# SEARCHING FOR HYPERGRAPHS USING REINFORCEMENT LEARNING

PARSA SALIMI AND TAMON STEPHEN

Abstract. We investigate the use of deep reinforcement learning methods for finding optimal combinatorial structures. In particular, we extend the methods in [Wag21] to hypergraphs, aiming to find examples that challenge the Fredman and Khachiyan [FK96] duality checking algorithm. In doing this, we highlight some natural optimization problems on pairs of transversal hypergraphs, where we present some preliminary extremal results.

The reinforcement learning approach is challenging to implement effectively, but in the end does find a non-obvious example that exceeds a well-known example in a key characteristic.

## 1. Introduction

Machine learning is currently being applied in many fields, including combinatorial optimization. Recently, Wagner [Wag21] applied elementary reinforcement learning methods to search for combinatorial objects that refute conjectures in extremal combinatorics.

In this paper, our aim is to extend Wagner's methods to search for simple hypergraphs that have characteristics that challenge the Fredman-Khachiyan algorithm. In doing this, we highlight some natural optimization problems on pairs of transversal hypergraphs, where we present some preliminary extremal results.

We find some promise for Wagner's method in the context of hypergraphs – in particular, it finds examples that perform well by various measures and would be difficult to find by hand. We also see how some of the limitations Wagner observed in searching for graphs assert themselves in the context of hypergraphs.

### 1.1. Outline of this paper.

In Section 2, we give an introduction to the hypergraph search problem and the Fredman-Khachiyan algorithm. In Section 3, we fix notation and introduce the deep reinforcement learning concepts used in the rest of the paper. (*** reorganize) In Section 3.1, we formulate the problem of hypergraph search as a reinforcement learning problem. We explore the effectiveness of the Cross-Entropy method and several other RL algorithms on this problem, and conclude that the cross-entropy method is easier to use and more reliable, while more advanced algorithms can produce better results in exceptional cases.

## 2. Hypergraphs and Problems of Interest

In this section we review hypergraphs and describe some combinatorial and algorithmic problems of interest.

### 2.1. Hypergraphs.

**Definition 2.1** (Hypergraphs). *Let $V = \{x_1, \dots, x_n\}$ be a finite set. A* hypergraph *on $V$ is a pair $\mathcal{H} = (V, \mathcal{E})$ where $\mathcal{E}$ is a set of subsets of $V$. The elements $E_1, \dots E_m$ of $\mathcal{E}$ are called the* edges *of $\mathcal{H}$ and $x_1, \dots x_n$ are* vertices *of $\mathcal{H}$. We refer to a hypergraph by its edge set $\mathcal{E}$ when there is no ambiguity. The* size *of a hypergraph $\mathcal{H}$ is the number of its edges, and is denoted by $|\mathcal{H}|$.*

An edge $E$ of a hypergraph is *minimal* if it doesn't contain another edge.

**Definition 2.2.** *A hypergraph is* simple *(or* Sperner*) if no edge contains another. Such structures are also sometimes called* clutters.

*Date*: June 21, 2023.

Unless otherwise stated, all the hypergraphs considered in this paper are simple.

We define the following two operations on pairs of hypergraphs with a common vertex set $V$:

$$\mathcal{F} \wedge \mathcal{G} = (V, \mathcal{E}) \text{ where } \mathcal{E} = \{A \cup B \mid A \in \mathcal{E}_{\mathcal{F}},\ B \in \mathcal{E}_{\mathcal{G}}\}, \text{ and}$$
$$\mathcal{F} \vee \mathcal{G} = (V, \mathcal{E}) \text{ where } \mathcal{E} = \mathcal{E}_{\mathcal{F}} \cup \mathcal{E}_{\mathcal{G}}$$

We now define the notion of transversal sets and transversal hypergraphs.

**Definition 2.3.** *Let $\mathcal{H} = (V, \mathcal{E})$ be a hypergraph. A* transversal *on $\mathcal{H}$ is a set $S \subseteq V$ such that $S \cap E \neq \varnothing$ for all $E \in \mathcal{E}$. A transversal is* minimal *if its minimal with respect to inclusion. The set of all minimal transversals of $\mathcal{H}$ is a hypergraph on V. It is called the* transversal hypergraph *of $\mathcal{H}$, and is denoted by $Tr(\mathcal{H})$. For those who prefer the terminology of clutters, the transversal hypergraph is known as the* blocker*.*

This leads to the following problems:

**Problem 2.4** (Transversal generation)**.** *Given a hypergraph $\mathcal{H}$ as input, compute $Tr(\mathcal{H})$.*

**Problem 2.5** (Transversal decision)**.** *Given simple hypergraphs $\mathcal{F}$ and $\mathcal{G}$ as input, decide whether or not $\mathcal{F} = Tr(\mathcal{G})$.*

Given simple hypergraphs $\mathcal{F}$ and $\mathcal{G}$, it can be shown [Ber89] that $\mathcal{F} = Tr(\mathcal{G})$ if and only if $\mathcal{G} = Tr(\mathcal{F})$. From this it follows that for simple hypergraphs $Tr(Tr(\mathcal{F})) = \mathcal{F}$ and that Problem 2.5 is equivalent to deciding if $\mathcal{G} = Tr(\mathcal{F})$. So in a sense, the input hypergraphs of our problem are interchangeable.

Transversal generation cannot be solved in polynomial time in the size of $\mathcal{H}$ since there are families of hypergraphs $\mathcal{F}$ such that $|Tr(\mathcal{F})|$ grows exponentially with $|\mathcal{F}|$. For example, the family of hypergraphs $\mathcal{J}_n = \{\{1,2\},\{3,4\}\dots\{2n-1,2n\}\}$ has size $n$, while $Tr(\mathcal{J}_n)$ has size $2^n$. Therefore, for these problems we use the notion of *output-sensitive complexity*, where algorithm time is compared to the *total size* of the input and output, i.e. $|\mathcal{H}| + |Tr(\mathcal{H})|$.

Computing the transversal hypergraph of a simple hypergraph is an important problem in computer science, and many algorithms have been proposed to solve this problem, surveys include [EMG08; GV17]. Perhaps the simplest and most well known of these is the sequential method, also known as Berge's algorithm [Ber89]. However, there is a well known family of hypergraphs $\mathcal{H}_i$ such that Berge's algorithm produces intermediate hypergraphs that are exponential in the size of the input and output [Tak08]. Moreover, Berge's algorithm is not suitable for jointly generating a hypergraph and its transversal from an oracle.

## 2.2. **Fredman and Khachiyan's Algorithms.**

Fredman and Khachiyan outlined two algorithms in [FK96], which we will call `FK-A` and `FK-B` for solving problem 2.5 in quasi-polynomial time. The `FK-A` algorithm has running time bounded by $n^{O(\log^2 n)}$ where $n = |\mathcal{F}| + |\mathcal{G}|$. The `FK-B` algorithm has running time $n^{o(\log(n))}$. It is not entirely clear that these bounds are tight, although Hertel [Her04] shows a superpolynomial behaviour of this example on `FK-A`.

