Finding extremal elements in liar games

BSc Thesis

Author: Bertalan Borsos

Supervisor: Dömötör Pálvölgyi

Eötvös Loránd Tudományegyetem
Természettudományi Kar

Budapest, 2015

I would like to thank Dömötör Pálvölgyi for his insightful comments, his patience and for his thoroughness with which he proofread my work over and over again. Without his restless work this Thesis would not have been able to come half the way it did.

Contents

1 Introduction

Liar games first became an important subject of research when algorithmic problem solving rose to prominence in the $20th$ century. The possibility of having a computer ask the questions and interpret the answer opens up a new dimension considering the speed at which modern computing machines can function. But first, a couple of words about the models we are going to analyze.

The most basic one is a simple search. There are two players in the game, the Questioner and the Responder. They previously agree on a set of objects from which the Responder picks one and the Questioner's job is to find out which one it was. This set of objects is usually referred to as the search space. There are several different perspectives in this but the main one usually is how fast the Questioner can find the chosen element. And by how fast we mean the number of questions required in the worst case. For instance, going one by one through the elements of the search space is not an effective method as if the object we are looking for comes up last, we require n queries, where n is the number of elements in our set. It is easy to see that the problem can be solved in $\lceil \log n \rceil$ questions using binary search. This process revolves around narrowing the options to half in every step by dividing the search space into two partitions with relatively the same number of elements in each then asking whether the one we are looking for is an element of the first partition. Regardless of the answer we rule out about half of the possibilities.

But what if some of the answers are erroneous? Or if the questions we are permitted to ask have certain limitations? Or if we do not need to find only one element? These are some of the factors defining the model we are working with. For example the popular game 20 Questions has answers limited to a simple "Yes" or "No". The notion of representing this with binary code comes very naturally. However, no lies were permitted at all. The concept of the Responder lying gains importance for its applications in information theory. In the time when telecommunication utilizes radio waves it is very much possible that due to some environmental effect the information transmitted is corrupted, or at least part of it is. If we presume a maximum amount of information (i.e., number of bits) can be corrupted, finding an efficient way of decoding a message is equivalent to finding the right element of the set of possible words with lies, where lies represent the corrupted bits. In some other models every bit has a chance, a probability of arriving to the receiver corrupted. In this case solving the problem is only possible with a certain probability. The smaller the chance the algorithm's solution is wrong the better it is considered.

There is also the option of having to find more than one element. To give an example to this one too, let us introduce another simple model. The search space is going to be human faces, each with unique features. We have a photograph of someone and we want to find the person who resembles them the most and another one who resembles them the least of all possibilities in our search space. Let v_κ be the characteristic vector for the person κ in our search space. This means that every coordinate of this vector represents a human visual feature and every coordinate of the vector is 1 if said person possesses the actual feature and 0 if they do not. For instance if we are only looking at the gender, color of eyes and color of hair, then someone who has the same eye and

hair color as the person we are looking for but not matching in gender would have a characteristic vector of $\begin{pmatrix}0\\1\\1\end{pmatrix}$. Then let us define $s(v_i)$ as the sum of the coordinates of v_i. Now finding the maximal and minimal element of the possible $s(v_i)$ values also solves our original problem of finding the most and least resembling person.

Along with the type of permitted questions, number of lies and properties of the search space we also categorize different models as adaptive or not adaptive. The difference basically is whether the answers of the Responder can have an effect on the next question we are going to ask. If our questions depend on the answers received so far we say that our model is adaptive as it adapts its approach based on information already obtained. In case of a non-adaptive search the entire series of questions to be asked can be input right at the start even before we receive the first answer. In case of an adaptive one we evaluate the answer to the first question and calculate the next one trying to figure out what would be the optimal thing to ask in light of the answer to the previous question. Then we repeat this in every step. If we have the option to search adaptively, it is almost always the better solution, as for instance in case of our task being having to find the biggest number out of n numbers, if we first ask whether a particular one is the biggest and the answer is yes, adaptive search algorithms stop right away as they have completed the task. A non adaptive process however, would continue to execute all queries and then when it is finished evaluate and find that the solution was actually transparent after the first question. However adaptivity is not black and white. We can define k-batch models, meaning that at a time a fixed number k questions are asked, the Questioner receives the answer to those, has time to evaluate the results and then asks k more question. With this definition a totally adaptive search can be considered a 1-batch game.

A popular example of this difference is often brought up with e-error correcting codes and their decoding. An e-error correcting code is a set of binary sequences with fixed maximum length in which we can always restore a message sent through a noisy channel if we presume that at most e bits can be corrupted in every word. A code is e-error correcting if and only if the minimum Hamming distance of its words is $2e+1$. If there is no feedback we consider decoding a message a nonadaptive search problem. We can however define a noiseless, delayless feedback channel through which the receiver can send information back to the sender. This message arrives immediately and has no chance of being corrupted. This defines the same problem with the component of adaption as the sender receives information on every bit before sending the next.

It is also possible to apply other types of regulations to the questions. For instance, if we forbid asking the same question more than one time. Or, in case of a totally ordered set, we only allow a binary question regarding the relation of two elements. Or maybe both. Of course this in some cases can make it impossible to find the maximum element of a set of numbers for instance, if we use a comparison based search process. If the two biggest numbers are compared and a lie is allowed, when we compare these two and the answer is false we are not going to be able to decide between the two. What can we say then?

In the following pages we are going to build up and discuss different liar game

models, starting from scratch and introducing multiple object finding methods, the concept of optimal algorithms for these tasks, lies and their limitations and a couple of questions that arise on the way. Unless otherwise noted, we presume adaptivity and no extra regulations for the questions.

2 Logarithmic search

If we have a finite set of elements and we want to find exactly one of them, the most used algorithm for this is logarithmic search. As described in the introduction it follows a simple thought process. We want to eliminate the most possibilities in every step, so we split the search space in half and ask the question: which half the target element is in. This is also a yes/no question so it is allowed when there are strict regulations regarding the queries. The runtime of this method is $\lceil \log n \rceil$ as it halves the search space in every iteration.

Remark 1. There is a strong connection between searching for a particular element in a set, and searching for a number amongst a set of integers if the set contains a finite number of elements. We can arbitrarily label the elements of the set with integers from 1 to n, where n is the number of elements in our set. In this case finding the label of an element is equivalent to finding the element itself, as the connection is basically a bijection. This simplifies any guessing game over a finite set to the same game with numbers.

Remark 2. If our set is finite and ordered we can label its elements with numbers from 1 to n, where n is the number of elements in the set. Then the run of a logarithmic search algorithm can be simulated with the questions regarding the different digits of the binary representation of this number. We are going to use these observations later on. Logarithmic search is often referred to as binary search, possibly because of this connection.

2.1 Using logarithmic search to find a specific element of a set with one lie permitted

In this section we introduce the concept of lies, what they represent and how we can deal with them. At this point it is no secret that the models we are working with operate on the Questioner using binary questions.

Definition 3 (Lie)**.** When the Responder gives an erroneous answer we are going to refer to this answer as a lie. If in a series of questions we are able to find all lies, we can reduce our problem to a simple search problem.

