DISTANCE-BASED INDEXING: OBSERVATIONS, APPLICATIONS, AND IMPROVEMENTS

by

MURAT TAŞAN

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Dissertation Advisor: Dr. Z. Meral Özsoyoğlu

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

January, 2006
This dissertation is the culmination of a lengthy career here at Case Western Reserve University, and it certainly was not accomplished without the generous aide of many people. First and foremost, I must thank my family for everything I have and will achieve. I thank my father, Yavuz — who is the most generous man I know — for as I continue to age I continue to appreciate your sacrifice for your family, and I hope I will be able to replicate your efforts when the time comes for me to do so. I thank my mother, Andrea, for the amazing courage you displayed while abruptly switching careers served as a catalyst for my stay in academia, a decision I will never regret thanks to you. I thank my elder sister, Nani, for always being the big sister I can look up towards and seek sound advice from, I will strive continually to follow in your footsteps in life. And I thank my younger sister, Yasmin, for inspiring me to perpetually work towards being the big brother that you can always look up to, as I hope my work inspires you to strive to achieve great things. You have all given me the strength needed to pursue my dreams, and without you I would certainly be lost. Thank you all so very much.

I must next give great thanks to my advisor, Dr. Z. Meral Özsoyoğlu. During my graduate studies, you have served as the standard for what an advisor should be. Neither driving me down research paths that I was not excited about, nor neglecting my interests, you have simply and beautifully advised me. Whether I was lacking focus, too focused, or simply stumped by a problem, your door and ears have always been open to me, something I deeply appreciate. I am so grateful to the time you have spent with me, and I am certain you will continue to serve as my advisor in life long after I have
completed my studies here.

To Dr. Gultekin Özsoyoğlu, you have undoubtedly been a second advisor through my academic career, and I thank you for that. Your insight, kindness, and perpetual smile have been greatly appreciated components of each day of mine, and I will thoroughly miss our frequent interactions.

There are countless others that have helped me throughout my stay here, in so many ways. As an undergraduate, I was fortunate to have a surrogate family of Bill, Harriet, Lissa, and Kayte. They immediately made me feel comfortable while I was away from home, and I will be indebted to them forever for their continual kindness.

Thanks much also to some of my newer family members — Scott, Jo, and Syb — for putting up with me lingering around your home so often when I find myself in need of a change, or simply some good company.

Also serving as family, the brothers of Phi Delta Theta deserve great thanks. The various degrees of responsibility (and irresponsibility) I developed with them has shaped me for the better, and the self-confidence I have now came as direct result of the trust we would regularly put in one another.

Serving as a true and selfless friend, Rachel Jewell is also warranted heartfelt thanks, for when help was needed she was always there.

For those that know me, you know that nearly every single day of the year I try to do something athletically. The many friends I have made during this daily ritual also are warranted thanks, for those breaks during the day are vitally important for my production and work. To the many I have swam, run, and played squash regularly with, thanks much (as my physical and
mental health is due in large part to you).

Those that I have interacted with on various other projects also deserve thanks for their great work and collaborations. In particular, all of those in the database lab have been a pleasure to work with throughout these years. You all have such great potential and I wish the best for all of you.

Finally, this document itself has been put together with the help of others, and it would not be inappropriate to list them as co-authors of this work. Z. Meral Özoysoglu has helped invaluably with guiding the direction of this research with her frequent questions and insights as to the missing components of a complete work. S. Cenk Sahinalp prompted much of my work in the search problem, and the time we spent with each other working was so intellectually stimulating it will not be forgotten by me.

Thanks to all of those mentioned above, and the countless others that have made my life so enjoyable so far. I will close this section by being bold enough to forward to you all a short piece of advice that Nani once gave me (although she may not even remember it, and yet and I often recall it), “Think Big!”

Murat Tasan
Contents

1 Multidimensional Data: The Mathematical Foundations 1
  1.1 Universes and Search Spaces . . . . . . . . . . . . . . . . . . 1
  1.2 Region Containment . . . . . . . . . . . . . . . . . . . . . . 4
  1.3 Distance Functions . . . . . . . . . . . . . . . . . . . . . . . 6
  1.4 Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2 MultidimensionalSearching 12
  2.1 Query Objects and Searching . . . . . . . . . . . . . . . . . 12
  2.2 Brute-Force Searching . . . . . . . . . . . . . . . . . . . . . 15
  2.3 Distance Computations as a Measure of Cost . . . . . . . . . . 16
  2.4 Indexing: Reducing Search Cost . . . . . . . . . . . . . . . . 17

3 General Search Space Trees 19
  3.1 Index Nodes: Regions with Structure . . . . . . . . . . . . . 19
     3.1.1 Object References . . . . . . . . . . . . . . . . . . . . . 20
     3.1.2 Nodes as Tree Elements . . . . . . . . . . . . . . . . . 21
  3.2 Search Space Tree Definition . . . . . . . . . . . . . . . . . . 22
3.3 Other Properties of Search Space Trees .......................... 23
3.4 Pruning in Search Space Trees ........................................ 29
3.5 Optimal Searching in Search Space Trees ......................... 32
  3.5.1 An Algorithm for Search Space Trees ......................... 34
  3.5.2 Range Searches are Optimal ................................. 40
  3.5.3 Bounds on Searching ........................................... 42

4 Distance-Based Indexing ................................................. 45
  4.1 Vector-Based Access Methods ..................................... 46
    4.1.1 Example Structures ........................................ 47
    4.1.2 Fixed-Length Vectors as Data ............................ 47
  4.2 Metric Trees ....................................................... 48
    4.2.1 Single-Pivot Metric Trees ............................... 49
    4.2.2 Multi-Pivot Metric Trees ............................... 50
  4.3 Matrix-Based Methods ............................................ 51
  4.4 Distance-Based Indexing on Fixed-Length Vectors ............. 52
  4.5 VP-Tree analysis .................................................. 53
    4.5.1 VP-Tree construction ..................................... 53
    4.5.2 VP-Tree pivot selection .................................. 55
    4.5.3 VP-Tree searching ......................................... 55
    4.5.4 Is storage cost ∝ performance? ......................... 56
  4.6 LAESA analysis ..................................................... 58
    4.6.1 LAESA construction (and pivot selection) ............. 59
    4.6.2 LAESA search .............................................. 59
5.4 Distance Based Indexing for Sequences via Compression Distance

5.4.1 Distance based indexing ...................................... 102
5.4.2 Updating VP trees for almost metrics ..................... 104

5.5 An alternative to compression distance: signatures .......... 106

5.6 Results on the Comparisons of the Approximations of Signatures and Compression Distances ................................................. 109

5.6.1 Generated Ordering vs True Ordering ....................... 110
5.6.2 Accuracy Tests on Generated Data .......................... 111
5.6.3 Genome Sequences and Agreement Tests .................. 112

5.7 Properties of String Spaces for Proximity Search Applications 113

5.7.1 Characteristics of Data Sets that Contain Evolutionarily-Related Members ......................................................... 114
5.7.2 Performance of VP Trees Under Polynomial and Exponential Pairwise Distance Distributions ............................... 115
5.7.3 Nearest Neighbor Search with Polynomial Distribution 119
5.7.4 Experimental Evaluation of VP Trees ....................... 120

6 Probabilistic Approaches to Distance-Based Indexing 130

6.1 probabilistic analysis of a metric tree .......................... 130

6.1.1 distance distributions .......................................... 130
6.1.2 predicting query ranges for k-nn search .................... 133
6.1.3 search events .................................................... 134
6.1.4 optimal metric trees ............................................ 144
6.2 PVP-Trees ................................................. 145
  6.2.1 PVP-Tree construction ............................. 145
  6.2.2 PVP-Tree searching ............................... 157
6.3 Interval matrices ........................................ 158
  6.3.1 Generalized matrix searches ...................... 159

7 Conclusions ................................................. 168
  7.1 Contributions ......................................... 168
Multidimensional indexing has long been an active research problem in computer science. Most solutions involve the mapping of complex data types to high-dimensional vectors of fixed length and applying either Spatial Access Methods (SAMs) or Point Access Methods (PAMs) to the vectorized data.

In more recent times, however, this approach has found its limitations. Much of the current data is either difficult to map to a fixed-length vector (such as arbitrary length strings), or maps only successfully to a very high number of dimensions. In both cases, Distance-Based Indexing serves as an attractive alternative, relying only on the pairwise distance information of data items to build indices that offer efficient similarity search retrieval.

In this work, distance-based indexing is approached first in a general
fashion, where a framework is laid out that encompasses both distance-based indexing methods as well as SAMs and PAMs. Shared properties of various seemingly unrelated data structures can be exploited, as is shown by the presentation of a single (and optimal) search algorithm that works on a variety of trees for a variety of different search types.

The motivation for distance-based indexing is then shown via an application of indexing strings (biological sequences, to be exact). By simply showing that a distance function satisfies the properties of a metric, it is illustrated that many forms of data, with various distribution characteristics can successfully be indexed with distance-based indexing.

Finally, a probabilistic approach towards indexing leads to an improved tree construction algorithm, as well as an information based search algorithm that searches the information stored in any data structure, regardless of the form (i.e. whether the structure is a tree or a matrix, the algorithm performs equally well).
Chapter 1

Multidimensional Data: The Mathematical Foundations

1.1 Universes and Search Spaces

Let the set $U$ denote the universe of all possible database objects for some particular domain. For example, a text database may be designed to store any string of any length drawn from a finite alphabet, in which case $U$ is the set of all possible strings. The size of $U$ may be unbounded (i.e. $|U| = \infty$, with $|X|$ denoting the cardinality of set $X$) — as in the case above — or bounded (e.g. the set of all social security numbers)$^1$. Let $D \subseteq U$ be a dataset of the currently stored objects in the database. $D$ represents a snapshot of the contents of a database at some point in time, and is analogous to a

---

$^1$In practical terms, all working universes must have an upper size limit — dictated by the amount of storage available.
relation instance in relational database theory.

When considering data with some underlying dimensionality, it is often convenient to look upon the universe of objects as a search space instead. The search space for some universe \( U \) is simply another set (of possibly infinite cardinality) consisting of members called locations. A more formal definition is given below.

**Definition 1 (Search Space Mapping).** Let a search space mapping be a total partial function taking on the form:

\[
\mu : U \rightarrow \mathcal{P}(S) \setminus \{\emptyset\},
\]

where \( S \) is considered to be the search space of universe \( U \) (with \( \mathcal{P}(X) \) denoting the powerset\(^2 \) of set \( X \) and \( \emptyset \) denoting the empty (or null) set).

There are two interesting properties of the codomain of mapping functions following the form in equation 1.1. First, note that the codomain does not contain the empty set, thus requiring every member of \( U \) to map to some representation in the search space. Secondly, the use of the powerset states that each object in \( U \) maps to a set of locations in \( S \), called a region. The justification for this is given in example 1.