The `FK-A` algorithm works by divide-and-conquer, either resolving one of a few simple cases directly, or by splitting the problem into two subproblems and recursing.

2.2.1. *Base cases.* The algorithm returns `false` if any of the following preconditions fail to hold:

(1) For any $I \in \mathcal{E}(\mathcal{F})$ and any $J \in \mathcal{E}(\mathcal{G})$, we have $I \cap J \neq \varnothing$.
(2) We have $\max\{|I| \in \mathcal{F}\} \leq |\mathcal{E}(\mathcal{G})|$ and $\max\{|J| \in \mathcal{G}\} \leq |\mathcal{E}(\mathcal{F})|$ for the pair $(\mathcal{F}, \mathcal{G})$.
(3) The inequality $\sum_{I \in \mathcal{F}} 2^{-|I|} + \sum_{J \in \mathcal{G}} 2^{-|J|} \geq 1$ holds.

The algorithm also returns `true` or `false` as appropriate in the case where $|\mathcal{E}(\mathcal{F})| \cdot |\mathcal{E}(\mathcal{G})| \leq 1$.

2.2.2. *Recursion.* At each recursion step, a *pivot vertex* $x$ is chosen from the common vertex set of $\mathcal{F}$ and $\mathcal{G}$. The algorithm then runs recursively on the instances $(\mathcal{F}_1, \mathcal{G}_0 \vee \mathcal{G}_1)$ and $(\mathcal{G}_1, \mathcal{F}_0 \vee \mathcal{F}_1)$ where:

$$\mathcal{F}_1 = (V, \mathcal{E} \setminus \mathcal{E}_x) \text{ where } \mathcal{E}_x = \{E \in \mathcal{E} \mid x \in E\}, \text{ and,}$$
$$\mathcal{F}_0 = (V, \mathcal{E}^*_x) \text{ where } \mathcal{E}^*_x = \{E \setminus \{x\} \mid x \in \mathcal{E}_x\}.$$

2.3. **Frequent Vertices.** Given a hypergraph $\mathcal{H} = (V, \mathcal{E})$, we say that a vertex $x \in V$ *occurs with frequency* $\varepsilon$ in $\mathcal{H}$ if

$$\varepsilon = \varepsilon_x(\mathcal{H}) := \frac{\#\{E : x \in E, E \in \mathcal{E}\}}{|\mathcal{H}|}.$$

Furthermore, we say that a vertex occurs with *frequency* at least $\varepsilon$ in a pair $(\mathcal{F}, \mathcal{G})$ if it occurs with frequency at least $\varepsilon$ in either $\mathcal{F}$ or $\mathcal{G}$. The *frequency* of $x$ in $(\mathcal{F}, \mathcal{G})$ is the maximum of $\varepsilon_x(\mathcal{F})$ and $\varepsilon_x(\mathcal{G})$ and is denoted by $\varepsilon_x(\mathcal{F}, \mathcal{G})$. The *maximum frequency* in $(\mathcal{F}, \mathcal{G})$ is the maximum, over all vertices $x$, of the frequency of $x$ in $(\mathcal{F}, \mathcal{G})$, and is denoted by $\varepsilon(\mathcal{F}, \mathcal{G})$. Since the transversal $\mathcal{G}$ is determined by $\mathcal{F}$, we can also simply write $\varepsilon(\mathcal{F})$.

The `FK-A` algorithm uses a frequent variable for pivoting. The analysis of this algorithm depends on the fact that for a transversal pair of hypergraphs, i.e. $(\mathcal{F}, Tr(\mathcal{F}))$ for some simple hypergraph $\mathcal{F}$, we have $\varepsilon(\mathcal{F}, Tr(\mathcal{F})) \geq \frac{1}{\log(|\mathcal{F}|+|Tr(\mathcal{F})|)}$ [FK96].

2.4. **Problems for Transversal Hypergraph Pairs.** This leads us to some natural questions about transervsal hypergraph pairs on $n$ vertices.

(1) For a given $n$, which simple hypergraphs $\mathcal{F}$ minimize $\varepsilon(\mathcal{F}, Tr(\mathcal{F}))$?
(2) For a given $n$, which simple hypergraphs $\mathcal{F}$ minimize $log(|\mathcal{F}| + |Tr(\mathcal{F})|) \cdot \varepsilon(\mathcal{F}, Tr(\mathcal{F}))$?
(3) For a given $n$, which simple hypergraphs $\mathcal{F}$ maximize the size of the recursion tree in the `FK-A` algorithm?

2.4.1. *Minimizing the maximum frequency.* This problem can be solved for some values of $n$ and nearly solved for other values of $n$ using the the following observations and constructions. The key is to focus of $s(\mathcal{F})$, the size of the smallest edge in $\mathcal{F}$.

**Lemma 2.6.** *For a simple $\mathcal{F}$ with a non-empty edge on $n$ vertices, $\varepsilon(\mathcal{F}) \geq \frac{s(\mathcal{F})}{n}$.*

*Proof.* Each edge of $\mathcal{F}$ contains at least $s(\mathcal{F})$ vertices, giving a total of at least $s(\mathcal{F})|\mathcal{F}|$ incidences of vertices on edges. Thus the average frequency of vertex incidences on edges over all vertices and edges is at least $s(\mathcal{F})/n$, and at least one vertex must match or exceed this frequency, giving the result. □

**Lemma 2.7.** *For a simple $\mathcal{F}$ with a non-empty edge on $n$ variables, $\varepsilon(\mathcal{F}) \geq \frac{1}{s(\mathcal{F})}$.*

*Proof.* Look at an edge $e$ attaining $s(\mathcal{F})$. Every edge of $Tr(\mathcal{F})$ must intersect $e$, and therefore contains an element of $e$. Therefore the average frequency in $Tr(\mathcal{F})$ of all elements in $e$ is at least one. Since there are $s(\mathcal{F})$ such elements, some element of $e$ must have frequency at least $\frac{1}{s(\mathcal{F})}$ in $Tr(\mathcal{F})$. □

Combining these two lemmas, we get that $\varepsilon(\mathcal{F}) \geq \max\{\frac{1}{k}, \frac{k}{n}\}$ for some integer $k \in \{1, 2, \ldots, n\}$. This maximum will be attained either at $k = \lfloor\sqrt{n}\rfloor$ or $k = \lceil\sqrt{n}\rceil$ or both if $n = k^2$. We can define $\varepsilon_n$ to be the minimum value of $\varepsilon(\mathcal{F})$ on graphs with $n$ vertices, thus we have shown $\varepsilon_n \geq \min\left\{\frac{1}{\lfloor\sqrt{n}\rfloor}, \frac{\lceil\sqrt{n}\rceil}{n}\right\}$. The following simple construction shows that this bound is attained when $n = k^2$ for some integer $k$.

