Finding maximal cliques in uniform hypergraphs using Baum-Eagon inequality

Victor Melinceanu
v.melinceanu@student.utwente.nl
University of Twente
The Netherlands

ABSTRACT

In this paper, we are going to analyze the performance of an algorithm for finding maximal cliques in K-uniform hypergraphs. The algorithm is based on a replicator dynamics method using Baum-Eagon inequality.

The following algorithm has been researched in the past on circulant K-hypergraphs, but in this paper, we are going to use a dataset composed of both circulant K-hypergraphs and K-hypertrees. We also present an optimization for the algorithm, that makes each iteration faster by a factor of n (number of nodes) than what the current state of the art presents. We do this through an easy combinatorial optimization. A slight optimization for memory complexity when working with K-hypergraphs is also proposed. We have tested the fast implementation both in Python and in C++, showing that C++ is 15 times faster on average.

We also present for the first time an algorithm to generate random K-hypertrees. An interesting observation made during the testing phase is that the size of every converged clique is always exactly k, which is the cardinality of the hyperedge. We provide a new theorem and the proof, stating the maximum clique size in a K-hypertree is always exactly k.

The algorithm was tested on a dataset consisting of 500 tests all converging to a correct maximal clique. The results show that the convergence of the algorithm in terms of iterations depends on k and the number of hyperedges, and not on the number of nodes.

KEYWORDS

Replicator dynamics, Circulant K-hypergraphs, K-hypertrees, Maximum cliques, optimization for replicator dynamics

1 INTRODUCTION

The maximum clique problem has been for a long time a work in progress. Various algorithms have been developed to solve the task, but no general solution that works very efficiently has been found, because of the problem being an NP-Hard problem. However the last couple of years, research on this topic has been going more towards the direction of finding approximation algorithms for special structures of hypergraphs.

The focus on circulant hypergraphs is growing nowadays due to their application in computer networks. In computer networks, circulant K-uniform hypergraphs have been used to model the topology of communication networks, where the vertices represent nodes or switches, and the hyperedges represent connections between nodes or groups of nodes. Circulant hypergraphs have several desirable properties for modeling network topologies, such as symmetry, regularity, and low diameter. Another application of hypergraphs is in informational biology, mainly in modeling gene interactions [2].

K-hypertrees have been described in different literature with different definitions. However, Tomescu [8] defined K-hypertrees that was useful for solving Bonferroni inequalities. This same definition was later used by Ojas [5] to further extend some tree properties. There is no practical way of generating random K-hypertrees so this paper will introduce a dynamic programming method for generating this specific hypergraph structure for the first time in literature.

This paper focuses on using the method described in *Ahmed and Still* [1]. The interest in this algorithm comes from the fact that it doesn't use exhaustive search (backtracking) to find a maximal clique, but it rather focuses on reducing the problem to finding local minimizers in homogeneous polynomials over the unit simplex. For this reason, the algorithm might perform well for specific types of hypergraphs, independent of the number of nodes.

2 PROBLEM STATEMENT AND STRUCTURE

It is important to notice that even though research has been done on finding maximal cliques for circulant K-uniform hypergraphs, none of the current literature has optimized the *replicator dynamics method.* The method presented in [1] for finding maximal cliques shows to behave differently in practice than most exhaustive searches for maximal cliques. Hence the interest in researching this method.

K-hypertrees have not been studied in practice enough and they might have some useful applications in the field. Besides them not being studied, also no method for generating random K-hypertrees has been described. For that matter, this paper focuses on these particular mathematical structures. To tackle the above problems we are going to focus on the following research question.
RQ: What is the performance of the replicator dynamics method when finding maximal cliques in circulant K-hypergraphs and K-hypertrees?
We are going to answer the research question above by answering the following sub-questions.

- **RQ1:** Can the current implementation of the algorithm be optimized?
- **RQ2:** How does the algorithm behave on circulant K-hypergraphs?
- **RQ3:** How can we generate random K-hypertrees?

TScIT 39, July 7, 2023, Enschede, The Netherlands

- **RQ4:** How does the algorithm behave on K-hypertrees?
- **RQ5:** Is it more optimal to use C++ over Python for the following algorithm and by how much?

RQ1 is answered in 4.3.2, 4.3.3 and 5.2. **RQ2** is answered in 5.1.2. **RQ3** is answered in 4.2. **RQ4** is answered in 5.1.1. **RQ5** is answered in 5.3.

3 RELATED WORK

First of all the problem of finding maximum cliques in K-uniform hypergraphs is NP-Hard. The subproblem of finding maximum cliques in circulant K-hypergraphs and K-hypertrees, at the current moment is considered to be in the same class of problems. By researching heuristics and approximation algorithms, we may be able to observe some special structures of hypergraphs for which the problem is solvable in polynomial time, or close to polynomial.

The simplest way to solve the problem of maximum clique finding is by doing a recursive backtracking. This is a naive method that leaves a lot of space for heuristics and optimizations. This algorithm was optimized by Östergård [9], using the Russian doll technique. The Russian doll algorithm [3] is used to prune through the recursion tree and it minimizes the search space as much as possible.

Iosif [4] has also studied this problem. In this paper, it was researched how the method replicator dynamics and descent method work in practice on circulant K-hypergraphs. It was noticed that the replicator dynamics method works faster. It was also observed that with the increase of the parameter k, the two algorithms provide more and more different efficiencies compared to one another.

Note that no research on finding maximal cliques in K-hypertrees has been conducted. Also, the definition provided by [8] is non-trivial and no practical method for generating K-hypertrees has been described.