Theorem 4. *If a finite set of arbitrary elements is given and the Responder selects one at will, the Questioner can find this element using less than* $\lceil \log n \rceil + \lceil \log (\lceil \log n \rceil + 1) \rceil + 1$ *questions, if the Responder is giving at most one erroneous answer.*

Proof. We are going to use the concept described in Remark 2. Let us say we have a finite search space S consisting of arbitrary elements. The Responder selects $t \in S$. We associate every element in S with an integer as described above and run logarithmic search on these numbers using the digit questions. So far we have used exactly as many questions as many digits these numbers have, which is $\lceil \log n \rceil$ at most, where n is the number of elements in S. The answer to one of these could be false. Here we add one more question: did you lie yet? The answer to that is either yes or no. However, if the

answer is no, it must be true as if it was false it would mean the Responder did indeed lie and then lied again denying it. But we only allowed one lie. This means all answers were correct and we have reached our goal using $\lceil \log n \rceil + 1$ queries. If the answer is yes, we have two options. Either the first series of queries was answered right and we are being lied to right now or there was a false answer in the original search. In this case if we find the erroneous bit we find the solution because we change that bit to 1 if it was 0 or to 0 if it was 1 and the element associated with the new label number is the one selected by the Responder. In order to do that let us define $\bar{S} := \{b_i \mid i \in \{1, .., n\}\}$, where b_i represents the answer of the Responder to the *ith* question. Then we add one more element to $\bar{S}$, the "did you lie" question. The task at hand is nothing else but finding one selected element (the erroneous answer) of a set of $\lceil \log n \rceil + 1$ elements. Naturally instead of asking one by one for every b_i whether it was erroneous, we can run logarithmic search. This in case of a set of $\lceil \log n \rceil + 1$ elements has a runtime of at most $\lceil \log (\lceil \log n \rceil + 1) \rceil$ questions. As $\lceil \log n \rceil$ is an integer, $\lceil \log n \rceil + 1$ also is, meaning the second ceiling sign is redundant. Combining this with the original search yields $\lceil \log n \rceil + \lceil \log (\lceil \log n \rceil + 1) \rceil + 1$ questions in the worst case. □

Remark 5. In fact, although the present version is easier to see, a method exists with a slightly better runtime than this one. More importantly it is also able to deal with nonadaptive problems. It is called Hamming code and it is widely considered a cornerstone in the theory of k-error correcting codes. However, this is not a subject of this current work.

3 Finding extremal elements with no lies using comparisons

Definition 6 (Extremal elements)**.** In a finite set of real numbers we will refer to the minimum and the maximum as extremal elements.

3.1 Comparisons

So far the problems we have touched on had very little regulations regarding the questions the Questioner was permitted to ask. Even limiting them to be yes/no questions was not that strict as someone looking for a specific element had the liberty to ask questions like "is this element in this particular subset?". Or even more bluntly "is it this element?". From now on we will focus on models where the only legal question is a comparison. The Questioner provides two different elements of the search space and the Responder tells which one is bigger. For this to make any sense of course we need to define an ordering relation on the elements of the search space. From this point on we are going to assume that there is a relation defined for every search space S meaning that for every $x, y \in S$ either $x \leq y$ or $x \geq y$ is true. The ordering is not necessarily strict as finding one of the maximal elements is enough but it is, obviously, necessarily transitive as this property is pivotal to most of the algorithms being used here. It is intuitive that if we are able to find the maximum of a set of elements some way, using

the same algorithm n times gives us the entire ordering. In practice however there are less straightforward but more efficient methods for solving this problem.

Remark 7. Considering that allowing the relation not to be strict does not really make a difference as in this case finding one of the maximal elements is sufficient and finding one of these maximal elements is exactly as hard as finding the single maximum in case of strict ordering we can assume, for the sake of simplicity, that for every two elements either $x < y$ or $x > y$ is true. We are going to use this assumption from now on.

3.2 Runtime

This time we are looking at the maximum search problem with comparisons but without lies. There are a number of methods for finding the maximal or minimal element of any totally ordered set using questions restricted only to a simple comparison of two elements to which the answer is binary: either the first one is greater than the last one or the other way around. An efficient algorithm can solve this problem using $n - 1$ comparisons.

Theorem 8. *Any algorithm designed to find the maximum (or minimum) of a set of totally ordered elements via comparisons requires* $n - 1$ *comparisons.*

To be able to prove this, two new definitions are required.

3.3 Introducing two new concepts

To be able to deal with certain situations easier we define two notions that might prove useful in following sections as well.

Definition 9 (Evil Adversary)**.** In the analysis of algorithms usually the goal is to find the runtime in the worst case scenario rather than the best one. The Evil Adversary is the symbol of everything going wrong for the user, or in our case the Questioner. This allows the Responder not only to give the least desirable answer for us but to change the maximum element at will during the run of the algorithm if it does not conflict with the answers given earlier. This is an efficient way of simulating the worst, most unlucky run of the algorithm in order to see if it still completes the task necessary and how badly does it effect complexity.

Definition 10 (Comparison Graph)**.** In case of a set with a high number of elements it can be extremely difficult to keep track of every comparison used in the search. This is likely to get even more messy when lies are also possible. A comparison graph is a simple representation of the run of a comparison based search algorithm. The vertices of comparison graph V represent the elements of the search space. At the start of the run there are no edges. After every comparison we add an edge connecting the two vertices symbolizing the two elements being compared then we direct the edge towards the "winner".

Remark 11. As we have assumed that the ordering relation is strict, the Comparison Graph for any algorithm dealing with a simple search problem without lies is likewise assumed an acyclic digraph. If there was a cycle of k edges in the Graph, $k - 1$ of its edges would determine the direction of the kth edge thus rendering it redundant to ask.

Let us use the ideas described above to prove Theorem 8.

Proof of Theorem 8. Indirectly presume that we have found the maximum element in less than $n-1$ comparisons. This means that Comparison Graph G has less than $n-1$ edges at the end of the algorithm. This means that the graph is necessarily disconnected. The algorithm has found maximum element v, thus $v \geq u \ \forall \ u \in G$. Now let us choose a different component of G than what v is in. This component G' has a local maximum element v' for sure, as the search space was a totally ordered set. Then we can have the Evil Adversary change the relation of v and v' to $v < v'$. As there are no edges between G and G' this change does not conflict with any previous comparisons, but v is not the maximum. □

3.4 Search algorithms

Now that we know the problem can not be solved using less than $n-1$ comparisons, let us present two processes that actually complete this task using that many queries. Probably the most instinctive idea is to search for the maximum linearly. We start from one of the elements and start comparing it with the others. As long as it is greater than the other elements, we keep it but as soon as we find an element greater than the current one, we select this new "temporary maximum" and continue with comparing it with the ones unused before. Obviously when we run out of untouched elements, the temporary maximum is the actual maximum. Also, the Comparison Graph of this method is a directed tree with the maximum having out-degree zero, meaning that based on Theorem 8 it is also optimal runtime-wise.

