Proceedings Paper:
Blikstad, J., van den Brand, J., Mukhopadhyay, S. et al. (2021) Breaking the quadratic barrier for matroid intersection. In: Khuller, S., (ed.) STOC 2021: Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing. STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, 21-25 Jun 2021, Virtual, Italy. Association for Computing Machinery (ACM), pp. 421-432. ISBN: 9781450380539. ISSN: 0737-8017.

https://doi.org/10.1145/3406325.3451092

Breaking the Quadratic Barrier for Matroid Intersection

Joakim Blikstad[1], Jan van den Brand[1], Sagnik Mukhopadhyay[1], and Danupon Nanongkai[1]

[1]KTH Royal Institute of Technology, Sweden
{blikstad,janvdb,sagnik,danupon}@kth.se

Abstract

The matroid intersection problem is a fundamental problem that has been extensively studied for half a century. In the classic version of this problem, we are given two matroids $\mathcal{M}_1 = (V, \mathcal{I}_1)$ and $\mathcal{M}_2 = (V, \mathcal{I}_2)$ on a comment ground set V of n elements, and then we have to find the largest common independent set $S \in \mathcal{I}_1 \cap \mathcal{I}_2$ by making *independence oracle queries* of the form "Is $S \in \mathcal{I}_1$?" or "Is $S \in \mathcal{I}_2$?" for $S \subseteq V$. The goal is to minimize the number of queries.

Beating the existing $\tilde{O}(n^2)$ bound, known as the *quadratic barrier*, is an open problem that captures the limits of techniques from two lines of work. The first one is the classic Cunningham's algorithm [SICOMP 1986], whose $\tilde{O}(n^2)$-query implementations were shown by CLS+ [FOCS 2019] and Nguyễn [2019].[1] The other one is the general cutting plane method of Lee, Sidford, and Wong [FOCS 2015]. The only progress towards breaking the quadratic barrier requires either *approximation* algorithms or a more powerful *rank oracle query* [CLS+ FOCS 2019]. No exact algorithm with $o(n^2)$ independence queries was known.

In this work, we break the quadratic barrier with a randomized algorithm guaranteeing $\tilde{O}(n^{9/5})$ independence queries with high probability, and a deterministic algorithm guaranteeing $\tilde{O}(n^{11/6})$ independence queries. Our key insight is simple and fast algorithms to solve a graph reachability problem that arose in the standard augmenting path framework [Edmonds 1968]. Combining this with previous exact and approximation algorithms leads to our results.

[1]More generally, these algorithms take $\tilde{O}(nr)$ queries where r denotes the rank which can be as big as n.

Contents

1 Introduction

Matroid intersection. The matroid intersection problem is a fundamental combinatorial optimization problem that has been studied for over half a century. A wide variety of prominent optimization problems, such as bipartite matching, finding an arborescence, finding a rainbow spanning tree, and spanning tree packing, can be modeled as matroid intersection problems [Sch03, Chapter 41]. Hence the matroid intersection problem is a natural avenue to study all of these problems simultaneously.

Formally, a matroid is defined by the the tuple $\mathcal{M} = (V, \mathcal{I})$ where V is a finite set of size n, called the *ground set*, and $\mathcal{I} \subseteq 2^V$ is a family of subsets of V, known as the *independent sets*, that satisfy two properties: (i) $\mathcal{I}$ is *downward closed*, i.e., all subsets of any set in $\mathcal{I}$ are also in $\mathcal{I}$, and (ii) for any two sets $A, B \in \mathcal{I}$ with $|A| < |B|$, there is an element $v \in B \setminus A$ such that $A \cup \{v\} \in \mathcal{I}$, i.e., A can be *extended* by an element in B. Given two such matroids $\mathcal{M}_1 = (V, \mathcal{I}_1)$ and $\mathcal{M}_2 = (V, \mathcal{I}_2)$ defined over the same ground set V, the matroid intersection problem asks to output the largest common independent set $S \in \mathcal{I}_1 \cap \mathcal{I}_2$. The size of such a set is called *rank* and is denoted by r.

The classic version of this problem that has been studied since the 1960s assumes *independence query* access to the matroids: Given a matroid $\mathcal{M}$, an independence oracle takes a set $S \subseteq V$ as input and outputs a single boolean bit depending on whether $S \in \mathcal{I}$ or not, i.e., it outputs 1 iff $S \in \mathcal{I}$. The matroid intersection problem assumes the existence of two such independence oracles, one for each matroid. The goal is to design an efficient algorithm in order to minimize the number of such oracle accesses, i.e., to minimize the independence query complexity of the matroid intersection problem. This is the version of the problem that we study in this work. Note that a more powerful query model called *rank query* has been recently studied in [LSW15, CLS+19]. We do *not* consider such model.

Previous work. Starting with the work of Edmonds in the 1960s, algorithms with polynomial query complexity for matroid intersection have been studied [EDVJ68, Edm70, AD71, Law75, Edm79, Cun86, LSW15, Ngu19, CLS+19]. In 1986, Cunningham [Cun86] designed an algorithm with query complexity $O(nr^{1.5})$ based on the "blocking flow" ideas similar to Hopcroft-Karp's bipartite-matching algorithm or Dinic's maximum flow algorithm. This was the best query algorithm for the matroid intersection problem for close to three decades until the recent works of Nguyễn [Ngu19] and Chakrabarty-Lee-Sidford-Singla-Wong [CLS+19] who independently showed that Cunningham's algorithm can be implemented using only $\tilde{O}(nr)$ independence queries. In a separate line of work, Lee-Sidford-Wong [LSW15] proposed a cutting plane algorithm using $\tilde{O}(n^2)$ independence queries. When r is sublinear in n, the result of [CLS+19, Ngu19] provides faster (subquadratic) algorithm than that of [LSW15], but for linear r (i.e., $r \approx n$), all of these results are stuck at query complexity of $\tilde{O}(n^2)$. This is known as the *quadratic barrier* [CLS+19]. A natural question is whether this barrier can be broken [LSW15, Conjecture 13].

The only previous progress towards breaking this barrier is by [CLS+19] and falls under the following two categories. Either we need to assume the more powerful *rank* oracle model where [CLS+19] provides a $\tilde{O}(n^{1.5})$-time algorithm. Or, we solve an *approximate* version of the matroid intersection problem, where [CLS+19] provides an algorithm with $\tilde{O}(n^{1.5}/\varepsilon^{1.5})$ complexity for $(1-\varepsilon)$-approximately solving the matroid intersection problem in the independence oracle model. Breaking the quadratic barrier with an *exact* algorithm in the *independence* query model remains open.

Our results. We break the quadratic barrier with both deterministic and randomized algorithms:

Theorem 1.1 (Details in Theorems 4.8 and 4.9)**.** *Matroid Intersection can be solved by*

- *a deterministic algorithm taking $\tilde{O}(n^{11/6})$ independence queries, and*
- *a randomized (Las Vegas) algorithm taking $\tilde{O}(n^{9/5})$ independence queries with high probability.*

By high probability, we mean probability of at least $1 - 1/n^c$ for an arbitrarily large constant c. While we only focus on the query complexity in this paper, we note that the time complexities of our algorithms are dominated by the independence oracle queries. That is, our deterministic and randomized algorithms have time complexity $\tilde{O}(n^{11/6}\mathcal{T}_{\text{ind}})$ and $\tilde{O}(n^{9/5}\mathcal{T}_{\text{ind}})$ respectively, where $\mathcal{T}_{\text{ind}}$ denotes the maximum time taken by an oracle to answer an independence query.

Technical overview. Below we explain the key insights of our algorithms which are fast algorithms to solve a graph problem called *reachability* and a simple way to combine our algorithms with the existing exact and approximation algorithms to break the quadratic barrier.

Reachability problem: In this problem, there is a directed bipartite graph G on n vertices with bi-partition $(S \cup \{s,t\}, \bar{S})$. We want to determine whether a directed (s,t)-path exists in G. We know the vertices of G, but not the edge set E of G. We are allowed to ask the following two types of *neighborhood queries*:

1. Out-neighbor query: Given $v \in \bar{S}$ and $X \subseteq S \cup \{s,t\}$, does there exists an edge from v to some vertex in X?
2. In-neighbor query: Given $v \in \bar{S}$ and $X \subseteq S \cup \{s,t\}$, does there exists an edge from some vertex in X to v?