4 THEORY AND METHODOLOGIES

To test the maximal clique finding algorithm using the replicator dynamics method, we need first some dataset of K-hypergraphs, since the methods described below were derived only for hypergraphs with hyperedges of size k. For that purpose, we are going to present the theory and methods used to generate circulant K-hypergraphs and K-hypertrees.

4.1 Circulant K-hypergraphs

First of all we are going to introduce circulant K-hypergraphs.

Definition 1 A circulant K-hypergraph is a hypergraph $G = (V, E)$, where V is the set of vertices and $E \subseteq P_k(V)$ is the set of hyperedges, where $P_k(V)$ is the set of all subsets of size k, such that for every $e \in E$ and for every possible rotation (shift operation) of e, there exists $e1 \in E$, such that $e1$ corresponds to that rotation.

Circulant K-hypergraphs have been extensively discussed and researched by Plant [6]. They devised a way to generate random circulant K-hypergraphs, given the parameters n - the number of nodes, k - the size of every hyperedge and d - an estimator for the proportion of hyperedges out of all the possible hyperedges in a K-hypergraph with n nodes, which is $\binom{n}{k}$. So the hypergraph will have approximately $\binom{n}{k}d$ hyperedges.

This paper is going to provide a general overview and implementation of the generation algorithm. For technical details the interested reader is referred to [4, 6]. To generate random circulant K-hypergraph with n nodes, first we need to generate all the binary canonical necklaces of size n containing k ones. This is achieved using the algorithm provided by Plant [6]. Then we assign each binary canonical necklace the same probability d, and proceed to choose randomly the set of binary canonical necklaces that will constitute the generating set for our circulant K-hypergraph. To satisfy *definition 1*, we need to take each binary canonical necklace from our generating set and perform all possible shift rotations on the following strings, resulting in a random set of hyperedges for a circulant K-hypergraph.

Using the method described in Plant 2018 [6] and in short above, we are able to generate a random dataset of circulant K-hypergraphs using parameters n, k and d.

4.2 K-hypertrees

This specific structure called K-uniform hypertrees has not been studied in much detail. There are various definitions for various purposes, trying to generalize the tree structure. In this paper, we focus on the definition of K-hypertrees as presented in Tomescu [8] in the context of solving *Bonferroni inequalities*.

Definition 2 [8] A K-hypertree is a K-uniform hypergraph $G = (V, E)$ such that for $k = 2$, G is a tree with vertex set V and for $k >= 3$ G is defined recursively by the following two rules:

- If $V = \{1, ..., k\}$ then G has a unique edge $\{1, 2, ..., k\}$.
- If $|V| > k$ then there exists a vertex $v \in V$ such that if $E1, ..., Eq$ denote all edges containing v then $E1 \setminus \{v\}, E2 \setminus \{v\}, ..., Eq \setminus \{v\}$ induce a (k - 1)-hypertree with vertex set $V \setminus \{v\}$ and the remaining edges of G induce a K-hypertree with vertex set $V \setminus \{v\}$.

We note that a 2-hypertree is just a normal tree and has $n - 1$ edges. A 1-hypertree is a singleton or a tree with exactly one node. A K-hypertree with k nodes has exactly one hyperedge, as *definition 2* suggests. This definition is not trivial and visualising what a K-hypertree would look like is challenging. To create an actual dataset of K-hypertrees we need to find a way to construct a K-hypertree based on *definition 2*.

From the definition one can clearly see the inductive steps. We have a base case when $n = k$ and we see that constructing a K-hypertree with n nodes, depends on K-hypertrees with $n - 1$ nodes and on (k - 1)-hypertrees with $n - 1$ nodes. Let's call the vertex v for the second rule of the definition the terminal node, since it's the last node to be added. We notice that when v is added to the structure, we should already have a K-hypertree with $n - 1$ nodes. Now all we need to do, according to the second rule, is to choose a random (k-1)-hypertree with $n - 1$ nodes, let's call it T, and append to each of the hyperedges of T node v.

Let's write the construction method described above in a mathematical way. Let $T[k][n]$ be the set of all K-hypertrees with n nodes. Then if we want to generate a random K-hypertree $t3$, we have to choose a random $t1 \in T[k][n-1]$ and a random $t2 \in T[k-1][n-1]$. The terminal vertex v which in this case is $v = n$, is the last vertex we are adding to $t1$. As the second rule states, add all $e \in t1$ as hyperedges of $t3$ and $\forall e1 \in t2$ append node n to $e1$ and add this hyperedge to $t3$.

$$t3 = t1 \bigcup t2 + \{n\},$$

where $t1 \in T[k][n-1], t2 \in T[k-1][n-1]$ and $t3 \in T[k][n]$.

As one can notice, the number of hyperedges of a K-hypertree with n nodes is constant. Let $S[k][n]$ be the number of hyperedges in a K-hypertree with n nodes. From the inductive step above, we can see that $S[k][n] = S[k][n-1] + S[k-1][n-1]$. Tomescu [8] gives a proof stating $S[k][n] = \binom{n-1}{k-1}$. Just to check that the number of edges in our newly generated hypertree is correct we can see that $S[k][n] = S[k][n-1] + S[k-1][n-1] = \binom{n-2}{k-1} + \binom{n-2}{k-2} = \binom{n-1}{k-1}$.

below we present a method that uses dynamic programming to generate random K-hypertrees. The algorithm runs in a bottom up manner and generates 10 random hypertrees for each n and k.
Prerequisites:

- push_back is a utility function that appends to the end of a set.
- vector<int> represents a set of integers.
- $T[n][k]$ is a set of trees, meaning it's a set of sets of hyperedges, which in turn means that it's a set containing sets of sets. So the empty element is represented by $\{\{\{\}\}\}$.