**Definition 2.8.** *A hypergraph $\mathcal{H} = (V, \mathcal{E})$ is called* r-uniform *if $|E| = r$ for all $E \in \mathcal{E}$.*

When $n = qr$, we can construct an $r$-uniform hypergraphs with $q$ disjoint edges on $n$ vertices. This graph is unique up to isomorphism, we denote it $\mathcal{U}_r^q$. Its transversal consists of the $r^q$ edges that intersect each edge of $\mathcal{U}_r^q$ exactly once. We can see that $\varepsilon(\mathcal{U}_r^q) = \max\{\frac{1}{q}, \frac{1}{r}\}$ as the frequency of any vertex is $\frac{1}{q}$ in $\mathcal{U}_r^q$ and $\frac{1}{r}$ in $Tr(\mathcal{U}_r^q)$.

In particular, when $n = k^2$ for some integer $k$, the hypergraph $\mathcal{U}_k^k$ shows that the bound $\varepsilon_n \geq \frac{1}{\sqrt{n}}$ is tight. If $n$ is not square, we can write $n = k^2 + \ell$ where $k = \lfloor\sqrt{n}\rfloor$ and note that by adding $\ell$ vertices to on edge of $\mathcal{U}_k^k$ we get a hypergraph on $n$ vertices where the maximum frequency of a variable is also $\frac{1}{k}$. We thus have

$$\min\left\{\frac{1}{\lfloor\sqrt{n}\rfloor}, \frac{\lceil\sqrt{n}\rceil}{n}\right\} \leq \varepsilon_n \leq \frac{1}{\lfloor\sqrt{n}\rfloor} \tag{1}$$

The minimum on the left is equal to $\frac{1}{\sqrt{n}}$ when $n$ is between $k^2$ and $k^2+k$ for $k = \lfloor\sqrt{n}\rfloor$. In particular, this gives $\varepsilon_n = \frac{1}{\lfloor\sqrt{n}\rfloor}$ for $n = 1, 2, 4, 5, 6, 9, 10, 11, 12$.

**Remark 2.9.** *We have $\varepsilon_3 = \frac{2}{3}$ and $\varepsilon_7 = \frac{3}{7}$ so the lower bound is tight and the upper bound is not for $n = 3$ and $n = 7$.*

*Proof.* For $n = 3$, an interesting case is the triangle, the complete 2-uniform graph on 3 vertices, commonly denoted $K_3$. This graph is its own transversal, with all variable frequencies $\frac{2}{3}$. It is the unique hypergraph on 3 vertices that generates a transversal pair with all vertcies having frequency less than 1. So we have $\varepsilon_3 = \frac{2}{3}$, and the lower bound is tight.

Similarly, for $n = 7$, the *Fanoplane*, the smallest finite projective plane, with 7 lines of 3 points each on 7 vertices is also its own transversal, and has all variable frequencies of $\frac{3}{7}$. Thus $\varepsilon_7 = \frac{3}{7}$, and again the lower bound is tight. These are very special well-known constructions that are not available in other dimensions. □

So in fact we have that $\varepsilon_n = \min\left\{\frac{1}{\lfloor\sqrt{n}\rfloor}, \frac{\lceil\sqrt{n}\rceil}{n}\right\}$ for each $1 \le n \le 12$ except perhaps $n = 8$, where we have $\frac{3}{8} \le \varepsilon_8 \le \frac{3}{7}$, with the upper bound from adding an isolated vertex to the fanoplane. We summarize this in a table:

| **n** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\varepsilon_n$ | 1 | 1 | $\frac{2}{3}$ | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{1}{2}$ | $\frac{3}{7}$ | $\in [\frac{3}{8}, \frac{3}{7}]$ | $\frac{1}{3}$ | $\frac{1}{3}$ | $\frac{1}{3}$ | $\frac{1}{3}$ |

**Remark 2.10.** *A natural modification is to exclude isolated vertices, we can define the analogous quantity $\epsilon_n$ over graphs without isolated vertices.*

We get immediately that $\varepsilon_n \le \epsilon_n$ so the $\min\left\{\frac{1}{\lfloor\sqrt{n}\rfloor}, \frac{\lceil\sqrt{n}\rceil}{n}\right\}$ lower bound for $\varepsilon_n$ applies to $\epsilon_n$. The hypergraph $U_k^k$ confirms that $\epsilon_{k^2} = \varepsilon_{k^2} = \frac{1}{k}$. Sorting through small cases, we see that $\mathcal{U}_1^1$, $\mathcal{U}_1^2$, $K_3$ and $U_2^2$ along with their duals are exactly the hypergraphs where $\epsilon_r$ and $\varepsilon_r$ are (jointly) attained for $r = 1, 2, 3, 4$.

The special constructions for $n = 3$ ($K_3$) and $n = 7$ (the fanoplane) do not contain isolated vertices, so these upper bounds also carry over. For $n = 8$ and upper bound of $\frac{1}{2}$ is available, but in general it is not clear that $\varepsilon_n = \epsilon_n$, or even that $\{\epsilon_n\}$ is monotone.

2.4.2. *Fredman and Khachiyan's function.* We now turn our attention to the function

$$\lambda(\mathcal{F}) := log(|\mathcal{F}| + |Tr(\mathcal{F})|) \cdot \varepsilon(\mathcal{F}, Tr(\mathcal{F})).$$

In this paper logarithms are taken base 2. This is the function used in Fredman and Khachiyan's analysis. While Fredman and Khachiyan's algorithm can use the most frequent variable for pivoting, knowing that it meets this guarantee, examples that approach the limits Fredman-Khachiyan's output-sensitive analysis will show a low frequency on a small number of clauses. By this measure, $\mathcal{U}_k^k$ does not do particularly well, as its dual is very large, with $k^k$ edges, so $\lambda(U_k^k) = \frac{1}{k}\log(k^k + k) > \log k = \frac{1}{2}\log n$, while the Fredman-Khachiyan analysis gives $\lambda(\mathcal{F}) \ge 1$.

We can see that $\lambda(\mathcal{F}) = 1$ is attained on graphs $\mathcal{F}$ that have a single edge containing a single vertex. Such graphs exist for each $n$, but consist mainly of isolated vertices. If we ignore graphs with isolated vertices, the question of finding graphs with, for instance $\lambda(\mathcal{F}) \le 2$ (the bound used in Fredman and Khachiyan's analysis), is an interesting one.

Gurvich and Khachiyan [GK97] found an infinite family of hypergraphs $\mathcal{GK}_r = (F_r, G_r)$ with $G_r = Tr(F_r)$ such that $\lambda(F_r) < 2$. This family is defined as follows:

- $F_1 = (V, \mathcal{E})$ with $V = \{1, 2\}$ and $\mathcal{E} = \{\{1\}, \{2\}\}$.
- $F_i = (\mathcal{F} \vee \mathcal{G}) \wedge (\mathcal{H} \vee \mathcal{I})$ where $\mathcal{F}, \mathcal{G}, \mathcal{H}$ and $\mathcal{I}$ are disjoint copies of $F_{i-1}$.