In other words, we can ask an oracle if a "right vertex" $v \in \bar{S}$ has an edge to or from a set X of "left vertices". This problem arose as a subroutine of previous matroid intersection algorithms that are based on finding augmenting paths [AD71, Law75, Cun86, CLS+19, Ngu19]. Naively, we can solve this problem with quadratic ($O(n^2)$) queries: find all edges of G by making a query for all possible $O(n^2)$ pairs of vertices. Cunningham [Cun86] used this algorithm in his framework to solve the matroid intersection problem with $O(nr^{1.5})$ queries. Recent results by [CLS+19, Ngu19] solved the reachability problem with $\tilde{O}(nd)$ queries, where d is the distance between s and t in G, essentially by simulating the breadth-first search process. Plugging these algorithms into Cunningham's framework leads to algorithms for the matroid intersection with $\tilde{O}(nr)$ queries. When d is large, the algorithms of [CLS+19, Ngu19] still need $\tilde{\Theta}(n^2)$ queries to solve the reachability problem. It is not clear how to solve this problem with a subquadratic number of queries. The key component of our algorithms is subquadratic-query algorithms for the reachability problem:

Theorem 1.2 (Details in Theorems 3.1 and 3.2)**.** *The reachability problem can be solved by*

- *a deterministic algorithm that takes $\tilde{O}(n^{5/3})$ queries, and*
- *a randomized (Las Vegas) algorithm that takes $\tilde{O}(n\sqrt{n})$ queries with high probability.*

Plugging Theorem 1.2 into standard frameworks such as Cunningham's does not directly lead us to a subquadratic-query algorithm for matroid intersection. Our second insight is a simple way to combine algorithms for the reachability problem with the exact and approximation algorithms of [CLS+19] to achieve the following theorem.

Theorem 1.3 (Details in Lemma 4.7)**.** *If there is an algorithm $\mathcal{A}$ that solves the reachability problem with $\mathcal{T}$ queries, then there is an algorithm $\mathcal{B}$ that solves the matroid intersection problem with $\tilde{O}(n^{9/5} + n\sqrt{\mathcal{T}})$ independence queries. If $\mathcal{A}$ is deterministic, then $\mathcal{B}$ is also deterministic.*

Theorems 1.2 and 1.3 immediately lead to Theorem 1.1. We provide proof ideas of Theorems 1.2 and 1.3 in the subsections below.

1.1 Proof idea for Theorem 1.2: Algorithm for the reachability problem

Before mentioning an overview of the algorithm for solving the reachability problem, we briefly mention what makes this problem hard. Note that if we discover that some $v \in \bar{S}$ is reachable from s, we can find all out-neighbors of v in $(S \cup \{s,t\})$ in $O(\log n)$ queries per such neighbor. We do this by using a binary search with out-neighbor queries, halving the size of the set of potential out-neighbors of v in each step. However, when we discover that some $v \in S$ is reachable from s, we cannot use the same binary-search trick to efficiently find the out-neighbors of v due to the *asymmetry* of the allowed queries, where we can make queries only for vertices $v \in \bar{S}$. Such asymmetry makes it hard to efficiently apply a standard (s,t)-reachability algorithm (such as breadth-first search) on the graph.[2]

Both our randomized and deterministic algorithms for the reachability problem follow the same framework below, where we partition vertices in $\bar{S}$ into *heavy and light* vertices and find vertices that can reach some heavy vertices. Our randomized and deterministic algorithms differ in how they determine whether a vertex is heavy or light.

Heavy/Light vertices. Our reachability algorithms run in phases and keep track of a set of vertices that are reachable from the source vertex s, denoted by F (for "found"). We can assume that F *contains all out-neighbors of vertices in* $F \cap \bar{S}$, because we can find these out-neighbors very efficiently by doing binary-search that makes $\tilde{O}(1)$ queries per out-neighbor. In each phase, the algorithm either

(a) increases the size of F by an additive factor of at least h for some parameter h (we use either $h = \sqrt{n}$ or $h = n^{1/3}$), or
(b) returns whether there is an (s,t) path.

Hence, in total, there are at most $\frac{n}{h}$ many phases. To this end, for every $v \in \bar{S}$, we say that v is F-*heavy* if either

(h1) v has at least h out-neighbors to $S \setminus F$, or
(h2) there is an edge from v to t.

If $v \in \bar{S}$ is not F-heavy, we say that it is F-*light*. We omit F when it is clear from the context. We emphasize that the notion of heavy and light applies only to vertices in $\bar{S}$. Two tasks that remain are how to determine if a vertex is heavy or light, and how to use this to achieve (a) or (b).

Heavy vertex reachability. First, we show how to achieve (a) or (b). We assume for now that we know which vertices in $\bar{S}$ are heavy or light. We can also assume that we know all out-going edges of all light vertices (e.g. black edges in Figure 1); this requires $\tilde{O}(nh)$ queries over all phases. Our main component is to determine a set of vertices that can reach *some* heavy vertex. (Heavy vertices are always in such set.) We can do this with $\tilde{O}(n)$ queries essentially by simulating a breadth-first search process *reversely* from heavy vertices. This process leaves us with subtrees rooted at the heavy vertices with edges pointed to the roots; see Figure 1 for an example. The actual algorithm is quite simple and can be found in Section 3.

Once all vertices that can reach some heavy vertices are found, we end up in one of the following situations:

- Some vertex in F can reach a heavy vertex v satisfying (h2). In this case, we know immediately that s can reach t via v.

[2] In contrast, in the *symmetric* case where in- and out-neighbor queries can be made for *every* vertex (and not just $v \in \bar{S}$), we can solve the reachability problem with $\tilde{O}(n)$ queries. This requires a simple breadth-first search starting from s where we discover neighbors using binary search.

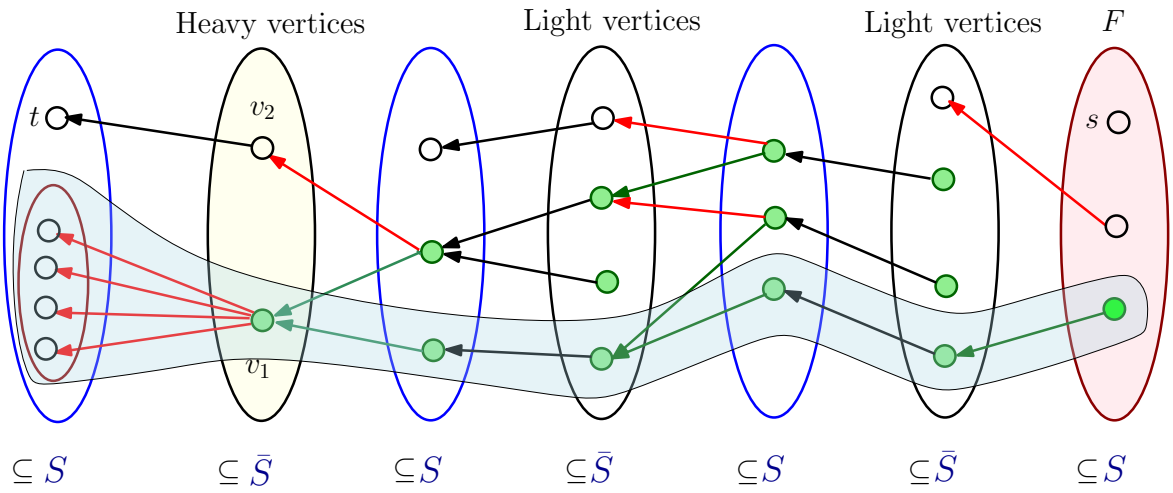

Figure 1: Example of the reverse BFS process to compute heavy vertex reachability. The vertices from S and $\bar{S}$ occur at alternate layers. The black edges (out-edges of light nodes) are known a priori. The green edges are traversed in the reverse BFS procedure whereas the red edges are not traversed in the reverse BFS. The green vertices are discovered in the reverse BFS. The green vertices form a tree rooted at v_1. Vertex v_1 is heavy because of its large out-degree. Vertex v_2 is heavy because t is its out-neighbor. The path from F to v_1 and the out-neighbors of v_1 are highlighted in light-blue, which is added to F after the reverse BFS. Note that, even though v_2 is reachable from F, the path from F to v_2 is not discovered, and hence the algorithm moves to the next iteration.

- Some vertex in F can reach a heavy vertex v satisfying (h1). In this case, we query and add all out-neighbors of v in $S \setminus F$ (taking $\tilde{O}(n)$ queries). This adds at least h vertices to F as desired.
- No vertices in F can reach any heavy vertex. In this case, we conclude that s does not reach t: to be able to reach t, s must be able to reach some vertex that points to t (and thus is heavy).

Heavy/light categorization. Again, this is where our randomized and deterministic algorithms differ. With randomness, we can use random sampling to approximate the out-degree of every vertex in $\bar{S}$ and find all out-going edges of vertices that are potentially light. This takes $\tilde{O}(nh + n^2/h)$ queries over all phases. For the deterministic algorithm, a naive idea is to maintain, for every $v \in \bar{S}$, up to h out-going neighbors of v in $S \setminus F$. The challenge is that when these neighbors are included in F, we have to find new neighbors. By carefully picking these neighbors, we can argue that in total only $\tilde{O}(n\sqrt{nh})$ queries are needed over all phases.

Summary. In total, in addition to the categorization of *heavy/light* vertices, we use $\tilde{O}(n)$ queries to solve the heavy vertex reachability problem in each of the $\tilde{O}(n/h)$ phases. We also need $\tilde{O}(nh)$ queries over all phases to find at most h out-neighbors in $S \setminus F$ of light vertices. So, in total, our algorithm uses $\tilde{O}(\frac{n^2}{h} + nh)$ queries plus the number of queries needed for the categorization, which is $\tilde{O}(\frac{n^2}{h} + nh)$ for randomized and $\tilde{O}(n\sqrt{nh})$ for deterministic algorithms.

