TOP: A Framework for Enabling Algorithmic Optimizations for Distance-Related Problems

Yufei Ding, Xipeng Shen
North Carolina State University
{yding8,xshen5}@ncsu.edu

Madanlal Musuvathi, Todd Mytkowicz
Microsoft Research
{madanm,toddm}@microsoft.com

ABSTRACT

Computing distances among data points is an essential part of many important algorithms in data analytics, graph analysis, and other domains. In each of these domains, developers have spent significant manual effort optimizing algorithms, often through novel applications of the triangle equality, in order to minimize the number of distance computations in the algorithms. In this work, we observe that many algorithms across these domains can be generalized as an instance of a generic distance-related abstraction. Based on this abstraction, we derive seven principles for correctly applying the triangular inequality to optimize distance-related algorithms. Guided by the findings, we develop *Triangular OPtimizer* (TOP), the first software framework that is able to automatically produce optimized algorithms that either matches or outperforms manually designed algorithms for solving distance-related problems. TOP achieves up to 237x speedups and 2.5X on average.

1. INTRODUCTION

Distance calculations are essential to many important algorithms across various disparate domains. For example, the commonly used clustering algorithm K-Means [22] is an iterative algorithm which computes the distances between every data point and each of a set of K cluster centers in order to decide which center is closest to each point. Nbody simulation [8] computes the distances between every particle and all its neighbors in every time step, in order to simulate the interplay of particles and their resulting movements. Other examples include K-Nearest Neighbor (KNN) [13], point-to-point shortest path in graphs [4], 3D image construction [3], and so on. These algorithms play a pivotal role in data analytics, graph analysis, digital imaging, scientific simulations, and many other domains. In each of these algorithms, distance calculations over a large number of data points form the typical performance bottleneck.

Researchers in those domains have devoted decades of efforts to create variations of those algorithms that minimize the number of distance calculations. Creating these efficient variants is often difficult and domain specific—the variants often differ in distance definition (i.e., what defines distance), calculation constraints (i.e., how they are employed), context of usage (i.e., when to use optimized implementations of distance definitions), and other aspects. Thus, these previous efforts are often *problem-specific*. The solutions come from a domain experts' deep analysis of the specific problem and algorithm, and is customized to that problem specifically. Coming up with such a solution usually takes the domain experts deep and nontrivial insights, theoretical analysis, and empirical measurements. This point is empirically backed up by the large number of research papers published in the premium venues in those domains. For instance, in the recent 10 years of top machine learning or data mining conferences, there are more than 20 papers on developing algorithms to optimize distance calculations for K-Means (e.g., [7, 12, 18, 25, 27]).

This work is motivated by an observation that, despite the many differences among those distance-related problems, the underlying mechanisms in which distance calculations have been optimized share commonality across seemingly disparate problem domains. This observation prompts us to examine the algorithm design process from the perspective of programming systems: If we can generalize the various distance-related problems into a single abstract form, we can develop an optimizing framework as a unified solution to such algorithmic optimization problems. As a consequence, such a framework saves the significant manual effort previously required to optimize distance-related algorithms. At the same time, the framework could provide a more systematic treatment to existing distance-related problems than previous manually optimized algorithms, which could lead to even better performance.

The result of our exploration is the *Triangular OPtimizer* (TOP), a compiler and runtime software framework that enables automatic algorithmic optimizations for various distance-based problems. Unlike typical program optimizations that optimize an implementation of an algorithm at an instruction level, the *algorithmic* optimizations that TOP enables change the algorithm used to find out certain relations between data points, thus generating new algorithms that are up to hundreds of times faster than existing ones.

With TOP, users specify the distance problem using a set of high-level and relatively intuitive APIs. TOP then automatically creates an optimized algorithm that minimizes the distance calculations for that problem. We have found that TOP is applicable to many problems involving distance-

Proceedings of the VLDB Endowment, Vol. 8, No. 10
Copyright 2015 VLDB Endowment 2150-8097/15/06.

```
/*
Goal: Cluster points in S into K classes with T containing all cluster centers.
S: a set of query points to cluster.
T: a set of target points (i.e., cluster centers).
N: a set of indices of points. |N|=|S|.
*/
... // declarations
TOP_defDistance(Euclidean); // distance definition
T = init();
changedFlag = 1;
while (changedFlag){
    // find the closest target (a point in T) for each point in S
    N = TOP_findClosestTargets(1, S, T);
    TOP_update(T, &changedFlag, N, S); // T gets updated
}
```

Figure 1: K-Means written in TOP API (detailed in Section 5). Prefix "TOP_" indicates calls to TOP API. They will be replaced with low-level function calls by TOP compiler, making the algorithm automatically avoid unnecessary distance calculations.

based calculations that meet the Triangular Inequality condition (see Section 4.1), regardless of the domain, definition of distances, distance calculation patterns, usage of the distances, and so on. Its generated algorithm matches or outperforms the algorithms manually designed by the domain experts. With TOP, decades of manual effort by domain experts could have been saved; it makes optimizing new distance-based problems much easier, and boost the performance of existing algorithms.

Specifically, we propose a simple abstraction, called *abstract distance-related problem*, to formalize various distance-related algorithms across seemingly disparate domains, in a unified manner. The abstraction allows a systematic examination of all kinds of scenarios related with distance computations, which in turn, leads to a spectrum of algorithmic optimization along with some automatic mechanisms for selecting the best optimization to use. We turn all these findings into a runtime library, the invocations of which in a program would automatically save unnecessary distance calculations for an arbitrary distance-related problem (that meets the triangular inequality condition).

Along with the library, we equip TOP with a set of APIs and a compilation module. Through the API, programmers can easily specify the distance problem, as illustrated in Figure 1. The compiler module then derives important properties of the problem, and inserts necessary calls of the library such that at runtime, unnecessary distance calculations can be effectively detected and avoided.

Our experiments show that TOP is able to produce algorithms that match or beat (by 2.5X on average) the state-of-the-art algorithms that have been designed by domain experts. It is able to generate new algorithms for problems, on which no prior work has applied triangular inequality optimizations, and achieves 237X speedups.

Overall, this work makes the following contributions:

- *Abstraction:* It offers an abstraction that unifies various distance-related problems, which lays the foundation for the development of automatic optimizing frameworks.
- *Algorithmic Optimization:* It develops the first set of principled analysis on how triangular inequality should be applied to a spectrum of distance-related problems, reduces the algorithmic optimizations into two key design questions (landmark selection and comparison ordering), reveals a strand of insights, and crystallizes them into seven principles in enabling effective distance-related optimizations.
- *TOP Framework:* It builds the first software framework that can automatically apply algorithmic optimization on various kinds of distance-related problems.
- *Results:* It shows that the automatic framework can yield algorithms that match or outperform manually designed ones. Some of the algorithms have never been proposed for the distance-related problems by domain experts.

Paper Structure. To develop a unifying software framework for addressing the various distance-related problems, there are three steps:

(1) Come up with an abstraction to represent the various problems in a unified manner;

(2) Based on the unifying abstraction, systematically examine the optimizations that have been manually applied to the various distance-related problems. Through the examination, uncover the essence of distance optimizations and find out the principles for applying them effectively.

(3) Develop the necessary API, compiler support, and runtime libraries to integrate all the findings and insights into a software framework, which can then automatically apply the optimizations to any problem of that class.

To the best of our knowledge, there is no prior work on any of the three steps. They hence all need some innovations and substantial work to develop; the second step turns out to be particularly challenging in our exploration.

The rest of the paper starts with two examples to convey some intuition on distance optimizations in Section 2, then presents the unifying abstraction in Section 3, describes the principled analysis of distance optimizations in Section 4, the TOP framework in Section 5, evaluations in Section 6, and finally concludes with a short summary.

2. EXAMPLES FOR INTUITION

In this section, we describe two example problems that involve distance calculations and point out some unnecessary distance calculations in them. The conveyed intuition will help understand the following discussions on the design of the proposed software framework.

