When do neural codes come from convex or good covers?

Caitlin Lienkaemper

July 23, 2015

Abstract

How does your brain keep track of your position in space? In a familiar environment, hippocampal neurons known as place cells become associated to regions of space known as their receptive fields. The firing patterns of place cells form a neural code. Modeling receptive fields as convex open regions of Euclidian space agrees closely with experimental results. Given a set of place fields, it is easy to determine the neural code which describes the place fields. Given a neural code, how and when can we find a set of convex place fields which generate the neural code? Neural codes on up to four neurons were classified by Curto et. al. We extend this classification to five neurons. Additionally, it is known that if a neural code has local obstructions, it is not convex. We provide a counterexample showing that the converse is not true in general. We conjecture that if a neural code has no local obstructions, it can be realized as a good cover.

1 Introduction

The brain's ability to represent and navigate within the physical world is fundamental to our everyday experience and ability to function. How does the brain represent space? John O'Keefe, May Britt Moser, and Evard Moser won the 2014 Nobel Prize in Physiology and Medicine for their discovery of place cells, grid cells, and head direction cells, all of which take part in rodents' and other animals' mechanisms for representing, navigating through, and forming memories of their environments [1]. Here, we focus on place cells, hippocampal neurons which become associated to regions of space known as their receptive fields. When an animal is in a receptive field, the place cell associated to that receptive field fires at a much higher rate than when it is outside that receptive field. If we have access to a record of both an animal's location and its place cell firing patterns, we can determine an encoding map, that is, a correspondence between place cell firing patterns and positions, and can use it to interpret further neural activity [1][3]. However, the brain does not have access to location data independent of neural activity, and must thus have a method for encoding meaningful information about spatial relationships through neural activity alone. Thus, our central question matches that of Curto, Itskov, Veliz-Cuba, and Youngs: "What can be inferred about the underlying stimulus space from neural activity alone?"[3]. An answer to this question would also be useful to neuroscientists working in conditions when location data is not available but neural activity data is, for instance, when investigating neural activity of subjects who are merely dreaming about or remembering a location.

Under the assumption that receptive fields form a good cover, the homotopy groups of the stimulus space can be retrieved from the neural code. Receptive fields have been observed to be convex, and covers composed of open convex sets are always good covers [1]. Therefore, as a preliminary to our central question, we are interested in when neural codes come from good covers and when neural codes come from convex open covers. Curto et al. proved that a neural code which has local obstructions cannot come from a good cover (and, thus, cannot come from a convex open cover), proved that the converse of this theorem holds true when considering up to four neurons, and classified all neural codes on up to four neurons [**?**]. Here, we present a classification of neural codes on five neurons, up to isomorphism, and procedures and source code which may be used to push these results into larger cases. We present a five neuron counterexample to the conjecture that all neural codes with no local obstructions are convex. As an intermediate step in this

effort, we produce a list of all simplicial complexes on five vertices, which is likely of general interest beyond this application. We turn our attention the weaker form of the converse, that all neural codes with no local obstructions come from good covers, discuss some strategies we have used towards its proof, and prove it in some special cases.

Section 2 provides background material on neural codes in general and describes our notation and assumptions. Section 3 motivates the definition of a local obstruction. Section 4 gives an overview of the computational work involved in classifying neural codes on 5 neurons. Section 5 presents our counterexample. The remainder of the paper outlines strategies we have tried in proving the that neural codes with no local obstructions come from good covers. Finally, we discuss future directions and biological applications.

2 Background

We approximate neural activity as binary: under this model, neurons are either firing or they are not. We encode the combinatorial data generated by the firing patterns of place cells as a neural code.

Codeword Given a set of neurons numbered $\{1, 2, \ldots, n\} = [n]$, a *codeword* is a subset of $[n]$ representing a set of neurons that fire together while no other recorded neurons fire.

For instance, $\{1, 2, 3\}$, $\{2\}$, and $\{\}$ are valid codewords on three neurons. We see that the set of codewords is the power set of $[n]$, thus there are 2^n codewords on n neurons.

Neural Code A *neural code* is a set of codewords, and represents the set of firing patterns a collection of neurons take over a period of time.

For instance,

$$\{\{1, 2, 3\}, \{2\}, \{1, 3\}, \{\}\}$$

is a valid neural code on three neurons, and means that at various times, neurons 1, 2 and 3 fire all together, neuron 2 fires alone, and neurons 1 and 3 fire together. For convenience, when it does not induce confusion, we omit the inner brackets and the commas between numbers. By convention, we always assume that neural codes contain the empty set. Thus

$$\{123, 2, 13\}$$

is interpreted as referring to the code

$$\{\{1, 2, 3\}, \{2\}, \{1, 3\}, \{\}\}.$$

Since each neural code is an element of the power set of the set of 2^n codewords, but we only accept neural codes that contain the empty set, there are 2^{2^n-1} neural codes on n neurons.

Maximal Codeword A codeword is maximal if it is not contained in any other codeword.

For instance, 123 is the only maximal codeword in $\{123, 2, 13\}$ because both 2 and 13 are contained in 123.

In this paper, we will focus entirely on neural codes which come from place cells. Neural codes on place cells encode information about a stimulus space, the part of the real world in which the animal is located. We define the stimulus space to be a subset of $\mathbb{R}^d$. To each neuron, we associate a receptive field: a subset of the stimulus space in which that neuron fires.

Represents Let $\mathcal{U} = U_1, \ldots, U_n$ be a collection of sets. $\mathcal{U}$ *represents* a neural code $\mathcal{C}$ if, for $\lambda \subset [n]$ the codeword λ is in $\mathcal{C}$ whenever $\bigcap_{i \in \lambda} \{U_i\} \setminus \bigcup_{i \notin \lambda} U_i$ is nonempty.

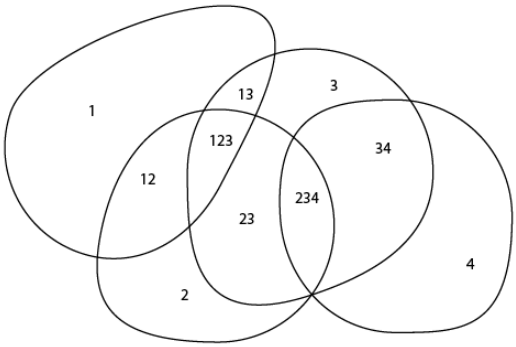

Figure 1: A set of four place fields representing the neural code $\{123, 234, 12, 13, 23, 34, 1, 2, 3, 4\}$.

Less formally, to write a neural code represented by a collection of sets taken as receptive fields, we number the sets and include a codeword in our code if the sets it references intersect outside of the intersection of any other sets. For an example of the encoding processs, see Figure 1.

While it is easy to generate a neural code from a set of receptive fields, it is unclear how to find a set of receptive fields that generate a given neural code. If we place no restrictions on our receptive fields, we can always find a set of place fields to generate a neural code. For instance, we can let there be a an open interval of the real line corresponding to each codeword, and then let each receptive field be the union of the intervals corresponding to codewords that neuron appears in. However, we are not interested in this sort of receptive field, because codes generated from receptive fields like this reveal nothing about the stimulus space, do not correspond to anything biological, and are not mathematically interesting. We revise the question by placing some requirements on our receptive fields. Experimentally, receptive fields have been observed to be convex. This motivates the following definition of a convex neural code.

Convex Neural Code A neural code $\mathcal{C}$ is *convex* if there exists a collection of convex open sets $U_1, U_2, \ldots, U_n \subset \mathbb{R}^d$ which represents $\mathcal{C}$. The *minimal embedding dimension* of a convex neural code is the smallest value of d for which this is possible.

We recall that every convex set is contractible, and that any intersection of a collection of convex sets is convex, and therefore also contractible. Thus, any collection of convex sets forms a good cover.

Good Cover A collection of open sets $\mathcal{U}$ is a good cover if each $U \in \mathcal{U}$ is contractible, and if any nonempty intersection of finitely many sets in $\mathcal{U}$ is contractible.

We gain much insight into neural codes by studying simplicial complexes associated to them.

Abstract Simplicial Complex An *abstract simplicial complex* Δ is a set of subsets of a vertex set which is closed under the operation of taking subsets. That is, if $\sigma \in \Delta$ and $\tau \subset \sigma$, then $\tau \in \delta$. If $\sigma \in \Delta$, then we call σ a face of Δ, if $\tau \in \sigma$, we call τ a *face* of σ.

We will work only with simplicial complexes with a finite vertex set, which we will take to be $[n]$ for some integer n. We often identify an abstract simplicial complex Δ with its geometric realization $|\Delta|$. To construct $|\Delta|$ given Δ, we construct n affinely independent points in $\mathbb{R}^d$ for some d and let the simplex on $i_1, i_2, \ldots, i_m$ be the convex hull of these m points. When it does not induce confusion, we will make no notational or other distinction between abstract and geometric simplicial complexes.

Since a simplicial complex is a collection of subsets of its vertex set which must be closed under the operation of taking subsets, the abstract simplicial complex of a neural code consists of the set of codewords together with all subsets of those codewords. For instance, the simplicial complex on the code $\{12, 2, 23\}$

is $\{1, 12, 2, 23\}$, since we must add the codewords 1 and 3 to make this code closed under taking subsets. We see that while each neural code has one simplicial complex, in general many neural codes will have the same simplicial complex. This is because the simplicial complex of a code is determined entirely by the set of maximial codewords.

