Maximum Clique Search in Circulant k-Hypergraphs

Lachlan Plant

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Computer Science (MCS)

Ottawa-Carleton Institute of Computer Science
School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa
Ottawa, Canada

Abstract

The search for max-cliques in graphs is a well established NP-complete problem in graph theory and algorithm design, with many algorithms designed to make use of internal structures of specific types of graphs. We study the extension of the problem of searching for max-cliques in graphs to hypergraphs with constant edge size k, and adapt existing algorithms for graphs to work in k-hypergraphs. In particular, we are interested in the generalization of circulant graphs to circulant k-hypergraphs, and provide a definition of this type of hypergraph. We design and implement a new algorithm to perform max-clique searches on circulant k-hypergraphs. This algorithm combines ideas from a Russian doll algorithm for max-cliques in graphs (Östergård 2002) with an algorithm based on necklaces for a class of circulant k-hypergraphs (Tzanakis, Moura, Stevens and Panario 2016).

We examine the performance of our new algorithm against a set of adapted algorithms (backtracking and Russian doll search for general k-hypergraphs, and necklace-based search for circulant k-hypergraphs) in a set of benchmarking experiments across various densities and edge sizes. This study reveals that the new algorithm outperforms the others when edge density of the hypergraph is high, and that the pure necklace-based algorithm is best in the case of low densities. Finally, we use our new algorithm to perform an exhaustive search on circulant 4-hypergraphs constructed from linear feedback shift register sequences on finite fields of order q that yields covering arrays. The search is completed for $2 \leq q \leq 5$ which solves the open case of $q = 5$ left by Tzanakis et al.

Acknowledgements

First, and most importantly, thank you to my supervisor Lucia Moura. In addition to your incredible assistance in this thesis, your support support during my classes was invaluable. Your help in navigating the school's bureaucracy, as well as always ensuring that I had TA work during the first two years of my studies made all the difference in my time at graduate school.

I would like to thank Vida Dujmovic and Brett Stevens for their work as examiners in the defence committee, and for their feedback and suggestions.

To my manager Dariush Eslimi, thank you for giving me the time I needed to finish this thesis while working full time. The time to meet with Lucia during the day, and the last minute vacations to finish this thesis were invaluable in the completion of it. The push you gave on the importance for me to finish this work was critical.

Lastly thank you to my family, Charles Plant, Susan Calverley, and Alex Plant. Your support and encouragement during the writing of this work over the years has kept me going, and kept me sane. Without your encouragement this work may never have seen completion.

Table of contents

1

Introduction

The problem of finding a maximum sized clique in a graph originated from social sciences in the work of Luce and Perry [10] through their study of social networks. They modelled social networks as a graph where edges represented relationships between people. The term clique denoted a group of people who all had a relationship with each other. The usage of the term clique was extended to the general study of graphs to denote groups of vertices which all share edges. The search for max-cliques in graphs has been shown to be NP-complete [8]. This means that no known algorithm is able to find the max-clique in polynomial time, and that finding a polynomial time algorithm for this problem would solve one of the most important problems in computer science of whether $P = NP$ [5].

Many methods of finding maximum cliques have been created, ranging from exhaustive searches to approximation algorithms. When searching for maximum cliques exhaustively, a backtracking algorithm is used to recursively search through valid cliques to find one which is of maximum size [2]. Östergård [12] improved upon basic backtracking by

using a technique called Russian Doll search [18], which uses aspects of dynamic programming within the max-clique search. By using the dynamic programming structure to iteratively search over increasingly larger subgraphs, Östergård was able to improve the search's ability to prune results which will not yield optimal results in an effort to reduce the overall runtime of the search.

In a graph, an edge is used to model a relationship between two vertices. This may be insufficient when modelling a real world problem, as in many problems a relationship may involve more than two entities. To use a similar structure to graphs in this model, we extend the size of the edges to sets which contain an arbitrary number of vertices. These structures with larger edges are referred to as hypergraphs, and if all edges contain the same number k of vertices, then it is a k-hypergraph. The concept of a clique extends naturally to k-hypergraphs by requiring all subsets of k vertices in the clique to define an edge. Unlike in graphs, the study of maximum cliques in hypergraphs has not been common. We find that these problems are generally more difficult to search for in hypergraphs than in graphs, even with the same density and number of vertices.

Due to the increased sizes of the edges in k-hypergraphs, some operations in the existing search algorithms will no longer work properly in k-hypergraphs. This includes the initialization of the algorithms, as well as the sub-algorithms used to filter out vertices which can not be added to the existing clique. In addition to this, the sub-algorithm used in each step of the search algorithm to filter the vertices grows in time complexity when moving from graphs to k-hypergraphs, from $O(n)$ to $O(n^{k-1})$ respectively, and becomes the most expensive operation in each step. This work defines the alterations to exhaustive backtracking searches and Östergård's search in order to use them in k-hypergraphs. In addition to this redefinition of the algorithms, this work examines different techniques to perform this expensive filtering operation, and provides experimental results detailing the cost and benefits of each.

As with most combinatorial problems, hypergraphs contain isomorphs, that is, ob-

jects with identical structure. The presence of large number of isomorphs can hinder the efficiency of a max-clique search. Consider a graph, or hypergraph, which contains many cliques of the same maximum size. Most search algorithms will search through all of these cliques, as they are hard to prune from the search tree using common bounding techniques such as the size bound [2]. Further consider graphs with many symmetries in which there are many isomorphic maximum cliques. One such graph is the circulant graph, in which all "rotations" of every edge is also an edge in the graph. Circulant graphs have been used in finding lower bounds on small Ramsey numbers [13], and circulant hypergraphs have been used in the generation of minimal covering arrays [17]. These graphs contain large number of isomorphic cliques, in fact every rotation of a clique is also a valid clique. With knowledge of the unique internal structure of these hypergraphs, optimized search algorithms can be created to avoid revisiting isomorphic partial solutions. Tzanakis et al. [17] developed an algorithm to only consider the lexicographically smallest clique among all of its isomorphs by combining backtracking search of max-cliques with the generation of necklaces in the work of Rusky, Savage, and Wang [15], and therefore avoiding revisiting isomorphic results.

The major contribution of this work is the design of a new algorithm for performing max-clique searches on circulant k-hypergraphs which we name Russian Necklace algorithm. By taking the structure of the efficient max-clique algorithm created by Östergård which is able to effectively prune branches of the search tree, and combining it with operations designed to exploit properties of cliques within circulant hypergraphs which will be presented here, we have been able to create an algorithm which rejects isomorphic cliques that appear in a previous major step of the search. In order to evaluate the performance of this algorithm, we have performed a set of experiments on random circulant k-hypergraphs in which the adapted versions of the Backtracking, Russian doll, and the algorithm designed by Tzanakis et al. were compared across a large set of circulant k-hypergraphs varying in edge size, density, and number of vertices.

The work of Tzanakis et al. [17] was done to generate covering arrays using linear

feedback shift registers of order q. In [17], they were able to perform an exhaustive search when $q \leq 4$. The search for $q \geq 5$ was not completed due to the time taken for the algorithm to complete the search, and was left as an open problem. In this work we perform an exhaustive search the case of $q = 5$ and close the problem for $q = 5$.

1.1 Thesis Organization

Chapter 2 focuses on giving definitions and background regarding strings and necklaces. In addition to giving definitions and basic proofs, we review algorithms for exhaustive generation and verification of necklaces, including proofs of the algorithms properties. Chapter 3 presents the definitions related to graphs and hypergraphs needed for later chapters. It also presents an overview on data structures to store graphs, and how these can be modified to store hypergraphs. Chapter 4 details the search for max-cliques on graphs. This chapter presents the basic backtracking algorithm as a foundation of the other algorithms in this work. Östergård's algorithm, based on a Russian doll search, is given here along with an overview of his experimental findings [12]. Chapter 5 focuses on the extension of the previous max-clique searches to k-hypergraphs. It details how to extend backtracking and Russian doll algorithms to handle k-hypergraphs. It presents the Necklace search algorithm for circulant k-hypergraphs created by Tzanakis et al [17]; indeed a contribution of our work is to reframe it to work with general circulant k-hypergraphs. This section also includes the main contribution of this thesis, the Russian Necklace algorithm along with the proofs of its correctness. Chapter 6 focuses on different sub-algorithms to optimize the cost of the slowest section of a max-clique search: the update on the set of candidates to be included in the clique. It outlines the various algorithms available and presents experimental results of their runtimes across various random circulant hypergraphs to determine the best strategy combinations. Chapter 7 includes the methodology and the experiments run to compare the performance of the max-clique search algorithms given in Chapter 5 on random circulant k-hypergraphs, as well as the results of the searches for optimal covering arrays, using hypergraphs and LFSRs. Chapter 8 details the conclusions of this work and future work in this area.

2

Necklaces

Some algorithms to find max-cliques in circulant hypergraphs presented in Chapter 5 rely on the concept of necklaces. A binary necklace is a collection of binary strings which can be obtained from each other via rotations. We present the definitions of necklaces and related concepts, as well algorithms to perform the verification of canonical necklaces, and for generation of all canonical necklaces of a given length.

2.1 Definitions and Necklace Properties

Binary strings are useful to represent sets, as we can see in the following definition.

Definition 2.1. Let S be a subset of an n-set $\{p_1, p_2, ..., p_n\}$. The *characteristic vector* of S is a vector $x \in \{0,1\}^n$ such that

$$x_i = \begin{cases} 1, & \text{if } p_i \in S, \\ 0, & \text{if } p_i \notin S. \end{cases} \tag{1}$$

Furthermore, the *characteristic string* of S is the corresponding binary string $x_1x_2...x_n$.

For example if $P = \{0, 1, 2, 3, 4, 5, 6, 7\}$ and $S = \{1, 4, 6\}$, the characteristic vector of S is (0,1,0,0,1,0,1,0), and the characteristic string is 01001010.

Definition 2.2. Let $\alpha = \alpha_0\alpha_1...\alpha_{n-1}$ be a string of length n. A string $\beta = \beta_0\beta_1...\beta_{n-1}$ is a *rotation* of α if and only if there exists a natural number s such that $\beta_i = \alpha_{(i+s) \ mod \ n}$ for all $0 \leq i \leq n-1$.

For use in future operations we denote the function $R(\alpha, i)$ to be the rotation of string α, i spaces to the right and $L(\alpha, i)$ to be the rotation of string α, i spaces to the left. Given the correspondence between subsets of an n-set and their binary characteristic string we can extend the notation of string rotation onto subsets.

Definition 2.3. The *rotation of a subset* A is the subset A' which corresponds to the rotation of the characteristic string of A.

Given set $N = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ and subset $A = \{1, 4, 6, 7, 8\}$, the right rotation by 2 places produces the set $A' = \{0, 1, 3, 6, 8\}$, as the characteristic string of A is $\alpha = 010010111$ and $R(\alpha, 2)$ is 110100101.

Definition 2.4. The *right justified characteristic string of a subset* A is the characteristic string of of the subset $R(A, (\max(N) - \max(A)))$.

Given set $N = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ and subset $A = \{1, 2, 4\}$, the characteristic string of A is $\alpha = 011010000$ and the right justified characteristic string is 000001101.

Definition 2.5. An n-bead, k-colour *necklace* is an equivalence class of k-ary strings of length n under rotation. In other words strings α and β are equivalent if there exists i such that $L(\alpha, i) = \beta$ (or alternatively $R(\alpha, i) = \beta$)

In this thesis, we concentrate on $k = 2$ or binary necklaces. Necklaces are most commonly used when defining strings, however from the definitions above we can extend this concept to sets through the characteristic string of the set. A necklace becomes an isomorphism class of subsets of an n-set under the rotation operation.

Definition 2.6. The *canonical necklace* of a necklace is the lexicographically smallest of the strings among the rotations of the necklace.

Figure 1 shows the necklaces of the length 6 binary strings, with the canonical necklace highlighted.

Figure 1: Necklaces of Length 6 Binary Strings

000000	**000001** 000010 000100 001000 010000 100000	**000011** 000110 001100 011000 110000 100001
000101 001010 010100 101000 010001 100010	**000111** 001110 011100 111000 110001 100011	**001001** 010010 100100
001011 010110 101100 011001 110010 100101	**001101** 011010 110100 101001 010011 100110	**001111** 011110 111100 111001 110011 100111
010101 101010	**010111** 101110 011101 111010 110101 101011	**011011** 110110 101101
011111 111110 111101 111011 110111 101111	**111111**	

Definition 2.7. The *canonical subset necklace* of a subset is either the empty set, or contains the smallest element of the base set and is the rotation whose right justified characteristic string is lexicographically smallest.

Figure 2 shows the subset necklaces of the set $\{0, 1, 2, 3, 4, 5\}$, with the canonical subset necklace highlighted. Their right justified characteristic strings are depicted in corresponding positions in Figure 1.

Figure 2: Subset Necklaces of $\{0, 1, 2, 3, 4, 5\}$

{}	**{0}** {1} {2} {3} {4} {5}	**{0,1}** {1,2} {2,3} {3,4} {4,5} {0,5}
{0,2} {1,3} {2,4} {3,5} {0,4} {1,5}	**{0,1,2}** {1,2,3} {2,3,4} {3,4,5} {0,4,5} {0,1,5}	**{0,3}** {1,4} {2,5}
{0,2,3} {1,3,4} {2,4,5} {0,3,5} {0,1,4} {1,2,5}	**{0,1,3}** {1,2,4} {2,3,5} {0,3,4} {1,4,5} {0,2,5}	**{0,1,2,3}** {1,2,3,4} {2,3,4,5} {0,3,4,5} {0,1,4,5} {0,1,2,5}
{0,2,4} {1,3,5}	**{0,2,3,4}** {1,3,4,5} {0,2,4,5} {0,1,3,5} {0,1,2,4} {1,2,3,5}	**{0,1,3,4}** {1,2,4,5} {0,2,3,5}
{0,1,2,3,4} {1,2,3,4,5} {0,2,3,4,5} {0,1,3,4,5} {0,1,2,4,5} {0,1,2,3,5}	**{0,1,2,3,4,5}**	

Next, we translate known facts about necklaces in the language of subset necklaces.

Proposition 2.8. *Let $N = \{0, 1, 2, ..., n-1\}$ and $S \subset N$ be a subset where $i = \max(S)$ and $0 \in S$. Let L be the largest set of sequential elements between 0 and i in N not in S. If $|L| < n - i - 1$ then S is a canonical subset necklace.*

Proof. Consider the characteristic string C of S. We divide C into 4 parts where $C = abcd$. The first part a consists of all elements from 0 to $\min(L) - 1$. Part b consists of elements in L, part c consists of elements $\max(L) + 1$ to i, and part d consists of elements $i + 1$ to $n - 1$. Since $0 \in S$ and $|L|$ is maximum, a begins and ends in a 1, and is non-empty. Likewise c begins and ends with a 1 and is non-empty. Both b and d are substrings of all 0's. The right justified string of C is $C' = dabc$. This string has $(n - i - 1)$ leading zeros followed by a 1, and ends in a 1. For a rotation of C' to be

lexicographically smaller than C' it must begin with at least $(n-i-1)$ 0's. From our assumption that $|L| < n-i-1$ there exists no substring of C' within abc that contains $n-i-1$ sequential 0s and so no rotation of C' can be lexicographically smaller. Since S contains 0 and the right justified characteristic string of S is lexicographically smallest, S is a canonical subset necklace. □

Corollary 2.9. *Let $N = \{0, 1, 2, ..., n-1\}$ and $S \subset N$ be a non-canonical subset necklace where $i = \max(S)$ and $0 \in S$. Let L be the largest set of sequential elements between 0 and i in N not in S. Then $|L| \geq n-i-1$.*

Proof. From Proposition 2.8 we know that $|L| < n-i-1$ is not possible, since this would make S a canonical subset necklace. □

It is however possible that $|L| = n-i-1$. Consider the set $N = \{0, 1, 2, 3, 4, 5, 6\}$ and $S = \{0, 1, 4\}$. $L = \{2, 3\}$ and $i = 4$. $|L| = 2 = 7-4-1$. S is not a canonical necklace as its rotation $S' = \{0, 3, 4\}$ has a lexicographically smaller right justified canonical string.

Corollary 2.10. *Let $N = \{0, 1, 2, ..., n-1\}$ and $S \subset N$ be a subset with $i = \max(S)$ and $0 \in S$. Let L be the largest set of sequential elements between 0 and i in N not in S. If $|L| > n-i-1$ then S is not a canonical subset necklace.*

Proof. We divide the canonical string C of S in the same way as in Proposition 2.8, and take the right justified string of C, *dabc*. The rotation to *bcda* has more leading 0s than *dabc* as b contains more sequential 0's than d and is therefore lexicographically smaller. S is therefore not a canonical subset necklace □

2.2 Canonical Necklace Verification

Naively, verifying that a string or a set is a canonical necklace can be done by iterating through all rotations of the string and checking if the string is the lexicographically smallest among all of its rotations. This approach will verify that the string is the canonical necklace in $O(n^2)$ time for a string of length n. When used in exhaustive search algorithms, this is too time consuming, as this computation will be done an exponential

number of times.

Duval [6] created an algorithm to factorize a string into Lyndon words, a concept similar to necklaces but with the property that the string is strictly smaller than all other rotations. This algorithm was adapted by Ruskey [14] to search for the canonical necklace of a given string. He showed that for any given string $S = s_0 s_1 \dots s_{n-1}$, we can find the lexicographically smallest rotation of S by taking the concatenation SS and searching for the lexicographically smallest substring of length $|S|$. Algorithm 1 is used to find the location of the lexicographically smallest rotation of S, and can be extended to verify if a string is a canonical necklace. This algorithm records the beginning position i of the lexicographically smallest sub-string seen so far, and loops through the positions j of SS to determine if a substring starting at j is lexicographically smaller than the one starting at i. In order to avoid string concatenation, which is a time expensive operation, the string SS is implicitly considered in determining the end of the main for loop and its positions accessed via modular arithmetic on the positions of S.

Algorithm 1 Lexicographical Minimal Rotation

```
1:  procedure GETNECKLACE(S)
2:      i ← 0, n ← |S|,
3:      j, k, p ← 1
4:      while j + k ≤ 2n do
5:          a ← S[(i + k − 1)  mod n]
6:          b ← S[(j + k − 1)  mod n]
7:          if a = b and k ≠ p then
8:              //case 1
9:              k ← k + 1
10:         else if a = b and k = p then
11:             //case 2
12:             j ← j + p
13:             k ← 1
14:         else if a > b then
15:             //case 3
16:             i ← j
17:             j ← j + 1
18:             k, p ← 1
19:         else
20:             //case 4; note a < b
21:             j ← j + k
22:             k ← 1
23:             p ← j − i
24:     return i
```

Next we develop our own arguments to reason about the correctness of Algorithm 1, which is given in Proposition 2.12.

Variable i is the start location of the lexicographically smallest substring seen so far, j is the beginning of the substring we are comparing against the string beginning at i, k is the length of the substrings we are comparing, and p is an upper bound that limits k. We begin with $i = 0$, as we have not seen any strings and the lexicographically smallest string starts at $i = 0$. Variable k is set to 1, and the beginning of the string we are comparing with is set to $j = 1$. For the remainder of this algorithm let S_a^b be the substring of SS starting at position a and having length b.

In the main loop of this algorithm, we have four main invariants:

1. $S_i^{k-1} = S_j^{k-1}$;

2. $p \leq j - i$;

3. $S_i^{k-x} \leq S_x^{k-x}$, for all $i < x < j$;

4. $i + k - 1 < j$.

The first invariant is that $S_i^{k-1} = S_j^{k-1}$. If k is 1 then this is trivially true as both S_i^{k-1} and S_j^{k-1} are empty strings. The only way for k to increase is if in the previous iteration of the loop, $s_{i+k-1} = s_{j+k-1}$ and so this equality invariant holds. The second invariant is that $p \leq j - i$, which initially holds. In case 2 (Line 10), j increases while i and p remain fixed, and $j - i$ increases. In case 3 and 4 (Lines 14 and 19) p is set to be equal to $j - i$. So the invariant remains true. The third invariant concerns starting locations between i and j. Since i holds the lexicographically smallest string discovered, and all values between i and j have been considered, for any value $i < x < j$, then $S_i^k \leq S_x^k$. The final invariant is that $i + k - 1 < j$. We prove this by showing that $k \leq p$ and combine with invariant 2 to yield invariant 4. In case 1, $k < p$ becomes $k \leq p$; in case 2, k is reset to 1; in case 3, k and p are reset to 1; and in case 4, values are reset such that $k = 1 \leq j - 1 = p$. From this we know that $k \leq p$ and since $p \leq j - i$, then $k - 1 < p \leq j - i$, and so $i + k - 1 < j$. This invariant gives us the important property that the strings being compared will never overlap with each other.

With these invariants in mind, we proceed to further analyse Algorithm 1. For a given step of the algorithm, we are comparing the value of s_{i+k-1} with s_{j+k-1}. There are 4 cases for the comparison between these values.

Case 1: $s_{i+k-1} = s_{j+k-1}$ and $k \neq p$ (Line 7).

We increase the value of k by one and continue to the next step of the algorithm.

Case 2: $s_{i+k-1} = s_{j+k-1}$ and $k = p$ (Line 10).

We have reached the maximum depth allowed for the comparisons of substrings starting at i and j. As we will show below, this is due to the presence of a repeating section of length p. In this case we increment j past this repeating section by incrementing it p places. We also reset k to 1. Proposition 2.12 will prove why we can increase j by p positions without the possibility of missing the lexicographically smallest rotation of S.

Case 3: $s_{i+k-1} > s_{j+k-1}$ (Line 14).

Since $s_{i+k-1} > s_{j+k-1}$ then $S_i^k > S_j^k$, and so we can update the location i of the lexicographically smallest substring found so far to j, we likewise update our k and p to 1.

Case 4: $s_{i+k-1} < s_{j+k-1}$ (Line 19).

In this case the substring starting at j is lexicographically larger than the one starting at i. We can move the search forward by incrementing j by k places and update k to 1 and p to $j - i$. Proposition 2.12 will prove why we can increase j by k positions, instead of only one.

Proposition 2.11. *Algorithm 1 evaluates all possible starting locations of j.*

Proof. As observed by Ruskey [14], we know that the lexicographically smallest rotation of S will be a substring of length $|S|$ in SS. The start location of this substring will be found in the first $|S|$ positions of SS so the minimal rotation of S will be found in $0 \leq i < |S|$. The algorithm will exit if $k + j > |SS|$. From invariant 4, $i + k - 1 < j$, and therefore $k \leq j - i \leq j$. If $j < |S|$ then $k < |S|$ and $j + k < |SS|$. In other words, if $j + k > |SS|$ then $j \geq |S|$. From this we can see that the algorithm only exits when there are no more valid locations of j to consider. □

Proposition 2.12. *Algorithm 1 finds the lexicographically smallest rotation of S.*

Proof. In order to show that Algorithm 1 finds the lexicographically smallest rotation of S, we will show that at the points where j is increased, no possible starting location for the lexicographically smallest rotation of S is skipped. From Proposition 2.11, we know that the algorithm will not exit while j is low enough for valid starting locations to still be present past j. We only need to verify that in the locations where j was increased to j', the gap between j and j' could not have contained the location of the lexicographically smallest rotation of S. There are two cases from the algorithm where this increase occurs: cases 2 and 4. We analyse each of them next.

Case 2: $s_{i+k-1} = s_{j+k-1}$ and $k = p$ (Line 10).

In this case we have found a repeated substring u. If $p = j - i$, then the string starting at location i must begin with uu, as the string starting at i is equal to the one starting

at j. We define this substring as $u = S_i^k$.

We then move j past this repeated section, this has the effect that the area between i and j is filled with repetitions of u, where $|u| = p$. If we previously moved j because $s_{i+k-1} < s_{j+k-1}$ then we trivially have a section which repeats once, and we had set the value of p to be the distance between them. If we move j because of this case, then we have added another repeating section between i and j. From this p becomes the length of the longest known repeating section between i and j, and p must be a divisor of $j - i$.

We can skip j forward past the repeating portion because, as we will show, we can now say that the string starting at j will not be the lexicographically smallest substring (and neither will the ones before position $j + p$). In the trivial case, if there is no $a > k$ where $S_i^a > S_j^a$ then i will never be assigned to the value of j. In the other case, we assume for a moment that for some $a > k$, we have $S_i^a > S_j^a$, i.e we assume that at some later point the substring starting at i will be lexicographically larger than the one starting at j. If S_i^a and S_j^a do not overlap or their overlap is less than $|u|$, then $S_i^a = u^x w$ and $S_j^a = u^x z$, where w is a prefix of u and $x \geq 1$. Since $S_i^a > S_j^a$ then $z < w$. The substring starting with z will be less than one starting with w and therefore $z < u$. From this, the substring starting with z will be less than the one starting at j. Similarly, if their overlap is of length greater than $|u|$, then $S_i^a = u^x w$ and $S_j^a = u^x z$, where w is a prefix of u and $x \geq 2$, and by the same argument j can not be the location of the lexicographically smallest substring. Because of this repetition, for all starting locations $j + b$ between j and $j + k - 1$ it is equivalent to starting at location $i + b$, which from invariant 4, will not be lexicographically smaller than start position i. This justifies the update of j to $j + p$.

Case 4: $s_{i+k-1} < s_{j+k-1}$ (Line 19)

Since $s_{i+k-1} < s_{j+k-1}$ then $S_i^k < S_j^k$ due to invariant 1. The substring starting at i will never be larger than the one starting at j and so we can update j. We do not however have to only increment j by 1. We know that $S_i^{k-1} = S_j^{k-1}$ from the first invariant. Take any $x < k$. Since $S_{i+x}^{k-x} < S_{j+x}^{k-x}$, and from the third invariant $S_i^{k-x} \leq S_{i+x}^{k-x}$, then $S_i^{k-x} < S_{j+x}^{k-x}$. From this, we do not have to consider any starting locations between j and $j + k - 1$, as they will all be lexicographically larger than the one starting at i.

Since the skipped positions can not contain the location of the lexicographically smallest rotation of S, and from Proposition 2.11 all valid starting positions must be examined before the algorithm can exit, Algorithm 1 finds the lexicographically smallest rotation of S. □

We can modify Algorithm 1 to verify if a string is a canonical necklace by changing the return value in Line 24 to indicate if $i = 0$. In addition to this, we can improve the performance for non-canonical necklaces. In Line 14, we have found a point where the lexicographically smallest rotation of the string is updated, which is an earlier indicator that at the end $i \neq 0$, and we can immediately return false. These two modifications on Algorithm 1 lead to Algorithm 2 for canonical necklace verification.

