# 1 Background

Extremal graph theory is concerned with the study of maximal or minimal graphs which satisfy particular properties. Extremality may be in regard to different graph invariants, for example the order or girth. This gives insight into how the global properties of a graph influence the local properties of subgraphs.

A classical result in extremal graph theory is by Turán.

**Theorem 1** (Turán). *Let $G$ be any graph on $n$ vertices, such that $G$ is $K_{r+1}$-free. Then the number of edges in $G$ is at most*

$$\frac{r-1}{r}\frac{n^2}{2} = \left(1 - \frac{1}{r}\right)\frac{n^2}{2}$$

Turán's Theorem is a a density result. Similar questions can be asked for other forbidden subgraphs, or even sets of forbidden subgraphs. These problems are often called Turán type problems.

At the opposite end of the spectrum, we can ask questions regarding the minimum degree of a graph. For example it can be shown that a graph of order $n$ with $\binom{n-1}{2}$ edges can contain an isolated vertex, even though almost every edge possible is present in the graph. This implies that graphs with very high densities can have no substantial structure on every vertex. To give some interesting results in this regard, it is often better to consider graphs with a minimum degree. This removes the trivial cases of isolated vertices. The following theorem due to Dirac is a nice result exploiting this consideration.

**Theorem 2** (Dirac). *A graph with $n \geq 3$ vertices is Hamiltonian if every vertex has degree $\frac{n}{2}$ or more.*

# 2 Definitions and Notation

The following section gives the definitions we are using for common words which may have some ambiguity within the realms of graph theory.

**Definition** (Graph). *A graph $G = (V, E)$ is an ordered pair comprising of a vertex set, $V$, and an edge set, $E$. No pair of vertices can connect to each other more than once, and no vertex can connect to itself.*

**Definition** (Degree/Valency). *The degree of a vertex, $v$, is the number of edges incident to $v$. Denote the degree of $v$ by $deg(v)$.*

**Definition** (Isolated Vertex). *We call $v$ isolated if $deg(v) = 0$.*

**Definition** (Path). *We call a graph a path of length $n$ if the vertices can be listed in order, $v_0, v_1, \ldots, v_n$ so that the edges are $v_{i-1}v_i$ for each $i \in \{1, 2, \ldots, n\}$.*

**Definition** (Cycle). *A cycle is a path with the added stipulation that $v_nv_0$ is an edge. We denote a cycle with $n$ vertices $C_n$.*

**Definition** (Complete graph). *A graph on $n$ vertices is complete if there is a path of length 1 between every pair of vertices. Denoted by $K_n$.*

**Definition** (Connected Graph). *We call a graph connected if there is a path from any vertex in the graph to every other vertex in the graph.*

**Definition** (Distance between two vertices). *The distance between two vertices, $u$, $v$, is the length of the shortest path from $u$ to $v$, denoted $d(u, v)$. If there is no path from $u$ to $v$, then $d(u, v) = \infty$ by convention.*

**Definition** (Eccentricity)**.** *The eccentricity of a vertex, $v$, is defined as $\epsilon(v) = \max\{d(v,u) : u \in V\}$. That is, it is the longest of all shortest paths.*

**Definition** (Radius)**.** *The radius, $r$, of a graph is given by $r = \min_{v \in V} \epsilon(v)$, that is, the smallest eccentricity of the graph.*

**Definition** (Diameter)**.** *The diameter, $d$, of a graph is given by $d = \max_{v \in V} \epsilon(v)$, that is, the largest eccentricity of the graph.*

**Definition** (Girth)**.** *The girth of a graph is the length of a shortest cycle in the graph.*

**Definition** (Regular Graph)**.** *A graph is said to be regular if $\forall u, v \in V$ it holds that $deg(u) = deg(v)$.*

**Definition** (Neighbourhood of a vertex)**.** *The neighbourhood of a vertex $v$ is the set of all vertices adjancent to $v$, denoted $neigh(v)$.*

# 3 Moore Graphs

A Moore graph is a regular graph of degree $d$ and diameter $k$ with

$$|V| = 1 + \sum_{i=0}^{k-1} (d-1)^i.$$

This is called the Moore bound. An equivalent definition, one which will be exploited later, is to define Moore graphs in terms of girth. A graph is a Moore graph if it has diameter $k$ with girth $2k+1$.