Nerve of a Cover Let $\mathcal{U}$ be a collection of sets.The nerve of $\mathcal{U}$, $N(\mathcal{U})$, is the simplicial complex whose vertex set is $\mathcal{U}$, and where we let there be a simplex on the collection $\{U_\alpha\}_{\alpha\in\lambda}$ if $\bigcup_{\alpha\in\lambda} U_\alpha$ is nonempty.

We now restate a classic theorem, presented well in [6]. We note that the theorem name predates any application to neuroscience.

Theorem 1 (The Nerve Theorem)**.** *If $\mathcal{U}$ is a good cover, then $N(\mathcal{U})$ is homotopy equivalent to $\mathcal{U}$.*

Then if receptive fields are convex, or if they just form a good cover, the simplicial complex of a neural code is homotopy equivalent to the union of the set of receptive fields. Thus, if receptive fields form a good cover of the stimulus space, the homotopy type of the stimulus space is recoverable from place cell activity alone [3]. Thus there is a good evolutionary reason for place fields to form a good cover.

3 Local Obstructions

Curto et. al recently proved that if a neural code has local obstructions, then it cannot come from a good cover [3]. We motivate and define the term local obstruction and give an example of how local obstructions indicate that codes do not come from good covers.

Example 1 Consider the neural code $\mathcal{C} = \{12, 13\}$. What happens when we try to find a collection of convex open sets U_1, U_2, and U_3 generating $\mathcal{C}$? We see that $U_1 = U_2 \cup U_3$ and $U_2 \cap U_3 = \emptyset$; the sets U_2 and U_3 form a disconnection of U_1. A convex set must be connected, thus there is no way to make U_1 a convex set, so this neural code is not convex.

Why else might a code fail to be convex?

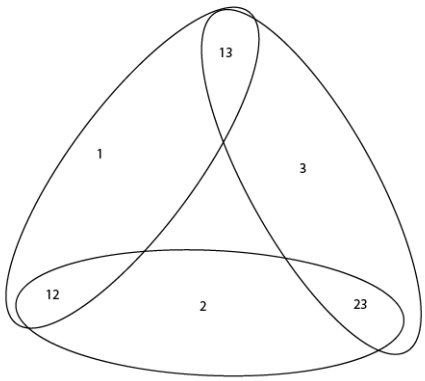

Figure 2: We see that $U_1 \cup U_2 \cup U_3$ is not contractible.

Example 2 In this case, we found that the structure of the code forced one place field not to be connected. Now, consider the neural code $\mathcal{D} = \{14, 124, 24, 234, 34, 134\}$. If we try to find convex open sets U_1, U_2, U_3, and U_4 which generate $\mathcal{D}$, we see that $U_4 = U_1 \cup U_2 \cup U_3$. Unlike last time, this does not force a disconnection of U_4, since U_1, U_2, and U_3 are not disjoint. However any way we try to draw U_4, we will fail. The problem is that U_4 must have a hole, since $U_4 = U_1 \cup U_2 \cup U_3$ which has a hole any way we draw it. Thus, we cannot draw U_4 as a contractible open set, so we cannot construct a good cover yielding $\mathcal{C}$.

We formalize this notion as follows:

Local Obstruction The neural code $\mathcal{C}$ has a *local obstruction* if there exists $\sigma \notin \mathcal{C}$ such that σ is the intersection of maximal codewords in $\mathcal{C}$ and $Lk(\sigma)$ is not contractible.

We see that the code $\mathcal{D} = \{14, 124, 24, 234, 34, 134\}$ has a local obstruction, since 4 is the intersection of maximal codewords 124,234, and 134, but $Lk(4) = \{1, 12, 2, 23, 3, 13\}$, which is not contractible. We see that $Lk(4)$ is the nerve of U_1, U_2, and U_3, so if U_1, U_2, and U_3 form a good cover, by the nerve theorem, $U_1 \cup U_2 \cup U_3$ is homotopy equivalent to $Lk(4)$. Thus, $U_4 = U_1 \cup U_2 \cup U_3$ is not contractible, and therefore is not convex.

We note several theorems and conjectures regarding local obstructions, they are properly treated in [4].

Theorem 2. *If a neural code is convex, then it has no local obstructions [4].*

In fact, they proved a slightly stronger theorem:

Theorem 3. *If a neural code comes from a good cover, it has no local obstructions [4].*

The converses to these two theorems have been conjectured.

Conjecture 1 Any code with no local obstructions is convex.

While this conjecture held true on up to four neurons, we present a counterexample on five neurons.

Conjecture 2 Any code with no local obstructions can be realized as a good cover.

We present some strategies for proving this.

4 Computational Work

Previously, neural codes with up to 4 neurons have been classified as having local obstructions or not. These classifications were done by hand. We automated this proccess using SageMath [8]. Our algorithm follows closely from the definition of a local obstruction. Given a neural code, we:

- create the simplicial complex of the code
- list all intersections of maximal faces of the simplicial complex
- find the link of each intersection of maximal faces
- find the homology groups of each link
- if any of these homology groups are nontrivial, we add the intersection of maximal faces to a list of codewords which must be added to the code for it to have no local obstructions
- we return the list of codewords which must be in the code in order for there to be no local obstructions.

We note that this approach will fail eventually, as having trivial homology groups does not imply contractibility in general, but does in these small cases.

While the naive approach to classifying all neural codes on 5 neurons would be to generate the list of all neural codes on 5 neurons and then check each one for local obstructions, this turns out to be somewhat impractical, as we recall that there are $2^{2^5-1} = 2^{31} = 2,147,483,648$ neural codes on 5 neurons. Instead, we used Nauty [7] to generate a list of all connected simplicial complexes on up to 5 vertices, up to symmetry. We worked only with connected simplicial complexes because any disconnected simplicial complex on five neurons can be expressed as the disjoint union of simplicial complexes on fewer than five vertices, and has thus been dealt with in previous work. We found that there are 157 connected simplicial complexes on 5

vertices, 14 connected simplicial complexes on 4 four vertices, 3 connected simplicial complexes on 3 vertices, and 1 connected simplicial complex on 0, 1, and 2 vertices. [**?**] Including disconnected simplicial complexes brings these counts to 1,2,5, 20, and 180, respectively, agreeing with the number of antichain covers of an $n-$set found listed in [9]. There is a clear bijection between the set of simplicial complexes on n vertices with the number of antichain covers of an $n-$set. Therefore, we can use the result in [9] that there are 16,143 antichain covers of an $n-$set to assert that there are 16,143 simplicial complexes on six vertices, up to symmetry. We do not produce a list of all simplicial complexes on six vertices with a description of all neural codes on each, since we view this as too large a data set to be useful at the moment.

Source code and the list of simplical complexes on five vertices with all required codewords can be found in the appendix.

In our list of simplicial complexes, we noticed that every code on a connected simplicial complex other then the simplicial complex $\{01234\}$. This motivates us to prove that if $\mathcal{C}$ is a neural code with no local obstructions on a simplicial complex $\Delta(\mathcal{C})$ which contains a nonempty intersection of maximal elements, then $\mathcal{C}$ must contain at least one codeword σ which is the intersection of maximal elements. An alternate way to phrase this is that unless Δ is the disjoint union of complete simplicial complexes, there exists $\sigma \in \Delta$ such that $Lk(\sigma)$ is not contractible.

Theorem 4. *Every finite simplicial complex which has a nonempty intersection of some maximal faces contains an simplex whose link is not connected.*

We present two slightly different proofs.

Proof. In a finite simplicial complex, there exists a simplex σ such that σ is the intersection of maximal faces $M_1 \cap M_2 \cap \cdots \cap M_n$, but σ is not contained in any other intersection of maximal faces. Then $Lk(\sigma)$ contains the simplexes $M_1 \setminus \sigma \ldots, M_n \setminus \sigma$. If $Lk(\sigma)$ were connected, then there would exist M_i and M_j such that $M_i \cap M_j \setminus \sigma$ was nonempty. Thus $M_i \cap M_j = \tau$, $\sigma \subset \tau$. This contradicts our assumption that σ is not contained in any other intersection of maximal faces. Thus, every finite simplicial complex which has a nonempty intersection of maximal faces contains a simplex whose link is not connected. □

Proof. Let $M_1, \ldots, M_n$ be some maximal faces of a simplicial complex, $\sigma = M_1 \cap \cdots \cap M_n$. Then $Lk(\sigma)$ contains $M_1 \setminus \sigma, \ldots, M_m \setminus \sigma$. (Note that we change the index because σ may be in more maximal faces than just $M_1, \ldots, M_n$. If $Lk(\sigma)$ is contractible, then it is connected. Thus there exist two maximal faces, M_i and M_j, for which $(M_i \setminus \sigma) \cap (M_j \setminus \sigma)$ is nonempty, since otherwise, $Lk(\sigma)$ cannot be path connected. Let $M_i \cap M_j = \tau$. Since M_i and M_j share at least one vertex outside σ, σ is properly contained in τ.

Thus, if σ is an intersection of maximal faces whose link is contractible, σ is a proper subset of another intersection of maximal faces. However, the set of intersections of maximal faces is finite, so there must exist an intersection of maximal faces which is contained in no other intersection of maximal faces. Therefore, if there exists a nonempty intersection of maximal faces, then there must exist an intersection of maximal faces whose link is not connected.

□

Incorporating this result into our algorithm for checking for local obstructions would speed it up in many cases. Since we know that if σ is an intersection of maximal faces that is contained in no other intersection of maximal faces, we do not have to compute $Lk(\sigma)$ or its homology groups to know that $Lk(\sigma)$ must be disconnected and thus not contractible. This would let us skip some of the slower steps of our computation.