1.2 Proof idea of Theorem 1.3: From reachability to matroid intersection

The standard connection between the matroid intersection problem and the reachability problem that is exploited by most combinatorial algorithms [AD71, Law75, Cun86, CLS+19, Ngu19] is based on finding augmenting paths in what is called the *exchange graph*. Given a common independent set

S of the two matroids over common ground set V, the *exchange graph* $G(S)$ is a directed bipartite graph over vertex set $V \cup \{s, t\}$ as in the reachability problem above with $\bar{S} = V \setminus S$. The edges of the exchange graph are defined to ensure the following property: Finding an (s, t)-path in the exchange graph amounts to augmenting S, i.e. finding a new common independent with a bigger size. Conversely, if no (s, t)-path exists in the exchange graph, it is known that S is of maximum cardinality and, hence, S can be output as the answer to the matroid intersection problem. Thus the problem of *augmentation* in the exchange graph can be reduced to the *reachability* problem where the neighborhood queries in the reachability problem correspond to the queries to the matroid oracles.[3]

Let us suppose that we can solve the *reachability* problem using $\mathcal{T}$ queries. An immediate and straightforward way of using this subroutine to solve matroid intersection is the following: Call this subroutine iteratively to find augmenting paths to augment along in the exchange graph, thereby increasing the size of the common independent set by one in each iteration. As the size of the largest common independent set is r, we need to perform r augmentations in total. This leads to an algorithm solving matroid intersection using $O(r\mathcal{T})$ independence queries.

To improve upon this, we avoid doing the majority of the augmentations by starting with a good approximation of the largest common independent set. We use the recent subquadratic $(1 - \varepsilon)$-approximation algorithm of [CLS+19, Section 6] that uses $\tilde{O}(n^{1.5}/\varepsilon^{1.5})$ independence queries to obtain a common independent set of size at least $r - \varepsilon r$. Once we obtain a common independent set with such approximation guarantee, we only need to perform an additional εr augmentations. This is still not good enough to obtain a subquadratic matroid intersection algorithm when combined with our efficient algorithms for the reachability problem from Theorem 1.2.

The final observation we make, is that for small $\varepsilon = o(n^{-1/5})$, we can combine the $\tilde{O}(n^{1.5}/\varepsilon^{1.5})$ approximation algorithm of [CLS+19] with an efficient implementation of Cunningham's algorithm (as in [CLS+19, Ngu19]) to obtain a $(1 - \varepsilon)$-approximation algorithm for matroid intersection using $\tilde{O}(n^{9/5} + n/\varepsilon)$ queries. This has a slightly better complexity than just running the approximation algorithm of [CLS+19]. The idea is to first run the the $\tilde{O}(n^{1.5}/\varepsilon'^{1.5})$ approximation algorithm with $\varepsilon' \approx n^{-1/5}$, and then run the Cunningham-style algorithm until the distance between s and t in the exchange graph becomes at least $\Theta(1/\varepsilon)$.

Our final algorithm is then:

1. Run the $\tilde{O}(n^{1.5}/\varepsilon^{1.5})$-query $(1 - \varepsilon)$-approximation algorithm from [CLS+19, Section 6] with $\varepsilon = n^{-1/5}$ to obtain a common independent set S of size at least $r - n^{4/5}$. This step takes $\tilde{O}(n^{9/5})$ queries.
2. Starting with S, run the Cunningham-style algorithm as implemented by [CLS+19, Section 5] until the (s, t)-distance is at least $\sqrt{\mathcal{T}}$ to obtain a common independent set of size at least $r - O(n/\sqrt{\mathcal{T}})$. This step takes $\tilde{O}(n(r - |S|) + n\sqrt{\mathcal{T}}) = \tilde{O}(n^{9/5} + n\sqrt{\mathcal{T}})$ queries.
3. For the remaining $O(n/\sqrt{\mathcal{T}})$ augmentations, find augmenting paths one by one by solving the reachability problem. This step takes $\tilde{O}(n\sqrt{\mathcal{T}})$ queries.

Hence we obtain a matroid intersection algorithm which uses $\tilde{O}(n^{9/5} + n\sqrt{\mathcal{T}})$ independence queries, as in Theorem 1.3.

1.3 Organization

We start with the necessary preliminaries in Section 2. In Section 3, we provide the subquadratic deterministic and randomized algorithms for augmentation. Finally, in Section 4, we combine these algorithms for augmentation with existing algorithms to obtain subquadratic deterministic

[3]The independence queries are more powerful than the neighborhood queries, but we are only interested in the neighborhood queries in our algorithm for the reachability problem.

and randomized algorithms for matroid intersection. In Section 3, we skip the description of an important subroutine called the heavy/light categorization. We devote Section 5 for details of this subroutine.

2 Preliminaries

Matroid. A *matroid* is a combinatorial object defined by the tuple $\mathcal{M} = (V, \mathcal{I})$, where the ground set V is a finite set of elements and $\mathcal{I} \subseteq 2^V$ is a non-empty family of subsets (denoted as the *independent sets*) of the ground set V, such that the following properties hold:

1. **Downward closure:** If $S \in \mathcal{I}$, then any subset $S' \subset S$ (including the empty set) is also in $\mathcal{I}$,
2. **Exchange property:** For any two sets $S_1, S_2 \in \mathcal{I}$ with $|S_1| < |S_2|$, there is an element $v \in S_2 \setminus S_1$ such that $S_1 \cup \{v\} \in \mathcal{I}$.

Matroid Intersection. Given two matroids $\mathcal{M}_1 = (V, \mathcal{I}_1)$ and $\mathcal{M}_2 = (V, \mathcal{I}_2)$ defined on the same ground set V, the *matroid intersection problem* is finding a maximum cardinality common independent set $S \in \mathcal{I}_1 \cap \mathcal{I}_2$. When discussing matroid intersection, we will denote by r the size of such a maximum cardinality common independent set and by n the size of the ground set V.

Exchange graph. Consider two matroids $\mathcal{M}_1 = (V, \mathcal{I}_1)$ and $\mathcal{M}_2 = (V, \mathcal{I}_2)$ defined on the same ground set V. Let $S \in \mathcal{I}_1 \cap \mathcal{I}_2$ be a common independent set. The *exchange graph* $G(S)$, w.r.t. to the common independent set $S \in \mathcal{I}_1 \cap \mathcal{I}_2$, is defined to be a directed bipartite graph where the two sides of the bipartition are S and $\bar{S} = V \setminus S$. Moreover, there are two additional special vertices s and t (that are not included in either S or $\bar{S}$) which have directed edges incident on them *only* from $\bar{S}$. The directed edges (or arcs) are interpreted as follows:

1. Any edge of the form (s, v) for $v \in \bar{S}$ implies that $S \cup \{v\}$ is an independent set in $\mathcal{M}_1$.
2. Similarly, any edge of the form (v, t) for $v \in \bar{S}$ implies that $S \cup \{v\}$ is an independent set in $\mathcal{M}_2$.
3. Any edge of the form $(u, v) \in S \times \bar{S}$ implies that $(S \setminus \{u\}) \cup \{v\}$ is an independent set in $\mathcal{M}_1$.
4. Similarly, any edge of the form $(v, u) \in \bar{S} \times S$ implies that $(S \setminus \{u\}) \cup \{v\}$ is an independent set in $\mathcal{M}_2$.

We are interested in the notion of *chordless* (s, t)-paths in $G(S)$ [Cun86, Section 2] which are defined next. For this definition, we consider a path as a sequence of vertices that take part in the path. A subsequence of a path is an ordered subset of the vertices (not necessarily contiguous) of the path where the ordering respects the path ordering.

Definition 2.1. An (s, t)-path p is *chordless* if there is no proper subsequence of p which is also an (s, t)-path. A chordless path in the exchange graph $G(S)$ is sometimes called an *augmenting path*.

Claim 2.2 (Augmenting path)**.** *Consider a chordless path p from s to t in $G(S)$ (if it exists), and let $V(p)$ be the elements of the ground set (or, equivalently, vertices in the exchange graph excluding s and t) that take part in the path p. Then $S \triangle V(p)$ is a common independent set of $\mathcal{M}_1$ and $\mathcal{M}_2$.*

If we examine the set $S \triangle V(p)$ obtained from Claim 2.2, it is clear that the number of elements added to the set S is one more than the number of elements removed from S. This observation immediately gives the following corollary, and shows the importance of the notion of exchange graphs.

Corollary 2.3. *The size of the largest common independent set of $\mathcal{M}_1$ and $\mathcal{M}_2$ is at least $|S|+1$ if and only if t is reachable from s in $G(S)$.*

It is useful to note that the shortest (s,t)-path in $G(S)$ is always chordless. Many combinatorial matroid intersection algorithms thus focus on finding shortest (s,t)-paths. The following claim relating the distance from s to t in $G(S)$ and the size of S is useful for approximation algorithms for matroid intersection.