There are only a few known examples, with most graphs having been classified as not being Moore graphs. The known examples are

- Complete graphs, $K_n$ on $n > 2$ vertices.
- Odd cycles, $C_{2n+1}$
- The Petersen graph
- The Hoffman-Singleton graph

There is one possible graph left to be classified, it is of degree 57 and diameter 2. Should this graph exist some things are known about it. It is not vertex transitive (Higman). Mačaj and Širáň also proved that the order of the automorphism group of it, should it exist, is at most 375.

# 4 Extremal $C_t$ free Graphs

We call a graph, $G$, extremal $C_t$ free if $G$ has maximal cardinality edge set and girth $g$ at least $t+1$. The set of extremal $C_t$ free graphs is denoted by $EX(n;t) = EX(n;\{C_3, C_4, \ldots, C_t\})$. The size of the edge set is the extremal number $ex(n;t) = ex(n;\{C_3, C_4, \ldots, C_t\})$.

The rest of the discussion will be concerned about fixing $t = 4$, that is, we will talk about graphs of girth 5. It is know for all $n < 33$ what $ex(n;4)$ is. The Hoffman-Singleton graph satisfies the conditions to be extremal $C_4$ free, and is the order larger than 33 that we know the exact value of the extremal number. Should the conjectured Moore graph exist, then it would also satisfy the conditions to be extremal $C_4$ free. There are construtive lower bounds on $ex(n;4)$ for any $n$ which it seems current computer searches will be able to find a representative from $EX(n;4)$.

# 5 Naïve Approaches to finding $C_3$ and $C_4$ free graphs

My early attempts at computer searches for $C_4$ free graphs were slow and not well thought out. On reflection, there are some faults with the methods used, which have been corrected in subsequent computer searches.

## 5.1 Binary Integer Linear Program

As a graph can be considered as an array of binary variables, it is natural to consider a binary integer linear program (BILP) approach to completing a partially filled in adjacency matrix. It proved successful with small orders and a lot of known edges, but it did not succeed for larger orders or small orders without many predefined edges.

The model used was akin to the following model.

- $n$ is the number of vertices
- $\delta :=$ minimum degree.
- $\Delta :=$ maximum degree.
- $G_{i,j} = 1$ if $v_i v_j$ is a known edge, otherwise $G_{i,j}$ is not defined.
- $A$ is the adjacency matrix of the graph.

$$\sum_{i=1}^{n} A_{i,j} \geq \delta \tag{1}$$

$$\sum_{i=1}^{n} A_{i,j} \leq \Delta \tag{2}$$

$$A_{i,j} + A_{j,k} + A_{k,i} \leq 2 \tag{3}$$

$$A_{i,j} + A_{j,k} + A_{k,l} + A_{l,i} \leq 3 \tag{4}$$

$$A_{i,i} = 0 \tag{5}$$

$$A_{i,j} = A_{j,i} \tag{6}$$

$$A_{i,j} = G_{i,j} \tag{7}$$

Constraint (1) and (2) encodes the degree of each vertex being between a known minimum and maximum degree. Constraint (3) ensures there are no three cycles, similarly constraint (4) ensures there are no four cycles. Constrain (5) ensures no vertex connects to itself. Constraint (6) makes sure the adjacency matrix is symmetric. Constraint (7) ensure our known edges are included in our output.

The idea for this approach came from using a similar approach to finding mutually orthogonal latin squares. It had similar results for that problem, in that it worked for small cases, but not for anything noteworthy. The issue with this approach is in the symmetry of a graph. There are multiple isomorphic graphs which all have the same three or four cycle, which a solver struggles to deal with.

An approach for removing this issue of symmetry was the following:

1. remove the constraint for four cycles
2. solve the resulting model
3. find all squares in the solution from the adjusted model
4. add in a constraint disallowing each of the squares found in the previous step
5. repeat from step 2 until no squares are discovered

This method worked much better. It still failed on anything noteworthy, but it got to within one or two squares a lot of the time.