K-Means is a popular clustering technique. It tries to group some points into K clusters. It runs iteratively, starting with K initial centers and stopping at convergence. In each iteration, it labels every point with the center that is closest to it, and then uses the average location of the points in a cluster to update its center. In the standard K-Means, each iteration computes the distances between every point and every center in order to find the center closest to each point. Some of the calculations are unnecessary. Consider Figure 2 (a), where, c_1' is the center of point q in the previous iteration and c_1' gets updated into c_1 at the end of the previous iteration; c_2' and c_2 are the centers of another cluster in the two iterations. If we can quickly get the upper bound of the distance between q and c_1, denoted as $d(q, c_1)$, and the lower bound of $d(q, c_2)$, we may compare them first.

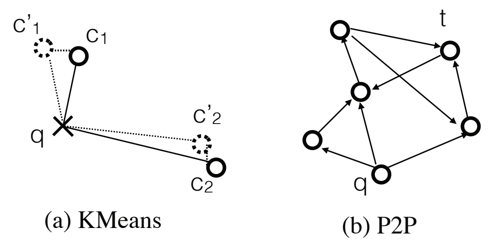


Figure 2: Example distance problems.

If the former is smaller than the latter, we can immediately conclude that c_2 is impossible to be the new center for x and avoid computing $d(q, c_2)$.

Point-to-point shortest path problem (P2P), illustrated in Figure 2 (b), is another example. It tries to find the shortest path between two points in a directed graph. The length of each edge in the graph is known. Distance in this problem is defined on a 3-tuple (two points and a path between them), meaning the length of that path between the two points. During the search for the shortest path among all paths between a query point (q) and a target point (t), we can avoid computing the full length of a path if we can quickly determine that the lower bound of its length is greater than the length of the shortest path encountered so far.

3. UNIFYING ABSTRACTION

Although both involve distance calculations, the two examples described in the previous section differ in many aspects, including their domains (data mining versus graph analysis), nature of problem (iteratively putting points into groups versus finding a path in a graph), distance definitions, and constraints of distance calculations (subject to graph connectivity or not). It is hence not a large surprise that even though both problems involve unnecessary distance calculations, no research has tried to find commonalities in the two problems and provide a general solution for them or other distance-related problems. We find many papers published on optimizations specific to each of the two problems for avoiding unnecessary distance calculations ([7,9] for K-Means, [14,17] for P2P). We also find such problem-specific manual efforts in many other distance-related problems, even for some problems residing in the same domain (e.g., K-Means [7,9] and KNN [16,28]).

Despite the differences among these problems, they are all related with distance calculations. A key view motivating this study is that if we can have an abstraction to represent all such distance-related problems, we may be able to derive a general approach to optimizing algorithms for such problems at that abstraction level.

Abstraction. We introduce the notion of *abstract distance-related problem* as follows: It is an abstract form of the problems that aim at finding some kind of relations between two sets of points, a query set and a target set; the relations are about a certain type of distances defined between the two sets of points under a certain set of constraints. We denote such a problem with a five-element tuple (Q, T, D, C, R):

- **Q**: the query set of points. It may contain one or more points in a space. It is the central entity of the relations of interest.
- **T**: the target set of points. It is the other party of the relations of interest.
- **D**: a type of distance between points.
- **C**: constraints related with the problem. They can be about the connectivity between Q and T, available memory in the system, or some other conditions. A special condition is whether the distance problem of interest involves many iterations of update on Q or T. If so, we call the problem an *iterative distance problem.*
- **R**: the relation of interest between Q and T. It is about the distances between those points, such as the lower bound of the distance, the closest targets to a query point, and so on.

Mappings from the Concrete. The abstraction unifies various distance-related problems into a single form, making automatic algorithmic optimization possible. Table 1 presents how six important distance-related problems in various domains can be mapped to the abstraction form. Each of the six problems has been extensively studied in its specific domain, but they have never been treated together in a unified manner. We next explain them and the mappings briefly.

KNN tries to find the K target points that are closest to a query point. The "instantiation" column on the second row in Table 1 shows how it maps to the *abstract distance-related problem.* In that column, "x" stands for a single point (the query), and "S" for a point set (the target). The distance could be Euclidean or other distances, the constraint is that the memory cost (memCost) should be within a given budget (spaceBudget) determined by the user or machine, and the relation of interest is to find K points in S that are closest to x. KNNjoin is similar to KNN except that Q is a set of points rather than contains a single point; it tries to find the K nearest neighbors for each point in Q.

We have described K-Means in the previous section. It maps to our abstraction well. The set of points to cluster is Q, the center set in each iteration is T (the superscript in S^t in Table 1 stands for iterative update), its constraints include the iterative property and memory limit, and the relation of interest is the closest target for a query point.

ICP is a technique for mapping the pixels in a query image with the pixels in a target image. It is an iterative process. In each iteration, it maps each pixel in a query image with a pixel in the target image that is similar to the query pixel, and then transforms the query image in a certain way.

As aforementioned, P2P is a graphic problem that tries to find the shortest path between two points (one in Q, the other in T) in a directed graph. Q and T are two sets of points on that graph. The distance of interest is about the path between points, and is hence subject to the connectivity among vertices in the graph. The relation of interest is the lower bound of the path length between two points.

Many algorithms have been manually designed specifically for each of the five problems for avoiding unnecessary computations: KNN [10, 16, 20, 28], KNNjoin [5, 11, 24, 29, 30], KMeans [7, 9, 18, 25], ICP [15], P2P [14, 17].

Nbody [8] simulates the interplay and movements of particles in a series of time steps. In each step, it computes the distances between every particle and all particles in its neighborhood in every time step, from which, it derives the force that particle is subject to, computes its movement accordingly, and updates its position. The algorithm has some

Table 1: Six Important Distance-Related Problems.

Problem	Domain	Description	Instantiation
KNN [13] [28]	Data Mining	Finding the K nearest neighbors of a query point	Q={x}, T=S, D*: Euclidean, C: memCost<spaceBudget, R: K points in S closest to x
KNNjoin [2] [24]	Data Mining	Finding the K nearest neighbors of each query point	Q=S_1, T=S_2, D*: Euclidean, C: memCost<spaceBudget, R: K points in T closest to each point in Q
K-Means [22] [7]	Data Mining	Clustering query points into K groups	Q=S_1, T=S^t, D*: Euclidean, C: memCost<spaceBudget & repeated invocations, R: the point in T that is closest to each point in Q
ICP [3] [15]	Image Processing	Matching two images	Q=${S_1}^t$, T=S_2, D*: Euclidean, C: memCost<spaceBudget & repeated invocations, R: the point in T that is closest to each point in Q
P2P [4] [14]	Graphics	Finding the shortest path between two points on a directed graph	Q=S_1, T=S_2, D: path length, C: memCost<spaceBudget & graph connectivity, R: lower bound of the distance between query and target
Nbody [8]	Computational Simulation	Simulate movements of particles caused by their interactions	Q=S^t, T=S^t, D: Euclidean, C: memCost<spaceBudget & repeated invocations, R: set of points in T that are no farther than r from a query point

S, S_1, S_2 are all sets of points, which may be identical or different; superscript t means that the set could get dynamically updated; x is one point; D* could be defined as other types of distance; r is a constant give beforehand.

variations. The one used in this work defines the neighborhood of a particle as a sphere of a given radius. The Q and T in this problem are identical, referring to the set of particles.

4. PRINCIPLES OF DISTANCE OPTIMIZATIONS

With the abstraction offering a unified representation of the various distance-related problems, it becomes possible to extract the essence of the various manually designed optimization to those problems, and reason about the principled ways for optimizing distance-related problems.

