TOP for Python[*]

Brian Clee
bpclee@ncsu.edu

Andrew McNamara
ajmcnama@ncsu.edu

Esha Sharma
esharma2@ncsu.edu

Abstract

Computing distances between sets of data points can be a non-trivial computational problem. Many distance-related problems exist across domains like data analytics and graph analysis, with these problems distance calculations become a computational issue, and as such manual optimizations to these problems have been studied extensively. In this work we build off the work done by Ding et al., who proposed a general abstract framework for improving distance-related problems by relying on the triangle inequality for distance calculations. In Ding et al.'s original work they found speedup improvements from general implementations up to $276x$ times and an average improvement of $2.5x$. In this paper we present an extension of Ding et al.'s original work which implements their triangle inequality framework in Python for three specific distance-related problems, the results of which agree with Ding et al.'s original findings.

Keywords Information systems applications, compilers, distance-related problems, triangle inequality

1 Introduction

A common problem in fields like data analytics and graph analysis is computing distances between sets of data points. In these fields much work has been done on manually tuning and optimizing their distance algorithms. Frequently, these optimizations involve using the triangle inequality for performance gains [6] [11] [12].

In this work we implement the framework *Triangular Optimizer* (TOP) [3], for generalizing solutions to these distance calculation problems, making use of the triangle inequality to improve on slow distance calculations.

By abstracting these problems and creating a general optimization solution each time a new distance calculation is used it does not have to be optimized to the specific problem. Furthermore, optimizing such a framework can lead to drastic performance gains across domains.

Though distance calculation problems differ in distance definitions and calculations, past work has shown that solutions to optimize these problems are very similar [3]. Hence, it is possible to have a generalized framework to optimize each of these problems. Each of the distance related problems have to be generalized before such a framework can be applied to them.

The TOP framework proposed by Ding et al. [3] provides APIs which abstract distance related problems to a general form and provides solutions to optimize the calculations for these problems. Such a framework provides a more efficient way to optimize all distance related problem which follow triangle inequality conditions and also provides a easier way to optimize new distance related problems.

In our own work presented here, we attempted to recreate the work done by Ding et al. and provide a TOP framework for Python, rather than in the original framework's C++. We further defined and implemented three example distance functions: point to point (P2P), K nearest neighbors (KNN), and K-Means clustering. For each of these distance related problems we programmed them in two versions, the original standard implementation using "normal" distance calculations, and a version which makes calls to our TOP API in order to perform distance calculations using the triangle inequality. Finally, in this work we present the comparison results between the original standard implementations of these popular distance related functions, and the improved TOP versions of these functions.

Our findings agree with prior work done by Ding et al. and support the use of the triangle inequality for performance gains across myriad problem domains.

For the rest of this paper we discuss related works in Section 2, and challenges encountered during our implementation in Section 3. We further discuss our results in Section 4 and our remaining issues and our lessons and reflections in Section 5. Finally we conclude in Section 6.

2 Related Works

There has been much prior work in problem-specific triangle inequality optimizations. For instance, much work has been done on optimizing the distance calculations for K-Means [4] [6] [9], as it is one of the more popular data mining approaches. Likewise much work has been done on optimizing distance calculations for other problems like KNN [7] [8] [11] [12], KNNjoin [1] [13], Nbody [5], P2P [2], and ICP [10].

The abundance of these problem-specific approaches help to motivate a more general abstract solution. Such a solution was proposed by Ding et al. [3], and is the primary influence

[*]This project is submitted in fufillment of the final course project requirements for CSC 512 taught in the Fall 2017 by Dr. Shen at NC State University.

for this work. Furthermore, since our own work is so highly motivated and extended from Ding et al., a more detailed analysis of their work is thus presented here.

Ding et al.'s specific contributions are the development of a TOP application programming interface (API) and corresponding compiler which allows for programmers to write general distance related functions. When these functions are being compiled by the TOP compiler any reference to the TOP API is resolved and the compiler handles optimizing problem specific distance optimizations using the triangle inequality.

Additionally, Ding et al.'s explicit contributions are:

- *Abstraction:* foundational work is presented on the development of automatic optimizing frameworks by providing an abstraction which unifies a broad range of distance-related problems.
- *Algorithmic Optimization:* they offer seven principles on enabling effective distance related optimizations and break algorithmic optimizations into two design questions of landmark selection and comparison ordering.
- *TOP Framework:* the TOP framework is presented which which offers automatic algorithmic optimizations on a broad range of distance-related problems.
- *Results:* performance of standard distance-related problem implementations is compared with the automatic TOP framework's implementation of the same distance problems. They found that the TOP framework can achieve up to 237x speedups, and more realistically 2.5x on average.

