



# Engineering Edge Orientation Algorithms

Henrik Reinstädler  

Heidelberg University, Germany

Christian Schulz  

Heidelberg University, Germany

Bora Uçar  

CNRS and LIP, ENS de Lyon, France

UMR5668 (CNRS, ENS de Lyon, Inria, UCBL1), France

---

## Abstract

Given an undirected graph  $G$ , the edge orientation problem asks for assigning a direction to each edge to convert  $G$  into a directed graph. The aim is to minimize the maximum out-degree of a vertex in the resulting directed graph. This problem, which is solvable in polynomial time, arises in many applications. An ongoing challenge in edge orientation algorithms is their scalability, particularly in handling large-scale networks with millions or billions of edges efficiently. We propose a novel algorithmic framework based on finding and manipulating simple paths to face this challenge. Our framework is based on an existing algorithm and allows many algorithmic choices. By carefully exploring these choices and engineering the underlying algorithms, we obtain an implementation which is more efficient and scalable than the current state-of-the-art. Our experiments demonstrate significant performance improvements compared to state-of-the-art solvers. On average our algorithm is 6.59 times faster when compared to the state-of-the-art.

**2012 ACM Subject Classification** Theory of computation → Design and analysis of algorithms

**Keywords and phrases** edge orientation, pseudoarboricity, graph algorithms

**Digital Object Identifier** 10.4230/LIPIcs.ESA.2024.97

**Related Version** *Full Version*: <https://arxiv.org/abs/2404.13997> [29]

**Supplementary Material** *Software (Source Code)*: <https://github.com/HeiOrient/HeiOrient> [28]  
archived at `swh:1:dir:be7317d125554a54dfd1c9d17521c500c4e1ebc3`

**Funding** We acknowledge support by DFG grant SCHU 2567/3-1. The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

## 1 Introduction

Graphs and networks play a vital role in our connected society for modelling and understanding complex problems. A graph consists of a set of vertices connected by edges, which may be directed or undirected, depending on the specific modeling requirements. For some applications, such as stabilizing telecommunication networks [31], it is necessary to orient each edge of an undirected graph, thereby converting it into a directed graph. In telecommunication networks a lower number of outgoing edges equates to a higher fault tolerance, as not too many connections would be affected by a fault in one of the connection hubs modelled as vertex. One frequently used quality metric for an orientation is the maximum out-degree of a vertex. Given an undirected graph  $G$ , the *edge orientation problem* asks for an orientation of  $G$  in which the maximum out-degree of a vertex is minimized.

The edge orientation problem has a wide range of applications [19]. Besides stabilizing telecommunication networks, other applications include storing optimal graphs [1] or analysis of structural rigidity [32]. In map labeling [23] dense areas of maps can only have one label. In order to formalize this concept, one related task is to identify the densest sub-



© Henrik Reinstädler, Christian Schulz, and Bora Uçar;  
licensed under Creative Commons License CC-BY 4.0

32nd Annual European Symposium on Algorithms (ESA 2024).

Editors: Timothy Chan, Johannes Fischer, John Iacono, and Grzegorz Herman; Article No. 97; pp. 97:1–97:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

graph, which is the set of vertices, that has the highest edge to vertex ratio. This task is called the *max-density* task and has further applications in bioinformatics, web analysis and scheduling [19]. For more applications and their details we refer the reader to Georgakopoulos and Politopoulos [19].

There are different approaches to solve the edge orientation problem exactly. One of the simplest algorithms is by Venkateswaran [31] which solves the problem in  $\mathcal{O}(m^2)$  time. The core idea of this algorithm is to repeatedly find paths with a breadth-first search from a vertex having high out-degree to a vertex having low out-degree. Once such a path is found, the orientation of each of the edges on the path is reversed. Thus by this operation the degree of the high out-degree vertex is reduced and conversely, the degree of the low-degree vertex is increased. Other approaches solve the problem by using a flow based formulation [2, 24]. Kowalik [24] gives an approximation scheme for this problem. These approaches analyse the theoretical aspects, without in depth practical study. The current best complexity bounds and the only implementation in Java of an exact version of Kowalik’s flow based solution are presented by Blumenstock [6]. However, an ongoing challenge in edge orientation algorithms is their scalability, particularly in handling large-scale networks with millions or billions of edges efficiently. As real-world networks continue to expand in size and complexity, there is a need for algorithms capable of scaling effectively to such massive datasets.

**Contribution.** We introduce a novel framework of algorithms inspired by Venkateswaran work [31] of improving paths to tackle the edge orientation problem. Our experimental results show the advantages of this approach over conventional flow-based formulations. Additionally, we offer an alternative proof of correctness for this framework. In addition to a verbatim implementation of the original algorithm, we provide an accelerated version in which we engineer all components of the algorithm. This includes efficient pruning of the search space, batch search of improving paths and better initialization routines that are used as starting point of the algorithm. Experiments show that our algorithm can scale to huge instances and that the performance improvements of our algorithm over the state-of-the-art are very large. For example our fastest algorithm is on average 6.59 times faster than our (faster) C++ reimplementation of the state-of-the-art solver.

This paper is organized as follows: We first introduce basic concepts and provide an extensive review of related work in Section 2 and 3. Section 4 provides an alternative proof of correctness using a flow-based formulation on a bipartite representation of the graph for the algorithm by Venkateswaran. We then proceed to discuss several speed-up techniques, including eager depth-first search, and explore various variations of breadth-first and depth-first search in Section 4. These new approaches are extensively benchmarked in Section 5, followed by our conclusions in Section 6.

## 2 Preliminaries

**Orientation Problem.** An undirected graph  $G = (V, E)$  consists of a vertex set  $V$  containing  $n$  vertices and an edge set  $E \subseteq \binom{V}{2}$  of  $m$  edges. An orientation  $O$  assigns a direction to each edge in  $G$  and results in a directed graph. We denote the direction of an edge by  $O(e) = u \rightarrow v$ , if  $e$  is oriented from  $u$  to  $v$ . For an edge  $e = \{u, v\}$  oriented as  $u \rightarrow v$ , we say that  $e$  is an outgoing edge of  $u$  and incoming edge of  $v$ . The out-degree  $d(O, v)$  of a vertex  $v$  in an orientation  $O$  is defined as the number of outgoing edges of  $v$ , that is,  $d(O, v) = |\{u : v \rightarrow u \in O\}|$ . If the orientation is clear from the context, we write  $d(v)$ . Similarly, the in-degree of a vertex is the number of its incoming edges. We call the vertices

with the largest out-degree in an orientation *peak vertices*. A path in an orientation is a finite sequence of vertices  $u_1, \dots, u_k$ , where there is an edge oriented  $u_i \rightarrow u_{i+1}$ . We call a path simple if no vertex is contained twice in the path. In an orientation, changing the orientation of an edge from  $v \rightarrow u$  to  $u \rightarrow v$  is called *flipping*. A path can be flipped by flipping all edges contained in it once. For a given graph  $G$ , the *edge orientation problem* asks for an orientation  $O$  such that  $d^* = \max_{v \in V} d(O, v)$  is minimized.

**(Pseudo-)arboricity.** A forest is a collection of edges, where each vertex is only connected by one path. The edges of an undirected graph can be partitioned into disjoint forests and the minimum number of forests is known as the *arboricity*. A more relaxed version of this problem is to decompose the graph into pseudoforests, where in every connected component there can be at most one cycle. The minimum number of pseudoforests partitioning the edges is called *pseudoarboricity*. The pseudoarboricity is known to be equivalent to the maximum out-degree of an optimal edge orientation [31]. The average density of a graph is defined as the ratio of edges to vertices. A sub-graph contains only edges between a subset of vertices. The maximum average density of any sub-graph is closely related to the pseudoarboricity. As shown by Picard and Queyranne [26] and Venkateswaran [31] the pseudoarboricity is equal the ceiling of the maximum average density of any sub-graph. Picard and Queyranne [26] show the arboricity is equal to either the pseudoarboricity or the pseudoarboricity plus one.