An important insight from this work is that all the previously proposed solutions are essentially just certain instantiation of triangular inequality in the context of the specific problem. In this section, we first give a formal presentation of triangular inequality—the basis for all the optimizations, and then discuss some conditions under which triangular inequality could help avoid unnecessary distance calculations for distance-related problems. After that, we present seven principles we attain for using triangular inequality for effective optimizations, which serve as the foundation for our automatic optimization framework TOP.

4.1 Triangular Inequality (TI): Concepts and Implications

We give the formal definition of TI as follows:

Let a, b, c represent three points and $d(a, b)$ represent the distance between a and b; *triangular inequality (TI)* states that $d(a, c) \leq d(a, b) + d(b, c)$.

Although TI does not hold for all kinds of distances, it holds for many common ones (e.g., Euclidean distance). It provides an easy way to compute both the lower bound and upper bound of the distance between two points as follows. Figure 3 gives an illustration.

$$|d(a, b) - d(b, c)| \leq d(a, c) \leq d(a, b) + d(b, c) \qquad (1)$$

Formula 1 offers the fundamental connection between TI and distance-related problems. Intuitively, if the lower or upper bound of the distance between two points could be used in place of their exact distance in solving a distance-related problem, the bounds provided by Formula 1 may save the calculation of their exact distance.

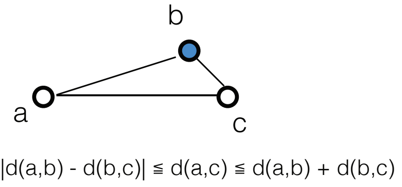


Figure 3: Illustration of distance bounds obtained from Triangular Inequality (*b* is a *landmark*).

But how using the bounds could help may not be immediately clear. As the formula shows, to get either the upper or lower bound of the distance between two points "a" and "c" in order to save the calculation of $d(a, c)$, we need two distances $d(a, b)$ and $d(b, c)$. So at the first glance, there seems to be no benefit but extra cost to use the bounds. However, when we consider the context of distance-related problems, the benefits become easy to see. It relates with the following two concepts we introduce.

Landmarks and Distance Reuses. Recall that in the distance-related problem this paper defines earlier, there are two sets of points, Q and T. Suppose that the objective is to find out the upper bounds of the Euclidean distances between every point in Q and every point in T. We compare two methods. The first directly computes all the distances between the two point sets; there would be $O(|Q| * |T|)$ distances to compute. The second picks a point x (e.g., randomly selected from Q or T), computes the distances between x and every point in Q and T, and then applies TI to obtain the upper bounds: $d(q, t) \leq d(q, x) + d(t, x)$. The number of distance computations would be $O(|Q| + |T|)$, much smaller than in the first method when $|Q|$ and $|T|$ are non-trivial. We call x an intermediate point or a *landmark*. Using more than one landmark can help tighten the obtained bounds (to be elaborated in the next section.)

We further examine the reasons for the saving. Fundamentally, the saving comes from reuses of the distances between a point and a landmark. The computations of the upper bounds between each point in Q and a point t in T all use $d(t,x)$ (i.e., $|Q|$ reuses), and similarly, the computations between a point q in Q and each point in T reuses $d(q,x)$ for $|T|$ times. We call such reuses *spatial reuses*, formally defined as the reuse of distances across points.

Besides spatial reuse, temporal reuse can also help. As mentioned in Section 3, some distance-related problems involve iterative updates to either Q or T. It is possible to use the counterpart (q') of a point in the previous iteration as the landmark for that point (q) in the current iteration. If the distance between q' and a target point t, $d(q',t)$, and the movement of the point between the two iterations, $d(q',q)$, are known (or properly estimated), the bounds of $d(q,t)$ can be computed with TI directly; no extra distance calculations would be needed. Such distance reuses across iterations are called *temporal reuses*. Notice that in that case, the distance bound needs only a scalar operation to compute, while in comparison, the distance between two d-dimension points requires d dimensional computations.

4.2 Principles for Optimization Designs

With landmarks and distance reuses, one can better understand the underlying reasons for TI to be helpful for optimizing distance-related algorithms. But to tap into the full potential of TI, the optimization needs to be adaptive to fit the properties of each individual algorithm, given that distance-related problems may vary in every component listed in Section 3. This section presents a set of design principles we attain through this study.

Applicability. First, we list the basic conditions a distance-related problem should meet such that a TI optimization applies:

(1) *Problem Condition:* The solution of the distance-related problem must involve some kinds of comparisons of distances among points.

(2) *Distance Condition:* The definition of the distance involved in the comparisons must obey triangular inequality.

The Problem Condition comes from the inequality nature of TI, while the Distance Condition is necessary for TI to hold. Many distance-related problems, including all the example problems discussed in Section 3, meet these conditions.

Design Objective and Dimensions. There are two primary considerations when designing a TI optimization: optimization quality and cost. The quality is about how much computation the optimization can help avoid. It is determined by both the tightness of the distance bounds offered by TI and the way the bounds are used in solving the distance-related problem. The cost is mainly about the space and time overhead introduced by the TI optimization. TI optimizations usually require some computations and auxiliary space to work. The objective of TI optimization design is to maximize the quality while minimizing the time overhead and confining the space cost to an acceptable level (e.g., within a memory budget).

We crystallize the many aspects in designing TI optimizations into two dimensions: how landmarks are defined and how they are used in distance comparisons. We next explain each of the two dimensions, along with seven principles for applying TI optimizations, which form the foundation of our framework TOP.

4.2.1 First Dimension: Landmark Definition

Definition of landmarks determines the tightness of the computed distance bounds, as well as the cost of TI optimization. We first explain some principles for effective definitions of landmarks, and then provide the whole taxonomy of definitions applicable to each category of problems.

Principle I. A good landmark for a pair of points should be close to either of the two points to get tight distance bounds. We prove it as follows. According to the definition of TI, for points a, b and a landmark c, the upper bound of the distance $d(a,b)$ through TI is $d(a,c)+d(b,c)$, while the lower bound is $|d(a,c)-d(b,c)|$. Their difference is $2*\min(d(a,c),d(b,c))$. Therefore, the closer the landmark c is to either a or b, the tighter the bounds are.

Principle II. Having more than one landmark can help TI tighten bounds, if the closestLandmark information is given. *ClosestLandmark* is about which landmark is closest to each point of interest. This principle directly follows Principle I: More landmarks, more choices, and the closestLandmark information allows TI to operate on the landmark that produces the tightest bound among all landmarks. Such information is sometimes free or easy to obtain. Principle IV will elaborate on this point.

Principle III. A landmark hierarchy can help strike a good tradeoff between cost and quality. Principle II says that more landmarks could help tighten bounds, but they could also increase the time and space overhead. A landmark hierarchy helps address the dilemma by having more than one level of landmarks. The bottom level has a relatively larger number of landmarks while a higher level has fewer; each landmark at a higher level represents a group of lower-level landmarks. Use of the fine-grained landmarks at the bottom level may help obtain a tight bound in critical situations, while use of the coarse-grained landmarks at the higher levels in other situations may help reduce the space and time overhead.

Figure 4 exemplifies the benefits of a landmark hierarchy. What it shows is a small step in K-Means clustering that tries to find the center closest to a query point q. Centers get updated in each iteration of K-Means. In Figure 4, we use a broken-line circle to represent the location of a center in the previous iteration—which, we call the *ghost* of the center. For instance, C_1' is the ghost of C_1 in Figure 4. A possible landmark hierarchy is to use the ghosts of all centers as the low-level landmarks, and treat a group of low-level landmarks lying closely to one another as a high-level landmark. For instance, the broken-line oval at the top of Figure 4, G_2', is a high-level landmark corresponding to the two low-level landmarks it contains.