**Example 1.** Consider the set of social security numbers:

\[
U = \{\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^9\},
\]

(where \( X^n \) is defined as the \( n \)-fold Cartesian product of set \( X \)). There are a variety of search space functions that can be considered for such a universe:

\[^2\mathcal{P}(X) = \{Y : Y \subseteq X\}.\]
Let $\mu_1$ provide a mapping from $U$ to a nine-digit sequence (vector) of natural numbers between zero and nine. This intuitive and straightforward approach provides a search space $S = U$, thus $\mu_1 : U \rightarrow \mathcal{P}(U)$.

Let $\mu_2 : U \rightarrow \mathcal{P}(\{i \in \text{naturals} : 0 \leq i \leq 81\})$ be defined such that each member of $U$ maps to a set containing a single natural number computed as the sum of each digit in the social security number (e.g. $\mu_2((5, 7, 9, 2, 4, 6, 0, 6, 9)) = \{48\}$).

Finally, an even more abstract mapping is possible. The first three digits of a social security number identifies the approximate location where the individual originated from (birthplace), such as the county, state, or possibly foreign nation. Let $\mu_3$ then be the set of all cities in the world (old and new). Then $\mu_3 : U \rightarrow \mathcal{P}(\mu_3 S)$ maps each social security number to the set of possible cities that the individual was born in.

Note that the ranges of mappings $\mu_1$ and $\mu_2$ will only contain sets of cardinality one, while $\mu_3$ illustrates the need for the powerset notation. One may still question the need for taking the powerset of the search space to be the codomain for all search space mappings however. Consider the case where the three digits $(6, 0, 9)$ identify a single city (say New York City) and the three digits $(9, 2, 9)$ specify Puerto Rico (and thus possibly any city in Puerto Rico). In such a case, the mapping function should not provide a single city as an output in one case, while providing a set of cities in another. By always returning a set, uniformity is guaranteed for all such mappings, making the use of them far simpler (e.g. no special cases are needed to distinguish
between a returned location vs a returned region).

The functions provided also illustrate the possible diversity of search spaces for some universe, and in some cases (such as the differences between $\mu_1$ and $\mu_2$) the usefulness of some mappings can easily be questioned. In fact, this idea of one mapping being “better” than another will be revisited upon in this work.

Example 1 clearly shows that very few search space mapping functions will be surjections$^3$ (since the powerset of the search space is always the codomain), and the example’s function $\mu_2$ also shows that the mappings need not be injections$^4$ either. This latter observation plays an important role later when one considers the use of mappings as defining equivalence classes on members of $U$.

1.2 Region Containment

It is often useful to be able to determine which objects in the original universe have mapped regions that are contained within some specified region in $S$. Definition 2 gives a function that formally describes this.

**Definition 2** (Region Containment). *Given a universe $U$, a search space $S$, and a set $M = \{\mu : (\mu : U \to \mathcal{P}(S))\}$ of search space mappings from $U$ to $\mathcal{P}(S)$, let the partial function $\text{contains} : \mathcal{P}(S) \times \mathcal{P}(U) \times M \to \mathcal{P}(U)$ be*

$^3$Surjections are functions whose range and codomain are identical.
$^4$One-to-one mappings.
defined as follows:

\[
\text{contains}(R, D, \mu) = \{ o \in D : \mu(o) \subseteq R \}. \tag{1.2}
\]

Equation 1.2 can be read as “the set of objects in dataset \(D\) that are contained in region \(R\) as specified by \(\mu\),” or “\(R\) contains objects \(\text{contains}(R, D, \mu)\) in \(D\), as defined by \(\mu\).”

**Example 2.** Continuing the social security example, one may request the following: “given a list of cities \(S \subseteq \mu_3\), find those individuals (i.e. social security numbers) that can be guaranteed to have been born in one of the cities in \(S\).” Letting \(U\), \(\mu_3\), and \(\mu_3\) be defined as in example 1, and letting \(D \subset U\) be the set of assigned social security numbers, the request can be answered as \(\text{contains}(S, D, \mu_3)\).

Note that as defined in equation 1.2, the region containment function may not return a finite set. Consider an infinitely large universe \(U\) (such as that of all possible strings from some finite alphabet). Letting \(S\) be a search space and \(\mu : U \to \mathcal{P}(S)\), \(|\text{contains}(S, U, \mu)| = |U| = \infty\), and thus no finite set can ever be returned from the function’s execution. In fact, this observation leads directly to lemma 1.

**Lemma 1.** Given universe \(U\), search space \(S\), and a mapping \(\mu : U \to \mathcal{P}(S)\), the following is always true:

\[
\text{contains}(S, U, \mu) = U.
\]

**Proof.** Assume the contrary, that \(\text{contains}(S, U, \mu) \neq U\), for some \(S\), \(U\), and \(\mu\). Then there are three possibilities (cases):
1. \( \exists x : x \in \text{contains}(S, U, \mu) \land x \notin U \), or

2. \( \exists y : y \in U \land y \notin \text{contains}(S, U, \mu) \), or

3. both case 1 and case 2 are true.

Case 1 must always be false. \( U \) has the greatest cardinality of all members in \( \mathcal{P}(U) \). Since \( \mathcal{P}(U) \) is the codomain of \( \text{contains} \), case 1 would imply that there is some set \( D \in \mathcal{P}(U) \) with \( x \in D \) such that \( |D| > |U| \), which is impossible.

Case 3 then must always be false on account of case 1.

Case 2 implies that there is some member of \( U \) that has no representation in \( S \). However, recall the definition of a search space mapping (definition 1). \( \mu \) must be a total partial function, thus requiring that all members of \( U \) have a valid mapping to some region in \( \mathcal{P}(S) \). Thus case 2 must always be false as well.

Since all three cases can never be true, a contradiction is achieved, and the initial hypothesis must be false, thus making the lemma true. \[\Box\]

### 1.3 Distance Functions

Given a search space defined on some universe, one can define a variety of distance functions that operate in the search space. A distance is some measure of the dissimilarity between two locations (or possibly regions, as described in this section). Thus, intuitively, distance functions will most
often return zero for the distance between two identical objects, and as the objects become more dissimilar the distance between them will increase.

While such a notion comes naturally when considering pairs of locations in a search space, one is often forced to deal with distances between regions in the search space as well. This leads to two distinct distance function forms, given in definitions 3 and 4.

**Definition 3 (Locational Distance Function).** Given some search space $S$, let a locational distance function be a total partial function of the form:

$$\delta : S \times S \to 0^+ \mathbb{R},$$

(1.3)

where $0^+ \mathbb{R}$ is the set of non-negative reals.

**Definition 4 (Regional Distance Function).** Given some search space $S$, let a regional distance function be a partial function of the form:

$$\Delta : \mathcal{P}(S) \times \mathcal{P}(S) \to 0^+ \mathbb{R}.$$  

(1.4)

Note that regional distance functions need not be total partial functions, as there may be instances where some distances can be left undefined (e.g. the distance between two empty sets).

Why specify two distance functions that are so similar? Recall that all search space mappings provide a region for each database object. Since most often one is concerned with the distance between the original database objects as they exist in some search space, the direct application of any locational distance function is not possible (sets of locations are outside the domain
of such distance functions). Instead, regional distance functions must be used. Nevertheless, most of the work done in identifying distances in search spaces is often done with locational distance functions, with regional distance functions providing an interface between universe members and search space locations.

**Example 3.** Given some universe $U$, a search space $S$, and a mapping between them $\mu : U \rightarrow \mathcal{P}(S)$, one can defined a regional distance function $\Delta : \mathcal{P}(S) \times \mathcal{P}(S) \rightarrow \mathbb{R}^{0+}$ in terms of a locational distance function $\delta : S \times S \rightarrow \mathbb{R}^{0+}$. For example, let some regional distance function $\Delta^\delta_{\text{avg}}$ be defined as follows:

$$\Delta^\delta_{\text{avg}}(R_1, R_2) = \frac{1}{|R_1 \times R_2|} \cdot \sum_{i \in R_1 \times R_2} \delta(i).$$

Thus, $\Delta^\delta_{\text{avg}}$ is the average locational distance $\delta$ between all location pairs in $(r_1, r_2) \in (R_1 \times R_2)$.

Example 3 illustrates how a regional distance function can employ the services of a locational distance function to do brunt of the work in computing actual distances. In fact, this is most often how regional and locational distance functions are used, provided they are compatible (i.e. if the regions and the locations are pulled from the same search space $S$ in both distance functions). Such regional distance functions are said to be using a locational distance function. Two regional distance functions that follow this mold are used so often, they deserve their own definitions.
Definition 5 (Minimum-Locational Regional Distance Function). Given a compatible locational distance function \( \delta : S \times S \to 0^+ \mathbb{R} \), let \( \Delta_{\text{min}}^\delta : \mathcal{P}(S) \times \mathcal{P}(S) \to 0^+ \mathbb{R} \) be the Minimum-Locational Regional Distance Function using \( \delta \), defined as:
\[
\Delta_{\text{min}}^\delta(R_1, R_2) = \inf \{ \delta(i) : i \in R_1 \times R_2 \},
\]
(1.5)
where \( \inf X \) denotes the infimum of set \( X \).

Definition 6 (Maximum-Locational Regional Distance Function). Given a compatible locational distance function \( \delta : S \times S \to 0^+ \mathbb{R} \), let \( \Delta_{\text{max}}^\delta : \mathcal{P}(S) \times \mathcal{P}(S) \to 0^+ \mathbb{R} \) be the Maximum-Locational Regional Distance Function using \( \delta \), defined as:
\[
\Delta_{\text{max}}^\delta(R_1, R_2) = \sup \{ \delta(i) : i \in R_1 \times R_2 \},
\]
(1.6)
where \( \sup X \) denotes the supremum of set \( X \).

So how to compute the distance between database objects (members of some universe \( U \))? Example 4 provides such an illustration.

Example 4. Continuing the third case in example 1, let \( U, \mu_3^S \) and \( \mu_3 \) be defined as in example 1, with the understanding that every social security number identifies a unique U.S. citizen\(^5\). Let \( \delta : \mu_3^S \times \mu_3^S \to 0^+ \mathbb{R} \) be defined as the distance (in km) between any two locations (cities) in \( \mu_3^S \) (with \( \delta \) being symmetric).

\(^5\)Let there exist another mapping from existing social security numbers to existing U.S. citizens and vice-versa (i.e. this mapping will be a bijection — both a surjection and an injection).
Now consider the question, “was Anne born closer to Bill than Curtis?”

Letting Anne’s, Bill’s, and Curtis’ social security numbers be denoted as \( o_A \), \( o_B \), and \( o_C \) respectively, the answer can be computed as:

\[
\begin{cases}
  \text{“yes”} & \text{if } \Delta_{\text{max}}^{\delta_3}(\mu(o_A), \mu(o_B)) < \Delta_{\text{min}}^{\delta_3}(\mu(o_A), \mu(o_C)), \\
  \text{“no”} & \text{if } \Delta_{\text{min}}^{\delta_3}(\mu(o_A), \mu(o_B)) > \Delta_{\text{max}}^{\delta_3}(\mu(o_A), \mu(o_C)), \\
  \text{“maybe”} & \text{otherwise.}
\end{cases}
\]

1.4 Metrics

A metric is a pair \((\Delta : \mathcal{P}(S) \times \mathcal{P}(S) \to [0^+, \mathbb{R}], S)\) such that \(\Delta\) satisfies properties (predicates) 1, 2, 3, and 4.