Claim 2.4 ([Cun86])**.** *If the length of the shortest (s,t)-path in $G(S)$ is at least d, then $|S| \geq (1-O(\frac{1}{d}))r$, where r is the size of the largest common independent set.*

Matroid query oracles. There are two primary models of query oracles associated with the matroid theory: (i) the independence query oracle, and (ii) the rank query oracle. The independence query oracle, given a set $S \subseteq V$ of a matroid $\mathcal{M}$, outputs 1 iff S is an independent set of $\mathcal{M}$ (i.e., iff $S \in \mathcal{I}$). The rank query oracle, given a set $S \subseteq V$, outputs the rank of S, $\text{rk}_{\mathcal{M}}(S) \stackrel{def}{=} \max_{T \subseteq S: T \in \mathcal{I}} |T|$, i.e., the size of the largest independent set contained in S. Clearly, if S itself is an independent set, then $\text{rk}_{\mathcal{M}}(S) = |S|$. Hence, a rank query oracle is at least as powerful as the independence query oracle. In this work, we are however interested primarily in the independence query oracle model. Next, we state two claims regarding the independence query oracle that we use in the paper.

Claim 2.5 (Edge discovery)**.** *By issuing one independence query each, we can find out*

(i) given a vertex $v \in \bar{S}$, whether v is an out-neighbor of s; or

(ii) given a vertex $v \in \bar{S}$, whether v is an in-neighbor of t; or

(iii) given a vertex $v \in \bar{S}$ and a subset $X \subseteq S$, whether there exists an edge from some vertex in X to v; or

(iv) given a vertex $v \in \bar{S}$ and a subset $X \subseteq S$, whether there exists an edge from v to some vertex in X.

Claim 2.5 follows from observing that we can make the following kinds of independence queries: (i-ii) whether $S \cup \{v\}$ is an independent set in $\mathcal{M}_1$ respectively $\mathcal{M}_2$, and (iii-iv) whether $S \cup \{v\} \setminus X$ is an independent set in $\mathcal{M}_1$ respectively $\mathcal{M}_2$. Note that these edge-discovery queries can simulate the neighborhood-queries in the reachability problem.

With these kinds of queries, we can perform a binary search to find an in-/out-neighbor of $v \in \bar{S}$. The following lemma is proven in [CLS+19, Lemma 11] and also mentioned in [Ngu19]. We skip the proof in the paper.

Claim 2.6 (Binary search with independence/neighborhood queries, [Ngu19, CLS+19])**.** *Consider a vertex $v \in \bar{S}$ and a subset $X \subseteq S \cup \{s,t\}$. By issuing $O(\log r)$ independence queries to $\mathcal{M}_1$, we can find a vertex $u \in X$ such that there is an edge (u,v) (i.e., u is an in-neighbor of v), or otherwise determine that no such edge exists. Similarly, by issuing $O(\log r)$ independence queries to $\mathcal{M}_2$, we can find a vertex $u' \in X$ such that there is an edge (v,u') (i.e., u' is an out-neighbor of v).*

We will assume InEdge(v,X) respectively OutEdge(v,X) are procedures which implement Claim 2.6.

3 Algorithms for augmentation

From Claim 2.2, we know the following: Given a common independent set S, either S is of maximum cardinality or there exists a (directed) (s,t)-path in the exchange graph $G(S)$. In this section, we consider the (s,t)-reachability problem in $G(S)$ using independence oracles. Our main results in this section are the following two theorems. We denote the size of S as $|S| = r$ in both of these theorems.[4]

Theorem 3.1 (Randomized augmentation)**.** *There is a randomized algorithm which with high probability uses $O(n\sqrt{r}\log n)$ independence queries and either determines that S is of maximum cardinality or finds an augmenting path in $G(S)$.*

Theorem 3.2 (Deterministic augmentation)**.** *There is a deterministic algorithm which uses $O(nr^{2/3}\log r)$ independence queries and either determines that S is of maximum cardinality or finds an augmenting path in $G(S)$.*

3.1 Overview of the algorithms

Section 1.1 gives an informal overview of the augmentation algorithm already. In this section, we provide more details so that the reader can be convinced about the correctness of the algorithm.

The algorithm for augmentation, denoted as AUGMENTATION algorithm for easy reference, runs in phases and keeps track of a set F of vertices that are reachable from the vertex s. Let F_S and $F_{\bar{S}}$ denote the bipartition of F inside S and $\bar{S}$, i.e., $F_S = F \cap S$ and $F_{\bar{S}} = F \cap \bar{S}$. In each phase, the algorithm will increase the size of F_S by an additive factor of at least h until the algorithm discovers an (s,t)-path (or, otherwise, discover there is no such path). Hence, in total, there are at most $\frac{|S|}{h}$ many phases. We now give an overview of how to implement each phase.

Note that, without loss of generality, we can assume that the set F_S contains all vertices that are out-neighbors of vertices in $F_{\bar{S}}$. This is because whenever a vertex $v \in \bar{S}$ is added to $F_{\bar{S}}$, we can quickly add all of v's out-neighbors in $S \setminus F_S$ into the set F_S by using Claim 2.6. This requires $O(\log r)$ independence queries for each such out-neighbor. Hence, in total, this procedure uses at most $O(n \log r)$ independence queries, since each $u \in S \setminus F_S$ is added in F_S at most once.

Heavy and light vertices. Before explaining what the algorithm does in each phase, we introduce the notion of *heavy* and *light* vertices: We divide the vertices in $\bar{S} \setminus F_{\bar{S}}$ into two categories. We call a vertex $v \in \bar{S} \setminus F_{\bar{S}}$ *heavy* if it either has an edge to t or has at least h out-neighbors in $S \setminus F_S$. The vertices in $\bar{S} \setminus F_{\bar{S}}$ that are not *heavy* are denoted as *light* (See Figure 1 for reference; the heavy nodes are highlighted in light-yellow). Note that both these notions are defined in terms of out-degrees, i.e., a heavy vertex can have arbitrary in-degree and so can a light vertex. Also, note that the notion of heavy and light vertices are defined w.r.t. to the set F_S. Because the set F_S changes from one phase to the next, so does the set of heavy vertices and light vertices.

Description of phase i. Let us assume, for the time being, that there is an efficient procedure to categorize the vertices in $\bar{S} \setminus F_{\bar{S}}$ into the sets of heavy and light vertices. We first apply this procedure at the beginning of phase i.

Now, for simplicity, consider an easy case: In phase i, there is a *heavy* vertex that has an in-neighbor in F_S. In this case, we can go over all vertices in $\bar{S} \setminus F_{\bar{S}}$ to find such a heavy vertex—this can be done with n many independence queries. Once we find such a heavy vertex, we include it in

[4] Note that r usually denotes the size of the maximum common independent set which is an upper bound on the size of the vertex set S. We abuse the notation and use r here to denote $|S|$.

F_S and all of its out-neighbors in $F_{\bar{S}}$. Note that, in this case, either of the following two things can happen: either we have increased the size of F_S by at least h as the heavy vertex has at least h out-neighbors in $S \setminus F_S$; or the heavy vertex we found has t as its out-neighbor in which case we have found an (s,t)-path.

Unfortunately, this may not be the case in phase i. In this case, we do an additional procedure called the *reverse breadth-first search* or, in short, *reverse BFS*. The goal of the reverse BFS is to find a heavy vertex reachable from F. Before describing this procedure, note the following two properties of the *light* vertices:

1. A light vertex will remain a light vertex even if we increase the size of F_S.
2. We can assume that we know all out-neighbors of any light vertex.

Property 2 needs some explanation. This property is true because of two observations: (i) All out-neighbors of a light vertex can be found out with $O(h \log n)$ independence queries using Claim 2.6, and (ii) because of Property 1, across all phases, we need to find out the out-neighbors of a light vertex *only once*. So, even though we need to make $O(nh \log n)$ queries in total, this cost amortizes across all phases.

The idea is, as before, to discover a heavy vertex which is reachable from F so that we can include all of its out-neighbors in F_S (for example, consider the heavy vertex v_1 in Figure 1). So our goal is to find some path from F to a heavy vertex (Consider the path starting from v_1 highlighted in light-blue in Figure 1). This naturally implies the need for doing a reverse BFS from the heavy vertices. We also note that any path from F to t must pass through a heavy vertex (the vertex just preceding t must by definition be heavy). Hence, if our reverse BFS fails to find a path from F to some heavy vertex, the algorithm has determined that no (s,t)-path exists.