Algorithm 2 Necklace Verification

```
1:  procedure isCanonicalNecklace(S)
2:      i ← 0, n ← |S|,
3:      j, k, p ← 1
4:      while j + k ≤ 2n do
5:          a ← S[(i + k − 1) mod n]
6:          b ← S[(j + k − 1) mod n]
7:          if a = b and k ≠ p then
8:              //case 1
9:              k ← k + 1
10:         else if a = b and k = p then
11:             //case 2
12:             j ← j + p
13:             k ← 1
14:         else if a > b then
15:             //case 3
16:             return false
17:         else
18:             //case 4; note a < b
19:             j ← j + k
20:             k ← 1
21:             p ← j − i
22:     return true
```

We show now that this algorithm runs in linear time.

Proposition 2.13. *Algorithm 2 performs an $O(n)$ number of comparisons.*

Proof. The main-loop exit depends on the value of $j + k$, and at each iteration of the loop, there is a constant number of comparisons performed. In the worst case, we assume that Case 3 is never true, and the algorithm runs to the completion of the main loop. This gives us 3 cases for the modification of j and k during each step:

Case 1: $j' = j,\ k' = k + 1$ so $j' + k' = j + k + 1$,

Case 2: $j' = j + k,\ k' = 1$ so $j' + k' = j + k + 1$,

Case 4: $j' = j + k,\ k' = 1$ so $j' + k' = j + k + 1$.

At every step, we increment the value of $j + k$ by one, therefore there will be at most $2n$ iterations of this loop, which is $O(n)$ comparisons in total. □

In conclusion, Algorithm 2 is a linear time algorithm for verifying if a string is a canonical necklace, which is a great improvement over the naive quadratic one. Therefore, in the remainder of this thesis, every call to isCanonicalNecklace(S) refers to the implementation given in Algorithm 2.

2.3 Canonical Necklace Exhaustive Generation

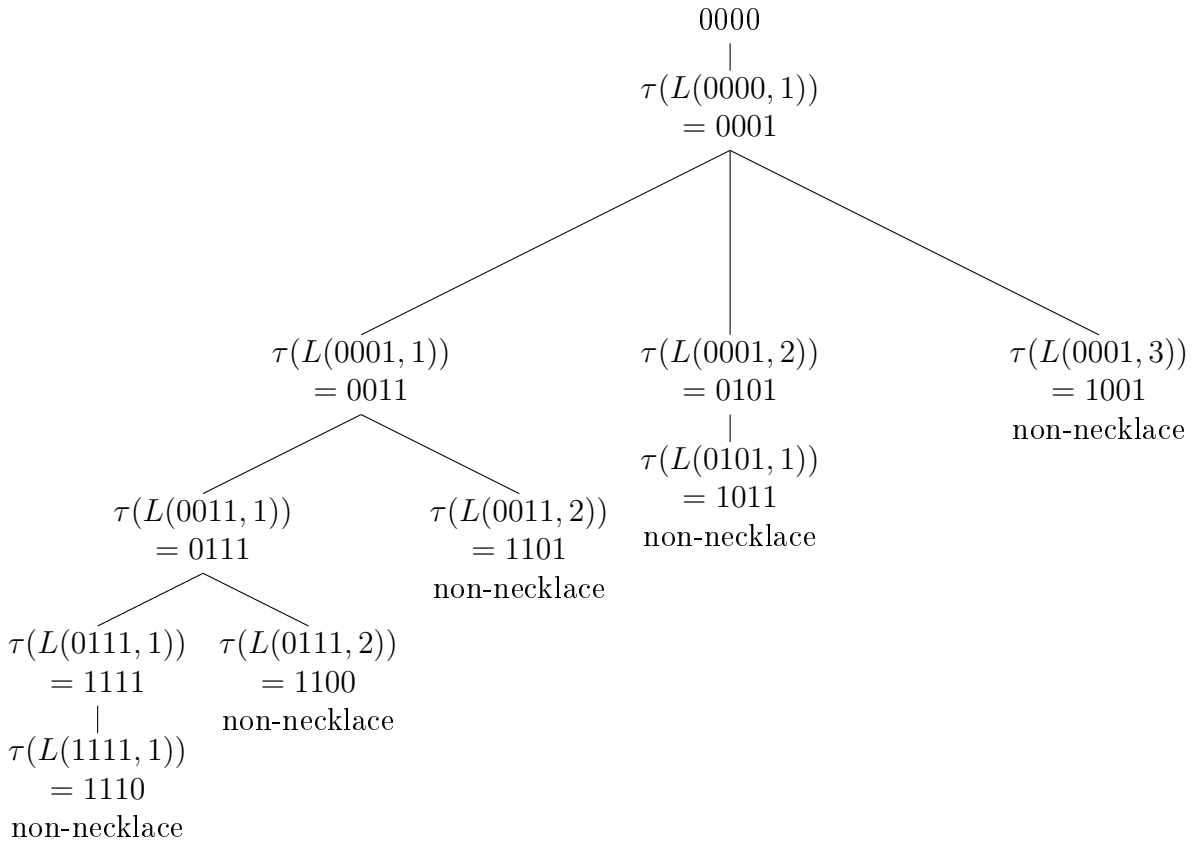
Rusky, Savage, and Wang [15] developed a fast algorithm for generating all binary canonical necklaces of a given length. The algorithm uses a recursive search and at each step takes a rotation of the current string, then flips the rightmost bit. We denote the function $\tau(X)$ to be the function which flips the right most element of the string from 0 to 1 or from 1 to 0, e.g $\tau(\text{“100010”}) = \text{“100011”}$. Algorithm 3 shows the recursive generation of the canonical necklaces. At each step, it prints the input string, which is a canonical necklace, then iteratively rotates the input string and generates a new string x through the τ function. If x is a canonical necklace it recursively calls with x as the input, if not it returns from this recursive call.

Algorithm 3 Generate Canonical Necklaces

```
1: procedure GENERATENECKLACES(y)
2:     print y
3:     done ← False
4:     while not(done) do
5:         y ← L(y, 1)
6:         x ← τ(y)
7:         if ISCANONICALNECKLACE(x) THEN
8:             GENERATENECKLACES(x)
9:         ELSE
10:            done ← True
11: PROCEDURE MAIN
12:    y ← 00...00
13:    PRINT y
14:    x ← τ(y)
15:    GENERATENECKLACES(x)
```

They were able to use two main properties to create this algorithm: first, once a non-canonical necklace has been found no children of that string in the search tree can be a canonical necklace. Secondly, once a non-canonical necklace has been found, none of its subsequent siblings can be canonical necklaces. Figure 3 shows the recursive construction of the length 4 canonical necklaces with this algorithm. For short, we write non-necklace to mean non-canonical necklace, indistinctively.

Figure 3: Necklace Generation for $n = 4$



By applying the above properties the algorithm can prune branches in which no necklaces can be found. This reduces the search space from naively checking all binary strings of a fixed length to verify which ones are canonical necklaces.

Proposition 2.14. *Algorithm 3 examines* $2(C-1)-1$ *strings, where* C *is the number of canonical necklaces.*

Proof. For every canonical necklace generated the algorithm enters into the loop in which creates new strings and performs recursive calls. This loop will only terminate once a non-canonical necklace has been found. This creates a pairing between the canonical necklaces, and non-canonical necklaces found. Every canonical necklace has exactly one direct child which is a non-canonical necklace. Likewise, every non-canonical necklace must have come directly from a parent which is a canonical necklace, as no recursion takes place on non-canonical necklaces. The only string that does not have a non-

canonical child is the initial string "00...00" printed in the main program which was not verified as it will always be a canonical necklace, and is only given one direct sibling i.e "00...01" which is also not checked. If there are C necklaces then there must be $C-1$ pairs of canonical and non-canonical necklaces checked. The total number of calls to isCanonicalNecklace is then $2(C-1)-1$ as "00...01" was not verified, but the corresponding non-canonical necklace was. □

2.4 Canonical Subset Necklace Generation

The algorithm from Rusky, Savage, and Wang [15] can be modified to generate Canonical Subset Necklaces, shown in Algorithm 5. The algorithm takes the standard algorithm to generate all subsets of an n-set (Algorithm 4) and applies the properties in Propositions 2.15 and 2.16 to bound the generation to only return canonical subset necklaces. In Algorithm 5, $RJ(X)$ denotes the right justified characteristic string of X, as in Definition 2.4.

Algorithm 4 Generate Subsets

```
1: procedure generateSubsets(Y, n)
2:     print Y
3:     \\ note max({}) = −1
4:     for a ← max(Y) + 1 to n − 1 do
5:         X ← Y ∪ {a}
6:         generateSubsets(X, n)
7: PROCEDURE main(n)
8:     Y ← {}
9:     generateSubsets(Y, n)
```

Algorithm 5 Generate Canonical Subset Necklaces

```
1: procedure GENERATESUBSETNECKLACES(Y, n)
2:     print Y
3:     for a ← max(Y) + 1 to n − 1 do
4:         X ← Y ∪ {a}
5:         if ISCANONICALNECKLACE(RJ(X)) then
6:             GENERATENECKLACES(X, n)
7:         else
8:             break
9: procedure MAIN(n)
10:    print {}
11:    Y ← {0}
12:    GENERATESUBSETNECKLACES(Y, n)
```

Proposition 2.15. *In Algorithm 5, no child of a non-canonical necklace can be a canonical necklace.*

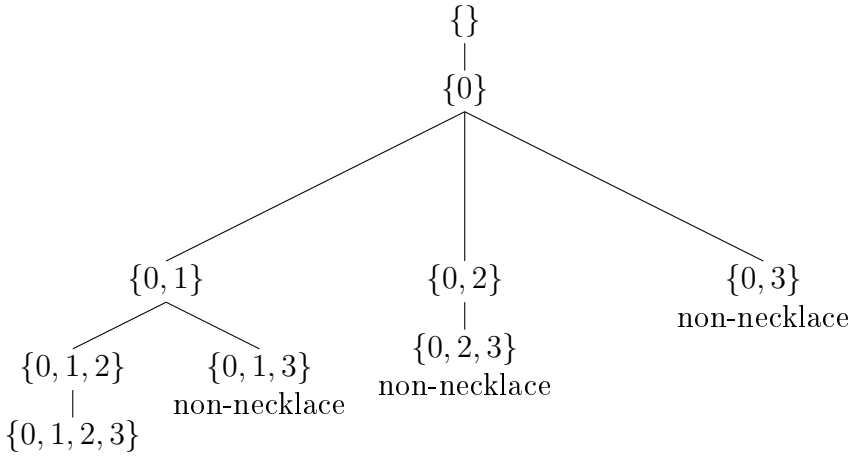
Proof. Let $N = \{0, 1, ..., n-1\}$ and X be a non-canonical necklace which contains 0 and has maximal element i. Let L be the largest set of sequential elements between 0 and i in N not in X. From Corollary 2.9, $|L| \geq n - i - 1$. For any child X' of X, $X \subset X'$, meaning L will remain unchanged. X' will contain an additional element j from $\{i+1, i+2, ...n-1\}$ and $j = \max(X')$. Thus $n-i-1 > n-j-1$ and so $|L| > n-j-1$. From Corollary 2.10, X' is not a canonical subset necklace. □

Proposition 2.16. *In Algorithm 5, no subsequent sibling of a non-canonical necklace can be a canonical necklace.*

Proof. Let $N = \{0, 1, ..., n-1\}$ and X be a non-canonical necklace which contains 0 and has maximal element i, and L be the largest set of sequential elements between 0 and i in N not in X. From Corollary 2.9, $|L| \geq n - i - 1$. In the subsequent sibling X' of X, $X' = X \setminus \{i\} \cup \{i+1\}$. Let L' be the largest set of sequential elements between 0 and $i+1$ in N not in X'. $|L'| \geq |L|$, as no new element less than i has been added to X'. It may however increase if $i - 1 \in L$ or if a new maximally sized set has been created which contains i. In either case $\max(X') = i + 1$ and $n - i - 1 > n - (i + 1) - 1$, so $|L'| \geq |L| \geq n-i-1 > n-(i+1)-1$ and therefore $|L'| > n-(i+1)-1$. From Corollary 2.10, X' is not a canonical subset necklaces. We can continue to apply this argument for the subsequent sibling X'' of X' until we have gone through all subsequent siblings. □

As no direct child of a non-canonical subset necklace can be a canonical subset necklace, it follows that the same can be said of its children, and so on. Once a non-canonical subset necklace has been discovered, no canonical subset necklace can be found in that branch of the generation tree, and it can be pruned. Similarly, for all subsequent siblings of non-canonical subset necklaces, no branch can contain a canonical subset necklace and can be pruned. As Algorithm 5 is based on the algorithm to generate all subsets, and only prunes branches which can not contain canonical subset necklaces, it must therefore generate all canonical subset necklaces of an n-set. Figure 4 shows the recursive construction of the canonical subset necklaces of the set $\{0, 1, 2, 3\}$ using this algorithm. Note the correspondence between the canonical necklaces in Figure 3 and canonical subset necklaces in Figure 4. The only difference in the corresponding nodes is that in the Subset Necklace Generation, set $\{0, 1, 2, 3\} = \{0, 1, \ldots, n-1\}$ has no child or sibling, while the Necklace Generation tests the non-necklace child and sibling of 1111.

Figure 4: Subset Necklace Generation



3

Graphs and Hypergraphs

In this chapter, we present the definitions related to graphs and hypergraphs needed in the following chapters. In particular, we review properties of circulant graphs, and give properties for what we define as circulant hypergraphs. We also present an overview on data structures to store graphs, and how these can be modified to store hypergraphs.

3.1 Graphs

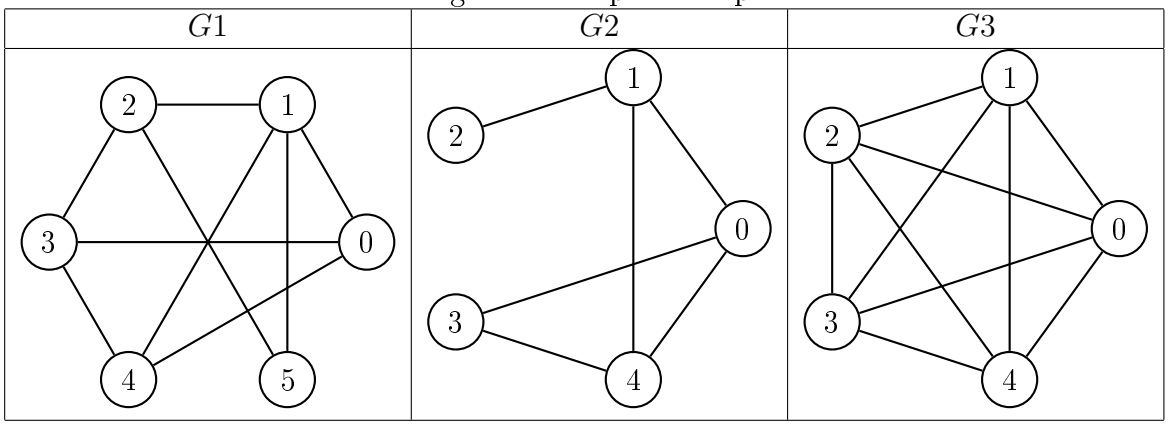
3.1.1 Definitions and Circulant Graph Properties

Definition 3.1. A *graph* G is a pair (V, E) of sets V and E where V is a finite set of elements called *vertices*, and E is set a of 2-sets of V called *edges*.

Definition 3.2. A *complete graph* $K = (V, E)$ is a graph in which all pairs of vertices are connected by an edge.

Figure 5 shows 3 graphs on 5 or 6 vertices, with graph $G3$ being complete.

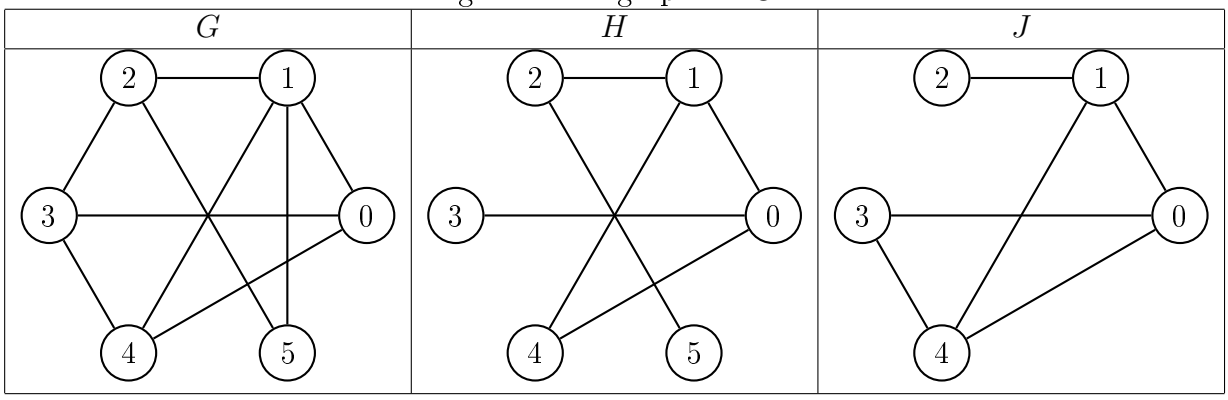
Figure 5: Graph Examples



Definition 3.3. A *subgraph* $H = (V_H, E_H)$ of a graph $G = (V, E)$ is a graph in which V_H is a subset of V, and E_H is a subset of E.

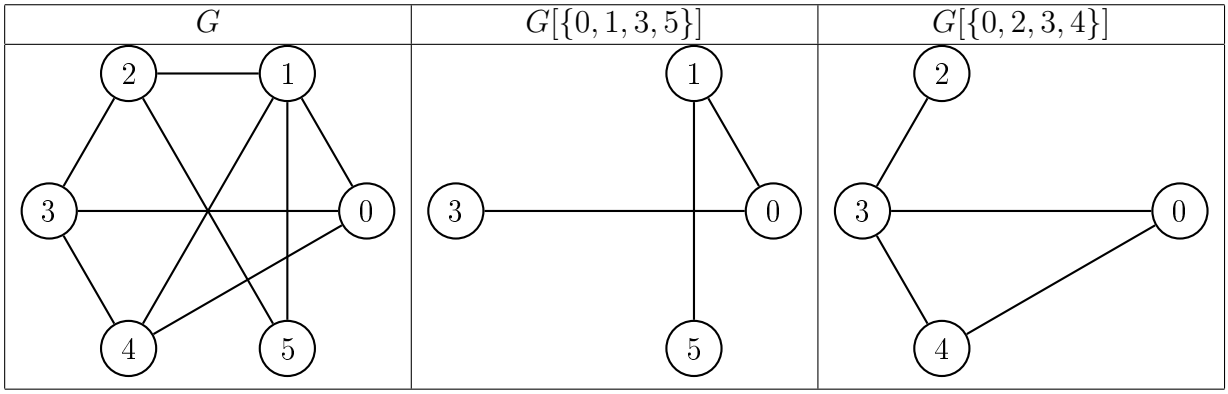
Edges may be present in E that would connect two vertices in V_H that may or may not be included in E_H. In Figure 6, graphs H and J are both subgraphs of G.

Figure 6: Subgraphs of G



Definition 3.4. The subgraph of graph $G = (V, E)$ *induced* by $A \subseteq V$ is the subgraph $G[A] = (A, E_A)$ where $E_A = \{\{v, w\} : v, w \in A \text{ and } \{v, w\} \in E\}$.

Figure 7: Induced Subgraphs of G



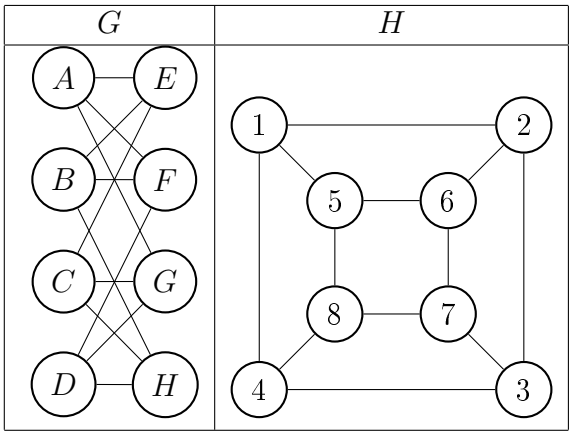
Definition 3.5. A clique C in graph $G = (V, E)$ is a subset of vertices of V in which the induced subgraph $G[C]$ is complete.

For a subset C, C is a maximal clique if there is no element $v_i \in V$ not in C such that $C \cup \{v_i\}$ is a clique. A maximum clique is a largest maximal clique within the graph. A clique of maximum size may not be unique, there may be multiple cliques of this size in the graph, each of these is considered a maximum clique.

Definition 3.6. Graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$ are *isomorphic* if there exists a bijection $f : V_1 \to V_2$ such that for all $v, w \in V_1$, $\{v, w\} \in E_1$ if and only if $\{f(v), f(w)\} \in E_2$.

In Table 8, graphs G and H are isomorphic; the following function f is an isomorphism: $f(A) = 1$, $f(B) = 6$, $f(C) = 8$, $f(D) = 3$, $f(E) = 5$, $f(F) = 2$, $f(G) = 4$, $f(H) = 7$.

Figure 8: Graph Isomorphism



Definition 3.7. An *automorphism* of a graph $G = (V, E)$ is an isomorphism between G and itself. An automorphism is a permutation on the vertex set, i.e. a bijection from V to V.

Definition 3.8. A *group* is a set S and operation $\circ$ such that the following properties hold:

1. Closure: For all $\alpha, \beta \in S$, $\alpha \circ \beta \in S$.
2. Associativity: For all $\alpha, \beta, \gamma \in S$, $(\alpha \circ \beta) \circ \gamma = \alpha \circ (\beta \circ \gamma)$.
3. Identity element: There exists an element $e \in S$ such that for all $\alpha \in S$, $\alpha \circ e = e \circ \alpha = \alpha$.
4. Inverse element: For each $\alpha \in S$ there exists an element $\alpha^{-1} \in S$ such that $\alpha \circ \alpha^{-1} = \alpha^{-1} \circ \alpha = e$.

Definition 3.9. For a group G, the *generators* $S \subseteq G$ is a set of elements from which every $g \in G$ can be expressed as a finite combination of elements of S under the group operation.

Definition 3.10. A *cyclic group* G is a group which is generated by a a single element a. This is denoted by $G = \langle a \rangle$

Proposition 3.11. *Let S_n be the set of all permutations of an n-set A. S_n is a group under the composition operation.*

Proof. Let $*$ be the composition operation.

1. S_n is closed under $*$ since the composition of permutations is also a permutation.
2. By definition, function composition is associative.
3. The identity under $*$ is the identity permutation i.e $Id : A \to A$ such that $Id(i) = i$.
4. Every permutation α has an inverse permutation, which is the permutation β such that $\beta(x) = y$ if $\alpha(y) = x$. It clearly follows that $\beta(\alpha(x)) = \alpha(\beta(x)) = x = Id(x)$

□

Theorem 3.12. *The automorphism group of a graph is a group under the operation of function composition.*

Proof. Let $*$ be the composition operation.

1. let α, β be automorphisms in $Aut(G)$. Since both α and β are automorphisms, they both produce permutations of G which are isomorphic to G. From the definition of a graph isomporphism, for all edges $\{v, w\} \in E$, $\{\alpha(v), \alpha(w)\} \in E$ and likewise for all edges $\{y, z\} \in E$, $\{\beta(y), \beta(z)\} \in E$. It must follow that $\{\beta(\alpha(v)), \beta(\alpha(w))\} \in E$ and therefore $\alpha * \beta \in Aut(g)$.
2. By definition, function composition is associative.
3. The identity automorphism Id is defined as $Id(v_i) = v_i$ for all v_i in V. For any automorphism $\alpha \in Aut(G)$, $(\alpha * Id)(v_i) = \alpha(Id(v_i)) = \alpha(v_i)$ and $(Id * \alpha)(v_i) = Id(\alpha(v_i)) = \alpha(v_i)$.
4. Every permutation α has an inverse permutation, which is the permutation β such that $\beta(x) = y$ if $\alpha(y) = x$ and $\beta(\alpha(x)) = \alpha(\beta(x)) = x = Id(x)$. For any automorphism $\alpha \in Aut(G)$, the inverse automorphism β must also be in $Aut(G)$. We assume $\alpha \in Aut(G)$ and $\beta \notin Aut(G)$. There must exist some

edge $\{v, w\} \in E$ such that $\{\beta(v), \beta(w)\} \notin E$. Let $\{a, b\}$ be the edge such that $\{\alpha(a), \alpha(b)\} = \{v, w\}$. Since $\alpha \in Aut(G)$, $\{a, b\} \in E$ and $\{v, w\} \in E$. As β is the inverse of α, $\{\beta(v), \beta(w)\} = \{\beta(\alpha(a)), \beta(\alpha(b)\} = \{a, b\}$. This violates the fact that $\{\beta(v), \beta(w)\} \notin E$ so $\beta \in Aut(G)$.

□

Definition 3.13. Given an integer $n \geq 1$ and a subset $S \subseteq \{1, 2, ..., \lfloor \frac{n}{2} \rfloor\}$, the *circulant graph* $CG_n(S)$ is the graph (V, E) where $V = \{v_0, v_1, ..., v_{n-1}\}$ is an n-set and $\{v_i, v_j\} \in E$ if and only if $\min\{|i-j|, n-|i-j|\} \in S$. For a graph $G = (V, E)$, G is circulant if there exists an ordering of the vertices $V = \{v_0, v_1, \ldots, v_n\}$, and a set $S \subseteq \{1, 2, ..., \lfloor \frac{|V|}{2} \rfloor\}$ such that $\{v_i, v_j\} \in E$ if and only if $\min\{|i-j|, n-|i-j|\} \in S$.

Figure 9 shows circulant graphs of six and nine vertices labelled by S.

From this definition, two further definitions can be shown to be equivalent.