Property 1 (Positivity).

\[
\forall R_1, R_2 \in \mathcal{P}(S) : \Delta(R_1, R_2) \geq 0. \quad (1.7)
\]

Property 2 (Definiteness).

\[
\forall R_1, R_2 \in \mathcal{P}(S) : R_1 = R_2 \iff \Delta(R_1, R_2) = 0. \quad (1.8)
\]

Property 3 (Symmetry).

\[
\forall R_1, R_2 \in \mathcal{P}(S) : \Delta(R_1, R_2) = \Delta(R_2, R_1). \quad (1.9)
\]

Property 4 (Triangle Inequality).

\[
\forall R_1, R_2, R_3 \in \mathcal{P}(S) : \Delta(R_1, R_2) \leq \Delta(R_1, R_3) + \Delta(R_3, R_2). \quad (1.10)
\]

10
The metric properties can be used in conjunction with the properties of certain search space trees (§3) to answer queries in rapid time. It should be noted, that there are relaxations to some of the metric properties as well. For example, pseudo metrics do not guarantee property 2, as it is possible to have \( \Delta(R_1, R_2) = 0 \) for some \( R_1 \neq R_2 \). As a different example, a slight relaxation in property 4 can lead to the almost metric, a notion introduced in §5.3.7. It is assumed that all distance functions discussed in the rest of this work are either metric or are some relaxed version (with the relaxations well-defined).
Chapter 2

Multidimensional Searching

2.1 Query Objects and Searching

Mappings from a universe to region in a search space are not particularly useful, unless there is some reason for having the search space in the first place. Similarity searching employing the use of a distance function is the most common application that fits this model, and indeed all of the necessary components needed to define some common similarity search forms are now available. Some of the most well-established similarity searches are the nearest-neighbor search and the range search, both of which return a set of objects in the database that are “close” to some query object \( q \).

**Definition 7** (Nearest-Neighbor Search). Given universe \( U \), a search space \( S \), a set \( M \) of search space mappings of form \( \mu : U \to \mathcal{P}(S) \), and a set \( F \) of regional distance functions of form \( \Delta : \mathcal{P}(S) \times \mathcal{P}(S) \to \mathbb{R}^{0+} \), let \( nn : U \times \mathcal{P}(U) \times M \times F \to \mathcal{P}(U) \) be the Nearest-Neighbor Search function, defined
as:

\[
\text{nn}(q, D, \mu, \Delta) = \{ o \in D : (\forall x \in D : \Delta(\mu(q), \mu(x)) < \Delta(\mu(q), \mu(o))) \}.
\]

(2.1)

**Definition 8 (Range Search).** Using \( U, M, S, \) and \( F \) as defined in definition 7, let \( \text{nn} : U \times \mathbb{R}_+ \times \mathcal{P}(U) \times M \times F \rightarrow \mathcal{P}(U) \) be the Range Search function, defined as:

\[
\text{range}(q, r, D, \mu, \Delta) = \{ o \in D : \Delta(\mu(q), \mu(o)) \leq r \},
\]

(2.2)

where \( r \) is the range of the search.

In English, equation 2.1 simply returns a set of database objects in \( D \) that are closer to \( q \) such that no other object in \( D \) is closer. “Closeness” is defined by the regional distance function \( \Delta \), and the regions in \( S \) are found using the mapping \( \mu \). Similarly, equation 2.2 returns a set of database objects that are within the range \( r \) from \( q \).

**Example 5.** Using \( U, S, \) and \( \mu_3 \) from example 1, and \( \delta \) from example 4, consider the following request, “find those individuals in database \( D \subset U \) that definitely were born within 50km of some query \( q \in U \) person.” This simple search is answered by solving for \( \text{range}(q, 50, D, \mu_3, \Delta_{\max}^\delta) \).

Another common search is the logical extension of the Nearest-Neighbor Search to return a greater number of results, such as the “ten nearest neighbors”.

13
Definition 9 (k-Nearest-Neighbor Search). Using $U$, $S$, $M$, and $F$ from definition 8, let $\text{knn} : U \times N \times \mathcal{P}(U) \times M \times F \rightarrow \mathcal{P}(U)$ be the k-Nearest-Neighbor Search function, defined as:

$$\text{knn}(q, k, D, \mu, \Delta) = \{ o \in D : \Delta(\mu(q), \mu(o)) \leq w_k \},$$  

(2.3)

for some $1 < k \leq |D|$, where $w_k$ is defined as below.

Let the regional distance between $\mu_3(q)$ and all members of $D$ be entered into a sequence of values $V = (v_1, v_2, \ldots, v_{|V|})$ in arbitrary (i.e. possibly unsorted) order (so $|V| = |D|$, and $V$ may contain repeats). Now let $W$ be a permutation of $V$ such that $W = (w_1, w_2, \ldots, w_{|W|})$ and $w_1 \leq w_2 \leq \cdots \leq w_{|W|}$ (i.e. $W$ is a sorted version of $V$).

Note that the Nearest-Neighbor Search is then simply a special case of the $k$-Nearest-Neighbor Search, computed by setting $k = 1$. Also note that as defined, the $k$-Nearest-Neighbor Search may return a set of cardinality greater than $k$. This is due to the possibility of ties (i.e. $\Delta(\mu(q), \mu(o_1)) = \Delta(\mu(q), \mu(o_2))$ for $o_1 \neq o_2$). In many practical situations, the existence of ties is disregarded for simplicity’s sake, as then the returned set is always of fixed length $k$ (and thus can easily be packed into a pre-sized array, for example).

The $k$-Nearest-Neighbor Search and Range Search have often been treated as separate entities, although one can formulate a slightly more comprehensive query that can easily represent both forms. Such a combination permits the construction of algorithms and data structures that need only handle a single query form.
**Definition 10** (k-Nearest-Neighbor Threshold Search). Let $U$, $S$, $M$, $F$, $X$, and $Y$ be defined as in definition 9. The $k$-nearest-neighbor threshold search $\text{knnr} : U \times \mathbb{N} \times \mathbb{R}^{0+} \times \mathcal{P}(U) \times M \times F \rightarrow \mathcal{P}(U)$ is then defined as:

$$\text{knnr}(q, k, r, D, \mu, \Delta) = \{ o \in D : \Delta(\mu(q), \mu(o)) \leq y_k \land \Delta(\mu(q), \mu(o)) \leq r \},$$

(2.4)

for some $1 < k \leq |D|$.

Note that a $k$-Nearest-Neighbor Threshold Search may return a set with cardinality less than $k$ (or even a set with cardinality greater than $k$), possibly making it more difficult to handle in some programming paradigms. Nevertheless, definition 10 allows the other three search types (definitions 7, 8, and 9) to be described by a single, more-powerful, function. A $k$-Nearest Neighbor Search can be performed by setting $r = \infty$ in equation 2.4, and similarly a Range Search is attainable by setting $k = |D|$. The usefulness of this single definition will be revealed in §3.5.1, where only a single search algorithm need be designed to handle the varying search forms.

### 2.2 Brute-Force Searching

Many possible strategies can be employed to answer the queries presented in §2.1. Immediately obvious is the naïve approach, which is simply to compute $\Delta(\mu(q), \mu(x))$ for all $x \in D$. This technique is commonly called the *brute-force* method, and it barely needs mentioning that this method is rarely ideal. On the other hand, a non-trivial observation about certain high-dimensional
data sets shows that the brute-force approach is the best one can hope for. This result is discussed further in §6.1.1, where properties of distance distributions are covered.

2.3 Distance Computations as a Measure of Cost

It is often the case in search spaces that the distance function $\Delta$ is quite complex\(^1\). Typical cases of this are character edit distances (such as the Levenshtein Edit Distance [57]), high-order $L^p$ norms, the Hausdorff Measure [44], and others. In fact, some distance measures — such as the Block Edit Distance [60] — are NP-Hard in their computation. Such distance functions render brute-force searching in large datasets intractable, as $|D|$ computations are then required to answer the query.

Consider the case where an alternative method for answering a query is presented that can provide the answer in $\log |D|$ computations. Clearly, one would select this technique over the brute-force approach. This intuitive decision leads directly to a method for comparing search techniques: the number of distance computations needed to answer queries. Throughout this work, search structures and algorithms are directly compared by posing identical queries over them and measuring the number of distance computations.

\(^1\)Actually, it is more often the underlying locational distance function $\delta$ that determines the true complexity, however if $\Delta$ is defined in terms of $\delta$, $\Delta$ must be at least as complex as $\delta$. 

16
computations needed to acquire the query results.

Using such a measure can indeed prove to be controversial, as in many cases constant factors (such as overhead costs not related to distance computations themselves) can lead to search methods that, while providing fewer distance computations than others, actually taking longer in “real time” (i.e. seconds needed to answer query). Nevertheless, clever algorithm improvements and shortcuts made by adept programmers can over time make these constant factors negligibility small. Such tricks cannot, however, reduce the number of computations, and thus this remains the unit of measurement for search method comparison. A large collection of previous works [17, 88, 27] adopt this measurement strategy.

In many data structures (such as the ubiquitous B-Tree family [29]), the cost of measurement is the number of pages accessed during search. Note that if the data items needed for computations are wisely packed on these pages, the two measures are nearly identical (i.e. if \( k \) items are stored on each page, and some constant fraction of each page accessed is required for distance computations, then the number of distance computations performed scales linearly with the number of pages accessed).

## 2.4 Indexing: Reducing Search Cost

Since the brute-force approach is not feasible for many data sets and complex distance functions, one must find better ways to access the data for searches. A *multidimensional index* built on a dataset \( D \) is a structure (with associated
search algorithm(s)) that attempts to allow for queries to be answered in fewer than $|D|$ computations.

As a simple example, consider an index in the back of a textbook. When trying to find information on a particular topic, one can either read the entire text and find all pages that cover the topic (the brute-force method), or use the index to jump directly to the appropriate pages. A table of contents, such as that appearing at the front of this work, is another example of a common index used daily by nearly all. Similarly, Google [38] is an index built over the collection of web pages on the Internet. Each page is associated with a group of terms, and pages are ranked, allowing users to rapidly find the best pages of interest without needing to crawl through the incredibly complex and large web.

In this work, indices are considered for multidimensional search, and they are evaluated via the average number of distance computations needed to answer typical\textsuperscript{2} queries (§2.3). The brute-force method will often be seen as an upper bound on search performance for all indices\textsuperscript{3}, and in many cases it will be observed that the lower bound can be precisely defined as well (via analyses of the properties of the structures and search algorithms).

