Inverted Code Theory: Manipulating Program Entropy

usH nalA, eiX xelA

91 hcraM, 1202

1 Abstract

We watched *TENET* and we were very confused and inspired by it. So, we just wanted to spread some of that confusion and inspiration. We also solved P = inverted NP by walking through a turnstile.

2 Preface: The Inversion of Entropy

We live in a twilight world. As a human race, we have collectively made many scientific discoveries and technological advancements in the past decade, from the creation of vegetarian beef to the COVID-19 vaccine (and maybe also COVID-19? Depends on who you ask.), and grown from both a humanitarian and a practical standpoint. Out of all these amazing creations, it is clear that Christopher Nolan's discovery of how to reverse the entropy of objects has been the most impactful in the computer science field overall. First seen in his historically accurate autobiographical film *TENET*, the reversal of entropy through a turnstile allows objects to move backwards in time, which proves to be useful in many applications of computational science. In this paper, we explore such applications after developing the Inverted Code Theory. Before reading this paper where we reveal astounding results from reversing the entropy of code, we suggest you to carefully watch *TENET* at least $\lfloor \sqrt{\pi} + e^{\log_2(your\ age)} \rfloor$ times to have a solid understanding about inverted entropy.

You will also need access to a turnstile, which will be able to invert the entropy of any object that goes through it (See more details here: `https://en.wikipedia.org/wiki/Turnstile`). Retail stores such as Home Depot, Best Buy, and even Walmart should have plenty in stock. Batteries not included (If they were, they would run out of juice by the time you bought it).

3 Inverted Code Theory (ICT)

3.1 Introduction

So what exactly constitutes as inverted code? Well, inverted code is defined as code written by an inverted person viewed from the perspective of an uninverted person; in particular, inverted code has reversed entropy, and is running *backwards* in time. Note purely inverted code isn't really useful: you will just see your programs run backwards to its initial state, which is quite pointless. However, programs with partially inverted code turn out to be extremely powerful and break new frontiers in computer science applications.

3.2 Temporal Sandwiching: How to Construct Stable Inverted Code

While we were able to construct inverted code with many techniques, the most straightforward method is via *Temporal Sandwiching*, or constructing a program such that the beginning and end are non-inverted, while a few lines in the middle are inverted. Consider a simple binary search:

```
1  def invertedBinarySearch(elem, L):
2      lo = 0
3      hi = len(L)-1
4      #begin inverted code
5      while lo < hi:
6          mid = int((lo+hi)/2)
7          if elem > mid:
8              lo = mid + 1
9          elif elem < mid:
10             hi = mid - 1
11         else:
12             return mid
13     #end inverted code
14     return None
```

While this may look like a regular piece of code, it is not. Lines 5 to 12 are actually inverted, while lines 1-4 and 13-14 are uninverted. To better understand how sandwiching occurs, let's walk through the construction of this piece of code:

1. First, we type out lines 1-4 uninverted. Now, if we were to step through a turnstile and invert both ourselves and the machine we are using, the code will begin to *untype* itself from the end of line 4, which is an undesirable effect. Hence, we will need to pad it with an extra redundant line such that after inverting it the redundant line will untype itself and we won't lose any of our important code.

2. Now, we pad our code with an extra redundant line and then we step into the turnstile with our computer and invert ourselves.

3. Right when the redundant line finished untyping itself (remember, we are now traveling back in time and so the code we wrote is disappearing), and before line 4 begins to untype, begin typing lines 5-12 of the binary search code. The cursor on the computer can only move in

1 direction, as it is a well-defined set of pixels on your screen. Since the will of a human is stronger than that of a computer, you can actually reverse the direction of the cursor and begin typing code in your inverted form.

4. Once you finish typing line 12, again pad your code with an extra line and uninvert yourself and the machine back to normal entropy.

5. As the redundant line finishes untyping itself (remember the redundant line was inverted: it was created backwards in time from an uninverted perspective, so it will untype in an uninverted state), type the final two lines. We now have a stable inverted code between two uninverted code, hence the term *Temporal Sandwiching*.