Similarly, define

- $G_1 = (V, \mathcal{E})$ with $V = \{1, 2\}$ and $\mathcal{E} = \{\{1, 2\}\}$
- $G_i = (\mathcal{F} \wedge \mathcal{G}) \vee (\mathcal{H} \wedge \mathcal{I})$ as before.

The $\mathcal{GK}_r$ family of examples is very sparse in the sense that they occur only on hypergraphs with $n = 2^{2^r - 1}$ vertices, for integer $r$. Following the analysis in Gurvich and Khachiyan's paper [GK97], we note that $F_r$ and $G_r$ have $2^{2^r-1}$ and $2^{2^r-2}$ edges respectively, with each variable appearing in a fraction of $2^{-r}$ and $2^{1-r}$ of the edges respectively. Thus $\lambda(\mathcal{GK}_k) = \log(3 \cdot 2^{2^r-2})2^{1-r} = 2 + 2^{1-r} \log \frac{3}{4}$, which is less than 2, but converges to 2 as $r$ increases.

Consider now $\lambda_n$, the minimum value of $\lambda(\mathcal{F})$ over hypergraphs on $n$ vertices without isolated vertices. Looking at small $n$, we have $\lambda_1 = \log(2) = 1$ which is achieved on $\mathcal{U}_1^1$. Similarly $\lambda_2 = \log(3)$ which is achieved on $\mathcal{U}_1^2$. On 3 variables, the minimum is achieved on the complete graph $K_3$, with $\lambda(K_3) = \frac{2}{3} \cdot \log(6)$. And on 4 variables, the minimum is achieved by the cycle $C_4$ with $\lambda(C_4) = \frac{1}{2} \cdot \log(6)$.

The hypergraphs identified in Section 2.4.1 provide additional examples with low values of $\lambda(\mathcal{F})$. The hypergraphs $\mathcal{U}^2_{\frac{n}{2}}$ formed by two disjoint edges of size $\frac{n}{2}$ (and, abusing notation slightly, $\frac{n+1}{2}$ and $\frac{n-1}{2}$ when $n$ is odd) with $n \geq 4$ have $\epsilon(\mathcal{F}) = \frac{1}{2}$. With $\frac{n^2}{4}$ or $\frac{n^2-1}{4}$ edges in their transversals, we have $\lambda(\mathcal{U}^2_{\frac{n}{2}}) = \frac{1}{2}\log(\frac{n^2+8}{4})$ for even $n$ and $\lambda(\mathcal{U}^2_{\frac{n}{2}}) = \frac{1}{2}\log(\frac{n^2+7}{4})$ for odd $n$. For larger $n$, these hypergraphs are not exceptional in terms of $\lambda$, but their $\lambda$ value does not exceed 2 until $n = 8$.

The self-dual fanoplane has $\lambda(\mathcal{F}) = \frac{3}{7}\log(14) \approx 1.63$.

**Example 2.11.** *Consider the following "double star" function on 8 variables and its transversal:*

$$\begin{aligned}\textit{2-star} &= \{\{1,2\},\{1,3\},\{1,4\},\{5,6\},\{5,7\},\{5,8\}\},\\ Tr(\textit{2-star}) &= \{\{1,5\},\{1,6,7,8\},\{2,3,4,5\},\{2,3,4,6,7,8\}\}.\end{aligned}$$

*Note that $\varepsilon_x(\textit{2-star}, Tr(\textit{2-star})) = \frac{1}{2}$ for all vertices $x$ and thus*

$$\varepsilon(\textit{2-star}, Tr(\textit{2-star})) = \frac{1}{2} = \varepsilon(F_2, G_2).$$

*Since 2-star and its dual combine for only edges, this construction actually exceeds the $(\mathcal{F}_2, \mathcal{G}_2)$ in the Fredman-Khachiyan measure, with $\frac{\log 10}{2} = \lambda(\textit{2-star}) < \lambda(\mathcal{F}_2, \mathcal{G}_2) = \frac{\log 12}{2}$. On the other hand, this isn't a particularly difficult case for the Fredman-Khachiyan algorithm.*

2.4.3. *Tree size.* In the previous section, the bound of $\frac{1}{\log(|\mathcal{F}|+|\mathcal{G}|)}$ is tight up to a factor of 2. However, it is not clear how the frequency at the top level, i.e. at the level of the original functions $\mathcal{F}$ and $\mathcal{G}$ influences the behaviour of Fredman and Khachiyan's at the lower levels of the recursion. It does appear that the examples of Gurvich and Khachiyan are difficult for FK-A in the sense that they generate a large recursion tree. Hertel [Her04] confirms that the size of these trees on these examples is superpolynomial.

Consider now a 4-uniform hypergraph with 2 disjoint edges, $\mathcal{U}_4^2$. This hypergraph on 8 vertices resembles $(\mathcal{F}_2, \mathcal{G}_2)$ in the parameters we have considered so far, matching it in frequency with $\varepsilon(\mathcal{U}_4^2) = \frac{1}{2}$ and slightly exceeding it on Fredman and Khachiyan's function with $\frac{\log 18}{2} = \lambda(\mathcal{U}_4^2) > \lambda(\mathcal{F}_2, \mathcal{G}_2) = \frac{\log 12}{2}$. However, due to the small number of edges in $\mathcal{U}_4^2$, the tree generated when checking duality by Fredman and Khachiyan's algorithm is fairly small.

In this case, the effort required to simply evaluate the objective function (tree size) is very significant, and makes it a natural target for computer searches. A complication is that the target is not as cleanly defined as with the other problems: for instance it depends on arbitrary choices and tie-breaking rules which are left to the implementer in the original paper of Fredman and Khachiyan, and are not yet standardized.

## 3. Reinforcement learning for combinatorial search

The functions considered in Section 2, namely $\varepsilon$, $\epsilon$, $\lambda$ and tree size, are highly non-linear in the input hypergraph $\mathcal{F}$, and it is not clear how to approach optimizing them. Thus we investigate using machine (reinforcement) learning to search for challenging examples.

In this section we describe how we can formulate searching for hypergraphs as a reinforcement learning problem. We first formulate the simpler problem of searching for graphs. We then extend this formulation for hypergraphs, and discuss several potential issues. We end by discussing an alternative formulation as a direction for future work.

## 3.1. **Definitions.**

*Reinforcement learning* formulates problem solving as an interaction between an *agent* and the *environment*. For our purposes, the environment is a representation for the problem to be solved. The learning agent interacts with the environment by taking *actions*. After taking an action, the agent receives two pieces of information from the environment: A real-valued *reward* that is specific to the decision making problem, and the new *state* that the agent is in after taking the action. The agent has no prior knowledge of the problem, and must learn to take good actions based only on the reward signals received from the environment.

