}[t] \geq level_i[t]$. Since each path of length $level_i[t]$ was saturated in iteration i then for the distance to not increase we would need a path containing some new edge (u, v) which satisfies $level_i[u] = level_i[v] + 1$. But because the *level* values are non-decreasing this path will take us from a node in layer k to a node in layer $k - 1$, thus increasing the path from s to t. □

Combining the two results above, we conclude that the algorithm always terminates and returns a valid maximum flow.

Complexity

The time complexity of finding the blocking flow is $O(|V|^2)$ (Section 3.3.1). The total number of iterations is $O(|V|)$, giving total time complexity of the algorithm of

$$O(|V|^2 \cdot |V|) = O(|V|^3)$$

.

Chapter 4

Boykov-Kolmogorov Algorithm

4.0.1 Introduction

Maximum flow algorithms have historically been developed and analyzed within the context of network optimization, with early applications in logistics, transportation, and communication networks [18]. In the 1990s, maximum flow algorithms found application in computer vision.

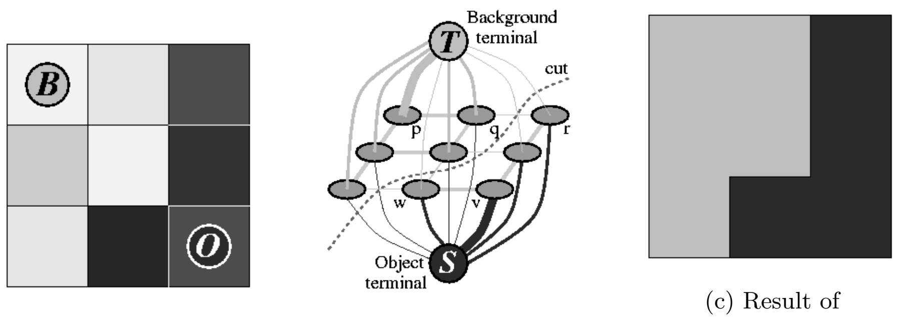
In computer vision, many fundamental tasks such as image segmentation, stereo correspondence, and multi-view reconstruction can be modeled as energy minimization problems. These energy functions are often reducible to graph cut formulations, where finding a minimum cut corresponds to solving a maximum flow problem. Traditional algorithms like Edmonds-Karp or Push-Relabel algorithm offer theoretical guarantees but struggle with the large, grid-structured graphs typical of vision problems.

Recognizing the specific structure of graphs instances for computer vision problems, Boykov and Kolmogorov proposed a max-flow algorithm optimized specifically for the structure and demands of vision applications. Their approach does not aim for the best worst-case complexity but instead achieves exceptional practical performance on the types of graphs encountered in computer vision.

4.0.2 Overview

Boykov-Kolmogorov [13] algorithm belongs to the class of maximum flow algorithms based on finding augmenting paths. In typical augmenting paths algorithms however the paths are searched using DFS or BFS and after each iteration a new search tree is built from scratch. Authors noticed that in computer vision problems this method is ineffective.

Instead of building new search tree for each augmenting path, the algorithm stores two search trees, one from the source and one from the sink. At each iteration one of the trees adopts a new node. An augmenting path is found once



(a) Image (b) Corresponding Graph (c) Result of segmentation

Figure 1: Example of a 2D segmentation for a 3x3 image. The edge weights in the corresponding graph align with pixel intensity. [19]

the trees "meet" each other. Once the path is saturated some nodes become disconnected from the search trees. A special procedure is implemented to either find a new path to the search trees or the nodes are removed from the trees.

4.0.3 Definitions

Definition 4.1. *Search trees S and T are trees with roots respectively at the source s and the sink t.*

In tree S all edges from each parent node to its children are non-saturated (have positive residual capacity), while in tree T edges from children to their parents are non-saturated. A node cannot belong to both trees at the same time. The nodes that are not in S or T are called free.

Definition 4.2. *A node v is called* active *if it is at the border of the search tree, ie it has a neighbour which does not belong to the same tree as v.*

A node that is not active is called passive.

Definition 4.3. *We define tree_capacity(u, v) as:*

$$\textit{tree\_capacity}(u,v) = \begin{cases} c_f(u,v) & \textit{if } u \in S, \\ c_f(v,u) & \textit{if } u \in T, \end{cases}$$

where $c_f(u, v)$ denotes the residual capacity of edge (u, v).