In this example, the usage of two levels of landmarks is as follows. The low-level landmark C_1' is used to compute the upper bound of the distance between q and C_1 (the new position of the center that was closest to q in the previous iteration); the bound is $UpBound(q,C_1)=d(q,C_1')+d(C_1',C_1)$. A high-level landmark is used to compute the lower bound of the distance between q and the group of centers corresponding to the landmark; $LowBound(q,G_i)$ is computed as the difference between $LowBound(q,G_i')$ and the maximal distance that the centers in G_i' have moved since the previous iteration. If $UpBound(q,C_1) < LowBound(q,G_i)$, no center in G_i is possible to be the center closest to q, and hence,

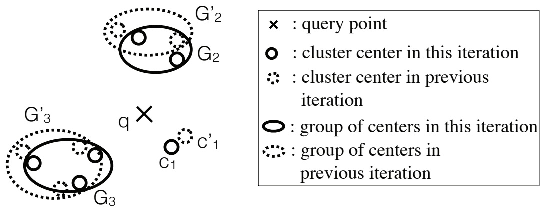

G'2
G2
G'3
q
C'1
C1
G3
× : query point
o : cluster center in this iteration
: cluster center in previous iteration
: group of centers in this iteration
: group of centers in previous iteration

Figure 4: Example of the use of landmark hierarchy in a step of K-Means.

no need to compute the distances between q and those centers. This example uses the low-level landmarks to ensure the tightness of $UpBound(q, C_1)$ because it is used in the comparisons with all lower bounds. It uses the high-level landmarks for lower bounds calculation to reduce the space and time overhead: Fewer lower bounds $LowBound(q, G'_i)$ need to be recorded than using low-level landmarks for lower bounds computations, and also, fewer lower bounds need to be updated across iterations. The example demonstrates the potential benefits of having a landmark hierarchy.

Principle IV. For iterative distance-related problems in which the locations of points in Q or T change slowly across iterations, the locations of the points in the previous iteration shall be considered as landmarks for the current iteration. We call the counterpart of a point in the previous iteration as the *ghost* of the point in this iteration. Using ghosts as landmarks has two advantages. First, it naturally leverages temporal reuse of distances because the distances (or distance bounds) from ghosts to some points are typically revealed in the previous iteration. These distances can be useful in computing distance bounds in the current iteration (e.g., the example in Figure 4.) Second, the *ClosestLandmark* information could come for free: The ghost of a point could serve as a landmark close to that point when points move slowly across iterations.

Principle V. For non-iterative problems (e.g., P2P) or the first iteration of an iterative problem, using landmarks to leverage spatial reuse is often beneficial. The reuse can often help save distance computations. There are many possible ways to select the landmarks. A method we find working well is to cluster points in Q or T (depending on the landmark selection as explained later) to create such landmarks. This method finds landmarks better representing the points and gives tighter bounds. The clustering can be lightweight; we find it enough to run K-Means for five iterations and use the centers as landmarks.

Taxonomy of Landmark Definitions. Guided by those five principles, we come up with a taxonomy of landmark definitions, shown in Figure 5. The graph shows the classifications of various distance-related problems into five categories based on whether the problem is iterative, whether Q equals T, and which point set gets updated across iterations (if the problem is iterative). The graph shows the set of landmark definitions suiting each of the categories. We explain each of the definitions as follows and then discuss how they are selected for a given distance-related problem.

(1) *1L1M, 1L2M, 2L1M, 2L2M:* In these definitions, "L" stands for "level", "M" stands for "landmarks". In all of

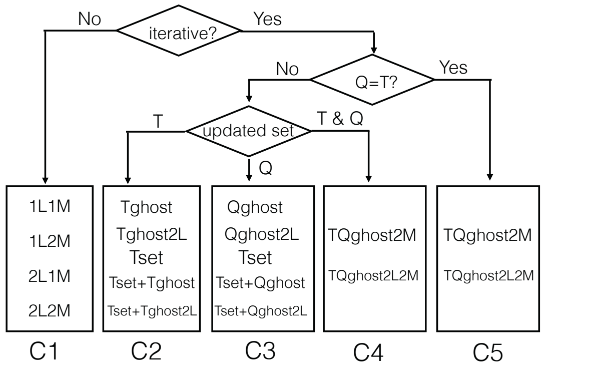


Figure 5: Taxonomy of landmark definitions in each category of distance-related problems.

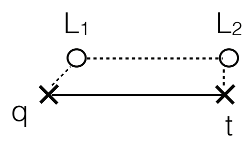
them, there are a number of landmarks created through simple clustering as Principle V mentions.

In "1L1M", the computation of the bounds of the distance between a query point and a target point is through one landmark (just like what Figure 3 shows), which shall be close to the target point. In "1L2M", the computation is through two landmarks, one shall be close to the query point, the other close to the target point, as illustrated in Figure 6. To compute the bounds between all pairs of query and target points, "1L1M" requires $(m * z + n)$ distances to be computed (z for the number of landmarks, m and n for the size of the query and target sets): It needs to compute the distance from every query point to every landmark, and the distance from every target to a landmark close to it (which is typically revealed already during the creation process of the landmarks). "1L2M" requires $(m + n + z_q * z_t)$ distances (z_q and z_t for the numbers of landmarks closest to queries and targets respectively) since it needs the distance from each query or target to only its closest landmark, and the distances between query-side landmarks and target-side landmarks. When there are much fewer landmarks than queries and targets, "1L2M" needs fewer distance computations. However, the bounds given by "1L2M" are usually not as tight as "1L1M" gives.

The landmark definitions in "2L1M" and "2L2M" are similar to those in "1L1M" and "1L2M", except that upon those basic landmarks, they derive some high-level landmarks. Two levels could lead to a better cost-benefit tradeoff (Principle III). The difference between "1L1M" and "1L2M" is just whether one or two landmarks are used in bounds computation. Although it is possible to have a hierarchy with more than two levels of landmarks, we have not observed much extra benefit with that increased complexity.

All these four definitions leverage spatial reuses. They suit non-iterative distance problems and the first iteration of iterative distance problems. We next explain the definitions specific to other iterations of iterative distance problems.

(2) *Tghost, Qghost:* These two definitions use either the ghosts of targets or queries as the landmarks, depending on which set gets updated across iterations (and hence has ghosts). As Principle IV mentions, using ghosts as landmarks have some advantages: The distances (bounds) from landmarks to points are often known and the *ClosestLandmark* information is often available.



$$d(q,t) \geqq d(L1,L2)-d(q,L1)-d(L2,t)$$
$$d(q,t) \leqq d(q,L1)+d(L1,L2)+d(L2,t)$$

Figure 6: Illustration of how two landmarks can be used for computing lower and upper bounds of distances.

(3) *Tghost2L, Qghost2L:* These two definitions are similar to *Tghost* and *Qghost* except that a set of high-level landmarks are introduced to complement the low-level landmarks to lower the space and time overhead (just like the differences between *2L1M* and *1L1M* mentioned earlier.)

(4) *Tset:* In the *Tset* definition of landmarks, points in the target set T are used as landmarks. The bounds of the distance between q and a target point t is obtained by applying TI to q, t, and $L(q)$, where, $L(q)$ is a target point close to q. This definition works when it is known which target is close to which query point. An example is K-Means, in which, every iteration determines the center closest to each query point. Although the centers may move across iterations, the movement is often small. As a result, the closest center to a query point in an iteration usually remains close to that query point in the next iteration. This definition is not applicable to non-iterative problems because the *CloseLandmark* information is not available in those problems. Usage of this definition for TI requires computation of $d(q, L(q))$ and $d(t, L(q))$; there are as many as $|Q|$ computations of $d(q, L(q))$, and $|T|^2$ computations of $d(t, L(q))$. When $|T| << |Q|$, the amount is still much less than pairwise distances between Q and T.

(5) *Tset+Tghost, Tset+Tghost2L:* These two definitions are a combination of Tset and Tghost or Tghost2L. The idea is to apply TI first to Tset landmarks. If the bounds are insufficient for avoiding the distance computation for a pair of query and target, Tghost or Tghost2L are then used for attaining tighter bounds for that pair. Such a combination could be beneficial because checks with Tset are faster to do, while the bounds from Tghost and Tghost2L are tighter. The combination gets the best of both worlds. Tghost2L is preferred over Tghost if space is an issue.

