lemmaLemmaLemmas theoremTheoremTheorems definitionDefinitionDefinitions

AuSearch

Harlan Connor

July 2, 2020

1 Problem

In Ciphey, we need to navigate a tree of unbounded depth, trying to find a solution node. Each node falls into the following types:

label= Has zero or more children (a confirmable cipher)

lbbel= Has one child which is definitely a solution, or no children (an inferable cipher)

lcbel= Has zero or more children, and can be done in neligible time (an encoding)

Evaluating the children of a node is an intensive task, and so is checking nodes, so we wish to minimise both actions. We must find such a node if there is any reasonable chance that it exists (above a threshold ℓ_p), and if it can be found in reasonable time ℓ_t.

To aid us in our quest, we have the following oracles for any node A:

- A metric that tells us either (for type I) the rough likelihood of there being any children (p_A), or (for type II) the rough likelihood of this being the final node (P_A); the probability of a successful crack
- The average running time of finding the children if there are some (t_A); the average runtime of a successful crack
- The average running time of finding out that there are no children (T_A); the average running time of an unsuccessful crack

If we let ω_A be the time taken to run if A is on the fastest route, Ω_A be the time taken if it isn't, and τ_A be the time taken to examine the rest of the tree. We can arrive at a rather simplistic optimal weight system right away:

$$W(A) = p_A \cdot (t_A + (1 - P_A)(\omega_A + \tau_A)) + (1 - p_A) \cdot (T_A + \tau_A)$$

Needless to say, τ_A is uncomputable without evaluating every node, as a trivial result of the undecidability of the Halting Problem. ω_A and Ω_A have similar issues.

In this paper, we set about trying to find a replacement heuristic for τ_A, as well as trying to approximate ω_A and Ω_A.

We will use ξ to represent the root node.

2 Modelling

We assume that there is no overhead in getting these stats. This is again untrue, but if we require that any non-trivial work is done in the actual gathering of children, at the expense of weakening the heuristic.

We assume that type I nodes will have a probability of being the solution P high enough that we can treat it as 1 in the weight calculation. However, this is tweakable within the algorithm, so I will just leave P as-is.

We also assume that there are no identical nodes. This is achieved in Ciphey by filtering out reoccurring values, which causes reality to move away from the heuristic a small bit.

We can assume that a successful crack means that we are on the right path.

3 Proposed Solution

Instead of storing the results as the intuitive tree data structure, it makes more sense to think about it as a list, as we do not care what the parent of a node is!

We let Info(A : Node) be the function that gives us the information about the node, and Evaluate(A : Node) be the function that gives us the result of procesing.

We can gather ω_ξ and Ω_ξ through sampling for each type of data, and use them as an approximation for any ω_A and Ω_A that we may encounter of that type.

Algorithm 1 (FindBestOrder(S : List[Node)**.**)]
We need to find the node A which has the lowest value of:

$$W(A) = p_A \cdot t_A + (1 - p_A) \cdot (T_A + W(N))$$

Where N is some a sequence formed from unique elements of $S \setminus \{A\}$, that minimises $W(A)$:

$$W(N) = \begin{cases} 0 & \text{if } |n| = 0 \\ p_{N_0} t_{N_0} + (1 - p_{N_0})(T_{N_0} + W(N')) & \text{otherwise} \end{cases}$$

RETURN $< A, N >$

Algorithm 2 (CipheySearch($CText$: data))**.**

1. Let the set $L =$ Evaluate(<Info($CText$), [ ] >
2. If $CText$ was the solution, return it.
3. Create our node pointer A
4. WHILE $|L| \neq 0$
   (a) Expand all encodings to their fullest depth
   (b) If we have been running for more than ℓ_t, ERROR "Timeout".
   (c) $< A, N >:=$ CALL FindBestOrder(L)
   (d) Derive p_N from $P_{[]} = 0$, $p_N := p_{N_0} + (1 - p_{N_0})p_{N'}$
   (e) If $p_N := p_{N_0} + (1 - p_{N_0})p_{N'}$ is less than ℓ_p, ERROR "Unlikely".
   (f) Remove A from L
   (g) $S :=$ Evaluate(A)
   (h) If it is the solution, RETURN A
   (i) If it has no children, CONTINUE
   (j) $L := L \cup$ Info(S)
5. ERROR "List exhausted"