**Integral Flows.** Given a directed graph  $G = (V, E)$ , a source vertex  $s \in V$ , a target vertex  $t \in V$ , and an edge capacity function  $c : E \rightarrow \mathbb{N}$ , a flow is a function  $f : E \rightarrow \mathbb{N}$  that satisfies two conditions: (i) the flow does not exceed the capacity in any edge; (ii) and the inflow in every vertex equals the outflow, except for  $s$  and  $t$ , which have respectively positive out- and inflows. Given a flow, one can define a residual capacity for each edge, which is equal to edge's original capacity minus the current flow. A residual network for a given flow consists of all edges with a positive residual capacity. If the flow on an edge is equal its capacity, the edge is called saturated. When changing the capacity of an edge, an edge can become under- or oversaturated and the flow needs to be updated.

**Bipartite  $b$ -Matching.** In a bipartite graph the vertex set can be partitioned in disjoint sets  $S, T$  such that all edges contain one vertex from  $S$  and  $T$  each. A matching  $M$  is a set of edges, no two of which share a vertex. A maximum matching is a matching with the largest number of edges. For a given positive integer  $b(v)$  for each vertex  $v$ , the  $b$ -matching problem asks for a set  $F$  of edges with the largest cardinality such that the vertex  $v$  is included in at most  $b(v)$  in  $F$ . A  $b$ -matching in a bipartite graph  $(S \cup T, E)$  can be found by modeling it as a flow problem as follows. First, add a source vertex  $s$  and a sink  $t$  to the graph. Then add edges between  $s$  every vertex in  $v$  in  $S$  with capacity  $b(v)$  as well as edges between each vertex  $w$  in  $T$  with capacity  $b(w)$  from  $w$  to  $t$ . The original edges are assigned unit capacity. The saturated edges of an integral max-flow from  $s$  to  $t$  are the edges in an optimal  $b$ -matching.

### 3 Related Work

**Edge Orientation Task and Pseudoarboricity.** There has been a wide variety of approaches to solve the edge orientation task and the identical pseudoarboricity task. One algorithm is by Venkateswaran [31] and underlies the algorithms proposed in this work. Venkateswaran gives an algorithm for computing an extremal orientation to minimize the maximum in-degree.

■ **Algorithm 1** Algorithm by Venkateswaran [31].

---

```

1: procedure VENKATESWARAN( $G = (V, E)$ )
2:    $O \leftarrow$  an arbitrary orientation of  $G$ 
3:    $k \leftarrow \max_{v \in V} d(O, v)$ 
4:    $S \leftarrow \{v \in V \mid d(O, v) = k\}$ 
5:    $T \leftarrow \{v \in V \mid d(O, v) \leq k - 2\}$ 
6:   while BFS finds path  $P=s, \dots, t$  from  $S$  to  $T$  in  $O$  do
7:     Flip  $P$  in  $O$ 
8:     Remove  $s$  from  $S$ 
9:     Remove  $t$  from  $T$  if  $d(O, t) = k - 1$ 
10:    if  $S$  empty then
11:       $k \leftarrow k - 1$ 
12:       $S \leftarrow \{v \in V \mid d(O, v) = k\}$ 
13:       $T \leftarrow \{v \in V \mid d(O, v) \leq k - 2\}$ 
14:    return  $k$ 

```

---

The algorithm can be easily translated to the out-degree setting, as shown in Algorithm 1. After arbitrarily initializing an orientation, the algorithm starts to search for improvements by finding paths between the set of vertices  $S$  with max out-degree  $k$  and the set of vertices  $T$  with out-degree strictly lower than  $k - 1$ . If  $S$  is empty, then  $k$  is reduced by one, and the sets  $S$  and  $T$  are reinitialized. If no path is found, the current  $k$  is returned as optimal. The correctness of this algorithm follows from the density of the sub-graph induced by the vertices visited by the failing BFS starting from  $S$ . These have a degree of at least  $k - 1$  and there is at least one with degree  $k$ , totaling an average density greater than  $k - 1$ , leading to a pseudoarboricity of  $k$  by the sub-graph density argument. It is proven that the running time of this algorithm is  $\mathcal{O}(m^2)$ , since each path can be found in  $\mathcal{O}(m)$  and the number of improvements can be bounded by  $m$  as well. The argument is that the total number of paths for one vertex is bounded by its out-degree, which again is bounded by the number of edges in total.

A 2-approximation of the pseudoarboricity or the maximum average density can be found in linear time [10, 19] by repeatedly deleting min degree vertices. Georgakopoulos and Politopoulos [19] provide an algorithm for finding the densest subset in a more general setting of set systems, improving on ideas by Goldberg [20] by pruning the graph during their binary search scheme. The algorithm can be used to compute the max out-degree, but does not find a suitable orientation.

Asahiro et al. [2] give a solution with running time  $\mathcal{O}(m^{3/2} \log d^*)$  based on flows for the problem and offer more results for approximating the related weighted edge orientation problem. Kowalik [24] gives an approximation scheme using flows that can be used to calculate exact solutions as well. They construct a virtual graph with a source and a sink vertex. Given an orientation  $O$  and a test value  $d'$  they add an edge between the source vertex and vertices with greater than  $d'$  out-degree with capacity  $d(O, v) - d'$ . Each vertex with lower than  $d'$  out-degree is connected to the sink with capacity  $d' - d(O, v)$ . For each oriented edge  $u \rightarrow v$ , there is an edge  $v \rightarrow u$  with capacity 1. The flow is then computed and all edges with a corresponding saturated edge are flipped. If there is an edge from the source without flow, the test failed and the maximum out-degree must be higher. With a binary search the optimal value  $d^*$  and a suitable orientation can be found in  $\log d^*$  steps.

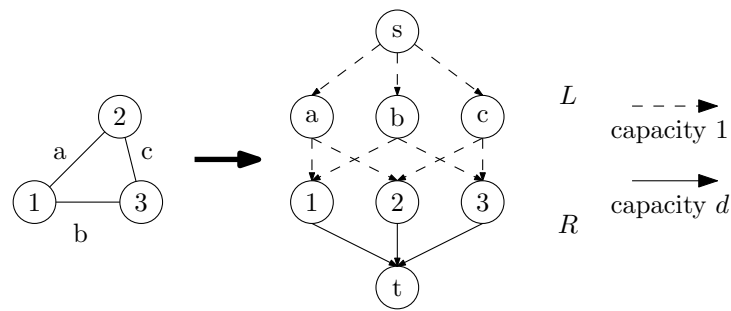
Blumenstock [6] presents bounds for the aforementioned flow-based solution using Dinic’s algorithm and almost unit capacity networks. The best general worst case bound for the problem is  $\mathcal{O}(m^{3/2}\sqrt{\log \log d^*})$ . Moreover, Blumenstock [6] also presents the first practical evaluation of algorithms for the pseudoarboricity problem by implementing the flow based algorithms by Kowalik [24] and Georgakopoulos and Politopoulos [19]. Experiments shows that Georgakopoulos and Politopoulos’s method using unit capacity networks is faster than other flow based solutions, including Kowalik’s algorithm. However, subsequent analysis revealed a subtle implementation error in the implementation of the algorithm which caused it to output incorrect results in rare cases. Once corrected, the performance advantage of this method is no longer apparent. We use both of these algorithms as the state-of-the-art in our experiments.