(6) *TQghost2M, TQghost2L2M: TQghost2M* is similar to *Tghost* except that to compute the bounds of a distance, it uses two landmarks: One is the ghost of the query, the other is the ghost of the target. To use this landmark definition, one needs to record the distances (or their bounds) between every pair of query and target in each iteration. That could incur large space and time overhead. *TQghost2L2M* includes high-level landmarks to lower the space and time cost (in a vein similar to *2L1M* versus *1L1M* mentioned earlier).

These two definitions apply only when Q=T or Q and T both get updated across iterations—when both Q and T have ghosts. On the other hand, the definitions that apply to the other cases do not apply to these two cases because those definitions assume no cross-iteration update of either Q or T or both.

Selecting Landmark Definitions. As Figure 5 shows, multiple landmark definitions may be applicable to a problem, and one definition can have many possible configurations (e.g., number of landmarks). We now explain how to select a definition fitting a given problem.

A suitable landmark definition should have an acceptable space cost and at the same time minimize the time for solving the problem. Space cost includes the space for storing landmarks and the distances (or bounds) between points and landmarks. It is mainly determined by the size of the problem and the number of landmarks. With such information, the cost can be easily computed analytically.

Execution time is more complicated. The TI optimization helps avoid some distance calculations between queries and targets, but also introduces time overhead, including the time for computing distance bounds between queries and targets, distances (or bounds) from landmarks to queries or targets, and extra comparisons among bounds and distances for avoiding distance calculations. The benefits and costs depend on the size of the problem, the number of landmarks, but also the locations or distributions of the queries and targets. It is more difficult to compute the time cost and benefit analytically. One option is to use runtime sampling to model the distributions of the points, and then infers the amount of distance computations each definition may avoid and estimates the time benefits and cost accordingly. Due to its complexity, we leave this option for future study. In this work, we instead use a sequence of rules obtained empirically for definition selection. These rules are not intended for optimal selections, but offer a simple way to make good selections in practice.

The rules together form a selection algorithm, outlined in Figure 7. For category 1, the algorithm uses 2-level landmarks if the platform is a distributed system, and 1-level otherwise. The number of top-level landmarks in the 2-level case equals the number of computing nodes on the platform. Regarding whether TI should be applied with one or two landmarks each time, the algorithm first examines how many landmarks the space budget allows if the one-landmark scheme is used. If it is too few (less than $\sqrt{(|Q|)}$), the one-landmark scheme is unlikely to offer tight distance bounds, and the two-landmark scheme should used. Because the two-landmark scheme does not require as many distances to be stored as one-landmark scheme requires, the space budget could allow more landmarks created and hence offer tighter distance bounds.

For category 2, the algorithm first decides whether Tset should be used. Since Tset needs the computation of the distances between every pair of targets, it applies only when $|T|$ is small (less than $0.01 * |Q|$). After that, the algorithm tries to decide whether Tghost or Tghost2L should be used in case that the bounds from Tset are not tight enough. One condition is whether there is enough space for Tghost. If so, d, the number of dimensions of the data space, is checked. Tghost is used only if d is large enough (no smaller than 1000). Otherwise, Tghost2L is used. The condition on d comes from the following reason. Tghost may avoid more distance calculations than Tghost2L does because it always uses low-level landmarks for bound computations. However, it adds more bound computations and distance checks than Tghost2L does—Tghost2L does bound computations and distance checks only once for a group of rather than every low-level landmark. So, Tghost is better only if a distance calculation is much more costly than a bound computation or check. The cost of a distance calculation is mainly determined by the number of dimensions of the data space, hence the condition. Treatment to category 3 is the same as to category 2 except that Qghost or Qghost2L rather than Tghost or Tghost2L is used.

For categories 4 and 5, the main question is whether 1-level or 2-level landmarks should be used. The conditions to check are the same as those checked for determining the number of landmark levels in category 2.

After the type of landmark definition is determined, function "configure" sets up the number of landmarks to generate. For category 1, the number of low-level landmarks is $2\sqrt{|Q|}$ for the query set and $2\sqrt{|T|}$ for the target set. Such numbers come from previous domain-specific explorations [24, 28], which each studies only a specific distance-related problem, but finds the same choice of the number of landmarks working well. If two levels are used, the number of landmarks at the top level equals the number of computing nodes in the distributed system. For the other categories, the number of low-level landmarks either equals $|T|$ or $|Q|$ since the landmarks are just their ghosts. When the 2-level scheme is used, the number of the top-level landmarks equals $\sqrt{2 * \sqrt{|X|} * |X|/10}$, where X should be replaced with T or Q depending on which set the landmarks are created for. This formula is a combination of the considerations for the spatial and temporal reuses. Recall that for iterative problems, we exploit spatial reuse for the first iteration and temporal reuse for the future iterations. The first part of the formula, $2 * \sqrt{|X|}$, is the best number of landmarks for it (as discussed in category 1, which leverages only spatial reuse). The second part of the formula, $|X|/10$, is a good choice for temporal reuse as found in our experiments. The formula is a geometric mean of the two.

4.2.2 Second Dimension: Comparison Order

Besides landmark definition, another important dimension for TI to work effectively is how the bounds TI produces are used. More specifically, the order in which the bounds are checked (called *comparison order*) can affect the computing efficiency significantly. For example, suppose the goal is to find a target closest to a query q. Let d_{min} be the shortest of the distances found so far between q and targets. For a target t, before computing $d(q,t)$, one can first check whether the lower bound of $d(q,t)$ (obtained through TI) is larger than d_{min} and skip computing $d(q,t)$ if so. In this example, the comparison order refers to the order in which the targets are checked. If the order is an ascending order of the lower bounds of $d(q,t)$ among all t, the check can stop immediately when it encounters one target whose lower bound is greater than d_{min}: All the remaining targets must have lower bounds greater than d_{min} as well because of the ascending order.

Let S be a set of target points whose distance bounds from a query point q are based on the same landmark l. Our analysis gives the following principle.

Principle VI. **When the objective is to find the point in S that is the closest to the query q, the comparison order should be the ascending order of $d(t,l)$ $(t \in S)$ if l is closer to q than to all the targets, and should be the descending order otherwise.** It is easy to see that such an order is equivalent to the ascending order of the lower bounds of the distances from the targets to the query: The lower bound equals $d(l,t) -$

```
input: query set $Q$, target set $T$, number of dimensions of the data
  space $d$, space budget $Budget$, category of the problem $cat$.
  if cat==1 then
    // to use 1-level or 2-level landmarks
    L=1;
    if distributedPlatform then
      L=2;
    end if
    // to use 1 or 2 landmarks as intermediate points
    M=1;
    nMax=maxLandmarks(Budget, cat, L, M, $|T|$, $|Q|$);
    if nMax< $\sqrt{|Q|}$ then
      M=2;
    end if
  end if
  if cat== (2 || 3) then
    // to decide whether Tset is to be used
    useTset=false;
    if $|T| < 0.01 * |Q|$ then
      useTset=true;
    end if
    // to select Tghost/Qghost or Tghost2L/Qghost2L
    if cat==2 then
      spaceNeeds = estimateSpaceCost(Tghost, $|T|$, $|Q|$,useTset);
    else
      spaceNeeds = estimateSpaceCost(Qghost, $|T|$, $|Q|$,useTset);
    end if
    L=1;
    if spaceNeeds>Budget || d<1000 then
      L=2;
    end if
  end if
  if cat==(4 || 5) then
    // to select TQghost2M or TQghost2L2M
    spaceNeeds = estimateSpaceCost(TQghost2M, $|T|$, $|Q|$);
    L=1;
    if spaceNeeds>Budget || d<1000 then
      L=2;
    end if
  end if
  // setting the # of landmarks based on space budget
  configure(Budget, cat, L, M, $|T|$, $|Q|$);
```

Figure 7: Algorithm *pickLandmarkDef* for selecting landmark definitions.