4 The Algorithm Implementation

4.1 Assumptions

- We know exactly what is going into the lists. Because the lists will only container crackers and decryptors
- We know exactly how many items will be in the list, because we know what is going into it
- the only thing we do not know is the specific ordering of the list based on the weight function
- We are implementing this as a regular Python list
- Nothing we do not know will be added to the list. I.E. The list is deterministic as a multi-set. It will only contain items we know, and will always be a specific length or an easily calculable length.

4.2 Data Structure Ideas

We can use a Complete Binary Tree. This data structure has O(log n) search and O(log n) pop / deletion. And because it is a **complete** binary tree, it can be implemented as a regular Python list.

More specifically, we will implement this tree as a Heap. A heap is a data structure that uses a CBT (Complete Binary Tree) as its backend. Most importantly, the first first element [0] will always be the smallest element.

Here are some more facts about heaps:

- The first element will always be the smallest
- The value of a node is always smaller than its children
- For every node, its first child is at $2 * k + 1$, its second child is at $2 * k + 2$, and its parent is at $floor((k + 1)/2)$.
- If H is a heap, then the following will never be false $h[k] <= h[2*k+1] and h[k] <= h[2*k+2]$.
- HeapReplace is equivalent to heapPop and heapPush.
- HeapPushPop is equivalent to heapPush followed by heapPop
- The Python heap module implements merge() which is used for merging sorted sequences into the heap.

4.3 Ideas

Each decryptor / cracker will return their values as heaps.

Since we have a families "encoding, basicEncryption, hashes" we perform Merge on these values. Since our assumption is that we know **exactly**, what each function is returning, and they are deterministic we can simply do insertion sort — but without the O(n) check to see if it is sorted. We do this check on non-deterministic inputs. But since we know the exact contents of the return, we do not need to perform this check. We simply insert.

When we merge, we also know the exact contents of the input to the merge. This means that again, we do not need to stress too much about how to effectively merge 2 things if we already know what the functions are. All we have to do is make sure we maintain the heap property.

At the end, we will have a merged list which follows the heap property.

So the idea is that instead of taking the time to turn the list into a heap everytime, we simply build the heap as we take the inputs from each node. And since we already know what the heap looked like before, and we know everything apart from the minimise value, we can use some clever CompSci to minimise the amount of time needed to Heapify the list.

Once we have a heap, naturally the first element will always be the lowest.

But this is assuming we need to evaluate the entire list every time.

I am thinking that the weight changes will not be significant. I.E. 80 cannot become 2. The weight changes will most likely be smaller than that change.

My thinking is this: * We have a list of weights, ω_A * We perform the crackers / dectyptors, and store the result in A * We now have list of weights ω_B. Our list now looks like $[\omega_A, \omega_B]$. **Path 1**. Normally we would recalculate every single weight, giving us an exponential growth rate. **Path 2**. Assuming that the weights do not massively change, we only re-calculate the weights we select. * For this algorithm, choose part 2. So we store all the new weights, and we hasify the list with both the old weights, and the new weights. If the element at index 0 is an old weight, we re-calculate it. If it is still the lowest, we use it. If not, we go to the 2nd lowest.

So the most expensive part is the maths equation, but finding the minimum is really cheap.

We will also delete trees

4.3.1 Issues with the above algorithm

* It may be possible for an old weight at 3 to be updated to the new lowest weight, which means the algorithm isn't optimal. * If the weights change every time, and they are relient on the previous weight, we will have to expend space complexity to store all weights until we update them.

4.3.2 Harlan's idea

It's a optimisation trick Basically, I calculate the time taken in a vacuum for an evaluation I. E. The expected time for that node Then I sort in ascending order, as the contribution of each one decreases as you go up the list However, I'm pretty sure that's not going to give accurate results Only non-insane ones Usually