**Bipartite Matching and Network Flows.** Finding a  $b$ -matching can be done by transforming the problem into an uncapacitated matching problem as described by Gabow [18]: vertices (and their respective edges) are replicated according to their capacity. Bipartite matching is a well-studied and understood part of computer science. The best known worst-case algorithm for solving bipartite matching is Hopcroft-Karp [22] with a complexity of  $\mathcal{O}(m \cdot \sqrt{n})$ . It can be seen as a specialization of Dinic’s algorithm for general flows. For graphs with capacity 1 per vertex, like in the bipartite matching case, the complexity of Dinic’s algorithm is  $\mathcal{O}(n^{1/2}m)$  according to Tarjan and Even [17] matching the bound from Hopcroft-Karp. For general flows, Dinic’s algorithm has a complexity of  $\mathcal{O}(n^2m)$  and better alternatives like Goldberg and Tarjans algorithm [21] are available in practice.

#### 4 Edge Orientation Framework and Engineering Techniques

We give an interpretation of the working of Algorithm 1 using maximum flows and  $b$ -matchings in bipartite graphs. This interpretation allows us to identify key aspects for engineering to improve the practical running time of the algorithm. For a given graph  $G = (V, E)$ , we construct a bipartite graph  $B_G = (L \cup R, E_B)$  with  $|E| + |V|$  vertices, and  $2|E|$  edges. Roughly speaking,  $L$  will represent the edges of the original graph and  $R$  will represent the vertices of the original graph. For each edge  $e = \{u, v\} \in E$ , we have a vertex  $\ell(e) \in L$ . For each vertex  $v \in V$ , we have a vertex  $r(v) \in R$ . For each  $e = \{u, v\} \in E$ , we have two edges in  $E_B$ : one between  $\ell(e)$  and  $r(u)$  and another between  $\ell(e)$  and  $r(v)$ . In this bipartite graph, an edge orientation can be described by an assignment of each  $L$  vertex to one of its two neighbors in  $R$ . Here, each  $L$ -vertex is assigned to a unique  $R$ -vertex, and each  $R$ -vertex can have multiple  $R$ -vertices assigned to it. The edge orientation problem can thus be solved by finding an assignment of  $L$ -vertices to  $R$  vertices which minimizes the maximum number of  $L$ -vertices assigned to an  $R$ -vertex. This can be solved by a network flow formulation.

Our network flow formulation starts with  $B_G$  and adds two special vertices  $s$  and  $t$  to  $B_G$ . The vertex  $s$  is connected to all  $L$ -vertices, and all  $R$ -vertices are connected to the vertex  $t$ . Then, a capacity of 1 is attributed to all edges between  $s$  and  $L$ -vertices, and also to all edges of  $B_G$ ; that is from a vertex  $\ell(e)$  to the vertices  $r(u)$  and  $r(v)$  for  $e = \{u, v\} \in E$ . If we attribute a capacity of  $d$  to edges  $(r, t)$  and have a feasible flow with a total flow value of  $|E|$  (out of  $s$ ), then all edges can be oriented while having an out-degree of at most  $d$ . For an exemplary simple graph  $G$  of three vertices and three edges, the flow graph obtained by adding  $s$  and  $t$  to  $B_G$  can be found in Figure 1.



■ **Figure 1** Example bipartite repr. transformation.

Due to the capacity of 1 for each  $(s, \ell)$  edge, every edge in the original graph will be oriented in only one direction. For an edge  $e = \{u, v\}$  in the original graph, the edge is oriented from  $u \rightarrow v$ , if in  $B_G$  there is a flow from  $\ell(e)$  to  $r(u)$  and  $v \rightarrow u$ , if there is a flow from  $\ell(e)$  to  $r(v)$ . Each edge is assigned a direction, since we search for a flow in which all  $|E|$  edges in the flow graph of the form  $(s, \ell)$  with  $\ell \in L$  are saturated. Clearly, the smallest value of  $d$  allowing this will be the optimal  $d^*$ . When a smaller value is tested and the outflow in  $s$  is not equal  $|E|$ , some edges are not assigned a direction. Since this is a classical capacitated network flow problem with integer capacities, an integer solution can be found with augmenting path-based algorithms [12, Theorem 26.10]. A binary search for the optimal  $d$  will have in the worst case  $\log d_G$  steps, where  $d_G$  is the maximum degree of a vertex in  $G$ . Instead of running a flow algorithm from scratch, the results of previous searches can be reused. When increasing the value of  $d$ , the previous flow can be used as a starting point. If we decrease the test value, the flow in oversaturated edges starting in  $R$  must be balanced first by reducing the flow along a path starting in  $s$  to  $t$ . Afterwards any augmenting path needs to be found.

**Correspondence to Venkateswaran’s Algorithm.** While our interpretation leads to a more sophisticated framework than Venkateswaran’s original description, there is a faithful correspondence. We explain this correspondence, which also establishes another correctness proof for Venkateswaran’s algorithm. Given an arbitrary orientation for the out-degree, we can construct the flow in the bipartite representation as follows: For each edge  $e = \{u, v\}$ , oriented  $u \rightarrow v$  we set the flow from  $\ell(e)$  to  $r(u)$  to 1 and  $\ell(e)$  to  $r(v)$  to 0. In the residual network there is a residual capacity of 1 between  $r(v)$  and  $\ell(e)$ . The obtained flow network is saturated in all edges  $(s, \ell), \ell \in L$ , since each edge is assigned a direction and therefore has a flow from  $\ell(e)$  to either  $u$  or  $v$ . The initial test value for the max out-degree is set to the maximum out-degree of the orientation and repeatedly decreased, until some edges from  $s$  to  $L$  become unsaturated.

In each decreasing step there is an oversaturated edge  $(r(v), t)$ , that needs to be decreased by one, at most for every  $v$  and along to a path to  $s$  the flow needs to be reduced by one. This path has length 2, since one arbitrary edge  $(\ell(e), r(v))$  with flow greater zero will be chosen and its flow as well as the flow on the single incoming edge  $(s, \ell(e))$  reduced. Afterwards an augmenting path needs to be found. This path must include  $\ell(e)$  as second vertex since all other edges  $(s, L)$  are saturated. Instead of first reducing and then finding an augmenting path, we can combine both steps. In other words, the flow algorithm needs to find distinct paths in the residual network between all vertices in  $R$  with oversaturated edges to  $t$  and vertices in  $R$ , that have edges towards edges with enough residual capacity. This can be



■ **Algorithm 2** Orientation by Exhaustive Search.

---

```

1: procedure EXHAUSTIVESHARCH( $G = (V, E), O$ )
2:   repeat
3:     for  $v \in V$  with  $d(v, O) = \max_{w \in V} d(O, w)$  do
4:       if  $P = \text{FINDPATH}(G, O, v)$  exists then
5:         Flip  $P$ 
6:       until no path has been flipped in last iteration
7:   return  $O$ 

```

---

done using breadth-first search algorithms. The vertices in  $R$  with oversaturated edges are exactly the vertices in  $S$ , since we decreased the test value by one. Similarly, the last vertices before  $t$  in augmenting paths need to be vertices  $r(v) \in R$  with residual capacity in  $(r(v), t)$  and enough unsaturated incoming edges. These vertices are exactly those vertices  $T$  in Venkateswaran algorithm. Due to the bipartite structure paths must include the vertices in  $L$ . The edge vertices  $L$  in a residual network of  $B_G$  can be only traversed in one direction by the BFS, exactly like the BFS in Venkateswaran. The only difference, compared to Venkateswaran, is that in our network, the path is twice as long, since each edge in  $G$  consists out of two subedges in  $B_G$ . When in our model a too low test value is set, augmenting paths cease to exist.

#### 4.1 The Proposed Framework

We now present our techniques and framework to improve the performance of Venkateswaran's algorithm in practice. We introduce data structures and algorithmic choices inspired by the maximum cardinality matching problems [15, 16, 22, 27]. Engineering techniques include a simple orientation improving algorithm with complexity  $\mathcal{O}(m)$  that yields a much better starting point for the path finding algorithm in practice. Moreover, we enhance the efficiency of path searches by either reusing information or batching the searches and exploring paths in reverse order.