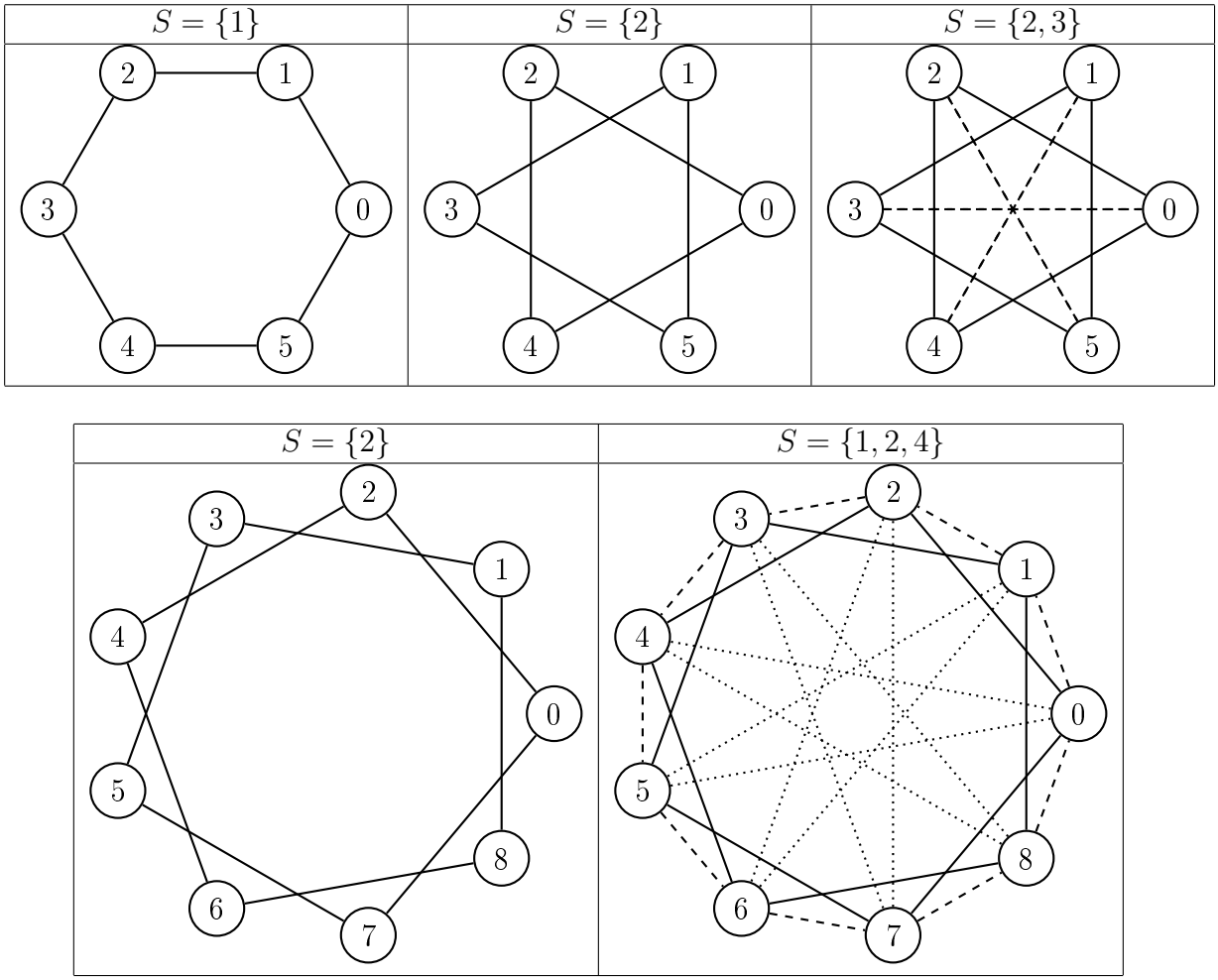
Proposition 3.14. *$G = (V, E)$ is circulant if and only if $Aut(G)$ contains a cyclic subgroup of size $|V|$ generated by the permutation $f : f(v_i) = v_{(i+1) \mod n}$, for some ordering of $V = \{v_0, v_1, ..., v_n\}$.*

Proof.

1. $G = (V, E)$ is circulant $\implies$ $Aut(G)$ contains a cyclic subgroup of size $n = |V|$ generated by the permutation $f : f(v_i) = v_{(i+1) \mod n}$.

Since G is circulant, there exists an ordering of vertices $V = \{v_0, v_1, ..., v_n\}$ and a set S such that $\{v_i, v_j\} \in E$ if and only if $\min(|i-j|, n-|i-j|) \in S$. We must first show that $f : f(v_i) = v_{(i+1) \mod n}$ is an automorphism of G. Since an automorphism is a permutation of the vertex set, for all edges $e = \{v_i, v_j\} \in E$ the permutation of e must also be in E for the permutation to be an automorphism. For simplicity we assume a vertex labelling such that $v_i = i$. From this $f(v_i) = v_{(i+1) \mod n} = f(i) = (i+1) \mod n$. For any edge $e = \{v_i, v_j\} \in E$, $\min(|i-j|, n-|i-j|) = s \in S$. We take the permutation of the vertices of this edge to form an edge $f(e) = \{f(v_i), f(v_j)\}$. Without loss of generality, we assume $i < j$. Since $i < n-2$, then $(i+1) < n-1$ and $(i+1) \mod n = i+1$. If $j < n-1$ then $j+1 < n$ and $(j+1) \mod n = j+1$ and

Figure 9: Circulant Graphs



$$\begin{aligned}
&\min(|f(i) - f(j)|, n - |f(i) - f(j)|) \\
&\quad = \min(|((i+1) \mod n) - ((j+1) \mod n)|, n - |((i+1) \mod n) - ((j+1) \mod n)|) \\
&\quad = \min(|(i - j)|, n - |(i - j)|) \\
&\quad = s \in S
\end{aligned}$$

so $f(e)$ must be an edge in G.

If $j = n - 1$ then $(j + 1) \mod n = 0$ and

$$\begin{aligned}
&\min(|i - (n-1)|, n - |i - (n-1)|) = s \\
&\quad = \min((n-1) - i, n - ((n-1) - i)) \\
&\quad = \min((n-1) - i, i + 1)
\end{aligned}$$

From this, e is an edge if $\min(|f(i) - f(j)|, n - |f(i) - f(j)|) = \min((n-1) - i, i + 1)$.

$\min(|f(i) - f(j)|, n - |f(i) - f(j)|)$

$= \min(|((i+1) \mod n) - (n \mod n)|, n - |((i+1) \mod n) - (n \mod n)|)$
$= \min(|((i+1) \mod n)|, n - |(i+1) \mod n|)$
$= \min((i+1), n-i-1|)$
$= s \in S$

so $f(e)$ must be an edge in G.

From Proposition 3.11 we know that $Aut(G)$ is closed under function composition and therefore $f(f(G)) \in Aut(G)$. We denote f^i as $f(f(\ldots f(G)))$ nested i times. From this, $f^i(v_j) = v_{(j+i) \mod n}$ and $f^n = Id$. The subgroup $(f, f^1, ..., f^n)$ is generated by a single element f and is therefore cyclic and contains $n = |V|$ elements.

2. $Aut(G = (V, E))$ contains a cyclic subgroup of size $|V|$ generated by the permutation $f : f(v_i) = v_{(i+1) \mod n} \implies G$ is circulant

From Proposition 3.11, we know that $Aut(G)$ is closed under function composition and therefore $f(f(G)) \in Aut(G)$. We denote f^i as $f(f(...f(G)))$ nested i times. From this, $f^i(v_j) = v_{(j+i) \mod n}$ and $f^n = Id$. The permutation $f : f(v_i) = v_{(i+1) \mod n}$ is equivalent to the rotation operation $R(\{v_j\}, 1)$ and in turn f^i is equivalent to $R(\{v_j\}, i)$.

We create the set $X = \{x_1, x_2, \ldots, x_n\}$ such that $x_i = \{\{a, b\} : \min\{|a-b|, n - |b-a|\} = i, \{a, b\} \in E\}$.

From here we can show that $|x_i| = n$ or 0.

There are $\frac{n(n-1)}{2}$ possible edges in the graph. Vertices are from the range $[0, n-1]$ so $i = 0$ is not possible as it implies $a = b$ or $a = 0, b = n$. Additionally, as the possible values of i can only exist in the lower half of the $n-1$ potential values of i, only $\frac{n-1}{2}$ possible values for i exist. A pair of vertices can only be placed in one set x_i. For any edge all rotations will be placed in the same x_i. For an arbitrary pair of vertices $\{a, b\}$ and all $1 \leq y \leq n-1$ we assume without loss of generality that $a < b$ and so $|a-b| = b-a = y$.
$R(\{a, b\}, y) = \{(a+y) \mod n, (b+y) \mod n\}$
$|((a+y) \mod n) - ((b+y) \mod n)|$
$= |((a+y) - (b+y)) \mod n|$
$= |(a-b) \mod n|$.

Let x_i be a set which contains between 1 and $n-1$ edges and $\{a, b\} \in x_i$. The size

of the cyclic subgroup is n so $f...f^n$ are in the subgroup. For any rotation of $R(\{a,b\},i)$ there exists a corresponding group member f^i and so $R(\{a,b\},i) \in E$. x_i can trivially have 0 edges present but can not have 1 to $n-1$ edges and must therefore have n edges present, or 0. We then construct $S = \{i : x_i \in X\}$

□

The permutation $f(v_i) = v_{(i+1) \mod n}$ when applied to $S \subseteq V$ is equivalent to the rotation operation $R(S,1)$ and in turn f^i is equivalent to $R(S,i)$

Proposition 3.15. *A graph $G = (V,E)$ is circulant if and only if there exists a set $\mathcal{N}$ of canonical binary necklaces of weight two and length $n = |V|$ such that, $E = \{L(N,i) : N \in \mathcal{N},\ 0 \leq i \leq |V|-1\}$.*

Proof.

1. $G = (V,E)$ is circulant $\implies$ there exists a set $\mathcal{N}$ of canonical binary necklaces of weight two and length $n = |V|$ such that, $E = \{R(N,i) : N \in \mathcal{N},\ 0 \leq i \leq |V|-1\}$.

Since G is circulant there exists a set S such that for all $v_i, v_j \in V$, $\{v_i, v_j\} \in E$ if and only if $\min(|i-j|, n-|i-j|) \in S$. For any element $s \in S$ we take the set $\{0,s\}$. Note that $\min(|0-s|, n-|0-s|) = \min(s, n-s)$. Since $s \leq \lfloor \frac{n}{2} \rfloor$, $\min(s, n-s) = s$ and so $\{0,s\} \in E$. From Proposition 3.14 we know that for $0 \leq i \leq n-1$ there exists an automorphic permutation of G based on $R(V,i)$ and so $R(\{0,s\},i) \in E$.

For any edge $\{a,b\} \in E$ we can then show that $\{a,b\}$ is a rotation of some edge $\{0,s\}$. We assume $a < b$ without loss of generality. Since $\{a,b\} \in E$, $\min(|a-b|, n-|a-b|) \in S$. This presents us with two cases, either $|a-b| = s$ or $n-|a-b| = s$, for $s \in S$. If $|a-b| = s$ then $a - b = -s$ since $a < b$ and so $b = s + a$. We apply the rotation $L(\{a,b\},a)$ to form a new edge $\{a-a, b-a\}$. Since $b = s+a$, $b-a = s+a-a = s$ and this edge is $\{0,s\}$. In the case of $n - |a-b| = s$, we know that $a < b$, so $|a-b| = b-a$, then $n-|a-b| = n-(b-a) = n+a-b = s$. We apply the rotation $L(\{a,b\},(n-b))$ to get the edge $e = \{(a+(n-b) \mod n, (b+(n-b)) \mod n)\}$.

$(b+(n-b)) \mod n$
$= n \mod n$
$= 0$
$(a+(n-b)) \mod n$

$= (n + a - b) \mod n$
$= s \mod n$
$= s$

So $e = \{s, 0\}$. From this we know that for each $s \in S$, all rotations of the edges $\{0, s\} \in E$ and all $e \in E$ are rotations of a set $\{0, s\}$. We select the canonical necklace of each of these sets to be a representative in $\mathcal{N}$ and define $\mathcal{N} = \{\{a, b\} : \{a, b\}$ is the canonical necklace of $\{0, s\}, s \in S\}$.

2. For $G = (V, E)$, if there exists a set $\mathcal{N}$ of length $n = |V|$ canonical binary necklaces of weight two such that, $E = \{R(N, i) : N \in \mathcal{N}, 0 \leq i \leq |V| - 1\}$, then G is circulant.
To demonstrate that G is circulant, we must be able to construct the set $S \subseteq \{1, 2, ..., \lfloor \frac{|V|}{2} \rfloor\}$ such that $\{v_i, v_j\} \in E$ if and only if $\min(|i - j|, n - |i - j|) \in S$.
For any edge $N = \{a, b\} \in \mathcal{N}$ there exists a rotation $R(N, i) = \{0, s\}$ such that $s \leq \lfloor \frac{n}{2} \rfloor$. We first take the rotation $R(N, (n - a))$ giving the edge $\{0, ((b + n - a) \mod n)\}$. If $((b + n - a) \mod n) \leq \lfloor \frac{n}{2} \rfloor$ we are done. Otherwise we take a second rotation $R(R(N, (n-a)), n-((b+n-a) \mod n))$. Since $(b+n-a) \mod n > \lfloor \frac{n}{2} \rfloor$, $n-((b+n-a) \mod n) \leq \lfloor \frac{n}{2} \rfloor$. This second rotation will give us the set $\{x, y\}$ where

$x = (0 + n - ((b + n - a) \mod n)) \mod n$
$= n - ((b + n - a) \mod n$
$y = (((b + n - a) \mod n) + n - ((b + n - a) \mod n)) \mod n$
$= (b + n - a + n - b - n + a) \mod n$
$= n \mod n$
$= 0.$

Thus, this rotation produces the edge $\{0, x\}$ where $x \leq \lfloor \frac{n}{2} \rfloor$. We use these x values to construct the set $S = \{x : \{0, x\} \in E\}$.

For any edge $e = \{a, b\} \in E$, $\{a, b\}$ is a rotation of $N \in \mathcal{N}$ and from above, a rotation of an edge $\{0, s\}$ where $s \in S$, and therefore $e = \{a, b\} = R(\{0, s\}, i)$ for some $0 \leq i \leq n - 1$. From this we can define $e = \{i, (s + i) \mod n\}$. If $s + i < n$ then $|i - (s + i)| = |-s| = s$ and so $\min(|a - b|, n - |a - b|) = \min(s, n - s) = s$. If $s + i \geq n$ then $(s + i) \mod n = (s + i) - n$.

We can substitute this into $|i - ((s+i) \mod n)|$ to get

$$\begin{aligned} |i - ((s+i) \mod n)| &= |i - (s+i-n)| \\ &= |n-s| \\ &= n-s \end{aligned}$$

From this, we know that $|a-b| = n-s$, and so

$$\begin{aligned} \min(|a-b|, n-|a-b|) &= \min(n-s, n-(n-s)) \\ &= \min(n-s, s) \\ &= s \in S \end{aligned}$$

From this, all edges in the graph will be represented by an element of S.

To show that there is no pair of vertices $a, b \in V$ such that $\min(|a-b|, n-|a-b|) \in S$ and $\{a,b\} \notin E$ a similar proof can be done as in section one of this proof. If $|a-b| = s$ we can apply the rotation $R(\{a,b\}, -a)$ to get the pair $\{0,s\}$ otherwise we apply the rotation $R(\{a,b\}, (n-b))$ to also get the edge $\{0,s\}$. This pair is a rotation of an edge $\{0,s\}$ which as shown above is a rotation of one of the edges $N \in \mathcal{N}$ and so must be an edge, violating our assumption that the pair was not an edge. Since we can form a set S based on our set of canonical necklaces the graph must be circulant.

□

This provides us with an important property regarding circulant graphs, namely that if an edge is present in the graph, all rotations of the edge are present in the graph.

3.1.2 Graph Data Structures

When storing a graph in memory, multiple techniques exist to represent the graph's edges. Each of these data structures prioritizes the time complexity of certain operations such as getting a list of all vertices adjacent to a given vertex, or verifying if vertices i and j form an edge. The simplest method is to simply store a list of the edge pairs. This uses a minimal amount of space, however on all other operations it becomes time intensive. Checking whether or not 2 vertices form an edge takes $O(|E|)$ time as all edges need to be checked to see if they match this pair. Similarly gathering all adjacent vertices for a vertex takes $O(|E|)$ time. The second main storage technique is an adjacency list. In

this structure we use an array a of lists to store the edges, where $a[i]$ contains a list of all vertices adjacent to vertex i. To look up an edge $\{u, v\}$ we search for v in $a[u]$, or for u in $a[v]$. This edge lookup will take $O(\max(deg(u), deg(v))) \subseteq O(n)$ time. To get the list of vertices adjacent to i we can return $a[i]$ in $O(deg(i)) \subseteq O(n)$ time (note we do not send $a[i]$ but a copy). The space required for this data structure is $O(|E| + n)$, as each edge will appear in two lists, and a full list of vertices is present. We can improve upon the adjacency list structure with an adjacency map. In this structure, the array of lists is replaced with an array of hashmaps [7]. This improves the edge lookup time to an expected $O(1)$ time, while requiring the same amount of space, and time for computing the list of vertices adjacent to a given vertex. Another possible data structure is the adjacency matrix, which is an $n \times n$ matrix m where $m[i][j] = 1$ if $\{i, j\} \in E$ and 0 otherwise. Verifying if two vertices form an edge takes $O(1)$ time. Gathering all adjacent vertices to a given vertex takes $O(n)$ time as the entire row of the matrix must be iterated over. The adjacency matrix will take $O(n^2)$ space, being by far the most wasteful for sparse graphs.

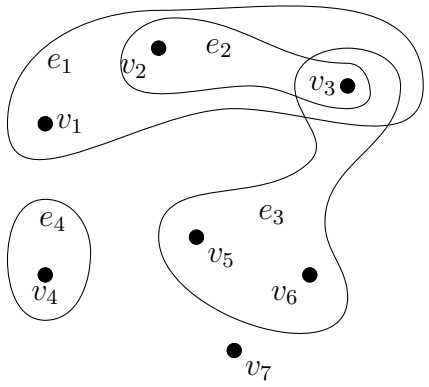
3.2 Hypergraphs

3.2.1 Definitions and Circulant Hypergraph Properties

Definition 3.16. A *hypergraph* H is a pair (V, E) of sets V and E where V is a finite set of elements called *vertices*, and E is set of subsets of V called *edges*.

Unlike graphs were each edge is a 2-subset, edges within hypergraphs can be of varying sizes within a single hypergraph, up to the size of the vertex set. Figure 10 shows a 7-vertex hypergraph with edges of size one to three.

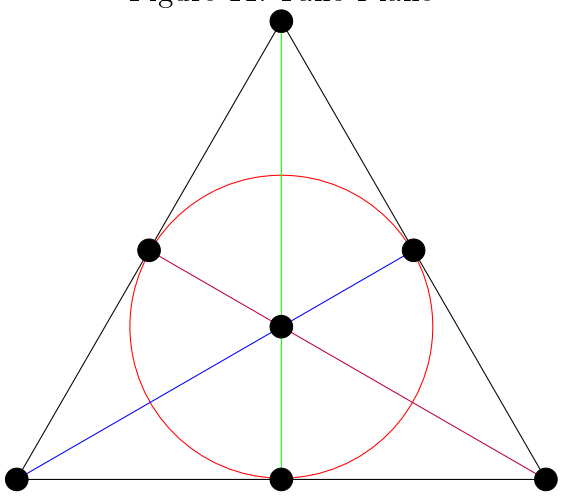
Figure 10: Hypergraph Example[9]



Definition 3.17. A *k-hypergraph* is a hypergraph in which all edges are k-subsets of V.

As a special case, a 2-hypergraph is a graph. Figure 11 [19] shows the Fano plane which is a 3-hypergraph. Each line passing through 3 vertices represents one edge.

Figure 11: Fano Plane



Definition 3.18. A *complete hypergraph* $H = (V, E)$ is a hypergraph in which E is the power set of V.

Definition 3.19. A *complete k-hypergraph* $H = (V, E)$ is a k-hypergraph in which every k-subset of vertices is an edge.

Definition 3.20. A *subhypergraph* $H = (V_H, E_H)$ of a hypergraph $HG = (V, E)$ is a hypergraph in which V_H is a subset of V, and E_H is a subset of E.

Definition 3.21. The subhypergraph of hypergraph $G = (V, E)$ *induced* by $A \subseteq V$ is the subhypergraph $H[A] = (A, E_A)$ where $E_A = \{e : e \subseteq A \text{ and } e \in E\}$.

Definition 3.22. Hypergraphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$ are *isomorphic* if there exists a bijection $f : V_1 \to V_2$ such that for all $e = \{e_1, ..., e_i\} \subseteq V_1$, $e \in E_1$ if and only if $\{f(e_1), ..., f(e_i)\} \in E_2$.

Definition 3.23. A *clique* C in a hypergraph $H = (V, E)$ is a subset of vertices of V in which the induced subhypergraph $H[C]$ is complete.

Definition 3.24. A *clique* C in a $k-$hypergraph $H = (V, E)$ is a subset of vertices of V in which the induced subhypergraph $H[C]$ is a complete k-hypergraph.

Definition 3.25. A k-hypergraph $H = (V, E)$ is *circulant* if there exists a relabeling of the vertices $V = \{0, 1, \ldots, |V|-1\}$ and a set $\mathcal{N}$ of canonical subset necklaces, each being a subset of V and having cardinality k, such that, $E = \{R(N, i)\colon\ N \in \mathcal{N},\ 0 \leq i \leq |V|-1\}$.

Proposition 3.26. *Let $H = (V, E)$ be a circulant k-hypergraph labeled $\{0, 1, \ldots, |V|-1\}$ and let $0 \leq i \leq |V|$. Then, $C \subseteq V$ is a clique if and only if $R(C, i)$ is a clique.*

Proof.

1. $C \subseteq V$ is a clique $\implies$ $R(C, i)$ is a clique

For any k-subset of vertices $\{e_1, \ldots, e_k\}$ in $D = R(C, i)$ there must be a corresponding k-subset $\{(e_1-i) \mod |V|, \ldots, (e_k-i) \mod |V|\}$ in C from the reverse rotation. Since C is a clique $\{(e_1-i) \mod |V|, \ldots, (e_k-i) \mod |V|\} \in E$. From Definition 3.25 all rotations of edges must also be edges and $\{e_1, \ldots, e_k\}$ is a rotation of $\{(e_1-i) \mod |V|, \ldots, (e_k-i) \mod |V|\}$ so $\{e_1, \ldots, e_k\} \in E$. D must therefore have all of its k-subsets being edges, and by definition is a clique.

2. $R(C, i)$ is a clique $\implies$ $C \subseteq V$ is a clique

We let $D = R(C, i)$ and therefore $C = R(D, -i)$. The same augment above can now be applied with D and $R(D, -i)$. □

Corollary 3.27. *For a circulant k-hypergraph $H = (V, E)$ and $0 \leq i \leq |V|$, $C \subseteq V$ is a maximal clique if and only if $D = R(C, i)$ is a maximal clique.*

Proof.

1. $C \subseteq V$ is a maximal clique $\implies$ $D = R(C, i)$ is a maximal clique

We assume that $C \subseteq V$ is a maximal clique and $D = R(C, i)$ is not. There must exist some vertex v such that $X = D \cup \{v\}$ is a clique. Applying the reverse rotation $R(X, -i)$ gives the subset $C \cup R(\{v\}, -i)$. From Proposition 3.26 this must be a clique but by our assumption C is maximal and we have reached a contradiction, so D must be maximal.

2. $D = R(C, i)$ is a maximal clique $\implies$ $C \subseteq V$ is a maximal clique

We let $D = R(C, i)$ and therefore $C = R(D, -i)$. The same augment above can now be applied with D and $R(D, -i)$. □

3.2.2 k-hypergraph Data Structures

The memory representation of the edges of a k-hypergraph presents a more difficult problem than in graphs. As the size of the edges grow, so does the possible number of edges, and therefore the amount of memory required, as there are $\binom{n}{k}$ possible edges as opposed to $\binom{n}{2}$ with graphs. If we have a 200-vertex graph, there are 19,900 possible edges, for a 200-vertex 5-hypergraph there are 2,535,650,040 possible edges. The standard edge list does not need modification for circulant k-hypergraphs. The other 3 representations, namely the adjacency list, adjacency map, and adjacency matrix, require modification to work with k-hypergraphs. When comparing storage techniques in graphs, we looked at the complexity of edge verification for a pair of vertices, and acquiring the list of adjacent vertices for a given vertex. In k-hypergraphs, the lookup of adjacent vertices to a vertex i is no longer an applicable operation, as with higher edge size this idea of a single vertex being adjacent to another no longer applies.

The simplest modification can be done to the adjacency matrix. Instead of an $n \times n$ matrix, this becomes an $n \times n \times \ldots \times n$ boolean matrix of k dimensions. The space requirement of this is $O(n^k)$. Verification of an edge is still $O(1)$ for this structure, as we are considering k constant. This new space requirement puts a large restriction on

the sizes of hypergraphs that can be searched using an adjacency matrix. If we have a 200-vertex graph, then the adjacency matrix would require 40,000 cells to store 19,900 possible edges. For a 200 vertex 5-hypergraph 320,000,000,000 cells would be required to store 2,535,650,040 possible edges.

A modification of the adjacency list can be done in two ways. The first one is to store a list of vertices, and for each vertex to store a list of incident edges. This takes $O(k|E| + n)$ space, while lookups may now take $O(|E|)$ instead of the $O(n)$ lookup in graphs. The second technique is to borrow from an adjacency matrix and referred to hereafter as a matrix adjacency list. A $(k-1)$-dimension matrix m is constructed to hold lists of vertices, specifically $m[i][j]\dots[x]$ holds the list of vertices which will form an edge with $\{i, j, \dots, x\}$. This reduces edge lookup to $O(n)$ while increasing the space complexity to $O(k|E| + n^{k-1})$. Both of these constructions can be applied to adjacency maps, with the list in either case being replaced by a hashmap. Both will perform edge lookup in an expected $O(1)$ time and have the same space complexity as their list counterparts. This constant time however will be slower for the first option of a single list, as the hashing of a full edge object will be more complex and take more time than the hashing of an integer used in the latter option.