You may wonder why the inverted code in the middle doesn't just disappear. Well, recall that the code is deleted as the cursor moves backward in time. Since the cursor is at the end of the program, and the final two lines are moving forward in time, the cursor has no choice but to move forward in time as well. Remember, pixels on the monitor are deterministic and well-defined; we cannot have the cursor simultaneously exist at two locations on the screen.

3.3 Time Complexity of Inverted Code

We continue our inquiry on inverted code by analyzing the big-O time complexity of our inverted binary search. A regular binary search runs in $O(\log(N))$, where N is the length of the list we search, but we are pretty curious about the computational time of our inverted binary search. Below we show our experimental data of the time it takes to run a regular binary search versus the inverted binary search on the same dataset:

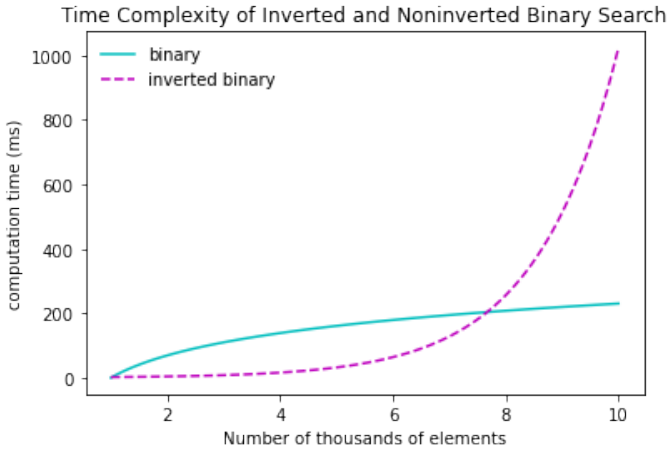

Figure 1: Experimental results of inverted binary search

Not surprisingly, the inverted binary search behaves differently from regular binary search. From our experimental data, it is clear that regular binary search is logarithmic, but unfortunately inverted binary search is exponential, so we cannot speed up the computational complexity of binary search just by inverting the while loop. However, there is indeed another method we can apply, namely Inverted Back-propagation, to actually make binary search constant time. We explain this in the application section later.

However, this result is quite astounding, because it turns out that exponential growth is the functional inverse of logarithmic growth, which brings us to the natural question: how will regular exponentially growing functions behave with inverted code? Consider the famous subset sum question, where we search whether a subset of a list of values sum up to particular value; this algorithm runs in exponential time, but what would happen if we invert all looping structures in the algorithm through Temporal Sandwiching? Below is our experimental result:

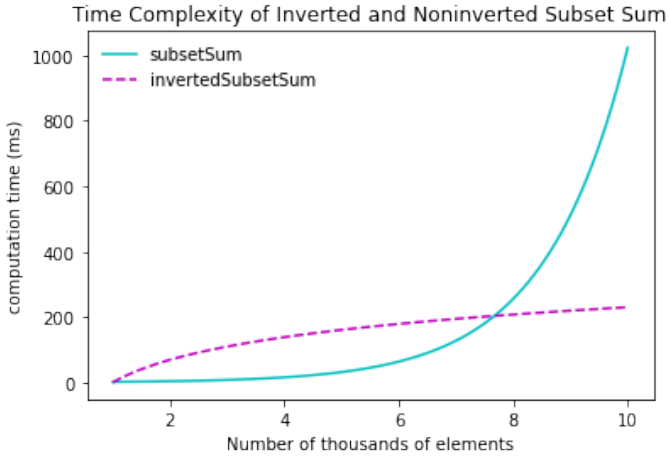

Figure 2: Experimental results of inverted subset sum

The implications of our experimental results are practically unbelievable. With an inverted subset sum, we turned an exponential algorithm into a logarithmic one, and so we drastically reduced the computation time! Just to be sure, we conducted a few more tests on some famous exponential algorithms, and the results were fascinating:

1. A* Graph Search with Manhattan Distance as a Heuristic

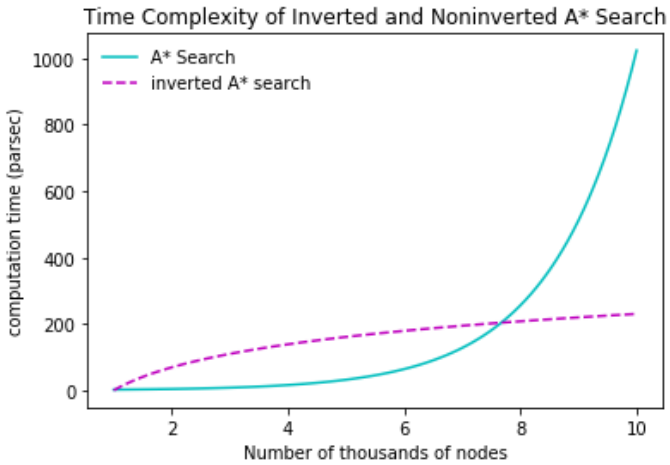

Figure 3: Experimental results of A* (Manhattan)

2. Traveling Salesman Problem

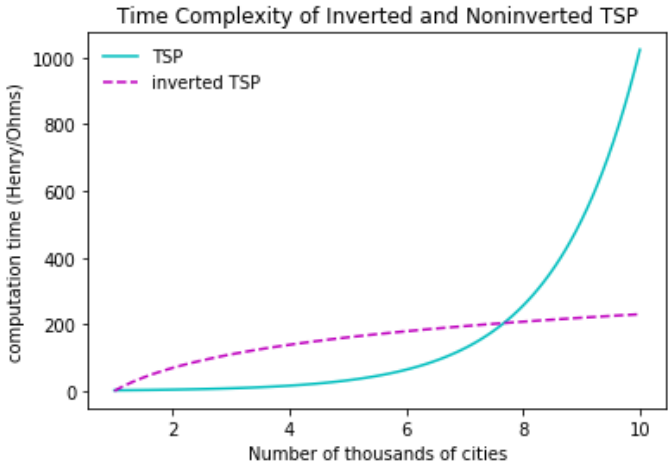

Figure 4: Experimental results of the Traveling Salesman Problem

As you can see, we have shown that exponential algorithms become logarithmic ones after inverting all of their loop structures!

Curious about how this works? We are too. But it's a cool result isn't it? And isn't what *really* matters? Like Mr. Nolan, we leave the proof as an exercise for the audience to figure out.

But, we can at least provide a theorem that we found:

Theorem 3.1. (Big-O Inversion). *Let Q be a chunk of code with all its loop structures inverted (i.e. they were typed by an inverted person) with an uninverted runtime complexity $O(f(n))$. Then the inverted runtime complexity $\widetilde{O}(f(n))$ is given by*

$$\widetilde{O}(f(n)) = O(f^{-1}(n)). \tag{1}$$

Remark. So with this functional inverse you may think that inverting constant time $O(1)$ code will result in code that runs infinitely. However, note that we cannot invert constant time code if it has no loops (the manual inversion by turnstiles can only be done with code that has loops!) So, our theorem will not apply to constant time code, and all is good.

But that would mean...

Corollary 3.2. (Solution to P = Inverted NP). *Consider an normal piece of code with exponential runtime (NP). Then, we can simply invert the loops in the code in a turnstile and get a polynomial runtime (P) version of the original code. Technically this is now a Temporally Sandwiched version of our original code, but it will provides a way to solve NP-hard problems in polynomial time. Hence P = Inverted NP.*

3.4 Inverted Race Conditions

Concurrent and parallel conventional code often falls prey to a set of pernicious errors known as race conditions. These errors, sadly, may also plague inverted code. Consider the following chunk of code:

```
A = [1,2,...,1000]

for i=1:|A|
    A[i] = 1

#begin inverted code
for j=1:|A|
    A[j] = 0
#end inverted code
```

We may conceptualize the execution of this code as two "threads" executing in parallel, one forward from the start of the process and another executing backward from its termination. Hence, these two threads will meet precisely in the middle, simultaneously updating A[50].

The question then arises: which thread of execution wins out? Again, we follow the footsteps of Professor Nolan and leave the proof for the audience.