Our general algorithm framework is shown in Algorithm 2. It is an adapted version of the algorithm by Venkateswaran. Instead of constructing the sets  $S$  and  $T$  explicitly, we iterate over all vertices that currently have the largest out-degree and try to find an improving path, which is a path in the orientation from one of these vertices to a vertex with lower out-degree. If such a path is found, it is flipped, resulting in a lower out-degree for the start vertex. The algorithm continues this process until no such path exists. The correctness of this general algorithm follows from Venkateswaran's algorithm. It has an equivalent termination criterion as the original algorithm, since when it finishes when no further improving path starting at peak vertices can be found. Instead of searching from all peak vertices at the same time, we search for every peak vertex separately. Like the algorithm by Venkateswaran, this algorithm has a time complexity of  $\mathcal{O}(m^2)$  since we need to find at most  $m$  improvements by the argument of Venkateswaran [31]. Furthermore, each improvement can be found with one DFS or BFS in  $\mathcal{O}(m)$ , yielding its total complexity.

#### 4.2 Engineering Techniques

We now discuss how we can improve the techniques to compute orientations. First, we discuss the linear 2-approximation data reduction and come up with a fast initialization algorithm. Secondly, we review different ways of finding improving paths and general ideas to optimize this search.

■ **Algorithm 3** FastImprove Algorithm.

---

```

1: procedure FASTIMPROVE( $G = (V, E), O$ )
2:   for  $v \in V$  do
3:     for  $e = v \rightarrow u \in O$  do
4:       if  $d(O, u) < d(O, v) - 1$  then
5:         Flip( $e$ )
6:     for  $e = u \rightarrow v \in O$  do
7:       if  $d(O, u) - 1 > d(O, v)$  then
8:         Flip( $e$ )
9:   return  $O$ 

```

---

### 4.2.1 Two Approximation Data Reduction

The *linear* 2-approximation proposed by Charikar [10] can be used as data reduction. It computes a 2-approximation  $d_{approx}$  using a bucket priority queue and vertices with degree  $\frac{d_{approx}}{2}$  or less can be removed safely due to the densest sub-graph argument. In other words, since the  $d_{approx}$  is a 2-approximation, the optimum out-degree must be at least  $\frac{d_{approx}}{2}$ . Vertices with a degree that is smaller or equal to this value can therefore be removed iteratively. The edges of these vertices are oriented outwards. The approximation needs to process every edge of the graph at least once and maintaining the needed priority queue is not negligible. Therefore, running it always as a preprocessing step can be costly, if the resulting guess is small and does not remove much of the graph. Instead, we propose to run the 2-approximation conditionally on the density of a graph  $\rho = \frac{m}{n}$ . If the density is smaller than some parameter  $c$ , we directly skip to find improving paths, instead of wasting time to compute a low approximation. In experiments we tune this parameter.

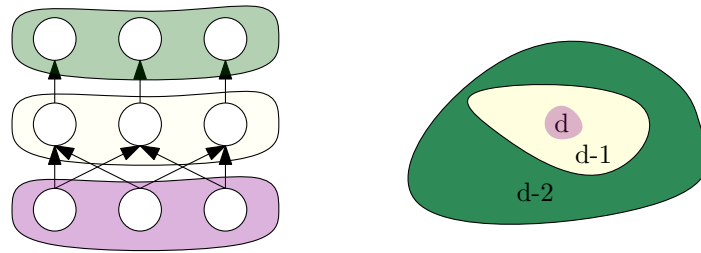
### 4.2.2 Fast Initialization

Venkateswaran's algorithm starts with an arbitrary orientation and thus has worst-case complexity  $O(m^2)$ . However, it is evident that starting with a good orientation can lead to a faster algorithm compared to an arbitrary initialization. The FASTIMPROVE-Algorithm in Algorithm 3 makes one pass over an arbitrary orientation and flips an edge  $v \rightarrow u$ , if such a flipping improves the largest out-degree of  $v$  and  $u$ . Since only direct improvements are found, the algorithm does not find the optimal solution. However, it can improve a given orientation significantly. In our implementation, vertices are represented by their IDs which are numbers in the range  $0, \dots, n - 1$ . Our algorithm initializes an orientation with  $u \rightarrow v$  for  $u > v$  and then applies the *FastImprove* algorithm to quickly improve it. The complexity of one pass is  $O(m)$ . In the experimental evaluation in Section 5, we show that this simple algorithm can tremendously speed up computation time of the overall algorithm.

### 4.2.3 Path Finding Algorithms

There are two basic ways to find paths in a directed graph: either breadth- or depth-first search. Breadth-first search works by using a queue and adding unvisited neighboring vertices following first in first out order. Depth-first search is usually implemented using a stack and a visited array. The visited array stores whether a vertex has been visited before, while the stack keeps track of the current path. The next not visited neighboring vertex is always added to the stack and marked as visited, the neighbors are explored recursively using the stack, i. e., depth first. Once all neighbors are explored the previous vertex of the stack is (re-)visited and its remaining neighbors are visited.





■ **Figure 2**

**Left:** Layered example orientation in a graph with nine edges and vertices.

**Right:** Visualization of the onion-like structure of the orientation problem.

**Batched BFS.** The batched BFS approach finds improving paths similar to the original idea by Venkateswaran of an adapted BFS. It does not fit the previously described framework in Algorithm 2. Instead of starting a BFS from every vertex, we start the BFS by putting all peak vertices in the queue and flip a path once we have found an improving path by breadth first search. We continue the search until all vertices have been reached or, in case of success, we found an improvement for all peak vertices. The search is continued until for none of the vertices we find an improving path. This yields a correct result, since we inserted all peak vertices at the start and thus will find a shortest improving path, if there exists one. However, we will not always find a path for all peak vertices at once.

**DFS.** We now look at ways to improve depth-first-search. Our ideas include checking neighboring vertices eagerly, ensuring independent paths by visiting peak vertices only as starting vertices of paths, reusing the visited array and eagerly ordering path searches.

*Early Check.* The first improvement is to check the out-degree of all neighbors before continuing exploring them in the recursion. This increases the overall complexity by a constant factor, but in practice speeds up the exploration massively by preventing unneeded recursion.

*DFS with Independent Paths.* We propose for the DFS to not continue traversal via other peak vertices while searching for improving paths. There is no added benefit in adding a peak vertex to a path during search as each peak vertex needs its own improving path. More precisely, they have to be distinct, since flipping one of the paths would change the orientation of the joint part of the two paths, effectively stopping the second path from being valid. Hence, we need to find independent paths from both of the vertices regardless.

*DFS with Shared Visited Array.* A DFS can reuse information stored in the visited array for multiple consecutive searches for improving paths. Here, we reset the visited array used in a classical DFS only after one pass over all peak vertices and not after every path search. Thus, we enhance efficiency by retaining the visited array's state across multiple consecutive searches to identify improving paths, rather than resetting it after each search. This method strategically excludes previously explored and non-improving sub-graphs from subsequent searches, significantly reducing computational redundancy. By maintaining the visited array across searches, we ensure that once a path from a peak vertex is improved, subsequent searches do not redundantly explore the same paths or sub-graphs already deemed non-improving. Figure 2 (Left) gives a simple example orientation, where not resetting the visited array is helpful: If the first path from the bottom left to the top left vertex is found and flipped, we do not want the search starting in the middle vertex to traverse via left tree that was already explored during the first DFS. Moreover, this

example orientation shows, why this DFS approach is better than the BFS described in the earlier section. The BFS would find at most two improving paths, because it eagerly finds predecessors. It would assign the middle vertices (yellow) only two distinct predecessors at most due to the graph structure, leaving one peak vertex (purple) without successors and leading to two paths for the bottom left vertex. However, we are only interested in one path per peak vertex. Thus, this leads to more path searches than necessary.