## 5.2 Search tree with poor cycle detection

### 5.2.1 Data structure for a graph

In this method, we use a double dimension array of integers to store the adjacency matrix of a graph.

### 5.2.2 Cycle detection

The powers of the adjacency matrix, $A$, of a graph can be used to count the number of three and four cycles in a graph. The number of three cycles is given by

$$\text{The number of three cycles in a graph} = \frac{\text{Tr}(A^3)}{6}. \tag{8}$$

This is straightforward to see, with the third power of the adjacency matrix giving the number of paths of length three. The division by six comes from repeated counting, each three cycle is counted twice for the direction you travel it, and three times for every vertex in the three cycle.

The number of four cycles is more complex. There is information contained in $\text{Tr}(A^4)$ that tells us about the four cycles, but there is extra information as well which needs to be removed. Assume $v, u, w \in V(G)$ for the following. Then the extra information can be summarised as follows:

- if $v$ is adjacent to $u$, then there is a path of length four from $v$ to $v$ given by $v, u, v, u, v$. The total amount of this extra information is
$$\sum_{v \in V(G)} \deg(v) = \text{Tr}(A^2).$$
- A walk of length two can be reversed to give a walk of length four. For example, a walk of $v, u, w$ can be made into $v, u, w, u, v$. The number of walks of length two need to be removed as well, this information is given by the sum of all non diagonal elements in $A^2$.
- Let $v$ be adjancent to both $u$ and $w$. Then there is a walk of length four $v, u, v, w, v$. There are
$$\sum_{v \in V(G)} \binom{\deg(v)}{2}$$
ways to pick these paths.
- Lastly, everything mentioned so far, and all the four cycles, is detected eight times. So we over count eight times the true number, so a division by eight is required.

These points give us the following formula

$$\text{Number of four cycles} = \frac{\text{Tr}(A^4) - \sum_{i,j}(A^2)_{ij} - \sum_{v \in V(G)} \binom{\deg(v)}{2}}{8}. \tag{9}$$

Equations (8) and (9) used in conjunction determine if a graph is of girth at least five.

### 5.2.3 A base graph

From here on out, the discussion will be about a graph on 33 vertices, that has been the focus of investigation, although a lot of the techniques described can be used on any graph of girth five.

The graph in question is the following, and will be refered to as *the base graph* from hereon.

tikzpicture will go here showing the base graph

### 5.2.4 Search tree

A rooted binary tree can be used to generate graphs with more edges than the base graph, and still keep a girth of five. The algorithm used in this approach is quite simple. It uses a stack to do a depth first search of the search tree, with the stack having one node in the tree to begin, the base graph, which is the root of our tree. The outline of the approach follows.[1]

1. Push the root tree onto the stack
2. Pop the top graph off the stack and store it in a variable for active manipulation
3. If the number of edges in the active graph is the conjectured maximum, stop.
4. Verify that the graph is of girth at least 5. If it's not, go to (2)
5. Generate the children of the stored graph
6. Push the left child onto the stack
7. Push the right child onto the stack
8. Go to (2)

In step (5) we need to generate the children of a graph. To generate the *left* child we leave the matrix as it is and increment which entry in the adjacency matrix we look at next time. To generate the *right* child we add an edge at the entry in the adjacency matrix we are looking at, then incrememnt which entry in the adjacency matrix we look at next time.

The purpose of pushing the left child before the right child is to explore the right portion of the tree first, as that is where it is suspected a solution will lie should it exist.

In step (4) we simply need to verify that equation (8) and (9) are both zero. That is, there are no three or four cycles. In practise, a slight modification of the two equations are used as the divisions are not required if the answer is zero, it will continue to be zero, and if the answer is non zero, it will continue to be non zero.

In verifying the graph is of girth at least 5, there are a number of other steps that can be taken as well, which have been omitted currently. There is more structure known about the graph which allows for more pruning conditions.

This method turned out to be very slow as step (4) required calculating $A^4$.