5 A Non-Convex Code With No Local Obstructions

In our list of neural codes on five neurons, we found a neural code with no local obstructions that it not convex. We present it as a counterexample to the conjecture that all codes with no local obstructions are convex. We first verify that the code

$$\mathcal{C} = \{1234, 012, 034, 013, 12, 34, 13, 01, 03, 3, 1\}$$

has no local obstructions. Its maximal codewords are

$$\{1234, 012, 034, 013\},$$

whose intersections are

$$\{12, 34, 13, 1, 3, 0\}.$$

Of these, only 0 is not in the code. However,

$$Lk(0) = \{12, 34, 13, 1, 2, 3, 4\},$$

which we can verify is contractible.

We now prove that $\mathcal{C}$ is not convex. Suppose to the contrary that there exist convex open sets U_1, U_2, U_3, and U_4 which represent $\mathcal{C}$. We take

$$p \in U_1 \cap U_2 \cap U_3 \cap U_4 = U_{1234},$$

$$q \in U_0 \cap U_1 \cap U_2 = U_{012},$$

and

$$r \in U_0 \cap U_3 \cap U_4 = U_{034}.$$

Since $p, q \in U_1 \cap U_2$ which is convex, the line segment $\overline{pq}$ is in $U_1 \cap U_2$. Likewise, $\overline{pr} \in U_3 \cap U_4$ and $\overline{qr} \in U_0$. The line segment $\overline{qr}$ begins in $U_0 \cap U_1 \cap U_2$ at q. To get to $r \in U_0 \cap U_3 \cap U_4$ without leaving U_0, it must pass directly from $U_0 \cap U_1 \cap U_2$ into $U_0 \cap U_1$, from $U_0 \cap U_1$ to $U_0 \cap U_1 \cap U_3$, and finally from $U_0 \cap U_1 \cap U_3$ into $U_0 \cap U_3$ and then $U_0 \cap U_3 \cap U_4$. Any deviation from this path would contradict either the openness or the convexity of U_0, U_1, U_2, U_3, or U_4. Thus, there exists the point $s \in U_0 \cap U_1 \cap U_3 \cap \overline{qr}$. There also must exist points $t_0 \in \overline{pq} \cap U_{1234}$ and $u_0 \in \overline{pr} \cap U_{1234}$, since U_{1234}, so there must exist a neighborhood around p which is contained in U_{1234} and we can take u_0 and t_0 in this neighborhood, as pictured in Figure 5

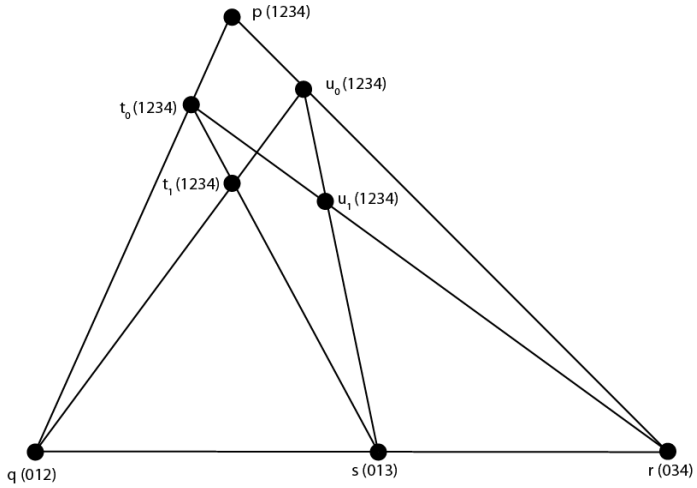

Figure 3: We see that p, q, and r form a triangle, with $s \in \overline{qr}$. Take $t_0, u_0 \in U_{1234}$.

Next, we introduce the line segments $\overline{su_0}$ and $\overline{st_0}$ and see that both line segments are contained within $U_1 \cap U_3$. We now introduce the line segments $\overline{t_0 r}$ and $\overline{u_0 q}$. By the convexity of $U_3 \cap U_4$,

$$\overline{t_0 r} \subset U_3 \cap U_4$$

and by the convexity of $U_1 \cap U_2$,

$$\overline{u_0 q} \subset U_1 \cap U_2.$$

Thus, the intersection

$$u_1 = \overline{su_0} \cap \overline{t_0 r} \subset U_1 \cap U_3 \cap U_4$$

and

$$t_1 = \overline{st_0} \cap \overline{u_0 q} \subset U_1 \cap U_2 \cap U_3.$$

However, the codewords 123 and 134 do not appear in $\mathcal{C}$, so if a point is in $U_1 \cap U_2 \cap U_3$ or $U_1 \cap U_3 \cap U_4$, then it is in U_4 and U_2 as well. Thus,

$$t_1 \subset U_{1234}$$

and

$$u_1 \subset U_{1234}$$

This leaves us in a position to repeat what we have done, since we see that the triangle whose vertices are q, r and the intersection of $\overline{qu_0}$ and $\overline{rt_0}$ is again a triangle with vertices in 012, 034, and 1234 and two designated points in 1234, t_1 and u_1. In general, we define the points

$$t_{n+1} = \overline{qu_n} \cap \overline{st_0}$$

$$u_{n+1} = \overline{qt_n} \cap \overline{su_0}.$$

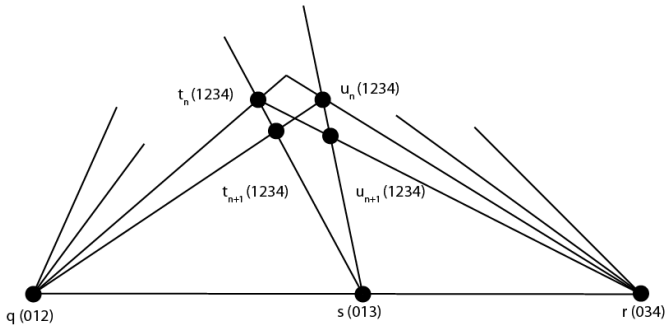

Figure 4: We see how to construct t_{n+1} and u_{n+1} from t_n and u_n.

We see that each $t_n \subset st_0$ and $u_n \subset su_0$, thus each $t_n, u_n \subset U_1 \cap U_3$. We see that if $t_n, u_n \subset U_{1234}$, then $\overline{qt_n} \in U_1 \cap U_2$ and $\overline{qu_n} \in U_3 \cap U_4$, thus $t_n, u_n \in U_{1234}$.

Since $t_0, u_0 \in U_{1234}$, $\{u_i\}$ and $\{t_i\}$ are sequences of points in $\mathcal{U}_{1234}$. These series both appear to converge to the point s. This is a contradiction, becauce $s \in U_0 \cap U_1 \cap U_3$, an open set, and thus there must exist a neighborhood $N_\epsilon(s)$ such that $N_\epsilon(s) \subset U_0 \cap U_1 \cap U_3$. However, the codeword 01234 $\notin C$, thus if $t_i, u_i \in U_{1234}$, $t_i, u_i \notin U_0$. So we can't find an open neighborhood around s which is contained within U_0, contradicting the openness of U_0.

We now need to prove that the sequences of points $\{t_i\}$ and $\{u_i\}$ do in fact converge to s. To this end, we introduce coordinates to our figure. We consider a slightly simpler case: we draw the following figure: an isosceles triangle with height y_0 and width $2b$, as pictured in Figure 1. Its vertices are the points $(0, y_0)$, $(b, 0)$, and $(-b, 0)$. We select two points at the same height, (l_0, h_0) and $(-l_0, h_0)$. We draw lines between these points and the origin and between these points and the lower vertices of the triangle, as pictured in Figure 2. We let $-l_1, h_1)$ be the intersection of the line from $(-l_0, h_0)$ to $(b, 0)$ with the line from $(0, 0)$ to (l_0, h_0). (Likewise, (l_1, h_1) is the intersection of the line from (l_0, h_0) to $(-b, 0)$ with the line from $(0, 0)$ to $(-l_0, h_0)$.) The point $(0, y_1)$ is the intersection of the line from (l_0, h_0) to $(-b, 0)$ and the line from $(-l_0, h_0)$ to $(b, 0)$.

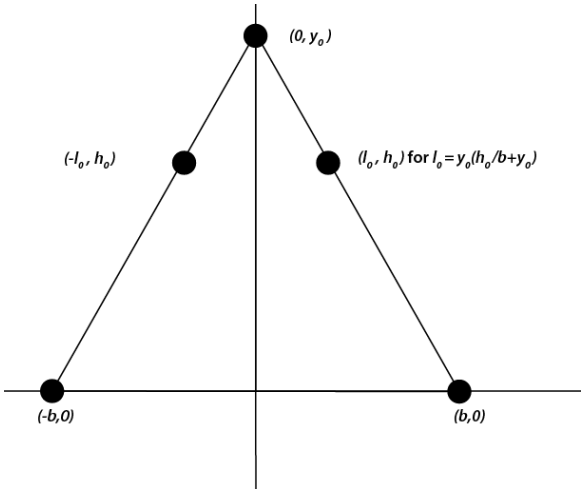

Figure 5: We add coordinites to prove the convergence.

We note that the triangle with vertices $(0, y_1)$, $(b, 0)$, and $(-b, 0)$ and "special points" (l_1, h_1) and $(-l_1, h_1)$ has the same "features" as the triangle we began with. Thus, we are in a position to define this as a recurrence, generating y_{n+1} from y_n and h_{n+1} from h_n the same way we found y_1 and h_1 from y_0 and h_0. (We can express l_n as a function of b, h_n, and y_n, and thus do not need to recursively define it.) In this way, we generate two sequences of real numbers: $\{y_0, y_1, y_2, \ldots\}$ and $\{h_0, h_1, h_2, \ldots\}$.