What remains is to find out how to implement the reverse BFS procedure efficiently. To this end, we exploit Property 2 of light vertices and assume that we know all edges directed from $\bar{S}$ to S that the reverse BFS procedure needs to visit. This follows from the following crucial observation: *No internal node of the reverse BFS forest is a heavy node*, i.e., in other words, the heavy vertices occur *only* as root nodes of the reverse BFS trees. This is because if, along the traversal of a reverse BFS procedure starting from a heavy node v, we reach another heavy node v', we can ignore v' as the reverse BFS starting from node v' has already taken care of processing v'. This means that any edge in $\bar{S} \times S$ that takes part in the reverse BFS procedure must originate from a light vertex and, hence, is known a priori due to Property 2. All it remains for the reverse BFS procedure is to discover in-neighbors of vertices in $\bar{S}$ using edges from $S \times \bar{S}$. By Claim 2.6, each such in-neighbor can be found by making $O(\log r)$ independence queries. In total, the reverse BFS procedure uses $\tilde{O}(n)$ independence queries.

Post-processing. Note that, in order to use Claim 2.2, the (s,t)-path needs to be chordless. However, the (s,t)-path p that the algorithm outputs has no such guarantee. So, as a post-processing step, the algorithm uses an additional $\tilde{O}(r)$ independence queries to convert this path into a *chordless* path: Consider any vertex $v \in V(p) \cap \bar{S}$ and assume u as the parent of v, and w as the child of v in the path p. The vertex v needs to check whether it has an in-neighbor other than u among the ancestors of v in $V(p)$ or an out-neighbor other than w among the descendants of v in $V(p)$. Since the length of the path obtained from the previous step is $O(r)$ (because of $|S| = r$ and the path does not contain any cycle), this requires $O(\log r)$ independence queries. If all vertices in $V(p) \cap \bar{S}$ have no such in or out-neighbors, then it is easy to see that p is indeed a chordless path. If there is such a (say) in-neighbor u' of v, then we remove all vertices of $V(p)$ between u' and v, and the resulting subsequence is still an (s,t)-path. A similar procedure is done when an out-neighbor is

discovered. In total, this takes $O(r \log r)$ independence queries, since each vertex can be removed from the path at most once.

Cost analysis. The total number of queries needed to implement phase i is a summation of two terms: (i) the number of queries needed to partition the vertices into heavy and light categories, and (ii) the number of queries needed to run the reverse BFS procedure. We have seen that (ii) can be implemented with $\tilde{O}(n)$ independence queries. For (i), we present two algorithms: a randomized sampling algorithm, and a deterministic algorithm which is slightly less efficient than the randomized one. This is the main technical difference between the algorithm of Theorem 3.1 and that of Theorem 3.2. The cost analysis for (i) is also amortized and the total number of queries needed across all phases is $\tilde{O}(\max\{nh, nr/h\})$ for randomized and $\tilde{O}(n\sqrt{rh})$ for deterministic implementation. Setting $h = \sqrt{r}$ for randomized and $h = r^{1/3}$ for deterministic, we see that total randomized query complexity of augmentation is $\tilde{O}(n\sqrt{r})$ and deterministic query complexity is $\tilde{O}(nr^{2/3})$.

3.2 Categorizing *heavy* and *light* vertices

We start with reminding the readers the definition of the *heavy* and *light* vertices in $\bar{S} \setminus F_{\bar{S}}$.

Definition 3.3. We call a vertex $v \in \bar{S} \setminus F_{\bar{S}}$ heavy if either (v, t) is an edge of $G(S)$ or v has at least h out-neighbors in $S \setminus F_S$. Otherwise we call v light.

To check whether v has an edge to t is easy and requires only a single independence query: "Is $S \cup \{v\}$ independent in $\mathcal{M}_2$?" The difficulty lies when this is not the case and we need to determine if v has outdegree at least h to $S \setminus F_S$. We present two algorithms to solve this categorization problem: one randomized sampling algorithm; and a less efficient deterministic algorithm. More concretely, we show the following two lemmas.

Lemma 3.4. *There is a randomized categorization procedure which, with high probability, categorizes heavy and light vertices in the set $\bar{S} \setminus F_{\bar{S}}$ correctly by issuing $O(n \log n)$ independence queries per phase and an additional $O(nh \log n)$ independence queries over the whole run of the* Augmentation *algorithm.*

Lemma 3.5. *There exists a deterministic categorization procedure which uses $O(n\sqrt{rh} \log r)$ queries over the whole run of the* Augmentation *algorithm.*

The proofs of these two lemmas are deferred to Section 5.

3.3 Heavy vertex reachability

In this section, we present the reverse BFS in Algorithm 3.7 and analyze some properties of it. Recall that the reverse BFS is run once in each phase of the algorithm to find some vertex in F which can reach some heavy vertex. We also remind the reader of the example in Figure 1. In this section, we prove the following.

Lemma 3.6 (Heavy vertex reachability)**.** *There is an algorithm (Algorithm 3.7:* ReverseBFS*) which, given F such that there are no edges from $F_{\bar{S}}$ to $S \setminus F_S$, a categorization of $\bar{S} \setminus F_{\bar{S}}$ into* heavy *and* light*, and all out-edges of the light vertices to $S \setminus F_S$, uses $O(n \log r)$ queries and either finds a path from some vertex in F to a heavy node, or otherwise determines that no such path exists.*

We next provide the pseudo-code (Algorithm 3.7).

Algorithm 3.7 ReverseBFS

Input: Categorization of $\bar{S} \setminus F_{\bar{S}}$ into *heavy* and *light*; and a set LightEdges containing all out-edges of the light vertices.
Output: A path from F to some heavy vertex, if one exists.

1: $Q \leftarrow \{v \in \bar{S} \setminus F_{\bar{S}}$ which are heavy$\}$
2: NotVisited $\leftarrow (S \cup \bar{S} \cup \{s, t\}) \setminus Q$
3: **while** $Q \neq \emptyset$ **do**
4: Pop a vertex v from Q.
5: **if** $v \in F$ **then**
6: **return** the path from v to a heavy vertex in the BFS-forest.
7: **else if** $v \in \bar{S} \setminus F_{\bar{S}}$ **then**
8: **while** $u =$ InEdge$(v,$ NotVisited$)$ is not $\emptyset$ **do**
9: Push u to Q and remove it from NotVisited.
10: **else if** $v \in S \setminus F_S$ **then**
11: **for** $u \in$ NotVisited such that $(u, v) \in$ LightEdges **do**
12: Push u to Q and remove it from NotVisited.
13: **return** "NO PATH EXISTS"

Correctness. We first argue that the algorithm is correct. When a vertex $v \in \bar{S} \setminus F_{\bar{S}}$ is processed by the algorithm, each unvisited in-neighbor will be added to the queue Q in the while loop in line 8. When a vertex $v \in S \setminus F_S$ is processed by the algorithm, any edge from NotVisited to v must originate from a light vertex, since NotVisited contains no heavy vertices and we are guaranteed that no edge from $F_{\bar{S}}$ to $S \setminus F_S$ exist. Hence Algorithm 3.7 will eventually process every vertex reachable, by traversing edges in reverse, from the heavy vertices.

Cost analysis. The only place Algorithm 3.7 uses independence queries is in line 8. Each vertex will be discovered at most once by the binary search in InEdge. This means that we do at most n calls to InEdge, each using $O(\log r)$ queries by Claim 2.6. Hence the reverse BFS uses $O(n \log r)$ queries per phase.

3.4 Augmenting path algorithm.

We now present the main augmenting path algorithm, as explained in the overview in Section 3.1.

Algorithm 3.8 AUGMENTATION

Input: Two matroids $\mathcal{M}_1 = (V, \mathcal{I}_1)$, and $\mathcal{M}_2 = (V, \mathcal{I}_2)$ and a common independent set $S \subseteq \mathcal{I}_1 \cap \mathcal{I}_2$.
Output: An augmenting (s,t)-path in $G(S)$ if one exists.

1: $F \leftarrow \{s\}$
2: LIGHTEDGES $\leftarrow \emptyset$

3: **while** $t \notin F$ **do**

Description of a phase.

4: Categorize $v \in \bar{S} \setminus F_{\bar{S}}$ into *heavy* and *light*. ▷ See Sections 5.1 and 5.2.
5: **for** each new *light* vertex v **do**
6: Use OUTEDGE to find all out-neighbors of v in $S \setminus F_S$.
7: Add edges (v,u) to LIGHTEDGES for each such out-neighbor u.

8: $p \leftarrow$ REVERSEBFS$(S, F,$ LIGHTEDGES$)$ ▷ See Section 3.3.
9: **if** $p =$ "NO PATH EXISTS" **then**
10: **return** "NO PATH EXISTS"
11: **else** ▷ p is a path from F to a heavy vertex
12: Denote by $V(p)$ the vertices on the path p.
13: Add all $v \in V(p)$ to F.
14: **for** $v \in V(p) \cap \bar{S}$ **do**
15: **while** $u =$ OUTEDGE$(v, (S \setminus F) \cup \{t\})$ is not $\emptyset$ **do**
16: Add u to F.

Post-processing.

17: Post-process the (s,t)-path found to make it *chordless*.
18: **return** the augmenting path.

Note that we have not specified if we are using the randomized or deterministic categorization of heavy and light vertices, from Sections 5.1 and 5.2. We will for now assume this categorization procedure as a black box which is always correct.