\textsuperscript{2}Typical queries will be accurately defined when considering distance distributions in §6.1.2.

\textsuperscript{3}In actuality, very poor search scenarios can lead to performance worse than brute-force, however this is quite rare and requires either (a) a terrible worst-case scenario, or (b) a horribly designed search structure/algorithm.
Chapter 3

General Search Space Trees

Search trees are a particular type of index that can be built over a single search space $S$. This restriction greatly simplifies the following discussions, although it is technically not necessary, as a series of mappings can be used to convert a set of different search spaces into a single one. Thus, for the remainder of this chapter, one is concerned with some index $T$ over some search space $S$.

3.1 Index Nodes: Regions with Structure

Search space trees, much like all other trees in computer science, are comprised of nodes, and thus the discussion appropriately begins with these structures. Indices\(^1\) will first be presented as sets of nodes, and for ease a single

---

\(^1\)The plural of “index” is given as both “indexes” and “indices” by most common dictionaries [1]. However, in this work, “indices” will be used as the plural form, while
Definition 11 (Node Regions). Each node \( n \) in some search space tree \( T \) is associated directly with a single region in \( S \), which can be found via the total partial function \( \text{region} : T \rightarrow \mathcal{P}(S) \).

3.1.1 Object References

Each node also can carry a set of references (pointers) to objects in some original universe.

Definition 12 (Indexes). Let the total partial function \( \text{indexes} : T \rightarrow \mathcal{P}(U^*) \) be defined such that \( \text{indexes}(n) \) returns the set of all object references stored by node \( n \). Here, \( U^* \) represents the set of all possible objects in all database domains (i.e. the union of all database domains). More formally, the following predicate must hold:

\[
\forall o \in U^*, \forall n \in T : o \in \text{indexes}(n) \implies \exists \mu : o \in \text{contains}(\text{region}(n), U^*, \mu).
\]

(3.1)

When there is a reference and the predicate in equation 3.1 holds, then it is said that “\( n \) indexes \( o \) under \( \mu \).”

The need for such an abstract definition and condition lies in the subtlety of the differences between the object universe and a search space. Recall that indices are built over search spaces, rather than the original universes. Thus, two drastically different universes \( U_1 \) and \( U_2 \) (with \( U_1 \cap U_2 = \emptyset \)) can employ

“indexes” is reserved to represent the verb form of the word, meaning “to index”.

20
the same index $T$ built over search space $S$ provided there exists mappings from both $U_1$ and $U_2$ to $\mathcal{P}(S)$. This notion is rather unintuitive, and rightly so. Most often, an index is built over a search space such that the object references stored at the nodes refer to a single dataset drawn from a single domain. Nevertheless, it is interesting to note that such “index sharing” is possible, although acquiring good performance is then considerably more difficult.

When considering a dataset $D$ of objects, it is said that “$T$ indexes $D$ under $\mu$” when the following predicate holds:

$$\forall o \in D, \exists n \in T : o \in \text{indexes}(n).$$  

(3.2)

3.1.2 Nodes as Tree Elements

As presented, a set $T$ of nodes is not particularly useful as an index. Since object references are present, one can certainly go from a node to the original data, but without some greater organizational structure, such a collection is simply a waste of space. This additional structure is obtained by associating each node $n$ with another node, called the parent of $n$.

**Definition 13** (Parent). *The partial function $\text{parent} : T \to T \cup \{\text{null}\}$ is defined such that $\text{parent}(n)$ is the parent of $n$. Each node can thus have exactly one parent.*

If node $n$ has $\text{parent}(n) = \text{null}$, then $n$ is said to be a root node.
3.2 Search Space Tree Definition

Section 3.1 simply considered sets of nodes as indices. The indices are most often arranged into a search space tree however, which is a set \( T \) of nodes that satisfies all of the following properties and their corresponding predicates.

**Property 5** (Single Root).

\[
|\{n \in T : \text{parent}(n) = \text{null}\}| = 1. \tag{3.3}
\]

Thus, \( T \) must contain exactly one root. Let this node be identified as \( \text{root}(T) \).

**Property 6** (Null-Parented Root).

\[
\text{parent}(\text{root}(T)) = \text{null}. \tag{3.4}
\]

**Property 7** (All In The Family).

\[
\forall n \in T : n \neq \text{root}(T) \implies \text{parent}(n) \in T. \tag{3.5}
\]

Each non-root node in \( T \) must have its parent be non-null and in \( T \) as well. Thus, all trees are disjoint from each other.

**Property 8** (Root Region is Search Space).

\[
\text{region}(\text{root}(T)) = S. \tag{3.6}
\]

The region for the root of \( T \) must be the entire search space \( S \) over which \( T \) is built. While the reason for this may not be clear at first, it is a very useful property for all trees \( T \) to have, as it greatly simplifies the search algorithm that is presented in §3.5.1.
Property 9 (Hierarchical Region Containment).

\[ \forall n \in T : n \neq \text{root}(T) \implies \text{region}(n) \subseteq \text{region}(\text{parent}(n)). \] (3.7)

The region of each node \( n \) must be a subset of the region of the parent of \( n \). This property is extremely important, and is the principle that serves as the basis for tree pruning, a technique which most search algorithms discussed in this work rely heavily upon.

Sets of nodes that satisfy properties 5, 6, 7, 8, and 9 are search space trees. The vast majority of existing multidimensional search structures supporting the similarity searches presented in §2.1 are trees that fall into this framework.

3.3 Other Properties of Search Space Trees

Search space trees (as defined in §3.2) may also satisfy certain other properties (predicates). These properties allow for a classification method for trees, which aids in identifying the appropriate modifications to construction and search algorithms over such trees (e.g. algorithmic “tricks” that take advantage of one of these properties can hold for all trees that satisfy that property). Identifying shared attributes between various trees can also aid in identifying those structures that are most appropriately employed based on external properties (such as some data distribution characteristic). Prior to the actual properties however, will be a collection of definitions and lemmas that will prove to be useful in later discussion.
Definition 14 (Leaf Nodes). Given search space tree $T$, let $\text{leaves}(T)$ be a function defined as:

$$\text{leaves}(T) = \{l \in T : (\neg \exists n \in T : \text{parent}(n) = l)\}. \quad (3.8)$$

Thus, leaf nodes cannot be parents of any other node in the tree.

Definition 15 (Internal Nodes). Given search space tree $T$, let $\text{internals}(T)$ be a function defined as:

$$\text{internals}(T) = T \setminus \text{leaves}(T). \quad (3.9)$$

Definition 16 (Ancestors). Given a search space tree $T$, let the sequence-returning function $\text{ancestors}$ be defined as $\text{ancestors}(n) = (a_1, a_2, \ldots, a_d)$, where $a_1 = \text{parent}(n)$, $a_{i+1} = \text{parent}(a_i)$, for all nodes $a_i \neq \text{root}(T)^2$. If $a \in \text{ancestors}(n)$, it is said that “$a$ is an ancestor of $n$”. $\text{ancestors}(\text{root}(T)) = () = \emptyset$.

Lemma 2 (All Nodes Share a Common Ancestor). Given a search space tree $T$, and any two nodes $n_1, n_2 \in T \setminus \text{root}(T)$, it must be true that $\text{ancestors}(n_1) \cap \text{ancestors}(n_2) \neq \emptyset$.

Proof. By the definition of a tree given in §3.2, the final element of $\text{ancestors}(n) = \text{root}(T)$ for all nodes $n \in T \setminus \text{root}(T)$. Thus at the very least, any two nodes must share this single common ancestor. □

---

2It can easily be made that $\text{ancestors}(\text{root}(T)) = (\text{null})$, however, without loss of any generality, it is easier to trim this null character from the ancestor sequence of each node.
Definition 17 (Node Depth). Let the depth of a node \( n \in T \) be \( |\text{ancestors}(n)| \) (i.e. the length of the ancestor sequence of \( n \)). Thus, \( \text{depth}(\text{root}(T)) = 0 \).

Definition 18 (Siblings). Given search space tree \( T \), let \( \text{siblings} : T \to \mathcal{P}(T) \) be a total partial function defined as:

\[
\text{siblings}(n) = \{ s \in T : \text{parent}(s) = \text{parent}(n) \}. \tag{3.10}
\]

Thus, siblings are nodes that share a common parent.

Lemma 3 (Siblings Share All Ancestors). Given search space tree \( T \) and any two sibling nodes \( n_1, n_2 \in T \), it is true that \( \text{ancestors}(n_1) = \text{ancestors}(n_2) \).

Proof. Follows directly from definitions 16 and 18. \qed

Definition 19 (Children Nodes). The dual to a parent of a node in a search space tree \( T \) is the set of children of a node. Let \( \text{children} : T \to \mathcal{P}(T) \) be a total partial function defined as:

\[
\text{children}(p) = \{ c \in T : \text{parent}(c) = p \}. \tag{3.11}
\]

Clearly, \( \text{children}(p) = \emptyset \) for any \( p \in \text{leaves}(T) \).

Definition 20 (Lowest Common Ancestor). Given a search space tree \( T \), let the lowest common ancestor of a set of nodes \( N \subseteq T \setminus \text{root}(T) \) be some node \( a \) defined as follows. Let \( A = \bigcap_{n \in N} \text{ancestors}(n) \) (i.e. the shared ancestors of all nodes in \( N \)).

No two nodes in \( A \) can have identical depths. To see this, assume there is a pair of same-depth nodes \( a_1, a_2 \in A \). They must be along different paths.
to the root, meaning any children of $a_1$ cannot have $a_2$ as an ancestor, and vice-versa. By definition 16, this would imply that $a_1, a_2 \notin A$, causing a contradiction.

$a$ can then be defined as the node in $A$ with the greatest depth (i.e. $\forall b \in A \setminus \{a\} : \text{depth}(a) > \text{depth}(b)$).

**Definition 21** (Descendants). Given a search space tree $T$, let the total partial function $\text{descendants} : T \rightarrow \mathcal{P}(T)$ be defined recursively as:

$$\text{descendants}(n) = \{c \in \text{children}(n)\} \cup \bigcup_{c \in \text{children}(n)} \text{descendants}(c),$$

with $\text{descendants}(l) = \emptyset$ for all $l \in \text{leaves}(T)$.

Given all of these helping definitions and lemmas, some interesting properties that search space trees may possess are now described.

**Property 10** (Space-Covering). A tree $T$ is considered space-covering on $D$ under $\mu$ when the following predicate holds:

$$\forall p \in T : \bigcup_{c \in \text{children}(p)} \text{region}(c) = \text{region}(p). \quad (3.12)$$

Property 10 states that the union of the regions of all sibling children of a node must equal the region of the parent node. This property leads directly to lemma 4.