We would like to know whether or not these sequences converge and, if so, what they converge to. It seems clear that they should converge, as they are both clearly bounded below by zero and monotonically decreasing. However, we'd like to establish this formally and to find the values they approach. To this end, we establish a formulas for y_{n+1} and h_{n+1} in terms of y_n, h_n, l_n, and b.

We know that $(-l_{n+1}, h_{n+1})$ is an intersection between the line from $(-b, 0)$ to (l_n, h_n) and the line from $(0, 0)$ to $(-l_n, h_n)$. In point-slope form, equations for these lines are:

$$y = \frac{h_n}{l_n + b}(x + b) \tag{1}$$

$$y = \frac{-h_n}{l_n}(x). \tag{2}$$

Thus, $(-l_{n+1}, h_{n+1}$ is a solution to

$$\begin{aligned}
\frac{h_n}{l_n + b}(-l_{n+1} + b) &= \frac{-h_n}{l_n}(-l_{n+1}) \\
\frac{bh_n}{l_n + b} &= \left(\frac{-h_n}{l_n} - \frac{h_n}{l_n + b}\right)(-l_{n+1}) \\
\frac{bh_n}{l_n + b} &= -\frac{h_n(2l_n + b)}{l_n(l_n + b)}(-l_{n+1}) \\
b &= -\frac{(2l_n + b)}{l_n}(-l_{n+1}) \\
l_{n+1} &= \frac{bl_n}{2l_n + b}
\end{aligned}$$

Thus

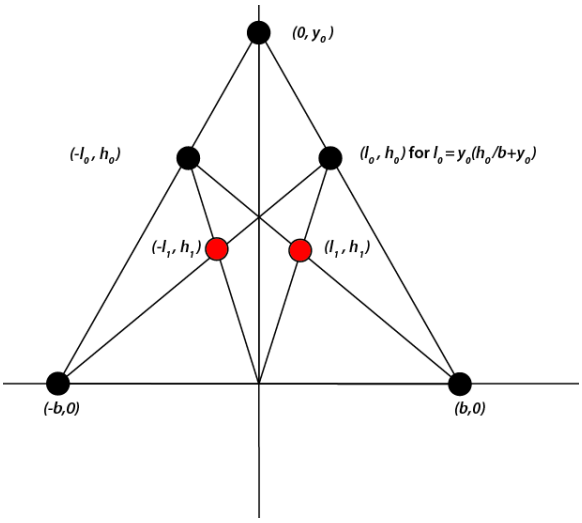

Figure 6: We construct (l_1, h_1) from (l_0, h_0).

$$l_{n+1} = \frac{bl_n}{2l_n + b} \tag{3}$$

$$h_{n+1} = \frac{-h_n}{l_n}(-l_{n+1}) = \frac{h_n}{l_n}\frac{bl_n}{2l_n + b} = \frac{bh_n}{2l_n + b} \leq h_n \tag{4}$$

Since (l_n, h_n) is on the line from $(0, 0)$ to (l_0, h_0), then $l_n = \frac{h_n}{h_0} l_n$, so if $h_n > 0$, then $l_n > 0$. Thus, if $h_n > 0$, the inequality

$$h_{n+1} = \frac{bh_n}{2l_n + b} < h_n$$

is strict. We have thus shown that $\{h_n\}$ is monotonically decreasing, thus by the monotone convergence theorem, it converges to its infimum. To show that it converges to zero, then, we must show that its infimum is zero. Assume to the contrary that the infimum of $\{h_n\}$ is $\alpha > 0$. Let $d_n = \frac{1}{h_n}$. Since $\{h_n\}$ converges to α, $\{d_n\}$ converges to $\frac{1}{\alpha}$. We see that

$$d_n = \frac{1}{h_n}$$

$$d_{n+1} = \frac{1}{h_{n+1}} = \frac{2l_n + b}{bh_n} = \frac{2l_0 h_n/h_0 + b}{hh_n} = \frac{2l_0}{h_0} + \frac{1}{h_n} = \frac{2l_0}{h_0} + d_n.$$

Thus $\{d_n\}$ diverges, so $\{h_n\}$ cannot converge to anything other than zero. Since we already know that $\{h_n\}$ converges, $\{h_n\}$ must converge to zero. Thus the points (l_n, h_n) converge to the origin. We recall that proof of this convergence was all we needed to show that $\mathcal{C} = \{1234, 012, 034, 013, 12, 34, 13, 01, 03, 3, 1\}$ has no local obstructions, but is not convex.

6 Restricted Codes

We gain a deeper understanding of a neural code by exploring the structure of its restricted codes, neural codes which are derived by "deleting" some neurons. We present some strategies for proving the Conjecture 2 using these ideas.

Restricted Code Let $\mathcal{C}$ be a neural code on n neurons. We say that $\mathcal{C}'$ is a *restricted code* of $\mathcal{C}$ if $\mathcal{C}'$ can be generated from $\mathcal{C}$ by ignoring input from one or more neurons.

For example, the code $\{123, 23, 13, 1\}$ is a subcode of $\{123, 234, 13, 1, 4\}$. Our course, we are not just interested in one subcode. For instance, the code $\{123, 234, 13, 1, 4\}$, or any four neuron subcode, has four neurons that can be generated by deleting one neuron. This motivates the following definition:

Partially ordered set of restricted codes For a neural code $\mathcal{C}$, we take $\mathcal{L}(\mathcal{C})$ to be the set of subcodes of $\mathcal{C}$, together with the relation $C < C'$ if C is a proper restricted code of C'.

Lemma If $\mathcal{D}$ is a convex neural code (or a code represented by a good cover), and $\mathcal{C}$ is any restricted code of $\mathcal{D}$, then $\mathcal{C}$ is convex (or is represented by a good cover).

Proof. Let $D_1, \ldots, D_n$ be the receptive fields generating $\mathcal{D}$. We see that each $\mathcal{D}_i$ is convex. Since $\mathcal{C}$ is a subcode of $\mathcal{D}$, the family of receptive fields generating $\mathcal{C}$ are a subfamily of $D_1, \ldots, D_n$. Thus the receptive fields generating $\mathcal{C}$ are convex, so $\mathcal{C}$ is a convex code. (Alternately, if $D_1, \ldots, D_n$ form a good cover, the set of receptive fields representing $\mathcal{C}$ are a subset of $D_1, \ldots, D_n$, and thus also form a good cover.) □

Theorem 5. *Let $\mathcal{L}(\mathcal{C})$ be the partially ordered set of restricted codes of $\mathcal{C}$. Every chain of $\mathcal{L}(\mathcal{C})$ contains a maximal convex restricted code. Likewise, every chain of $\mathcal{L}(\mathcal{C})$ contains a maximal subcode with no local obstructions and a maximal subcode which comes from a good cover.*

Proof. Let C be a chain of $\mathcal{L}(\mathcal{C})$. We know that the two smallest codes on this chain are convex, since the first is a code on one neuron and the second is a code on two neurons. The chain is a totally ordered finite set, so we can index its elements with integers. (Specifically, we index each subcode by the number of neurons it is on.) By the preceding lemma if $m < n$ and the subcode on this chain indexed by n is convex, then the subcode indexed by m is convex. All codes are either convex or not convex. Thus, since the code at the bottom of the chain is convex maximal intermediate code $\mathcal{C}_M$ such that $\mathcal{C}_M$ is convex, but $\mathcal{C}_N$ is not convex if $M \subset N$.

If $\mathcal{C}_M$ is the maximal convex subcode, then the code $\mathcal{C}_N$ which covers $\mathcal{C}_M$ is the minimal non-convex subcode. We see this because if there were a smaller non-convex subcode on this chain, it would have to be $\mathcal{C}_M$, which is convex by assumption. □

By the same argument, each chain of $\mathcal{L}(\mathcal{C})$ has a maximal code which can be realized as a good cover and a maximal code with no local obstructions. It is not obvious whether or not these are all always the same code, though our counterexample shows that the maximal convex code may sometimes be smaller than the maximal code which has no local obstructions and the maximal code which can be represented by a good cover.

These ideas provide a framework for an inductive proof of Conjecture 2. The base case, that all neural codes with one neuron are representable by good covers if and only if they have no local obstructions, is trivially true. Thus, Conjecture 2 reduces to the following:

Conjecture If $\mathcal{C}$ has no local obstructions and there exists a good cover representing every restricted code of $\mathcal{C}$, then there exists a good cover representing $\mathcal{C}$.

Another way of viewing this is as the statement that if we add a neuron to a neural code which is represented by a good cover, and our new code is no longer representable by a good cover, then a local obstruction should appear.

If we know that a large neural code is not convex, we would like to be able to identify which neurons are at fault. In other words, we would like to be able to locate the largest restricted codes which are convex. This would be useful information if we have data from many suspected place cells, and want to identify which ones may really be place cells, and which ones likely are not.

7 Discussion

The conjecture that any neural code with no local obstructions is false, but we would still like a way of determining which neural codes are convex. Giusti et al. have proven that any neural code closed under intersection is convex. Some conjecture that any neural code closed under intersection of maximal faces is convex. However, not every convex neural code is closed under intersection or maximal intersection. Thus, checking for local obstructions is to weak a test to exclude all non-convex codes, but testing for closure under intersections of maximal codewords is to strong a test for all convex codes to pass it. We would like to find an easily computable property that indicates whether or not a code is convex. We have not yet disproved the conjecture that all neural codes with no local obstructions can be generated by good covers, and hope to prove this conjecture in future work.