To further discuss Ding et al.'s results, in their work they defined and implemented six different distance-related functions: KNN, KNNjoin, K-Means, ICP, P2P, and NBody. For each of these different problems the authors implemented them using their standard approaches along with implementations using the TOP framework. From here, they were able to compare performance between the original implementation and the TOP implementations, an overview of these performance comparisons is shown below in Table 1.

Table 1. Results from Ding et al.'s TOP framework implementation

Distance Problem	**Average Speedups (X times)**	**Average Comps Saved (%)**
KNN	17.0	93.0
KNNjoin	77.1	95.6
K-Means	27.6	92.9
ICP	193.4	99.6
P2P	9.8	93.2
Nbody	237.6	99.4

As seen in Table 1 Ding et al. were able to achieve tremendous speed improvements from their TOP implementations, with the max speedup being 237.6, and the min speedup being 9.8. Additionally, in terms of computations saved by percentage, all six distance problem implementations in TOP had at least 90% average computations saved, with a minimum of 92.0% and a maximum of 99.6% saved.

3 Implementation and Challenges

For our own implementation of Ding et al.'s aforementioned work, we ran into a number of challenges which ultimately limited our implementations. In this section we discuss the challenges encountered, and how they affected our final implementation, along with a detailed explanation of our implementation.

To begin, we originally chose this project because our entire team had experience with Python, and the problem described seemed interesting and non-trivial. We were made aware at project selection that this project was one of the more open-ended projects available; however, we had yet to realize how much this would impact our implementation. In the end, this open-ended nature of this specific project made implementation incredibly difficult, as we often felt like we had a lack of clearly defined project objectives and requirements.

Due to this uncertainty it took us most of the semester just to understand our project goals and problem requirements, and by the time we had a firm grasp on project requirements it was far too late to completely replicate the work done by Ding et al. in their original work. As such, by the second project deliverable we began to re-scope our own personal goals for the project and drastically reduced our overall scope.

While originally we had intended to fully implement the work done by Ding et al. on six distance-related problems, due to the above mentioned challenges we eventually reduced our scope to only include three distance-related problems. Further, originally we were going to implement three iterations of each distance problem: the original "standard" implementation, our TOP framework implementation, and a manually tuned and optimized implementation.

This would have better mirrored the work done by Ding et al.; however, due to time constraints and personal domain knowledge gaps we ended up only implementing two iterations of each distance problem: the original "standard" implementation (sourced from existing open source implementations), and our TOP framework implementation. With these two implementations we were still able to make performance comparisons between the normal implementations and our TOP implementations, we discuss these specific results in Section 4.

Aside from problem implementations, we also ran into some challenges in implementing the compiler responsible for automatically converting distance problems with our API calls into our TOP implementations. Since Python is an interpreted language there was not much precedence for implementing a normal compiler, and it certainly wasn't as straight forward as it would have been in a traditional compiled language like C or JAVA.

Because of this we initially struggled understanding how to properly implement a compiler for our Python implementations. Our original plan was to adapt the Python 2 to 3 converter[1] for our own purposes as it was essentially a fully featured Python compiler that would allow us to compile our code down to abstract syntax trees (ASTs) for manipulation and then back to Python code (as opposed to .pyc compiled code). Additionally, we found that some prior work already existed in adapting this Python 2 to 3 compiler[2], and we had planned on basing our own implementation on work such as this.

However, as we struggled with problem understanding we found that this original plan for our compiler implementation was fairly difficult, it did not allow us to easily compile our Python down to AST form. After struggling with adapting the Python 2 to 3 compiler we eventually decided to find another solution and found a tool called Rope[3].

Rope is a Python refactoring tool, and by using it we were able to perform in-line replacements of our Python code. This allowed us to build a Python "compiler" using Rope to replace any and all mentions of our TOP API with the API code directly. Behind the scenes Rope has a robust implementation using ASTs to refactor Python code. This approach allowed us to simulate the act of a compiler replacing our API calls with the API code directly.

4 Results

As mentioned in Section 3 due to time and understanding constraints our original project scope had to be reduced to only include the implementation of three distance-related problems in two forms: the original "standard" implementation, and a version which uses our TOP API and compiler to include triangle inequality optimizations. In this section we discuss the results of these two implementations, covering speedups between the distance-related problems.