*Eager Path Search.* The *FastImprove* algorithm only uses direct improvements and reduces their number, leaving longer improving paths to be discovered. Therefore, our problem has an onion-like structure, multiple layers of vertices with the same out-degree are surrounding some core peak vertices after the initialization. By our classical approach we would find improvements from core to the lower degree outer layers and slowly growing the number of peak vertices. This leads to prolonged paths, as there will be multiple paths needed to be found while decreasing the out-degree of a peak vertex by one at a time. An example for this is shown in Figure 2 (Right), the vertices with out-degree  $d$  will need to be improved multiple times via the vertices with out-degree  $d - 1$ . Instead of slowly searching consecutively, we propose to first find paths for the outer layers of the problem. We define  $i$  as the number of layers we reduce eagerly in the reverse order of their out-degree, i. e.,  $d - i$ ,  $d - i + 1$ , and so on. Moreover, if only fewer vertices than some parameter  $c$  have an out-degree of  $d - i + k$  in the orientation, we propose to repeatedly search up to  $k$  improvements with a DFS. Intuitively, this reduces the number of times the algorithm needs to collect these vertices and find paths.

As both ideas are combined with the idea of the reused visited array, this method does not return an exact solution always: Suppose  $d$  is the current max out-degree, then it can occur, that there is no improvement for  $d - 1$ , but there would exist some path from  $d$  out-degree vertices to  $d - 2$  out-degree vertices through  $d - 1$  vertices. Since the  $d - 1$  vertices are checked first, there is no improving path found and the visited information is kept, leading to all  $d - 1$  vertices marked as visited. The searches started in  $d$  cannot traverse over these vertices and the search is unsuccessful. Therefore, it requires to run either a DFS or a BFS afterwards. We propose to decide based on the maximum size of the layers and the resulting max out-degree whether to run a batched BFS or DFS.

Moreover, the choice of  $i$  and  $c$  is crucial, in Section 5 we test static values for  $c$  and  $i$ . Additionally, we test for  $i$  a dynamic value of  $\sqrt{\max d - \rho}$  with  $\rho = \frac{m}{n}$  as average density of the graph and  $\max d$  being the out-degree of the starting orientation. The maximum out-degree of the starting orientation  $\max d$  and  $\rho$  are the natural lower and upper bounds for this problem. We chose the square root as a natural damping function in order to not explore too many layers eagerly.

## 5 Experimental Evaluation

**Methodology.** We implemented our algorithms and data structures in C++ 20. We compiled our program and all competitors using g++-12.1 with full optimization turned on (-O3 flag). In our experiments, we use two types of machines provided by a cluster for our experiments. Both machines are equipped with 20-core Intel Xeon Gold 6230 processors running at 2.10 GHz and having a cache of 27.5 MB. Machine type A has two sockets and 96 GB of RAM, machine type B has four sockets with 3 TB of RAM. For the five graphs with more than 1 billion edges we use the machine type B. We run each algorithm on each instance five times and use the arithmetic mean of the running time of these independent runs in the experiments. Up to four experiments were run in parallel on machine type A and 6 on

machine type B. The order of the experiments was random. The experiments that did not finish within 5 hours were only repeated once. The running times reported include the setup of data structure needed by the algorithms, but does not include the initial loading of the graphs from the hard drive. For comparisons, we use performance profiles as proposed by Dolan and Moré [14]. We plot which fraction of instances is solved by an algorithm within  $\tau t_{opt}$ , for  $\tau \geq 1$  and  $t_{opt}$  being the best running time reported by any algorithm on a given instance. A higher fraction at a lower  $\tau$  means that more instances are solved within this  $\tau$ , implicating a better performance.

**Instances.** We use graphs from the SuiteSparse Sparse Matrix Collection [13] with more than 1 million vertices for benchmarking. The set contains 67 graphs from a wide range of applications. Statistics on the instances, including the number of edges, the number of vertices, the minimum and the maximum degree of a vertex, and the number of connected components, can be found in the full version of the paper [29]. The largest graph in our set has more than five billion edges. Based on the average density of the graphs, we chose nine representative instances for evaluating different parameters for the proposed algorithms and running preliminary experiments faster. These representative instances are called **test set** in the following and are not included in the geometric means and performance profiles in the final experiment.

**State-of-the-art.** Blumenstock [6] provided us with a Java code implementing Kowalik’s exact algorithm. We ported that implementation with Dinic’s algorithm to C++ and validated that our implementation is on average 1.42 times faster than the Java implementation (compiled with OpenJDK 17 and run sequentially) on the instances used by Blumenstock [6]. A more detailed comparison of the running time of these implementations is reported in the full version of the paper [29]. We have also implemented Georgakopoulos and Politopoulos’s approach to be able to do a more conclusive comparison than what was available in [6]. We can run this algorithm with Dinic’s algorithm or Push-Relabel algorithm for better performance. We use a Push-Relabel implementation in our final experiments for comparison since it is 1.6 times faster than Dinic’s algorithm on our implementation. The algorithm by Georgakopoulos and Politopoulos computes only the pseudoarboricity, i.e., the objective value of an orientation, and not an orientation itself. As described by Blumenstock [6] our implementation computes the pseudoarboricity and uses Kowalik’s reorientation scheme once to obtain an orientation. In the following, we refer to our implementation of Georgakopoulos and Politopoulos [19] by *GEP* and to our implementation of the exact Kowalik [24] by *Kowalik*. Preliminary experiments on the test dataset showed that using the 2-approximation initialization before *Kowalik* and *GEP* results in, respectively, 1.90 and 3.02 times faster running time than not using this initialization. Therefore, we run the final experiment with 2-approximation for both of the state-of-the-art methods. We note that our experiments with *GEP* and *Kowalik* are more comprehensive and conclusive than what were available in earlier work [6].

**Implemented Algorithms.** For our main approaches we implemented a vanilla DFS (*DfsNaive*) and the algorithms devised in Section 4.2. These include Fast Improvement (*FastImprove*), DFS with improvements (*Dfs*), BFS (*Bfs*) and the eager path search (*EPS*). Our implementation of 2-approximation (*TwoA*) and the 2-approximation conditioned on the average density (*TwoADens*) can be run before all algorithms considered here.

## 5.1 Parameter Study for the Proposed Algorithms

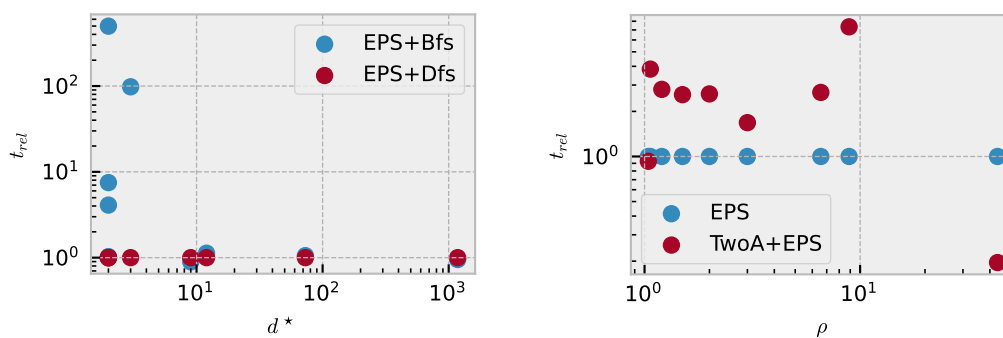
In the subsequent sections, we adjust our parameters and investigate the different algorithmic components described in this paper using the nine graphs selected specifically for parameter tuning.