$d(l,q)$ if the landmark is closer to the query, and equals $d(l,q) - d(l,t)$ otherwise.

The following principle is symmetric to Principle VI.

Principle VII. **When the objective is to find the target that is the farthest from the query, the comparison order should be the descending order of the distances from the targets to the landmark.** This order is equivalent to the descending order of the upper bounds of distances from the targets to the query, since the upper bound equals $d(l,t)+d(l,q)$ regardless where l is.

When the two principles are used, many targets that are impossible to be the closest or farthest could be skipped from consideration. It helps avoid computing not only the distances from them to the query, but also the lower bounds of those distances since the principles are based on the distances from targets to landmarks rather than lower bounds.

5. TOP FRAMEWORK

To turn the abstraction and optimization into applicable tools, we design a software framework named TOP (which stands for triangular optimization). TOP consists of three components: a set of APIs that users can use to formally define a particular distance-related problem, a runtime library that implements the principles and rules for creating optimized algorithms to fit the user-defined distance problem, and a compiler module that helps the runtime obtain necessary information.

5.1 APIs

We introduce a small set of APIs, with which, users can easily define their distance-related problem in a way that it can be analyzed and handled by the TOP compiler module and runtime. The APIs in our current implementation are intended to be used with C or C++ languages; it can be easily modified to work with other languages.

As Section 3 lists, there are five components of a distance-related problem: query set Q, target set T, constraints C, distance definition D, and inter-point relations of interest R. The APIs contain entries for specifying each of them, as summarized in Figure 8. It includes some predefined structures for a data point and a point set. It has a cost matrix structure TOP_costMat for expressing connection constraints among points (e.g., points in a graph). Let M be a TOP_costMat; if $M[i, j] >= 0$, there is an edge from point i to point j with edge weight equaling $M[i, j]$; otherwise, no edge between them. It is symmetric if the graph is undirected. Figure 8 omits some structures defined for representing sparse matrices or graphs, and some APIs to facilitate users in constructing cost matrices.

In addition, the APIs for constraints contain a TOP_update function, which users may implement to update a point set S. Its returned value in "changedFlag" indicates whether the point set gets actually updated. This function helps compiler and runtime determine whether the distance problem updates a point set and which set it is. The APIs for distance definition include a function to specify the distance in the problem if it is one of a set of predefined distances (Euclidean etc.) that are amenable to TI. It has another function which users may implement to define their own distances, in which case, it would be the users' responsibility to ensure that the distance is amenable to TI. Automatic inference of the property could be possible, but not in the current implementation of TOP yet.

The final part of the APIs is for specifying the kind of relations of interest between query points and target points. TOP currently includes four basic relations: get the lower bound of a distance, get the upper bound of a distance, find a certain number of targets that are closest or farthest to a query point, find all the targets that reside within or beyond a certain distance from the query point. It turns out that they are enough to cover all the relations in common distance-related problems. Consider the relations of the six problems listed in Table 1. TOP_findClosestTargets() can be used to express the relations in KNN, KNNjoin, K-Means, and ICP; TOP_getLowerBound() can be used for P2P; TOP_findTargetsWithin() can be used for NBody. A usage example is the fifth statement in the K-Means example in Figure 1. The purpose of such a statement is to inform the TOP compiler the relations of interest. Users do not need to implement that relation function; the compiler and runtime of TOP automatically determine the best ways to implement it under the guidance of the seven principles, as detailed in the next section.

Using the APIs to define a distance problem is simple. For example, the six lines of code as we have illustrated in Figure 1 compose the main part of the code for K-Means.

```
Predefined structures:
TOP_point, TOP_pointSet, TOP_costMat, …

API for Constraints:
TOP_update (TOP_pointSet S, int * changedFlag, …);
some facilities for cost matrix construction;

API for Distance:
TOP_defDistance (enum);
TOP_defDistance (TOP_point, TOP_point, TOP_costMat);

API for Relation:
TOP_getLowerBound (TOP_pointSet, TOP_pointSet, TOP_costMat);
TOP_getUpperBound (TOP_pointSet, TOP_pointSet, TOP_costMat);
TOP_findClosestTargets (int, TOP_pointSet, TOP_pointSet, TOP_costMat);
TOP_findFarthestTargets (int, TOP_pointSet, TOP_pointSet, TOP_costMat);
TOP_findTargetsWithin (float, TOP_pointSet, TOP_pointSet,TOP_costMat);
TOP_findTargetsBeyond (float, TOP_pointSet, TOP_pointSet, TOP_costMat);
```

Figure 8: Core APIs defined in TOP.

The first statement defines that the kind of distance is Euclidean distance, the second statement invokes a function (which a programmer writes) to load data, the third statement initiates a flag variable to indicate the convergence of the algorithm, the while loop runs until the results converge, the fifth statement indicates the kind of relations of interest, and the sixth statement indicates that the target data set T could get updated—the programmer may write the body of update function to instantiate the update process.

With the semantics conveyed by the invocations of the APIs, the TOP compiler and runtime can then automatically optimize the algorithm by selecting the best landmarks and applying triangular inequality to materialize the algorithm with the amount of distance calculations minimized.

5.2 Runtime Library and Compiler

The runtime library consists of three parts. The first part is for selecting and configuring landmark definitions. At its core is a function *pickLandmarkDef* that implements the algorithm for selecting and configuring landmark definitions as what was shown in Figure 7 in Section 4.2.1. A runtime invocation of this function will determine the landmark definition suiting the particular problem instance. The second part is for materializing the TI optimization. It contains a set of functions that implement the TI-based optimization for the various kinds of relations listed at the bottom part of the APIs in Figure 8. For each of the relations, a number of versions are created with each as an optimized algorithm based on one type of landmark definition. Each of them records necessary bounds or distances for the TI to work, and applies TI by drawing on the landmarks to avoid as many distance computations as possible. These first two parts of the TOP runtime library form the low-level APIs of TOP. The implementation of the high-level APIs contains some condition checks such that they invoke the correct TI-optimized algorithms by calling the right low-level APIs in the runtime library.

For instance, the library contains 15 functions, which each implement a TI-based algorithm for finding the closest targets for a query point. They all try to use TI to estimate the lower bound of the distance between a query point and a target and avoid computing their distances if the lower bound is larger than the current minimum distance. They differ in what landmarks are used for getting the lower bounds, and in the operations related with the maintenance of the landmarks. An invocation of *TOP_findClosestTargets* selects one of them based on the category of the current problem and the chosen definition of the landmarks.

The versions in the library subsume and go beyond existing manually designed problem-specific algorithms that leverage TI, thanks to the taxonomy we have obtained through this systematic study.

The main functionalities of the compiler module are twofold. First, it inserts invocations of some low-level API calls (e.g., *pickLandmarkDef*) into the original program. Second, it analyzes the code to determine whether the problem is iterative and which data set gets updated across iterations. It passes these information to the runtime library by inserting several low-level API calls before the invocation of *pickLandmarkDef*. In a similar way, it helps inform the TOP runtime library other necessary information (e.g., size and dimensionality of data sets) that are collected at runtime. The implementation of the compiler is based on LLVM [21].

6. EVALUATION

To demonstrate the efficacy of TOP, we run it on six algorithms and compare with two versions: the *original* and the *manual*. We implement the former by following the classic algorithms as shown in the first references in the leftmost column in Table 1. They have no triangular optimizations applied. We implement the latter by following the second references in the same column in Table 1, which are the recent papers on manually applying triangular optimizations to those algorithms. Because we find no previous work that applies triangular optimizations to N-Body, its *original* and *previous* versions are identical. All code is in C++.