In general, our results are in line with Ding et al.'s findings. As Table 2 shows below, for each distance-related problem our TOP implementation saw significant speedups compared to the original implementations which relied on more traditional euclidean distance functions.

Our timings for Table 2 were gathered by running each test case 200 times and taking the average running time across this larger sample. In doing this we ensure that our times are not based on one specific CPU cycle, and instead are more generalized to be in line with what one would expect a normal running condition to be like. Further, all results presented here are taken from the same machine in order to avoid machine specific biases.

Table 2. Average run times for each distance-related test problem, times gathered by running each problem 200 times and taking the average.

Distance Problem	Average Original Time (Seconds)	Average TOP Time (Seconds)
P2P	2.57e-05	1.97e-05
KNN	0.0141	0.00635
K-Means	0.0205	0.0121

Additionally, if we compare the running times further we see in Table 3 that all of our distance-related problems saw significant speedups with our TOP implementation. As Table 3 shows the minimum speedup was 1.30X times for P2P, while the maximum speedup was 2.22X times for KNN.

When comparing these results with the original results from Ding et al. shown in Table 1 we see a smaller factor of speedups across our distance-related problems. The most likely reason for this is because the distance problems examined in Ding et al. were much more complex than our own, and they were tested more rigorously as such. Because of this, while our results do show speedups, because the problem implementations were much simpler, the speedups do not appear as extreme Ding et al.'s results.

Table 3. Average speedups (X) offered by TOP on our specific distance-related problem implementations.

	P2P	KNN	K-Means
TOP	1.30	2.22	1.69

5 Discussions

In this section we will discuss the remaining issues and work that needs to be implemented for our project, along with the lessons and experiences learned throughout the semester's work, and finally the remaining future work for this project.

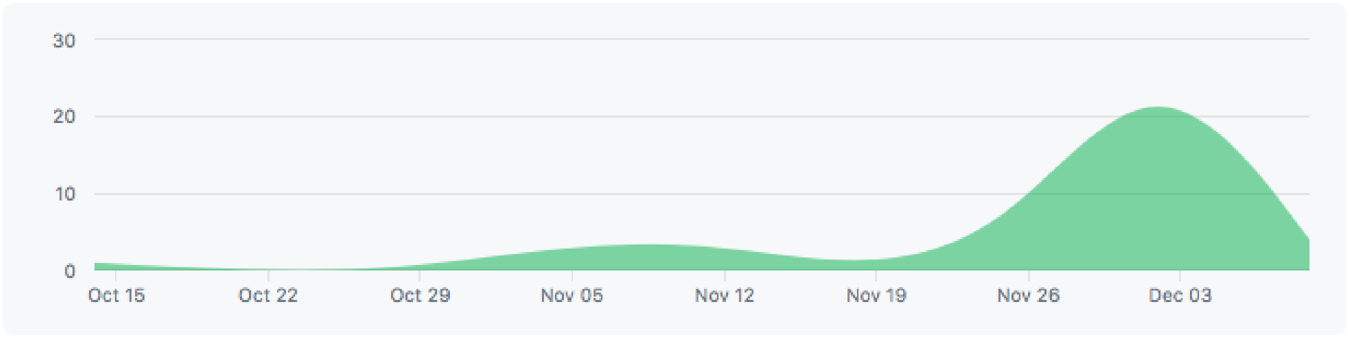
5.1 Remaining Issues

As mentioned earlier in Section 3 we had to re-scope our project goals midway through our timeline. As such, there are a number of features which we will discuss here that still need to be implemented to approach a more thorough replication of Ding et al.'s work.

To begin with, we only implemented three of the distance-related problems that Ding et al. originally implemented. We still need to create implementations of the remaining three distance problems: KNNjoin, ICP, and Nbody. Further, in our existing three implementations for KNN, K-Means, and

[1]https://docs.python.org/2/library/2to3.html
[2]http://python3porting.com/fixers.html
[3]https://github.com/python-rope/rope

Figure 1. Graph of commit history to our GitHub repository over the project's duration.



P2P, we currently only implement them in two ways: the original "standard" implementation, and our TOP framework implementation. To more accurately replicate Ding et al.'s work, we need to further implement the manual optimization forms of each of these distance-related problems for further comparisons with the TOP framework.

Additionally, our TOP API framework needs improvement to become a more generalized framework to be applied to a broader range of distance-related problems. In our current implementation that API is more or less build for our three specific distance-related problems, and it needs more generalization to be a true abstract framework to be used across a larger grouping of distance-related problems.