Initially, we set the base cases for the dynamic programming. Then we generate 10 unique trees for each parameter choice using the dynamic programming approach explained above. After running the algorithm T will be populated with random hypertrees of needed sizes. The approximate complexity of this algorithm is

$$10 * \sum_{k=2}^{K} \sum_{n=k}^{N} \binom{n-1}{k-1}$$

since we have to iterate through each hyperedge of a tree in order to create it and we do this 10 times for each n and k. As stated above there are

$$\binom{n-1}{k-1}$$

hyperedges in a K-hypertree with n nodes.

Algorithm 1 Generate K-hypertrees

```
$T[1][1] = \{\{\{0\}\}\}$
for $i = 2$ to $N$ do
    $T[1][i] = T[1][i-1]$
    $T[1][i]$.push_back($\{\{i-1\}\}$)
end for
for $k = 2$ to $K$ do
    for $n = k$ to $N$ do
        if $k = n$ then
            $T[k][n]$.push_back($\{\{\}\}$)
            for $i = 0$ to $n-1$ do
                $T[k][n]$.back().back().push_back($i$)
            end for
        else
            vector<vector<int» $t1, t2, e, e1$
            int $sz$ = size($T[k-1][n-1]$), sz1 = size($T[k][n-1]$)
            int $tt = 0$
            while size($T[k][n]$) $\neq 10$ do
                $tt = tt + 1$
                if $tt = 100$ then
                    break
                end if
                int $r1$ = rand() % sz1, $r$ = rand() % sz
                $t1 = T[k][n-1][r1]$
                $t2 = T[k-1][n-1][r]$
                vector<vector<int» t3 = t1
                for each $e$ in $t2$ do
                    vector<int> e1 = e
                    $e1$.push_back($n-1$)
                    t3.push_back($e1$)
                end for
                if find($T[k][n]$, t3) = end($T[k][n]$) then
                    $T[k][n]$.push_back(t3)
                end if
            end while
        end if
    end for
end for
```

4.3 Replicator dynamics method to find maximal clique in K-hypergraph

In this paper we present a method that appeared in Ahmed and Still [1] for finding maximal cliques in K-hypergraphs. The idea was first presented in Bulò and Pelillo 2007 [7].

First we define the Lagrangean function of a K-hypergraph $G = (V, E)$ as $L_G(x) \colon \Delta \mapsto \mathbb{R}$

$$L_G(x) = \sum_{e \in E} \Big(\prod_{j \in e} x_j \Big), \tag{1}$$

where

$$\Delta = \{x \in \mathbb{R}^n \colon x >= 0, \sum_{i=0}^{n-1} x_i = 1\} \tag{2}$$

is the unit simplex [7]. Let's consider the function $h_G(x) \colon \Delta \mapsto \mathbb{R}$, defined below

$$h_G(x) = L_G(x) + \tau \sum_{i=0}^{n-1} x_i^k, \tau > 0 \tag{3}$$

We indicate with $h_G^j(x)$ the partial derivative of h_G with respect to x_j,

$$\mathrm{h}_G^j(x) = \tau \mathrm{k} \mathrm{x}_j^{k-1} + \textstyle\sum_{e \in E} 1_j \in e \prod_{i \in e \setminus \{j\}} x_i$$

Using the following theorem from [7], we can link the above theory to th problem of finding maximal cliques in K-hypergraphs.

Theorem 1 Let G be a K-hypergraph and $0 < \tau < \frac{1}{2^k - 2}$. A vector $x \in \Delta$ is a global/local minimizer of $h_{\bar{G}(x)}$ if and only if it is the characteristic vector of a maximum/maximal clique of G.
Define for a subset

$$S \subseteq N$$

the characteristic vector $x^S \in \Delta_n$ by

$$x^S = \begin{cases} \frac{1}{|S|}, & i \in S \\ 0, & i \notin S \end{cases}$$

Theorem 1 implicitly provides an isomorphism between the set of maximal cliques of a k-hypergraph G and the set of local/global minimizers of the function $h_{\bar{G}}$ over Δ. This allows us to find a local minimizer of the polynomial $h_{\bar{G}}$ and then to map that minimizer to a maximal clique of G.

To find a local minimizer for the homogeneous polynomial $h_{\bar{G}}$, we are going to use the *Baum-Eagon inequality* as stated in *theorem 5* in [7].

$$z_i = x_i \frac{\partial P(x)}{\partial x_i} / \sum_{j=0}^{n-1} x_j \frac{\partial P(x)}{\partial x_j}, i = 0, ..., n-1,$$

where P is a homogeneous polynomial.

The iteration starts with a random vector $x \in \Delta_n$. To find the local minimizer of $h_{\bar{G}}$ over Δ_n, [7] suggests the following transformations derived from the *Baum-Eagon inequality.*

$$y_i = \frac{x_i^{(t)} [k\xi - h_{\bar{G}}^i (x^{(t)})]}{k\xi - \sum_{j=0}^{n-1} x_j^{(t)} h_{\bar{G}}^j (x^{(t)})}$$

where t is the current iteration and y is the new vector created from applying the transformation to x.

4.3.1 Trivial implementation of the replicator dynamics method. The algorithm runs on $\bar{G}$ and not on G. This means that instead of using the set of hyperedges E we use the complement set, $\bar{E}$. We need to compute the complement of our K-hypergraph. Note that computing the complement of a K-hypergraph can be done in multiple ways and often the computation takes much more time than required, since one has to work with a subset that often spans the complete power set. below we show a way to compute the complement in time complexity $\binom{n}{k}\log_2(\binom{n}{k}k)$. The $\binom{n}{k}$ factor comes from iterating through all subsets of n choose k elements. The log_2 factor comes from the python hashmap, that holds all the hyperedges from G.