Another approach to build a tree for a comparison graph is to start from the lowest level. We create pairs of the elements at random and compare them. The "loosers" obviously can not be maximal as there already is at least one other element higher in the ordering. The "winners", a set of $\frac{n}{2}$ elements are paired again and this iteration continues until we have selected the maximum. The number of comparisons required for this is $\frac{n}{2}$ for the lowest level, $\frac{\frac{n}{2}}{2} = \frac{n}{4}$ for the next and so forth yielding $\sum_{i=1}^{\frac{n}{2}} \frac{n}{2^i} = n-1$. This process is called tournament search.

Remark 12. Although methods for finding the maximum element usually use $n-1$ comparisons the real importance of having multiple ways for this lies in the sorting algorithms based on them. A sorting algorithm does not only find the maximum, it finds the exact ordering of the set. There are a lot of sorting algorithms such as bubble sort, insertion sort, selection sort, heap sort, quick sort and many more. They differ in their runtime and when they are usually used in practice. For instance, selection sort is based on the linear search described earlier and has a runtime of $\mathcal{O}(n^2)$ and is almost never used in reality. Heap sort however, is based on tournament search and is very often an optimal method with a runtime of $\mathcal{O}(n \log n)$.

3.5 The weakness of the intuitive solution for extremal element search

It is trivial that if we are able to find the maximum number of a set with an algorithm, finding the minimum can be done with the same algorithm if we negate every element then run the maximum search. However this usually runs into redundant comparisons. Let us introduce an example.

Example 1. *Let the search space be* $S := \{1, 2, 3, 4\}$*. We use linear search, only in the wrong order. We compare* 4 *and* 3*, then* 4 *and* 2*, then* 4 *and* 1*. We have found that the maximum is* 4*. Now we negate the elements and do the same search.* $-4 < -3$*,* $-3 < -2$*,* $-2 < -1$*. The conclusion is that* 1 *is the minimum. But we would have been able to use the results of the first search to achieve the same result in half the time. In fact every single comparison in the second run of the maximum search was useless, as we already had the relation of these elements from search number one.*

Now we approach the same problem with tournament search. We compare 1 *and* 2*,* 3 *and* 4*. Then we compare the winners* 2 *and* 4 *and we successfully found the maximum. Negating the elements and running the same search for the minimum comes next.* $-1 > -2$*,* $-3 > -4$*,* $-1 > -3$*. Once again we find that two out of three comparisons were redundant.*

3.6 Double tournament search

Now we provide a method to find the minimum and the maximum using less than $\lceil \frac{3}{2}n \rceil - 2$ comparisons. We are going to refer to this method as double tournament search. The name comes from the nature of comparisons used in it. We first pair the elements randomly. Using $\lfloor \frac{n}{2} \rfloor$ comparisons we get a set of $\frac{n}{2}$ elements, the winners and a set of the same size, the losers. Obviously, none of the winners can be the minimum as there is at least one element smaller than him, the one we just compared it with. Likewise none of the losers can be the maximum. The comparisons we are going to use to find the maximum are going to be amongst the winners only, just as the comparisons required to find the minimum will only consider losers. In this case though all we have to do is find the maximum of two sets of $\lceil \frac{n}{2} \rceil$ or $\lfloor \frac{n}{2} \rfloor$ elements, which according to Theorem 8 uses $\lceil \frac{n}{2} \rceil - 1$ comparisons at most even when considering the possible odd one out in case of n being an odd number. This adds up to $2(\lceil \frac{n}{2} \rceil - 1) + \lceil \frac{n}{2} \rceil = \lceil \frac{3}{2}n \rceil - 2$.

Corollary 13. *The problem of extremal element search without lies can be solved using* $\lceil \frac{3}{2}n \rceil - 2$ *comparisons.*

Remark 14. The algorithm also serves as a proof for this upper bound. In fact, $\lceil \frac{3}{2}n \rceil - 2$ is also a lower bound. We are going to prove this in a later section.

Remark 15. An idea emerging at this point is whether checking for an element and trying to prove it is maximal or proving of every other element that they can not be the maximum is faster. When we consider lies this notion is going to become vital for designing algorithms.

4 Maximum search with k lies

4.1 Differences from the original model

So let us suppose there might be one erroneous answer given by the Responder. Introducing lies in a comparison based search effects the Comparison Graph in a way that might seem inconvenient at first but has a really niche use. With erroneous answers possible the Comparison Graph might now contain directed cycles.

Theorem 16. *If the Comparison Graph contains a directed cycle, at least one of its edges is erroneous.*

Proof. Suppose every one of its edges is true. This means $x_1 < x_2 < ... < x_k < x_1$ also holds for the elements of the search space forming the cycle. But then $x_1 < x_1$ is also true based on the transitive property of the relation. However, we have presumed our relation being strict in Remark 7. □

Remark 17. The Questioner is able to use this to find lies. The shorter the cycle the easier it is to find the lie, as the lie can be found for instance by asking every question in it until one of the comparisons result in an edge with the opposite direction it originally had. These two edges can not be true at the same time because of Remark 7. Forcing a directed cycle is a tool the Questioner can use to find lies as fast as possible. We are going to use this later on.

Consistency Property. The Adversary never lies. This also means no cycles are possible.

We are going to assume this Property several times from now on.

The corresponding theorems are always true even with the possibility of cycles but we are going to presume the Consistency Property for sake of simplicity, meaning the Comparison Graphs do not contain cycles. Since in every model we are analyzing $n > k$ by a pretty big margin, from now on we are going to assume as a rule of thumb that the Adversary never lies. This is something we are going to use multiple times in future proofs. Regardless of the number of lies allowed the Evil Adversary is not going to want to contradict his answers by creating directed cycles. If he does, the Questioner can simply disregard all edges of the cycle and proceed knowing $k - 1$ further lies are possible including both the edges already present and questions yet to be asked. The fact that this actually helps the Questioner is not trivial and will be proved each time for the current problem.

4.2 The battleship analogy

The battleship analogy is simply a way of describing the inverse approach to maximum finding. The intuitive thought is that when we want to find the maximum element, we have to compare it with every other element, not necessarily directly but using the transitive property of the ordering relation we have defined. This might work well

without lies but with the introduction of potentially erroneous information it runs into a problem. If we imagine the Comparison Graph for a search like this even if we find the maximum it might be a false result. For instance if we use tournament search and the first comparison was wrong right away we have sentenced the actual maximum to losers and we are not going to find it ever again using the algorithm as we have presumed it not being maximal and we are not even going to compare it with anything. The point of this analogy is to imagine the elements of the search space as battleships floating on top of the sea. Comparisons are cannonballs fired from one ship to another. In case a query yields $x_i > x_j$, x_j takes a hit. A ship sinks if it takes too many hits. But what is too many? Well, if we allow k number of lies and a ship has taken $k+1$ hits it cannot be the maximum, as even if k of the $k+1$ answers were erroneous, there is at least one true answer meaning there is at least one element higher than x_j. Now let us introduce this concept a bit more formally.