Finally, additional work could be done on the compiler side. For instance, our compiler could be made more robust if we further leveraged ASTs, as discussed in Section 3. While our current implementation using Rope works, there are certain criterion imposed by the Rope framework in order for our API to work. These criterion add unnecessary complication to our framework, making it less generalized and abstract. For instance, there are specific constraints on where comments can be included, and for each function from the API must be imported individually (i.e. you cannot just import the TOP api module).

One approach to alleviate these issues would be to better understand the Rope framework and find a way to use it without these complications and better leverage the ASTs that it is building. Yet another approach would be to build our own custom Python compiler from the ground up for this specific problem. This was likely what Ding et al. were forced to do; however, in their work they used C++ which lends itself better to building a compiler. By custom building a full compiler for this problem we would be able to incorporate it without the same aforementioned constrictions placed on our framework by Rope. Additionally, we could design our compiler in such a way to better make use of ASTs for problem enhancement.

5.2 Lessons, Experiences, and Reflections

In general we learned a great deal about our specific project during this semester's work, and here we will discuss a list of specific guidelines to better approach similar group work in the future, along with the specific problems which motivated those guidelines.

To begin with, we found from this group project that ***project selection*** is critical. As mentioned in Section 3, all of our group members had prior experience working with Python, which was one of our main reasons for choosing this specific project. However, we quickly found that the open ended nature of this specific project made figuring out specific project requirements and goals incredibly difficult. In the end we had to re-scope quite a bit from our original goals, and ended up spending most of the semester simply trying to understand the project requirements.

Another experience to be learned from is ***time management***. Like many group projects, or projects in general, we suffered from some time management issues which made the final days before the submission deadline quite frantic. While we did spend time each week throughout the semester working together and individually towards the project goals, because we had such a loose grasp on project requirements it became easy to push off work for later. In the end, this resulted in us completing the bulk of our development and implementation in the final weeks of the class. For future projects better time management would help alleviate some of our deadline stress we experienced.

To further illustrate this time management issue, a discussion of our code base is needed. For this project we collaborated using GitHub, and our group repository is publicly available[4] for inspection. As far as group collaboration, investigating the number of commits and additions to the repository shows a pretty even breakdown between group members. Of course, some members committed less frequently than others, but in these cases the commits tended to be larger and more substantial. On the other hand, some members committed more frequently with smaller changes.

[4]https://github.com/arewm/TOP-python

Furthermore, Figure 1 shows a graph of the commit contributions to our master branch (excluding merges). As this graph shows, our time management suffered at the beginning of the project, partially due to external obligations discussed below. This graph illustrates the typical, yet undesired work rush towards the project deadline to finish deliverables in time for submission.

Finally, the last takeaway from our experience with this project deals with ***external obligations***. While working on this project throughout the semester two of our group members also defended their doctoral written preliminary examinations[5]. The addition of these major external obligations made creating time for this group project difficult until after both of the exams had been defended. While not a fault of this class, or the group project members, considering each group member's external obligations at the start of a project could help for better project management throughout the semester. For instance, if we had better communicated when these exams were to take place at the beginning of this group project, we would have been better able to plan around these times when one or both group members would be otherwise unavailable.

In general, our experience with this group project was a favorable one, as each of us has certainly experienced worse examples of group projects. It helped that all three group members were Ph.D. students and were able to devote the time, work, and commitment to this group project when needed. Our major takeaways from this group project experience are to spend more time in the future on project selection, more focus on time management, and an acknowledgment and proper planning around individual external obligations. In the future if these aspects are prioritized from the start of a group project a better final product, and more enjoyable experience will be shared by the group members.

5.3 Future Work

In order to address the limitations expressed in Section 5.1, work needs to continue on both the compiler as well as the test cases.

For the compiler to properly parse the TOP API, function calls are not properly supported and the API file must be contained within its own module. Additionally, the compiler only supports function calls and not method calls. Since the TOP API might be improved by using classes and methods in the API definition, it would be useful to expand functionality past only supporting function calls.

In terms of the test cases used for TOP, future work includes expanding the code to include more difficult problems. Since some of the cases are only solving simple problems, the benefit of TOP over traditional methods are not readily apparent. By developing more complex test cases, it would be useful to investigate the limits of improvements that can be made with TOP.

Finally, it would be insightful to compare the speedups compared with TOP as we implemented it with the speedups discovered in Ding et. al.'s implementation. Since the code running time is not comparable, we would have to look at whether similar ranges of speedups can be realized when using Python for development as compared with C++.