4.0.4 Algorithm

As described previously, the algorithm maintains two search trees, S and T, throughout its execution. They are represented using two arrays:

$$\textsc{tree}[v] = \begin{cases} S & \text{if } v \in S, \\ T & \text{if } v \in T, \\ \varnothing & \text{otherwise,} \end{cases}$$

and

$$\text{PARENT}[v] = \begin{cases} u & \text{if } u \text{ is the parent of } v \text{ in its respective search tree,} \\ \varnothing & \text{if } u \text{ is a free node or disconnected from its tree.} \end{cases}$$

The algorithm proceeds by repeatedly iterating a main loop until no augmenting path from the source to the sink exists. At that point, the maximum flow value has been determined and can be returned. The main loop consists of three stages, growth, augmentation and adoption, explained in detail below the pseudocode.

Algorithm 7 BOYKOV-KOLMOGOROV (MAIN LOOP)

```
 1: Initialize total flow $F \leftarrow 0$
 2: Initialize $S \leftarrow \{s\}$, $T \leftarrow \{t\}$
 3: Initialize $A \leftarrow \{s, t\}$                ▷ Active nodes
 4: Initialize $O \leftarrow \varnothing$            ▷ Orphan nodes
 5: while true do
 6:     $P \leftarrow$ GROW($A$)
 7:     if $P = \varnothing$ then
 8:         break
 9:     end if
10:     augmented_flow ← AUGMENT($P$)
11:     $F \leftarrow F+$ augmented_flow
12:     ADOPTORPHANS($O$)
13: end while
14: return F
```

1. Growth Stage. In this stage, active nodes are selected, and their corresponding search trees expand toward neighboring nodes through edges with positive residual capacity. Newly added nodes are as active. The stage terminates either when the trees cannot grow anymore or when an augmenting path from the source s to the sink t is found.

An augmenting path is detected when a node in tree S attempts to grow toward a node in tree T, or vice versa. The path is then reconstructed using the PARENT[v] array.

Algorithm 8 Grow(A)

```
while $A \neq \varnothing$ do
    $v \leftarrow A$.front()
    $A$.pop()
    found_path ← false
    for all each $(v, w) \in E$ do
        if found_path then
            break
        end if
        if $tree\_capacity(v, w) > 0$ then
            if TREE[$w$] $= \varnothing$ then
                TREE[$w$] ← TREE[$v$]
                PARENT[$w$] ← $v$
                active.push($w$)
            else if TREE[$w$] $\neq$ TREE[$v$] then
                if TREE[$v$] = SOURCE then
                    spath ← $v$, tpath ← $w$                ▷ Store the found path
                else
                    spath ← $w$, tpath ← $v$
                end if
                found_path ← true
                active.push($w$)
                break
            end if
        end if
    end for
    if found_path then
        return true
    end if
end while
return false
```

2. Augmentation Stage. Once an augmenting path is found, the algorithm computes the bottleneck capacity Δ along the path and pushes flow of value Δ through the network.

For every edge (p, q) on the augmenting path that becomes saturated, the algorithm performs the following:

- If both p and q belong to tree S, then q is disconnected from the search tree (i.e., PARENT[q] $\leftarrow \varnothing$) and added to the set of orphans O.
- If both p and q belong to tree T, then p is disconnected from the search tree and added to O.

This process may break the trees into disconnected forests, which are restored in the next stage.

Algorithm 9 AUGMENT(P)

```
 1: $\Delta \leftarrow \min_{e \in P}\{c_f(e)\}$          ▷ Find the bottleneck capacity $\Delta$ on path $P$
 2: for each edge $e \in P$ do
 3:     PUSH($e, \Delta$)                    ▷ Push flow $\Delta$ through all edges in $P$
 4: end for
 5: for each edge $(u, v) \in P$ do
 6:     if $tree\_capacity(u, v) = 0$ then
 7:         if TREE[$u$] = TREE[$v$] = $S$ then
 8:             PARENT[$v$] $\leftarrow \varnothing$
 9:             $O \leftarrow O \cup \{v\}$                    ▷ mark $q$ as orphan
10:         else if TREE[$u$] = TREE[$v$] = $T$ then
11:             PARENT[$u$] $\leftarrow \varnothing$
12:             $O \leftarrow O \cup \{u\}$                    ▷ mark $u$ as orphan
13:         end if
14:     end if
15: end for
16: return $\Delta$
```

3. Adoption Stage. In the adoption stage, the algorithm attempts to reattach orphaned nodes to their respective trees. For each orphan node, a new parent is looked for among its neighbors. A node q is a valid new parent of an orphan p if:

- q belongs to the same tree as p,
- the residual capacity $tree\_capacity(q, p)$ is positive,
- the path from q to the root of the tree is valid (i.e., all nodes along the path have defined parents).