Definition 18 (Potential function α'_k)**.** For every vertex v of Comparison Graph G, $\alpha'_k(v)$ equals to $k+1$ minus the in-degree of v.

Remark 19. Actually, $\alpha'_k(v)$ is exactly the number of comparisons v has to lose in order to prove it can not be maximal. We can imagine that as a counter starting from the number of possible lies plus one. At the start every element has α'_k of $k+1$ and this decreases by one every time v loses a comparison. If v has $k+1$ losses and k lies are allowed then based on the pigeonhole principle at least one of them was true so v did indeed lose at least one comparison, thus it can not be the maximum as there is at least one element bigger than v. This also means any vertex v with $\alpha'_k(v) = 0$ or lower can not be the maximum.

Remark 20. By the definition of α'_k it is possible it takes negative values throughout the run of the search. Whenever the α'_k value of a vertex reaches 0 its further decrease does not hold any useful information for us. For the sake of simplicity we can use an alternate potential with a minimum of 0.

Definition 21 (Potential function α_k)**.** Just for clarity reasons we define $\alpha_k(v) = max\{\alpha'_k(v), 0\}$ for every element of the search space.

Lemma 22. *If there is one element v left with $\alpha_k(v) \neq 0$, it is the maximum.*

Remark 23. In the case of no lies allowed, $\alpha_0(v) = 1$ for every vertex in the Comparison Graph at the beginning. As the search progresses every time we compare two elements the loser has $\alpha_0(v) = 0$ immediately. This fits into the process of both linear and tournament searches very well. In case of one lie allowed $\alpha_1(v) = 2$ for every $v \in G$. Then our task is to find a method of comparisons that eventually leaves only one $u \in G$ with $\alpha_1(u) \neq 0$.

4.3 Runtime

The main difficulty here is dealing with a possibly high number of lies. Even using Comparison Graphs and/or other data structures does not simplify the problem enough. If

one was to use the α function described above, the possibility of using more comparisons than necessary is inevitable if we do not utilize the information that can be deduced from the transitive property of the relation. We do however have a number of upper and lower bounds.

Theorem 24. *Finding the maximum is always possible using* $(k+1)(n-1)+k$ *comparisons*

Remark 25. All that is required for this is to use a maximum search algorithm, any efficient one with a runtime of $n-1$ comparisons, and modify it so that every time two items are compared, we compare them until we get the same answer $k+1$ times.

Theorem 26 (Ravikumar et al, [8])**.** *Finding the maximum element of a finite totally ordered set with k lies permitted can not be done using less than* $(k+1)(n-1)$ *comparisons.*

Proof. Suppose we have less than $(k+1)(n-1)$ comparisons. This is equivalent to a Comparison Graph of n points with the sum of in-degrees being less than f$(k+1)(n-1)$. Because of the Consisteny Property we also assume the Graph is acyclic. This means that based on the pigeonhole principle there are at least two elements with in-degree $\leq k$. There is an Adversary's Strategy in which one is the maximum and one in which the other one is, meaning the Questioner could not have found the maximum as whichever of the two he chooses, the Adversary can manipulate the ordering so that the other one was the maximum. □

This result can also be improved using case by case analysis.

Theorem 27 (Ravikumar et al, [8])**.** *Finding the maximum element of a finite totally ordered set with k lies permitted can not be done using less than* $(k+1)(n-1)+k$ *comparisons.*

5 Finding the minimum and the maximum with k lies

Similarly to the matters discussed above we will analyze the optimal algorithms for finding the maximum and the minimum. We have previously taken a look at parallel extremal element finding and searching with a fixed number of lies. Now we combine these two and define our model as follows.

There is a given finite set of elements, with a total ordering relation defined on it. The Questioner is allowed to ask an unlimited number of questions but every question must be whether element v of the search space is bigger than u or vice versa. There is a fixed number k known by the Questioner and the Responder prior to the search. At most k times the Responder is allowed to give an erroneous answer. We call this an adaptive problem or a 1-batch problem as the Questioner can alter his strategy based on the answer in every step, every bit actually. The goal of the Questioner is to find both the minimal and the maximal element of this set using the least possible comparisons. As usual we will consider the worst possible case for the Questioner as defined by the notion of the Evil Adversary described in Definition 9.

5.1 Potential function β_k

The battleship analogy can be extended to this problem. As we have discussed before, it is usually easier to eliminate possible extremal elements than to prove of a randomly chosen element that it is the maximum or the minimum. This is especially true for models with a higher number of lies allowed. To make this process easier we will introduce a potential based on α_k.

Definition 28 (Potential function β_k)**.** The concept is very similar to the one used in the maximum search, only now we need to keep track of both extremes. In order to do that we define our double value potential function as follows. We say that if our original Comparison Graph was V, let V' stand for the reverse Comparison Graph in which every edge is redirected to the opposite direction. For every $v' \in V'$, v' has in-degree equal to the out-degree of v and out-degree equal to the in-degree of v. It is basically just the extension of the thought we have used before with negating numbers to find the minimum using a maximum search algorithm. Based on this, α_k is a tool for finding elements that can not be maximal in V', thus can not be minimal in V. Now we say for every v in the search space $\beta_k(v) = (\alpha_k(v), \alpha_k(v'))$. In other words the β_k values of any element of the search space are the number of comparisons it has to lose to prove it can not be the maximum and the number of comparisons it has to win to prove it is not the minimum, given of course k erroneous answers are allowed.

Claim 29. *If $\beta_k(v) = (0, 0)$, v can not be an extremal element.*

Theorem 30. *If there are exactly two elements with $\beta_k(v) \neq (0, 0)$, they are the extremal elements. Furthermore, if for both either the number of wins or the number of losses is k, we can tell exactly which one is the maximum and which one is the minimum.*

5.2 Upper bound for $k = 1$

Theorem 31 (Gerbner et al, [2])**.** *If we presume one erroneous answer is possible, we can find the minimum and the maximum element using $\lceil \frac{87}{32} n \rceil + 12$ comparisons.*

To prove this we need to provide a Questioner's Strategy of comparisons that lets him find both the maximum and the minimum of a set no matter what the Evil Adversary opts to answer. This however can prove difficult sometimes. Let us introduce some ideas that might be useful in proving both upper and lower bounds.

Remark 32. A comparison graph may contain true answers if and only if it is acyclic. If there is a directed cycle, one of its edges is necessarily false as described in the Proof of Theorem 16. If we find a directed cycle, we can simply disregard all of its edges and continue the search. We can be sure that amongst every other comparison, including already existing edges in the Comparison Graph and forthcoming ones, there might be $k - 1$ lies at most.

Remark 33. In order to prove any lower bound, we have to provide a Questioner's Strategy solving the problem, no matter the Responder's Strategy.

Remark 34. In order to prove any upper bound, we have to provide an Adversary's Strategy so that no matter what the Questioner asks, there are multiple possibilities still remaining after the given number of questions.

Based on this we now provide a way that allows the Questioner to find the extremal elements in $\lceil \frac{87}{32}n \rceil + 12$. For the sake of simplicity, from now on we omit all ceiling signs.