Algorithm 2 Complement(n, E, k)

```
$mp$ ← map of hyperedges in $G$
$a$ ← empty vector representing the current subset
$al$ ← empty vector for the complement hyperedges
gen$i$
if size($a$) = $k$ then
    if ¬$mp[a]$ then
        $al$.push_back($a$)
    end if
    return
end if
for $ii = i$ to $n$ do
    $a$.push_back($ii$)
    GEN($ii + 1$)
    $a$.pop_back()
end for
GEN(0)
```

Then we choose $\tau = \frac{1}{2^k}$ to satisfy *theorem 1.* We then initialise x with random numbers from 0 to 1, that is why we normalize by dividing every element of x by $\sum_{i=0}^{n-1} x_i$. We normalize because we need a random starting vector $x \in \Delta_n$. After convergence the

Algorithm 3 Replicator dynamics method

```
$\xi \leftarrow \max(\tau, \frac{1}{k!})$
$x$ ← random array of size $n$
$s \leftarrow \sum x$
$x \leftarrow \frac{x}{s}$
$it \leftarrow 0$
while True do
    $it \leftarrow it + 1$
    if $it = 10000$ then
        break
    end if
    $H \leftarrow 0.0$
    $q\_sum \leftarrow 0.0$
    $h$ ← empty list
    for $j \leftarrow 0$ to $n - 1$ do
        $y\_sum \leftarrow 0.0$
        for each $e$ in $\bar{E}$ do
            if $j$ is in $e$ then
                $y\_prod \leftarrow 1.0$
                for each $i$ in $e$ do
                    if $i \neq j$ then
                        $y\_prod \leftarrow y\_prod \cdot x[i]$
                    end if
                end for
                $y\_sum \leftarrow y\_sum + y\_prod$
            end if
        end for
        $y\_sum \leftarrow y\_sum + \tau \cdot k \cdot x[j]^{k-1}$
        $h[j] \leftarrow y\_sum$
        $H \leftarrow H + y\_sum \cdot x[j]$
    end for
    $y$ ← array of size $n$ initialized with zeros
    for $j \leftarrow 0$ to $n - 1$ do
        $y[j] \leftarrow \frac{x[j] \cdot (k \cdot \xi - h[j])}{k \cdot \xi - H}$
    end for
    s $\leftarrow \sum y$
    if s > 1.0001 or s < 0.999 then
        $y \leftarrow \frac{y}{s}$
    end if
    if $||y - x|| < 1e - 6$ then
        $x \leftarrow y$
        break
    end if
    $x \leftarrow y$
end while
return $(x, it)$
```

algorithm will provide a characteristic vector $x \in \Delta_n$ and the number of iterations it took. This vector will correspond to a maximal clique of our initial hypergraph G. We allow for a maximum of

10000 iterations. This algorithm is similar to the one implemented by Iosif. [4], and has the same time complexity of $O(n * |\bar{E}| * k)$. However the algorithm can be easily made faster by iterating only through hyperedges that contain a certain node, as opposed to iterating through all the hyperedges each time. The complexity thus becomes $O(|\bar{E}| * k)$, making it n times faster.

4.3.2 Optimization of the trivial implementation. The scope of the optimization is to speed up each iteration. The algorithm remains logically exactly the same, but by changing the way we calculate the formulas, we can speed it up by a factor of n. For this optimization we define edg_j as the set of all hyperedges containing node j, each hyperedge being represented as an index from $\bar{E}$. All edg_j should be pre-calculated.

Algorithm 4 Fast iteration implementation

```
1:  $pe \leftarrow$ array of size $|\bar{E}|$
2:  for $j$ in range(len($\bar{E}$)) do
3:      pe[$j$] $\leftarrow$ 1.0
4:      for $e$ in $\bar{E}[j]$ do
5:          pe[$j$] $\leftarrow$ pe[$j$] $\cdot x[e]$
6:      end for
7:  end for
8:  for $j$ in range($n$) do
9:      $y\_sum \leftarrow 0.0$
10:     for $e$ in edg[$j$] do
11:         $y\_prod \leftarrow$ pe[$e$]
12:         if $x[j] > 0$ then
13:             $y\_prod \leftarrow \frac{y\_prod}{x[j]}$
14:         end if
15:         $y\_sum \leftarrow y\_sum + y\_prod$
16:     end for
17:     $y\_sum \leftarrow y\_sum + \tau \cdot k \cdot x[j]^{k-1}$
18:     h.append($y\_sum$)
19:     $H \leftarrow H + y\_sum \cdot x[j]$
20: end for
```

This code should be replaced in the trivial implementation (3) on lines **14 - 30**. Let's now analyse the time complexity. In this algorithm we do the same as in the trivial one, but we iterate only over the sets of hyperedges that contain node j. Also we noticed that instead of calculating the product of each hyperedge each time we want to calculate the derivative (see line 18-23 in 3), we can precompute the product once (see line 2-6 in 4) and then divide by the current x_j. The complexity on line 2-6 of the fast algorithm (4), is obviously $O(|\bar{E} * |k)$, since for each edge in the complement we iterate through all the nodes in the edge, and there are k node in each hyperedge. Line 8-10, we iterate through each hyperedge of each node. We know that each hyperedge will appear k times in the vector *edg*, because k nodes are part of that hyperedge. So the complexity of line 8-10 as well as the complexity of the entire algorithm is $\boldsymbol{O(|\bar{E}| * k)}$. Note that this optimization doesn't speed up the number of iterations required to converge, but it makes the calculation of each iteration faster by a factor of n.