We start by stating some invariants of Algorithm 3.8.

1. F contains only vertices reachable from s. In fact, for each vertex in F we have found a path from s to this vertex.

2. LIGHTEDGES contains all out-edges from light vertices to $S \setminus F$.

3. In the beginning of each phase, there exists no $v \in F$, $u \in (S \cup \{s,t\}) \setminus F$ such that (v,u) is an edge in $G(S)$. This is because whenever $v \in \bar{S}$ is added to F, all v's neighbors are also added, see line 15.

Correctness. When the algorithm outputs an (s,t)-path, the path clearly exists, by Invariant 1. So it suffices to argue that the algorithm does not return "NO PATH EXISTS" incorrectly. Note that the algorithm only returns "NO PATH EXISTS" when REVERSEBFS does so, that is when there is no path from F to a heavy vertex (by Lemma 3.6). So suppose that this is that case, and

also suppose, for the sake of a contradiction, that an (s,t)-path p exists in $G(S)$. Denote by v the vertex preceding t in the path p. By Invariant 3 we know that v is not in F. But then v is heavy, since (v,t) is an edge of $G(S)$. Hence a subpath of p will be a path from F to the heavy vertex v, which is the desired contradiction.

Number of phases. We argue that there are at most $\frac{r}{h}+1$ phases of the algorithm. After a phase, either the algorithm returns "NO PATH EXISTS" (in which case this was the last phase), or some path p was found by the reverse BFS. Then $V(p)$ must include some heavy vertex v. Then all neighbors of v will be added to F in line 15. Thus we know that either t was added to F (in which case this was the last phase), or at least h vertices from S was added to F. Since $|S| \leq r$ in the beginning, this can happen at most $\frac{r}{h}$ times.

Number of queries. We analyse the number of independence queries used by different parts of the algorithm:

- ReverseBFS (Algorithm 3.7) is run once each phase, and uses $O(n \log r)$ queries per call by Lemma 3.6. This contributes a total of $O(\frac{nr \log r}{h})$ independence queries over all phases.
- Each $u \in S$ is discovered at most once by the OutEdge call on line 15. So this line contributes a total of $O(n \log r)$ independence queries.
- Each vertex becomes *light* at most once over the run of the algorithm. When this happens, the algorithm finds all of its (up to h) out-neighbors on line 6, using OutEdge calls. This contributes a total of $O(nh \log r)$ independence queries.
- The post-processing can be performed using $O(r \log r)$ independence queries, as explained in Section 3.1.
- The *heavy*/*light*-categorization uses $O(nh \log n + \frac{nr \log n}{h})$ independence queries when the randomized procedure is used, by Lemma 3.4. When the deterministic categorization procedure is used, we use $O(n\sqrt{rh} \log r)$ independence queries instead, by Lemma 3.5.

We see that in total, the algorithm uses:

- $O(n\sqrt{r} \log n)$ independence queries with the randomized categorization, setting $h = r^{1/3}$.
- $O(nr^{2/3} \log r)$ independence queries with the deterministic categorization, setting $h = \sqrt{r}$.

The above analysis proves Theorems 3.1 and 3.2.

Remark 3.9. When the randomized categorization procedure fails, Algorithm 3.8 will still always return the correct answer, but it might use more independence queries. So Algorithm 3.8 is in fact a Las-Vegas algorithm with expected query-complexity $O(n\sqrt{r} \log n)$.

Remark 3.10. We note that our algorithm can not be used to find which vertices are reachable from s using subquadratic number of queries.

4 Algorithm for fast Matroid Intersection

There are two hurdles to getting a subquadratic algorithm for Matroid Intersection. Firstly, standard augmenting path algorithms need to find the augmenting paths one at a time. This is since after augmenting along a path, the edges in the exchange graph change (some edges are added, some removed). This is unlike bipartite matching, where a set of vertex-disjoint augmenting paths can be

augmented along in parallel. It is not clear how to find the augmenting paths faster than $\Theta(n)$ each, so these standard augmenting path algorithms are stuck at $\Omega(nr)$ independence queries.

To overcome this, Chakrabarty-Lee-Sidford-Singla-Wong [CLS+19, Section 6] introduce the notion of *augmenting sets*, which allows multiple parallel augmentations. Using the augmenting sets they present a subquadratic $(1-\varepsilon)$-approximation algorithm using $\tilde{O}(\frac{n^{1.5}}{\varepsilon^{1.5}})$ independence queries:

Lemma 4.1 (Approximation algorithm [CLS+19])**.** *There exists an $(1-\varepsilon)$ approximation algorithm for matroid intersection using $O(\frac{n\sqrt{n\log r}}{\varepsilon\sqrt{\varepsilon}})$ independence queries.*

The second hurdle is that when the distance d between s and t is high, the breadth-first algorithms of [CLS+19, Ngu19] use $\tilde{\Theta}(dn)$ independence queries to compute the distance layers, which is $\Omega(nr)$ when $d \approx r$.[5] Here our algorithm from Section 3 helps since it can find a single augmenting path using a subquadratic number of independence queries, even when the distance d is large.

So our idea is as follows:

- Start by using the subquadratic approximation algorithm. This avoids having to do the majority of augmentations one by one.
- Continue with the fast implementation [CLS+19, Section 5] of the Cunningham-style blocking flow algorithm.
- When the (s,t)-distance becomes too large, fall back to using the augmenting-path algorithm from Section 3 to find the (few) remaining augmenting paths.

Algorithm 4.2 subquadratic Matroid Intersection

1: Run the approximation algorithm (Lemma 4.1) with $\varepsilon = n^{1/5}r^{-2/5}\log^{-1/5} r$ to obtain a common independent set S of size at least $(1-\varepsilon)r = r - n^{1/5}r^{3/5}\log^{-1/5} r$.

2: Starting with S, run Cunningham's algorithm (as implemented by [CLS+19]), until the distance between s and t becomes larger than d.

3: Keep running AUGMENTATION (Algorithm 3.8) from Section 3 and augmenting the current common independent set with the obtained (s,t)-path (as in Claim 2.2) until no (s,t)-path can be found in the exchange graph.

The choice of d will be different depending on whether we use the randomized or deterministic version of Algorithm 3.8. In order to run Algorithm 4.2, we need to know r so that we may choose ε (and d) appropriately. However, the size r of the largest common independent set is unknown. We note that it suffices, for the purpose of the asymptotic analysis, to use a $\frac{1}{2}$-approximation $\bar{r}$ for r (that is $\bar{r} \le r \le 2\bar{r}$). It is well known that such an $\bar{r}$ can be found in $O(n)$ independence queries by greedily finding a maximal common independent set in the two matroids. Now we can bound the query complexity of Algorithm 4.2.

Lemma 4.3. *Line 1 of Algorithm 4.2 uses $O(n^{6/5}r^{3/5}\log^{4/5} r)$ independence queries.*

[5] Note that unlike in Section 3, we now use the normal definition of r as the size of the maximum-cardinality common independent set of the two matroids.

Proof. The approximation algorithm uses $O(\frac{n^{1.5}\sqrt{\log r}}{\varepsilon^{1.5}}) = O(n^{6/5}r^{3/5}\log^{4/5} r)$ independence queries, when $\varepsilon = n^{1/5}r^{-2/5}\log^{-1/5} r$. □

Lemma 4.4. *Line 2 of Algorithm 4.2 uses $O(n^{6/5}r^{3/5}\log^{4/5} r + nd\log r)$ independence queries.*

Proof. There are two main parts of Cunningham's blocking-flow algorithm.

- Computing the distances. The algorithm will run several BFS's to compute the distances. The total number of independence queries for all of these BFS's can be bounded by $O(dn\log r)$, since the distances are monotonic so each vertex is tried at a specific distance at most once. For more details, see [CLS+19, Section 5.1].
- Finding the augmenting paths. Given the distance-layers, a single augmenting path can be found in $O(n\log r)$ independence queries, by a simple depth-first-search. Again, we refer to [CLS+19, Section 5.2] for more details. Since we start with a common independent set S of size $(1-\varepsilon)r = r - n^{1/5}r^{3/5}\log^{-1/5} r$, we know that S can be augmented at most $n^{1/5}r^{3/5}\log^{-1/5} r$ additional times. Hence a total of $O(n^{6/5}r^{3/5}\log^{4/5} r)$ independence queries suffices to find all of these augmenting paths.

□

Remark 4.5. We note that if we skip Line 3 in Algorithm 4.2, we thus get a $(1-\frac{1}{d})$-approximation algorithm (by Claim 2.4), using $\tilde{O}(n^{6/5}r^{3/5}+nd)$ independence queries, which beats the $\tilde{O}(n^{1.5}/\varepsilon^{1.5})$ approximation algorithm when $\varepsilon = o(n^{1/5}r^{-2/5})$.