Proof of Theorem 31. The idea is pretty simple. We will try to create pairs of the elements with matching β_1 values. At the beginning $\beta_1(v) = (2, 2)$ for every $v \in V$. We create a matching and use the results to partition the elements into two sets, V_w, the winners and V_l, the losers. This so far was $\frac{n}{2}$ comparisons. Every element in V_w has $\beta_1 = (2, 1)$ and every element in V_l has $\beta_1 = (1, 2)$. Now in the second round we split the elements in V_w into two using a maximum matching and $\frac{n}{4}$ comparisons. That provides us with $V_{w,w}$ and $V_{w,l}$ with β_1 values of $(2, 0)$ and $(1, 1)$ respectively. Similarly with another $\frac{n}{8}$ comparisons we can split $V_{w,l}$ into $V_{w,l,w}$ and $V_{w,l,l}$ with β_1 values of $(1, 0)$ and $(0, 1)$. Now take the elements in V_l that were paired with the teams in $V_{w,l,l}$. We denote this subset of V_l by V_{l+}. We now create a matching in V_l so that every element in V_{l+} is paired with another element from V_{l+}. These $\frac{n}{4}$ comparisons yield another two subsets, $V_{l,l}$ and $V_{l,w}$ with β_1 values of $(0, 2)$ and $(1, 1)$. Here we note that $\mid V_{l+} \cap V_{l,w} \mid = \frac{n}{16}$. Now we use a fifth matching in $V_{l,w}$ that is an extension of a matching of $V_{l+} \cap V_{l,w}$. After another $\frac{n}{8}$ comparisons we have $V_{l,w,l}$ and $V_{l,w,w}$ with $(0, 1)$ and $(1, 0)$. Now we compare the elements of $V_{l,w,w} \cap V_{l+}$ with their first round, original opponents. Note that those comparisons went in favor of the elements in V_w, this is the reason they are now in V_w.

From this point we consider two options. The first one is if all the comparisons go exactly the same way as the first time. This means that now all the teams involved in this last, sixth round of matching now have β_1 of $(0, 0)$, so are ruled out as extremal elements. This means that after $\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{4} + \frac{n}{8} + \frac{n}{32} = \frac{41}{32}n$ comparisons we have $\frac{n}{16}$ elements with $(0, 0)$, $\frac{n}{4}$ each of $(2, 0)$ and $(0, 2)$, and $\frac{7}{32}n$ of $(0, 1)$ and $(1, 0)$. The number of elements in our search space is not necessarily a power of 2, so we must address this issue as well, as there can be an additional element for every possible β_1 value. In order to reach our position described in Theorem 30 we need $2(\frac{n}{4} + \frac{n}{4}) + 1(\frac{7}{32}n + \frac{7}{32}n) + 14 = \frac{46}{32}n + 14$ comparisons. Now with clever pairing, electing elements with β_1 of $(0, 0)$ not to play and pairing $(2, 0)$ and $(0, 2)$ teams with each other respectively yields the total number of comparisons used in this algorithm, $\frac{46}{32}n + 12 + \frac{41}{32}n = \frac{87}{32}n + 12$.

Now let us consider what would have happened if the repeated comparisons would have yielded parallel edges with not matching direction. Note that this obviously means one of them is erroneous, so there can only be one of these double edges, as we have presumed at most 1 erroneous answer. Now we can use Remark 32. We simply delete both edges, not knowing which one was the real result. We lose two edges but gain a lot more. For every $v \in V$, where $\beta_1(v) = (x, y)$ let $\beta_1(v) = (x - 1, y - 1)$. This means that after the $\frac{41}{32}n$ comparisons we have every element with $(0, 0)$, $(1, 0)$ or $(0, 1)$, with the exception of one possible $(1, 1)$ element in case n was odd. This means we can finish the search with at most n queries afterwards yielding $\frac{73}{32}n$ comparisons. □

Reflecting on the final lines of the proof above, we can note that the Evil Adver-

sary would never contradict his answers in the sixth round as it leads to the Questioner reaching his goal faster. This branch of the proof could have been left out if we presumed the Consistency Property.

5.3 Algorithm for arbitrary k

Now we describe a generic algorithm that can be used for any k. As an algorithm can be defined as a Questioner's strategy, the goal is the same as with $k = 1$. Find the minimum and the maximum with k lies possible using comparisons only. The method described here can be implemented for arbitrary k, but its runtime is not going to be optimal. We are going to look at the required number of queries later, now let us describe the process itself.

Definition 35 (β Class Matching Algorithm)**.** Very similarly to the algorithm used for the one lie case, the generic method relies on pairing elements with the same β_k values. If k lies are allowed at the beginning of the search $\beta_k(v) = (k+1, k+1)$ holds for every $v \in V$. These values decrease every time the out-degree or the in-degree of v increases. Based on Theorem 30, the condition we have to reach is every element having $\beta_k = (0, 0)$, except for two, one of which must have at least one β_k value of 0. In this case we know those are the extremal elements. If we reach this point, we are finished so we are going to refer to this condition as the win condition.

Note that every time we pair two elements with (a, b) together, where $a, b \neq 0$, after the comparison we will have one with $(a-1, b)$ and the other one with $(a, b-1)$. Now we start the algorithm by creating a maximum matching of the search space, then continue with creating pairs of equal β_k. When all the elements have $(0, i)$ or $(0, j)$, where $i, j \in \{1, ..., k+1\}$, all that changes when pairing equal β_k value elements is that the one possible outcome of the comparison is one of the elements retaining its β_k value and for the other it changes from $(0, i)$ to $(0, i-1)$ or from $(i, 0)$ to $(i-1, 0)$.

Using these steps we are going to reach the win condition in a finite number of comparisons, as $\sum_{v \in V} \beta_k(v)_1 + \beta_k(v)_2$ decreases strictly in every step if we do not use $(0, 0)$ elements in comparisons. The process also ensures that there is always at least one element with positive number of losses still required and another one with positive number of wins required to prove they can not the maximum and the minimum respectively.

Remark 36. This process relies solely on the β_k function, without using any additional information that could be attained from the Comparison Graph. This also implies that the problem could probably be solved using less comparisons in most cases.

We have seen that the β Class Matching Algorithm ends in a finite number of steps, but how many steps does it actually take to find the extremal elements with this method? To evaluate this, we first introduce $C(n, k)$.

Definition 37 ($C(n, k)$)**.** $C(n, k)$ denotes the minimum number of comparisons required to reach the win condition described in the definition of the β Class Matching Algorithm if we presume that a maximum number of k lies are possible and our search space consists of n elements.

Claim 38. *$C(n,k)$ comparisons are also enough to find the maximum and the minimum in the liar game model we have been working with. This is a direct result of Theorem 30.*

At this point it would be suitable to have a function, c, that for every $v \in V$, denotes the number of queries the Questioner needs to have v reach $(0,0)$, thus eliminating it from the race for extremity. This notion is a bit too wacky to be the definition of a function, as it can not be ensured that this value is finite for every $v \in V$. In fact, the extremal elements will never reach $(0,0)$, no matter the number of queries. If they ever reached $(0,0)$, it would mean they are not extremal, which is an obvious contradiction. For this reason we take a different approach.