The circulant structure of hypergraphs allows us to lower the space complexity of the adjacency matrix, matrix adjacency list, and matrix adjacency hashmap. For a circulant k-hypergraph $H = (V, E)$, from Definition 3.25 we know that for every edge $e \in E$, there must exist at least one rotation of e which contains the vertex 0. We can remove a dimension from our matrices by assuming the existence of this vertex. For any edge $e = \{e_1, e_2, ..., e_k\} \in E$ we can verify its presence in the graph by verifying the presence of the edge $L(e, e_1) = \{0, e_2 - e_1, ..., e_k - e_1\}$. Note that as long as edges to be verified are created in sorted order then this shift operation does not require modular arithmetic. Since the first dimension in the adjacency matrix or matrix adjacency hashmap will be 0 for all edges we can remove this dimension entirely. In an adjacency matrix this would store the hypergraphs edges in $O(n^{(k-1)})$ space. For matrix adjacency hashmap and ma-

trix adjacency lists this will use $O(|E| + n^{(k-2)})$ space.

In the adjacency map structure, the space required will increase with the hypergraph's density. We can however use a strategy to control the size of the hashmaps. If the density of the hypergraph is less than 50% we store in the hashmap the vertices that will complete an edge. If the density of the hypergraph is greater than 50% we store in the hashmap the vertices that will not complete an edge, and invert the values returned by the edge lookup. This enables us to have hashmaps which on average contain less than half of the vertex set, regardless of the actual hypergraph density.

In our implementations described in Chapters 6 and 7, we use the matrix adjacency hashmap with the above improvement based on density. This enables fast $O(1)$ verification that a set of vertices is an edge, which enables several large max-clique searches. However with larger 4-hypergraphs and 5-hypergraphs the memory requirement becomes extensive.

The following chapters contain details on the algorithms used in the exhaustive search for max-cliques in graphs, and circulant k-hypergraphs. These techniques include the basic algorithms which can be used on any graph or k-hypergraph, as well as the ones that exploit properties of circulant k-hypergraphs to avoid searching isomorphic results.

4

Max-Clique Search in Graphs

This chapter details techniques for searching for max-cliques in graphs. It presents the basic Backtracking algorithm, as well as Östergård's Russian doll algorithm and his experimental results.

4.1 Backtracking Search for Cliques

The search for the maximum clique (max-clique) is a time expensive search since the problem of finding a maximum clique is an NP-Complete problem, and as such no known algorithm can solve this problem efficiently. In order to solve this problem to optimality, we must use exhaustive search techniques to search through all viable candidate cliques to discover the maximum one.

The basis for all of the algorithms to follow comes from the standard backtracking search algorithm, presented in Algorithm 6. This technique for max-clique searching was developed by Pardalos [2]. The algorithm follows a basic branch-and-bound structure

in which not all valid solutions are searched over, but only those which can produce a better result than one previously found will be considered.

Algorithm 6 Max-clique Backtracking Search

```
1:  procedure BacktrackRecurse(C, U)
2:      if |U| = 0 then
3:          if |C| > |maxClique| then
4:              maxClique ← C
5:          return
6:      while U ≠ ∅ do
7:          i ← min{i ∈ U}
8:          U ← U \ {i}
9:          X ← GraphFilter(C, i, U)
10:         C ← C ∪ {i}
11:         if |C| + |X| ≤ |maxClique| then
12:             return
13:         BacktrackRecurse(C, X)
14:         C ← C \ {i}
15:     RETURN
1:  procedure BacktrackSearch(Graph)
2:      Global maxClique ← ∅
3:      U ← Graph.Verts
4:      BacktrackRecurse({}, U)
5:      return |maxClique|
```

In the recursive section of the algorithm, the algorithm is passed the current clique being examined C, and the current candidate set U. At this point all elements of the candidate set are viable vertices to be added to the clique, that is to say for every element $i \in U$ and $j \in C$, $\{i, j\}$ is an edge in the graph. U is first checked to determine if it is empty, and if it is, we check the size of the current clique against the largest clique found so far. If our new clique is larger, we record it before returning. Line 6 begins the branching section of the algorithm. While the candidate set is not empty, we first remove the minimal member i from the candidate set and form a new candidate set X by filtering out all members of the candidate set that do not form an edge with i. Vertex

i is then added to the clique. The size of the clique plus the size of the new candidate set is checked against the largest found clique. In the best case all elements of the candidate set will eventually be added into the current clique, but if doing so will not generate a clique larger than the largest one found so far, it would be a waste of time to search this branch and so the branch is pruned. This pruning is using the so called "size bound". The search is then repeated for the new clique and candidate sets. The algorithm is initialized with an empty clique as the current clique and the entire set of vertices as the candidate set.

4.1.1 Filtering

The most expensive operation in the backtracking search is the filtering of the candidate set, called in line 9 of Algorithm 6. This step requires that the entire candidate set be looped through and each element verified to still be eligible for the clique. We can expedite the process in two ways. The first way is by only verifying the presence of an edge between each candidate and the newest element to be added. Since we know that at every step the entire candidate set has been verified with every previously added element in the clique, we know that its inclusion in the clique will not violate the clique property. The only element in the clique not yet compared to the candidates is the most recent addition and we can then restrict our comparisons to this element. After the filtering (Algorithm 6 line 11) a size bound is taken, and the branch may be pruned if the new candidate set does not contain a sufficient number of elements to create a new max-clique. We can also apply this check during the filtering. We record the number of times an element has failed to be added into the new candidate set, and at each failure, compare this to the maximum number of removals before this size bound can be reached. If this value is reached, we return an empty candidate set, causing the algorithm to prune this branch.

Algorithm 7 Candidate Filter

```
1:  Global G = (V, E), maxClique
2:  procedure GraphFilter(C, i, U)
3:      X ← ∅
4:      m ← (|C| + |U| + 1) − |maxClique|
5:      for x ∈ U do
6:          if {i, x} ∈ E then
7:              X ← X ∪ {x}
8:          else
9:              r ← r + 1
10:             if r ≥ m then
11:                 return ∅
12:     return X
```

4.2 Russian Doll Search

Verfaillie et al.[18] developed a variant of branch-and-bound backtracking for solving constraint optimization problems. Instead of looking at the largest problem initially and then iteratively shrinking the problem size as the backtracking search does, their technique instead looks at small subproblems first, and uses information from them to prune the search tree more effectively in later stages. This algorithm is referred to as the Russian Doll algorithm due to its resemblance to Russian nesting dolls. Östergård [12] used this technique in order to solve max-clique problems. This application requires an ordering of the vertices of the graph and the following propositions.

Proposition 4.1. *For a graph* $G = (V, E)$ *and some vertex* $0 \leq i < |V|$ *where* $A = G[\{i+1, ..., |V|-1\}]$, $B = G[\{i, i+1, ..., |V|-1\}]$, *let* C_A *be a max-clique in* A *and* C_B *be a max-clique in* B. *If* $|C_B| > |C_A|$ *then* $i \in C_B$

Proof. Graph A is an induced subgraph of B since A contains a subset of the verities in B and therefore any clique found in A must also be a valid clique in B. Since $|C_B| > |C_A|$, $C_B \not\subseteq \{i+1, ..., |V|-1\}$ as if it were, it would also be a clique in A and could not be larger than C_A. Since $C_B \subseteq \{i, i+1, ..., |V|-1\}$ and $C_B \not\subseteq \{i+1, ..., |V|-1\}$ we conclude that $i \in C_B$.

□

Proposition 4.2. *For a graph $G = (V, E)$ and some vertex $0 \leq i < |V|$ where $A = G[\{i+1, ..., |V|-1\}]$, $B = G[\{i, i+1, ..., |V|-1\}]$, let C_A be a max-clique in A and C_B be a max-clique in B. If $|C_B| > |C_A|$ then $|C_B| = |C_A| + 1$*

Proof. By Proposition 4.1, $i \in C_B$ so $|C_B \setminus \{i\}| = |C_B| - 1$. In addition, $C_B \setminus \{i\}$ must be a clique in A and since C_A is a max-clique in A, $|C_B| - 1 = |C_B \setminus \{i\}| \leq C_A$. Thus $|C_B| \leq |C_A| + 1$. Therefore if $|C_B| > |C_A|$ then $|C_B| = |C_A| + 1$.

□

In a normal max-clique backtracking search, the first branch of the search is over the largest search space, with all vertices being in the initial candidate set. The search space then gets progressively smaller at each successive step. With the exception of the size of the maximum clique found, no knowledge of structure is kept by the backtracking algorithm and each step is run in near isolation from the previous. Russian doll search (Algorithm 8) instead attempts to use solutions to smaller sub-searches to more effectively prune later searches, starting with the smallest possible problem, a search over just one vertex, and iteratively enlarging the search space at each step. This is done in reverse order of the vertex labels, starting with only the largest labelled vertices and growing towards the lowest.

An initial empty max-clique is created as well as an array $dynamicValues$, where $dynamicValues[i]$ records the largest clique that can be found in the induced subgraph $G_{\{i,...,maxVal\}}$. Starting at the smallest possible sub-problem of a single vertex, the search is iteratively grown to include more vertices in each search. Step i searches for the largest clique that can be found in $G[\{i, ..., maxVal\}]$. From Proposition 4.1, we know that the only way we can grow the clique from one seen in a previous sub-problem search is if $i \in C$, for $G[\{i, i+1, ..., |maxVal|\}]$. Instead of initializing this search with the empty clique as we would do in backtracking, we can instead initialize it with the clique containing i. This restricts the search and improves the time taken to search each sub-problem. The candidate set is restricted to $\{i+1, ..., maxVal\}$.

Algorithm 8 Max-clique Russian Doll Search

```
1:  procedure RussianDollRecurse(C, U, step)
2:      if |U| = 0 then
3:          if |C| > |maxClique| then
4:              maxClique ← C
5:          return |maxClique|
6:      while U ≠ ∅ do
7:          j ← min{U}
8:          if dynamicValues[j] + |C| ≤ |maxClique| then
9:              return |maxClique|
10:         U ← U \ {j}
11:         X ← GraphFilter(C, j, U)
12:         C ← C ∪ {j}
13:         if |C| + |X| ≤ |maxClique| then
14:             return |maxClique|
15:         max ← RussianDollRecurse(C, X, step)
16:         IF max > dynamicValues[step + 1] THEN
17:             RETURN max
18:         C ← C \ {j}
19:     RETURN |maxClique|

1:  procedure RussianDollSearch(Graph)
2:      Global maxClique ← ∅
3:      Global dynamicValues ← int[|Graph.Verts|]
4:      maxVal ← |Graph.Verts| − 1
5:      for i = maxVal downto 0 do
6:          cands ← GraphFilter({}, i, {(i + 1), ..., maxVal})
7:          dynamicValues[i] ← RussianDollRecurse({i}, cands, i)
8:      RETURN dynamicValues[0]
```

The recursive section of this algorithm largely resembles the backtracking search in Algorithm 6. The algorithm is passed the current clique being examined C, and the current candidate set U. U is first checked to determine if it is empty, and if it is, we check the size of the current clique against the largest clique found so far. If our new clique is larger, we record it as the new max-clique before returning the size of the max-clique. While the candidate set is not empty we first remove the minimal member j from the candidate set. We then perform the Russian doll bound (line 8). From our previous search of the subgraph $G[\{i, ..., maxVal\}]$ we know that $dynamicValues[j]$ is the largest clique that can be formed using vertices $\{j, ..., maxVal\}$. If $|C| + dynamicValues[j] \leq |\text{maxClique}|$ we know that we can not create a new max-clique using from this branch of the search and so the branch is pruned. If this is not the case then a new candidate set X is created by filtering out all members of the candidate set which do not form an edge with i. Vertex i is then added to the clique. The size bound is then verified and the search is then repeated for the new clique and candidate sets. If one of these searches returns a max-clique larger than the one found in the previous step, the entire recursion of the current step can be terminated, since from Proposition 4.2 we can only grow the max-clique by at most one element in each step.

Östergård performed experiments using both his algorithm and a regular backtracking search algorithm across 2 data sets. The first data set used was random graphs. These graphs varied across edge density and the number of vertices. In this experiment, edge density specifies the percent of all possible edges that are included in the graphs. Östergård's results show a near universal decrease in total computation time over the existing backtracking search algorithm on these random graphs. In some larger graphs, this difference was as large as 45% when comparing the total runtime of each search algorithm over an identical graph. In addition to random graphs, Östergård used a set of DIMACS benchmarking graphs [16], and algorithms by Wood [20], and Xue [1] to analyse the performance of his algorithms. On many of the benchmarking graphs the performance improvement was substantial, however this was not universal. For some specific graph structures, such as Hamming graphs, the performance of the Russian doll

algorithm fell behind that of one or both of the other algorithms.

5

Max-Clique Search in Circulant Hypergraphs

A great deal of study has gone into the search for maximum cliques on graphs, however this is not the case for max-cliques in hypergraphs. This chapter presents the modifications needed for the Backtracking and Russian Doll algorithms to work with hypergraphs, a technique for searching for cliques in circulant hypergraphs using necklaces, as well as a new algorithm for searching for cliques in circulant hypergraphs.

5.1 Filtering

When performing a graph search for cliques, the filtering algorithm takes only the newest element being added and the candidate set. Since this was done for each element in the clique when it was added, and no new element has been added to the candidate set, it is unnecessary to check the candidate set against the elements in the clique. For a clique

of size C and a candidate set of size X, the filtering operation runs in $O(X)$ time.

For hypergraphs a similar concept can be used, however the clique formed must also be used in the filtering. For a k-hypergraph, all sets of $k-1$ elements from the set have been checked with every element of the candidate set to ensure there exists an edge consisting of those elements. When the newest element is added to the clique, we must check every $(k-2)$-subset of vertices in the clique with the new element and each possible vertex of the candidate set. With the same parameters as before, the filtering algorithm runs in $O(C^{k-2}X)$ time. The effect of this is a considerably slower search for maximum cliques than the one for graphs with the same number of vertices and density. Chapter 6 will deal with the structure and bounding that can be done within the candidate filtering to reduce the time of this operation.

Algorithm 9 shows the filtering algorithm on a k-hypergraph. This filtering does not involve bounding as it was seen in Algorithm 7. The algorithm is given a clique C, new element i, and candidate set U. Each element x of U is iterated over, and for each $(k-2)$-subset of C an edge check is performed with this $(k-2)$-subset, i, and x. If this edge is not present, x cannot be included in the clique and it is discarded. If every $(k-2)$-subset shares an edge with i and x then x is added to the new candidate set.

Algorithm 9 Hypergraph Filter

```
1:  Global H = (V, E), k = |x ∈ E|
2:  procedure HyperGraphFilter(C, i, U)
3:      X ← ∅
4:      for x ∈ U do
5:          canAdd ← true
6:          for each (k − 2)-subset a of C do
7:              if a ∪ {i, x} ∉ E then
8:                  canAdd ← false
9:                  break
10:         if canAdd then
11:             X ← X ∪ {x}
12:     return X
```

5.2 Adapting Clique Searches For Hypergraphs

When initializing the search for cliques in Algorithm 6 the clique is initially empty when the first recursive call is made, and in Algorithm 8 a single vertex is used for the clique. In both cases, filtering is not done until a single vertex has been added to the clique. From this single element, every vertex in the candidate set can be easily filtered. When initializing a hypergraph search however, the filtering algorithm requires that at least $k-1$ elements be present in the clique. Two approaches can be used to accommodate for this. The first, being the naive approach, is to simply not perform any filtering if $k-1$ elements are not in the clique yet. This approach necessitates enumerating all $k-1$ sets of the candidate set, many of which will not create edges. A faster technique is to iterate through the list of edges and initialize each run to begin with an edge. A special filtering is done on the candidate set to run through all $(k-1)$ subsets of this initial clique of size k, to ensure that each element in the candidate set is compatible with the initial clique.

Algorithm 10 Initial Filter

```
1:  Global k
2:  procedure InitialFilter(C, U)
3:      X ← ∅
4:      for x ∈ U do
5:          canAdd ← true
6:          for each (k − 1)-subset a of C do
7:              if !isEdge(a ∪ {x}) then
8:                  canAdd ← false
9:                  break
10:         if canAdd then
11:             X ← X∪ {x}
12:     return X
```

In the hypergraph Backtracking search (Algorithm 11), we begin by iterating through the edge set and setting the clique to be the current edge. We create an initial candidate set by taking all vertices greater than the largest vertex in the current edge. This is done to prevent searching the same space in multiple iterations. This candidate set is filtered using the initial filtering algorithm, and the recursive search is called. With the

exception of the filtering algorithm called, the recursive search remains unchanged from the graph backtracking search.

Algorithm 11 Clique in Hypergraph Backtracking Search

```
1:  procedure BacktrackRecurse(C, U)
2:      if |U| = 0 then
3:          if |C| > |maxClique| then
4:              maxClique ← C
5:          return
6:      while U ≠ ∅ do
7:          i ← min(U)
8:          U ← U \ {i}
9:          X ← HyperGraphFilter(C, i, U)
10:         C ← C ∪ {i}
11:         if |C| + |X| ≤ |maxClique| then
12:             return
13:         BacktrackRecurse(C, X)
14:         C ← C \ {i}
15:     return
1:  procedure BacktrackingSearch(Hypergraph H = (V, E),k)
2:      Global maxClique ← ∅
3:      Global edgeSize ← k
4:      n ← |V|
5:      for e ∈ E do
6:          C ← e
7:          U ← {max(e) + 1, ..., n − 1}
8:          X ←initialFilter(C, U)
9:          BacktrackRecurse(C, X)
10:     return |maxClique|
```

The Russian Doll algorithm (Algorithm 12) requires slightly more alteration than the Backtracking algorithm. We must first initialize some values of the $dynamicValues$ array. We set positions $n-1-(k-2)$ through $n-1$ to $k-1, k-2, ..., 1$. Since we do not perform actual searches of cliques this small, we need to initialize these values to prevent failures in the Russian doll bound during the recursive step. At step i we take the loop over the edges which contain i and use each edge to begin the recursive search. An initial clique is formed using this initial edge e, a candidate set is created by filtering the set $\{\max(e)+1, ..., n-1\}$, and the recursive search is called. In the graph version, we only needed to break from the recursion in the case where we have grown the maxclique in this step. In hypergraphs we must also break from the loop which is iterating over the edge set and calling the recursive function.

Algorithm 12 Clique in Hypergraph Russian Doll Search

```
1:  procedure RussianDollRecurse(C, U, step)
2:      if |U| = 0 then
3:          if |C| > |maxClique| then
4:              maxClique ← C
5:          return |maxClique|
6:      while U ≠ ∅ do
7:          j ← min(U)
8:          if dynamicValues[j] + |C| ≤ |maxClique| then
9:              return |maxClique|
10:         U ← U \ {j}
11:         X ← HyperGraphFilter(C, j, U)
12:         C ← C ∪ {j}
13:         if |C| + |X| ≤ |maxClique| then
14:             return |maxClique|
15:         max ← RussianDollRecurse(C, X, step)
16:         if max > dynamicValues[step + 1] then
17:             return max
18:         C ← C \ {j}
19:     return |maxClique|

1:  procedure RussianDollSearch(H = (V, E),k)
2:      Global maxClique ← ∅
3:      Global dynamicValues ← int[|V|]
4:      for i = 0 to k − 1 do
5:          dynamicValues[maxVal − i] ← i + 1
6:      for i = ((n − 1) − edgeSize) downto 0 do
7:          S ← {e ∈ E : min(e) = i}
8:          for e ∈ S do
9:              C ← e
10:             U ← {(max(e) + 1), ..., (n − 1)}
11:             X ←initialFilter(e, U)
12:             if RussianDollRecurse(e, X, i) > dynamicValues[i + 1] then
13:                 break
14:         dynamicValues[i] ← |maxClique|
15:     return dynamicValues[0]
```

5.3 Necklace Search

Although both previous algorithms work to generate max-cliques, neither make use of the circulant properties of the graphs being studied. When generating cliques using normal backtracking search, care is taken to avoid searching for isomorphic results. This is normally achieved through a search in lexicographical order. For circulant graphs and hypergraphs a large number of isomorphic cliques will exist for any one clique found, as all rotations of that clique are themselves cliques, and so these isomorphic searches must be avoided.

One technique that can be used to avoid searching these isomorphic search spaces is to search only subsets which are canonical subset necklaces. From Proposition 3.26, we know that all rotations of a clique must also be valid cliques and therefore for any clique C of maximum size, all rotations of C must also be cliques of maximum size. By searching only over the cliques which are canonical subset necklaces we are able to remove isomorphic branches from the search tree, which we know will never yield new optimal results.

The Necklace search algorithm was created by Tzanakis et al. [17] as an optimized way to search for cliques in hypergraphs using the technique to generate canonical subset necklaces. Algorithm 5 which generates all canonical subset necklaces follows a similar structure to the basic backtracking algorithm. It branches over all subsets which can be created from an initial set, using a set of candidate values, and prunes branches which cannot form a canonical subset necklace. In order to search only for max-cliques we further restrict the algorithm to search over canonical necklaces which are cliques. This is done through the filtering of candidates which will not construct a valid clique, and by filtering results which will not form a larger clique than the largest found so far.

To initialize the Necklace search algorithm (Algorithm 13) an initial empty max-clique is stored. For each edge $e \in E$, e is checked to verify if it is a canonical subset

Algorithm 13 Max-clique Necklace Search

```
1: procedure NecklaceRecurse(C, U)
2:     if |U| = 0 then
3:         if |C| > |maxClique| then
4:             maxClique ← C
5:         return
6:     while U ≠ ∅ do
7:         i ← min(U)
8:         U ← U \ {i}
9:         if not(isNecklace(C ∪ {i})) then
10:            break
11:        X ← HyperGraphFilter(C, i, U)
12:        C ← C ∪ {i}
13:        if |C| + |X| ≤ |maxClique| then
14:            return
15:        NecklaceRecurse(C, X)
16:        C ← C \ {i}
17:    return
1: procedure NecklaceSearch(Hypergraph H = (V, E),k)
2:     Global maxClique ← ∅
3:     Global edgeSize ← k
4:     n ← |V|
5:     for e ∈ E do
6:         C ← e
7:         if not(isNecklace(C)) then
8:             continue
9:         U ← {max(e) + 1, ..., n − 1}
10:        X ←initialFilter(e, U)
11:        NecklaceRecurse(C, X)
12:    return |maxClique|
```

necklace, and if so it is used as an initial clique C. An initial candidate set U is formed from $\{\max(e)+1, ..., n-1\}$ and filtered to form a candidate set X. The recursive portion of the algorithm is called with C and X.

In the recursive section of the algorithm, the algorithm is passed the current clique C, and the current candidate set U. As with regular backtracking, at this point all elements of the candidate set are viable members of the clique. U is first checked to determine if it is empty, if it is we compare the size of the current clique against the largest clique found so far. If our new clique is larger, we record it before returning. While the candidate set is not empty we first remove the minimal member i from the candidate set. We the check to see if the subset $C \cup \{i\}$ is a canonical subset necklace using Algorithm 2. If it is we can continue, if not we can prune the branch, as no further children of this branch can be necklaces, as was shown in Algorithm 4. We form a new candidate set X by filtering out all members of the candidate set which do not form an edge with i and all $(k-2)$ subsets of C. Vertex i is then added to the clique. At this point we check the size bound and prune the branch if our candidate set has insufficient elements to produce a new max-clique. The search is then repeated for the new clique and candidate sets.

5.4 Russian Necklace Algorithm

The goal of this work has been to build upon previous algorithms to produce a new algorithm that takes advantage of properties of each, to combine the advantages of Russian doll search with the isomorphic space reduction of the necklace search algorithm. The two techniques however are not initially compatible with each other. In the Necklace algorithm (Algorithm 13) the whole vertex set is considered while generating the max-clique, this allows us to compare rotations of the clique easily to verify that our current clique is a canonical subset necklace, and by extension, our final clique will be a canonical subset necklace. In the Russian doll algorithm, we only consider a subgraph, and as such many rotations of the current clique will include vertices not considered in the current subgraph. If we were to restrict ourselves to searching over only cliques which are canonical subset necklaces, then we would not be able to solve these subproblems to

use in the Russian doll bound. We are able to adapt how we handle rotations however by considering only right direction rotations which do not place elements into the set of vertices outside of the subgraph. Before presenting the new algorithm we must first present two new properties of cliques within circulant k-hypergraphs.

We denote by $H[S]$ the subgraph of $H = (V, E)$ induced by vertices $S \subseteq V$, that is $H[S] = (S, E(S))$ where $E(S) = \{e \in E : e \subseteq S\}$

Proposition 5.1. *Let $H = (V = \{0, 1, 2, ..., n-1\}, E)$ be a circulant k-hypergraph. Let $0 \leq i < n-1$. Let C_j be a maximum clique of $H[\{j, j+1, ..., n-1\}]$ where $0 \leq j \leq n-1$. If $|C_i| > |C_{i+1}|$ then $i \in C_i$ and $n-1 \in C_i$.*

Proof. $C_i \subseteq \{i, i+1, ..., n-1\}$ and since $|C_i| > |C_{i+1}|$, $C_i \not\subseteq \{i+1, ..., n-1\}$. It follows that $i \in C_i$. We assume that $n+1 \notin C_i$. $C_i \subseteq \{i, i+1, ..., n-2\}$ and $R(C_i, 1) \subseteq \{i+1, ..., n-1\}$. From Proposition 3.26, $R(C_i, 1)$ must be a clique in H, and since $R(C_i, 1) \subseteq \{i+1, ..., n-1\}$, $R(C_i, 1)$ must be a clique in $H[\{i+1, ..., n-1\}]$. This leads to a contradiction as $|R(C_i, 1)| > |C_{i+1}|$, where C_{i+1} is the largest clique in $H[\{i+1, ..., n-1\}]$. □

Proposition 5.2. *Let $H = (V = \{0, 1, 2, ..., n-1\}, E)$ be a circulant k-hypergraph. Let C_j be a maximum clique of $H[\{j, j+1, ..., n-1\}]$ where $0 \leq j \leq n-1$. Let $0 \leq i < n-1$. If $|C_i| > |C_{i+1}|$ then there does not exists a pair of vertices x, y where $0 \leq x < y < i$ such that $y - x \geq i$ and $C_j \cap \{x, x+1, ..., y\} = \emptyset$.*

Proof. We assume that there exists a pair of elements x, y where $0 \leq x < y < i$ such that $y-x \geq i$ and $C_i \cap \{x, x+1, ..., y\} = \emptyset$. We let $j = y-x$. We take the rotation $A = L(C_i, x)$. From Proposition 3.26 set A must be a clique in H. Since $C_i \cap \{x, x+1, ..., y\} = \emptyset$, $A \cap \{0, 1, ..., j+1\} = \emptyset$ and A is a clique in $H[\{j+1, j+2, ..., n-1\}]$. Since $j \geq i$ $H[\{j+1, j+2, ..., n-1\}]$ is a subgraph of $H[\{i+1, i+2, ..., n-1\}]$, and so A must be a clique in $H[\{i+1, i+2, ..., n-1\}]$. This contradicts the fact that $|C_i| = |A| > |C_{i+1}|$, and therefore there does not exists a pair of vertices x, y where $0 \leq x < y < i$ such that $y - x \geq i$ and $C_j \cap \{x, x+1, ..., y\} = \emptyset$. □

We now present the new algorithm that combines the structure of the Russian doll algorithm with the isomorphism avoidance of the Necklace algorithm. In the remainder

of this paper this algorithm will be known as the Russian Necklace algorithm (Algorithm 15). The basis of the Russian Necklace algorithm is Algorithm 12: Max-clique Russian Doll Search in that it is searching over iteratively larger subgraphs in order to more effectively prune later searches. The first improvement over the Russian doll algorithm is the fixing of a second element within each search. In the original Russian doll algorithm we were able to fix the element i in the search at step i. From proposition 5.1, we know that in every step we must include vertex $n-1$ in order to grow the size of the maximum clique. This forced element prevents searching for results which are isomorphic to ones searched in step $i+1$.