The **memory complexity** of this algorithm can also be easily calculated. We have a list of hyperedges and a list of the complement, adding up to a total of $\binom{n}{k}$ hyperedges each having k nodes, so the complexity for even holding the preliminary data in memory is $O(\binom{n}{k} * k)$. The *edg* vector contains $|\bar{E}| * k$ hyperedges, represented as indeces from $\bar{E}$, which means that the memory complexity for *edg* is $O(|\bar{E}| * k)$. Because $|\bar{E}| < \binom{n}{k}$ the overall memory complexity is $\boldsymbol{O(\binom{n}{k} * k)}$.

4.3.3 Memory optimization for storing K-hypertrees. Currently we are storing each hyperedge as a vector of integers. Usually an integer has 32-64 bits in modern programming languages, meaning that we require $32 * k$ bits to store a hyperedge. However instead of using a vector of integers, we could use a binary string, where $s_i = 1$, if $i \in edge$ and $s_i = 0$ otherwise, with $i = 0, ..., n-1$. In C++ as well as Python, there is a data structure called **bitset**. It contains a collection of bits, each occupying exactly 1 bit in memory. So storing n bits will require exactly n bits. We can encode each of hyperedge using a **bitset** of size n containing k ones representing the nodes in the hyperedge. Now storing a hyperedges requires n bits instead of $32 * k$ bits.

Of course this optimization makes sense only when $n < 32*k$, or $\frac{n}{k} < 32$. So for K-hypergraphs with $\frac{n}{k} < 32$, the memory complexity becomes $\boldsymbol{O(\frac{\binom{n}{k}*n}{32})}$. The time complexity shouldn't be affected by this change because bitsets allow most operations in constant or almost constant time. For example for finding the next set bit after some index, the bitset does that in $O(\frac{m}{32})$, where m is the size of the bitset.

5 EXPERIMENTS

Multiple experiments have been conducted in order to find out more about the fast iteration implementation and about the proposed replicator dynamics method.

5.1 Testing the fast iteration implementation

To test the fast implementation of the replicator dynamics method for finding maximal cliques in K-hypergraphs we first need a dataset. We made use of the generation algorithms described above to generate random circulant K-hypergraphs and random K-hypertrees. What we care about with each test is what is the number of iterations until the algorithm converges to a maximal clique. To check at the end if the algorithm converged correctly to a maximal clique, we take the given set returned by the algorithm and check if it's a maximal clique. For this experiment we tested the Python fast iteration implementation.

5.1.1 K-hypertrees. Since K-hypertrees with n nodes contain exactly $\binom{n-1}{k-1}$ hyperedges, we have analyzed for $k = \overline{3,5}$ and $n <= 30$ because higher values require a lot of computational time. For each k and n we have generated 4 different random trees.

5.1.2 Circulant K-hypergraphs. For this type of structure we picked n, k and d as opposed to just n and k for K-hypertrees. Here again we decided to consider $k = \overline{3,5}$ and both $n = \overline{10,19}$ and $n = \overline{50,59}$. However for $n = \overline{50,59}$ we have used k = 3,4 since computation takes a lot of time. For d we chose values starting from 0.35 to 0.95 with an increase of 0.05. So finally the dataset contains a random circulant K-hypertree for each n, k and d chosen.

5.2 Comparing efficiencies between the trivial and fast implementation

We know for a fact that the fast iteration implementation is faster than the trivial implementation, mathematically. However to find out about the practical aspect of the optimization we have conducted a couple of tests. For conducting the tests we chose the circulant K-hypergraph. We tested for $d = 0.35$, $k = 4$ and $n = \overline{50, 56}$. We care about the average run time of an iteration per test. Here we used the trivial and fast iteration implementation both in Python.

5.3 Comparing efficiencies between Python and C++ fast implementation

To show the readers the difference in the efficiencies of Python and C++ implementation, we also conducted an experiment to find out more about the factor by which these two implementations differ. Here we are going to see the entire run time of the algorithm. We tested again for circulant K-hypergraphs with $d = 0.35$ and $d = 0.95$, $k = 4$ and $n = \overline{50, 56}$.

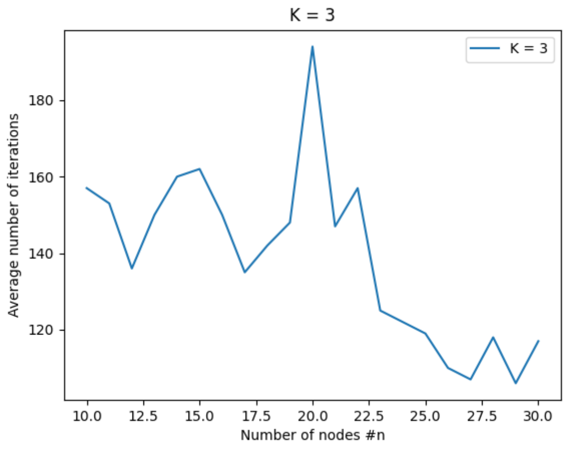
6 RESULTS

For all the experiments conducted, Python 3.11.3 with Sublime Text 3 as editor and C++14 with Codeblocks 20.03 as IDE was used. The experiments were run on a Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz on Windows 10.

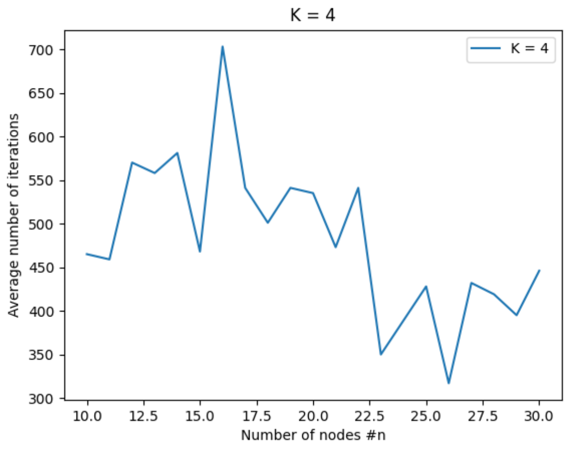
6.1 Testing the fast iteration implementation

All of the tests converged and have given a correct maximal clique, testing the implementation of the fast iteration method over a dataset of 500 graphs.