If such a node q is found, the orphan is reattached to the tree by setting PARENT[p] $\leftarrow q$. If no such parent can be found, p becomes a free node, and any of its children in the tree are also added to the orphan set. This stage continues until all orphans have been processed.

Algorithm 10 ADOPT(O)

```
while $O$ is not empty do
    $u \leftarrow O$.front()
    $found\_new\_parent \leftarrow$ false
    for each edge $(u, v) \in E$ do
        if TREE[$v$] = TREE[$u$] and $tree\_capacity(u, v) > 0$ and $v$ has a valid path to root then
            PARENT[$u$] $\leftarrow v$
            $found\_new\_parent \leftarrow$ true
            break
        end if
    end for
    if not $found\_new\_parent$ then
        for each neighbor $v$ of $u$ do
            if TREE[$v$] = TREE[$u$] then
                if $tree\_capacity(u, v) > 0$ then
                    active.push($v$)
                end if
                if PARENT[$v$] = $u$ then
                    PARENT[$v$] $\leftarrow \varnothing$
                    $O$.push($v$)
                end if
            end if
        end for
        TREE[$u$] $\leftarrow \varnothing$
    end if
end while
```

4.0.5 Correctness and Complexity

We aim to establish two main properties of the algorithm: its termination and its computational complexity.

Correctness. To prove correctness, we must show that the algorithm terminates and produces a valid maximum flow.

- **Termination:** Clearly the main loop will be executed a finite number of times since each iteration finds an augmenting path with bottleneck capacity of at least 1. The growth and augment stage are linear in terms of network size and therefore terminate as well. The fact that the adoption stage is finite follows from the following lemma:

  Lemma 4.1. *Each node can become an orphan at most once in a single stage.*

Proof. Consider a processed orphan u. If u becomes a free node then by definition it will no longer take part in the adoption stage. Then assume that u receives a new parent v. The node v must satisfy the condition that all nodes along the path from v to the root have defined parents. Since a node can become an orphan in the adoption stage only if its ancestor becomes one and all ancestors of u have defined parents, then it is clear that u cannot become an orphan again. $\square$

- **Maximum Flow:** The algorithm terminates when there is no edges with positive residual capacities from nodes belonging to search trees. Then the trees are disconnected in the residual graph and therefore the flow is maximal.

Complexity. Unlike in algorithms such as Dinic's or the Malhotra-Kumar-Maheshwari (MKM) algorithm, the augmenting paths found by the Boykov-Kolmogorov algorithm are not necessarily shortest paths. Thus, the bounds established for those algorithms do not directly apply here.

The only general upper bound we can establish is the trivial one:

$$O(|V|^2|E||f|)$$

where $|f|$ is the value of the maximum flow. This follows from the fact that each augmentation increases the total flow by at least one (provided the capacites are integers), and each operation within a stage is polynomial with respect to the graph size.

Chapter 5

Implementation and Comparison

5.1 Introduction

The algorithms described above were implemented in C++ according to the presented pseudocodes. NetworKit and Koala libraries were used for graph data structures. The source code was added to the Koala-NetworKit GitHub repository.

5.2 Implementation Details

The algorithms are implemented in the following files of the Koala repository:

- `include/flow/MaximumFlow.hpp` - Header for all maximum flow algorithms
- `cpp/flow/PushRelabel.cpp` - Push Relabel algorithm
- `cpp/flow/MKMFlow.cpp` - Malhotra Kumar Maheshwari algorithm
- `cpp/flow/BKFlow.cpp` - Boykov-Kolmogorov algorithm
- `test/testMaximumFlow.cpp` - Simple unit tests
- `benchmark/benchmarkMaximumFlow.cpp` - Benchmark framework for testing algorithms on graphs provided in DIMACS [20] format

5.3 Results

5.3.1 Datasets

For testing, two datasets of graph instances were used. First being a set of real life examples of graphs used in computer vision problems and second dataset consists of synthetically generated graphs.