The agent does maintain an internal structure, such as a neural network [Probably this is a place for references], that takes the current state as input and suggestions actions. *Deep reinforcement learning* refers to using deep neural networks, i.e. those with many layers.

The learning takes place via a series of *episodes* where the agent interacts with the environment and modifies its internal structure based on the rewards seen during the episode. For now we are treating this as a black box.

## 3.2. **Reinforcement Learning for Graphs.**

The following setup is used by Wagner [KB14] in applying reinforcement learning to graph problems. Let $\mathcal{G}_n$ be the set of graphs on $n$ vertices, and let $f$ be the function that we want to optimize.

We define $s$ as follows. Let $\mathcal{A} = \{0, 1\}$, $k = n(n-1)/2$ and $E = (e_1, ..., e_k)$ be an (arbitrary) ordering of the edges of $K_n$. Then for a given graph $G$, define $s(G) = a_1, a_2, \ldots, a_k$ where $a_i = 1$ if and only if $e_i \in G$. That is, the $i$th action decides if edge $e_i$ is included in $G$. In this way, a graph is generated with a sequence of $k$ actions. It is easy to see that $s$ is a bijection.

The optimal action at each time step should only depend on the current state, and not otherwise on the history of actions taken so far. Intuitively, at time step $i$, we have a partial graph constructed by making $i-1$ binary decisions, and we are deciding whether or not to include the edge $e_i$ in our graph. We define the reward to be 0 on all but the terminal timestep, at which point we have constructed a graph $G$ and the reward is simply $f(G)$, the objective function applied to $G$. The transition function records our chosen action in the state, and advances the timestep.

## 3.3. **Reinforcement Learning for Hypergraphs.**

We can extend the ideas in the previous section to hypergraphs by fixing an order and walking through potential hyperedges.

Let $\mathcal{H}_n$ be the set of simple hypergraphs on $n$ vertices and $f$ a reward function. Let $\mathcal{A} = \{0, 1\}$, $k = 2^n$ and $E = (e_1, \ldots, e_k)$ be an ordering of the elements of $2^{[n]}$. As in Section 3.2, the hyperedges are ordered arbitrarily, the states encode the set of edges selected so far and the hyperedge currently under consideration, the transition functions add the current edge or not and then advance the edge under consideration, and the reward function is zero until all edges have been considered, and then the reward of the graph at the end.

Since we are searching for simple hypergraphs, we finish with a simple post-processing step of removing all hyperedges that are supersets of another edge via pairwise comparisons.

**Remark 3.1.** *Machine learning has been used extensively to help solve combinatorial optimization problems, often producing heuristic solutions, see for example [BLP21; Bel+17; Maz+20]. These problems aim to generate an optimal output conditioned on some input, and many of the innovations in this area are about how to do this. They borrow ideas from language translation using neural networks, and use encoder-decoder architectures for this. This does not immediately translate to our context, but see for example [Che+21] for some ideas in that direction.* [What is the essential difference between what they are doing and what we are doing?]

[This also seems very interesting: [Faw+22], it makes progress through machine learning, leading to additional insights [KM22].]

3.4. **Specific objectives.** We now consider a few specific targets for this approach. We will aim to optimize over simple hypergraphs with a fixed number $n$ of vertices over various values of the parameter $n$, and consider the following reward functions:

(1) $f_1(\mathcal{H}) = |\mathcal{H}|$ the size of $\mathcal{H}$. This is a basic problem that illustrates some of the challenges of extending the learning framework to hypergraphs. Maximizing $f_1$ is equivalent to finding a largest antichain in a Boolean lattice $2^{[n]}$ where $n = |V(\mathcal{H})|$. This is actually the first example mentioned by Wagner [KB14], although it is unclear if that he actually attempted to solve it. [If he did, it seems like he would have used a different approach, since he mentions an objective of number of clauses − number of nested pairs.] ts

By Sperner's well known theorem, the maximum value of $f_1$ is $\binom{n}{\lfloor n/2 \rfloor}$ and is obtained by letting $\mathcal{E} = \{E \in P(V) \mid |E| = \lfloor n/2 \rfloor\}$. This function thus provides a good starting point to test our approach, as the optimal solutions are well characterized.

(2) $f_2(\mathcal{H}) = -\varepsilon(\mathcal{H}, Tr(\mathcal{H}))$. Maximizing $f$ is equivalent to finding a hypergraph $\mathcal{H}$ with smallest maximum frequency in $(\mathcal{H}, Tr(\mathcal{H}))$.

(3) $f_3(\mathcal{H}) = -\varepsilon(\mathcal{H}, Tr(\mathcal{H})) \cdot \log(|\mathcal{H}| + |Tr(\mathcal{H})|)$. This performance measure computes the ratio of the maximum frequency, with the lower bound $\frac{1}{\log(|\mathcal{H}|+|Tr(\mathcal{H})|)}$ discussed earlier. In this way, this function also takes into account the total size. That is, among the functions with low maximum frequency, it favours those with smaller total size.

(4) For now, we will not pursue the tree size function introduced in Section [**treesize**], in favour of working with cleaner combinatorial examples.

(5) [A bit of a side question, but I'm curious if there are any non-trivial transversal pairs that actually approach the $\sum_{\mathcal{F},\mathcal{G}} 2^{|I|}$ bound of 1? If not, could we push FK's anaylsis a bit?] ts

(6) [If we are looking through general hypergraphs, then intersecting families are interesting.] ts

**Remark 3.2.** [Further to the last point, there are actually a lot of natural questions about joint properties of a hypergraph and its dual. Is there any systematic study of this? We should check. ts

For some of the examples that we see here, either $\mathcal{F}$ or $Tr(\mathcal{F})$ disconnected, with the RL find a notable exception. If we require this as well, then we lose some of our constructive lower bounds. ]

3.5. **Alternative formulations for learning hypergraphs.** The definition of our state space in 3.2 was somewhat counter intuitive, as the feedforward architecture we have considered so far restricts us to fixed length inputs [I still do think that the formulation is odd, because it encodes positional arguments as a seperate sparse input. But it turned out to work better than the RNN input structure. ] ps Moreover, this formulation incorporates no prior information about the structure of hypergraphs, and the sequential nature of decision making. Indeed, the basic formulation is general enough to be used in searching for binary strings of length $2^n$, so long as the fitness function is well defined and easy to compute as a function of binary strings. This is undesirable in that searching for binary strings appears to be a harder problem than searching for simple hypergraphs. [These are all valid criticisms of our approach. Below are two attempts at solving them. The first one was not successful, we didn't implement the second one. A disadvantage of the second attempt is that it's pretty hard to implement. Both of these are interesting alternatives in my opinion, so maybe we should keep them] ps In this section, we describe alternative formulations and modifications, which are somewhat inspired by the methods in section **??**. We have only implemented the alternative proposed in 3.5.1.