Definition 39 ($c(v)$)**.** We define symmetric potential function $c : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ so that $c(v) = c(\beta_k(v))$. For any $l \in \mathbb{N}$, $c(0,l) = c(l,0) = l$. All other values are defined by the recursive equation $2c(j,l) = c(j-1,l) + c(j,l-1) + 1$.

The recursion in Definition 39 symbolizes the connection between the elements of the search space. If the Questioner compares two elements and they have matching β_k values, which they should of course when using the matching algorithm we described in Definition 35, after the comparison of two elements of (j,l), we get one $(j-1,l)$ and one $(j,l-1)$. The plus one comes from the fact that we need that one comparison in order to reduce the β_k values to lower class ones. Now we make the following two observations.

Claim 40. *If we compare two elements $v, u \in V$, so that $\beta_k(v) = \beta_k(u) \neq (0,0)$, the sum $\sum_{w \in V} c(w)$ decreases by exactly one in every step, because of the recursion defining $c(v)$. On the other hand using the pigeonhole principle, as long as we have at least $(k+1)^2$ elements with $\beta_k \neq (0,0)$ we can always create such pairings. This means that the win condition can be reached in $c(k+1,k+1)n + \Theta_k(1)$ comparisons.*

Claim 41. *It is also easy to see that whenever two elements are compared with β_k (j,l) and (m,p), the two possibilities for β_k value change are as follows. Either we have $(j,(l-1)^+)$ and $((m-1)^+,p)$ or $((j-1)^+,l)$ and $(m,(p-1)^+)$, where x^+ stands for $max\{0,x\}$. Obviously in any Adversary's Strategy, the Adversary's goal is to have the sum $\sum_{w \in V} c(w)$ decrease by the smallest possible amount. Similarly to Claim 40, if we add the decrease for the two possible outcomes of the comparison, $\sum_{w \in V} c(w)$ decreases by 2. This means that for one of the outcomes of every comparison the sum decreases by at most 1. The Adversary will answer every query to give minimum information, meaning the sum will decrease by 1 at most in a step. This gives $C(n,k) \geq c(k+1,k+1)n - c(k+1,0) - c(k+1,k+1) \geq c(k+1,k+1)n - 3k - 3$.*

At this point it is pretty clear that finding $c(l,l)$ for arbitrary l is pivotal in reaching our final goal of this section, evaluating $C(n,k)$. There are two different approaches to this problem. One is using dynamic programming to obtain $c(l,l)$. As the recursion holds for any two integers, $2c(l,l) = c(l-1,l) + c(l,l-1) + 1$. This process involves calculating $c(x,y)$ for every $x, y \in \mathbb{N}$, where $x, y < l$. This should pose little problem to a computer, however we only really need the values of $c(l,l)$, so finding an explicit

form for this makes things a lot easier. Let us start with dynamic programming. The following piece of C++ code evaluates the $c(j,l)$ values for $j,l \in \{1,...,10\}$ and prints them.

Example 2.

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    double c [20][20] = { }; //Array
    for (int i=0; i<=9; i++) //Updating initial values
    {
        c[0][i]=i;
        c[i][0]=i;
    }

    for (int k=2; k<=18; k++) //Using the recursion
    {
        for(int j=1; j<=k-1; j++)
        {
                c[k-j][j]=((c[k-j-1][j]+c[k-j][j-1]+1)/2);
        }
    }