## 5.3 Search tree with good cycle detection

### 5.3.1 Requirements of implementation language and hardware

It is important to note the operations the programming language used to implement this method must have. The following are required of the programming language:

- a guaranteed 32 bit unsigned integer (denoted uint32_t)
- logical *and*, *or*, *not* (denoted &, |, and $\sim$ respectively)
- unsigned bitshift (left bitshift denoted by << and right bitshift by >>)

and it would be benificial to have the following functions on the hardware for increased efficiency

- popcount(n) which counts the number of set bits in a binary string

[1] going to update this with something from an algorithm package so it looks better

### 5.3.2 Data structure for a graph

The structure used is now an array of 32 bit integers of length 32. We only need to worry about 32 of the vertices as we can guarantee one vertex in the base graph is not going to be a part of a three or four cycle, and it won't ever be manipulated. The connection between two vertices, $i$ and $j$ is represented by the $j$th bit in the $i$th index of the array, and because of the symmetry of the adjacnecy matrix, vice versa. It is convienent to also have the following constants stored so as not to waste time calculating them frequently:

- DEPTH_MASK = ($\sim$((uint32_t) 0) >> (32 - 22))
- N = (1 << 31)

It is also worth noting that the depth a particular matrix is down the search tree is stored within the adjacency matrix itself, rather than as a seperate variable. This is the purpose of the constant DEPTH_MASK, to remove it before checking for three or four cycles. The constant N is used to detect if there is a bit set in the first entry of a uint32_t variable, which is where we store if a graph has a three or four cycle in it.

### 5.3.3 Cycle detection

We now don't have a natural way to multiple two matrices together to generate a power of a matrix as we've limited the amount of space we're using to represent the connection of two edges to a single bit. We will exploit the fact that we only ever manipulate one connection before checking the girth again to get around calculating powers of matrices.

The following assumes that the edge connecting vertices $i$ and $j$ has just been added.

To check for three cycles, we only need to check if the neighbourhoods of $i$ and $j$ share a vertex. We conviently have a sorted list of the vertices in the graph, and an indicator if they are adjancent for every vertex. The we simply need to do a logical *and* on the entries in the array of 32 bit integers corresponding to vertex $i$ and $j$, and if the resultant is greater than zero, then the intersection of their neighbourhoods is non empty.

Checking for four cycles is more complex. It requires checking to see if vertex $u \neq j$ is not in the neighbourhood of $j$, if it isn't then checking that the intersection of $j$ and $u$'s neighbourhoods have at least two elements in common, $v$ and $w$. If these conditions are satisfied then we have a four cycle $j, v, u, w, j$. So we need to verify that these conditions are not met for all $u \in V(G)$. Checking if $u = j$ is trivial, so assume that $u \neq j$. Then the objective is to make sure the neighbourhoods of $j$ and $u$ don't share two or more vertices. This is achieved through the popcount function. So if we take the popcount of the logical *and* of vertex $u$ and $j$ and do a popcount on the resultant we end up with the cardinality of the intersection of the neighbourhoods of $u$ and $j$. Repeated this for all $u \in V(G)$ and if all of the intersections are less than two, then there is no four cycle in the graph.

### 5.3.4 Search tree

Essentially the same method is used as in section 5.2, a rooted binary tree is used to generate graphs with more edges than the base graph. The algorithm used is the same, with the only difference being in implementation rather than the ideas being used. We still use a stack to do a depth first search, prioritising the right portion of the tree first. We have added a new pruning condition as well, we record the depth a graph is in the tree so we can verify there are enough edges left to reach a lower bound on the maximum number of edges. If we have $n$ edges already set in the graph, and we are only $m$ levels above the maximum depth possible, and the smallest number of edges desired in the graph is $m+n+k$ where $k > 0$, then we don't generate any children of that graph.

This method is much faster than the prior methods as no powers of matrices are required, and even better than that, most operations that are conducted frequently are bitwise operations, which are quick. The only function that may take a long portion of time to calculate is the popcount function, but this is inbuilt on a lot of CPUs now.

# 6 Mask Approach to finding $C_3$ and $C_4$ free graphs

## 6.1 Requirements of implementation language and hardware

As well as the requirements mentioned in section (5.3.1) it would also be benificial to have the following functions on the hardware for increased efficiency

- clz($n$) which counts the number of leading zeroes in a binary string.