3.5.1. *Variable length state spaces with RNNs.* Recurrent neural networks(RNNs) and their variants can easily handle variable length inputs, and have traditionally been used for sequential decision making problems, such as time-series prediction. Since generating hypergraphs is also a sequential problem, RNNs seem to be well suited for our problem. Additionally, RNNs have been used in solving combinatorial optimization problems, as outlined in section **??**. By allowing variable length states, we can simplify our formulation of the state space, by letting each state simply be the variable length binary string of the edges we have considered so far. This eliminates the need to pad the state with 0s and one-hot encode the edge under consideration.

Additionally, feedforward networks have a fixed width, so for hypergraphs with large vertices, feedforward networks may not be able to capture long-term dependencies between the edges. Therefore it seems reasonable to hypothesize that RNNs generalise better. We test this hypothesis in section 4

3.5.2. *Choosing the next edge without a fixed ordering.* In section 3.3, we added edges to the hypergraph one by one, from a predetermined ordering of the edges. Let's refer to this method as *static generation.*

There are some problems with static generation. Generating a hypergraph with $n$ vertices takes $2^n$ steps. Moreover, this approach doesn't capture the complex relationship and the dependencies that the edges have with each other. Finally, the probability of including edge $e_i$ is conditioned only on the previous edges, and while this is inevitable, it is unnatural to condition the inclusion of an edge on a a-priori fixed set of previous edges.

For this reason, and motivated by the TSP formulation in section **??**, we propose to consider the next edge to be included among all available edges. That is, at time step $t$, instead of deciding to include a fixed edge $e_t$, we decide the next edge to include, among all the remaining edges. As in **??**, we can do this either by computing a value function for each edge, conditioned on the current partial tour, or by outputting a probability distribution over all the remaining edges.

An advantage of this approach is that hypergraphs can be constructed with a much smaller number of actions. In static generation, we need $2^n - 1$ actions to generate any hypergraph, whereas in the proposed formulation, we generate a hypergraph $\mathcal{F}$ in roughly $|\mathcal{F}|$ time steps. This can be beneficial as the reward signals are less sparse.

However, notice that for hypergraphs, the number of edges to consider grows exponentially as the number of vertices grows, and so the action space is exponentially large. This might make learning more difficult. In section **??**, we modify this approach by considering only a small subset of the possible edges at each time step.

Furthermore, unlike the TSP, there is no natural stopping criterion. We can add edges for as long as we like, but we want to stop once we have a good construction. This can be fixed by adding a special action at each state, which ends the episode and returns the hypergraph constructed so far. For these reasons, we decided to forgo implementing this method.

**Remark 3.3.** *An alternative to our approach is to cast the problem as sequential decision-making process, where at each step we are allowed to make small moves (in the sense of Hamming distance) in either $\mathcal{F}$ or $Tr(\mathcal{F})$.* [Is the idea to add edges, or modify existing ones? The Hamming distance seems to suggest the latter. Also, how do we avoid repeatedly considering the same edge?]

3.6. **Formulating an RL task.** We describe the components that define an RL task. At each time step, the agent can be in one of many *states*. The set of all states is denoted by $\mathcal{S}$. At time step $t$, the agent receives some $S_t \in \mathcal{S}$, and it selects an *action* $A_t \in \mathcal{A}$, where $\mathcal{A}$ is the set of possible actions. The agent then receives a *reward* $R_t \in \mathcal{R}$ where $\mathcal{R} \subseteq \mathbb{R}$ is the set of possible rewards, and a new state $S_{t+1}$. This process gets repeated until the agent enters a *terminal state* at time step $T$, where the episode ends.

Given a state $s' \in S$ and reward $r \in \mathcal{R}$, we have the well defined *state transition function* $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, which returns the next state, given the previous state and the action taken at that state. The environment dynamics are completely characterised by this function. That is, the next state $S_t$ depends only on the immediately preceding state and action, and not on the previous states and actions. Similarly, the reward function $r : \mathcal{S} \times \mathcal{A} \to \mathcal{R}$ returns a reward corresponding to each state-action pair.

3.7. **Solving an RL task.** A *policy* $\pi$ is a solution to a given RL problem. For each state $s$, the policy $\pi : \mathcal{S} \to \mathcal{A}$ recommends the next action to be taken. Our goal in Reinforcement Learning is to develop a good policy, so we need a way to compare different policies.

Given a sequence of rewards $R_{t+1}, R_{t+2}, \dots, R_T$, we denote their sum by $G_t$ and call this the *return* at time step $t$. We define the *return* $G_t$ at time step $t$ to be the sum of rewards $R_t + R_{t+1} + \cdots + R_T$.

Given a policy $\pi$, The *value function* $v_\pi(s)$ is the expected return when starting in state (s) and following $\pi$:

$$v_\pi(s) \coloneqq \mathbb{E}_\pi[G_t \mid S_t = s] \,.$$

Similarly we have the *action-value function*

$$q_\pi(s,a) \coloneqq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

which is the expected value of taking action $a$ from state $s$ and following $\pi$ thereafter.

We can now compare policies: Given two policies $\pi$ and $\pi'$, we define a partial order and say $\pi \geq \pi'$ to mean that $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. A policy $\pi_*$ is *optimal* if $\pi_* \geq \pi$ for all policies $\pi$.

All optimal policies share the unique optimal value function:

$$v_*(s) \coloneqq \max_\pi v_\pi(s).$$

Similarly we define:

$$q_*(s,a) \coloneqq \max_\pi q_\pi(s,a) = \mathbb{E}\left[R_{t+1} + v_*(S_{t+1}) \mid A_t = a, S_t = s\right].$$

This last equation is known as the *Bellman optimality equation.*

The goal of reinforcement learning then is to find policies that are optimal or near optimal. This can be done by either learning a policy function directly, or by estimating the value function instead, and then differentiate between actions based on their estimated value functions.

Producing an optimal solution to an RL task will require learning a policy or value function, but in practice the state space $\mathcal{S}$ is too large for such a function to fit in memory. Solving RL in these cases calls for using function approximation techniques, such as *Deep Neural Networks*(DNNs). Deep learning (DL) methods have been used to effectively approximate solutions to RL problems [Mni+13a], and more recently, this method has been applied to find heuristics for combinatorial optimization problems [KHW19; Kha+17; BLP21]. In the next section we formulate our combinatorial search problem as an RL task and attempt to use DL methods to solve it.

3.8. **The Cross entropy method.** The Cross Entropy method is the algorithm used in [Wag21] to generate combinatorial examples. It is rather different from the other common reinforcement learning algorithms, and so we briefly explain it here.