The second improvement allows us to better prune branches which will not yield optimal results. At any given step i, Proposition 5.2 tells us that in order for our algorithm to improve the result of the previous step we cannot have a set of $i+1$ sequential vertices not in our final clique. Instead of simply maintaining this property in the clique during the search, we can extend this to end searches where this set of sequential missing elements exist in the candidate set. During any recursive step, we assume that the final clique generated will include all elements of the current clique, as well as all elements of the current candidate set. If all elements of the candidate set are included in the eventual clique, then we know no set of $i+1$ sequential elements can exist within the candidate set. Likewise if some elements of the candidate set will be later filtered out in a later step, any set of $i+1$ sequential elements missing from the current candidate set will be missing from its filtered subset. When this filtered subset is eventually added to the clique, a sequence of at least $i+1$ sequential elements will be missing from the clique. For the remainder of this work, this bound will be referred to as the gap bound. Algorithm 14 demonstrates how this bounding is done given a clique and candidate set. This algorithm assumes that vertex $n-1$ is in the clique from the first improvement.

As with the Russian Doll algorithm, an initial empty max-clique is created as well as an array $dynamicValues$, where $dynamicValues[i]$ records the largest clique that can be found in the induced subhypergraph $H[\{i, ..., maxVal\}]$. Starting at the small-

Algorithm 14 Largest Candidate Gap

```
1:  procedure LargestGap(C, U)
2:      A ← C \ {n − 1}
3:      x ← (min(U) − max(A)) − 1
4:      prev ← A[|A| − 1]
5:      for curr = 0 to |U| − 1 do
6:          y ← (U[curr] − U[prev] − 1)
7:          x ← max(x, y)
8:          prev ← curr
9:      y ← ((n − 1) − max(U) − 1)
10:     x ← max(x, y)
11:     return x
```

est possible sub-problem of a single vertex, the search is iteratively grown to include more vertices in each search. Step i searches for the largest clique that can be found in $G[\{i, ..., maxVal\}]$. We initialise the clique to be an edge e which has smallest element i and add $n-1$ if it is not already in e, as from Proposition 5.1 we know that in order to grow the clique vertices i and $n-1$ must be included. We perform the initial filtering of the candidate set $\{i+1, ..., n-2\}$ and begin the recursive step.

The recursive algorithm is passed the current clique being examined C, and the current candidate set U. Set U is first checked to determine if it is empty, in which case we compare the size of the current clique against the size of the largest clique found so far. If our new clique is larger, we record it as the new max-clique before returning the size of the max-clique. We then check for the largest gap among the vertices under consideration. Using Algorithm 14 we find the largest gap that exists between elements, and if that is larger than the current step due to Proposition 5.2, we can prune the branch by returning. While the candidate set is not empty we first remove the minimal member j from the candidate set. We then perform the Russian doll bound and prune the branch if appropriate. At line 18, we perform a simpler version of the gap bound. We look at only the last element added to the clique and the current vertex j. If this gap is larger than the step number we know that this will cause a pruning at the recursive call, and all subsequent iterations of the current loop will have larger gaps; in this case we do not have to search any farther in this branch and can return. If this is not the case, then a

Algorithm 15 Max-clique Russian Necklace Search

```
1: procedure RussianNecklaceRecurse(C, U, step)
2:     if |U| = 0 then
3:         if |C| > |maxClique| then
4:             maxClique ← C
5:         return |C|
6:     if LargestGap(C, U) > step then
7:         return |C|
8:     x ← max(C \ {n − 1})
9:     while U ≠ ∅ do
10:        j ← min(U)
11:        if dynamicValues[j] + |C| ≤ |maxClique| then
12:            return |C|
13:        U ← U \ {j}
14:        X ← HyperGraphFilter(C, j, U)
15:        C ← C ∪ {j}
16:        if |C| + |X| ≤ |maxClique| then
17:            return |C|
18:        if (j − x) > step then
19:            return |C|
20:        max ← RussianNecklaceRecurse(C, X, step)
21:        if max > dynamicValues[step + 1] then
22:            return max
23:        C ← C \ {j}
24:    return |maxClique|

1: procedure RussianNecklaceSearch(H = (V, E),k)
2:     Global maxClique ← ∅
3:     Global dynamicValues ← int[|V|]
4:     n ← |V|
5:     for i = 0 to k − 1 do
6:         dynamicValues[n − 1 − i] ← i + 1
7:     for i = (n − 1 − k) downto 0 do
8:         S ← {e ∈ E : min(e) = i}
9:         for e ∈ S do
10:            C ← e ∪ {n − 1}
11:            U ← {(max(e) + 1), ..., n − 2}
12:            X ← initialFilter(e, U)
13:            if RussianNecklaceRecurse(C, X, i) > dynamicValues[i + 1] then
14:                break
15:        dynamicValues[i] ← |maxClique|
16:    return dynamicValues[0]
```

new candidate set X is created by filtering out all members of the candidate set which do not form an edge with i and all $(k-2)$-subsets of C. Vertex i is then added to the clique. The size bound is then performed and the search is repeated for the new clique and candidate sets. If one of these searches returns a max-clique larger than the one found in a previous step, the entire recursion of the current step can be terminated, since by Proposition 4.2, we can only grow the max-clique by at most one element at each step. In addition to this, we break from the loop over the edge set, as each edge iteration of this loop is in the same overall step of the algorithm.

The remaining chapters of this work will deal with the optimization of the filtering algorithm, and the performance of the search algorithms across a variety of sizes k values and densities.

6

Filtering Optimization and Experiments

This chapter presents two main structures which can be used when performing the filtering operation in the max-clique search using the previous algorithms. In addition to this, it presents various bounds which can be applied to the filters, as well as compares the results of these bounds and filter structures in experimental tests.

In a graph search, the act of filtering the candidate set based on the newest element added to the clique is straightforward and efficient. The candidate set is looped through, and if there is an edge with the newest vertex to be added to the clique, then the vertex is kept, otherwise it is discarded. We do not need to do any comparisons with any other elements of the clique, as they were compared with the candidate at the step where they were added to the clique. In hypergraphs however this is not the case. Consider a 4-hypergraph. For a set of vertices to be considered a clique, all possible 4-subsets of

vertices within this set must form an edge. If at some stage in our algorithm we have our newest element to be added x, and some clique and candidate set, C and S, the check for whether any candidate can be added to the new candidate set must involve combinations of both the new element and the elements already in the set. Precisely, all pairs from C must be checked with both x and every element in the candidate set. More generically for any k-hypergraph, the filtering operation must check every $(k-2)$-subset of S together with x, checking for a joint edge with every candidate in the candidate set. An operation which previously took $O(|S|)$ time now takes $O(|S||C|^{k-2})$ time.

For all backtracking clique searches considered, the most expensive step is in the main loop of the filtering operation. For this reason multiple orderings and techniques of performing this filtering were analysed. These broadly fell into two main categories: candidate-major and clique-major where each denotes the outer section of the loop.

6.1 Filtering Algorithms

In a candidate-major filter (Algorithm 16), the outermost loop iterates over the candidate set, and for every candidate, iterates over all possible $(k-2)$-subsets of the clique and verifies if an edge is present consisting of: the $(k-2)$-subsets set, the newest element added, and the candidate. Once a missing edge has been found, or all $(k-2)$-subsets sets of the candidate set are checked, the candidate is added to a new candidate set. Once all candidates have been checked this new candidate set is returned.

In a clique-major filtering (Algorithm 17), the opposite is done. The algorithm iterates over each $(k-2)$-subset of the clique, and for each of these, iterates over the entire candidate set. If the edge does not exists for this new candidate set then the candidate is removed from the candidate set.

Algorithm 16 Candidate-Major Filter Base

```
1:  Global H = (V, E), k = |x ∈ E|
2:  procedure HyperGraphFilter(C, i, U)
3:      X ← ∅
4:      for x ∈ U do
5:          canAdd ← true
6:          for each (k − 2)-subset a of C do
7:              if a ∪ {i, x} ∉ E then
8:                  canAdd ← false
9:                  break
10:         if canAdd then
11:             X ← X∪ {x}
12:     return X
```

Algorithm 17 Clique-Major Filter

```
1:  Global H = (V, E), k = |x ∈ E|
2:  procedure CliqueMajorFilter(C, i, U)
3:      X ← U
4:      for each (k − 2)-subset a of C do
5:          for x ∈ X do
6:              if a ∪ {i, x} ∉ E then
7:                  X ← X \ {x}
8:      return X
```

In addition to the above structural consideration, there also exists a need to provide pruning within the filtering algorithm itself. By including certain operations in the filtering, the entire filtering algorithm can be exited once it becomes clear that any result will provide insufficient candidates to generate a new max-clique.

6.1.1 Size Bound

The first adaptation to the filtering algorithms above is to include the size bound in the filter. In both the candidate-major (Algorithm 16) and clique-major (Algorithm 17) algorithms $m = (|C| + |U| + 1) - |\text{maxClique}|$ is calculated to be the number of vertices that need to be removed from the candidate set to cause a size bounding to occur, that is, the combined size of the current clique and candidate set are compared to the largest clique found by the algorithm so far. We also initialize a counter r to record the number

of rejected vertices. When a new element is removed from the candidate set due to its inability to form an edge with a $(k-2)$-set of the clique, we increment r and check to see if $r \geq m$. If this is the case, no further filtering of the candidate set is required, as the new candidate set will never be large enough to form a new max-clique, so we return an empty candidate set. This filtering algorithm can be used with any of the four search algorithms in this work, as a size bound is present in all.

Algorithm 18 Candidate-Major Filter Size

```
1:  Global H = (V, E), maxClique, k = |x ∈ E|
2:  procedure HyperGraphFilter(C, i, U)
3:      X ← ∅
4:      m ← (|C| + |U| + 1) − |maxClique|
5:      r ← 0
6:      for x ∈ U do
7:          canAdd ← true
8:          for each (k − 2)-subset a of C do
9:              if a ∪ {i, x} ∉ E then
10:                 canAdd ← false
11:                 break
12:         if canAdd then
13:             X ← X∪ {x}
14:         else
15:             r ← r + 1
16:             if r ≥ m then
17:                 return ∅
18:     return X
```

Algorithm 19 Clique-Major Filter Size

```
1:  Global H = (V, E), k = |x ∈ E|, maxClique
2:  procedure HyperGraphFilter(C, i, U)
3:      X ← U
4:      m ← (|C| + |U| + 1) − |maxClique|
5:      r ← 0
6:      for each (k − 2)-subset a of C do
7:          for x ∈ X do
8:              if a ∪ {i, x} ∉ E then
9:                  X ← X \ {x}
10:                 r ← r + 1
11:                 if r ≥ m then
12:                     return ∅
13:     return X
```

6.1.2 Russian Doll Bound

Applying the Russian doll bound in the filtering operation works differently in each of the candidate-major and clique-major filtering techniques. In the candidate-major structure (Algorithm 20) we record a boolean variable *first* to record if we have added a vertex to the new candidate set yet. We start the filtering by looping through the candidate set U in ascending order. When we find a candidate that can be added to the new candidate set, we check if this is the first element to be added. If it is, we see what is the largest clique that can be formed with elements starting with this candidate and compare it the maxClique. If we can not improve on this result no further filtering is needed, as a Russian doll bound will occur after the filtering and the branch will be pruned. We return an empty candidate set to force the branch to terminate.

Algorithm 20 Candidate-Major Filter Russian Doll

```
1: Global H = (V, E), maxClique, k = |x ∈ E|
2: procedure HyperGraphFilter(C, i, U)
3:     X ← ∅
4:     first← true
5:     for x ∈ U do
6:         canAdd ← true
7:         for each (k − 2)-subset a of C do
8:             if a ∪ {i, x} ∉ E then
9:                 canAdd ← false
10:                break
11:        if canAdd then
12:            X ← X∪ {x}
13:            if first then
14:                if |C| + 1 + dynamicValues[x] > |maxClique| then
15:                    first ← false
16:                else
17:                    return ∅
18:    return X
```

In the clique-major structure (Algorithm 20), we perform a similar operation of recording and checking the first element to be added to the new candidate set. We do not however know at any step if the vertex will be added to the final candidate set, as by the time that this is known, almost all filtering has been performed. At each iteration of the inner loop, we track the first element to not be rejected during that loop, and consider it to be the first element of the final candidate set. This first element may be the first element in the real final candidate set, or it may be removed later, in either case no element smaller than it will be in the final candidate set and no larger element will have a higher value in dynamicValues and so it is used to perform the Russian Doll bound.

Algorithm 21 Clique-Major Filter Russian Doll

```
1:  Global H = (V, E), k = |x ∈ E|, maxClique
2:  procedure HyperGraphFilter(C, i, U)
3:      X ← U
4:      for each (k − 2)-subset a of C do
5:          first ← true
6:          for x ∈ X do
7:              if a ∪ {i, x} ∈ E then
8:                  if first then
9:                      if |C| + 1 + dynamicValues[x] > |maxClique| then
10:                         first ← false
11:                     else
12:                         return ∅
13:             else
14:                 X ← X \ {x}
15:     return X
```

Unlike the size bound, this bound can only be used in the Russian Doll search, and Russian Necklace search.

6.1.3 Gap Bound

The addition of the gap bound used in the Russian necklace algorithm can be used in the filtering algorithm as well. In both cases, we first assume that the gap bound has been performed for the clique itself in previous steps. In the candidate-major structure (Algorithm 22), we initialise a temporary variable l to initially be i, the newest element added to the clique. As we loop through the candidate set in ascending order, when we know that a vertex x can be added, we check the distance between x and l. If it is greater than the current step of the algorithm, we return the empty set to cause a pruning of the search. Otherwise x becomes the new l value and we continue. In a clique-major structure (Algorithm 23) this is done for every temporary candidate set, as the gaps in the candidate set can only grow as further filtering steps occur.

Algorithm 22 Candidate-Major Filter Russian Gap

```
1:  Global H = (V, E), step, k = |x ∈ E|
2:  procedure HyperGraphFilter(C, i, U)
3:      X ← ∅
4:      l ← i
5:      for x ∈ U do
6:          canAdd ← true
7:          for each (k − 2)-subset a of C do
8:              if a ∪ {i, x} ∉ E then
9:                  canAdd ← false
10:                 break
11:         if canAdd then
12:             X ← X ∪ {x}
13:             if x − l − 1 > step then
14:                 return ∅
15:             else
16:                 l ← x
17:     return X
```

Algorithm 23 Clique-Major Filter Gap

```
1:  Global H = (V, E), k = |x ∈ E|, step
2:  procedure HyperGraphFilter(C, i, U)
3:      X ← U
4:      for each (k − 2)-subset a of C do
5:          l ← i
6:          for x ∈ X do
7:              if a ∪ {i, x} ∈ E then
8:                  if x − l − 1 > step then
9:                      return ∅
10:                 else
11:                     l ← x
12:             else
13:                 X ← X \ {x}
14:     return X
```

As these bounds within the filtering act independently of each other, multiple bounds can be used in the same filtering algorithm by combining the algorithms together, as long as they follow the same overall structure of candidate-major or clique-major. These combinations are: size and Russian doll, size and gap, gap and Russian doll, size Russian doll and gap.

6.2 Experimental Results

Each of the 4 algorithms discussed in this work (backtracking, necklace search, Russian doll search, and Russian necklace search) were analysed with all possible ways of incorporating bounds within the filtering operation. Using 4-hypergraphs as an example set, each algorithm was run with all possible filtering operations at 20%, 40%, 60%, and 80% densities. For each hypergraph, four different hypergraph sizes were used. Two main points of comparison are used for determining algorithm performance: the number of times a set of vertices was checked to determine if it is an edge, and the total time of the search. In the tables below the following notation is used. The filtering base (candidate or clique) is given as well as a set of letters from S,R,G. The letters represent if the size bound, Russian doll bound, and gap bound respectively are included in the filtering. Within the table the number of edge checks are compressed to save space in larger searches. For these values, B represents billions, while M represents millions.

6.2.1 Backtracking Search

For the Backtracking search at 20% density, Table 1, the inclusion of the size bound yielded the better results than the basic filters. There is a universal reduction in both the time, and number of edges checked when using this filtering technique. Between candidate-major and clique-major filtering (with size bound) the number of edges checked in the clique-major filtering is more than in candidate-major filtering, and the time is generally higher (with the exception of 125 vertices).

Table 1: Backtracking Filters: density=0.2

Vertices	75		100		125		150	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	1.407M	0.06	5.136M	0.20	14.008M	0.61	32.473M	1.84
Candidate+S	1.304M	0.08	4.813M	0.20	13.233M	0.62	30.866M	1.76
Clique	1.407M	0.09	5.136M	0.23	14.008M	0.66	32.473M	2.14
Clique+S	1.361M	0.08	4.989M	0.23	13.653M	0.59	31.748M	1.89

At 40% density, Table 2, similar results are seen, size bounds within the filter reduce time and edge checks, while the use of the candidate as the major loop element shows a greater reduction than the use of the clique. In the final test the clique-major filter was fastest, however this margin is only 1.3%.

Table 2: Backtracking Filters: density=0.4

Vertices	75		100		125		150	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	3.680M	0.12	15.236M	0.58	45.152M	2.01	111.546M	5.74
Candidate+S	3.283M	0.11	14.086M	0.56	42.518M	1.95	106.274M	5.44
Clique	3.680M	0.12	15.236M	0.67	45.152M	2.29	111.546M	5.58
Clique+S	3.473M	0.14	14.662M	0.66	43.871M	2.11	109.039M	5.37

When density is 60%, Table 3, we begin to see a large difference between the performance of the candidate-major and clique-major algorithms. In their base forms without bounding, they perform the same number of edge checks, however the time taken by the clique-major filter is much higher than the candidate-major one. As before, the size bound reduces the time and edge checks, and the best performance is from the candidate-major plus size filter.

Table 3: Backtracking Filters: density=0.6

Vertices	50		75		100		125	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	1.433M	0.03	11.771M	0.33	56.282M	1.86	184.376M	6.33
Candidate+S	1.298M	0.05	10.856M	0.33	48.425M	1.73	160.251M	5.99
Clique	1.433M	0.05	11.771M	0.44	56.282M	2.53	184.376M	8.35
Clique+S	1.336M	0.05	11.084M	0.44	50.485M	2.06	166.347M	6.46

In high density hypergraphs (80%), Table 4, the performance difference between candidate and clique-major algorithms becomes most apparent. Clique-major filters are universally slower than candidate-major. In the largest graphs, the clique-major filter took twice as long to search the same hypergraph.

Table 4: Backtracking Filters: density=0.8

Vertices	50		75		100		125	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	8.736M	0.19	95.554M	2.40	612.244M	14.52	2.530B	62.95
Candidate+S	6.739M	0.14	79.086M	2.04	518.526M	12.89	2.169B	59.05
Clique	8.736M	0.28	95.554M	2.96	612.244M	21.36	2.530B	138.36
Clique+S	7.020M	0.20	81.024M	2.50	528.217M	16.18	2.206B	101.70

6.2.2 Necklace Search

At 20% density, Table 5, the clique-major filtering technique yielded optimal results in runtime in both cases where no bounding and size bounding occured during the filter. As before, the inclusion of the size bound within the filter caused fewer edge verifications in both the clique-major and candidate-major versions. While the size bound clique-major version had more edge verifications, its runtime was lower than the candidate-major version.

Table 5: Necklace Filters: density=0.2

Vertices	275		300		325		350	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	7.881M	1.26	10.944M	1.76	14.838M	2.45	19.716M	3.26
Candidate+S	7.719M	1.23	10.737M	1.67	14.579M	2.40	19.398M	3.34
Clique	7.881M	1.19	10.944M	1.67	14.838M	2.36	19.716M	3.14
Clique+S	7.819M	1.19	10.866M	1.62	14.741M	2.26	19.600M	2.98

At 40% density, Table 6, the clique-major version of the unbounded filter showed a lower runtime than the candidate-major version. In smaller graphs, the clique-major plus size bound algorithm had lower runtimes than the corresponding candidate-major ones. In the largest hypergraph however the time performance was equal.

Table 6: Necklace Filters: density=0.4

Vertices	275		300		325		350	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	33.961M	5.26	48.225M	8.33	66.700M	11.53	90.251M	16.41
Candidate+S	33.272M	5.30	47.258M	8.11	65.369M	11.45	88.453M	15.65
Clique	33.961M	5.48	48.225M	8.30	66.700M	11.42	90.251M	16.11
Clique+S	33.570M	5.37	47.659M	8.50	65.902M	10.95	89.153M	15.68

At 60% density, Table 7, the candidate-major algorithm and its derivatives have passed the clique-major algorithms in runtimes. This gap is maintained throughout the varying graph sizes.

Table 7: Necklace Filters: density=0.6

Vertices	125		150		175		200	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	6.019M	0.64	13.557M	1.51	26.853M	3.24	50.267M	6.63
Candidate+S	5.476M	0.59	12.425M	1.44	24.810M	3.01	46.833M	6.21
Clique	6.019M	0.69	13.557M	1.61	26.853M	3.24	50.267M	6.72
Clique+S	5.619M	0.64	12.729M	1.56	25.379M	3.14	47.823M	6.32

In the final 80% density tests, Table 8, the performance of the candidate-major algorithm is the clear leader across all hypergraph sizes. Again the size bound reduces both edge checks and time across both filtering structures.

Table 8: Necklace Filters: density=0.8

Vertices	75		100		125		150	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	5.466M	0.33	26.416M	1.81	87.254M	6.63	236.474M	19.02
Candidate+S	4.889M	0.28	23.708M	1.65	79.028M	5.90	203.041M	17.24
Clique	5.466M	0.37	26.416M	2.01	87.254M	7.60	236.474M	21.14
Clique+S	4.982M	0.36	24.075M	1.78	80.145M	6.94	207.579M	18.55

When searching over low density hypergraphs using the Necklace search the size bounded clique-major filtering algorithm should be used. In denser hypergraphs the size

bounded candidate-major algorithm should be used.

6.2.3 Russian Doll Search

The Russian doll algorithm is the first time in which we see an additional check being added into the filtering. In this example we see both the Russian doll bound, the size bound, or both bounds used in the filtering. In 20% density hypergraphs, Table 9, the fastest searches were done with the clique-major filtering, despite having more edge checks in its bounded implementations. We also see a detriment to the algorithm's performance when the Russian doll bound is included. When the Russian doll bound is added to a filter with a size bound, the number of edge checks does not drop in any meaningful way. In addition to this, the added steps in the computation cause the overall search time to go up. When run as the only bound in the filter it does reduce the number of edges checked from the basic algorithm, however this reduction is less than seen with the size bound.

Table 9: Russian Doll Filters: density=0.2

Vertices	75		100		125		150	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	1.397M	0.06	5.105M	0.36	14.003M	1.31	32.466M	1.92
Candidate+R	1.397M	0.08	5.105M	0.34	14.003M	1.05	32.466M	1.90
Candidate+S	1.296M	0.08	4.785M	0.36	13.229M	1.15	30.860M	1.86
Candidate+SR	1.296M	0.06	4.785M	0.37	13.229M	1.14	30.860M	1.90
Clique	1.397M	0.09	5.105M	0.37	14.003M	1.14	32.466M	1.84
Clique+R	1.395M	0.11	5.099M	0.42	13.993M	0.76	32.448M	1.97
Clique+S	1.352M	0.09	4.959M	0.33	13.648M	0.67	31.741M	1.84
Clique+SR	1.352M	0.09	4.959M	0.27	13.648M	0.59	31.741M	1.81

In 40% density hypergraphs, Table 10, we see the performance of the clique-major and candidate-major structures become even. In the 100-vertex hypergraph the candidate-major structure shows the best performance. In the 125-vertex graph there is less than a 1% difference in many of the runtimes, and in the 150-vertex hypergraph there the clique-major structure has the lowest runtimes. In some cases, the inclusion of the Russian doll

bound with the size bound improved the overall runtime by up to 8%, however in the majority of the tests the addition of the Russian doll bound had either no effect on runtime, or increased it when compared to using only the size bound.

Table 10: Russian Doll Filters: density=0.4

Vertices	75		100		125		150	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	2.549M	0.12	14.998M	0.59	43.883M	2.26	110.392M	8.28
Candidate+R	2.548M	0.11	14.994M	0.56	43.876M	2.07	110.379M	5.57
Candidate+S	2.345M	0.11	13.900M	0.56	41.449M	2.22	105.305M	5.65
Candidate+SR	2.345M	0.11	13.900M	0.56	41.449M	2.06	105.305M	5.88
Clique	2.549M	0.14	14.998M	0.72	43.883M	2.32	110.392M	6.13
Clique+R	2.545M	0.14	14.982M	0.72	43.855M	2.34	110.339M	5.85
Clique+S	2.447M	0.12	14.452M	0.62	42.711M	2.12	107.987M	5.30
Clique+SR	2.447M	0.08	14.452M	0.62	42.711M	2.04	107.987M	5.37

At 60% density, Table 11, the candidate-major filter leads in performance across all tests. In half of tests, the inclusion of the Russian doll bounds improved the overall search time when combined with the size bound, while performing worse on its own against the size bound filters.