Acknowledgements

This work would not have been at all possible withour my mentor, Dr. Anne Shiu. Much of this work was undertaken jointly with Zev Woodstock, who found the counterexample discussed in section 5 and worked heavily on the proof. I had many helpful discussions with Dr. Jacob White (who led me to Nauty), Lauren Grimley, and Bryan Félix.

This project was conducted as part of the NSF-funded REU in Mathematics at Texas A&M University (DMS-1460766), Summer 2015.

References

[1] Burgess, Neil *The 2014 Nobel Prize in Physiology or Medicine: A Spatial Model for Cognitive Neuroscience*. 2014, http://www.ncbi.nlm.nih.gov/pmc/articles/PMC4276740/

[2] Tancer, Martin *A counterexample to Wegners conjecture on good covers*. 2010, <http://arxiv.org/abs/1008.1895>

[3] C. Curto, V. Itskov, A. Veliz-Cuba, N. Youngs *The neural ring: an algebraic tool for analyzing the intrinsic structure of neural codes*. Bull Math Bio. 2013 http://arxiv.org/pdf/1212.4201v2.pdf

[4] Carina Curto, Elizabeth Gross, Jack Jeffries, Katie Morrison, Mohamed Omar, Zvi Rosen, Anne Shiu, and Nora Youngs *What makes a neural code convex?* (in preparation)

[5] Chad Guisti, Vladimir Itskov, William Kronholm *On convex codes and intersection violators* (preprint)

[6] A. Björner, *Nerves, fibers and homotopy groups*, J. Combinatorial Theory, Ser.102 (2003), 88-93.

[7] B. D. McKay and A. Piperno, *Practical Graph Isomorphism, II*, J. Symbolic Computation (2013) 60 94-112. http://dx.doi.org/10.1016/j.jsc.2013.09.003 Preprint version at arxiv.org

[8] W. A. Stein et al. Sage Mathematics Software (Version 6.7), The Sage Development Team, 2015, http://www.sagemath.org

[9] OEIS Foundation Inc. (2011), The On-Line Encyclopedia of Integer Sequences, http://oeis.org/A006602

Appendix 1: Source Code

Below are the functions we used to classify neural codes on five neurons.

```
#given a set of sets, returns the intersection of all sets in that set.
def intersect_all_c(setList):
    first = setList[0]
    length = (setList).cardinality()
    for i in range (0, length):
        first = first.intersection(setList[i])
    return first

#given a set of sets, generates all intersections of collections of those sets, and returns
those which are not already in the code
#if fed a set of maximal faces, will generate all intersectios of maximal faces
def intersections_c(setList):
    result= []
    subsets = Subsets(setList)
    for i in range(1, (subsets).cardinality()):
        result.append(intersect_all_c(subsets[i]))
    return Set(result)-setList

#given a homology, returns true if the homology group is trivial in any dimension
def is_trivial_c(homology):
    length = len(homology.viewvalues())
    trivial_complex = SimplicialComplex([range(0, length)])
    return homology == trivial_complex.homology()
```

```
#the function link_checker takes a neural code as an input and outputs the
links of all intersections of elements in the code that are not in the code
def link_checker_c(code):
    simplicial_complex = SimplicialComplex(code)
    maximal = simplicial_complex.facets()
    links_to_check = intersections_c(code)
    bad_links = []
    homologies = []
    for i in range(0,(links_to_check).cardinality()):
        current_link = simplicial_complex.link(Simplex(links_to_check[i]))
        if (is_trivial_c(current_link.homology())==false):
            bad_links.append(current_link)
            homologies.append(current_link.homology())
    return (links_to_check, bad_links, homologies)

#this function takes a neural code as input and outputs true if has no local obstructions and
false otherwise
def bool_link_checker(code):
    simplicial_complex = SimplicialComplex(code)
    maximal = simplicial_complex.facets()
    if maximal.cardinality()==1:
        return true
    else:
        links_to_check = intersections_c(code)
        for i in range(0,(links_to_check).cardinality()):
            current_link = simplicial_complex.link(Simplex(links_to_check[i]))
            if (is_trivial_c(current_link.homology())==false):
                return false
        return true




#Given a neural code, returns detailed information about it. Will not give the
right result if the code does not contain the empty set.
#to do: say whether a neural code has no local obstructions because it is closed
under intersection of maximal codewords, or because the links of intersections
of maximal codewords are contractible
def detailed_link_checker(code):
    simplicial_complex = SimplicialComplex(code)
    maximal = simplicial_complex.facets()
    n_max = maximal.cardinality()
    if n_max == 1:
        print "The neural code " + str(code) + " is convex because it only has one maximal face."
        return true
    else:
        links_to_check = intersections_c(code)
        missing_codewords = []
        for i in range(0,(links_to_check).cardinality()):
            current_link = simplicial_complex.link(Simplex(links_to_check[i]))
```

```
            if (is_trivial_c(current_link.homology())==false):
                missing_codewords.append(links_to_check[i])
        if (missing_codewords == []):
            print "The neural code " + str(code) +
             " has no local obstructions, and has maximal faces " + str(maximal) + " ."
            return true
        else:
            print "The neural code " + str(code) + " has a local obstruction.
            This could be remedied if the following codewords were added: " + str(missing_codewords)

#input: a neural code
#output: a list of faces required for that neural code to not have local obstructions
def another_link_checker(code):
    simplicial_complex = SimplicialComplex(code)
    maximal = simplicial_complex.facets()
    n_max = maximal.cardinality()
    links_to_check = intersections_c(code)
    missing_codewords = []
    for i in range(0,(links_to_check).cardinality()):
        current_link = simplicial_complex.link(Simplex(links_to_check[i]))
        if (is_trivial_c(current_link.homology())==false):
            missing_codewords.append(links_to_check[i])
    return missing_codewords

def required_codewords(codes):
    result = []
    for code in codes:
        result.append(another_link_checker(code))
    return result

def optional_max_inter(code):
    simplicial_complex = SimplicialComplex(code)
    maximal = simplicial_complex.facets()
    links_to_check = intersections_c(code)
    missing_codewords = []
    for i in range(0,(links_to_check).cardinality()):
            current_link = simplicial_complex.link(Simplex(links_to_check[i]))
            if (is_trivial_c(current_link.homology())==false):
                missing_codewords.append(links_to_check[i])
    #print (Set(missing_codewords), links_to_check)
    num_optional_faces = len([f for f in simplicial_complex.face_iterator()])-
    len(simplicial_complex.facets())-len(missing_codewords)
    return (missing_codewords, links_to_check.difference(Set(missing_codewords)),
    num_optional_faces)


def tuples_to_sets(the_list):
    outputlist = []
    for big_tuple in the_list:
        outerset = Set([])
        for small_tuple in big_tuple:
```

```
            innerset = Set(small_tuple)
            outerset = outerset.union(Set([innerset]))
            outerset = outerset.union(Set([Set([])]))
        outputlist.append(outerset)
    return outputlist


def make_table(complex_list):
    complex_set = tuples_to_sets(complex_list)
    missing_codeword_list = []
    optional_codeword_list = []
    num_optional_list = []
    for current_complex in complex_set:
        current_tuple = optional_max_inter(current_complex)
        missing_codeword_list.append(current_tuple[0])
        optional_codeword_list.append(current_tuple[1])
        num_optional_list.append(current_tuple[2])
    return latex(table(columns=[complex_set, missing_codeword_list, optional_codeword_list,
     num_optional_list]))


#input: a list of neural codes as tuples of tuples
#output: a list of intersections of maximal faces whose links are convex
def find_optional_max_inter(code_list):
    result = []
    for code in code_list:
        result.append(optional_max_inter(code))
    return result
```

Here are the functions we used to generate the set of simplicial complexes on five vertices. Nauty generates lists of hypergraphs, rather than of simplicial complexes, so we needed to sort through the output to find those hypergraphs which contained only maximal faces.

```
#input: two ordered tuples of integers, the smaller one listed first
#output: boolean, whether the smaller is contained in the larger
def containment_check(smaller, larger):
    counter_s = 0
    counter_l = 0
    while (counter_l < len(larger)):
        #print "counters: S-"+str(counter_s)+ "   L-" +str(counter_l)
        if (counter_s >= len(smaller)):
            return true
        elif (smaller[counter_s] == larger[counter_l]):
            counter_s = counter_s + 1
            counter_l = counter_l + 1
            if counter_s==len(smaller):
                return true
        elif (smaller[counter_s] < larger[counter_l]):
            #print "first false"
            return false
        elif (smaller[counter_s] > larger[counter_l]):
```

```
            counter_l = counter_l + 1
    #print "second"
    return false

#input: tuple of tuples of integers. tuple of tuple is ordered by length;
 tuples of integers are in ascending order
#output: whether there are any containments in that tuple
def max_only_check(code):
    for i in range(0,len(code)):
        for j in range((i+1),len(code)):
            if (len(code[j])>len(code[i])):
                if (containment_check(code[i], code[j]) == true):
                    return false
    else:
        return true


def check_list_for_maximality(test):
    good_complexes = []
    bad_complexes = []
    for code in test:
        if max_only_check(code):
            good_complexes.append(code)
        else:
            bad_complexes.append(code)
    return (good_complexes, "bad: ", bad_complexes)
```

Classification of Neural Codes on 5 Neurons

What follows is a list of connected simplicial complexes on 5 vertices, defined in terms of their maximal faces, together with the non-maximal faces which must be contained in each code on them which has no local obstructions.