**Naive Methods.** We conducted a comparative analysis of several DFS implementations: one without any of the techniques outlined in Section 4.2.3 (*DfsNaiveWithoutShared*), one that incorporates all the proposed techniques (*Dfs*), another that reuses the visited array (*DfsNaiveShared*), and the batched BFS (*Bfs*). All these implementations utilized the *FastImprove* approach, and we present only the average execution times. Our findings indicate that the *Dfs* method is the most efficient, being significantly faster than the others. Specifically, the *DfsNaiveShared* is 2.02 times slower, and the *Bfs* is notably slower by a factor of 10.38. The slowest was the *DfsNaiveWithoutShared*, which was 206.443 times slower than the optimized *Dfs*. Based on these results, we will exclusively use the optimized *Dfs*, i.e. DFS with improvements, for the subsequent parameter tuning experiments.

**Fast Improve.** To demonstrate the effectiveness of the *FastImprove* technique, we conducted experiments on the small dataset, running *Dfs* both with and without the technique. The variant lacking *FastImprove* failed to complete within 5 hours on the instance from the MAWI set, which notably has an extremely high maximum degree of approximately 63M. In stark contrast, the *FastImprove*-enabled variant completed the same instance in just 18.84 seconds. Across the remaining eight instances, the non-*FastImprove* variant performed consistently slower, averaging 27% longer processing times. Given that omitting *FastImprove* led to a timeout on one instance and consistently slower performance on others, we use the *FastImprove* algorithm for all instances.

**Eager Path Search.** We now optimize the parameters for our eager path search. In the eager path search we search for improving paths not only for the peak vertices, but for layers with lower out-degree. The parameters are the number of layers eagerly searched and the maximum size of a layer that will be searched for even more eagerly. Furthermore, we probe which method (*Bfs* or *Dfs*) should be used to finalize the orientation.

We tested the number of layers eagerly searched for values  $i \in \{2, 5, 10, \sqrt{\max d - \rho}\}$  with  $\rho = \frac{m}{n}$  and  $\max d$  being the current maximum out-degree of the starting orientation. We used the optimized *Dfs* to compute the final orientation. For the sampled set the dynamic approach  $i = \sqrt{\max d - \rho}$  yields the best results on average. The static approach with  $i = 5$  is only 2 % slower, while the approaches with  $i = 2$  and 10 are 7.3 resp. 8.3 % slower. We select the dynamic approach for our final experiment and now progress into the second parameter the eager size. If a layer size is smaller than the eager size the algorithm tries to find multiple improvements consecutively, instead of requiring the vertices to be collected again. In this experiment we combined the dynamic approach and DFS with the eager size parameter  $c \in \{10, 100, 1000, 10000\}$ . The best geometric mean we can report for  $c = 100$ , for  $c = 10, 1000, 10000$  we report a 1.5%, 0.6% resp. 0.16% higher geometric mean for the running time. Therefore, we select  $c = 100$  for our final experiment. Finally, we investigate when to run either the *Bfs* or *Dfs* after the eager path search. Figure 3a shows the running time divided by the optimal value plotted against the min max out-degree. We observe that for lower out-degree ( $< 10$ ) it is more beneficial to run the optimized *Dfs*. On one instance with low out-degree using *Bfs* is 60 times slower. However, on instances with higher out-degree the *Bfs* is consistently faster by a small relative margin. We run *Dfs*, if the out-degree is less than 10, and otherwise *Bfs*.



(a) The running time of *EPS+Dfs* and *EPS+Bfs* normalized by *EPS+Dfs* plotted against the resulting min max out-degree. Low out-degree solutions are solved faster by *Dfs* as finishing method. (b) The running time of *EPS* with or without 2-approximation plotted against the average density  $\rho = \frac{m}{n}$  of the graph (normalized by *EPS*).

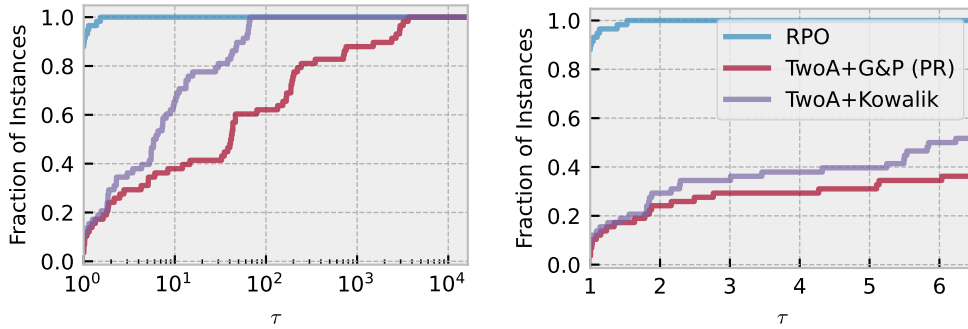
### Figure 3

**Two Approximation.** We now look into the efficiency of the 2-approximation as outlined in Section 4.2.1. Specifically, we investigate the effectiveness of not always executing the two-approximation prior to the eager path search. The outcomes of this study are illustrated in Figure 3b, where we plot the average density of the graphs against their average running time. The experiment indicates that for graphs with low average density, pre-running the two-approximation does not yield beneficial results. We run the 2-approximation exclusively for graphs with an average density exceeding 10 to achieve reasonable reductions on such graphs.

**Final Algorithm.** The final configuration of our algorithm *RapidPathOrientation* (RPO) is as follows: After using the 2-approximation conditionally on the average density ( $\frac{m}{n} > 10$ ) and executing the *FastImprove* algorithm, it runs the *EPS* with dynamic layer count of  $\sqrt{(\max d - \rho)}$ . If less than 100 vertices are in a layer they are searched for more eagerly. Finally, in order to produce a correct result a *Bfs* is run for out-degree higher than 10 and the specialized *Dfs* for lower degree.

## 5.2 Comparison with State-of-the-Art

We now compare our approach to the state-of-the-art on the whole dataset. In Figure 4 the performance profile for our algorithm (*RapidPathOrientation*) is shown in comparison to the algorithm by Kowalik [24] (*TwoA+Kowalik*) and Georgakopoulos & Politopoulos [19] (*TwoA+GEP*) with combined with the 2-approximation on 58 problem instances. On average our algorithm is 6.59 times faster than *TwoA+Kowalik* and 36.27 times faster than *TwoA+GEP*; moreover on 88 % of instances it is the fastest approach. As seen in the right plot of Figure 4, *RapidPathOrientation* solves all instances within a factor of  $\tau < 1.56$  of the fastest algorithm (on an instance) and the competitor approaches solve around 50 % of instances within a factor of  $\tau = 6$ . Since the profile of *RapidPathOrientation* is higher than both *TwoA+Kowalik* and *TwoA+GEP* for all values of  $\tau$ , we conclude that the proposed approach is faster than the reimplementations of the state-of-the-art methods. Moreover, as shown in the preliminary experiment, the speed up over the Java implementations is even more pronounced. We report detailed per instance results of the running times in Table 2. Table 1 shows the running times averaged per sub dataset. We now give a more detailed discussion of the performance of the different algorithms for the subgroups of our dataset.



■ **Figure 4** Running time performance profile on 58 instances for state-of-the-art algorithms *TwoA+Kowalik*, *TwoA+G&P* and our *RapidPathOrientation* (*RPO*) approach. On the right hand side we present a detailed zoom to values up to 6.5.

In general, it can be observed that competing algorithms can solve instances from the MAWI subset faster. Out of all the instances for which our approach is not the fastest, our algorithm experiences the highest relative slowdown on the `mawi_130` instance. Here, *RapidPathOrientation* needs 56% more time than both competing algorithms. This is due to the fact that the 2-approximation is very effective on the MAWI dataset, however the 2-approximation is not run by our algorithm on these instances as they have a low average density ( $\frac{m}{n} < 2$ ). When running our approach on these instances with the 2-approximation algorithm as a preprocessing algorithm, we achieve similar running times.