Table 11: Russian Doll Filters: density=0.6

Vertices	50		75		100		125	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	1.425M	0.05	11.766M	0.47	48.890M	1.90	164.786M	6.04
Candidate+R	1.424M	0.05	11.762M	0.48	48.875M	1.78	164.764M	6.18
Candidate+S	1.292M	0.03	10.852M	0.44	42.249M	1.68	143.822M	5.24
Candidate+SR	1.292M	0.05	10.852M	0.33	42.248M	1.67	143.821M	5.41
Clique	1.425M	0.06	11.766M	0.47	48.890M	2.28	164.786M	8.05
Clique+R	1.417M	0.06	11.723M	0.47	48.695M	2.36	164.307M	7.75
Clique+S	1.330M	0.05	11.081M	0.42	43.944M	1.76	149.050M	6.21
Clique+SR	1.330M	0.06	11.081M	0.42	43.944M	1.92	149.050M	5.97

At 80% density, Table 12, the candidate-major structure has the best overall runtimes by a considerable margin. The inclusion of the Russian doll bound improved the runtimes

in five of the eight tests where it was used with the size bound, but once again performed worse in all tests where it was the only bound used.

Table 12: Russian Doll Filters: density=0.8

Vertices	50		75		100		125	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	6.209M	0.14	89.709M	2.37	611.980M	19.77	2.511B	64.27
Candidate+R	6.184M	0.17	89.560M	2.01	611.456M	16.57	2.510B	64.52
Candidate+S	4.975M	0.14	74.786M	1.86	518.366M	15.94	2.154B	57.53
Candidate+SR	4.975M	0.12	74.785M	1.76	518.366M	13.03	2.154B	58.24
Clique	6.209M	0.23	89.709M	2.93	611.980M	21.79	2.511B	101.23
Clique+R	6.129M	0.20	89.077M	2.90	608.482M	22.06	2.500B	100.39
Clique+S	5.135M	0.16	76.524M	2.36	528.065M	17.27	2.191B	77.84
Clique+SR	5.135M	0.16	76.524M	2.45	528.065M	16.83	2.191B	75.57

6.2.4 Russian Necklace Search

The possible filtering combinations for the Russian necklace algorithm are the most extensive, with 16 possible filtering algorithms to use. At 20% density, Table 13, the bounding within the filters had a negligible, if any, effect on the number of edges checked. There was a corresponding negligible difference (<3%) in the runtimes between any two algorithms.

Table 13: Russian Necklace Filters: density=0.2

Vertices	275		300		325		350	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	1.199M	2.28	1.818M	4.45	2.794M	6.05	4.080M	8.53
Candidate+G	1.199M	2.31	1.818M	4.38	2.794M	6.10	4.080M	8.39
Candidate+R	1.199M	2.22	1.818M	4.37	2.794M	6.07	4.080M	8.49
Candidate+RG	1.199M	2.32	1.818M	4.35	2.794M	6.07	4.080M	8.41
Candidate+S	1.199M	2.29	1.818M	4.45	2.794M	6.08	4.080M	8.31
Candidate+SG	1.199M	2.31	1.818M	4.43	2.794M	6.12	4.080M	8.41
Candidate+SR	1.199M	2.28	1.818M	4.41	2.794M	6.01	4.080M	8.41
Candidate+SRG	1.199M	2.28	1.818M	4.34	2.794M	6.05	4.080M	8.46
Clique	1.199M	2.31	1.818M	4.41	2.794M	6.05	4.080M	8.44
Clique+G	1.199M	2.32	1.818M	4.32	2.794M	6.07	4.080M	8.42
Clique+R	1.199M	2.29	1.818M	4.37	2.794M	6.07	4.080M	8.44
Clique+RG	1.199M	2.29	1.818M	4.31	2.794M	6.08	4.080M	8.46
Clique+S	1.199M	2.32	1.818M	4.40	2.794M	6.07	4.080M	8.41
Clique+SG	1.199M	2.28	1.818M	4.32	2.794M	6.07	4.080M	8.42
Clique+SR	1.199M	2.29	1.818M	4.40	2.794M	6.05	4.080M	8.44
Clique+SRG	1.199M	2.31	1.818M	4.37	2.794M	6.10	4.080M	8.42

At 40% density, Table 14, the clique-major structure performs better in the base version, and in all versions that do not use the gap bound. When the gap bound is used the candidate-major structure becomes the best. Despite once again not lowering the edge count in a meaningful way, the inclusion of the bounds have a large effect on the runtime of the algorithm, with differences of up to four seconds occurring at the largest graph size. At the largest size the lowest runtime comes from the use of all bounds, although for some graph size the best runtime did not include the Russian doll bound.

Table 14: Russian Necklace Filters: density=0.4

Vertices	275		300		325		350	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	47.054M	11.22	72.544M	16.71	108.992M	27.86	157.769M	43.45
Candidate+G	47.054M	11.28	72.544M	16.58	108.992M	28.72	157.769M	37.39
Candidate+R	47.054M	11.43	72.544M	16.58	108.992M	28.25	157.769M	41.01
Candidate+RG	47.054M	11.23	72.544M	16.65	108.992M	28.20	157.769M	37.71
Candidate+S	47.054M	11.82	72.543M	16.71	108.991M	27.69	157.769M	42.54
Candidate+SG	47.054M	11.04	72.543M	16.41	108.991M	28.05	157.769M	37.47
Candidate+SR	47.054M	11.42	72.543M	16.89	108.991M	28.22	157.769M	41.61
Candidate+SRG	47.054M	11.04	72.543M	16.61	108.991M	28.35	157.769M	37.33
Clique	47.074M	11.15	72.574M	16.63	109.038M	28.17	157.835M	41.98
Clique+G	47.068M	11.09	72.566M	16.55	109.025M	29.17	157.817M	38.11
Clique+R	47.073M	11.19	72.574M	16.44	109.037M	30.03	157.834M	38.00
Clique+RG	47.067M	12.96	72.565M	16.79	109.024M	28.03	157.816M	37.58
Clique+S	47.074M	11.01	72.574M	16.89	109.038M	29.67	157.835M	40.92
Clique+SG	47.068M	11.47	72.566M	16.60	109.025M	27.86	157.817M	38.27
Clique+SR	47.073M	11.22	72.574M	16.86	109.037M	27.91	157.834M	41.48
Clique+SRG	47.067M	11.54	72.565M	16.68	109.024M	28.59	157.815M	38.06

At 60% density, Table 15, the inclusion of all bounds produces the fastest algorithm in both the candidate-major, and clique-major structures. As before, while the clique-major structure is initially faster than candidate-major structure, the inclusion of all bounds makes candidate-major the fastest across all tests, having the fewest edge verifications.

Table 15: Russian Necklace Filters: density=0.6

Vertices	125		150		175		200	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	8.376M	1.03	20.954M	2.36	45.162M	5.77	91.269M	12.07
Candidate+G	8.376M	0.94	20.953M	2.48	45.160M	5.83	91.263M	12.62
Candidate+R	8.376M	0.94	20.954M	2.37	45.163M	5.48	91.271M	11.84
Candidate+RG	8.376M	0.94	20.953M	2.50	45.160M	5.80	91.263M	13.26
Candidate+S	8.361M	0.90	20.908M	2.34	45.054M	5.37	91.021M	11.98
Candidate+SG	8.361M	0.92	20.907M	2.50	45.052M	5.77	91.017M	12.84
Candidate+SR	8.361M	0.97	20.907M	2.53	45.054M	5.68	91.021M	11.89
Candidate+SRG	8.360M	0.89	20.905M	2.26	45.048M	5.12	91.006M	10.87
Clique	8.428M	0.97	21.064M	2.51	45.407M	5.74	91.839M	11.56
Clique+G	8.414M	0.89	21.023M	2.51	45.313M	5.88	91.622M	12.60
Clique+R	8.426M	0.97	21.062M	2.36	45.406M	6.01	91.837M	12.50
Clique+RG	8.411M	0.94	21.021M	2.50	45.312M	6.01	91.619M	12.62
Clique+S	8.425M	0.89	21.056M	2.53	45.386M	5.76	91.789M	12.29
Clique+SG	8.411M	0.89	21.016M	2.46	45.296M	5.96	91.581M	12.78
Clique+SR	8.423M	0.92	21.053M	2.54	45.384M	5.77	91.784M	12.06
Clique+SRG	8.409M	0.90	21.014M	2.53	45.295M	6.07	91.578M	11.98

Lastly, at 80% density, Table 16, a similar result is seen to the 60% density tests. The candidate-major test gives the best performance when combined with all other bounds in both edge verifications and runtime.

Table 16: Russian Necklace Filters: density=0.8

Vertices	75		100		125		150	
Filter	EC	T(s)	EC	T(s)	EC	T(s)	EC	T(s)
Candidate	3.984M	0.25	19.227M	1.33	63.014M	4.52	162.492M	12.11
Candidate+G	3.973M	0.22	19.157M	1.14	62.754M	3.81	161.549M	11.29
Candidate+R	3.988M	0.22	19.244M	1.11	63.044M	3.87	162.553M	11.53
Candidate+RG	3.973M	0.20	19.159M	1.14	62.749M	3.93	161.550M	11.61
Candidate+S	3.794M	0.25	18.129M	1.08	59.167M	3.95	150.201M	11.04
Candidate+SG	3.780M	0.20	18.048M	1.08	58.898M	3.60	149.467M	10.50
Candidate+SR	3.791M	0.20	18.116M	1.08	59.136M	3.88	150.150M	11.06
Candidate+SRG	3.681M	0.20	17.168M	1.03	54.906M	3.60	142.370M	10.16
Clique	4.414M	0.23	21.725M	1.22	72.062M	4.38	187.276M	12.28
Clique+G	4.319M	0.25	21.260M	1.22	70.577M	4.31	183.303M	12.21
Clique+R	4.404M	0.23	21.687M	1.25	71.920M	4.27	186.938M	12.45
Clique+RG	4.307M	0.23	21.215M	1.20	70.422M	4.09	182.965M	11.75
Clique+S	4.267M	0.20	20.667M	1.14	67.966M	4.02	173.352M	11.45
Clique+SG	4.187M	0.23	20.315M	1.14	66.930M	3.99	171.175M	11.26
Clique+SR	4.259M	0.23	20.649M	1.17	67.909M	4.18	173.278M	11.54
Clique+SRG	4.182M	0.25	20.309M	1.12	66.902M	4.15	171.149M	11.42

6.3 Conclusions

For the Necklace and Backtracking searches, the optimal filtering technique is highly dependent on the density of the hypergraph. In a low density hypergraph, the clique-major structure should be used, while in the higher density graphs the candidate-major structure should be used. In either case, the size bound should be included into the filter to reduce the overall runtime. For Russian Doll searches the result is less clear. In the lowest density graphs ($< 40\%$), a clique-major structure should be used. In the higher density graphs, candidate-major structure should be used. As with the previous algorithms, the size bound should be included, as it provided a universal reduction in the runtime and number of edge verifications. The use of the Russian doll property within the bound however remains unclear. The number of cases where its inclusion with the size bound increased performance was roughly equal to the number of times where its inclusion was detrimental. Other structures within the hypergraph not studied

here may be the cause of the inconsistent performance of this bound. For the Russian Necklace Search, the choice of filter has little to no effect at lower densities, so any can be used. In higher density hypergraphs the candidate-major algorithm with all bounds included should be used. In the following chapter, which compares the performances of the searches with each other, Backtracking, Russian Doll, and Russian Necklace searches will all use the candidate-major filter with the size bound, while Russian Necklace search will use the candidate-major filter with all bounds.

7

Experimental Comparison Between Max-clique Algorithms

In this chapter, we do an experimental evaluation of the Russian Necklace algorithm by comparing its performance to the other algorithms, namely Backtracking, Russian Doll, and Necklace. These experiments are divided into two sections, searches on randomly generated circulant k-hypergraphs, and circulant k-hypergraphs generated using LFSRs. There is not a standard set of benchmarking hypergraphs for circulant k-hypergraphs as is the case for graphs, and so the experimentation was done using random circulant k-hypergraphs. For these experiments 3 main parameters were used: number of vertices, edge size, and edge density. In order to model the performance of each algorithm, for each parameter set of density and edge size, all algorithms were run with increasing vertex counts until the execution time of a single search surpassed 5 minutes. This process started with the smallest k-hypergraphs on 60 vertices and increased the number of vertices by 5, in each step. An increase of 5 vertices is equivalent to a potential search

space 32 times larger than the previous step. Hypergraphs were generated at random and the same set of hypergraphs was used between all algorithms. In the second data set, The Russian Necklace and Necklace algorithms were run on circulant k-hypergraphs generated using LFSRs. This search was done in order to expand on previous work, and search for new minimally sized covering arrays.

7.1 Implementation Details

In order to mitigate any differences in non-algorithm based segments of the implementation (logging, timing, graph storage, and edge lookups) the testing software was written in Java using abstract classes with only the search algorithm, and in the case of the Russian Necklace algorithm the filter, being different for each search. The hypergraphs were stored using the modified adjacency matrix structure detailed in Section 3.2.2. All experiments were done on an Intel Core i7-4790 CPU @ 3.60GHz CPU running Windows 7 OS and Java 7.

7.2 Data Set 1: Random Circulant Hypergraphs

When testing and comparing algorithms using random graphs, the graph must have the condition that when selecting the edge set at random, all pairs of vertices have an equal probability of becoming an edge. Similarly for k-hypergraphs all k-subsets must have equal probability of forming an edge. To generate a random graph, or k-hypergraph with a given density $0 \leq d \leq 1$, each possible edge e is looped over and a random number $0 \leq r \leq 1$ is generated. If $r \leq d$ then e is included as an edge, otherwise it is not. Each edge has uniform probability d of being included in the graph, and the graph will have an expected density of d.

When generating circulant graphs and hypergraphs however, this method of edge generation no longer provides a truly random set of edges. Some edges will have a higher

probability of selection over others due to the variances in the rotational period of the edge selected. Since the graph is circulant, the selection of one edge will force the inclusion of all rotations of this edge into the hypergraph. Since the number of unique rotations of edges are not always equal, edges with larger periods will have a higher probability of selection causing the graph to no longer be truly random. Consider as an example a 4-hypergraph with 8 vertices. The possible edges of the hypergraph are shown in Figure 12, grouped by rotations.

Figure 12: Rotations and Edge Probability

Edge	Rotations			Period	Probability
$[0,1,2,3]$	$[1,2,3,4]$ $[4,5,6,7]$ $[0,1,2,7]$	$[2,3,4,5]$ $[0,5,6,7]$	$[3,4,5,6]$ $[0,1,6,7]$	8	$\frac{8}{70} \approx 11.4286\%$
$[0,1,2,4]$	$[1,2,3,5]$ $[0,4,5,6]$ $[0,1,3,7]$	$[2,3,4,6]$ $[1,5,6,7]$	$[3,4,5,7]$ $[0,2,6,7]$	8	$\frac{8}{70} \approx 11.4286\%$
$[0,1,2,5]$	$[1,2,3,6]$ $[1,4,5,6]$ $[0,1,4,7]$	$[2,3,4,7]$ $[2,5,6,7]$	$[0,3,4,5]$ $[0,3,6,7]$	8	$\frac{8}{70} \approx 11.4286\%$
$[0,1,2,6]$	$[1,2,3,7]$ $[2,4,5,6]$ $[0,1,5,7]$	$[0,2,3,4]$ $[3,5,6,7]$	$[1,3,4,5]$ $[0,4,6,7]$	8	$\frac{8}{70} \approx 11.4286\%$
$[0,1,3,4]$	$[1,2,4,5]$ $[0,4,5,7]$ $[0,2,3,7]$	$[2,3,5,6]$ $[0,1,5,6]$	$[3,4,6,7]$ $[1,2,6,7]$	8	$\frac{8}{70} \approx 11.4286\%$
$[0,1,3,5]$	$[1,2,4,6]$ $[1,4,5,7]$ $[0,2,4,7]$	$[2,3,5,7]$ $[0,2,5,6]$	$[0,3,4,6]$ $[1,3,6,7]$	8	$\frac{8}{70} \approx 11.4286\%$
$[0,1,3,6]$	$[1,2,4,7]$ $[2,4,5,7]$ $[0,2,5,7]$	$[0,2,3,5]$ $[0,3,5,6]$	$[1,3,4,6]$ $[1,4,6,7]$	8	$\frac{8}{70} \approx 11.4286\%$
$[0,1,4,5]$	$[1,2,5,6]$	$[2,3,6,7]$	$[0,3,4,7]$	4	$\frac{4}{70} \approx 5.7143\%$
$[0,1,4,6]$	$[1,2,5,7]$ $[0,2,4,5]$ $[0,3,5,7]$	$[0,2,3,6]$ $[1,3,5,6]$	$[1,3,4,7]$ $[2,4,6,7]$	8	$\frac{8}{70} \approx 11.4286\%$
$[0,2,4,6]$		$[1,3,5,7]$		2	$\frac{2}{70} \approx 2.8571\%$

In order to account for edges with shorted periods, and therefore lower probabilities of inclusion, not all edges are considered when performing the random generation. Only edges which are canonical subset necklaces are selected as candidate edges for the k-hypergraph. Since each edge can have only one rotation which is a canonical subset necklace, the probability of that edge being included becomes the probability of its canonical subset necklace being selected. The edges then have a uniform probability of being selected and the graph can be considered random. One consequence of this however is that the true density of the graph may differ from the parameter given for density, as selecting an edge with a lower period will include fewer edges in the final graph. Therefore, the density parameter used should be interpreted as the probability of a canonical subset necklace to be in the hypergraph, and only an approximation of the actual edge density.

7.2.1 Experimental results for random circulant 3-hypergraphs

Figure 13 shows the performance of all four algorithms on 3-hypergraphs ranging in density between 95% and 75%. In many of these graphs, and in addition many of the graphs below, the performance of the two naive algorithms, Russian doll and backtracking, show very little difference in terms of performance. In most iterations, the time difference between these algorithms was around 1 second. Due to this the lines they produce on the graph nearly perfectly overlap and in most cases appear as a single line.

In these high density hypergraphs, the performance of the new Russian Necklace algorithm greatly outperformed the Necklace algorithm and both naive algorithms. In the case of 95% density hypergraphs, the algorithm took 11 seconds to search the 100-vertex hypergraph while the Necklace algorithm took 239 seconds. Both naive algorithms surpassed the 5 minute threshold at 90 vertices.

As the density of the hypergraphs is reduced, the performance difference between both the Necklace algorithm and the Russian Necklace algorithm continue to hold. In most cases, the Russian Necklace algorithm is able to search graphs considerably larger in the same time as the Necklace algorithm.

Figure 13: k=3 Runtimes by Edge Density: 95%-75%

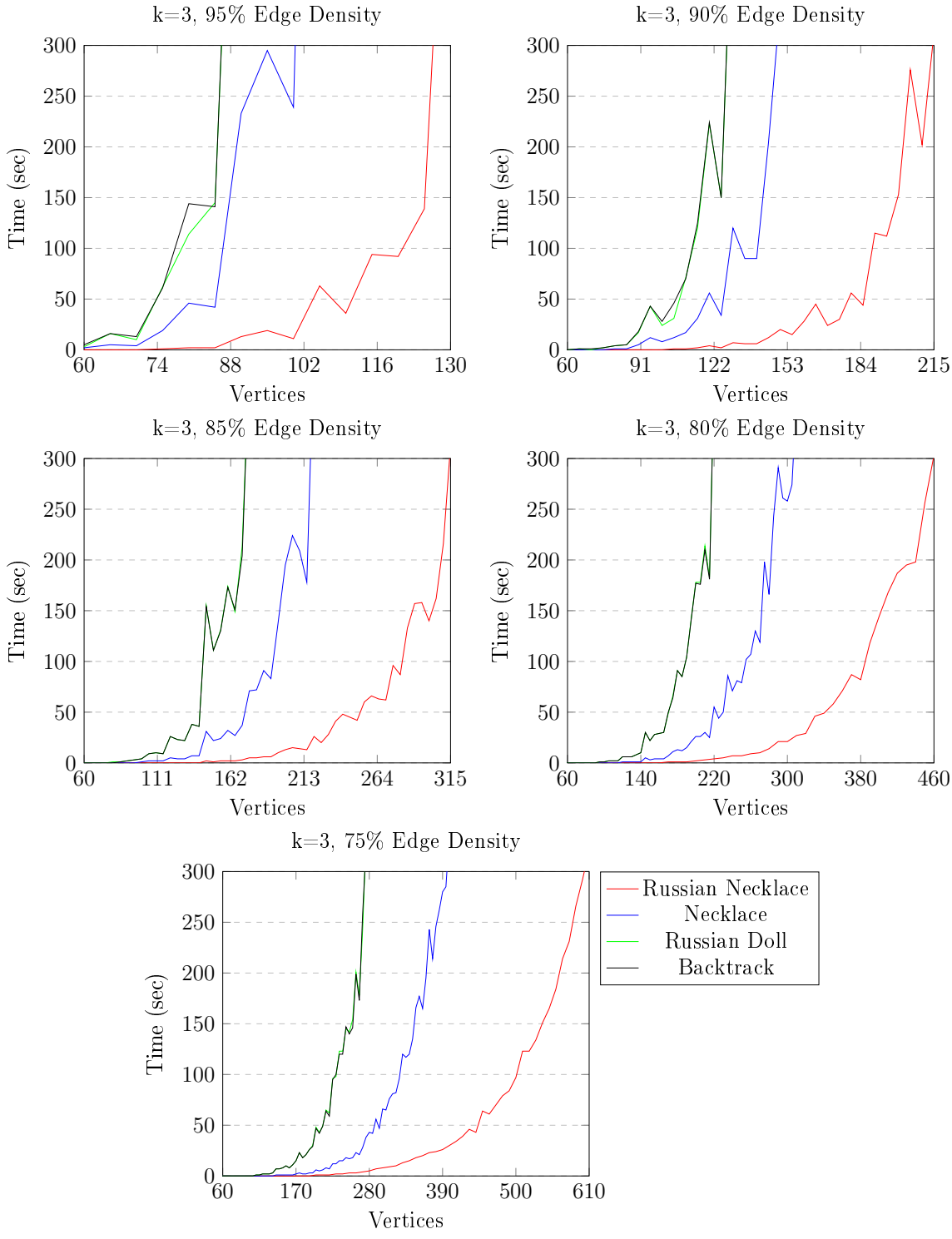


Figure 14 shows the performance of all four algorithms on 3-hypergraphs ranging in density between 70% and 50%. As was the case with the higher density hypergraphs, in mid density hypergraphs the performance of the two naive algorithms is both near identical, and worse than the performance of the Necklace and Russian Necklace algorithms.

In these mid density algorithms, we see a shift in the performance difference of the Necklace and Russian Necklace algorithms. At 70% the Russian Necklace algorithm still outperforms the Necklace algorithm, but as we approach 55% density the gap in performance between the algorithms shrinks. At the 50% density point the two algorithms have similar run-times, while the Russian Necklace algorithm is still able to complete a search on a 1180-vertex graph compared to a maximum of 1120-vertex for Necklace. Although the Russian Necklace algorithm still outperforms the Necklace algorithm, the difference between the two performances has been greatly reduced from what was seen at lower densities.

The decrease of the density of graphs in successive experiments also leads to an increase in the overall size of the graphs which could be searched by all algorithms. When the density was set to 95% the largest graphs searchable by the naive algorithms was 90, while the Necklace algorithm was able to search up to size 110 and Russian Necklace achieved a hypergraph of 130 vertices. At 50% density, the naive algorithms reached 565 vertices while the Necklace and Russian Necklace algorithms achieved 1160 and 1170 vertices respectively.