Simplicial Complex	Required Codewords	Excludable Intersections of Maximal faces	Number of optional codewords	Number of codes with no local obstructions
{{0, 1, 2, 3, 4}, {}}	[]	{}	30	1073741824
{{}, {0, 2, 3, 4}, {0, 1}}	[{0}]	{}	14	16384
{{0, 1, 2}, {}, {0, 3, 4}}	[{0}]	{}	10	1024
{{0, 1, 3, 4}, {0, 1, 2}, {}}	[{0, 1}]	{}	16	65536
{{0, 1, 2, 3}, {}, {0, 1, 2, 4}}	[{0, 1, 2}]	{}	20	1048576
{{}, {0, 3, 4}, {0, 2}, {0, 1}}	[{0}]	{}	7	128
{{}, {1, 3, 4}, {0, 2}, {0, 1}}	[{0}, {1}]	{}	6	64
{{1, 2, 3, 4}, {}, {0, 2}, {0, 1}}	[{2}, {0}, {1}]	{}	12	4096
{{2, 3}, {}, {0, 2, 4}, {0, 1}}	[{2}, {0}]	{}	6	64
{{0, 2, 3}, {}, {0, 2, 4}, {0, 1}}	[{0, 2}, {0}]	{}	8	256
{{0, 2, 3}, {0, 1}, {}, {1, 2, 4}}	[{2}, {0}, {1}]	{}	8	256
{{0, 2, 3}, {}, {2, 3, 4}, {0, 1}}	[{2, 3}, {0}]	{}	8	256
{{1, 2, 3, 4}, {0, 2, 3}, {}, {0, 1}}	[{2, 3}, {0}, {1}]	{}	14	16384
{{1, 2, 3, 4}, {}, {0, 2, 3, 4}, {0, 1}}	[{2, 3, 4}, {0}, {1}]	{}	18	262144
{{0, 1, 4}, {0, 1, 2}, {}, {0, 1, 3}}	[{0, 1}]	{}	11	2048
{{0, 1, 2}, {}, {0, 2, 3, 4}, {0, 1, 3}}	[{0, 2}, {0, 3}, {0}, {0, 1}]	{}	14	16384
{{0, 1, 2}, {}, {0, 1, 3}, {0, 3, 4}}	[{0, 3}, {0, 1}]	{{0}}	10	1024
{{}, {0, 1, 2}, {1, 2, 3}, {0, 3, 4}}	[{1, 2}, {3}, {0}]	{}	10	1024
{{1, 2, 3, 4}, {0, 1, 2}, {}, {0, 3, 4}}	[{3, 4}, {0}, {1, 2}]	{}	16	65536
{{0, 1, 3, 4}, {0, 1, 2}, {}, {0, 2, 3, 4}}	[{0, 3, 4}, {0, 2}, {0}, {0, 1}]	{}	18	262144
{{0, 1, 2, 3}, {0, 1, 3, 4}, {}, {0, 1, 2, 4}}	[{0, 1, 4}, {0, 1, 2}, {0, 1, 3}, {0, 1}]	{}	20	1048576
{{1, 2}, {}, {0, 3, 4}, {0, 2}, {0, 1}}	[{2}, {0}, {1}]	{}	5	32
{{}, {0, 4}, {0, 2}, {0, 3}, {0, 1}}	[{0}]	{}	4	16
{{}, {1, 4}, {0, 2}, {0, 3}, {0, 1}}	[{0}, {1}]	{}	3	8
{{}, {1, 2, 4}, {0, 2}, {0, 3}, {0, 1}}	[{2}, {0}, {1}]	{}	5	32
{{1, 2, 3, 4}, {}, {0, 2}, {0, 3}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	11	2048
{{2, 4}, {1, 3}, {}, {0, 2}, {0, 1}}	[{2}, {0}, {1}]	{}	2	4
{{1, 3}, {}, {1, 2, 4}, {0, 2}, {0, 1}}	[{2}, {0}, {1}]	{}	5	32
{{1, 3}, {}, {0, 2}, {2, 3, 4}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	4	16
{{3, 4}, {}, {1, 2, 3}, {0, 2}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	4	16
{{}, {1, 2, 4}, {1, 2, 3}, {0, 2}, {0, 1}}	[{1, 2}, {2}, {0}, {1}]	{}	6	64
{{1, 2, 3}, {}, {0, 3, 4}, {0, 2}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	7	128

{{}, {1, 3, 4}, {1, 2, 3}, {0, 2}, {0, 1}}	[{1, 3}, {2}, {0}, {1}]	{}	6	64
{{1, 2, 3, 4}, {}, {0, 3, 4}, {0, 2}, {0, 1}}	[{3, 4}, {2}, {0}, {1}]	{}	13	8192
{{}, {1, 3, 4}, {0, 3, 4}, {0, 2}, {0, 1}}	[{3, 4}, {0}, {1}]	{}	7	128
{{}, {1, 3, 4}, {0, 2}, {2, 3, 4}, {0, 1}}	[{3, 4}, {2}, {0}, {1}]	{}	6	64
{{1, 2, 4}, {2, 3}, {}, {0, 2, 4}, {0, 1}}	[{2, 4}, {2}, {0}, {1}]	{}	6	64
{{1, 3, 4}, {2, 3}, {}, {0, 2, 4}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	6	64
{{0, 2, 3}, {1, 2, 3}, {}, {0, 2, 4}, {0, 1}}	[{0, 2}, {2, 3}, {0}, {1}]	{{2}}	8	256
{{0, 2, 3}, {1, 2, 3}, {}, {2, 3, 4}, {0, 1}}	[{2, 3}, {0}, {1}]	{}	9	512
{{0, 2, 3}, {0, 3, 4}, {}, {0, 2, 4}, {0, 1}}	[{0, 4}, {0, 2}, {0, 3}, {0}]	{}	7	128
{{0, 2, 3}, {1, 3, 4}, {}, {0, 2, 4}, {0, 1}}	[{4}, {0, 2}, {3}, {0}, {1}]	{}	8	256
{{0, 2, 3}, {}, {2, 3, 4}, {0, 2, 4}, {0, 1}}	[{2, 4}, {0, 2}, {2, 3}, {2}, {0}]	{}	6	64
{{1, 2, 3, 4}, {0, 2, 3}, {}, {0, 2, 4}, {0, 1}}	[{2, 4}, {2, 3}, {0, 2}, {0}, {1}, {2}]	{}	12	4096
{{0, 2, 3}, {1, 2, 4}, {}, {2, 3, 4}, {0, 1}}	[{2, 4}, {2, 3}, {0}, {1}]	{{2}}	8	256
{{1, 2, 3, 4}, {0, 2, 3}, {0, 1, 2}, {}, {0, 1, 3}}	[{3}, {0}, {1, 2}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	8	256
{{0, 2, 3}, {0, 1, 2}, {}, {0, 1, 3}, {0, 1, 4}}	[{0, 2}, {0, 3}, {0}, {0, 1}]	{}	9	512
{{0, 1, 4}, {0, 1, 2}, {}, {0, 2, 3, 4}, {0, 1, 3}}	[{0, 4}, {0, 2}, {0, 3}, {0}, {0, 1}]	{}	14	16384
{{1, 2, 3, 4}, {0, 1, 2}, {}, {0, 2, 3, 4}, {0, 1, 3}}	[{3}, {0}, {1, 2}, {2, 3, 4}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	12	4096
{{0, 1, 2}, {0, 3, 4}, {}, {0, 2, 4}, {0, 1, 3}}	[{0, 4}, {0, 2}, {0, 3}, {0}, {0, 1}]	{}	8	256
{{1, 2, 3, 4}, {0, 1, 2}, {0, 3, 4}, {}, {0, 1, 3}}	[{3}, {1, 2}, {3, 4}, {1}, {1, 3}, {0, 3}, {0, 1}]	{{0}}	12	4096
{{0, 1, 2}, {1, 2, 3}, {0, 3, 4}, {}, {0, 1, 3}}	[{1}, {1, 3}, {1, 2}, {0, 3}, {0, 1}]	{{3}, {0}}	8	256
{{0, 1, 4}, {1, 2, 3}, {0, 3, 4}, {}, {0, 1, 2}}	[{3}, {1, 2}, {0, 4}, {0, 1}]	{{0}, {1}}	10	1024
{{0, 1, 2}, {1, 2, 3}, {0, 3, 4}, {}, {1, 2, 4}}	[{1, 2}, {4}, {3}, {0}]	{}	11	2048
{{1, 2, 3, 4}, {0, 1, 3, 4}, {0, 1, 2}, {}, {0, 2, 3, 4}}	[{0, 3, 4}, {0}, {1, 2}, {2, 3, 4}, {3, 4}, {1, 3, 4}, {0, 2}, {1}, {2}, {0, 1}]	{}	14	16384
{{}, {0, 1, 3, 4}, {0, 1, 2, 3}, {0, 2, 3, 4}, {0, 1, 2, 4}}	[{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {0}, {0, 4}, {0, 2, 4}, {0, 2}, {0, 1, 4}, {0, 1, 2}, {0, 3}, {0, 1}]	{}	14	16384
{{1, 3, 4}, {0, 3, 4}, {0, 2}, {1, 2}, {}, {0, 1}}	[{3, 4}, {2}, {0}, {1}]	{}	6	64
{{0, 2}, {1, 2}, {1, 4}, {}, {0, 3}, {0, 1}}	[{2}, {0}, {1}]	{}	2	4
{{1, 3, 4}, {0, 2}, {1, 2}, {}, {0, 3}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	4	16
{{0, 2}, {1, 2}, {0, 4}, {}, {0, 3}, {0, 1}}	[{2}, {0}, {1}]	{}	2	4
{{0, 2}, {}, {0, 4}, {1, 2, 3}, {0, 3}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	4	16
{{0, 2}, {1, 2, 3, 4}, {0, 4}, {}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	10	1024
{{0, 2}, {}, {0, 1}, {1, 2, 3}, {0, 3}, {1, 2, 4}}	[{1, 2}, {2}, {3}, {0}, {1}]	{}	5	32
{{1, 3, 4}, {0, 2}, {0, 1}, {}, {0, 3}, {1, 2, 4}}	[{1, 4}, {2}, {3}, {0}, {1}]	{}	5	32
{{2, 3}, {0, 2}, {1, 3}, {0, 4}, {}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	1	2
{{2, 3}, {0, 2}, {1, 3}, {0, 1}, {}, {1, 2, 4}}	[{2}, {3}, {0}, {1}]	{}	4	16