We test each algorithm on a number of inputs and problem settings (e.g., number of clusters K of K-Means). When selecting inputs, we try to include the inputs that have been used in the previous papers on manual optimizations; when they are unavailable or not detailed, we include public datasets that are commonly used in the domains. Table 2 lists the input size, data dimensions, and problem settings. P2P and Nbody have no variation in settings.

Since the optimized algorithms have the same semantic as the original algorithm has, they produce the same outputs. Our discussion focuses on performance (running time). The performance data are collected on a workstation equipped with Intel i5-4570 CPU and 8G memory. Each performance number comes from the average of five repeated runs.

6.1 Results Overview

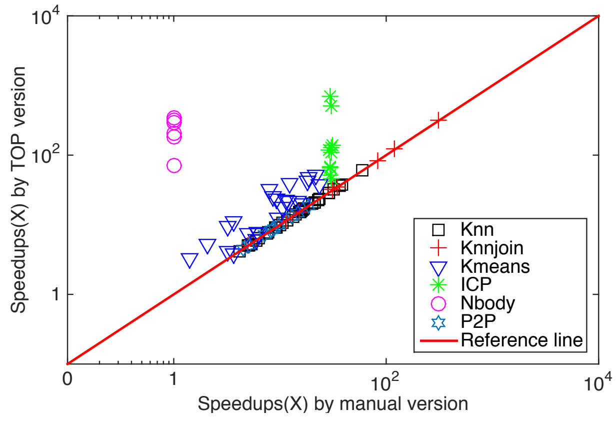
The graph in Figure 9 reports the speedups brought by the automatic optimizations by TOP and by the experts' manual optimizations; one point per input and problem setting. The baseline is the execution time of the *original* version (without triangular optimizations). The speedups by both TOP and previous manual optimizations are substantial, as much as 50X and 20X on average as the table in Figure 9 shows. The accelerations come primarily from the savings of distance computations enabled by triangular inequality optimizations. As the table in Figure 10 reports, the savings are larger than 93% for all the benchmarks.

Note that in Figure 9, all points lie either on or above the reference line, indicating that TOP brings a similar or greater speedup than the manual optimizations do for all algorithms and inputs. The extra benefits of TOP are especially prominent on the iterative algorithms: ICP, K-Means, and N-Body. On them, guided by the seven principles, TOP is able to configure the triangular optimizations to strike a better tradeoff between optimization overhead and benefits. We provide some deeper insights by examining each of the three algorithms in detail next.

Table 2: Datasets and Problem Settings.

Program	Dataset source	Dataset size & dim	Settings
KNN	11 from UCI [1]†	2K-10^6; 6-707	k=10;40;160
KNNjoin	network, gassensor, kegg from UCI & muenchen [26]	14K-430K; 2-129	k=10;100
K-Means	4 from UCI & 4 image sets [27]◇	14K-2.5M; 4-384	k=16;256;1000
ICP	abalone, krkopt, letter, poke2 from UCI	4K-28K; 6-16	diff⋆=5%;20%
P2P	6 graphs [14]	260K-1.5M	-
Nbody	6 derived from 48-15cr.xyz & 32-15cr.xyz [19]	5K-440K; 3	-

†: car, krkopt, abalone, magic, shuttle, letter, poke1(2), sat, segment, dorothea;
◇: network, gassensor, kegg, census, caltech101, notreDame, tiny, ukbench
⋆: difference between query and target images



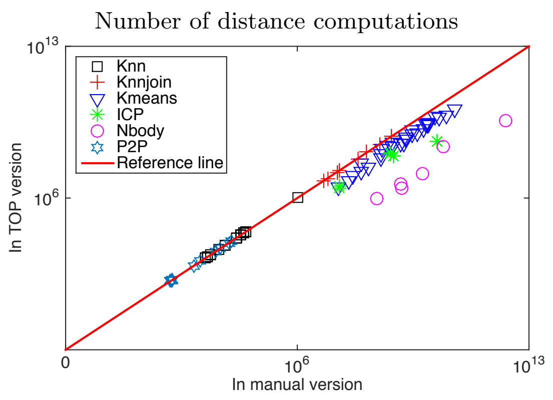
Average Speedups (X)

Prog	KNN	KNNjoin	K-Means	ICP	P2P	Nbody	*geomean*
TOP	17.0	77.1	27.6	193.4	9.8	237.6	50.3
Manual	16.6	76.4	9.6	29.9	10.1	-	20.6

Figure 9: Speedups brought by TOP and manual optimizations over basic algorithm implementations. The graph shows the results for each input and setting; the table reports the geometric average.

6.2 Analysis in Detail

ICP. ICP iteratively finds the best mapping between two images with the query image getting transformed after each iteration. The *pickLandmarkDef* algorithm of TOP selects C3:Tghost2L as the landmarks on all inputs and settings. The manual version uses a strategy similar to our *Tset* scheme. It pre-computes the distances between every two target points, and uses them when applying triangular inequality to detect and avoid unnecessary distance computations. TOP gets significantly larger speedups than the manual version (193X vs. 30X on average), for two reasons. First, it has much smaller overhead. The pre-computation in the manual version has a quadratic complexity in terms of the number of



Average Percentage of Computations Saved by the Optimizations

Prog	KNN	KNNjoin	KMeans	ICP	P2P	Nbody	*geomean*
TOP	93.0	95.6	92.9	99.6	93.2	99.4	95.6
Manual	93.0	95.6	84.37	97.5	93.2	-	92.6

Figure 10: The graph shows the number of computations in the optimized algorithms; the table reports the percentage of computations saved by the optimization.

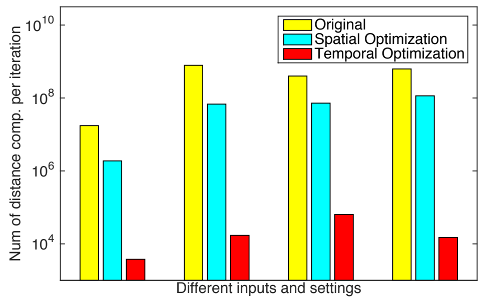


Figure 11: Both spatial and temporal optimizations save many distance computations for ICP.

data points, while the cost of TOP is close to linear, thanks to the uses of ghost points across iterations and the landmark hierarchy. Second, TOP exploits both spatial reuses (in the first iteration) and temporal reuses (in the follow-up iterations) while the manual version exploits only spatial reuses. Figure 11 shows the benefits of the two kinds of reuses in saving distance calculations (note the exponential scale of Y-axis). The second reason makes TOP avoid more distance calculations than the manual version does as shown in the table in Figure 10, while the first reason is the dominant factor for the significantly larger speedups—as both versions avoid most computations, their overhead becomes critical for the overall performance. The large overhead of the manual version weighs similarly over the entire execution time across all settings, hence its similar speedups across all settings; TOP, on the other hand, gives larger speedups on larger datasets.

K-Means. On K-Means, the target set T is the set of cluster centers, which is larger for settings with more clusters (i.e., a larger k). According to the pickLandmarkDef algorithm in Figure 7, TOP selects C2:Tset+Tghost2L when k is small and C2:Tghost2L otherwise. The manual version computes and maintains t lower bounds between a query point and its t closest target points and another lower bound for all the other target points. It uses these bounds to leverage triangular inequality for detecting and eliminating unnecessary distance computations. The idea is similar to our C2:Tghost2L, but not as efficient. First, it has a larger overhead to maintain those lower bounds because it needs to keep an increasing order among those bounds in every iteration. Second, the last lower bound (of all points except the closest t target points) is often too loose to effectively detect redundant distance computations. In addition, the previous work requires t to be a value between $k/4$ and $k/8$; the space cost for storing lower bounds frequently exceeds memory budget for large problems, causing the approach to fail. In our experiments, we extend the approach such that t is set to the largest possible value if space is an issue. Despite the extension, for the two reasons mentioned, TOP gives much larger speedups than the manual version does (27X vs. 9.6X on average). The advantage is more substantial when k increases, because the overhead of the manual version in maintaining the lower bounds becomes more significant.