Figure 14: k=3 Runtimes by Edge Density: 70%-50%

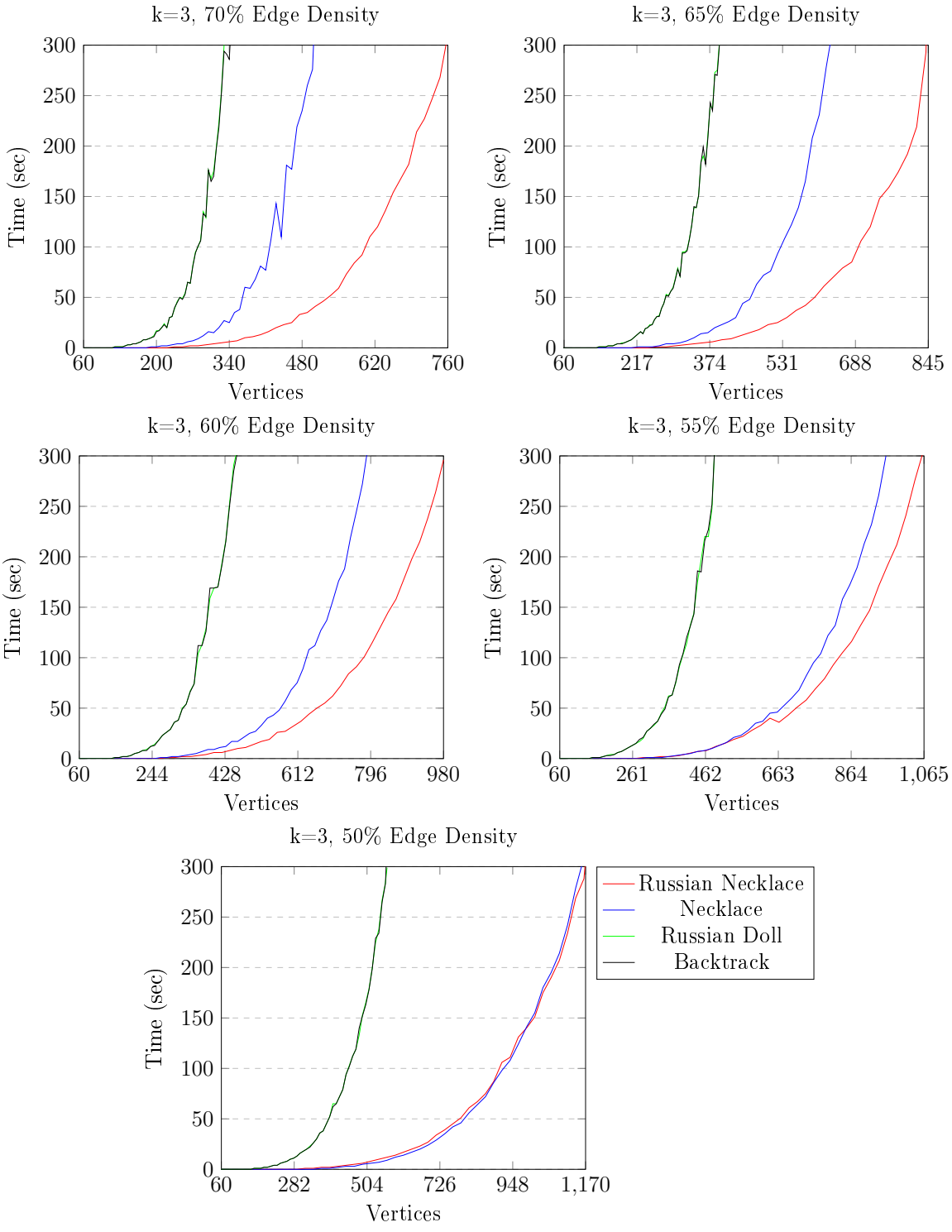
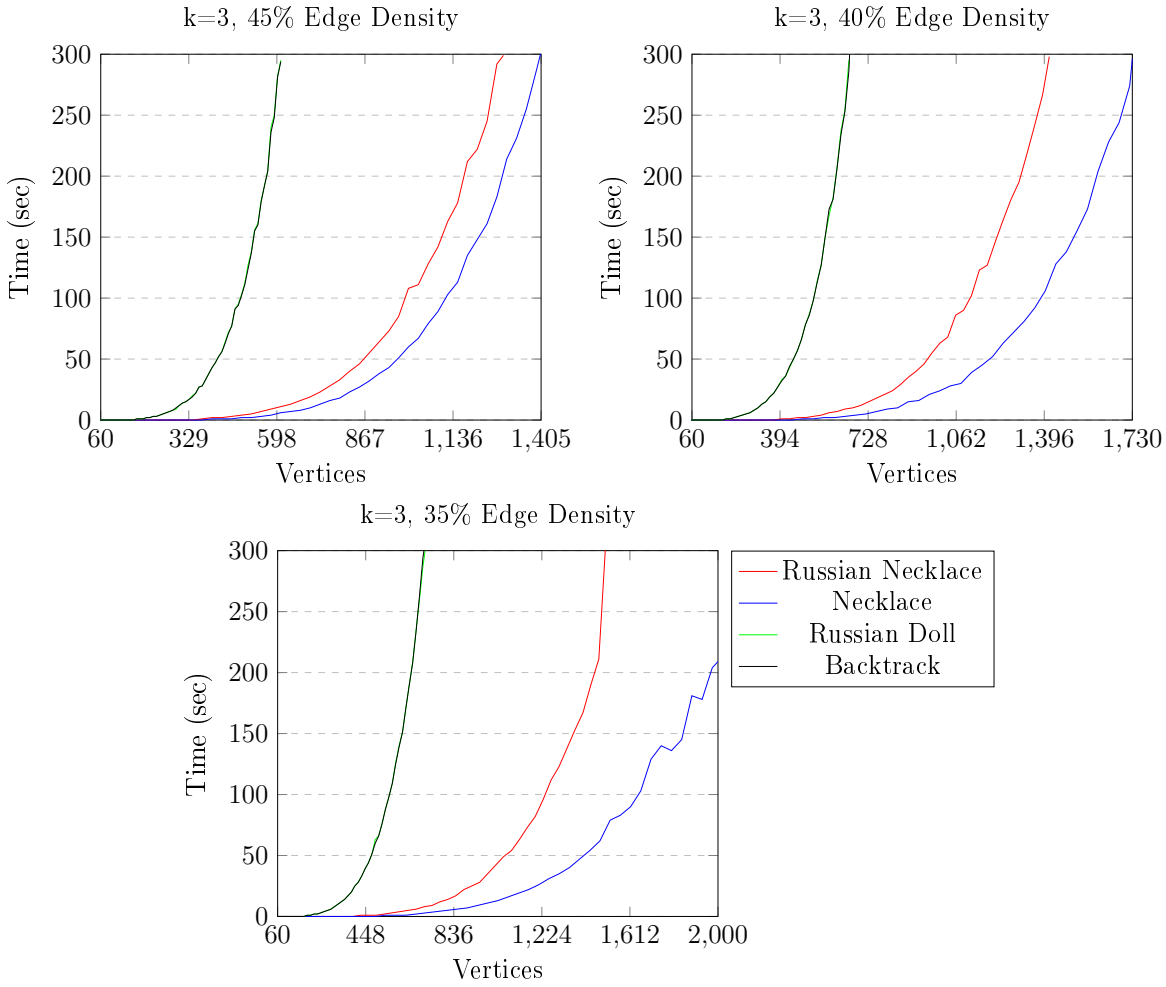


Figure 15 gives the results at the lowest densities measured in these experiments, 45% to 35%. The most immediate result of this is that we can see the point where the performance of the Russian Necklace algorithm is surpassed by the Necklace algorithm. At 45% density the performance of the Necklace algorithm has surpassed that of the Russian Necklace, and further reductions in density increases the performance difference.

Figure 15: k=3 Runtimes by Edge Density: 45%-35%



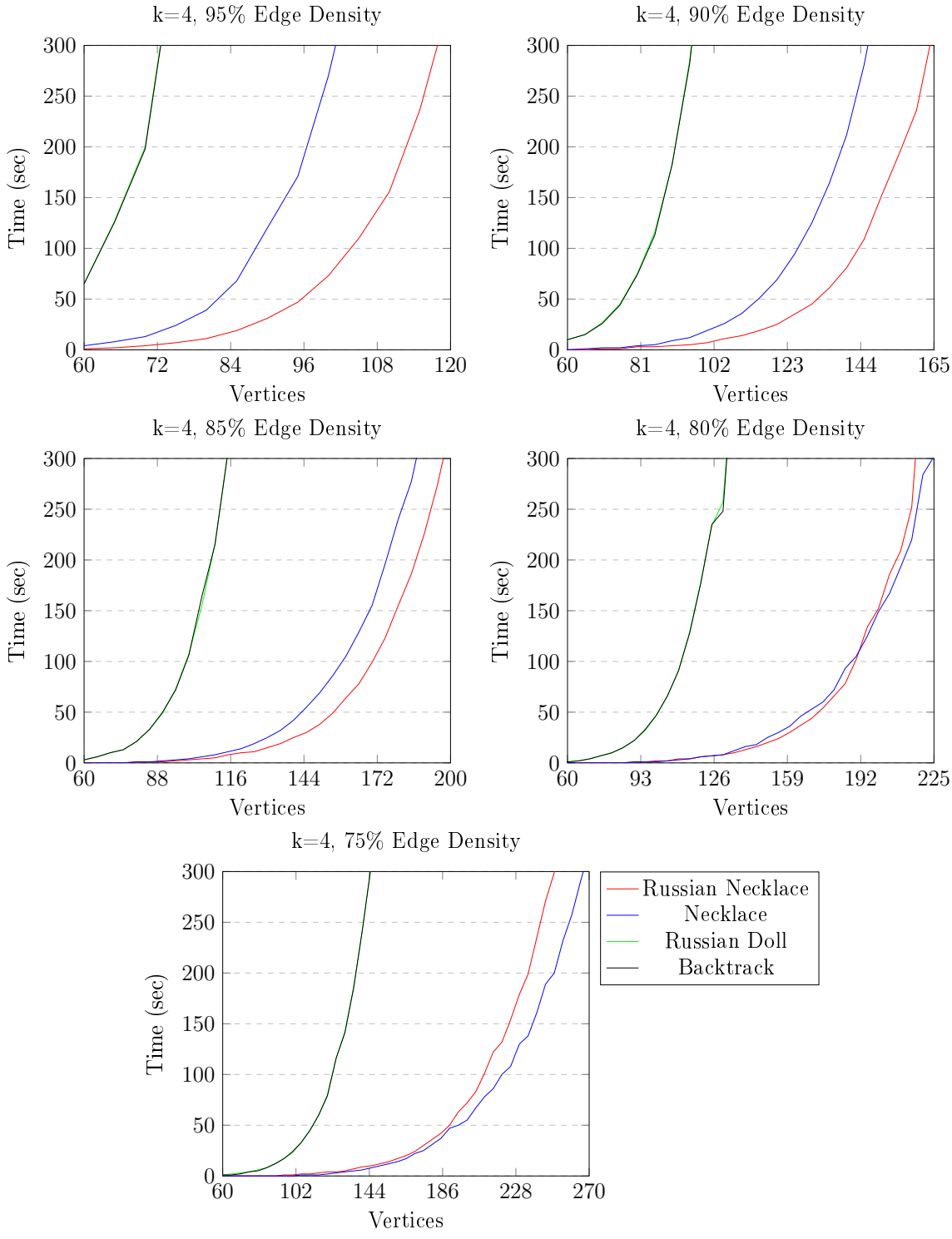
7.2.2 Experimental results for random circulant 4-hypergraphs

Figure 16 shows the performance of the algorithms on 4-hypergraphs ranging in density between 95% and 75%. As before with 3-hypergraphs, the two naive algorithms showed similar performances. Backtracking and Russian doll both have performances considerably lower than the Necklace and Russian Necklace algorithms. At the point where both reach the threshold of five minutes the Necklace and Russian Necklace algorithms both take less than 30 seconds to solve the same hypergraphs.

As was the case in 3-hypergraphs, the Russian Necklace algorithm was able to search larger hypergraphs than the Necklace algorithm. As the density reduces however, the Necklace algorithms performance was able to match that of the Russian Necklace. In 3-Hypergraphs this occurred at 50% density, in 4-hypergraphs this crossover point occurs at 80% density.

Overall the size (number of vertices) of the hypergraphs that were searchable was greatly reduced when increasing the edge size of the hypergraphs. At 90% density the largest hypergraph searched dropped from 215 to 165 vertices. This is due to the increased complexity of the filtering operation. This decrease in hypergraph size was seen across all algorithms and densities in the 4-hypergraph experiment set.

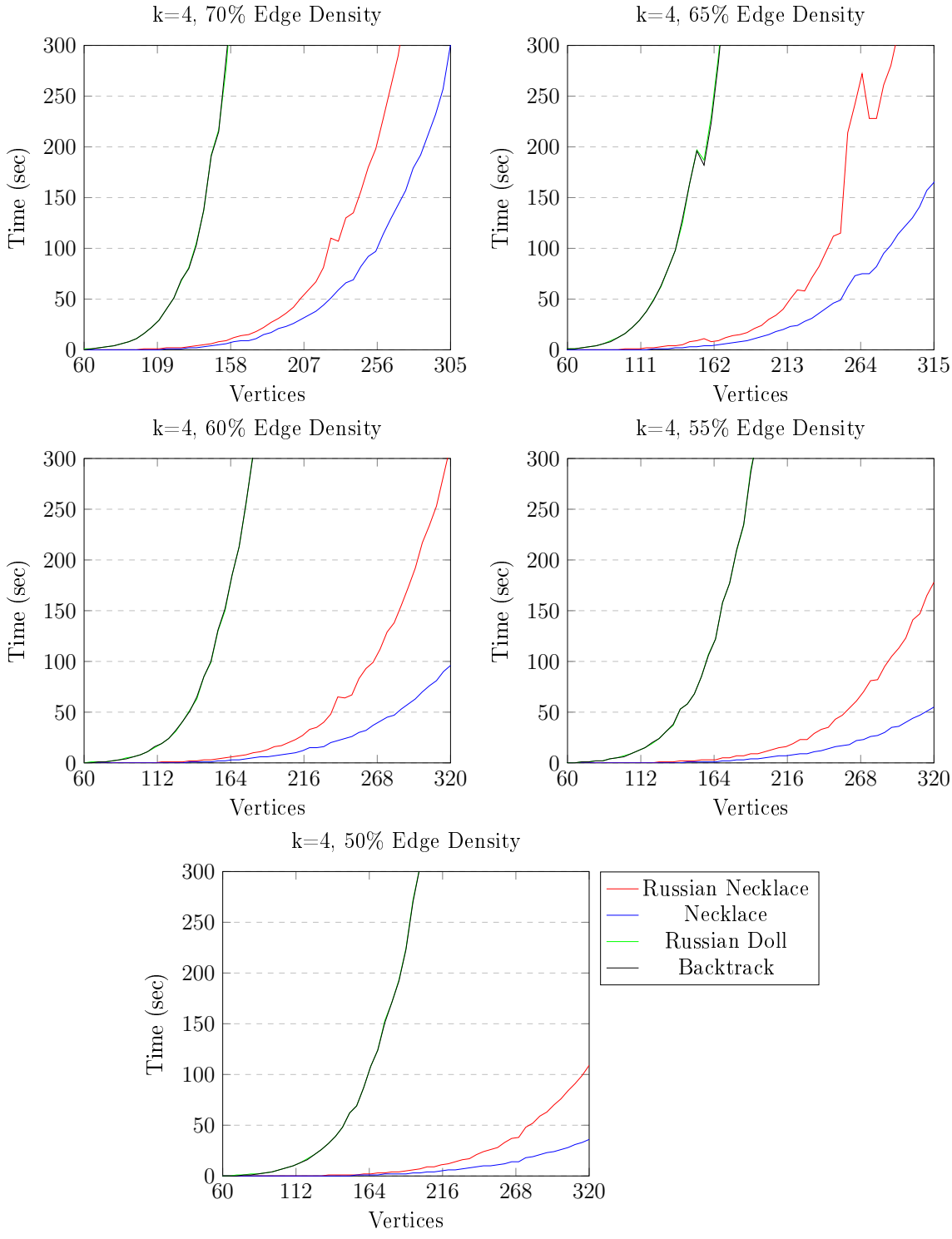
Figure 16: k=4 Runtimes by Edge Density: 95%-75%



In the $70\% - 50\%$ density 4-hypergraphs (Figure 17), we begin to see the widening of the gap between the Necklace and the Russian Necklace algorithms. As the density is reduced, the Necklace algorithm is able to outperform the Russian Necklace algorithm. Once again the performance of the Backtracking and Russian Doll algorithms is much poorer than the algorithms designed for use in circulant hypergraphs. In this density range, the Necklace algorithm was able to search the largest hypergraphs in the allotted 5 minutes. As with the higher density hypergraphs, the search times were much longer than in 3-hypergraphs, due to the added complexity of the filtering operations.

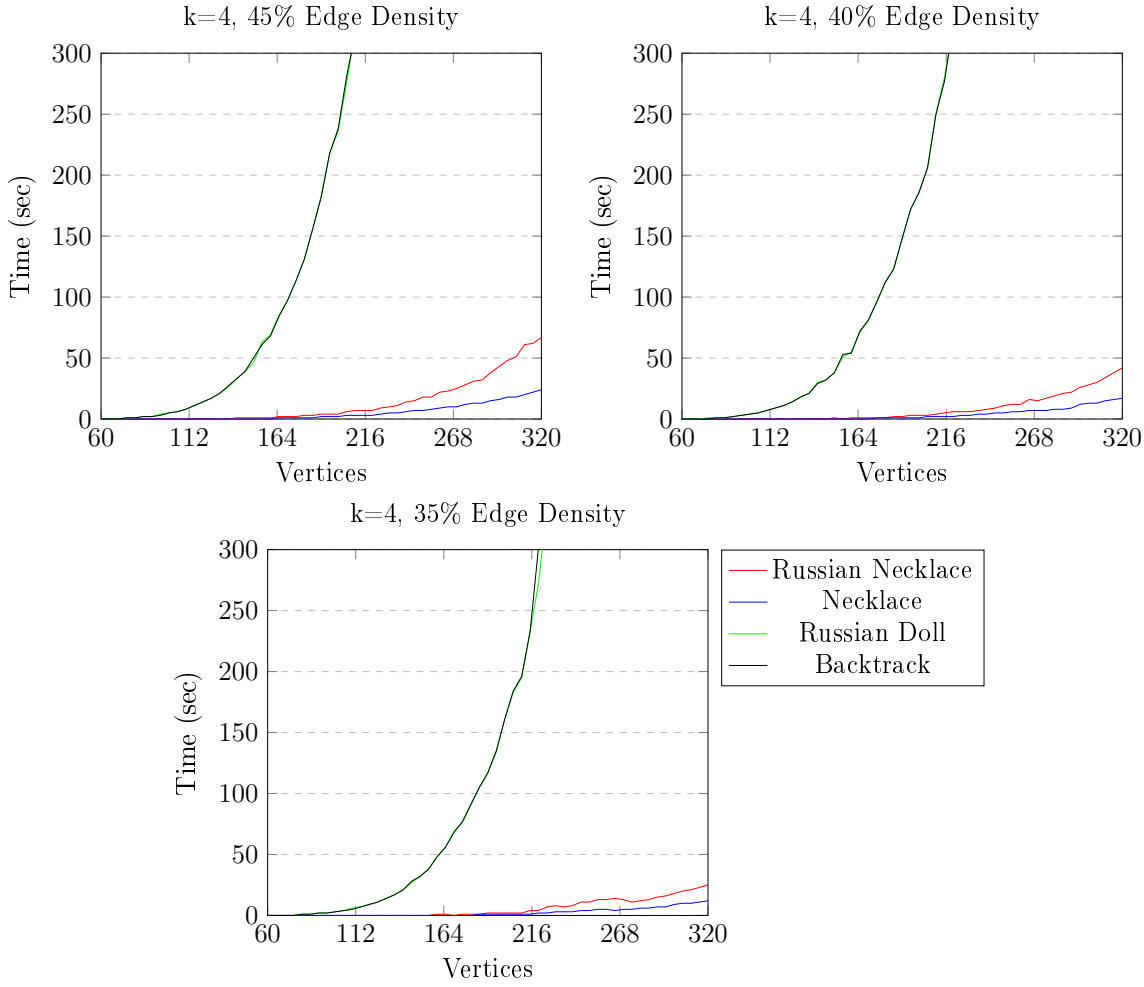
Due to the system limitation on memory, a limit occurred in the ability to test 4-hypergraphs beyond a certain size. This limitation began to manifest at 320 vertices. The point where this limit occurred however was sufficiently large to see the establishment of the runtime curves for all algorithms. We assume from previous experiments that the runtimes will follow the established exponential curves as we further increase the size, as the runtime for all types of this exhaustive search is exponential. As the Necklace algorithm has established itself as having lower runtimes across all tested hypergraph sizes at this density we assume this trend will continue and the lowest runtimes will be achieved from this algorithm.

Figure 17: k=4 Runtimes by Edge Density: 70%-50%



In the lowest density 4-hypergraphs tested (Figure 18), the memory restriction is once again at work limiting the largest hypergraph searched to 320 vertices. Again, we have sufficient data to extrapolate the curves and make inferences against larger hypergraphs. If the curves were to follow what has been seen in all other tests we would still expect the Necklace algorithm to outperform the Russian Necklace algorithm, as we can see lower search times for the available values, and a trend towards the same exponential curve as other experiment sets.

Figure 18: k=4 Runtimes by Edge Density: 45%-35%



7.2.3 Experimental results for random circulant 5-hypergraphs

For 5−hypergraphs the largest size attainable was 160 vertices. This was again due to the memory limitation. In the $95\% - 75\%$ density hypergraphs, Figure 19, the performance of the naive algorithms has dropped considerably. At 95% they are able to search only 2 hypergraphs, the 60, and 65-vertex ones. The performance of the Russian Necklace algorithm once again surpasses that of the Necklace algorithm at the highest densities. The point in which the Necklace algorithm begins to outperform the Russian Necklace algorithm occurs at a higher density in 5-hypergraphs than in 3-hypergraphs and 4-hypergraphs. This crossover occurs between 85% and 90% densities, as opposed to 80% in 4-hypergraphs, and 50% in 3-hypergraphs.

In the $70\% - 50\%$ density hypergraphs, Figure 20, the Necklace algorithm outperforms all other algorithms. This data set is again limited to 160 vertices, however the curves have been established by this point and demonstrate the performance gap between the algorithms. As before the Backtracking and Russian doll algorithms show similar performance. This same performance is seen in the $45\% - 30\%$ density hypergraphs in Figure 21.