## 6.2 Data structure for a graph

The structure used is mostly the same as in section 5.3.2, but with the addition of another array of 32 bit integers of length 32, which we are going to call a *mask*. For any given $C_3$ and $C_4$ free graph, the mask will have a 1 in position $i, j$ if an edge can be added to the graph between vertices $i$ and $j$ such that the graph will remain $C_3$ and $C_4$ free, and a 0 otherwise.

## 6.3 Cycle detection

Cycle detection is now not done in generating each child as we know that if we add an edge that is active in the mask then it is not going to generate a $C_3$ or $C_4$ subgraph. Instead, every time an edge is added, we have to update the mask by checking to see if adding any of the possible edges causes a new $C_3$ or $C_4$ subgraph, and if so, changing the entry in the mask to a 0. This may seem longer at first, but it becomes clear quite quickly that both theoretically and experimentally this approach is much quicker. If we can eliminate edge $uv$ at depth $n$ in the search tree, then it only needs to be eliminated once, rather than every time at depth $n + k$, where $k$ is the difference between the current depth and the depth $uv$ would be checked in the non masked approach. This gives an upper bound on the total number of nodes that don't need to be search as a result of this approach of $2^k$.

While updating the mask, checking to see if any edge we add would cause a $C_3$ or $C_4$ subgraph is done in the same way cycle detection is done in 5.3.3. To speed up the process of finding which edges are candidates to be added to the graph, find the first non zero entry in the mask array, and then use the clz($n$) function to find the first occurance of a bit set, which then gives the $i, j$ entry required[2].

## 6.4 Search tree

The same underlying method is used as in 5.3.4, although some of the pruning conditions can be made more aggressive now. We can improve the pruning based on depth to instead prune if the number of bits set in the graph and the number of bits set in the mask summed come to less than the desired number of edges required. Although not mentioned earlier, there is a condition to prune if the degree of any vertex is too small or big. It is now possible to tell if it is too small at any point while manipulating that vertex. If the sum of the number of edges already set from that vertex, and the number of possible edges left to be added is less than the minimum degree required, then we can prune at that point.

# 7 Multithreaded Search Tree

So far everything that has been mentioned has implicitly assumed running on a single thread, no care has been taken in ensuring thread safety. Being able to explore the search tree with more than one thread would be ideal though, although care needs to be taken to ensure the same sub trees are not being searched and a possible graph is not being discarded.

One approach is to have a vector of $n + 1$ stacks, where $n$ is the number of threads that are used to search the tree. The stack in position 0 is going to be called the *main* stack, and the

[2]Need to add the actual algorithm because it's more sophisticated than this.

other $n$ stacks are called *worker* stacks. We're also going to have $n + 1$ threads, with the main thread calling $n$ other threads, which are called *worker* threads. Each worker thread is given a unique identification number 1 through to $n$, which is a parameter of the function used to invoke the thread.

The main stack is filled by a breadth first search of the tree until $kn$ elements are in a queue, where $k$ is a constant that should be bigger than 30, although this number will need to be adjusted based on experimental trials per problem. That queue is then transfered to the main stack.

After the main stack has all of the elements generated, the worker threads are spawned. Worker thread $i$ places the top ten (again, this number will need to be changed depending on the problem) elements of the main stack onto stack $i$, for all $i \in \{1, \ldots, n\}$. When the worker thread is accessing the main stack, it needs to lock access to just it's thread so as not to search the same subtree with different threads, or remove an element that has not be copied yet. This does result in blocking, and is a slow down at the beginning of the search when all $n$ worker threads want access to the main stack, this is a situation that can not be helped.

After the worker stack has ten elements, the worker thread unlocks access to the main stack and start searching the tree using the elements it has access to. Should the stack end up empty, it will once again grab ten more elements from the main stack, again blocking the other worker threads from accessing it. This is less of an issue as it will be very rare two worker threads want to access the main stack at the same time after they all get the initial ten elements.

The reason that $k \geq 30$ is picked is for balancing the worker threads. If $k = 10$, then it is conceivable that one worker stack will produce a very small subtree and the associated thread will finish quickly, without any more work to do. This would be a waste of resources. But with $k \geq 30$, if one worker stack ends up producing a small subtree, then there is more work for it to do, with the idea being that small subtrees are going to be evenly distributed amoung all threads.