Let $F : \mathcal{S} \to P_\mathcal{A}$ be a neural network representing a stochastic policy. That is, for each state, $N$ produces a probability distribution over the actions. The agent then samples this probability distribution to generate actions. Fix a large positive integer $N$, and generate $N$ episodes $(e_1,\ldots,e_n)$ using $F$. For each episode $e_i$, record the cumulative reward $R_i$. Now, if $F$ is initialized properly, there is a high variability in the trajectory of the $N$ episodes taken by the agent

The idea of the cross entropy method is this: we take the top $p$ percent of the episodes (the *elite* episodes) and treat their corresponding trajectories as ideal. More precisely, let $(S_ij, a_i)$ be the state-action pair for the $j$th action in the $i$th elite episode. Note that this is also an input-output pair for the neural network. For each pair, we update the weights of the network using stochastic gradient descent (SGD), using the binary cross-entropy loss as our loss function. In a similar way, update the weights of the neural network for all the actions taken during each episode. This finishes one iteration of the algorithm. In the next iteration, we again generate $N$ episodes, this time using our updated weights, and repeat the process. This method is presented in more detail in [Wag21].

## 4. Computational results

In this section we gather our computational results with reinforcement learning. We test the following algorithms:

(1) `ES-FFN` : Evolutionary Strategies (ES) with feedforward networks. this is a simple modification to Wagner's approach [Wag21].
(2) `ES-RNN` : Evolutionary Strategies with RNN : modifies Wagner's approach slightly to use RNNs as explained in section 3.5.1
(3) `DQN` : Deep Q-learning as described in [Mni+13b]

(4) `PPO` : proximal policy optimization [Sch+17]

The DQN and PPO algorithms are close to state of the art in their respective domains. DQN is learns the Q-functions, while PPO is an actor-critic algorithm that optimizes the policies directly.

We borrow the ES algorithm from [Wag21] with slight modifications to accomodate for hypergraphs. We make further modification to this code for the RNN version. The DQN and PPO algorithms are implemented using PyTorch and the Stable Baselines library [Hil+18]. For these algorithms, we kept the default hyperparameters provided by the library. Since we are mainly interested in the best construction found, these are the only results that we record here. That is, for each algorithm, we report the best results over multiple iterations. We also describe the best hypergraphs found overall.

We test these algorithms against the first three reward functions described in section 3.4.

## 4.1. Best hypergraphs found ($\varepsilon$ objective).

4.1.1. *5 variables.* The following hypergraphs achieve $\epsilon = \frac{1}{2}$

$$F = [(2,3,5),(1,4)], G = [(2,4),(1,2),(3,4),(1,5),(4,5),(1,3)]$$

$$F = [(2,3,5),(4,5),(1,2),(1,3,4)] G = [(1,3,4),(2,4),(2,3,5),(1,5)]$$

4.1.2. *6 variables.* The following hypergraphs that were found achieve $\epsilon = \frac{1}{2}$

$$F = [(2,4,6),(2,5),(1,4),(1,3,5)], G = [(1,5,6),(4,5),(1,2),(2,3,4)]$$

$$F = [(2,5),(1,4),(3,6)], G = [(1,3,5),(1,2,6),(1,2,3),(1,5,6),(3,4,5),(2,3,4),(4,5,6),(2,4,6)]$$

$$F = [(3,5),(1,2,4,6)], G = [(3,4),(1,5),(2,3),(4,5),(5,6),(3,6),(2,5),(1,3)]$$

$$F = [(2,3),(2,5),(1,4),(4,6)], G = [(1,2,6),(2,4),(3,4,5),(1,3,5,6)]$$

4.1.3. *7 Variables.* The following gives $\varepsilon = \frac{5}{11}$. Note that this is not optimal, since we found that $\varepsilon_7 = \frac{3}{7}$. However, this is the best example that was found and usually the algorithm will get stuck at $\frac{1}{2}$.

$$F = [(3,6,7),(3,5,6),(1,2,3),(2,5,7),(4,6,7),(1,4,5),(1,2,4)]$$
$$G = [(1,2,6),(1,5,7),(3,4,7),(2,5,7),(1,5,6),(1,3,7),(3,4,5),(2,3,4),(2,4,6),(1,6,7),(2,5,6)]$$

4.1.4. *8 variables.* Using just the original approach in [Wag21], we have found a hypergraph on 8 variables that beats the $\mathcal{GK}_2$ minimum frequency of $\frac{1}{2}$. It is the hypergraph:

$$\mathcal{J} = \{\{2,6,7,8\},\{3,7,8\},\{1,4,5\},\{2,5,6\},\{1,3,4\}\}.$$

Note that we have

$$Tr(\mathcal{J}) = \{\{3,4,6\},\{2,3,5\},\{8,1,2\},\{8,3,5\},\{4,5,7\},\{2,4,7\},\{1,5,7\},$$
$$\{1,3,6\},\{8,4,5\},\{1,2,7\},\{3,5,7\},\{1,2,3\},\{1,6,7\},\{8,1,6\},\{8,4,6\},$$
$$\{3,5,6\},\{8,2,4\},\{8,1,5\},\{2,3,4\},\{4,6,7\}\}.$$

The frequencies are $\varepsilon_i(\mathcal{J}, Tr(\mathcal{J})) = 2/5$ for all variables $i$ in both the primal and dual. Therefore, we have $\varepsilon(\mathcal{J}, Tr(\mathcal{J})) = 2/5$. Note that this is lower than $\varepsilon\mathcal{GK}_2$.

We were surprised to find this irregular hypergraph that surpasses the $\mathcal{GK}_(2)$ example. This encouraged us to search for also for more regular examples by hand, which led us to considering the fanoplane on 7 vertices, which has an even lower $\varepsilon$ value ($\frac{3}{7}$). Modifying the fanoplane by twinning one vertex gives a graph on 8 vertices with 7 edges; the dual of this has 10 edges. This pair also attains a $\varepsilon$ value of $\frac{2}{5}$.

It seems quite possible then, that $\varepsilon_8 = \frac{3}{7}$ (by adding an isolated vertex to the fanoplane), but $\epsilon_8 = \frac{2}{5}$. This would give separation between $\varepsilon$ and $\epsilon$, while also showing that the latter is not monotone.

## 4.2. Best hypergraphs found ($\lambda$ objective).

4.2.1. *4 variables.* On 4 variables the algorithm quickly finds the $C_4$ example that we showed to be optimal in section 2.4.2

4.2.2. *5 variables.* The ES algorithm converges to the following hypergraphs, which achieves $\lambda = \log(8) \cdot \frac{1}{2}$

$$F = [(2,3,4),(1,5)], G = [(1,2),(1,4),(4,5),(2,5),(1,3),(3,5)]$$

4.2.3. *6 variables.* The ES algorithm converges to the following hypergraph pair with $\lambda = \log(8) \cdot \frac{1}{2}$

$$F = [(2,5),(3,4),(1,4),(5,6)], G = [(4,5),(2,4,6),(1,2,3,6),(1,3,5)]$$

4.2.4. *7 variables.* The following pair gives $\lambda = \log(10) \cdot \frac{1}{2}$

$$F = [(1,4),(1,7),(2,6),(3,6),(2,5),(3,5)], G = [(1,5,6),(4,5,6,7),(1,2,3),(2,3,4,7)]$$

4.2.5. *8 variables.* The ES algorithm struggled to learn a useful hypergraph. In our experiments, the training process did not produce better hypergraphs than those produced by chance from the initial parameters.