All benchmarks were conducted on a personal computer equipped with an Intel Core i7-6700HQ processor using the Ubuntu 20.04.6 LTS operating system.

5.3.2 Instances from Computer Vision

The first dataset constists of graph instances used in computer vision such as video segmentation, audio restoration, resolution upscaling and mesh segmentation [21]. The instances were taken from the Min-Cut/Max-Flow Problem Instances Library from the Technical University of Denmark [22]. A comprehensive comparison of different maximum flow algorithms designed specifically for computer vision problems on this dataset can be found here [23].

During testing a cutoff of 15 minutes (900 seconds) was set. Time execution of algorithms that failed to terminate in that time are marked with "X".

Dataset	**Vertices**	**Edges**	**PR**	**MKM**	**BK**
graph3x3	2,002	47,872	0.21	1.54	0.79
graph5x5	2,002	139,600	1.96	2.47	47.5
rome99	3,353	8,870	8.16	3.51	0.04
super_res-E1	10,494	62,364	21.74	22.50	0.11
super_res-E2	10,494	103,164	26.40	28.01	0.13
super_res-Paper1	10,494	62,364	23.55	21.98	0.09
texture-Temp	14,452	239,688	1.41	64.95	12.87
handsmall.segment	15,522	94,334	538.07	159.76	1.21
printed_graph8	16,322	289,096	376.32	368.46	1.87
printed_graph14	10,514	59,7100	X	160.26	876.19
printed_graph31	9,986	564,928	X	135.15	749.40
bunny.segment	97,526	730,938	X	X	194.71
texture-Cremer	44,034	783,128	X	X	554.83

Table 2: Benchmarking Results for Maximum Flow Algorithms. Time in seconds.

5.3.3 Random Graphs

The following dataset was created using the `washington.c` generator authored by Richard Anderson [24]. The source code for the generator can be found here [25]. The generator creates 9 different types of graphs of which the ones listed are used in our dataset:

- `Mesh Graph/Square Mesh`: A graph with grid like structure and terminals (sink and source) connected to the nodes in the first and last layer.
- `Random Level Graph`: A layered graph with edges between nodes in ith and $i+1$th layer.
- `Line Graph`: Similar to layered graph but with edges allowed between nodes in the same level and between layers i and $i+k$.
- `Matching Graph`: A bipartite graph.
- `Dinic Bad Case`: A graph that forces all $|V|-1$ iterations of the main loop in the Dinic (and MKM) algorithm.
- `Push Relabel Bad Case`: An empirical bade case for the push relabel algorithm.

Dataset	**Vertices**	**Edges**	**PR**	**MKM**	**BK**
mesh	2,502	7,450	5.30	5.73	0.18
mesh	5,627	16,800	26.93	25.54	0.56
mesh	15,627	46,750	395.13	607.72	3.96
matching	602	1,800	0.40	0.04	0.01
matching	4,002	12,000	22.28	2.89	0.02
matching	10,002	30,000	187.4	20.56	0.06
line	2,002	5,940	8.26	4.20	0.16
line	8,002	23,945	158.92	209.03	2.41
line	14,002	41,950	412.11	870.23	6.12
dinic_bad	300	597	0.01	0.55	0.01
dinic_bad	1,000	1,997	0.01	21.12	0.10
dinic_bad	2,000	3,997	0.01	152.62	0.55
dinic_bad	5,000	9,997	0.01	X	1.84
push_rel_bad	1,803	2,401	0.17	0.37	0.12
push_rel_bad	6,003	8,001	0.57	1.08	3.71
push_rel_bad	21,003	28,001	5.20	54.82	22.35
level	3,000	8,600	6.01	2.68	0.24
level	6,000	17,400	15.06	19.54	1.60
level	9,000	26,600	153.13	73.60	7.25
level	15,000	44,500	242.34	219.09	18.28

Table 3: Benchmarking Results for Maximum Flow Algorithms. Time in seconds.