Lemma 4.6. *Line 3 of Algorithm 4.2 uses $O(\frac{r}{d}\mathcal{T})$ independence queries, where $\mathcal{T}$ is the number of independence queries used by one invocation of* Augmentation *(Algorithm 3.8).*

Proof. After line 2, the algorithm has found a common independent set of size at least $(1-O(\frac{1}{d}))r = r - O(\frac{r}{d})$, by Claim 2.4. This means that only $O(\frac{r}{d})$ additional augmentations need to be performed. □

By Lemmas 4.3, 4.4 and 4.6, we see that Algorithm 4.2 uses a total of $O(n^{6/5}r^{3/5}\log^{4/5} r + nd\log r + \frac{r}{d}\mathcal{T})$ independence queries. If we pick $d = \sqrt{\frac{r\mathcal{T}}{n\log r}}$ we get the following lemma.

Lemma 4.7. *If the query complexity of* Augmentation *is $\mathcal{T}$, then matroid intersection can be solved using $O(n^{6/5}r^{3/5}\log^{4/5} r + \sqrt{nr\mathcal{T}\log r})$ independence queries.*

Combining with Theorems 3.1 and 3.2 we get our subquadratic results.

Theorem 4.8 (Randomized Matroid Intersection)**.** *There is a randomized algorithm which with high probability uses $O(n^{6/5}r^{3/5}\log^{4/5} r)$ independence queries and solves the matroid intersection problem. When $r = \Theta(n)$, this is $\tilde{O}(n^{9/5})$.*

Theorem 4.9 (Deterministic Matroid Intersection)**.** *There is a deterministic algorithm which uses $O(nr^{5/6}\log r + n^{6/5}r^{3/5}\log^{4/5} r)$ independence queries and solves the matroid intersection problem. When $r = \Theta(n)$, this is $\tilde{O}(n^{11/6})$.*

Remark 4.10. The limiting term for the the randomized algorithm is between line 1 and line 2. If a faster approximation algorithm is found, the same strategy as above might give an $\tilde{O}(nr^{3/4})$-query algorithm.

5 Algorithm for heavy/light categorization

In this section, we finally provide the algorithm for the categorization of vertices in $\bar{S} \setminus F_{\bar{S}}$ into heavy and light vertices as defined in Definition 3.3.

5.1 Randomized categorization

In this section, we prove the following lemma (restated from Section 3.2).

Lemma 3.4. *There is a randomized categorization procedure which, with high probability, categorizes heavy and light vertices in the set $\bar{S} \setminus F_{\bar{S}}$ correctly by issuing $O(n \log n)$ independence queries per phase and an additional $O(nh \log n)$ independence queries over the whole run of the* AUGMENTATION *algorithm.*

We will use X to denote $S \setminus F$. Let the out-neighborhood of a vertex $v \in \bar{S} \setminus F_{\bar{S}}$ inside X be denoted as $\mathsf{Ngh}_X(v)$. Consider the family of sets $\{\mathsf{Ngh}_X(v)\}_{v \in \bar{S} \setminus F_{\bar{S}}}$ residing inside the ambient universe X. We want to find out which of these sets are of size at least h (i.e., correspond to the heavy vertices) and which of them are not (i.e., corresponds to the light vertices). To this end, we devise the following random experiment.

Experiment 5.1. *Sample a set R of k elements drawn uniformly and independently from X (with replacement) and check whether $R \cap \mathsf{Ngh}_X(v) = \emptyset$.*

It is easy to check the following: For any $v \in \bar{S} \setminus F_{\bar{S}}$, Experiment 5.1 is successful with probability:

$$\Pr_R[R \cap \mathsf{Ngh}_X(v) = \emptyset] = \left(1 - \frac{|\mathsf{Ngh}_X(v)|}{|X|}\right)^k.$$

Note that, to perform this experiment for a vertex v, we need to make a single independence query of the form whether $(S \setminus R) \cup \{v\} \in \mathcal{I}_2$. Next, we make the following claim.

Claim 5.2. *There is a non-negative integer k such that the following holds:*

1. *If $|\mathsf{Ngh}_X(v)| < h$, then Experiment 5.1 succeeds with probability at least 3/4, and*
2. *If $|\mathsf{Ngh}_X(v)| > 10h$, then Experiment 5.1 succeeds with probability at most 1/4.*

Before proving Claim 5.2, we show the rest of the steps of this procedure. For every vertex, we repeat Experiment 5.1 $s = O(\log n)$ many times independently. By standard concentration bound, we make the following observations:

1. If $|\mathsf{Ngh}_X(v)| < h$, strictly more than $s/2$ experiments succeed with very high probability.[6]
2. If $|\mathsf{Ngh}_X(v)| > 10h$, strictly less than $s/2$ experiments succeed with very high probability.

Hence, we declare any vertex for which strictly less than $s/2$ experiments succeed as *heavy*. The probability that a light vertex can be classified as heavy by this procedure is very small due to Property 1. On the other hand, a vertex with $|\mathsf{Ngh}_X(v)| > 10h$ will be correctly classified as heavy with a very high probability. However, a heavy vertex with $|\mathsf{Ngh}_X(v)| \leq 10h$ may not be correctly classified. So, for such vertices, we want to check in a brute-force manner. To this end, we discover the set $\mathsf{Ngh}_X(v)$ for any vertex v which is not declared heavy and make decisions accordingly.

[6] Recall that by *very high probability* we mean with probability at least $1 - n^{-c}$ for some arbitrary large constant c.

Bounding the error probability. We argue that we can bound the error probabilities from Properties 1 and 2 over the whole run of the AUGMENTATION algorithm by a union bound. Say that the error probabilities of Properties 1 and 2 is bounded by n^{-c} for some large constant $c \geq 10$. In each phase we categorize at most n vertices, and there is at most $\frac{r}{h} < n$ phases. Hence, the probability that — over the whole run of AUGMENTATION (Algorithm 3.8) — that any vertex is misclassified as heavy, or that the procedure decides to discover a set $\mathsf{Ngh}_X(v)$ with $\mathsf{Ngh}_X(v) > 10$, is at most n^{-c+2}. Similarly we note that in the algorithm for Matroid Intersection (Algorithm 4.2) we run AUGMENTATION at most r times, so the error probability is at most n^{-c+3}.

Cost analysis. As mentioned before, each instance of Experiment 5.1 can be performed with a single query. As there are $O(n \log n)$ experiments in total in each phase of the algorithm, the number of queries needed to perform all experiments over the whole run of the AUGMENTATION algorithm will be is $O(\frac{nr \log n}{h})$ (recall that the number of phases is r/h). Now consider the part of the algorithm where we need to discover the set $\mathsf{Ngh}_X(v)$ for any vertex v which is not declared heavy after the completion of all experiments in a phase. For each such vertex, this will take at most $O(|\mathsf{Ngh}_X(v)| \log n) = O(h \log n)$ queries (due to Claim 2.6). Note that we only need to make these kinds of queries from each vertex once over the whole run of the algorithm (as in future queries we already know all v's neighbors and can answer directly). Hence, the total number of such queries is at most $O(nh \log n)$ across all phases of the algorithm.

Proof of Claim 5.2. First we note that if $|X| \leq 10h$, case 2 is vacuously true, so we may pick $k = 0$ such that Experiment 5.1 always succeeds. So now assume that $|X| > 10h$ and let $x = \frac{h}{|X|} \in (0, \frac{1}{10})$. We want to show that there exists some positive integer k satisfying $(1-x)^k \geq \frac{3}{4}$ and $(1-10x)^k \leq \frac{1}{4}$. Pick $k = \left\lceil \frac{\log \frac{1}{4}}{\log(1-10x)} \right\rceil$. Then $k \geq \frac{\log \frac{1}{4}}{\log(1-10x)} > 0$, which means that $(1-10x)^k \leq \frac{1}{4}$. We also have that $k \leq \frac{\log \frac{1}{4}}{\log(1-10x)} + 1 < \frac{\log \frac{3}{4}}{\log(1-x)}$ (since $x \in (0, \frac{1}{10})$), which means that $(1-x)^k \geq \frac{3}{4}$. □

5.2 Deterministic categorization

In this section we prove the following lemma (restated from Section 3.2).

Lemma 3.5. *There exists a deterministic categorization procedure which uses $O(n\sqrt{rh} \log r)$ queries over the whole run of the AUGMENTATION algorithm.*

The main idea of the determinsitic categorization is the following: For each $v \in \bar{S} \setminus F_{\bar{S}}$, our deterministic categorization keeps track of a set $N_v \subseteq \mathsf{Ngh}_X(v)$ of h out-neighbors to v (if that many out-neighbors exist). Then we can either use N_v as a proof that v is heavy, or when we failed to find such a N_v we know that v is light.

In each phase, some of the vertices in N_v may be added to F (and thus removed from X). This may decrease the size of N_v. In this case we would like to find additional out-neighbors to add to N_v, until $|N_v| = h$, or determine that $|\mathsf{Ngh}_X(v)| < h$. One possible and immediate strategy would be to use Claim 2.6 to find a new out-neighbor of v in $O(\log n)$ independence queries. However, adding arbitrary neighbors from $\mathsf{Ngh}_X(v) \setminus N_v$ will be expensive: over the whole run of the algorithm potentially every vertex in S will be added to N_v at some point which will require $\tilde{O}(nr)$ many independence queries in total for all N_v's—this is far too expensive than what we can allow. Instead, we want to be device a better strategy to pick $u \in \mathsf{Ngh}_X(v) \setminus N_v$.