4.3. **Comparison of RL algorithms.** The optimal results produced above were obtained using the original ES algorithm with fully-connected networks. Once configured correctly, the PPO and DQN algorithms were also capable of reproducing most of these results [check all of them...]. We did not find any noticeable improvements by using ES with an RNN instead of a feedforward network. RNNs are also slower to train than FNNs, so we do not recommend using them. ps

For our use case, the ES algorithm is simpler to use compared to PPO and DQN. This is because the ES algorithm explicitly tries to produce good trajectories, and for this reason it stores the best trajectories for purposes of training. Since our goal is not merely to find optimal values for the score function, but also to report the hypergraphs that actually achieved this optimum, the optimal trajectories are important objects to us. The ES algorithm is trained to find these optimal trajectories, so it is easy to recover this information directly from the algorithm. In contrast, most implementations of DQN, PPO and other famous RL algorithms don't use trajectories in the same way, and so extracting this information from the trained agent is more difficult.

## 5. Acknowledgements

The authors thank Saimon Islam for discussion, and in particular for uncovering the result of Hertel [Her04]. The authors were supported by an NSERC grant to TS.

## References

[Bel+17] Irwan Bello et al. *Neural Combinatorial Optimization with Reinforcement Learning.* 2017. arXiv: `1611.09940 [cs.AI]`.

[Ber89] Claude Berge. *Hypergraphs: Combinatorics of Finite Sets.* North Holland, 1989.

[BLP21] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. "Machine learning for combinatorial optimization: A methodological tour d'horizon". In: *European Journal of Operational Research* 290.2 (2021), pp. 405–421. DOI: `10.1016/j.ejor.2020.07.063`.

[Che+21] Lili Chen et al. *Decision Transformer: Reinforcement Learning via Sequence Modeling.* 2021.

[EMG08] Thomas Eiter, Kazuhisa Makino, and Georg Gottlob. "Computational aspects of monotone dualization: A brief survey". In: *Discrete Applied Mathematics* 156.11 (2008), pp. 2035–2049. DOI: `10.1016/j.dam.2007.04.017`.

[Faw+22] Alhussein Fawzi et al. "Discovering Faster Matrix Multiplication Algorithms with Reinforcement Learning." In: *Nature* 610 (2022), pp. 47–53.

[FK96] Michael L. Fredman and Leonid Khachiyan. "On the Complexity of Dualization of Monotone Disjunctive Normal Forms". In: *Journal of Algorithms* 21.3 (1996), pp. 618–628. DOI: 10.1006/jagm.1996.0062.

[GK97] Vladimir Gurvich and Leonid Khachiyan. "On the frequency of the most frequently occurring variable in dual monotone DNFs". In: *Discrete Mathematics* 169.1-3 (1997), pp. 245–248. DOI: 10.1016/s0012-365x(96)00090-8.

[GV17] Andrew Gainer-Dewar and Paola Vera-Licona. "The Minimal Hitting Set Generation Problem: Algorithms and Computation". In: *SIAM Journal on Discrete Mathematics* 31.1 (2017), pp. 63–100. DOI: 10.1137/15m1055024.

[Her04] Philipp Hertel. "Some Algorithmic and Complexity Results for Monotone Boolean Duality (Hypergraph Transversal)". MA thesis. University of Toronto, 2004.

[Hil+18] Ashley Hill et al. *Stable Baselines.* https://github.com/hill-a/stable-baselines. 2018.

[KB14] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. 2014. URL: http://arxiv.org/abs/1412.6980.

[Kha+17] Elias Khalil et al. "Learning Combinatorial Optimization Algorithms over Graphs". In: *Advances in Neural Information Processing Systems.* Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/d9896106ca98d3d05b8cbdf4fd8b13a1-Paper.pdf.

[KHW19] Wouter Kool, Herke van Hoof, and Max Welling. "Attention, Learn to Solve Routing Problems!" In: *International Conference on Learning Representations.* 2019.

[KM22] Manuel Kauers and Jakob Moosbauer. *The FBHHRBNRSSSHK-Algorithm for Multiplication in $\mathbb{Z}_2^{5\times 5}$ is still not the end of the story.* 2022. DOI: 10.48550/ARXIV.2210.04045. URL: https://arxiv.org/abs/2210.04045.

[Maz+20] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. *Reinforcement Learning for Combinatorial Optimization: A Survey.* 2020. arXiv: 2003.03600 [cs.LG].

[Mni+13a] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning.* 2013. arXiv: 1312.5602 [cs.LG].

[Mni+13b] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning.* 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[Sch+17] John Schulman et al. *Proximal Policy Optimization Algorithms.* 2017. DOI: 10.48550/ARXIV.1707.06347. URL: https://arxiv.org/abs/1707.06347.

[Tak08] Ken Takata. "A Worst-Case Analysis of the Sequential Method to List the Minimal Hitting Sets of a Hypergraph". In: *SIAM Journal on Discrete Mathematics* 21.4 (2008), pp. 936–946. DOI: 10.1137/060653032.

[Wag21] Adam Zsolt Wagner. *Constructions in combinatorics via neural networks.* 2021. arXiv: 2104.14516 [math.CO].

## A. The result of running FK-A on $(\mathbf{F_2}, \mathbf{G_2})$

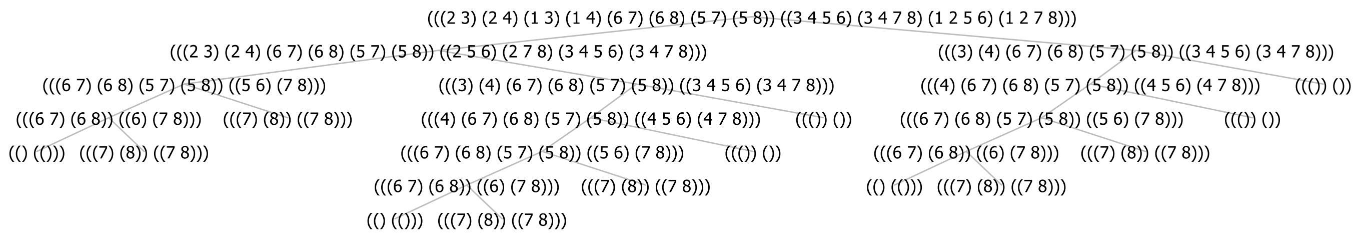

Figure 1. The recursion tree associated with running the `FK-A` on $(F_2, G_2)$ with maximum frequency pivoting and lexicographic tiebreaker.

Faculty of Mathematics, University of Waterloo
*Email address*: `p2salimi@uwaterloo.ca`

Department of Mathematics, Simon Fraser University
*Email address*: `tamon@sfu.ca`