{{2,4},{0,2},{1,3},{0,4},{},{0,1}}	[{4},{2},{0},{1}]	{}	1	2
{{0,2},{1,3},{0,4},{0,1},{},{1,2,4}}	[{4},{2},{0},{1}]	{}	4	16
{{0,2},{1,3},{0,4},{},{2,3,4},{0,1}}	[{4},{2},{3},{0},{1}]	{}	3	8
{{0,3,4},{0,2},{1,3},{0,1},{},{1,2,4}}	[{4},{2},{3},{0},{1}]	{}	6	64
{{0,2},{1,3},{0,1},{},{2,3,4},{1,2,4}}	[{2,4},{2},{3},{0},{1}]	{}	5	32
{{2,4},{3,4},{0,2},{1,3},{},{0,1}}	[{4},{2},{3},{0},{1}]	{}	0	1
{{3,4},{0,2},{1,3},{0,1},{},{1,2,4}}	[{4},{2},{3},{0},{1}]	{}	3	8
{{3,4},{0,2},{1,2,3},{0,1},{},{1,2,4}}	[{4},{3},{0},{1},{1,2},{2}]	{}	4	16
{{0,3,4},{0,2},{1,2,3},{0,1},{},{1,2,4}}	[{4},{3},{0},{1},{1,2},{2}]	{}	7	128
{{1,3,4},{0,2},{1,2,3},{0,1},{},{1,2,4}}	[{0},{1},{1,3},{1,2},{1,4},{2}]	{}	5	32
{{1,3,4},{0,3,4},{0,2},{1,2,3},{},{0,1}}	[{3,4},{0},{1},{1,3},{2}]	{{3}}	7	128
{{1,3,4},{0,2},{1,2,3},{},{2,3,4},{0,1}}	[{3,4},{2,3},{3},{0},{1},{1,3},{2}]	{}	4	16
{{1,3,4},{0,3,4},{0,2},{},{2,3,4},{0,1}}	[{3,4},{2},{0},{1}]	{}	8	256
{{0,3,4},{2,3},{0,1},{},{0,2,4},{1,2,4}}	[{2,4},{3},{0},{1},{0,4},{2}]	{{4}}	6	64
{{0,2,3},{},{0,1},{1,2,3},{0,2,4},{1,2,4}}	[{2,4},{2,3},{0,2},{0},{1},{1,2},{2}]	{}	6	64
{{0,2,3},{0,3,4},{},{1,2,3},{0,2,4},{0,1}}	[{0},{0,4},{2,3},{0,2},{1},{0,3}]	{{2},{3}}	7	128
{{0,2,3},{1,3,4},{},{1,2,3},{0,2,4},{0,1}}	[{0},{4},{2,3},{0,2},{1},{1,3}]	{{2},{3}}	8	256
{{0,2,3},{0,3,4},{},{2,3,4},{0,2,4},{0,1}}	[{2,4},{3},{0},{0,4},{3,4},{4},{2,3},{0,2},{2},{0,3}]	{}	1	2
{{0,2,3},{0,3,4},{1,2,3,4},{},{0,2,4},{0,1}}	[{2,4},{3},{0},{0,4},{3,4},{4},{2,3},{0,2},{1},{2},{0,3}]	{}	7	128
{{0,2,3},{1,3,4},{},{2,3,4},{0,2,4},{0,1}}	[{2,4},{0},{3,4},{2,3},{0,2},{1},{2}]	{{4},{3}}	6	64
{{0,2,3},{},{1,2,3},{2,3,4},{0,2,4},{0,1}}	[{2,4},{2,3},{0,2},{0},{1},{2}]	{}	7	128
{{0,2,3},{0,1,3},{0,1,4},{},{0,1,2},{1,2,3}}	[{3},{0},{1,2},{2,3},{0,2},{1},{1,3},{2},{0,3},{0,1}]	{}	3	8
{{0,2,3},{0,1,3},{0,1,4},{0,1,2},{},{2,3,4}}	[{0},{4},{2,3},{0,2},{0,3},{0,1}]	{{2},{3}}	9	512
{{0,2,3},{0,1,3},{1,2,3,4},{0,1,4},{0,1,2},{}}	[{3},{0},{1,2},{1,4},{2,3},{0,2},{1},{1,3},{2},{0,3},{0,1}]	{}	8	256
{{0,1,3},{1,2,3,4},{0,1,4},{0,1,2},{},{0,2,3,4}}	[{3},{0},{1,2},{0,4},{1,4},{2,3,4},{4},{0,2},{1},{1,3},{2},{0,3},{0,1}]	{}	9	512
{{0,2,3},{0,3,4},{0,1,3},{0,1,2},{},{0,2,4}}	[{0,4},{0,2},{0,3},{0},{0,1}]	{}	9	512
{{0,3,4},{0,1,3},{},{0,1,2},{1,2,3},{0,2,4}}	[{0},{1,2},{0,4},{0,2},{1},{1,3},{0,3},{0,1}]	{{2},{3}}	6	64
{{0,3,4},{0,1,3},{1,2,3,4},{0,1,2},{},{0,2,4}}	[{2,4},{3},{0},{1,2},{0,4},{3,4},{4},{0,2},{1},{1,3},{2},{0,3},{0,1}]	{}	6	64
{{0,3,4},{0,1,3},{0,1,4},{},{0,1,2},{1,2,3}}	[{0},{1,2},{0,4},{1},{1,3},{0,3},{0,1}]	{{3}}	7	128
{{0,3,4},{0,1,4},{},{0,1,2},{1,2,3},{2,3,4}}	[{1,2},{0,4},{3,4},{2,3},{0,1}]	{{4},{2},{3},{0},{1}}	10	1024