The performance differences are very big especially on the low density and low out-degree instances of the flowipm22 set. For example, on the `spielman_k600` instance *TwoA+G&P* is 243.10 times and *TwoA+Kowalik* is 65.46 times slower than our new approach. In general, this is due to the fact that the FlowIPM22 instances have a high maximum degree compared to their optimum out-degree of 3. The competitors cannot limit their search space properly and have to perform many reorientations using their flow algorithm.

Similarly, we can observe advantages of our approach on random generated graphs in the dimacs10 set, like the `de1aunay_n24` instance. *RapidPathOrientation* is around 80 times faster than *TwoA+Kowalik* and 221 times faster than *TwoA+G&P*. Moreover, the *TwoA+G&P* algorithm has issues with some instances from numerical backgrounds (`huge*`) in the dimacs10 instances, resulting in a running time much higher than the other approaches (up to 16393.74 times slower). The *TwoA+G&P* algorithm has to solve more flows to converge according to its binary search scheme, while its pruning mechanism is not that efficient on these instances. The pruning removes vertices with unsaturated edges connected to the target vertex, when a new lower bound for the pseudoarboricity is accepted, which does not happen often on those instances. In general, our new algorithm *RapidPathOrientation* solves instances from numerical backgrounds and road networks, represented in the dimacs10 set, significantly faster than the previous approaches.

On the first instance from the gap dataset, `gap-urand`, *TwoA+G&P* requires 81481.60 seconds to finish, 15.71-fold of what our new approach and 6.31-fold of what *TwoA+Kowalik* require to finish. On the `gap-kron` instance, both competitors are 1.38 (*TwoA+G&P*) resp. 1.34 (*TwoA+Kowalik*) times slower. On all five instances from genbank set, our new approach is about 1.9 times faster than each of the competitors. There are no significant deviations in this dataset. On the single instance from the law dataset we report that *RapidPathOrientation* is 1.63 times faster than *TwoA+Kowalik* and 2.76 times faster than

■ **Table 1** Geometric mean running time in seconds grouped by sub data set and for all instances.

	<i>TwoA+GEP</i>	<i>TwoA+Kowalik</i>	RPO
flowipm22 [9]	104.62	33.57	<b>0.57</b>
gap [4]	12 267.57	4808.19	<b>2628.25</b>
genbank [5]	70.14	70.63	<b>36.69</b>
law [8, 7]	5.02	2.97	<b>1.82</b>
mawi [11]	8.40	<b>8.39</b>	10.07
snap [25]	8.86	5.15	<b>1.85</b>
sybrandt [30]	5340.81	3064.19	<b>2226.32</b>
dimacs10 [3]	337.96	21.49	<b>1.86</b>
<b>all</b>	144.85	26.31	<b>3.99</b>

*TwoA+GEP*. The snap dataset contains social graphs and road networks. On the three road networks our approach requires less than 1/10th of the running time in comparison to the fastest competitor (*TwoA+Kowalik*). For two of the social graphs (*com-youtube,as-skitter*) all approaches require nearly exactly the same time. The *com-friendster* instance is solved by *TwoA+GEP* about 1.13 times faster than our approach. On the *com-livejournal* the *TwoA+Kowalik* is faster by a factor of 1.08.

The sybrandt set contains the two biggest instances in our dataset. For the *moliere'16* *TwoA+Kowalik* is the fastest algorithm, our approach is 14% slower. *RapidPathOrientation* solves the *agatha'15* instance 2.16 times faster than the fastest competitor *TwoA+Kowalik*.

## 6 Conclusion

We have proposed a new framework for algorithms solving the edge orientation problem based on the ideas of Venkateswaran [31] and gave a new flow-based proof. We have investigated a vast variety of engineering techniques for the problem. Our techniques include a fast improvement heuristic, specialized depth-first-search, scheduling path searches more eagerly as well as running a data reduction based on a 2-approximation algorithm. Experiments have shown that our final algorithm outperforms the fastest exact state-of-art algorithm by a factor of 6.59 on average. Especially on low density and low out-degree instances, like road networks and instances from numerical backgrounds, our algorithm outperforms its competitors. Only on low density, high out-degree instances the competitors have an advantage and compute an orientation slightly faster.

There are multiple areas of future work. Most importantly, we think that an explicit parallelism for the proposed algorithms is worthwhile as on massive instances we still observe very large running times overall.



## 97:16 Engineering Edge Orientation Algorithms

**Table 2:** Average running time per instance in seconds. Sorted by  $d^*$ . 5 repetitions.

	$d^*$	<i>TwoA+GEP</i>	<i>TwoA+Kowalik</i>	RPO
adaptive	2	5508.26	8.80	<b>1.40</b>
asia_osm	2	56.45	14.28	<b>1.16</b>
belgium_osm	2	7.17	1.72	<b>0.17</b>
europa_osm	2	288.53	88.28	<b>6.65</b>
germany_osm	2	61.71	13.60	<b>1.62</b>
g'b'_osm	2	33.52	7.72	<b>0.96</b>
hugebubbles'00	2	70 001.31	31.00	<b>4.27</b>
hugebubbles'10	2	31 454.70	44.19	<b>8.66</b>
hugebubbles'20	2	27 227.08	46.97	<b>8.49</b>
hugetrace'00	2	2901.07	7.22	<b>0.98</b>
hugetrace'10	2	8391.65	22.87	<b>3.50</b>
hugetrace'20	2	7916.97	31.78	<b>5.27</b>
hugetric'00	2	3568.83	9.04	<b>1.22</b>
hugetric'10	2	4805.79	13.60	<b>2.46</b>
hugetric'20	2	7161.24	14.68	<b>2.67</b>
italy_osm	2	31.45	8.07	<b>0.76</b>
n'l'_osm	2	10.50	2.15	<b>0.22</b>
roadnet-ca	2	11.33	3.44	<b>0.25</b>
roadnet-pa	2	5.71	1.72	<b>0.14</b>
road_central	2	92.38	26.98	<b>4.60</b>
road_usa	2	145.56	34.94	<b>3.56</b>
spielman_k200	2	14.16	5.42	<b>0.08</b>
spielman_k300	2	51.19	16.52	<b>0.38</b>
spielman_k400	2	123.57	39.95	<b>0.64</b>
spielman_k500	2	250.26	79.07	<b>1.24</b>
spielman_k600	2	559.14	150.55	<b>2.30</b>
333sp	3	2408.29	95.47	<b>3.19</b>
as365	3	2820.13	148.65	<b>3.90</b>
delaunay_n20	3	72.27	19.73	<b>0.47</b>
delaunay_n21	3	168.78	48.29	<b>1.01</b>
delaunay_n22	3	413.96	123.88	<b>2.20</b>
delaunay_n23	3	950.79	322.33	<b>4.82</b>
delaunay_n24	3	2275.57	816.72	<b>10.28</b>
m6	3	1415.00	132.64	<b>4.08</b>
naca0015	3	613.64	24.18	<b>0.86</b>
kmer_v1r	3	132.97	133.62	<b>58.33</b>
nlr	3	1500.28	176.09	<b>4.70</b>
roadnet-tx	3	7.13	2.26	<b>0.15</b>
venturilevel3	3	25.09	3.44	<b>0.59</b>
kmer_u1a	5	40.80	40.71	<b>21.67</b>
kmer_v2a	8	32.63	33.10	<b>18.30</b>