4 Applications of ICT

4.1 Inverted Back-propagation and Feedback Loops

So far, we have two big conclusions:

1. We can invert exponential code to make it logarithmic
2. We should not invert binary code as that will make it exponential

But, what if we can do better? It turns out that we can use inversion to speed up binary search as well, with a technique called inverted back-propagation.

Typically, in control theory we can feedback our final result into the beginning of the program to make adjustments or to steer our program into a desired direction. Now, what if we perform this feedback via a constant back-propagation in *time* throughout our program instead of at the end?

Consider a typical binary search: in one ply, we must check the midpoint and see if it is greater or less than the object we wish to find, and shift either the left boundary or right search boundary to the midpoint. It takes $O(log(N))$ iterations to complete. Now, what if after the first iteration in binary search, we invert our code and send back in time the direction that we chose to search (either left or right of the midpoint)? Then, if we uninvert this code once it reaches the beginning of the program, it will contain the information about which way (either left or right) that we will choose to go. So, the computational time is now one less iteration than the original program.

But, if we do this after each iteration of the loop, then we will reach a state where at the beginning of the program we already know where each ply of binary will choose to go. So, there will be no need to go through the checks in each iteration at all, and our binary search will be $O(1)$! Below is an outline (loop code omitted for clarity) of what the heck inverted back-propagation in binary search code will look like:

```
1  import TENET.invertedBackPropagation as ibp
2
3  def binSearch(elem, L):
4      lo = 0
5      hi = len(L)-1
6      (hi,lo) = ibp.__uninvert__(binSearch)
7      while lo < hi:
8          #...
9          #binary search loop code
10         #...
11         ibp.__invert__(binSearch, vars = (hi, lo))
12     return None
```

This is the first demostration of the `TENET` package that we developed for Python version 3.8 and above. First note that we are not using Temporal Sandwiching here: all of the code here is uninverted. We also provide the documentation of the package functions used:

1. `ibp.__invert__(f, vars)` : This function takes in a function f, and inverts f if it is currently running forward in time. The `vars` argument will relay the values of current variables back in time.
2. `ibp.__uninvert__(f)` : This function takes in a function f, and uninverts f if it is currently running back in time. It returns the variable values that were pass in when the code was inverted.

Now, let code trace through a few iterations to see what's going on:

1. We run `binSearch()`, and the while loop begins (line 6 is ignored since the program is running forward in time)
2. After one iteration of the loop, line 11 will invert the code and relay the new values of the boundaries `hi, lo` back. From here, the code will diverge in time: one version will travel *back in time*, while the other version will continue *forward in time* into the second iteration of the loop.
3. The version that travels back in time will go line by line from line 11 back to line 6, where it is uninverted and the values of `hi, lo` are updated. Now, the code travels forward in time again with new values for lo and hi!
4. The version that travels forward in time will enter in the second iteration of the loop, where it will hit line 11 again and then invert itself to travel back to the beginning of the program with an even better update of `hi, lo`.
5. Eventually, there will exist a version of the code such that it will only take 1 iteration to find the element or return `None`, and so in that version the binary search code will be constant time!

For those who are visual learners, here's a diagram representing the flow of the binary search program:

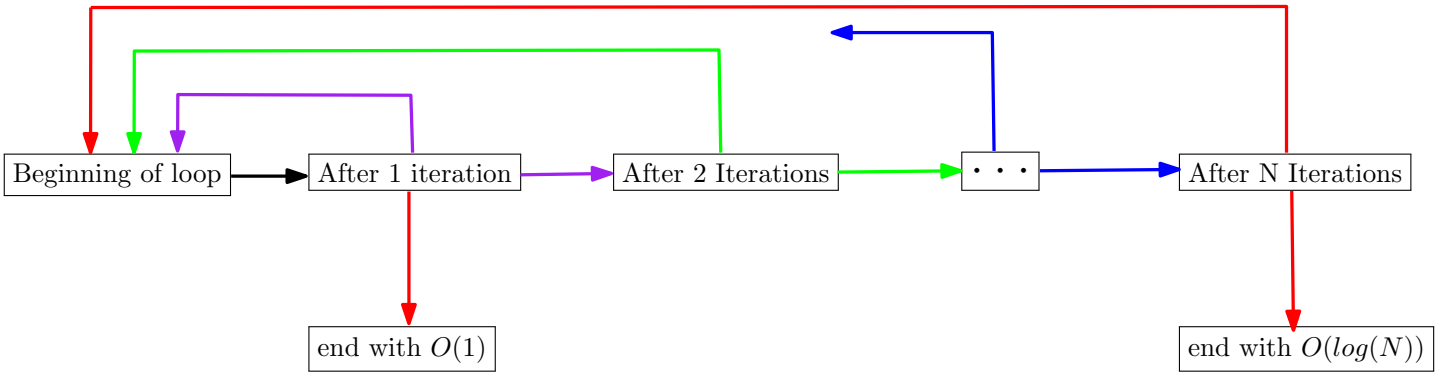

Figure 5: Visual of inverted back-propagating binary search

The similar colored arrows represent the same program branching off after each iteration, where one version of the program continues forward in time while the other relays information back in time through inversion.

In fact, we have an important theorem that generalizes the experimental findings above:

Theorem 4.1. (The Fundamental Theorem of Inverted Code Theory). *Given any piece of code with runtime $O(f(n)) > O(1)$, there exists a finite amount of locations to invert and uninvert the code such that the code subsequently runs in $O(1)$.*

The proof, of course, is trivial, and left as an exercise for the audience to figure out.

4.2 Qubit turnstiles

So we have displayed a method to invert code in time and uninvert it at different locations to effectively propagate information about the values of particular variable. But there is still one glaring question that we have not addressed: how does the program exactly invert itself to go back in time? The program cannot autonomously step into a turnstile, and even if it could, it would be quite inefficient. Luckily for us, the future versions of us sent us an *Intel j9 Core Processor*, where $j = \sqrt{-1}$, that is a quantum processing unit. We developed the `TENET` Python package using this processor, and you will need it to run inverted code.

We won't go into the details of how this processor works, but essentially it contains trillions of quantum qubits that have two states: spin-up (normal) and spin-down (inverted). Each qubit acts as a turnstile, as normal code will be run with the spin-up state, but any calls from the `TENET` package to invert code will make a spin-down state qubit run that part of the code. In other words, the spin-down inverted state will be able to run the program backwards (until the program is uninverted again). We have just sent our first shipment of *j9 Core Processors* to the CMU bookstore, so you should be able to purchase one for yourself so that you can run inverted code as well!

4.3 Temporal Pincer Reinforcement Learning

The `TENET` package also provides a reinforcement learning environement, namely one that utilizes Temporal Pincers. As Christopher Nolan pointed out in the film, Temporal Pincers are critical to the success of inverted operations. For those who need a refresher, a Temporal Pincer Movement is an operation where two identical teams, one traveling forward in time and the other traveling back in time inverted, constantly relay information to each other until they meet at a midpoint in time. This allows the forward-moving team to obtain information about the future.

The package `TENET.temporalPincerRL` provides many useful functions to train RL programs in constant time. Similar to backpropagation, the currently training robot going forward in time will learn from the *already trained* version of itself from the future going back in time to become the very best like no other robot ever was. You will need the qubit turnstiles in the *j9 Core Processor* to do so.

5 Conclusion

So what have we accomplished? Oh, nothing much, just

1. Watched *TENET* a few **time**
2. Solved $P = NP$ by going back in **time**
3. Showed that all code can be converted into constant run**time**
4. Had a fun **time**
5. Manipulated entropy many **time**
6. Manipulated space**time**
7. **time**

References

[1] Nolan, Chrisopher. (`https://www.imdb.com/name/nm0634240/`)

[2] What is a turnstile (`https://en.wikipedia.org/wiki/Turnstile`)

[3] TENET Summary (`https://justpaste.it/1zjef`)

[4] How does a turnstile work (`https://www.reddit.com/r/tenet/comments/exn7n7/infographic_on_how_the_machine_works_and_why/`)

[5] What is a Temporal Pincer (`https://www.cbr.com/tenet-temporal-pincer-movements-explained/`)