{{0, 1, 3, 4}, {1, 2, 3, 4}, {0, 1, 2, 3}, {}, {0, 2, 3, 4}, {0, 1, 2, 4}}	[{0, 2, 3}, {4}, {1, 2}, {0, 4}, {1, 2, 4}, {1, 3, 4}, {1, 3}, {0, 3}, {2, 4}, {0, 3, 4}, {3}, {0}, {1, 2, 3}, {1, 4}, {2, 3, 4}, {0, 2, 4}, {3, 4}, {2, 3}, {0, 2}, {0, 1, 3}, {1}, {0, 1, 4}, {0, 1, 2}, {2}, {0, 1}]	{}	0	1
{{1, 3, 4}, {0, 3, 4}, {0, 2}, {1, 2}, {}, {2, 3, 4}, {0, 1}}	[{3, 4}, {2}, {0}, {1}]	{}	8	256
{{0, 2}, {1, 3}, {1, 2}, {0, 1}, {}, {2, 3, 4}, {0, 3}}	[{2}, {3}, {0}, {1}]	{}	4	16
{{1, 3, 4}, {0, 2}, {1, 2}, {0, 1}, {}, {2, 3, 4}, {0, 3}}	[{3, 4}, {2}, {3}, {0}, {1}]	{}	5	32
{{0, 2}, {1, 3}, {1, 2}, {0, 4}, {}, {0, 3}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	1	2
{{3, 4}, {0, 2}, {1, 2}, {0, 4}, {}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{1, 3, 4}, {0, 2}, {1, 2}, {0, 4}, {}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	3	8
{{0, 2}, {}, {0, 4}, {0, 1}, {1, 2, 3}, {0, 3}, {1, 2, 4}}	[{4}, {3}, {0}, {1}, {1, 2}, {2}]	{}	4	16
{{1, 3, 4}, {0, 2}, {}, {0, 1}, {1, 2, 3}, {0, 3}, {1, 2, 4}}	[{3}, {0}, {1}, {1, 3}, {1, 2}, {1, 4}, {2}]	{}	4	16
{{1, 3, 4}, {0, 2}, {0, 1}, {}, {2, 3, 4}, {0, 3}, {1, 2, 4}}	[{2, 4}, {3}, {0}, {1, 4}, {3, 4}, {4}, {1}, {2}]	{}	3	8
{{0, 3, 4}, {0, 2}, {1, 3}, {2, 3}, {0, 1}, {}, {1, 2, 4}}	[{4}, {2}, {3}, {0}, {1}]	{}	6	64
{{2, 3}, {0, 2}, {1, 3}, {1, 2}, {0, 4}, {}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	1	2
{{3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 4}, {}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{2, 3}, {0, 2}, {1, 3}, {0, 4}, {0, 1}, {}, {1, 2, 4}}	[{4}, {2}, {3}, {0}, {1}]	{}	3	8
{{0, 2}, {1, 3}, {0, 4}, {0, 1}, {}, {2, 3, 4}, {1, 2, 4}}	[{2, 4}, {4}, {3}, {0}, {1}, {2}]	{}	4	16
{{0, 3, 4}, {0, 2}, {1, 3}, {0, 1}, {}, {2, 3, 4}, {1, 2, 4}}	[{2, 4}, {3, 4}, {3}, {0}, {1}, {2}]	{{4}}	6	64
{{2, 4}, {3, 4}, {0, 2}, {1, 3}, {}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{3, 4}, {0, 2}, {1, 3}, {0, 1}, {}, {0, 3}, {1, 2, 4}}	[{4}, {2}, {3}, {0}, {1}]	{}	3	8
{{1, 3, 4}, {0, 3, 4}, {0, 2}, {1, 2, 3}, {0, 1}, {}, {1, 2, 4}}	[{0}, {1, 2}, {1, 4}, {3, 4}, {1}, {1, 3}, {2}]	{{4}, {3}}	6	64
{{1, 3, 4}, {0, 3, 4}, {0, 2}, {}, {1, 2, 3}, {2, 3, 4}, {0, 1}}	[{3, 4}, {2, 3}, {3}, {0}, {1}, {1, 3}, {2}]	{}	6	64
{{1, 3, 4}, {0, 2}, {}, {0, 1}, {1, 2, 3}, {2, 3, 4}, {1, 2, 4}}	[{2, 4}, {3}, {0}, {1, 2}, {1, 4}, {3, 4}, {4}, {2, 3}, {1}, {1, 3}, {2}]	{}	0	1
{{1, 3, 4}, {0, 3, 4}, {2, 3}, {0, 1}, {}, {0, 2, 4}, {1, 2, 4}}	[{2, 4}, {3}, {0}, {0, 4}, {1, 4}, {3, 4}, {4}, {1}, {2}]	{}	4	16
{{0, 2, 3}, {}, {0, 1}, {1, 2, 3}, {2, 3, 4}, {0, 2, 4}, {1, 2, 4}}	[{2, 4}, {2, 3}, {0, 2}, {0}, {1}, {1, 2}, {2}]	{}	7	128
{{0, 2, 3}, {0, 3, 4}, {}, {0, 1}, {1, 2, 3}, {0, 2, 4}, {1, 2, 4}}	[{2, 4}, {0}, {1, 2}, {0, 4}, {2, 3}, {0, 2}, {1}, {2}, {0, 3}]	{{4}, {3}}	5	32
{{0, 2, 3}, {0, 3, 4}, {}, {1, 2, 3}, {2, 3, 4}, {0, 2, 4}, {0, 1}}	[{2, 4}, {3}, {0}, {0, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {2}, {0, 3}]	{}	2	4
{{0, 2, 3}, {1, 3, 4}, {}, {1, 2, 3}, {2, 3, 4}, {0, 2, 4}, {0, 1}}	[{2, 4}, {3}, {0}, {3, 4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}]	{{4}}	5	32
{{0, 2, 3}, {0, 1, 3}, {0, 1, 4}, {}, {0, 1, 2}, {1, 2, 3}, {2, 3, 4}}	[{3}, {0}, {1, 2}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	4	16

{{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {0, 1, 4}, {0, 1, 2}, {}, {0, 2, 4}}	[{0, 4}, {0, 2}, {0, 3}, {0}, {0, 1}]	{}	10	1024
{{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {1, 2, 3, 4}, {0, 1, 2}, {}, {0, 2, 4}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	5	32
{{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {}, {0, 1, 2}, {1, 2, 3}, {0, 2, 4}}	[{3}, {0}, {1, 2}, {0, 4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	3	8
{{0, 3, 4}, {0, 1, 3}, {}, {0, 1, 2}, {1, 2, 3}, {2, 3, 4}, {0, 2, 4}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	0	1
{{0, 3, 4}, {0, 1, 3}, {0, 1, 4}, {}, {0, 1, 2}, {1, 2, 3}, {0, 2, 4}}	[{0}, {1, 2}, {0, 4}, {0, 2}, {1}, {1, 3}, {0, 3}, {0, 1}]	{{2}, {3}}	7	128
{{0, 3, 4}, {0, 1, 3}, {0, 1, 4}, {}, {0, 1, 2}, {1, 2, 3}, {2, 3, 4}}	[{3}, {0}, {1, 2}, {0, 4}, {3, 4}, {2, 3}, {1}, {1, 3}, {0, 3}, {0, 1}]	{{4}, {2}}	5	32
{{1, 2}, {0, 4}, {}, {2, 3, 4}, {1, 3, 4}, {0, 2}, {0, 3}, {0, 1}}	[{3, 4}, {4}, {3}, {0}, {1}, {2}]	{}	4	16
{{0, 4}, {}, {1, 2, 4}, {1, 3, 4}, {0, 2}, {1, 2, 3}, {0, 3}, {0, 1}}	[{3}, {0}, {1, 2}, {1, 4}, {4}, {1}, {1, 3}, {2}]	{}	3	8
{{}, {2, 3, 4}, {1, 2, 4}, {1, 3, 4}, {0, 2}, {1, 2, 3}, {0, 3}, {0, 1}}	[{2, 4}, {3}, {0}, {1, 2}, {1, 4}, {3, 4}, {4}, {2, 3}, {1}, {1, 3}, {2}]	{}	0	1
{{1, 2}, {0, 4}, {}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{0, 4}, {}, {1, 2, 4}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	3	8
{{1, 2}, {}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 3}, {0, 1}}	[{2}, {3}, {0}, {1}]	{}	1	2
{{0, 4}, {}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{1, 4}, {}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{}, {1, 2, 4}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	3	8
{{0, 3, 4}, {}, {2, 3, 4}, {1, 2, 4}, {1, 3, 4}, {0, 2}, {1, 2, 3}, {0, 1}}	[{2, 4}, {3}, {0}, {1, 2}, {1, 4}, {3, 4}, {4}, {2, 3}, {1}, {1, 3}, {2}]	{}	2	4
{{0, 2, 3}, {0, 3, 4}, {}, {0, 2, 4}, {1, 2, 4}, {1, 3, 4}, {1, 2, 3}, {0, 1}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {1, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}]	{}	0	1
{{0, 2, 3}, {0, 3, 4}, {}, {2, 3, 4}, {0, 2, 4}, {1, 2, 4}, {1, 2, 3}, {0, 1}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {2}, {0, 3}]	{}	2	4
{{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {}, {0, 2, 4}, {0, 1, 4}, {0, 1, 2}, {1, 2, 3}}	[{3}, {0}, {1, 2}, {0, 4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	4	16
{{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {1, 2, 3, 4}, {}, {0, 2, 4}, {0, 1, 4}, {0, 1, 2}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {1, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	4	16
{{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {}, {2, 3, 4}, {0, 2, 4}, {0, 1, 2}, {1, 2, 3}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	0	1
{{0, 3, 4}, {0, 1, 3}, {}, {0, 2, 4}, {1, 2, 4}, {0, 1, 4}, {0, 1, 2}, {1, 2, 3}}	[{2, 4}, {0}, {1, 2}, {0, 4}, {1, 4}, {4}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{{3}}	3	8

{{0, 3, 4}, {0, 1, 3}, {}, {2, 3, 4}, {0, 2, 4}, {0, 1, 4}, {0, 1, 2}, {1, 2, 3}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	1	2
{{2, 4}, {1, 2}, {0, 4}, {}, {3, 4}, {0, 2}, {1, 3}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{0, 4}, {}, {2, 3, 4}, {1, 2, 4}, {1, 3, 4}, {0, 2}, {1, 2, 3}, {0, 3}, {0, 1}}	[{2, 4}, {3}, {0}, {1, 2}, {1, 4}, {3, 4}, {4}, {2, 3}, {1}, {1, 3}, {2}]	{}	0	1
{{1, 2}, {0, 4}, {}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{0, 4}, {}, {1, 2, 4}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	3	8
{{0, 2, 3}, {0, 3, 4}, {}, {2, 3, 4}, {0, 2, 4}, {1, 2, 4}, {1, 3, 4}, {1, 2, 3}, {0, 1}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {1, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}]	{}	0	1
{{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {}, {0, 2, 4}, {1, 2, 4}, {0, 1, 4}, {0, 1, 2}, {1, 2, 3}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {1, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	1	2
{{0, 3, 4}, {0, 1, 3}, {}, {2, 3, 4}, {0, 2, 4}, {1, 2, 4}, {0, 1, 4}, {0, 1, 2}, {1, 2, 3}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {1, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	0	1
{{1, 2}, {0, 4}, {1, 4}, {}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {}, {2, 3, 4}, {0, 2, 4}, {1, 2, 4}, {0, 1, 4}, {0, 1, 2}, {1, 2, 3}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {1, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	0	1
{{2, 4}, {1, 2}, {0, 4}, {1, 4}, {}, {3, 4}, {2, 3}, {0, 2}, {1, 3}, {0, 3}, {0, 1}}	[{4}, {2}, {3}, {0}, {1}]	{}	0	1
{{0, 2, 3}, {0, 3, 4}, {0, 1, 3}, {}, {2, 3, 4}, {0, 2, 4}, {1, 2, 4}, {1, 3, 4}, {0, 1, 4}, {0, 1, 2}, {1, 2, 3}}	[{2, 4}, {3}, {0}, {1, 2}, {0, 4}, {1, 4}, {3, 4}, {4}, {2, 3}, {0, 2}, {1}, {1, 3}, {2}, {0, 3}, {0, 1}]	{}	0	1