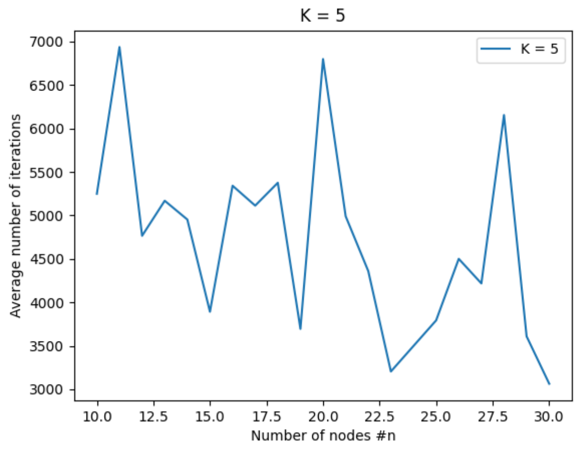
6.1.1 K-hypertrees. We have computed for every n and k the average number of iterations for all 4 random K-hypertrees. below we show the plots with the results obtained for each $k = \overline{3, 5}$.



We can notice that the number of iterations it takes for the algorithm to converge is not dependent on the number of nodes, since the line is not increasing, but rather it is oscillating, for all $k = \overline{3, 5}$. This is a positive result because the scope of the algorithm is to find maximal cliques for graphs of arbitrary size and this proves that in practice the number of iterations until convergence will not depend on n, but rather on the number of hyperedges.



We can also notice that the maximum average number of iterations for $k = 3$ is smaller than the minimum average number of iterations for $k = 4$. Same holds for $k = 4$ and $k = 5$.



6.1.2 Observation, new theorem and proof. After running all 300 K-hypertrees tests, the algorithm always converged to a clique of size k. This suggests the following theorem.

THEOREM 6.1. *For all K-hypertrees the size of the maximum clique is always exactly k.*

PROOF. We prove this statement using mathematical induction. First let $P(n, k) = true$ if the size of the maximum clique in any K-hypergraph is exactly k. Let $T(n, k)$ be any random K-hypertree with n nodes. Define a clique of size c in a K-hypergraph as a set containing all the $\binom{c}{k}$ subsets of size k composed of c nodes. We'll prove that $P(n, k) = true\ \forall n, k, n >= k$.

Base Cases:

- $P(k, k) = true\ \forall k$, because $T(k, k)$ contains the hyperedge $\{1, 2, ..., k\}$ which is a clique of size k.

- $P(n, 2) = true\ \forall n >= 2$, because for $k = 2$ the 2-hypertree is the same as a regular tree and the size of the maximum clique in any tree is 2.

Inductive Step: We want to find out if $P(n, k)$ is true, for that we assume that $\forall n1 <= n, k1 <= k, n1 \neq n, k1 \neq k$, $P(n1, k1) = true$, for some arbitrary positive integers $n, k >= 3, n >= k$.

Since we want to prove that $P(n, k) = true$, we only care about the statements $P(n-1, k)$ and $P(n-1, k-1)$. We know they both are *true* from the inductive step.

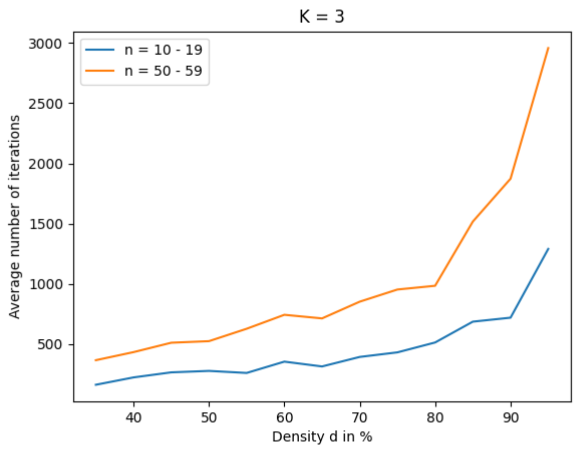
From the *definition 1* we know that $T(n, k) = T(n-1, k) \bigcup T(n-1, k-1) + \{n\}$. We know that $P(n-1, k) = true$, so the $T(n-1, k)$ hypertree contains a maximum clique of size k. However for $T(n-1, k-1)$, we know that we append node n to each hyperedge of the hypertree, as stated by rule 2 of the definition. So the size of a maximum clique for $T(n-1, k-1) + \{n\}$ increases from $k-1$ to k. Now we know that both components of the new hypertree we are constructing contain a maximum clique of size k. The only way $T(n, k)$ could have a clique of size greater than k, is if $T(n-1, k)$ and $T(n-1, k-1) + \{n\}$ contained parts of a clique of size $k+1$, that when concatenated would create the clique of size $k+1$ (**1**). We want to prove that $P(n, k) = true$, meaning there is no clique of size greater than k. So we will prove that there is no clique of size $k+1$ in $T(n, k)$ using proof by contradiction.