**Lemma 4** (Leaf Union Equals Root). Given a space-covering tree $T$ built over search space $S$ with some $\mu : U \rightarrow \mathcal{P}(S)$, the following predicate holds:

$$\bigcup_{l \in \text{leaves}(T)} \text{region}(l) = \text{region}(	ext{root}(T)) = S. \quad (3.13)$$

26
Proof. Follows directly from properties 10, 9, and 8. \qed

**Corollary 1** (Leaf Union Covers Search Space). Lemma 4 implies the following predicate also holds:

\[ \forall o \in U, \exists L \in \mathcal{P}(\text{leaves}(T)) : o \in \text{contains}\left(\bigcup_{l \in L} \text{region}(l), U, \mu\right). \quad (3.14) \]

Proof. Follows directly from lemma 4 and property 8. \qed

While seemingly trivial at first, lemma 4 (and more directly corollary 1) provides a useful insight to search behavior when analyzing the minimum possible depths of search paths in §3.5.3.

Property 11 concerns overlapping regions, or more precisely, the lack of overlapping regions.

**Property 11** (Disjoint Siblings and Trees). A node \( n \) in search space tree \( T \) is considered to have disjoint siblings if the intersection of the regions of \( \text{children}(n) \) is disjoint. Tree \( T \) is called disjoint if all of its nodes have disjoint siblings, or more formally when the following predicate holds:

\[ \forall n \in T : \bigcap_{c \in \text{children}(n)} \text{region}(c) = \emptyset. \quad (3.15) \]

**Lemma 5** (Disjoint Leaves Cannot Cover Same Objects). Given a disjoint search space tree \( T \) built over \( S \), with some \( \mu : U \rightarrow \mathcal{P}(S) \), the following predicate holds:

\[ \forall o \in U : ((\exists l_1 \in \text{leaves}(T) : o \in \text{contains}(\text{region}(l_1), U, \mu)) \implies \]

\[ (\neg \exists l_2 \in \text{leaves}(T) \setminus \{l_1\} : o \in \text{contains}(\text{region}(l_2), U, \mu))). \quad (3.16) \]
Proof. Consider any two leaf nodes \( l_1 \) and \( l_2 \) in \( T \).

If \( l_1 \) and \( l_2 \) are siblings, then the proof is trivial, as equation 3.15 is immediately applied to show that the regions of \( l_1 \) and \( l_2 \) cannot both cover the same region.

If \( l_1 \) and \( l_2 \) are not siblings, then let \( a \) be the lowest common ancestor of \( l_1 \) and \( l_2 \). \( l_1, l_2 \notin \text{children}(a) \), since \( l_1 \) and \( l_2 \) are not siblings. However, there must exist two sibling children of \( a \), \( a_{l_1} \) and \( a_{l_2} \), such that \( a_{l_1} \) is an ancestor of \( l_1 \) but not \( l_2 \), and \( a_{l_2} \) is an ancestor of \( l_2 \) but not \( l_1 \). By property 11, the regions of \( a_{l_1} \) and \( a_{l_2} \) cannot both cover the same region. Applying property 9 then shows that the regions of \( l_1 \) and \( l_2 \) cannot both cover the same region either. \( \square \)

**Corollary 2 (Disjoint Leaves Cannot Index Same Object).** Given a disjoint search space tree \( T \), the following predicate holds:

\[
\forall o, \forall n_1 \in \text{leaves}(T), \forall n_2 \in \text{leaves}(T) \setminus \{n_1\} : o \in \text{indexes}(n_1) \implies o \notin \text{indexes}(n_2). \quad (3.17)
\]

**Proof.** Follows directly from lemma 5 and definition 12. \( \square \)

Again, lemma 5 and corollary 2 seem trivial, however they imply a useful property about bounds on tree storage requirements, as discussed in §3.5.3.

Both properties 10 and 11 imply interesting results that relate all nodes at some given depth \( k \) in a tree \( T \).
3.4 Pruning in Search Space Trees

The word “search” in search space trees indicates that the primary purpose of such structures is to assist in answering queries (such as those discussed in §2.1). The various properties of search space trees already covered can be exploited to answer queries quickly, primarily by eliminating subtrees rooted at some nodes in the tree.

To begin, first consider some search space tree $T$ that indexes $D$ under $\mu$. Any of the possible queries can be answered by simply performing a tree walk (any full tree walking algorithm will do), and at each node $n$ visited, computing $\Delta(\mu(q), \mu(x))$ for all $x \in \text{indexes}(n)$. This is similar to a brute-force method in that every data object indexed by all nodes will be used in the distance computation. If each data item in $D$ is indexed exactly once in $T$, this leads to $|D|$ distance computations, thus a full tree walk can do no better than a brute-force search.

Now consider the possibility that during the course of a tree walk some node $u$ can remain unvisited, with the guarantee that no data object indexed by $u$ is in the answer set. This will clearly lead to fewer distance computations, and is a desirable situation. Consider further the possibility that when such a node $u$ is identified, no descendant of $u$ needs to be visited either (with the same guarantee). This notion is described more formally by the following constructs.

**Definition 22 (q-region).** Given some query object $q \in U$, a search space mapping $\mu : U \rightarrow \mathcal{P}(S)$, a regional distance function $\Delta : \mathcal{P}(S) \times \mathcal{P}(S) \rightarrow \mathbb{R}$,
and a threshold $r \in \mathbb{R}_{0+}$, let the $q$-region of $q$ with radius $r$ be defined by the function:

$$q_{\text{region}}(q, r, \mu, \Delta) = \{x \in S : \Delta(\mu(q), \{x\}) \leq r\}.$$  (3.18)

Definition 22 is quite similar to definition 8, although here the function returns a region in $S$, as opposed to objects in some data set $D$. Note that in definition 22, if the mapping $\mu$ always returns singleton sets in $S$, query regions will be “balls” in space$^3$, centered at $\mu(q)$.

**Definition 23** (Prune Value). Given some query object $q \in U$, a region $R \subseteq S$, a search space mapping $\mu : U \rightarrow \mathcal{P}(S)$, and a regional distance function $\Delta : \mathcal{P}(S) \times \mathcal{P}(S) \rightarrow \mathbb{R}_{0+}$, let the prune value of region $R$ given $q$, $\mu$, and $\Delta$ be defined as:

$$\text{prune}(q, R, \mu, \Delta) = \begin{cases} 
\inf \{r \in \mathbb{R}_{0+} : q_{\text{region}}(q, r, \mu, \Delta) \cap R \neq \emptyset\} & \text{if } \mu(q) \cap R = \emptyset, \\
0 & \text{if } \mu(q) \cap R \neq \emptyset.
\end{cases}$$  (3.19)

The prune value of region $R$ given $q$, $\mu$, and $\Delta$ is thus the smallest radius $r$ defining the $q$-region such that $R$ and the $q$-region intersect. This fundamental notion is at the heart of all search trees, and allows for nodes within a tree to be *pruned.*

---

$^3$This actually depends on the distance function used, although in most situations (such as edit distances and $L_p$ norms).
Lemma 6 (Node Pruning). Given a search space tree $T$ that indexes dataset $D$ under $\mu$, and some range search $\text{range}(q, r, D, \mu, \Delta)$, consider a node $n \in T$. The following predicate holds:

$$(r < \text{prune}(q, \text{region}(n), \mu, \Delta)) \implies \left( \forall o \in \text{indexes}(n) : o \notin \text{range}(q, r, D, \mu, \Delta) \right).$$

(3.20)

Proof. Follows directly from definitions 23, 22, and 12. □

Corollary 3 (Subtree Pruning). Lemma 6 leads directly to the predicate:

$$(r < \text{prune}(q, \text{region}(n), \mu, \Delta)) \implies \left( \forall d \in \text{descendants}(n), \forall o \in \text{indexes}(d) : o \notin \text{range}(q, r, D, \mu, \Delta) \right).$$

(3.21)

Proof. Follows directly from lemma 6 and property 9. □

Theorem 1 (Rooted Subtree Pruning). Given a search space tree $T$ that indexes dataset $D$ under $\mu$, and some range search $\text{range}(q, r, D, \mu, \Delta)$, consider a node $n \in T$. The following predicate holds:

$$(r < \text{prune}(q, \text{region}(n), \mu, \Delta)) \implies \left( \forall d \in \text{descendants}(n) \cup \{n\}, \forall o \in \text{indexes}(d) : o \notin \text{range}(q, r, D, \mu, \Delta) \right).$$

(3.22)

Proof. Follows directly from combination of the consequences in the implications in equations 3.20 and 3.21. □

Theorem 1 permits the elimination of entire subtrees (a node $n$ and all of $\text{descendants}(n)$) from a search space tree when the antecedent in equation 3.22 is true. Note that equation 3.22 is an implication, not a logical
equivalence (i.e. an “if”, not an “iff”). It is not necessarily true that if all data objects indexed by a node \( n \) or its descendants are not in the set \( \text{range}(q, r, D, \mu, \Delta) \), that the subtree rooted at \( n \) can be pruned from a search. Such situations in fact arise quite often, where one is forced to search a node in a tree even though no object indexed by the subtree rooted at that node is in the answer set of the query.

One may note that theorem 1 is provided with reference to only one of the search types, the range search (definition 8). It will be shown in §3.5.2 that all other search types can be searched utilizing the same principle as that seen with range searches, without loss of generality.

The notion of pruning nodes from trees and eliminating entire collections of data objects from consideration during a search is mathematically equivalent to defining equivalence classes on such regions. Prune values of regions (nodes) effectively classify all items indexed at a rooted subtree into a single equivalence class that can be pruned as a single entity. This is a common idea that holds in all search space trees.

### 3.5 Optimal Searching in Search Space Trees

The pruning principles described in §3.4 provide the foundation upon which a search algorithm can now be discussed.
Algorithm 1: knnr-search

**Input:** query object $q \in U$, answer list size $k \in \mathbb{N}$, threshold $r \in 0^+\mathbb{R}$, search space tree $T$ indexing $D \subseteq U$ under $\mu : U \rightarrow \mathcal{P}(S)$, distance function $\Delta : \mathcal{P}(S) \times \mathcal{P}(S) \rightarrow 0^+\mathbb{R}$

**Variables:** priority queues $P$ and $V$ (lower values have priority), objects $o$ and $x$ (holds nodes or data objects), distance $d$

**Output:** answer set $\text{knnr}(q, k, r, D, \mu, \Delta)$ (entries in $V$)

begin

clear ($P$); clear ($V$); $x \leftarrow \text{null}$; $o \leftarrow \text{root}(T)$; $d \leftarrow 0$;

while size ($V$) $< k$ and $d \leq r$ and size ($P$) $> 0$ do

if $o \in T$ then

foreach $x \in \text{indexes}(o)$ do

push ($P$, $x$, prune($q$, $\mu(x)$, $\mu$, $\Delta$));

endforeach

foreach $x \in \text{children}(o)$ do

push ($P$, $x$, prune($q$, region($x$), $\mu$, $\Delta$));

endforeach

else $o \in D$

push ($V$, $o$, $d$);

end

end

/* in case of tie in $P$ between data items and nodes, data items have priority */

d $\leftarrow$ lastPoppedValue ($P$);

/* get the value in $P$ of the last item popped from */

return $V$
3.5.1 An Algorithm for Search Space Trees

Algorithm 1 operates by starting with a $q$-region of radius 0 and continues to grow the search radius by the smallest increments needed to continue the search. Essentially, the current search radius is always stored in the variable $d$, which is set at the start of the algorithm to be zero. Provided the priority queue $P$ is not empty, the result queue $V$ is not full (i.e. $V$ has fewer than $k$ entries), and that the next queue entry to be examined has a prune value $d \leq r$ (where $r$ is the search threshold), the popped queue entry $o$ is expanded.