N-Body. N-Body simulates the interplay and movements of particles iteratively. Because the dimensionality is usually low (two or three) in N-body problems, an $KD-tree$ based optimization is the most effective solution to them. Instead of finding the most efficient solution, this work uses the basic N-Body algorithm as a test case to see whether TOP can effectively optimize a distance-related problem on which no TI optimizations have been proposed before. The result shows that TOP selects C5:TQghost2L2M for creating its landmarks on all the inputs. The TI optimizations by TOP help save about 99.4% distance computations compared to the original version, yielding over 238X speedups.

Other Algorithms. TOP selects C1:1L2M for the landmark creation for the three non-iterative algorithms (KNN, KNNjoin, P2P). These algorithms benefit only from spatial reuses, which is relatively more straightforward to do. Previous manual optimizations can already achieve similar performance as TOP achieves. They can all save more than 93% of the distance computations and give 9-77X speedups. In both the TOP and manual versions, we adopt the comparison order as Section 4.2.2 describes, which boosts the speedups substantially (e.g., 1.5-4X speedups for KNN).

6.3 Discussions

A distinctive feature of TOP is the landmark hierarchy built through grouping. It brings large benefits on iterative algorithms as aforementioned. The number of groups decides the benefit-cost tradeoff. TOP uses a simple rule to decide the appropriate number of groups as showed in Section 4.2.1. Figure 12 provides a sensitivity study on the number of groups. The graphs show the overall running times when different numbers of groups are used. Each curve in the graphs corresponds to the performance in one problem setting. The curves all show the same trend: As the number of groups increases, the overall running time first decreases, and then increases, reflecting the tension between the cost in maintaining the bounds associated with the landmarks and the benefits in removing redundant computations. The stars in the graphs indicate the number of groups decided by TOP. The performance they give is less than 13% away from that of the best decisions.

The distance optimizations introduce no hazards for parallelism of the data processing. Different data points can

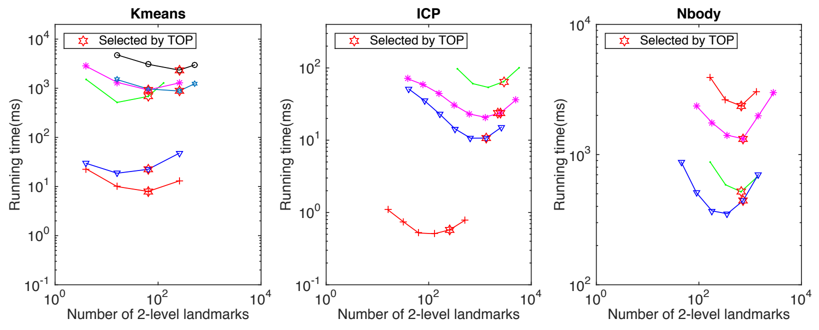


Figure 12: Running time changes when different numbers of 2-level landmarks are used.

still be processed in parallel. Our implementation of the optimized K-Means in a parallel framework Graphlab [23] confirms that a similar degree of large speedups exists in the parallel versions of the optimized algorithms [6].

7. CONCLUSION

At a high level, TOP shares some similarity with relational query optimization in the way that both ask for just a specification of what to compute, and determines a good plan for executing it automatically. This work, for the first time, makes such a paradigm possible and beneficial for optimizing distance-related algorithms. It develops the first set of principled analysis on how triangular inequality should be applied to a spectrum of distance-related problems. The resulting framework TOP is able to produce algorithms that either match or beat (by 2.5X on average) manually designed algorithms for a list of important problems.

Acknowledgement We thank the VLDB'15 reviewers for their suggestions. This material is based upon work supported by DOE Early Career Award and the National Science Foundation (NSF) under Grant No. 1464216, 1320796 and Career Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DOE or NSF.

8. REFERENCES

[1] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[2] C. Böhm and F. Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowledge and Information Systems, Springer*, 6(6):728–749, 2004.

[3] Y. Chen and G. Medioni. Object modeling by registration of multiple range images. In *Robotics and Automation, IEEE*, pages 2724–2729, 1991.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numerische mathematik*, volume 1, pages 269–271, 1959.

[5] H. Ding, G. Trajcevski, and P. Scheuermann. Efficient similarity join of large sets of moving object trajectories. In *Temporal Representation and Reasoning, IEEE*, pages 79–87, 2008.

[6] Y. Ding, X. Shen, M. Musuvathi, and T. Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *ICML*, 2015.

[7] J. Drake and G. Hamerly. Accelerated k-means with adaptive distance bounds. In *5th NIPS Workshop on Optimization for Machine Learning*, 2012.

[8] V. Eijkhout. *Introduction to High Performance Scientific Computing*. Lulu. com, 2010.

[9] C. Elkan. Using the triangle inequality to accelerate k-means. In *ICML*, volume 3, pages 147–153, 2003.

[10] C. Elkan. Nearest neighbor classification. *University of California–San Diego*, 2007.

[11] T. Emrich, F. Graf, H.-P. Kriegel, M. Schubert, and M. Thoma. Optimizing all-nearest-neighbor queries with trigonometric pruning. In *Scientific and Statistical Database Management, Springer*, pages 501–518, 2010.

[12] A. Fahim, A. Salem, F. Torkey, and M. Ramadan. An efficient enhanced k-means clustering algorithm. *Journal of Zhejiang University SCIENCE A, Springer*, 7(10):1626–1633, 2006.

[13] E. Fix and J. L. Hodges Jr. Discriminatory analysis-nonparametric discrimination: consistency properties. In *DTIC Document*, 1951.

[14] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM*, pages 156–165, 2005.

[15] M. Greenspan and G. Godin. A nearest neighbor method for efficient ICP. In *3-D Digital Imaging and Modeling, IEEE*, pages 161–168, 2001.

[16] M. Greenspan, G. Godin, and J. Talbot. Acceleration of binning nearest neighbor methods. In *Vision Interface, IEEE*, pages 337–344, 2000.

[17] R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *ALENEX/ANALC*, pages 100–111, 2004.

[18] G. Hamerly. Making k-means even faster. In *SDM, SIAM*, pages 130–140, 2010.

[19] H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *Parallel and Distributed Systems, IEEE*, 17(7):606–618, 2006.

[20] J. Z. Lai, Y.-C. Liaw, and J. Liu. Fast k-nearest-neighbor search based on projection and triangular inequality. *Pattern Recognition, Elsevier*, 40(2):351–359, 2007.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, IEEE*, pages 75–86, 2004.

[22] S. Lloyd. Least squares quantization in pcm. In *Information Theory, IEEE*, volume 28,2, pages 129–137, 1982.

[23] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[24] W. Lu, Y. Shen, S. Chen, and B. C. Ooi. Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1016–1027, 2012.

[25] W. K. Ngai, B. Kao, C. K. Chui, R. Cheng, M. Chau, and K. Y. Yip. Efficient clustering of uncertain data. In *Data Mining, 2006. ICDM'06, IEEE*, pages 436–445, 2006.

[26] OpenStreetMap. Open data commons open database license, 2014.

[27] J. Wang, J. Wang, Q. Ke, G. Zeng, and S. Li. Fast approximate k-means via cluster closures. In *Computer Vision and Pattern Recognition (CVPR), IEEE*, pages 3037–3044, 2012.

[28] X. Wang. A fast exact k-nearest neighbors algorithm for high dimensional search using k-means clustering and triangle inequality. In *Neural Networks (IJCNN), IEEE*, pages 1293–1299, 2011.

[29] C. Yu, B. Cui, S. Wang, and J. Su. Efficient index-based knn join processing for high-dimensional data. *Information and Software Technology, Elsevier*, 49(4):332–344, 2007.

[30] D. Zhang, C.-Y. Chan, and K.-L. Tan. Nearest group queries. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management, ACM*, page 7, 2013.