Figure 19: k=5 Runtimes by Edge Density: 95%-75%

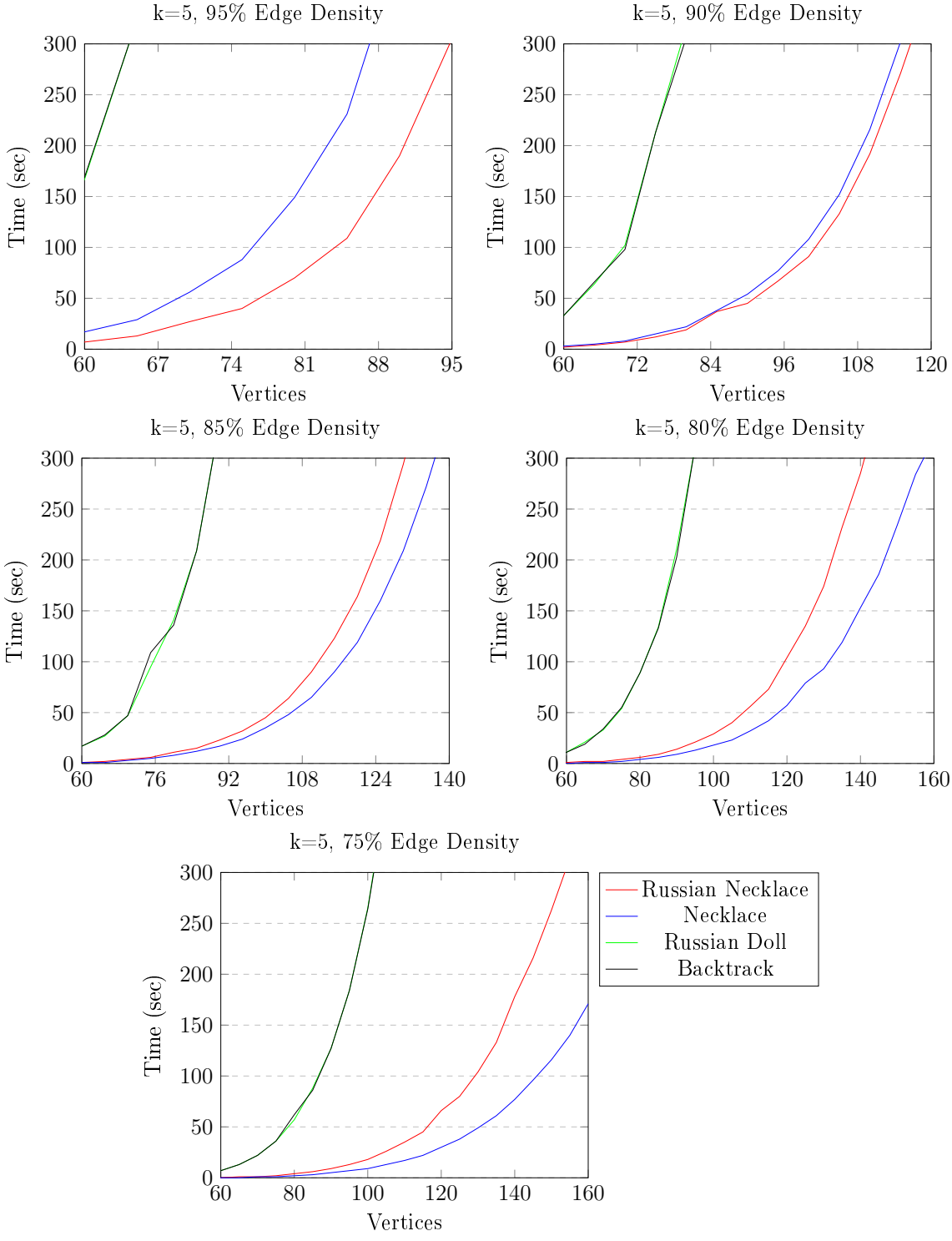


Figure 20: k=5 Runtimes by Edge Density: 70%-50%

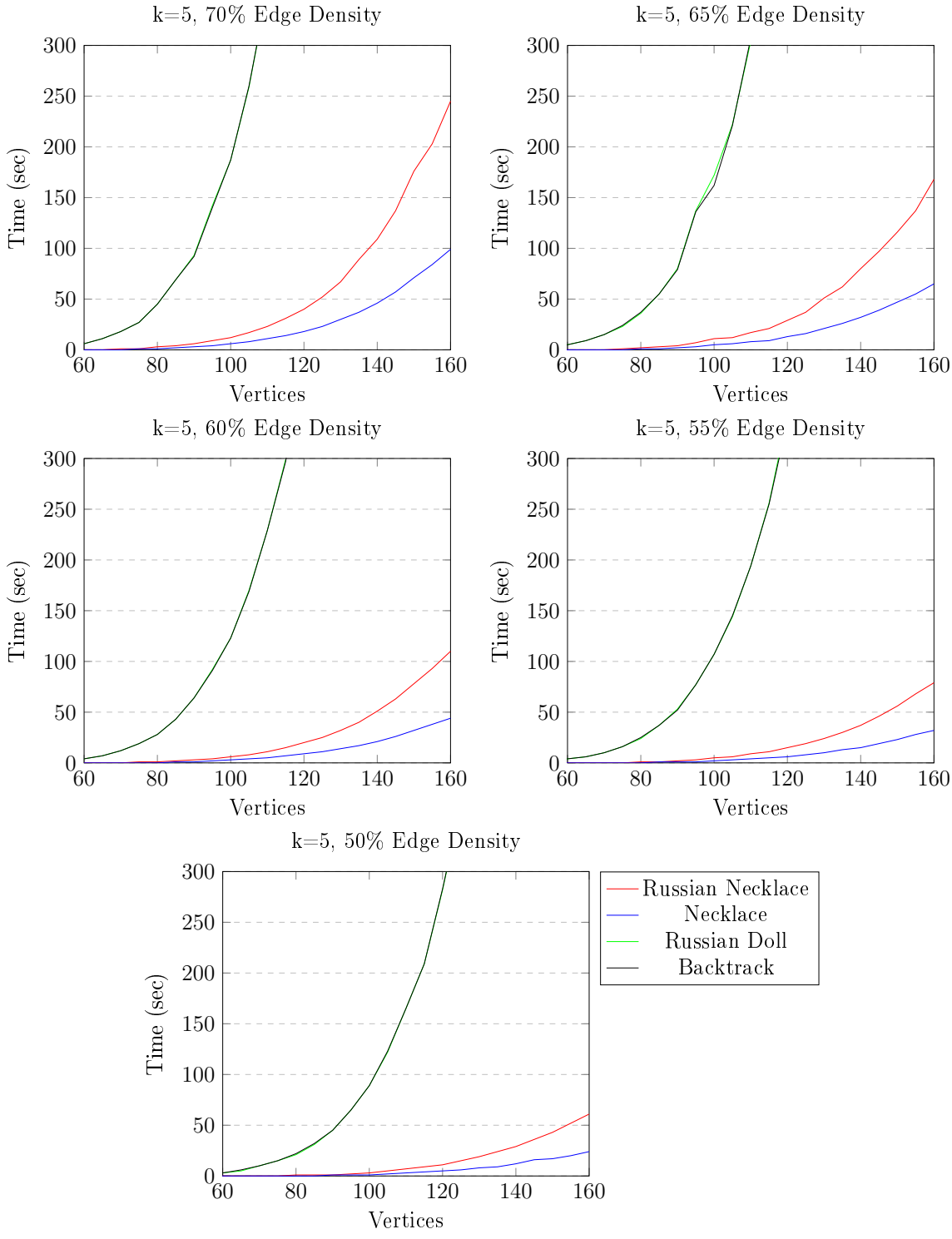
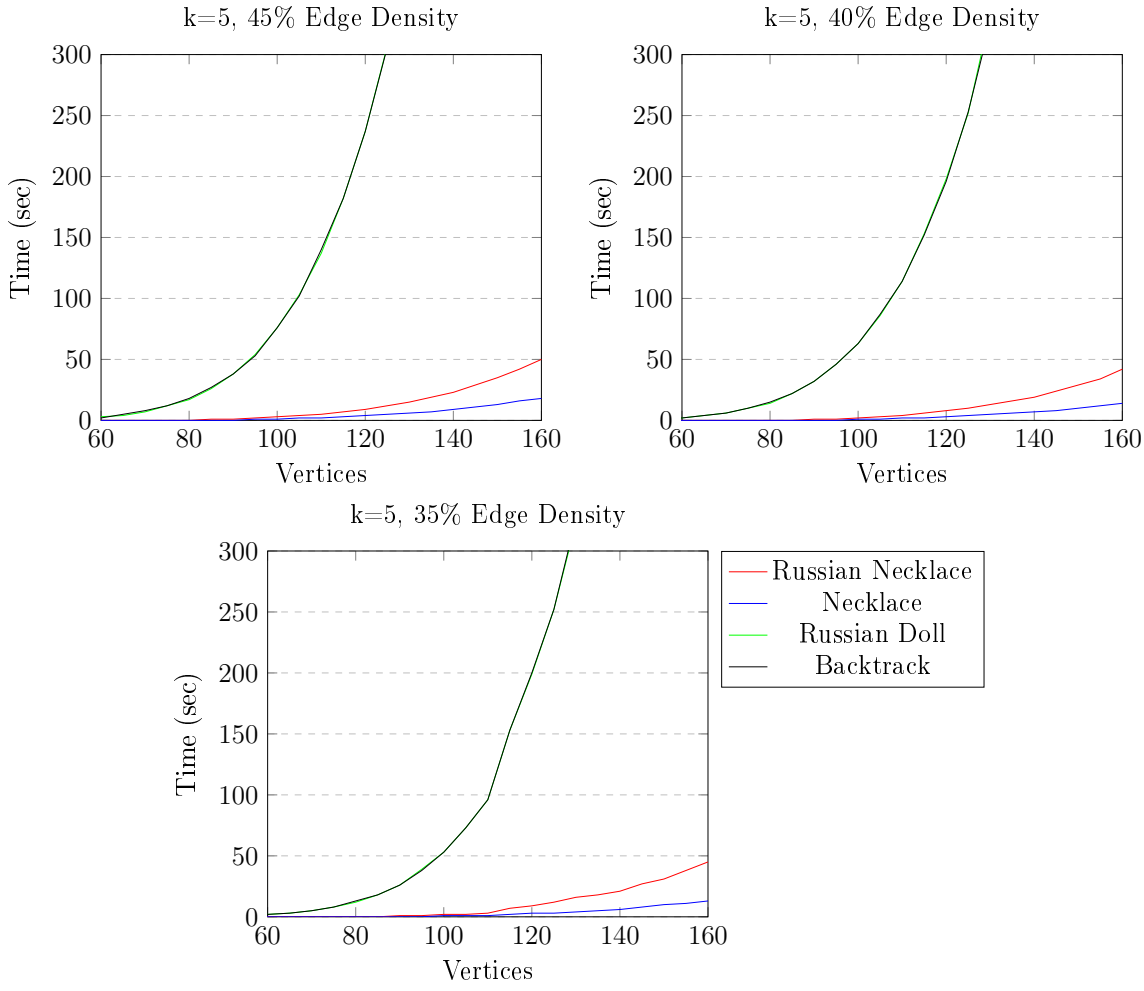


Figure 21: k=5 Runtimes by Edge Density: 45%-35%



7.3 Data Set 2: Circulant 4-Hypergraphs from LFSR

The initial creation of the Necklace search was done to facilitate the search for covering arrays by using circulant hypergraphs generated using linear feedback shift registers (LFSR) in earlier work by Tzanakis et al. [17]. Covering arrays are combinatorial objects which can be used in the generation of software tests. The definitions to follow are from Colbourn [4], and Moura, Mullen, and Panario [11]. In general for this section $\mathbb{F}_q$ denotes a finite field of order q.

Definition 7.1. A *Covering Array* $CA(N; t, k, v)$ is an $N \times k$ matrix with symbols from a v-ary alphabet G such that in each $N \times t$ subarray, each t-tuple in G^t appears at least once as a row.

Definition 7.2. In an $N \times k$ array with symbols from a v-ary alphabet G, a t-subset T of columns is *covered* if in the $N \times t$ subarray indexed by T, each t-tuple in G^t appears at least once as a row.

Definition 7.3. Let $f(x) = x^m + c_{m-1}x^{m-1} + \cdots + c_1 x + c_0 \in \mathbb{F}_q[x]$ and $I = (b_0, ..., b_{m-1}) \in \mathbb{F}_{q^m}$. The sequence $S(f, I) = (a_0, a_1, ...)$ defined as

$$a_i = \begin{cases} b_i & \text{if } 0 \leq i < m, \\ -c_{m-1}a_{i-1} - c_{m-2}a_{(i-2)} - \ldots - c_1 a_{i-m-1} - c_o a_{i-m} & \text{if } i \geq m. \end{cases} \tag{2}$$

is a *linear feedback shift register (LFSR) sequence over* $\mathbb{F}_q$ with *characteristic polynomial* f and initial values $(b_0, \ldots, b_{m-1})$.

Definition 7.4. Let f be an irreducible polynomial of degree m over $\mathbb{F}_q$. If $\alpha \in \mathbb{F}_{q^m}$ is a root of f and $\mathbb{F}_{q^m} = \{0, \alpha^0, \alpha^1, \ldots, \alpha^{q^m-2}\}$ then α is a *primitive element* of $\mathbb{F}_{q^m}$, and f is a *primitive polynomial*.

Definition 7.5. The *period* of a sequence is the number of elements generated before the sequence repeats.

The maximal period of a polynomial f over $\mathbb{F}_q$ is $q^m - 1$. A primitive polynomial has a period equal to the maximum period.

Definition 7.6. Let f be a primitive polynomial over $\mathbb{F}_q$ of degree m and let S be the LFSR sequence (of maximal period) generated by f on some arbitrary set of initial values $T \neq (0, \ldots, 0)$. Let $C_i^n(S)$ be the interval of S of length n beginning at position i. The *subinterval array of* f *of length* n is a $q^m \times n$ array $A^n(f)$ where

$$A^n(f) = \begin{bmatrix} C_0^n(S) \\ C_1^n(S) \\ \vdots \\ C_{q^t-1}^n(S) \\ 00\ldots0 \end{bmatrix}$$

Tzanakis et al. [17] used subinterval arrays to build covering arrays using the following construction. For a pair of primitive polynomials f, g over $\mathbb{F}_q$ of degree 4, we vertically concatenate the subinterval arrays $A^{\frac{q^4-1}{q-1}}(f)$ and $A^{\frac{q^4-1}{q-1}}(g)$ and remove one of the rows containing all 0's to form a large array B. Array B will not normally form a covering array, however a subset of columns of B will, and a covering array can be constructed by finding the largest set of columns of B which form a covering array. Furthermore, we can rephrase this problem as the problem of finding a max-clique in the following 4-hypergraph. Let $H_{q,f,g} = (V, E)$ be the hypergraph where V is the indices of columns of B and each 4-subset of columns is an edge if and only if the corresponding subarray is covered. Tzanakis et al. showed that for each m-subset M of columns in B which is covered, then for all rotations $M' = R(M, i)$, M' is covered. From this we know that if an edge e is present, all rotations $R(e, i)$ are present in the edge set. This hypergraph is therefore circulant. The maximum clique is the largest set of columns which will form a covering array.

Tzanakis et al. searched for max-cliques on $H_{q,f,g}$ for q a prime power in $[2, \ldots, 23]$ and every possible $f, g \in \mathbb{F}_q[x]$. In order to reduce the number of pairs to evaluate, they were able to prove some properties of the polynomials that allowed for the reduction of pairs to a small set (for example for $q = 4$ only 7 pairs of polynomials are required). This is done by fixing the first LFSR to a single one for all searches and selecting the second one from one of the established equivalent classes.

For $q = 2, 3, 4$ they were able to complete the search for every relevant pair of polynomials, and determined the covering array with the largest number of columns for each of them. However for $q \geq 5$, each of their searches did not complete, and they reported the largest clique found at the point where the search was terminated. These results still yield current records of covering arrays among Colbourn's tables of covering arrays [3], but it is left as an open question whether these values could be improved upon. In this section we settle the case of $q = 5$ by performing a complete search of the $q = 5$ hypergraphs, and confirming that the best clique found in the partial search is optimal. We

also revisited their results of $q = 2, 3, 4$. The case of $q = 7$ appears to be out of reach for the current implementations. The remainder of this section describes how we used this difficult dataset to confirm the effectiveness of our new Russian Necklace algorithm and report on our results.

In this data set, we first ran the searches on $q = 2, 3, 4$ LFSRs presented in [17] using both the Russian Necklace and Necklace algorithms. Table 17 shows the results of these searches. This table provides the time taken [RN Time, NK Time], and number of nodes [RN Nodes, NK Nodes] in the search tree for each search, as well as a reference to the LFSRs used [q, α's], density of the hypergraph [d], and max-clique sizes [MC]. For the single pair of hypergraphs from the $q = 2$ LFSRs, the search was trivial and took similar times for both algorithms. For the set of $q = 3$ hypergraphs, the Necklace search required approximately twice the number of nodes in the search tree, and took twice as long to run as the Russian Necklace algorithm. For $q = 4$ hypergraphs the Russian Necklace algorithm ran three times faster than the Necklace algorithm, and searches were completed in less than 12 seconds each. The full set of $q = 5$ hypergraphs were searched, taking between one and four hours for each search. The Necklace search performed 4 times worse on average than the Russian Necklace algorithm. We can see a clear difference in the performance of the Russian Necklace algorithm and the Necklace algorithms on these hypergraphs. The Necklace algorithm performs significantly worse during these searches, and the Russian Necklace algorithm should be used in later work. The $q = 7$ hypergraph search using the Russian Necklace algorithm was aborted after 2 days. Based on the exponential trend in the runtime for each step of the Russian Necklace algorithm established from previous searches, and the established trend after the 2 days of running, the runtime for $q = 7$ is expected to be extremely large.

Table 17: Results for Circulant Hypergraphs from LFSR

q	α's	$\|V\|$	d (%)	MC	RN Nodes	RN Time	NK Nodes	NK Time
2	1,7	15	69.230	6	98	0:00:00:1174	171	0:00:00:1953
3	1,7	40	75.467	9	23,232	0:00:00:4687	36,792	0:00:00:7968
3	1,11	40	80.282	10	24,543	0:00:00:5248	39,063	0:00:00:8954
3	1,13	40	80.107	9	23,764	0:00:00:5421	45,538	0:00:00:8452
4	1,3	85	87.532	17	3,567,014	0:00:07:6440	9,898,739	0:00:27:8461
4	1,7	85	85.030	17	2,611,565	0:00:05:6160	7,009,781	0:00:19:5781
4	1,9	85	87.532	12	5,061,145	0:00:11:1228	15,069,882	0:00:41:1686
4	1,13	85	87.532	12	5,432,480	0:00:11:6064	15,404,529	0:00:42:1046
4	1,21	85	87.330	12	5,255,894	0:00:11:2927	14,187,758	0:00:38:5634
4	1,29	85	87.532	17	3,452,242	0:00:07:6440	10,005,604	0:00:28:9578
4	1,37	85	85.030	17	2,578,597	0:00:05:6940	6,985,333	0:00:19:4845
5	1,7	156	91.368	16	2,683,547,098	3:54:21:1033	6,134,969,456	8:39:42:7434
5	1,11	156	91.335	16	1,982,032,139	3:27:30:3462	6,208,479,757	8:22:55:8228
5	1,17	156	91.263	13	2,369,951,412	3:38:26:2212	8,899,220,743	10:57:54:4306
5	1,23	156	91.368	16	2,013,735,362	3:17:47:5576	5,937,207,993	7:59:51:4535
5	1,29	156	91.263	13	2,357,303,914	3:42:43:3005	8,825,998,885	10:34:30:4996
5	1,31	156	91.487	14	1,944,185,789	1:44:57:6043	7,444,111,830	10:01:27:4149
5	1,41	156	90.973	14	2,103,450,639	1:35:26:3755	7,391,899,093	5:25:18:3458
5	1,43	156	91.335	15	2,023,046,470	1:34:33:2259	6,017,721,671	8:08:33:7267
5	1,47	156	86.676	12	937,795,472	1:00:07:3437	3,129,362,174	3:33:59:8125
5	1,53	156	89.470	14	1,366,079,584	1:01:51:9345	4,810,185,753	6:05:18:5671
5	1,61	156	90.973	14	2,086,462,019	1:35:39:5419	7,453,156,714	9:20:50:8353

7.4 Conclusions

This work has found no single algorithm which will yield optimal results in all cases of edge size and density of circulant k-hypergraphs and instead shows which algorithm will be better depending on the properties of the hypergraph being searched. Lack of knowledge of the structure of the graph will always lead to poor performance, as can be seen in the results of both the basic Backtracking and Russian doll algorithms.

When deciding between the new Russian Necklace algorithm presented in this work and the previous Necklace algorithm presented by Tzanakis et al. [17] for a given circulant hypergraph, two main metrics can be used. The first being the edge size of the hypergraph, which will be known before any analysis of the graph itself is done. The second is the density of edges in the graph, which can be found during the graphs generation, or as a static calculation before the graph is searched. As shown in the experimental results for random k-hypergraphs, higher density and lower edge size hypergraphs should be searched using the new Russian Necklace algorithm, while higher edge size or lower density hypergraphs should be searched using the Necklace algorithm.

For the 4-hypergraphs generated from LFSRs the best algorithm is the Russian Necklace algorithm which outperformed the Necklace algorithm by a factor of 4 in the largest instances. The Russian Necklace algorithm has been able to verify the optimality of the hypergraph generated by LFSRs labeled q5m4_1 and q5m4_7 in [17] by completing all exhaustive searches for $q = 5$ which was not previously achieved. The problem of searching hypergraphs where $q = 7$ or larger remains beyond the performance of this implementation.

8

Conclusions

In this thesis, we have analysed defining properties of circulant graphs, namely the cyclic inclusion of their edges, and defined circulant k-hypergraphs in a way to maintain these properties. These properties have allowed for further insight into the structure of the hypergraphs, which have been exploited in the search for max-cliques. Previous search techniques for max-cliques on graphs, namely the Backtracking, and Russian Doll searches, have been reviewed, and the extension of these techniques to searches on k-hypergraphs has been proposed and studied (Section 5.2). We presented the algorithm by Tzanakis et al. [17] in a general form to find cliques in general circulant k-hypergraphs (Necklace algorithm in Section 5.3).

By exploiting properties required to grow the max-clique between successive induced subgraph searches, we have defined an efficient algorithm to search for max-cliques in circulant k-hypergraphs, while rejecting isomorphic cliques that appear in a previous major step of the algorithm. The use of our new Russian Necklace algorithm in various

sizes and types of k-hypergraphs is summarized below.

As the search for max-cliques in hypergraphs requires an exponential number of nodes in the search tree, the procedures performed during each step of the search must be optimized. In Chapter 6, we have performed an analysis of two different algorithm structures for the candidate filtering step, as well as analysed the inclusion of bounding within the candidate filtering, to allow for an immediate exit when the candidate set is proven unable to produce a new max-clique.

In order to determine the performance of the Russian Necklace algorithm, as well as the Necklace algorithm, and both naive algorithms, we have performed a set of benchmarking experiments on random k-hypergraphs of varying density, size, and edge size (Section 7.2). The results of this experimentation provide a guide for selecting an algorithm to search a given circulant k-hypergraph, given its edge size and density. In addition to this comparison on random hypergraphs, we have performed searches on hypergraphs constructed from LFSRs to construct new covering arrays (Section 7.3). For theses problems, exhaustive search was only completed before up to $q = 4$, while we are able to complete exhaustive search up to $q = 5$. This gives definitive results on the largest covering arrays which can be formed from LFSRs where $q \leq 5$.

8.1 Conclusions on optimization of candidate filtering techniques

One of the most expensive procedures during each step of the max-clique search is the filtering of the candidate set to remove any vertices which are no longer valid members of the current clique. Different algorithm structures, and bounding techniques within this filtering can have an impact on the performance of this step, and therefore the performance of the search algorithm as a whole. Between the two major structures analysed in this work, namely the candidate-major, and clique-major structures, we have shown that the correct structure depends on the characteristics of the hypergraph being searched. In sparser hypergraphs, the clique-major structure has the best results, while in denser hypergraphs, the candidate-major structure has the lowest runtimes and

edge verifications. When adding bounding within the filtering algorithm, the size bound produces a universal decrease in both time and edge verification. The inclusion of the Russian doll bound produced mixed results, being both advantageous in reducing the runtime in some cases, while also being detrimental in others. The inclusion of the gap bound within the filter provided an improvement in overall search times when it was implemented, in both cases where it was used alone, and used in combination with other bounds.

8.2 Conclusions on the use of the Russian Necklace algorithm on random circulant k-hypergraphs

When compared with naive[1] algorithms such as Backtracking and Russian Doll, algorithms which take into account the circulant structure of the hypergraph will have faster searches. This was seen in the performance of the Necklace and Russian Necklace algorithms. These algorithms were faster across all hypergraphs searched, over all parameter sets. The strength of those naive algorithms is that they can be applied to any hypergraph to find a max-clique. When searching for novel results, which normally involves searching large hypergraphs with a specific structure, they are not generally usable.

The benefit of the Russian Necklace algorithm over the Necklace algorithm is dependent on two main parameters of the random circulant k-hypergraphs, namely the density of the hypergraph and the size of the edges. In general, k-hypergraphs with the highest density should be searched using the Russian Necklace algorithm. As the density lowers, the Necklace algorithms performance overtakes that of the Russian Necklace algorithm and should be used instead. The density where the performance of the Necklace algorithm improves on the performance of the Russian Necklace algorithm relies heavily on the size k of the edges of the k-hypergraph. As the edge size increases, so does the density where the two algorithms become equal in performance. From the experimental results in Chapter 7, random k-hypergraphs of 90% density and above should always be

[1]The term "naive" here refers to the algorithms that are unaware of the circulant structure of the hypergraphs.

searched with the Russian Necklace algorithm, random k-hypergraphs of 50% density and below should be searched with the Necklace algorithm, and for densities between these, it depends on the size of the edges.

8.3 Conclusions on the use of the Russian Necklace algorithm on circulant k-hypergraphs constructed via LFSRs

The strongest use-case for the Russian Necklace algorithm is in the search for cliques in k-hypergraphs generated by LFSRs. The high density and large size of these hypergraphs make the selection of the Russian Necklace algorithm the most appropriate when compared with the other algorithms examined in this work. In the smallest k-hypergraphs, the performance of the Russian Necklace algorithm did not greatly differ from that of the Necklace algorithm, these hypergraphs however were trivial in terms of the time required to search them. The performance difference of three to four times, as seen in the hypergraphs where $q \geq 4$, allows not only for faster searches, but also for more searches to be performed in the same amount of time. This last aspect is critical when trying to search over a larger number of hypergraphs in order to exhaust all possible constructions. The Russian Necklace algorithm was able to exhaustively search over the $q = 5$ open case to definitively determine the size of the covering array that could be constructed.

8.4 Future work

8.4.1 Data structures for hypergraphs and circulant hypergraphs

In Chapter 3, we discussed various data structures for graphs and hypergraphs, and implemented the one best suited for our applications. As the search for max-cliques in hypergraphs is not heavily studied, a full comparison between the techniques for their storage does not exist. Most important among these is in the area of the extended dimensions of adjacency maps. The two possible techniques of either using arrays of hashtables to add dimensions, or hashing a more complicated object such as a tuple could be compared to quantify the trade off between storage space, and the time taken

to verify the existence of an edge. This and other data structure studies could allow for larger hypergraphs to be searched, especially hypergraphs with larger edge sizes, as the storage technique used limited these searches.

8.4.2 Further research on Russian Necklace algorithm

Another area for future work is in exploring different variations on the Russian Necklace algorithm. This work can be divided into three main areas: parallelization of the algorithm, isomorph elimination, and tuning of the algorithm itself. Currently, the algorithm behaves sequentially, i.e each step is evaluated in order and is run to completion before the next step is begun. Full paralelization of the steps of the algorithm is not possible, as there is a reliance on the results of previous steps to accurately apply the Russian doll bound. It may be possible however to parallelize a certain number of steps together. At any step i, the initial clique is formed from an edge within the hypergraph, and so the candidate set, whose elements are used for the Russian doll bound, contains elements from the set $\{i+k, \ldots, n-1\}$. The Russian doll bound is therefore not reliant on the results of steps i to $i+k-1$. These steps could be run in parallel to increase the performance of the algorithm. This performance gain will increase with higher k values, which are harder to solve and more time consuming.

In the second area of improvement, the gap bound and inclusion of the vertex $n-1$ in the current clique do a great deal in avoiding isomorphic results. These bounds however do not fully eliminate the possibility of performing isomorphic searches during an individual step of the overall algorithm. One area of future work is in fully eliminating, if possible, the searching of isomorphic results. This work will require further examination of circulant k-hypergraphs, and their internal structures and properties.

Lastly, specific tuning of the implementation of the algorithm can be done to improve performance. Certain operations within each step of the algorithm are not needed during all steps of the algorithm. An example of this is the gap bound. Until the algorithm

is at step $\frac{n}{2}$, the gap bound will never be useful, as it is not possible to have a gap of vertices larger than $\frac{n}{2}$ when only considering $\frac{n}{2}$ vertices. The extra computation required during the calculation of the gap bound may be turned off during the earlier steps of the algorithm until it is possible for the gap bound to allow pruning.

8.4.3 Completing the search for covering arrays from LFRS for values $q \geq 7$

The problem of searching for max-cliques on hypergraphs constructed from LFSRs remains open when $q \geq 7$. In order for this problem to be solved using Russian Necklace, further work must be done in improving the Russian Necklace algorithm to reduce the search times required. By combining knowledge of the appropriate data structures for edge storage, with the improvements to the Russian Necklace algorithm listed above, the Russian Necklace algorithm may be used to complete some of these searches. The Russian Necklace algorithm itself does not benefit from precalculated lower bounds, however an optimized version of the Necklace algorithm could be used in this way. Using a heuristic search to find a large starting clique to use with the size bound, before the Necklace algorithm is run, would enable the Necklace algorithm to perform better bounding from the beginning, and reduce the overall time of the search.

References

[1] E. Balas and J. Xue. Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring. *Algorithmica*, 15:397–412, 1996. (Cited on page 45.)

[2] R. Carraghan and P. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, 1990. (Cited on pages 1, 3, and 39.)

[3] C. Colbourn. Covering array tables for t=2,3,4,5,6. URL: http://www.public.asu.edu/~ccolbou/src/tabby/catable.html. (Cited on page 100.)

[4] C. Colbourn. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)*, 58:121–167, 2004. (Cited on page 98.)

[5] S. Cook. The complexity of theorem-proving procedures. *STOC '71 Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. (Cited on page 1.)

[6] J. Duval. Factorizing words over an ordered alphabet. *Journal of Algorithms*, 4:363–381, 1983. (Cited on page 10.)

[7] M. Goodrich, R. Tamassia, and M. Goldwasser. *Data Structures and Algorithms in Java 6th edition, 619-625.* (Cited on page 33.)

[8] R. Karp. *Reducibility among Combinatorial Problems.* in Miller, R. E.; Thatcher, J. W., Complexity of Computer Computations. Pages 85-103. (Cited on page 1.)

[9] Kilom691. An example of a hypergraph. Retrieved on 14/06/2018 from https://en.wikipedia.org/wiki/Hypergraph/media/File:Hypergraph-wikipedia.svg. (Cited on pages 34 and 112.)

[10] R. Luce and A. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14:95–116. (Cited on page 1.)

[11] L. Moura, G. Mullen, and D. Panario. Finite field constructions of combinatorial arrays. *Des. Codes Cryptography*, 78:197–219, 2012. (Cited on page 98.)

[12] P. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120:197–207, 2002. (Cited on pages 1, 4, and 42.)

[13] Stanisław P. Radziszowski. Small ramsey numbers. *Electronic J. Combinatorics*, Dynamic Survey 1, 2014. (Cited on page 3.)

[14] F. Ruskey. *Combinatorial Generation (pre-publication), October 1, 2003,289 pages.* (Cited on pages 10 and 13.)

[15] F. Ruskey, C. Savage, and T. Wang. Generating necklaces. *Journal of Algorithms*, 13:414–430, 1992. (Cited on pages 3, 16, and 19.)

[16] Rutgers. Dimacs max-clique benchmark graphs.
URL: ftp:/dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/. (Cited on page 45.)

[17] G. Tzanakis, L. Moura, D. Panario, and B. Stevens. Constructing new covering arrays from lfsr sequences over finite fields. *Discrete Mathematics*, 339:1158–1171, 2016. (Cited on pages 3, 4, 53, 98, 100, 101, 103, and 104.)

[18] G. Verfaillie, M. Lemaitre, and T. Schiex. Russian doll search for solving constraint optimization problems. *Proceedings of the Association for the Advancement of Artificial Intelligence*, pages 181–187, 1996. (Cited on pages 2 and 42.)

[19] E. Weisstein. Fano plane. from mathworld–a wolfram web resource. Retrieved on 14/06/2018 from http://mathworld.wolfram.com/FanoPlane.html. (Cited on page 34.)

[20] D. Wood. An algorithm for finding a maximum clique in a graph. *Operations Research Letters*, 21:211–217, 1997. (Cited on page 45.)

List of Figures

List of Tables

List of Algorithms