Expansion starts by considering the type of the popped queue entry $o$. If $o$ is a tree node, all data items indexed by $o$ are inserted into the priority queue with their prune values. Note that for these actual data objects, computing the prune value requires a full distance computation, and thus the number of times line 6 executes is a lower bound on the cost of the search. All children of $o$ are also inserted into the priority queue as well with their prune values. Computing the prune values of children nodes may require distance computations, although it will be shown that most wisely constructed trees utilize some properties (in particular property 4) to avoid this. If $o$ is an actual data object, it is inserted directly into the result queue $V$ with the actual distance (computed as the prune value in line 6).

Handling $k$-th Nearest Neighbor Ties

It will be assumed that if one asks a $k$-Nearest-Neighbor Search, no more than $k$ data items will be in the answer set. If all distances between a
query’s region and those in the dataset are distinct, then this is always the case anyhow. When there are ties however (i.e. two objects $o_1$ and $o_2$ such that $\Delta(\mu(q), \mu(o_1)) = \Delta(\mu(q), \mu(o_2))$, values can be chosen for $r$ and $k$ such that one of two situations must take place. In the first scenario, the answer set can contain more than $k$ elements by including any objects in $D$ that can tie to be the $k$-th nearest neighbor of $q$. In the second scenario, the answer set can contain at most $k$ elements by simply arbitrarily (randomly) including only enough of those objects tied for the $k$-th nearest neighbor such that the answer set does not exceed $k$ in cardinality. For simplicity’s sake, this latter option is chosen by algorithm 1. Adopting the former strategy would only require trivial modifications to the algorithm, but add unneeded complexity to the presentation. For further simplicity, it will be assumed for the remainder of this work that there are no $k$-th nearest neighbor ties (i.e. that the distance from a query to its $k$-th nearest neighbor is distinct, for all queries and all $k$). This assumption greatly simplifies the presentation, and once again does not impact the results as without loss of generality the case with ties can be added to each scenario without incurring any increase in time/space complexity of the strategies given here.

**Algorithm Properties**

**Lemma 7** (algorithm 1 is online). Algorithm 1 is online in the sense that the result queue $V$ is built in increasing order of distances (prune values). More formally, assume that $(v_1, v_2, \ldots, v_k)$ is the sequence of values associated with the entries pushed into $V$. It always holds that $v_{i+1} \geq v_i$. 

35
Proof. $d$ is initialized to zero at the start of the algorithm. $d$ is always updated by taking the value associated with the last popped entry in $P$. Since $P$ is a priority queue giving precedence to lesser values, $d$ can never get smaller provided the prune values in $P$ are never less than zero, which follows directly from equation 3.19.

The advantage of having an online algorithm should be immediately clear. Most begin examining the results of a similarity search with the first nearest-neighbor, then the second, and so on. Even when the search may perform slowly in “real” time, provided the results are given to the user at least as fast as the user can process them, an online algorithm may be feasible where an online form of the algorithm operating on the same data structure with identical queries may be impractical.

**Lemma 8** (algorithm 1 is sound). *Algorithm 1 is sound in that it is guaranteed to return no incorrect answer.*

Proof. To prove that algorithm 1 is sound, it is sufficient to be able to disprove the existence of a scenario where an incorrect answer is returned. Prior to that however, one must define a correct answer. Returned queue $V$ is correct if each element in $V$ is in $\text{knnr}(q, k, r, D, \mu, \Delta)$, and the value of each element $x$ in $V$ is the actual distance $\Delta(\mu(q), \mu(x))$. Note that in the case of ties at the $k$-th nearest neighbor, any correct answer suffices, although there may be more than one possible answer. This follows directly from the assumption made regarding the non-existence of $k$-th nearest neighbor ties. To make the proof simpler however, we can assume that there is exactly one
correct answer, and without loss of generality the proof will remain applicable (because we could just exhaustively apply the remaining arguments to each one of the correct answers instead).

An answer can be incorrect if:

1. \( V \setminus knnr(q, k, r, D, \mu, \Delta) \neq 0 \) or \( knnr(q, k, r, D, \mu, \Delta) \setminus V \neq 0 \), or
2. one of the elements in \( V \) is associated with an incorrect distance, or
3. both of the above situations arises.

Case 1 is disproved via contradiction. Assume that there exists some data item \( x \) that induces case 1. Then this element must exist in either \( V \) or \( knnr(q, k, r, D, \mu, \Delta) \), but not both.

If \( x \in V \) but not in \( knnr(q, k, r, D, \mu, \Delta) \) then it is not a real answer (and is thus a false positive).

- Assume that the main loop terminated because \( d > r \). Then the last item entered into \( V \) must have prune value \( d' \) less than \( r \) (permitting the previous iteration of the main loop). Since the algorithm is online (lemma 7), it follows inductively that any previous entry must have had prune value less than \( d' \). Thus this scenario is impossible.

- Assume that the main loop terminated because \( |V| = k \). It follows directly from the previous argument that all entries must have had their prune values be less than \( r \). Thus the violating element must be the \( k \)-th (and final) entry \( x \). If \( x \) is a false positive, it must then violate
the threshold. This violation would have been caught at the start of
the main loop at the previous iteration, however. Thus this scenario is
impossible.

- Assume finally that the main loop terminated because $|P| = 0$. By
  the previous two arguments, all items entered in $V$ must be correct,
  and thus $k$ must be set to its maximum of $|D|$. In this case, $D =
  \text{knnr}(q, k, r, D, \mu, \Delta)$, in which case the violating entry $x$
  is not in $D$. Thus this scenario is also impossible.

Now if $x \in \text{knnr}(q, k, r, D, \mu, \Delta)$ but not in $V$, then a real answer is
missing (a false negative). This implies that $x$ is either in $P$, or the node
$n$ that indexes $x$ is in $P$, or an ancestor of $n$ is in $P$. In either case, if $x$
is a false negative, there must exist some entry in $V$ with prune value greater
than $x$’s prune value.

- Let $x$ be still in $P$ with prune value (distance) $p$. The prune value of
  the $k$-th nearest neighbor in $V$ then must be less than $x$ (by lemma
7). Thus no entry in $V$ has prune value greater than $p$, causing a
contradiction.

- Let $n$ or some ancestor of $n$ be still in $P$ with prune value $p$. By direct
  application of theorem 1, the prune value of $x$ will be no less than $p$.
  Thus if $x$ is a false negative, there must exist some entry in $V$ with
  prune value greater than $p$. By the previous argument (considering
  when $x$ is still in $P$), this causes a contradiction.
Case 1 is thus proven. Case 2 is far simpler to prove, as one needs only show that the prune value for actual dataset entries in $P$ is identical to the computed distance. For any data object $x \in P$, the prune value is computed as $\text{prune}(q, \mu(x), \mu, \Delta)$. Expanding the nested functions in equation 3.19 leads directly to $\text{prune}(q, \mu(x), \mu, \Delta) = \Delta(\mu(q), \mu(x))$. Case 2 is thus proven. Case 3 is also thus proven, and the lemma is proved.

**Lemma 9** (algorithm 1 is complete). *Algorithm 1 is complete in that it is guaranteed to return some answer set for all executions (even if that answer is empty).*

*Proof.* The algorithm will always proceed to the start of the main loop. In this loop, exactly one queue pop will occur each time. Thus the execution of the loop is guaranteed to terminate if the maximum number of items than can ever be entered into the queue is finitely upper bounded. Since a tree has a finite number of nodes, and each node has a finite number of indexed data objects, it thus follows that the main loop must terminate, and $V$ is returned (even if empty).

As an extreme case, consider when $k = 0$. Here, the loop will never begin, however $V$ has already been initialized and will be returned empty (which is a correct result for $k = 0$).

**Theorem 2** (algorithm 1 is sound and complete).

*Proof.* Follows directly from lemmas 8 and 9.
3.5.2 Range Searches are Optimal

Algorithm 1 is useful beyond its soundness, completeness, and its online property. It is also optimal in that the fewest possible number of distance computations are made to guarantee soundness.

**Lemma 10** (Range Searches are Optimal). *Using algorithm 1 to answer a query range*(q,r,D,µ,∆) = knnr(q,|D|,r,D,µ,∆) *on tree T indexing D under µ results in the fewest number of expanded nodes (and thus the fewest number of distance computations) needed to guarantee soundness, and thus algorithm 1 is optimal if distance computations performed.*

*Proof.* In order to guarantee the correct answer is returned by knnr(q,|D|,r,D,µ,∆), one must examine all data objects indexed by all nodes in T that intersect with q-region(q,r,µ,∆). To see this, consider some node n ∈ T that is not examined during search, but does intersect with the q-region (i.e. that region(n) ∩ q-region(q,r,µ,∆) ≠ ∅). Let this intersection be represented as region R. Now let there be some o ∈ knnr(q,|D|,r,D,µ,∆) be inserted into T such that o ∈ indexes(n) that will not be entered in the priority queue P. Since all entries in V must go through P first, o will be missing from V, leading to a false negative. This has already shown to be impossible (lemma 8).

Thus, only those nodes with prune value greater than r can be left unexamined, while all others must be examined. To be more precise, *exactly* those nodes with prune value greater than r are left unexamined, meaning the nodes that are examined *must* be examined, no more, no less. This provides
a lower bound on the number of distance computations, as the examination of a node $n$ leads to direct distance computations for all $o \in \text{indexes}(n)$. The result then follows that in order to provide a sound answer set, exactly the minimum number of data objects are used for distance computations.

Note that the cost for each examined node $n$ is $|\text{indexes}(n)|$, ignoring the cost of computing the prune value for all of $\text{children}(n)$. Some search structures will incur distance computations for these steps (by using pivots, a notion discussed in depth in §4.2). One will see later in this work however, that each pivot can simply be included in $\text{indexes}(n)$, thus removing this additional cost completely.

Lemma 10 is a powerful result showing that algorithm 1 is optimal for range searches. It is now shown that the cost of a $k$-Nearest-Neighbor Search is identical to that of a range search if one is given the distance to the $k$-th nearest neighbor ahead of time.

**Lemma 11** ($k$-Nearest-Neighbor Searches Termination). If algorithm 1 is used for some search $\text{knn}(q, k, D, \mu, \Delta) = \text{knnr}(q, k, \infty, D, \mu, \Delta)$, the value of $d$ at the start of the last iteration of the main loop is equal to $\Delta(\mu(q), \mu(x))$, where $x$ is the $k$-th nearest neighbor of $q$.