Assume there exists a clique of size $k+1$ in $T(n, k)$. From (**1**) we know that hyperedges of this clique are contained both in $T(n-1, k)$ and in $T(n-1, k-1) + \{n\}$, meaning that each hypertree must contain at least one hyperedge from this maximal clique of size $k+1$ (**2**). Because $T(n-1, k-1) + \{n\}$ is a set of hyperedges all containing node n and because (**2**), we know that n needs to be part of the $k+1$ clique. WLOG let's assume that the other nodes part of the $k+1$ clique are $\{1, 2, .., k\}$. A clique of size $k+1$ in a K-hypertree will have $\binom{k+1}{k} = k+1$ hyperedges. $\binom{k}{k-1} = k$ of those hyperedges need to contain node n, since it's a clique. So only 1 hyperedge not containing node n is present in the clique, namely $\{1, 2, ..., k\} \in T(n-1, k)$. Now for all the other k hyperedges that also contain n, we need to show that all the $\binom{k}{k-1}$ subsets with n appended are present, since it's a clique of size $k+1$. But if all the $\binom{k}{k-1}$ subsets with n appended are present in $T(n, k)$, then the subsets $\binom{k}{k-1}$ were also present in $T(n-1, k-1)$, menaing that $T(n-1, k-1)$ has a maximum clique of size k, but it doesn't since $P(n-1, k-1) = true$ means that the maximum clique has size $k-1$. So we arrived at contradiction.

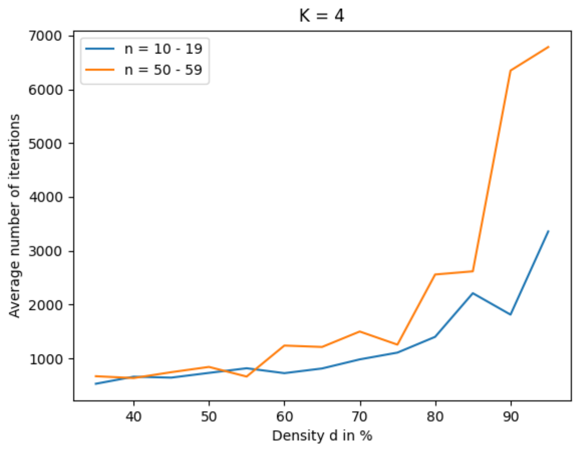
Therefore, by the principle of mathematical induction, the claim is proved for all positive integers n and k. □

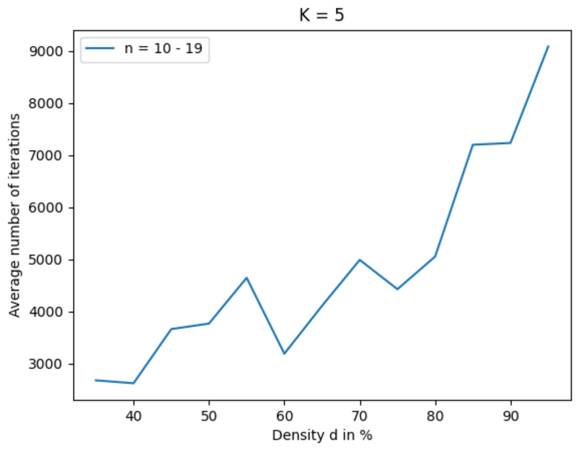
We provide a new property to K-hypertrees, that generalizes the definition of trees. It's a known fact that the maximum clique size in any tree is 2. Since trees are 2-hypertrees, the property holds. We proved for any k that a K-hypertree, defined by [8], has a maximal clique of size k.

6.1.3 Circulant K-hypergraphs. For this structure we decided to split the results into 2 datasets, one with $n = \overline{10, 19}$ and the other with $n = \overline{50, 59}$.



In the above plot we can see how for smaller n we get less number of iterations on average. However in the $k = 4$ plot, the discrepancy is very small between the two lines, showing that the number of iterations is more dependant on the number of hyperedges, or so to say it is dependant on the structure of the graph, it's k and d parameters.



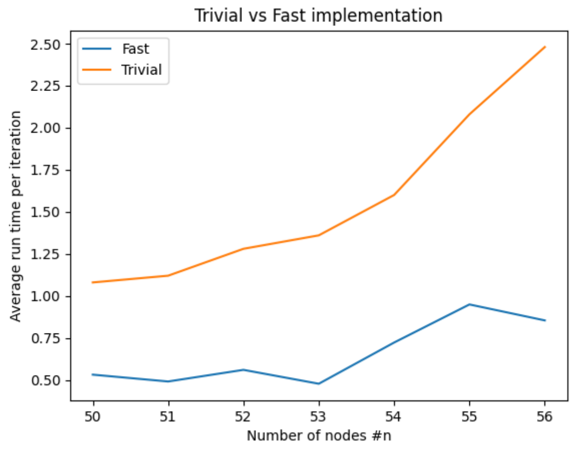


Judging by all 3 plots, we can see that the average number of iterations increases with d, especially when d gets close to 0.95, so to say close to a complete K-hypergraph. The number of iterations also increases considerably with the increase of k, because a higher value of k means more possible hyperedges in the hypergraph. So the number of the iterations is dependant on the number of hyperedges. Even if the complement of a dense graph has a small amount of hyperedges, the algorithm doesn't converge in less iterations, an iteration will be faster, but the number of iterations show to increase. This could be explained due to the fact that there are a small number of edges in the complement of a dense hypergraph, the characteristic vector changes slower, thus a higher number of iterations is required.

We note that for larger hypergraphs, the algorithm will still converge but it will take more iterations than 10000. While running some test cases with k = 6, the algorithm converged once in 83000 iterations. For larger hypergraph cases also the precision error could be changed from 10^{-6} to 10^{-8}, for a more accurate result. Since we are dealing here with a lot of floating numbers precision does play somewhat of a role.