Determinisitc strategy. For $u \in X$ we will denote by the *weight* of u, or $w(u)$, the number of sets N_v which contain u. Note that these weights change over the run of the algorithm. Also, note

that the values $w(u)$ can be inferred from the sets N_v's which are known to the querier. Hence, we can assume that the querier knows the weights of elements in X. When u is moved from X to F, $w(u)$ new out-neighbors must be found, one for each $v \in \bar{S} \setminus F_{\bar{S}}$ for which the set N_v contained u.

This motivates the following strategy: *Whenever we need to find a new out-neighbors of v, we find $u \in \mathsf{Ngh}_X(v) \setminus N_v$ that minimizes $w(u)$.* To perform this strategy, we note that the binary-search idea from Claim 2.6 can be implemented to find a u which minimizes $w(u)$. Indeed, if $\{u_1, u_2, \dots u_{|X|}\} \subseteq X$ with $w(u_1) \leq w(u_2) \leq \dots \leq w(u_{|X|})$, the binary search can first ask if there is an edge to $\{u_1, \dots, u_{\lfloor |X|/2 \rfloor}\}$ with a single query. If this was the case we recurse on $\{u_1, u_2, \dots u_{\lfloor |X|/2 \rfloor}\}$, otherwise recurse on $\{u_{\lfloor |X|/2 \rfloor + 1}, \dots, u_{|X|}\}$. This will guarantee that a the u_i which minimizes $w(u_i)$ will be found.[7]

Cost Analysis. For each $v \in \bar{S}$ we will at most once determine that N_v cannot be extended, i.e. that $|\mathsf{Ngh}_X(v)| < h$. This will require $O(n)$ independence queries in total. The remaining cost we will amortize over the vertices in $V = S \cup \bar{S}$. Consider that we find some out-neighbor $u \in X$ to some vertex $v \in \bar{S}$, using the above strategy. This uses $O(\log r)$ independence queries. We will charge this cost to u if $w(u) \leq \frac{n\sqrt{h}}{\sqrt{r}}$, otherwise we will charge the cost to v. We make the following observations:

1. For $u \in S$, the total cost we charge to it at most $O(\frac{n\sqrt{h}}{\sqrt{r}} \log r)$.
2. For $v \in \bar{S}$, the total cost we charge to it is at most $O(\sqrt{rh} \log r)$.

Property 1 is easy to see, since we charge the cost $O(\log r)$ to it at most $O(\frac{n\sqrt{h}}{\sqrt{r}})$ times. To argue that Property 2 holds, let $u \in S$ be the first vertex which got added to N_v which had weight $w(u)$ strictly more than $\frac{n\sqrt{h}}{\sqrt{r}}$ (at the moment it was added to N_v). At this point in time, we know that for all remaining $u' \in \mathsf{Ngh}_X(v) \setminus N_v$, must have $w(u') \geq w(u) > \frac{n\sqrt{h}}{\sqrt{r}}$. Note that we can bound the total weight $\sum_{u \in X} w(u) = \sum_{v \in \bar{S} \setminus F_{\bar{S}}} |N_v| \leq nh$ at any point in time. Because of this upper bound, there can be at most $\frac{nh}{n\sqrt{h}/\sqrt{r}} = \sqrt{rh}$ such u'. Hence we can charge vertex v at most $\sqrt{rh}$ more times.

Since there are at most r vertices $u \in S$ and n vertices $v \in \bar{S}$, we conclude that the total cost (over all phases) for the deterministic categorization is $O(n\sqrt{rh} \log r)$. This proves Lemma 3.5.

6 Open Problems

A major open problem is to close the big gap between upper and lower bounds for the matroid intersection problem with independent and rank queries. A major step towards this goal is to prove an $n^{1+\Omega(1)}$ lower bound. It will already be extremely interesting to prove such a bound for deterministic algorithms. It is also interesting to prove a cn lower bound for randomized algorithms for some constant $c > 1$ (the existing lower bound [Har08] holds only for deterministic algorithms).

Another major open problem is to understand whether the rank query is more powerful than the independence query. Are the tight bounds the same under both query-models? Two important intermediate steps towards answering this question is to achieve an $\tilde{O}(n\sqrt{r})$-query exact algorithm and an $\tilde{O}(n/\mathrm{poly}(\epsilon))$-query $(1-\epsilon)$-approximation algorithm under independence queries (such bounds have already been achieved under rank queries [CLS+19]). We conjecture that the tight bounds are $\tilde{O}(n\sqrt{n})$ under both queries when $r = \Omega(n)$.

[7] We actually use the same strategy to initialize the sets N_v: We discover out-neighbors u in the increasing order of $w(u)$.

We believe that fully understanding the complexity of the reachability problem will be another major step towards understanding the matroid intersection problem. We conjecture that our $\tilde{O}(n\sqrt{n})$ bound is tight for $r = \Omega(n)$.

It is also very interesting to break the quadratic barrier for the weighted case. This barrier can be broken by a $(1-\epsilon)$-approximation algorithm by combining techniques from [CLS+19, CQ16][8], but not the exact one.

Related problems are those for minimizing submodular functions. Proving an $n^{1+\Omega(1)}$ lower bound or subquadratic upper bound for, e.g., finding the minimizer of a submodular function or the non-trivial minimizer of a symmetric submodular function. Many recent studies (e.g. [RSW18, GPRW20, LLSZ20, MN19]) have led to some non-trivial bounds. However, it is still open whether an $n^{1+\Omega(1)}$ lower bound or an $n^{2-\Omega(1)}$ upper bound exist even in the special cases of computing minimum st-cut and hypergraph mincut in the cut query model.

Acknowledgment

This project has received funding from the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreement No 715672. Jan van den Brand is partially supported by the Google PhD Fellowship Program. Danupon Nanongkai and Sagnik Mukhopadhyay are also partially supported by the Swedish Research Council (Reg. No. 2019-05622).

References

[AD71] Martin Aigner and Thomas A. Dowling. Matching theory for combinatorial geometries. *Transactions of the American Mathematical Society*, 158(1):231–245, 1971.

[CLS+19] Deeparnab Chakrabarty, Yin Tat Lee, Aaron Sidford, Sahil Singla, and Sam Chiu-wai Wong. Faster matroid intersection. In *FOCS*, pages 1146–1168. IEEE Computer Society, 2019.

[CQ16] Chandra Chekuri and Kent Quanrud. A fast approximation for maximum weight matroid intersection. In *SODA*, pages 445–457. SIAM, 2016.

[Cun86] William H. Cunningham. Improved bounds for matroid partition and intersection algorithms. *SIAM J. Comput.*, 15(4):948–957, 1986.

[Edm70] Jack Edmonds. Submodular functions, matroids, and certain polyhedra. In *Combinatorial structures and their applications*, pages 69–87. 1970.

[Edm79] Jack Edmonds. Matroid intersection. In *Annals of discrete Mathematics*, volume 4, pages 39–49. Elsevier, 1979.

[EDVJ68] Jack Edmonds, GB Dantzig, AF Veinott, and M Jünger. Matroid partition. *50 Years of Integer Programming 1958–2008*, page 199, 1968.

[GPRW20] Andrei Graur, Tristan Pollner, Vidhya Ramaswamy, and S. Matthew Weinberg. New query lower bounds for submodular function minimization. In *ITCS*, volume 151 of *LIPIcs*, pages 64:1–64:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[8] From a private communication with Kent Quanrud

[Har08] Nicholas J. A. Harvey. Matroid intersection, pointer chasing, and young's seminormal representation of S_n. In *SODA*, pages 542–549. SIAM, 2008.

[Law75] Eugene L. Lawler. Matroid intersection algorithms. *Math. Program.*, 9(1):31–56, 1975.

[LLSZ20] Troy Lee, Tongyang Li, Miklos Santha, and Shengyu Zhang. On the cut dimension of a graph. *CoRR*, abs/2011.05085, 2020.

[LSW15] Yin Tat Lee, Aaron Sidford, and Sam Chiu-wai Wong. A faster cutting plane method and its implications for combinatorial and convex optimization. In *FOCS*, pages 1049–1065. IEEE Computer Society, 2015.

[MN19] Sagnik Mukhopadhyay and Danupon Nanongkai. Weighted min-cut: Sequential, cut-query and streaming algorithms. *CoRR*, abs/1911.01651, 2019.

[Ngu19] Huy L. Nguyen. A note on cunningham's algorithm for matroid intersection. *CoRR*, abs/1904.04129, 2019.

[RSW18] Aviad Rubinstein, Tselil Schramm, and S. Matthew Weinberg. Computing exact minimum cuts without knowing the graph. In *Proceedings of the 9th ITCS*, pages 39:1–39:16, 2018.

[Sch03] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.