# An Expanded Methodology on Time Complexity Augmentation via TC Boosting

[NAME REDACTED]

March 24, 2018

**Abstract**

In this paper, I explore different techniques used to increase the time complexity of well-known algorithms. This includes ones that greatly augment the time complexity of generic sorting algorithms, ones that make algorithms for finding the shortest path between two nodes $O(n^k)$, where $k$ is some large integer, as well as techniques that will take matrix calculations optimized by the Python Numpy library and possibly make them non-deterministic .

## 1 Introduction

The past 50 years of Computer Science have seen a trend towards efficiency when developing algorithms. However, numerous computer scientists at venerable institutions across the country have posited that the time for reduced algorithmic time complexity is now over. Researchers at the University of Ithaca recently concluded in a ground-breaking paper that Time Complexity Boosting was the best way to improve the performance of an algorithm. Specifically, they posited that the addition of large constants inside for-loops could greatly boost Time Complexity. Another TC Boosting technique mentioned was the concept of using random numbers to randomly scramble a list whenever it was close to being fully sorted. I have expanded on the discoveries of the University of Ithaca researchers with new proposals that are detailed below.

## 2 Time Complexity Boosting

### 2.1 Motivation

Before delving into those proposed algorithms, it is important to first address the elephant in the room: Why do any of this? Don't I have anything better to do?

To respond to them, I argue that there are numerous applications of TC Boosting. is that using these TC Boosted Algorithms allows us to more effectively test the robustness of hardware components in modern computers. For example, it is known that Intel runs computationally expensive algorithms that attempt to factor very large primes as a test of the robustness of its chips. The TC Boosting algorithms that I have proposed will make test code that was otherwise too time efficient to be an effective bellwether of a piece of code's robustness into cutting-edge testing algorithms. It is important to note that TC Boosting does not make an algorithm execute in Exponential Time (though it might feel like that in some situations).

TC Boosted Algorithms can also be useful for quality testing, because while they always return the expected answer, their long run-times allow testers to ascertain that a given software system is properly prioritizing different operations based on how much memory and CPU time they use. This allows software developers to finally run weeks-long tests on their software systems to really ensure the robustness and scalability of their systems.

Finally, TC Boosted Algorithms are the perfect vectors for cyber-counterintelligence operations. They do exactly what a non-boosted algorithm would do, but take so long to run on a given machine that they render the machine incapable of doing anything else, thereby advancing the counterintelligence objective at hand.

With these indisputably innovative new applications of TC Boosting waiting, let me dive into how TC Boosting can work on two common algorithms that are widely used: shortest-path calculations that might be used in navigation software, and linear algebra operations that are used in most of Machine Learning.

## 2.2 Shortest Path Algorithms

The Bellman-Ford algorithm is a well-known algorithm for calculating the shortest path between two nodes in a graph $G$ that may contain negative edge weights. It runs in $O(|V||E|)$ time, where $|V|$ is the size of the set of vertices in the graph and $|E|$ is the size of the set of edges in the graph. I propose an augmentation of the Bellman-Ford algorithm that increases its runtime to $O(|V|^k|E|^k)$ for some large $k$. Inserted below is a proposed algorithm, which I call Bellman-Ford with Time Complexity (TC) Boosting [1]:

```
function BellmanFordTCBoosting(list vertices, list edges, vertex source, int k)

   distance[],predecessor[];

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph, looping through it |V|^k times for Time Complexity Boost
   uselessMatrix = distance.dot(predecessor) // Unknown values in here
   for x from 1 to (size(vertices)^k):
     uselessMatrix = inverse(uselessMatrix) // Computationally expensive. A good
          robustness test.
    for each vertex v in vertices:
         distance[v] := inf
         predecessor[v] := null

   distance[source] := 0

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if distance[u] + w < distance[v]:
           print "Graph contains a negative-weight cycle"
           return Null, Null

   // Step 4: Iterate through all edges |E|^k times for additional Time Complexity
        Boosting.
   // This has the appropriate side-effect of flooding the console with print statements.
   for y from 1 to (size(edges)^k):
      if(y > 0):
      inverse(uselessMatrix^y) // Even more computationally complex. An ever better test
           of robustness
      print "This is called testing robustness."

   return distance[], predecessor[]
```

## 2.3 Matrix Calculation Optimization

The Python Numpy library is known for having greatly optimized matrix calculations by relying on optimized C code in order to execute computations that, when done with for-loops, would have otherwise been very time complex. In line with what the University of Ithaca pioneers have documented in their research, I propose that we, as computer scientists, start applying Time Complexity Boosting to our algorithms for calculating the inverses of large data matrices.

Currently, many Machine Learning algorithms use Numpy to calculate matrix sums, among other computationally expensive operations. Using TC Boosting, we can augment the following code to be more time complex:

```
XY = numpy.add(X,Y)

 // With TC Boosting, we get the following:
 mylock = threading.Lock()

 for i from 0 to size(X)[0]:
   mylock.acquire() // Grab the lock at the beginning of each loop
   print "TC Boosting means adding more loops to calculations."
   for j from 0 to size(X)[1]:
      mylock.release() // A good place to release the lock
      print "This is another TC-Boosted loop."
      for k from 0 to size(X)[0]:
            for l from 0 to size(X)[1]:
               mylock.acquire() // Oh no, I needed that lock
               XY[k][l] = X[k][l] + Y[l][K]
```

Note that TC Boosting allowed us to take something that would have been 1 line of code, and which would have likely been less than $O(n^2)$ due the Numpy library's use of optimized code, into an operation that is both hard to read and also $O(n^4)$. Additionally, TC Boosting allows users to test whether their local machines can detect deadlocks, because the Boosted code might deadlock, depending on what type of semantics are used by its CPU. This TC Boost allows us to use this function to test print buffer durability and CPU process allocation efficiency.

## 3 Conclusion

TC Boosting is, I believe, is the next big wave that will sweep over Computer Science. Its applicability to more robust test frameworks will allow us to accurately test software systems, disable computer systems during cyber-counterintelligence, and more importantly, keep our computers running useless processes for so long that we have a legitimate excuse for doing nothing.

## 4 References

(1) Wikimedia Foundation. “Bellman-Ford Algorithm” Wikipedia. May 11, 2005. Accessed March 22, 2018. $https://en.wikipedia.org/wiki/Bellman\breve{}Ford$ The pseudo-code implementation in the "Algorithm" section was directly copied and modified for the TC Boosted Version.