6.2 Comparing efficiencies between the trivial and fast implementation

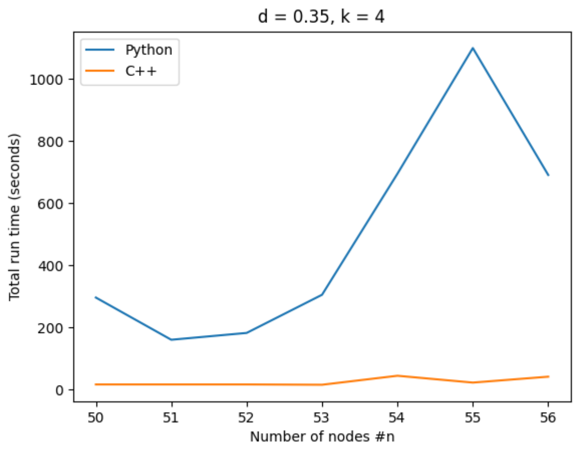
Again this experiment was conducted on circulant K-hypergraphs with $d = 0.35$, $k = 4$ and $n = \overline{50, 56}$.

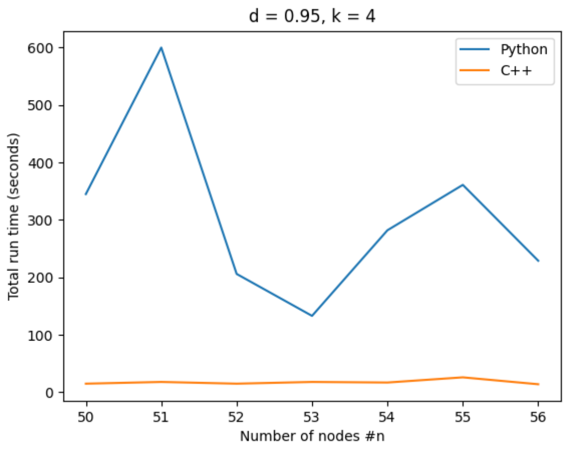


From the plot above we can clearly see that the fast implementation is at least 3 times faster in practice. Besides that we can see how the *Trivial* line grows much faster once n starts to grow. This is explained due to the n factor in the time complexity of the trivial implementation, which is absent in the fast implementation. The *Fast* line also grows with n, because once n increases, the number of expected hyperedges also increases, thus increasing the run time of the iteration.

6.3 Comparing efficiencies between Python and C++ fast implementation

We conducted the experiments on two datasets, one for sparse and one for dense K-hypergraphs.





The plots above show the difference between C++ and Python efficiency. C++ has always converged in under 44 seconds, whereas Python even got to 1100 seconds. After computing the average iteration run time over all tests in Python and C++, we have calculated the average factor by which C++ is faster than Python. For the first plot, with $d = 0.35$, the factor is 14.3. For the second plot, with $d = 0.95$, the factor is 16.7. So it turns out that in practice C++ works around 15 times faster than Python, when it comes to algorithmic implementation. This should suggest to the readers how much time you could save if you use the faster language.

7 CONCLUSION

After careful testing of the fast replicator dynamics method the following conclusions could be drawn. The following algorithm shows potential use in finding maximal cliques in K-hypergraphs, since the number of iterations and the time complexity is not dependent on n. However future optimizations should focus on minimizing the number of iterations by using line minimization for example, which is described in [1]. If the number of iterations could be brought down, then the algorithm would be very useful in practice.

In current literature the $O(\bar{E} * k)$ implementation is state of the art for the replicator dynamics method. The fast implementation works much better in practice, especially for K-hypergraphs with larger value of n. If we have K-hypergraphs with the property $\frac{n}{k} < 32$, then we can use a memory optimization by representing hyperedges with bitsets instead of vectors of integers.

It is advised to use C++ for algorithmic implementations if efficiency is what you're looking for. Even without the use of pointers or anything fancy, C++ turned out to be around 15 times faster than Python.

Finally, K-hypertrees defined by [8], may have more uses in the future, due to the proof of *theorem 6.1* showing another close connection between hypertrees and trees.

REFERENCES

[1] Ahmed Faizan and Still Georg. 2021. Two methods for the maximization of homogeneous polynomials over the simplex. *Computational Optimization and Applications* 80 (2021), 523–548. https://doi.org/10.1007/s10589-021-00307-1

[2] Heath E. Jefferson B. et al. Feng, S. 2021. Hypergraph models of biological networks to identify genes critical to pathogenic viral response. *BMC Bioinformatics* 22 (2021). https://doi.org/10.1186/s12859-021-04197-2

[3] M. Lemaitre G. Verfaillie and T. Schiex. 1996. Russian Doll Search for Solving Constraint Optimization Problems. 1 (1996).

[4] A. Iosif. 2022. An empirical evaluation of approximation algorithms to find maximal cliques in hypergraphs. http://essay.utwente.nl/89626/

[5] Ojas Parekh. 2003. Forestation in Hypergraphs: Linear K-Trees. *Electronic Journal of Combinatorics* 10 (11 2003). https://doi.org/10.37236/1752

[6] Lachlan Plant. 2018. Maximum Clique Search in Circulant k-Hypergraphs. (2018). https://doi.org/10.20381/ruor-22717

[7] Samuel Rota Bulò and Marcello Pelillo. 2007. A Continuous Characterization of Maximal Cliques in k-Uniform Hypergraphs. https://doi.org/10.1007/978-3-540-92695-5_17

[8] Ioan Tomescu. 1986. Hypertrees and Bonferroni inequalities. *Journal of Combinatorial Theory, Series B* 41, 2 (1986), 209–217. https://doi.org/10.1016/0095-8956(86)90044-4

[9] Patric R.J. Östergård. 2002. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics* 120, 1 (2002), 197–207. https://doi.org/10.1016/S0166-218X(01)00290-6 Special Issue devoted to the 6th Twente Workshop on Graphs and Combinatorial Optimization.