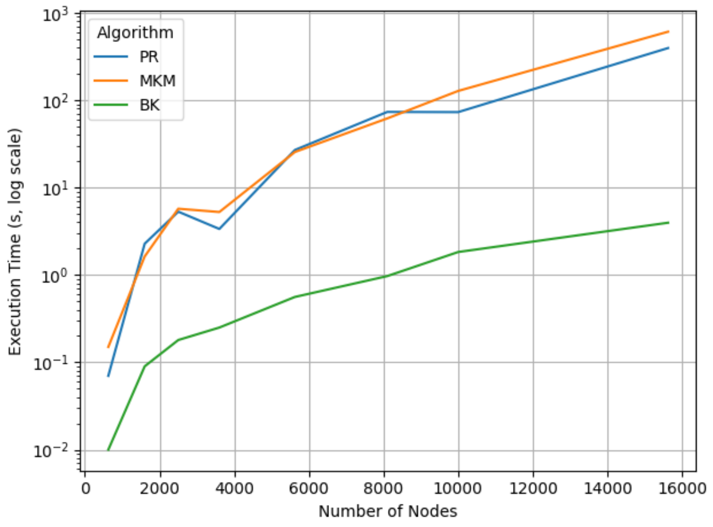


Figure 2: Mesh instances

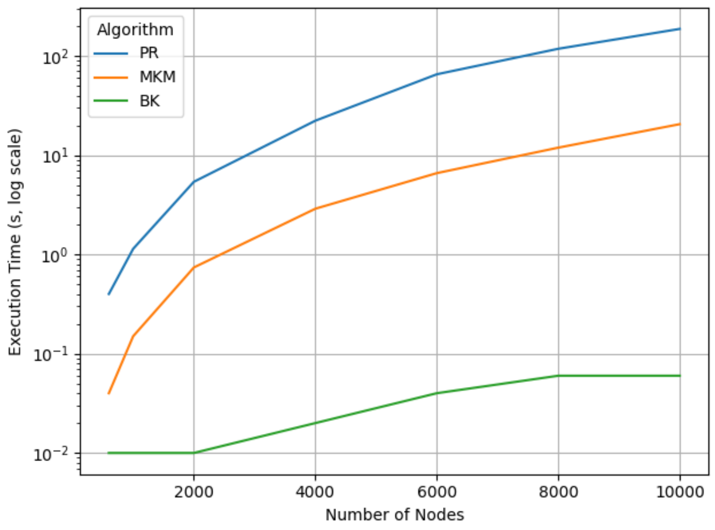


Figure 3: Matching instances with average degree of 4

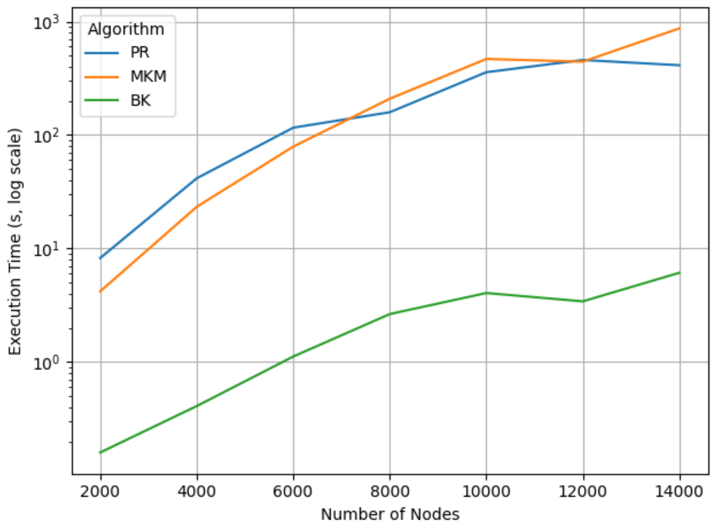


Figure 4: Line instances

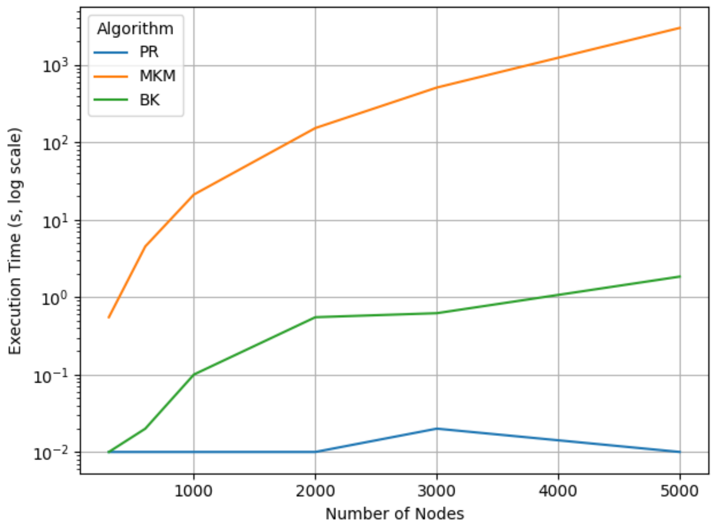


Figure 5: Bad case for Dinic instances

5.3.4 Results

To better understand the empirical time complexity of the three maximum flow algorithms, we performed nonlinear regression to fit models of the form $t = c \times n^d$, where t is the execution time and $n = |V|$ is the number of nodes in the graph. The fitted exponent d gives insight into how the runtime scales as the input size grows. Table Table 4 summarizes these results across different graph datasets.

The results indicate that the Boykov-Kolmogorov (BK) algorithm generally exhibits near-quadratic scaling ($d \approx 1.7 - 2.0$) on all tested graph types, reflecting superior practical scalability. In contrast, Push-Relabel (PR) and Malhotra-Kumar-Maheshwari (MKM) tend to have higher exponents, often between 2.2 and 3, with MKM showing particularly poor scaling (cubic) in challenging cases like the Dinic bad case graph.

Dataset	**Algorithm**	**Fitted Model**	**Interpretation**
Mesh	PR	$t \sim n^{2.50}$	Quadratic scaling, slow
	MKM	$t \sim n^{2.47}$	Quadratic scaling
	BK	$t \sim n^{1.78}$	Sub-quadratic, fastest scaling
Line	PR	$t \sim n^{2.12}$	Quadratic scaling
	MKM	$t \sim n^{2.78}$	Near cubic scaling
	BK	$t \sim n^{1.93}$	Near quadratic, better
Matching	PR	$t \sim n^{2.21}$	Quadratic scaling
	MKM	$t \sim n^{2.17}$	Quadratic scaling
	BK	$t \sim n^{0.73}$	Nearly linear, very efficient
Dinic Bad Case	PR	$t \sim n^{0.10}$	Almost constant
	MKM	$t \sim n^{3.02}$	Cubic scaling, worst case
	BK	$t \sim n^{1.95}$	Near quadratic scaling

Table 4: Fitted empirical complexity models of the form $t = c \times n^d$ for execution time t vs. number of nodes $n = |V|$.