Further, in our own comparisons it would be beneficial to also compare the average number of computations being performed by our test cases. In Ding et al.'s original results they compared the average computation % saved from the TOP implementations, a similar metric should be explored for our own implementation. Additionally, in Ding et al.'s work they not only compared their TOP implementations with the original "standard" implementations, but also with manually optimized versions of the distance-related problems. For our own work it would be helpful to get access to these manual optimizations and translate them to Python for further comparisons.

6 Conclusions

For this work we extended and attempted to replicate the work done by Ding et al. in creating a TOP framework for Python. We successfully implemented a TOP API and compiler which transforms three distance-related problems into optimized versions rellying on the triangle inequality for distance calculations. Our results mirror that of Ding et al. and show our TOP implementations have a general speedup and improvement over their original "standard" implementations.

To summarize, in our paper we discussed our problem motivations and description, along with an in depth analysis of related work, focusing on Ding et al.'s motivating work. Additionally, we covered our specific implementation and the challenges we encountered throughout the semester which shaped our implementation. We further discuss the results of our implementations and their similarity to Ding et al.'s findings. Finally we discussed the remaining issues for our project, and how we would go about approaching them. We further discussed our lessons and reflections on our group project experience from this semester, and offer a set of three specific guidelines for a better group project experience in the future.

Ultimately, our experience on this group project was a positive one, not mired by typical group project setbacks like uncooperative group members. Rather, in our experience all group members held up their responsibilities to the group, and all contributed fairly to the project goals.

Specifically, Andrew *(ajmcnama)* was primarily responsible for the compiler research and implementation. Esha *(esharma2)* was responsible for collecting the test cases, while Esha and Brian *(bpclee)* together worked on implementing

[5]https://www.csc.ncsu.edu/academics/graduate/degrees/phd.php

the APIs for the test cases. Additionally, much of the writing of this final report was handled by Brian, while both Andrew and Esha contributed to specific sections of the report, like Section 5.

In the end, all group members were satisfied with their contributions in relation to the rest of the group, and our shared experience working together on this project was generally a favorable one, we learned much about how the triangle inequality can be used to improve distance-related problems.

Further, we learned how intelligent use of an API and a compiler can allow for abstraction of these distance-related problems to gain the speedups offered by the triangle inequality, while at the same time not "reinventing the wheel".

References

[1] Christian Böhm and Florian Krebs. 2004. The k-nearest neighbour join: Turbo charging the KDD process. *Knowledge and Information Systems* 6, 6 (2004), 728–749.

[2] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[3] Yufei Ding, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Top: A framework for enabling algorithmic optimizations for distance-related problems. *Proceedings of the VLDB Endowment* 8, 10 (2015), 1046–1057.

[4] Jonathan Drake and Greg Hamerly. 2012. Accelerated k-means with adaptive distance bounds. In *5th NIPS workshop on optimization for machine learning*. 42–53.

[5] Victor Eijkhout. 2014. *Introduction to High Performance Scientific Computing*. Lulu. com.

[6] Charles Elkan. 2003. Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*. 147–153.

[7] Charles Elkan. 2011. Nearest neighbor classification. *elkan@ cs. ucsd. eduâĂŰ,âĂŰ January* 11 (2011), 3.

[8] Tobias Emrich, Franz Graf, Hans-Peter Kriegel, Matthias Schubert, and Marisa Thoma. 2010. Optimizing All-Nearest-Neighbor Queries with Trigonometric Pruning.. In *SSDBM*. Springer, 501–518.

[9] AM Fahim, AM Salem, F Af Torkey, and MA Ramadan. 2006. An efficient enhanced k-means clustering algorithm. *Journal of Zhejiang University-Science A* 7, 10 (2006), 1626–1633.

[10] Michael Greenspan and Guy Godin. 2001. A nearest neighbor method for efficient ICP. In *3-D Digital Imaging and Modeling, 2001. Proceedings. Third International Conference on*. IEEE, 161–168.

[11] Jim ZC Lai, Yi-Ching Liaw, and Julie Liu. 2007. Fast k-nearest-neighbor search based on projection and triangular inequality. *Pattern Recognition* 40, 2 (2007), 351–359.

[12] Xueyi Wang. 2011. A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE, 1293–1299.

[13] Cui Yu, Bin Cui, Shuguang Wang, and Jianwen Su. 2007. Efficient index-based KNN join processing for high-dimensional data. *Information and Software Technology* 49, 4 (2007), 332–344.