**Proof.** Follows directly from lemma 7.

**Corollary 4** ($k$-th nearest neighbor distance oracle). If one knows the distance $d$ to the $k$-th nearest neighbor $x$ of some object $q$ ahead of time (say via an oracle), then the cost of answering $\text{knnr}(q, k, \infty, D, \mu, \Delta)$ is the same as the cost of answering $\text{range}(q, |D|, d, D, \mu, \Delta)$. 

41
Proof. Follows directly from lemma 11 and theorem 2. □

Theorem 3 ($k$-Nearest-Neighbor Searches with Threshold are Optimal). Using algorithm 1 to answer a query $knnr(q, k, r, D, \mu, \Delta)$ on tree $T$ indexing $D$ under $\mu$ results in the fewest number of expanded nodes (and thus the fewest number of distance computations) needed to guarantee soundness, and thus algorithm 1 is optimal in distance computations performed.

Proof. Follows directly from corollary 4 and lemma 10. □

Theorem 3 is an important results that allows algorithm 1 to be used for all search space trees. Each implementation needs only differ by their methods for computation of prune values, which is closely related to the structure characteristics and is part of each tree’s definition.

### 3.5.3 Bounds on Searching

When executing algorithm 1 on some tree $T$, one can color each node that is expanded. Since the root node is always expanded by default prior to the start of the main loop, it will always be colored (as all searches must at the very least consider the root). Other nodes are colored as they are popped from the priority queue in the main loop of the algorithm (line 11). Now consider the colored tree at the termination of the algorithm.

**Definition 24** (Search Paths). A search path is an $l$-length path (sequence) $P = (p_1, p_2, \ldots, p_l)$ of nodes in a search space tree $T$. Let $p_1 = \text{root}(T)$ for all paths. $\text{parent}(p_{i+1}) = p_i$ for all nodes in a search path. The terminal node of an $l$-length search path $P$ is $p_l$. 

42
The number of terminal nodes in a colored tree after algorithm execution is precisely the number of search paths in the tree.

**Lemma 12** (Termination Nodes in Space-Covering Trees are Leaves). Consider the coloring of a space-covering tree $T$ after a search $knnr(q, k, r, D, \mu, \Delta)$ using algorithm 1. Let $N$ be the set of termination nodes. Then $N \subseteq \text{leaves}(T)$.

**Proof.** It is sufficient to show that no internal nodes in tree $T$ can be termination nodes. This will be shown by contradiction. Let $n \in \text{internals}(T)$ be some terminal node a search path $P$. Then $n$ was colored (expanded) during algorithm execution. For this expansion to occur, it must have been true that $r \geq \text{prune}(q, \text{region}(n), \mu, \Delta)$. By equation 3.19 then, $q \cdot \text{region}(q, r, \mu, \Delta) \cap \text{region}(r) \neq \emptyset$. By equation 3.12 then, there must exist some child node $c \in \text{children}(n)$ such that $r \geq \text{prune}(q, \text{region}(c), \mu, \Delta) = \text{prune}(q, \text{region}(n), \mu, \Delta)$, and this node must have been entered into the priority queue.

Now, assuming distinct $k$-th nearest neighbor distances (§3.5.1), if $n$ is indeed a terminal node, the algorithm has terminated with $c$ still in the priority queue. It is clear (from above) that the main loop can not have terminated from a threshold violation, as $\text{prune}(q, \text{region}(c), \mu, \Delta) \leq r$ is still in the priority queue. The algorithm cannot have terminated from an empty priority queue, since $c$ is still in the queue. Finally, the algorithm cannot have terminated from a full answer queue, for if $n$ was expanded with prune value $\text{prune}(q, \text{region}(n), \mu, \Delta)$, it must have been at the top of the priority queue at a loop iteration when the answer queue was not yet full.
Since \( \text{prune}(q, \text{region}(c), \mu, \Delta) = \text{prune}(q, \text{region}(n), \mu, \Delta) \), \( c \) must also be at the top of the queue, and thus no other object can have been popped prior to \( c \) to fill the answer queue.

\[ \square \]

**Corollary 5** (Lower Bound on Search Performance of Space-Covering Trees).

The fewest number of nodes expanded during execution of algorithm 1 on a space-covering tree \( T \) is equal to \( \min_{l \in \text{leaves}(T)} \text{depth}(l) + 1 \).

**Proof.** Follows directly from lemma 12.

\[ \square \]

An interesting consequence of lemma 12 is that insertions are always possible without reorganization. Consider inserting a new data object \( x \) into \( D \) that is indexed by \( T \) under \( \mu \). One can simply perform search \( \text{knnr}(x, 1, 0, D, \mu, \Delta) \). While no answer will be found (the returned queue will be empty, following the distinct \( k \)-th nearest neighbor assumption), each of the colored termination nodes (and thus also any ancestor of these termination nodes) are candidates for indexing \( x \).

If a tree \( T \) is not space-covering however, search paths may terminate at internal nodes. While this is desirable often during search as it decreases the lower bound on performance, it does make handling dynamic data structures (that accept insertions and deletions of data items) considerably more difficult because of the insertion problem just described (i.e. in non-space-covering trees, node regions may have to grow to accommodate new insertions, and should then shrink upon deletions to regain the benefits of possible internal terminal nodes during search).
Chapter 4

Distance-Based Indexing

*Distance-based indexing* is a general name for multidimensional access methods that are built to efficiently answer similarity searches (§2.1), but without the dependence on a coordinate system. With such methods, a distance can be defined over the original data objects, as provided that distance function is a metric, the relative distances is the only information needed for index construction and search. Mathematically, this is similar to dimensionality reduction, however rather than reducing objects themselves to fixed-length vectors, here it is the pairwise distances between all objects in space that becomes the single dimension worked with.

The most common forms of distance-based indexing are *metric trees*, which are search space trees (§3). There is another class of methods that work on matrices of pairwise distances, and these are intuitively called *matrix-based* in this work.

Chapter 3 described a class of search structures in a very general and
abstract way. A primary structure discussed in this work that falls into this class is called a metric tree, of which there are many particular forms. In fact, a further characterization of the properties of different metric trees will be done in this chapter, allowing a sub-classification system for these types of search space trees.

4.1 Vector-Based Access Methods

Prior to the discussion of metric trees, it is worth describing a group of multidimensional indices that are the most dominant form of tree that falls into the classification of search space trees. These will be described here as vector-based access methods, however in the literature a variety of alternative names are used, including point access methods, spatial access methods, and others. The reason for the alternative name used here is that each of these methods relies on have some notion of a coordinate system. Formally, regions and locations in the search space are described by using values in fixed-length vectors. Thus $n$-dimensional rectangles require specifying the precise locations of $n$ corners. Similarly, a $n$-dimensional sphere requires knowledge of the center of that sphere (via the $n$ coordinates) as well as the radius. For many forms of data, such methods are quite efficient, and some are very briefly described in this section. In all cases, algorithm 1 can be employed, however the computation of the prune values for regions can become quite convoluted, especially when the shapes of regions is non-standard throughout the structure (as is the case with SR-Trees, described in §4.1.1).
4.1.1 Example Structures

Examples of the partitioning are the intersection hyperplanes present with \( \text{kd-Trees} \) [80, 11, 12], the minimum bounding rectangles seen in the \( \text{R-Tree} \) family [40, 67, 53, 10, 14, 78], and the pattern-driven space decompositions exemplified by \( \text{Z-orderings} \) [64]. Many other such examples exist, and the inquiring reader can find much of it in a thorough survey written by Günther and Gaede [36]. Since these methods all rely on a fixed and well-defined notion of dimensionality, they will be referred to as vector-based indices for the remainder of the work.

4.1.2 Fixed-Length Vectors as Data

While the mentioned vector-based indices are quite efficient for some forms of data, the primary limitation is that they require all data to map to a fixed dimensional space. Moreover, these structures tend to operate best when the number of dimensions in the space is rather small. Dimensionality reduction techniques can be used to assist with this problem (which is a function of the \textit{curse of dimensionality}, discussed in §4.4). Such methods include FastMap [34] and multidimensional scaling [83].

Nevertheless, if the data does not map efficiently to a fixed-length vector space, dimensionality reduction techniques can only add to the lossiness of the transformed data objects. Consider indexing arbitrarily lengthed strings, for example. While some mapping techniques have been developed that are constrictive (i.e. they prevent false negatives during search), worst-case scenarios
are easily generated to illustrate a high number of false positives [52]. Thus, utilizing vector-based indices can only be seen as a filtering step, with all positive hits returned by the method requiring filtering afterwards. Note that this does not permit algorithm 1 to operate for \( k \)-nearest neighbor searches, although it still operates fine for range searches.

Besides strings, other forms of data follow the mold of being difficult to map to fixed dimensions. Audio and visual streams, financial data (also streaming), images data, arbitrary texts, and streaming communication data are typical examples of data types that are dealt with daily by modern systems. The limited applicability of vector-based methods on such data has driven the need for alternative indexing methods that can handle complex data types.

### 4.2 Metric Trees

Metric trees are just like any other search space tree in that regions are defined (which create equivalence classes). Unlike vector-based methods however, these regions have no coordinate information. Instead, regions are defined relative to *pivots*, which are specially chosen objects in the data set. The general idea of a pivot can be summed up as, “what the space looks like from my point of view”, and in particular, “how far away from me is every other object (or region) in the space?” Some metric trees are discussed here, although for a more comprehensive survey, one should consult more comprehensive surveys that are available [42, 25].
4.2.1 Single-Pivot Metric Trees

Single-pivot metric trees are identified by having a exactly one pivot associated with each internal node in the tree. Given some node and its region then, children nodes can be defined based on relative distances (or shells) to the pivot.

One of the earliest examples of such a method was introduced by Burkhard and Keller [20], who considered spaces with a distance function that returned only integer values. Given a space, one data item is picked as the “pivot”, while all of the other data items then sat on concentric rings corresponding to the distance between the items and the pivot (the center of the rings). The process is recursively applied to each ring, creating a tree structure. Some more recent works extend upon this idea and are likewise focused on discrete-valued distance functions [7, 6, 24]

The continuous version of Burkhard and Keller’s tree is theoretically described in the work, but little research was done on such structures until Uhlmann presented his work on metric trees (which incidentally is the first work known to the author to coin the phrase “metric tree”) [84]. In that short — yet influential — work, both single-pivot and multi-pivot (§4.2.2) are described at a high level. The first, and simplest, type is the single-pivot tree, which was independently introduced by Yianilos and called the Vantage-Point Tree (VP-Tree) [88]. Rather than placing data items onto rings corresponding to distances from a single pivot (called a vantage point in VP-Trees), they are instead placed into subsets corresponding to disjoint
intervals based on their distance to the pivot. (A more detailed description of this is provided in section 4.5.1.) VP-Trees have been further modified in more recent works in attempts to improve search performance by constructing forests of disjoint trees to reducing branching events in search paths [89, 90], as well as using two pivots at each node that index the same region [16]. Some recent modifications to expand the structure to a dynamic form have also been performed [26]. A greater discussion of VP-Trees is given later §4.5, as a part of this work is a probabilistic construction algorithm for such trees.