The benchmarking results clearly show that the Boykov-Kolmogorov (BK) algorithm consistently outperforms both Push-Relabel (PR) and Malhotra-Kumar-Maheshwari (MKM) algorithms across almost all graph types and sizes. Its runtime remains exceptionally low even as the number of vertices and edges increases, indicating superior scalability and practical efficiency, particularly for structured graphs like mesh (Figure 2), matching , and line.

Push Relabel generally performs better than MKM algorithm on many datasets, especially in bigger instances. However, it performs poorly on the matching instances graphs (Figure 3). Interestingly, PR performed unexpectedly well on an instance it was theoretically supposed to struggle with. Push Relabel performs more steadily across all graph types, but tends to be slower overall, with particularly high runtimes in large-level and mesh graphs (Figure 2), making it less suitable for large-scale instances. MKM on the other performs slightly better than PR on small instances (Figure 4), but on larger graphs starts to stay behind. Its performance degrades significantly in the Dinic bad case graph (Figure 5), as expected, due to the nature of the graph forcing the maximum number of iterations in its main loop.

Overall, these results align well with what is observed in the corresponding performance graphs. BK is the clear winner in practical performance, MKM is effective on small instances but vulnerable to specific worst-case structures, and PR, while robust, lags in speed. This benchmarking underscores the importance of algorithm selection based on input structure in maximum flow applications.

Bibliography

[1] Theodore Harris and F. S. Ross. *Fundamentals of a Method for Evaluating Rail Net Capacities.* 1955.

[2] Lester Ford and Delbert Fulkerson. “Maximal Flow Through a Network”. In: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.

[3] Jack Edmonds and Richard Karp. “Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems”. In: *Journal of the Association for Computing Machinery* 19.2 (1972), pp. 248–264.