    for (int i=0; i<=9; i++) //Displaying values
    {
        for (int j=0; j<=i; j++)
        {
            cout << "c(" << i << "," << j << "): " << c[i][j] <<"    " ;
       }
        cout << endl;
    }
    return 0;
}
```

Using this algorithm we can obtain $c(l,l)$ values up to a pretty big l. The recursion could also have been implemented with a recursive function call, but in order to find $c(l,j)$ for arbitrary l and j, even the recursion needs to find most of the lower class c values. Given this we present the table of numbers we have just found.

	c(*, 0)	c(*, 1)	c(*, 2)	c(*, 3)	c(*, 4)	c(*, 5)	c(*, 6)	c(*, 7)	c(*, 8)	c(*, 9)
c(0, *)	0	1	2	3	4	5	6	7	8	9
c(1, *)	1	1.5	2.25	3.125	4.0625	5.03125	6.01562	7.00781	8.00391	9.00195
c(2, *)	2	2.25	2.75	3.4375	4.25	5.14062	6.071812	7.04297	8.02344	9.0127
c(3, *)	3	3.125	3.4375	3.9375	4.59375	5.36719	6.22266	7.13281	8.07812	9.04541
c(4, *)	4	4.0625	4.25	4.59375	5.09375	5.73047	6.47656	7.30469	8.19141	9.11841
c(5, *)	5	5.03125	5.14062	5.36719	5.73047	6.23047	6.85352	7.5791	8.38525	9.25183
c(6, *)	6	6.01562	6.07812	6.22266	6.47656	6.85352	7.35352	7.96631	8.67578	9.46381
c(7, *)	7	7.00781	7.04297	7.13281	7.30469	7.5791	7.96631	8.46631	9.07104	9.76743
c(8, *)	8	8.00391	8.02344	8.07812	8.19141	8.38525	8.67578	9.07104	9.57104	10.1962
c(0, *)	9	9.00195	9.0127	9.04541	9.11841	9.25183	9.46381	9.76743	10.1962	10.6692

As $c(l, j) = c(j, l)\ \forall j, l \in \mathbb{N}$, the table is diagonally symmetric. Although all values are listed, all we really need is the diagonal elements. Now we make some observations considering earlier results.

Remark 42. The $c(l, l)$ results give us an upper bound for the comparisons required to find the minimum and the maximum with k lies, as the upper bound for the number of comparisons required for reaching the win condition, denoted by $C(n, k)$ is in strong correlation with $c(l, l)$ because of Claims 40 and 41. This gives us that using the β Class Matching Algorithm we can find the extremal elements with 1 lie in less than $2.75n + \Theta_k(1)$. We have, however, proved earlier in Proof 31, that this can be done in $(2.75-\frac{1}{32})n+\Theta_k(1)$ comparisons. This also suggests that for arbitrary k, $c(k, k)n+\Theta_k(1)$ comparisons are enough to find both extremal elements in every case but this upper bound is not tight, rather the same task would be possible using less comparisons. Here we note, that the β Class Matching Algorithm does not use additional information of the Comparison Graph, only the function values and as we have seen in the algorithm described in the Proof of Theorem 31. However, the Comparison Graph could be used to obtain the same result faster. One of the ways for this that we already have touched on earlier in Remark 17, is forcing cycles, where the Responder either contradicts his answers or lets the Questioner receive more information with a single question than necessary.

Although dynamic programming gives a fast and easy way to find $c(l, l)$ for a smaller k, it might take a long time to compute for a bigger number. Finding an explicit form for any $c(l, j)$ can prove difficult but as we have noted before we only really need identical values of the function.

Definition 43 ($c'(v)$)**.** Let $c'(v) = c'(\beta_k(v)) = c'(x, y) = 2^{x+y}c(x, y) - (a + b)2^{a+b-1}$. This transforms the recursion to a form that is a much easier to handle.

$$c'(x, y) = c'(x - 1, y) + c'(x, y - 1).$$

The starting values are defined by

$$c'(z, 0) = c'(0, z) = z2^{z-1}.$$

This resembles a binomial recursion, unfortunately with different starting values. We can still use the binomial theorem to define the following equation for c'.

Claim 44. *For every* $x, y > 0$ $c'(x,y) = \sum_{i=1}^{x} c'(i,0)\binom{x-i+y-1}{y-1} + \sum_{j=1}^{x} c'(0,j)\binom{y-1+x-j}{x-1}$.

This formula can be used to determine $c'(l,l)$.

Claim 45. *For every* $l \in \mathbb{N}$, $c'(l,l) = 2\sum_{i=1}^{l} c'(i,0)\binom{2l-i-1}{l-1} = \sum_{i=1}^{l} i2^i\binom{2l-i-1}{l-1}$.

This can be simplified very neatly using the definition of $\binom{n}{k}$ and the following lemma.

Lemma 46. $\sum_{i=1}^{l} i2^i\binom{2l-i-1}{l-1} = l\binom{2l}{l}$.

Proof.

$$2\sum_{i=1}^{l} i2^{i-1}\binom{2l-i-1}{l-1} = 2\sum_{i=1}^{l}\binom{2l-i-1}{l-1}\cdot\sum_{j=0}^{i} i\binom{i-1}{j} = 2\sum_{i=1}^{l}\binom{2l-i-1}{l-1}\cdot\sum_{j=0}^{i} j\binom{i}{j}.$$

As from the property of binomial coefficients

$$2^{i-1} = \sum_{j=0}^{i}\binom{i-1}{j}.$$

We now have

$$2\sum_{i=1}^{l}\binom{2l-i-1}{l-1}\cdot\sum_{j=0}^{i} j\binom{i}{j} = 2\sum_{j=1}^{l} j\sum_{i=j}^{l}\binom{i}{j}\binom{2l-i-1}{l-1}.$$

The inner part of this counts the number of binary sequences of length $2l$, with $l+j$ coordinates of 1. (Each part of the sum counts the ones where the $(j+1)th$ 1 comes at the $(i+1)st$ position. This gives us

$$\sum_{i=j}^{l}\binom{i}{j}\binom{2l-i-1}{l-1} = \binom{2l}{j+l}.$$

After a couple more steps of algebraic transformation we have

$$2\sum_{j=1}^{l} j\sum_{i=j}^{l}\binom{i}{j}\binom{2l-i-1}{l-1} = 2\sum_{j=1}^{l} j\binom{2l}{j+l} =$$

$$= 2\sum_{j=1}^{l}(k+j)\binom{2l}{j+l} - 2\sum_{j=1}^{l} l\binom{2l}{j+l} =$$

$$= 4l\sum_{j=1}^{l}\binom{2l-1}{l+j-1} - 2l\sum_{j=1}^{l}\binom{2l}{j+l} = 2(l2^{2l-1} - (l2^{2l-1} - \frac{1}{2}l\binom{2l}{l})) = l\binom{2l}{l}.$$

□

This also means that $c'(l,l) = l\binom{2l}{l}$, $\forall l \in \mathbb{N}$. This is a direct result of the Lemma above.

Corollary 47. *Using this and how we originally defined $c'(x,y)$, we get $c(l,l) = l + c'(l,l)/2^{2l} = l(1 + \binom{2l}{l}/2^{2l})$. Now we are able to use this to find $C(n,k)$.*

Theorem 48 (Gerbner et al, [2])**.** $C(n,k) = (k+1)(1 + \binom{2(k+1)}{k+1}2^{-2(k+1)})n + \Theta_k(1) = (k + \Theta(\sqrt{k}))n + \Theta_k(1)$.

Proof. After having found the explicit form of $c(k,k)$, all we need to do is use this with Claims 40 and 41. The substitutions give $c(k+1,k+1) - 3k - 3 \leq C(n,k) \leq c(k+1,k+1) + \Theta_k(1)$. This leads to $C(n,k) = (k+1)(1+\binom{2(k+1)}{k+1}2^{-2(k+1)})n + \Theta_k(1)$. □

Corollary 49. *It is clear that $C(n,k)$ being an upper bound for the runtime of the β Class Matching Algorithm is also an upper bound for the number of comparisons the Questioner needs to find the minimum and the maximum. As we have touched on earlier, these bounds are usually not tight as the additional information in the Comparison Graph is left unused.*

5.4 Lower bound for $k = 0$

Earlier we have seen that in case no answer can be erroneous, $\lceil \frac{3}{2}n \rceil - 2$ comparisons are sufficient for finding the extremal elements. The algorithmic proof of this described a method, a preplanned way of choosing questions for the Questioner showed a process that completes this task using no more than $\lceil \frac{3}{2}n \rceil - 2$ comparisons. We called this algorithm Double Tournament Search. Now we present a lower bound for the same task that actually shows that $\lceil \frac{3}{2}n \rceil - 2$ comparisons are also necessary in the worst case.

Theorem 50 (Pohl, [7])**.** *Finding the minimum and the maximum of a totally ordered set of n elements can not be done in less than $\lceil \frac{3}{2}n \rceil - 2$ comparisons in the worst case.*