	$d^*$	<i>TwoA+GEP</i>	<i>TwoA+Kowalik</i>	RPO
channel-'b050	9	1031.67	10.47	<b>0.66</b>
packing-'b050	9	76.12	4.54	<b>0.41</b>
kmer_a2a	10	110.12	109.49	<b>59.56</b>
kmer_p1a	10	87.13	89.17	<b>48.26</b>
rgg_n_2_20_s0	12	0.88	0.77	<b>0.25</b>
rgg_n_2_21_s0	12	7.86	3.46	<b>0.59</b>
rgg_n_2_22_s0	13	9.89	6.09	<b>1.11</b>
rgg_n_2_23_s0	14	13.85	9.49	<b>2.22</b>
rgg_n_2_24_s0	14	22.65	18.12	<b>5.24</b>
gap-urand	17	81 481.60	12 910.44	<b>5183.51</b>
mawi'345	40	2.26	2.29	<b>2.08</b>
com-youtube	46	0.71	0.58	<b>0.58</b>
mawi'000	58	4.41	<b>4.38</b>	4.48
mawi'030	73	<b>8.16</b>	8.20	9.67
mawi'130	78	<b>14.67</b>	14.67	22.85
as-skitter	90	1.08	<b>1.03</b>	1.04
mawi'330	93	33.95	<b>33.65</b>	48.18
agatha'15	97	22 506.38	9267.07	<b>4291.18</b>
com-livejournal	194	4.26	<b>4.14</b>	4.51
com-orkut	228	27.61	14.00	<b>9.18</b>
molieres'16	232	1267.39	<b>1013.19</b>	1155.05
com-friendster	274	<b>917.42</b>	1056.45	1039.20
kron_g500-'20	908	13.62	5.73	<b>2.65</b>
hollywood-2009	1 104	5.02	2.97	<b>1.82</b>
kron_g500-'21	1 178	21.94	10.85	<b>5.78</b>
gap-kron	2 369	1846.96	1790.70	<b>1332.63</b>

## References

- 1 Oswin Aichholzer, Franz Aurenhammer, and Günter Rote. *Optimal graph orientation with storage applications*. Universität Graz/Technische Universität Graz. SFB F003-Optimierung und Kontrolle, 1995.
- 2 Yuichi Asahiro, Eiji Miyano, Hirotaka Ono, and Kouhei Zenmyo. Graph orientation algorithms to minimize the maximum outdegree. *Int. J. Found. Comput. Sci.*, 18(2):197–215, 2007. doi:10.1142/S0129054107004644.
- 3 David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In Reda Alhajj and Jon G. Rokne, editors, *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*, pages 73–82. Springer, 2018. doi:10.1007/978-1-4939-7131-2\_23.
- 4 Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015. doi:10.48550/arXiv.1508.03619.
- 5 Dennis A. Benson, Mark S. Boguski, David J. Lipman, and James Ostell. Genbank. *Nucleic Acids Research*, 24(1):1–5, 1996. doi:10.1093/nar/24.1.1.
- 6 Markus Blumenstock. Fast algorithms for pseudoarboricity. In Michael T. Goodrich and Michael Mitzenmacher, editors, *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pages 113–126. SIAM, 2016. doi:10.1137/1.9781611974317.10.
- 7 Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proc. of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011. doi:10.1145/1963405.1963488.
- 8 Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the 13th Int. World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press. doi:10.1145/988672.988752.
- 9 Léopold Cambier, Chao Chen, Erik G. Boman, Sivasankaran Rajamanickam, Raymond S. Tuminaro, and Eric Darve. An algebraic sparsified nested dissection algorithm using low-rank approximations. *SIAM Journal on Matrix Analysis and Applications*, 41(2):715–746, 2020. doi:10.1137/19M123806X.
- 10 Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In Klaus Jansen and Samir Khuller, editors, *Approximation Algorithms for Combinatorial Optimization, Third International Workshop, APPROX 2000, Saarbrücken, Germany, September 5-8, 2000, Proceedings*, volume 1913 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2000. doi:10.1007/3-540-44436-X\_10.
- 11 Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. Traffic data repository at the WIDE project. In *Proceedings of the Freenix Track: 2000 USENIX Annual Technical Conference, June 18-23, 2000, San Diego, CA, USA*, pages 263–270. USENIX, 2000. URL: <http://www.usenix.org/publications/library/proceedings/usenix2000/freenix/cho.html>.
- 12 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, Cambridge, MA, 3rd edition, 2009. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
- 13 Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. doi:10.1145/2049662.2049663.
- 14 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002. doi:10.1007/s101070100263.
- 15 Iain S. Duff. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Softw.*, 7(3):315–330, 1981. doi:10.1145/355958.355963.
- 16 Iain S. Duff, Kamer Kaya, and Bora Uçar. Design, implementation, and analysis of maximum transversal algorithms. *ACM Transactions on Mathematical Software*, 38:13:1–13:31, 2011. doi:10.1145/2049673.2049677.

- 17 Shimon Even and Robert Endre Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975. doi:10.1137/0204043.
- 18 Harold N. Gabow. An efficient reduction technique for degree-constrained subgraph and bidirected network flow problems. In David S. Johnson, Ronald Fagin, Michael L. Fredman, David Harel, Richard M. Karp, Nancy A. Lynch, Christos H. Papadimitriou, Ronald L. Rivest, Walter L. Ruzzo, and Joel I. Seiferas, editors, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing, 25-27 April, 1983, Boston, Massachusetts, USA*, pages 448–456. ACM, 1983. doi:10.1145/800061.808776.
- 19 George F. Georgakopoulos and Kostas Politopoulos. MAX-DENSITY revisited: a generalization and a more efficient algorithm. *Comput. J.*, 50(3):348–356, 2007. doi:10.1093/comjnl/bx1082.
- 20 Andrew V Goldberg. Finding a maximum density subgraph, 1984. URL: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/1984/CSD-84-171.pdf>.
- 21 Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988. doi:10.1145/48014.61051.
- 22 John E. Hopcroft and Richard M. Karp. An  $n^{5/2}$  algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973. doi:10.1137/0202019.
- 23 Konstantinos G. Kakoulis and Ioannis G. Tollis. On the multiple label placement problem. In *Proceedings of the 10th Canadian Conference on Computational Geometry, McGill University, Montréal, Québec, Canada, August 10-12, 1998*, 1998. URL: <http://cgm.cs.mcgill.ca/cccg98/proceedings/cccg98-kakoulis-multiple.ps.gz>.
- 24 Łukasz Kowalik. Approximation scheme for lowest outdegree orientation and graph density measures. In Tetsuo Asano, editor, *Algorithms and Computation, 17th International Symposium, ISAAC 2006, Kolkata, India, December 18-20, 2006, Proceedings*, volume 4288 of *Lecture Notes in Computer Science*, pages 557–566. Springer, 2006. doi:10.1007/11940128\_56.
- 25 J. Leskovec. Stanford Network Analysis Package (SNAP). <http://snap.stanford.edu/index.html>, June 2014.
- 26 Jean-Claude Picard and Maurice Queyranne. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks*, 12(2):141–159, 1982. doi:10.1002/net.3230120206.
- 27 Alex Pothén and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, December 1990. doi:10.1145/98267.98287.
- 28 Henrik Reinstädler, Christian Schulz, and Bora Uçar. HeiOrient. Software, DFG-SCHU 2567/3-1, swhId: `swh:1:dir:be7317d125554a54dfd1c9d17521c500c4e1ebc3` (visited on 2024-08-07). URL: <https://github.com/HeiOrient/HeiOrient>.
- 29 Henrik Reinstädler, Christian Schulz, and Bora Uçar. Engineering edge orientation algorithms, 2024. doi:10.48550/arXiv.2404.13997.
- 30 Justin Sybrandt, Ilya Tyagin, Michael Shtutman, and Ilya Safro. AGATHA: automatic graph mining and transformer based hypothesis generation approach. In Mathieu d’Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux, editors, *CIKM ’20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, pages 2757–2764. ACM, 2020. doi:10.1145/3340531.3412684.
- 31 Venkat Venkateswaran. Minimizing maximum indegree. *Discret. Appl. Math.*, 143(1-3):374–378, 2004. doi:10.1016/j.dam.2003.07.007.
- 32 Walter Whiteley. The union of matroids and the rigidity of frameworks. *SIAM Journal on Discrete Mathematics*, 1(2):237–255, 1988. doi:10.1137/0401025.