[4] Efim Dinic. “Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation”. In: *Soviet Mathematics Doklady* 11 (1970), pp. 1277–1280.

[5] Vishv Malhotra, M Pramodh Kumar, and Shachindra Maheshwari. “An $O(V^3)$ algorithm for finding maximum flows in networks”. In: *Information Processing Letters* 7.6 (1978), pp. 277–278.

[6] Andrew Goldberg and Robert Tarjan. “A New Approach to the Maximum-Flow Problem”. In: *Journal of the ACM (JACM)* 35.4 (1988), pp. 921–940.

[7] Ravindra Ahuja and James Orlin. “A Fast and Simple Algorithm for the Maximum Flow Problem”. In: *Operations Research* 37.5 (1989), pp. 748–759.

[8] Aleksander Madry. “Computing Maximum Flow with Augmenting Electrical Flows”. In: *Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS).* New Brunswick, NJ: IEEE, 2016, pp. 593–602.

[9] James Orlin. “Max flows in $O(nm)$ time, or better”. In: *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing (STOC).* Ed. by Dan Boneh, Tim Roughgarden, and Joan Feigenbaum. New York, NY: Association for Computing Machinery, 2013, pp. 765–774.

[10] Li Chen et al. “Maximum Flow and Minimum-Cost Flow in Almost-Linear Time”. In: *Journal of the ACM* 72.3 (2025), pp. 1–103.

[11] Earl Lee, John Mitchell, and William Wallace. "Restoration of Services in Interdependent Infrastructure Systems: A Network Flows Approach". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews* 37.6 (2007), pp. 1303–1317.

[12] Noriko Imafuji and Masaru Kitsuregawa. In: *IEICE Transactions on Information and Systems* 87.2 (2004), pp. 407–415.

[13] Yuri Boykov and Vladimir Kolmogorov. "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.9 (2004), pp. 1124–1137.

[14] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009.

[15] Lester Ford and Delbert Fulkerson. *Flows in Networks*. Princeton, NJ: Princeton University Press, 1962.

[16] Boris Cherkassky and Andrew Goldberg. "On Implementing Push-Relabel Method for the Maximum Flow Problem". In: *Algorithmica* 19 (1997), pp. 390–410.

[17] Shimon Even. *The Max Flow Algorithm of Dinic and Karzanov: An Exposition*. Tech. rep. MIT/LCS/TM-80. Laboratory for Computer Science, Massachusetts Institute of Technology, 1976.

[18] Dorothy Greig, Bruce Porteous, and Allan Seheult. "Exact Maximum A Posteriori Estimation for Binary Images". In: *Journal of the Royal Statistical Society Series B: Statistical Methodology* 51.2 (1989), pp. 271–279.

[19] Yuri Boykov and M.-P. Jolly. "Interactive Graph Cuts for Optimal Boundary & Region Segmentation of Objects in N-D Images". In: *Proceedings of the Eighth IEEE international Conference on Computer Vision. ICCV 2001*. Vol. 1. 2001, pp. 105–112.

[20] DIMACS. *DIMACS Maximum Flow Format*. Accessed: 2025-06-23. URL: `https://lpsolve.sourceforge.net/5.5/DIMACS_maxf.htm`.

[21] University of Waterloo. *Max-Flow problem instances in vision*. Accessed: 2025-06-23. URL: `https://vision.cs.uwaterloo.ca/data/maxflow`.

[22] Patrick Jensen, Niels Jeppesen, Anders Dahl, and Vedrana Dahl. *Min-Cut/Max-Flow Problem Instances for Benchmarking*. Accessed: 2025-06-23. 2022. URL: `https://data.dtu.dk/articles/dataset/Min-Cut_Max-Flow_Problem_Instances_for_Benchmarking/17091101`.

[23] Patrick Jensen, Niels Jeppesen, Anders Dahl, and Vedrana Dahl. "Review of Serial and Parallel Min-Cut/Max-Flow Algorithms for Computer Vision". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45.2 (2022), pp. 2310–2329.

[24] David Johnson and Catherine McGeoch. *Network Flows and Matching: First DIMACS Implementation Challenge*. Providence, RI: American Mathematical Society, 1993, pp. 1–17.

[25] Richard Anderson. *Washington Generator*. Accessed: 2025-06-23. URL: `http://archive.dimacs.rutgers.edu/pub/netflow/generators/network/washington`.