Proof. We are going to prove the theorem with an approach very similar to β_k. In this particular case no lies are allowed, so we are using β_0. This also means that for every $v \in V$, β values are initialized as $(1,1)$, any element losing a comparison can not be maximal and any element winning a comparison can not be minimal since every result is presumed true. This gives us four subsets in the search space. V_w denotes the set of elements that already came out on top in at least one comparison. They can not be minimal but can still be maximal. Likewise V_l contains elements with at least one loss. They can not be the maximum but still can be the minimum. $V_{w,l}$ holds elements that have ruled themselves out as extremal elements by winning and losing at least one comparison. V_- is the set of elements that have not yet partaken in a comparison. Here we note that these partitions are actually the β_0 classes of the vertex set of the Comparison Graph. We presume that at the start of the search we have no information. This means every element has $(1,1)$, $\mid V_- \mid = n$, with all the other three subsets being empty. The problem is solved when $\mid V_{w,l} \mid = n - 2$. From here we proceed with case by case analysis on how the elements move between the subsets. There are 10 possible comparisons considering the subsets if V.

1. $V_w \leftrightarrow V_w$
   $\mid V_w \mid = \mid V_w \mid -1, \mid V_{w,l} \mid = \mid V_{w,l} \mid +1$
2. $V_w \leftrightarrow V_l$
   Partitions remain unchanged or $\mid V_w \mid = \mid V_w \mid -1, \mid V_l \mid = \mid V_l \mid -1, \mid V_{w,l} \mid = \mid V_{w,l} \mid +2$
3. $V_w \leftrightarrow V_{w,l}$
   Partitions remain unchanged or $\mid V_w \mid = \mid V_w \mid -1, \mid V_{w,l} \mid = \mid V_{w,l} \mid +1$
4. $V_w \leftrightarrow V_-$
   $\mid V_- \mid = \mid V_- \mid -1, \mid V_{w,l} \mid = \mid V_{w,l} \mid +1$ or
   $\mid V_- \mid = \mid V_- \mid -1, \mid V_l \mid = \mid V_l \mid +1$
5. $V_l \leftrightarrow V_l$
   $\mid V_l \mid = \mid V_l \mid -1, \mid V_{w,l} \mid = \mid V_{w,l} \mid +1$
6. $V_l \leftrightarrow V_{w,l}$
   Partitions remain unchanged or $\mid V_l \mid = \mid V_l \mid -1, \mid V_{w,l} \mid = \mid V_{w,l} \mid +1$
7. $V_l \leftrightarrow V_-$
   $\mid V_- \mid = \mid V_- \mid -1, \mid V_{w,l} \mid = \mid V_{w,l} \mid +1$ or
   $\mid V_- \mid = \mid V_- \mid -1, \mid V_w \mid = \mid V_w \mid +1$
8. $V_{w,l} \leftrightarrow V_{w,l}$
   Partitions remain unchanged
9. $V_{w,l} \leftrightarrow V_-$
   $\mid V_- \mid = \mid V_- \mid -1, \mid V_l \mid = \mid V_l \mid +1$ or
   $\mid V_- \mid = \mid V_- \mid -1, \mid V_w \mid = \mid V_w \mid +1$
10. $V_- \leftrightarrow V_-$
    $\mid V_w \mid = \mid V_w \mid -1, \mid V_l \mid = \mid V_l \mid -1, \mid V_{w,l} \mid = \mid V_{w,l} \mid +2$

If there is at least one element in V_- it can be the maximum or the minimum, so as long as there are elements left in V_-, we are not done. Similarly, if V_w and/or V_l contains more than one element those can still be optimal in that extreme. Thus as described earlier we focus on the potential decrease of $\sum_{v \in V} c(v)$ and denote an outcome of the comparisons listed above as favorable if this decrease is higher. $c(v)$ is the function defined in Definition 39. For sake of simplicity we declare its values used here: $c(0,0) = 0, c(0,1) = c(1,0) = 1, c(1,1) = 1.5$. Here we note that using only comparisons of type $1, 5$ and 10 (and one extra 7 or 4 in case n is odd) we can achieve our goal using $n - 2 + (n/2)$ comparisons. A special case of this is described in Corollary 13. Also

the Questioner now is never going to ask a question of type $2, 3, 6$ or 8 as the Evil Adversary would manipulate the elements so that the less favorable result occurs every time and the Questioner gains no information whatsoever. To prove the lower bound what we need to do now is to show that comparison types $4, 7$ and 9 can not improve the bound. Now consider the following. After i comparisons the set is partitioned so that $(\mid V_w \mid, \mid V_l \mid, \mid V_{w,l} \mid, \mid V_- \mid) = (a, b, c, d)$, but the relation between the elements of V_- and V_l is unknown since V_- elements did not take part in any comparison. So it is possible that every element in V_- is greater than any element in V_l, and a comparison between these two partitions would always yield the less favorable first alternative. It is also possible that $x < y$ holds true for any $x \in V_l, y \in V_w$ and once again a comparison between V_w and V_l would yield the less favorable first alternative for a result. This implies that for an arbitrary state of $(\mid V_w \mid, \mid V_l \mid, \mid V_{w,l} \mid, \mid V_- \mid)$, the number of comparisons required to find the minimum and the maximum is $\mid V_w \mid + \mid V_l \mid + \lceil \frac{3}{2} \mid V_- \mid \rceil - 2$. For the presumed starting state of $(0, 0, 0, n)$ this gives us exactly $\lceil \frac{3}{2} n \rceil - 2$ comparisons. $\square$

Remark 51. The proof above used a case by case analysis between β classes of the search space. The partitions $V_w, V_l, V_{w,l}$ and V_- actually stand for the elements with β_0 of $(1, 0), (0, 1), (0, 0)$ and $(1, 1)$ respectively. This indicates that proving lower bounds with an analogous method to the one used for $k = 0$ would be possible for $k > 0$ as well. This, however would require analyzing a quickly increasing number of cases.

5.5 Lower bounds for $k > 0$

We have previously seen that obtaining the maximum (or minimum) alone requires at least $(k + 1)n - 1$ queries in the worst case. Theorem 50 also shows that finding the extremal elements without lies might require $\lceil \frac{3}{2} \rceil n - 2$ comparisons. We continue by presenting results for $k > 0$.

Theorem 52 (Gerbner et al, [2])**.** *Finding the minimum and maximum of a totally ordered set of n elements if one lie is allowed is not possible using less than $\lceil \frac{87}{32} n \rceil - 3$ comparisons in the worst case.*

The best known lower bounds for this problem for a higher k are usually of the form $(k + 1 + c_k)n - D$, where D is a reasonably small constant. Indeed we have seen that $c_0 = 0.5$ and $c_1 = \frac{23}{32}$ with D staying a single digit number for $k = 0, 1$. Although $k \geq 2$ is still a subject of research, we do have an estimate of c_k.

Theorem 53 (Aigner, [1])**.** *For $\forall\ k \in \mathbb{N}$, $c_k = \Omega(2^{-5k/4})$.*

For a while it was conjectured that for every k, there is an algorithm that solves this problem using $(k + 1 + \epsilon)n$ comparisons. This was disproved in recent studies and instead the following was proved.

Theorem 54 (Pálvölgyi, [5])**.** *Finding the minimum and the maximum in this model with k lies allowed requires at least $(k + 1.5)n + \Theta(k)$ comparisons.*

References

[1] M. Aigner, Finding the minimum and the maximum, *Discrete Applied Mathematics* 74 (1997), 1-12

[2] D. Gerbner, D. Pálvölgyi, B. Patkós, G. Wiener, Finding the biggest and smallest element with one lie, *Discrete Applied Mathematics*, 158, 9 (2010) 988–995

[3] M. Hoffmann, J. Matousek, Y. Okamoto, P. Zumstein, Minimum and maximum against k lies, http://arxiv.org/abs/1002.0562

[4] T. K. Moon, Error correction coding, *John Wiley and sons*, (2005)

[5] D. Pálvölgyi, Lower bounds for finding the maximum and minimum elements with k lies, *Acta Univ. Sapientiae, Informatica*, 3, 2 (2011) 224–229

[6] A. Pelc, Searching games with errors - Fifty years of coping with liars, *Theoretical Computer Science*, 270 (2002)

[7] I. Pohl, A sorting problem and its complexity, *Communications of the ACM* 15 (1972), 462-464

[8] B. Ravikumar, K. Ganesan, K. B. Lakshmanan, On selecting the largest ekenebt ub spite of erroneous information, STACS (1987), 88-99