One of the immediately clear properties of VP-Trees is that they are disjoint trees. This means that each data object is indexed often exactly once in the tree, providing an upper bound on the storage requirements. The simplest forms for VP-Trees are also space-covering, although this is not required, and in practice most do not hold this property in an attempt to improve search performance.

4.2.2 Multi-Pivot Metric Trees

Unlike single-pivot metric trees, those with multiple pivots at each node operate more akin to clustering methods, and subdivide the space not by the shells around a single pivot, but instead by spheres centered at each pivot. Uhlmann’s version of this is called a Generalized-Hyperplane Tree (GH-Tree), which simply picks two objects as pivots at each node and divides the space into two children nodes based on which pivot each object is closer to. GH-
Trees are very closely related to Bisector Trees [22, 63].

The Generalized Near-Neighbor Access Tree (GNAT) [17] uses (possibly) more than two pivots in an attempt to preserve the geometry of the data in the space. The subdivisions at each node would be performed similarly to that in the GH-Tree (each data item is associated with a subset such that every item in the set is closer to one particular pivot than any other pivot), although flexibility in the number of pivots at each node was introduced in hopes of gaining tighter and more logical clusters. Unfortunately, the construction cost for such a structure deemed its use impractical.

The M-Tree [27] removed some of the flexibility of the GNAT but preserved the general idea of using many pivots (called routing objects by the authors) and following the GH-Tree strategy. The primary difference with M-Trees is that they are built bottom-up rather than top-down. This enables M-Trees to handle dynamic data sets (through reorganization methods akin to those found in B-Trees [9, 29]).

### 4.3 Matrix-Based Methods

Rather than using a hierarchical space decomposition to index the space, one can also ignore the construction of equivalence classes and instead simply store a collection of interesting pairwise distances between data items in the space. The resulting structures are often stored as sparse matrices, offering direct access to any precomputed distance.

Triangle inequality is still used during the search, but rather than prun-
ing subspaces from the space, individual data items are instead eliminated from consideration. This increase in granularity may come at the expense of increased storage requirements. The Approximating and Eliminating Search Algorithm (AESA) [68], the Linear AESA (LAESA) [61], and the Fixed Queries Array [24] are examples of such matrix-based solutions. (A more detailed description of LAESA will be provided in section 4.6.)

4.4 Distance-Based Indexing on Fixed-Length Vectors

While the field of indexing methods has evolved to overcome the problems associated with vector-based methods, it should be immediately clear that vector-based data can certainly be indexed via these alternative methods (provided the distances used are still metrics). In fact, it has been shown for both M-Trees [27] and MVP-Trees [16] that when dealing with high-dimensional vectors, distance-based indexing can outperform traditional vector-based solutions. Thus, it is possible that distance-based methods tend to suffer from the curse of dimensionality at a later (higher dimensional threshold) than some vector-based indices. The curse of dimensionality is the observation that at high dimensions, the difference between the farthest neighbor and the nearest neighbor of arbitrary dimensions is negligible. Put another way, all data objects seem about as far from each other as all others. Some theoretical insights on this phenomenon can be found in recent
works [15].

4.5 VP-Tree analysis

4.5.1 VP-Tree construction

The binary VP-Tree is one of the simplest examples of a Metric Tree (as single pivot solutions allows for easier analysis). Construction proceeds by picking a pivot (vantage point) \( v \) from the current working set \( R \) of database items and computing the distance between \( v \) and all remaining items in the set (the method of choosing each pivot is described in section 4.5.2). The median \( m \) of these computed distances is then found:

\[
m = \text{median}\{d(v, x) : x \in R \setminus \{v\}\}.
\]

A balanced tree structure can be formed by recursively calling the construction procedure on the two subsets:

\[
R_l = \{x \in R \setminus \{v\} : d(x, v) < m\} \quad \text{and},
\]
\[
R_r = \{x \in R \setminus \{v\} : d(x, v) \geq m\}.
\]

Note that while the original VP-Trees are restricted to having exactly two children, the construction process can be easily modified to create trees with any \( l > 2 \) children (by sorting the computed distances and partitioning the sorted list into \( l \) sublists). Such trees are often called Multi-Way VP-Trees, and it has been shown that search performance does not improve much with such a modification. A further exploration of the consequences of increasing \( l \) is discussed though in section 4.5.4.
An elementary form of the VP-Tree requires only \( l - 1 \) values be stored at each internal node (as the division lines separating the children). This version of the tree is space-covering (any possible future data item will fall into some region), and any search that progresses to an internal node in such a tree must progress into at least one of the leaf descendants of that node. Such trees will be called Loose VP-Trees (and represent the \( \text{vp-tree} \) in the original Yianilos work), for a reason to be clear immediately. Beyond the memory needed to store the structure of the tree itself, Loose VP-Trees require \( \approx \frac{(nl - n)}{l} < n \) stored values (where \( n \) is the size of the full dataset).

Rather than simply storing the \( l - 1 \) values separating the children nodes from each other, an internal node can instead store \( 2l \) distances. For each child of an internal node, both the least and the greatest distances between the node’s pivot and items indexed by the child can be stored, tightening the subspace. Search paths in such trees can terminate at internal nodes, unlike Loose VP-Trees. The resulting trees are called Tight VP-Trees, and require \( \approx 2n \) stored distances.

Regardless of the tree type, note how the distance between any internal node’s pivot and all data items indexed under that node must be computed (in order to facilitate the partitioning). The results of these computations can be stored, creating for each data item an ancestral history of distances to all pivots above it. Trees that store the full ancestral distance history of each data item are called Tight-Historical VP-Trees, and require \( \Theta(n \log_l n) \) stored distances. (Yianilos calls these \( \text{vp}^*\)-trees in the original work.)
4.5.2 VP-Tree pivot selection

Obviously the selection of one pivot over another at each step of the construction will have an impact on the final tree. It has been shown [88] that a non-trivial improvement (in query performance) can be made by selecting some pivots over others. The ultimate conclusion drawn in that study is that pivots should be chosen from “the corners of the space”, which unfortunately doesn’t mean much in spaces with no notion of dimensionality. These observations did however eventually lead to the following pivot selection strategy.

For each internal node, a group of candidate pivots is chosen at random. The distances to the other items in the space is computed and the candidate pivot providing the greatest second moment (variance) about the median distance is the ultimately selected pivot. An analysis of this technique is provided by Yianilos, but the analysis exists only for binary VP-Trees (the extension to higher fanout levels is not directly clear) with unimodal distance distributions (relative to the pivots). Nevertheless, this heuristic has remained the best such strategy for single-pivot indices to date.

4.5.3 VP-Tree searching

A DFS (depth-first search) is most commonly employed to search VP-Trees (and indeed all Metric Trees). Once the search process has proceeded to an internal node, the distance between the query object and the node’s pivot item is computed. If that pivot item is part of the answer set (i.e. within the current query radius), it is included. Each child is then examined, and if
the child can be pruned from the search (using the triangle inequality), the next child is examined. If any child cannot be pruned from the search, that child is then recursively searched.

As previously mentioned, this strategy should be modified (for all Metric Trees) to adopt the optimal search algorithm that is basically a best-first tree search. The memory required during algorithm execution will increase (as it requires the use of a priority queue to keep track of unexplored nodes), but relative to the cost of the index alone this rarely is prohibitive.

It should also be mentioned here why Tight-Historical VP-Trees are often used. They have the added ability (over Metric Trees without “memory”) to offer possible pruning of an internal node’s subspace relative to the pivot of any ancestor node. If pruning of a node is not achievable relative to the parent’s pivot then, the algorithm can simply continue walking up the tree, attempting to prune the region relative to each successive ancestor’s pivot.

4.5.4 Is storage cost $\propto$ performance?

A resounding question that is brought up with Tight-Historical VP-Trees is whether or not the (often significantly) increased storage makes a large impact on query performance. To the authors’ knowledge, no published study has previously been released measuring the gains in query performance relative to the increase in required storage. Figure 4.1 presents the results of experiments comparing the average query performance of two VP-Trees whose storage requirements grow on different orders (based on database size). As
Figure 4.1: Plots showing the average performance and storage of binary VP-Trees; the left and right plots correspond to 9d and 16d image data sets, respectively. Solid lines show average required storage (as marked on the right Y-axes). Dashed lines represent the average performance (as marked on the left Y-axes) for 1-nn queries (with no threshold). Open-point lines indicate Tight-Historical VP-Trees, while the closed points indicate Tight VP-Trees. Note how the difference in query performance remains relatively small when compared to the difference in required storage.

expected, the $O(n \log n)$ storage (Tight-Historical VP-Tree) index outperforms the $O(n)$ storage (Tight VP-Tree) index, although the performance gains do not scale at all with the increased memory consumption. In all experiments performed (including range queries that are not illustrated in figure 4.1), Tight-Historical VP-Trees offered no more than a 20% decrease in average query response cost over Tight VP-Trees.

If ample memory is available then, one certainly would gain in performance using Tight-Historical VP-Trees over those without stored ancestral
history. While the $O(n \log n)$ storage cost may seem prohibitive at first, the true cost for these trees declines with increased fanout at each internal node (which increases the base of the logarithm in the cost figure). Figure 4.2 shows that the average query performance remains relatively constant as the fanout is increased, thus reducing the storage requirements. Thus, provided the query performance does not suffer any noticeable (or intolerable) hindrance, increasing the fanout can be used as a strategy to allow employment of Tight-Historical VP-Trees even when memory restrictions exist. For the image and random data used throughout the experiments, the first noticeable performance drop occurred for fanout levels of five and greater.

For very large data sets still though, any base of the logarithm in the $O(n \log n)$ storage cost figure will still be too large. In these cases, $O(n)$-storage VP-Trees (Loose and Tight) can be used and still offer query performance that rivals that of Tight-Historical VP-Trees.

### 4.6 LAESA analysis

An alternative to using Metric Trees for distance-based indexing is to use matrix-based methods. The LAESA is such a method that opts to forego space-partitioning in favor of storing some (hopefully) useful precise distances.
4.6.1 LAESA construction (and pivot selection)

If the database is represented as a set $P$ of items, the matrix produced by the construction step of the LAESA contains all of the distances between each element of a base set $B \subseteq P$ and every other element of $P$. That is, the matrix contains all distances $d(b, p)$ such that $b \in B$ and $p \in P \setminus \{b\}$. The only parameter the construction method requires then is the size $m$ of the base set to create.

Construction begins by selecting an arbitrary element from $P$ to be the first base set item (the first pivot). The distance between this pivot and all other data items is computed and stored in the matrix. Whichever data item is furthest from the first pivot is selected as the second pivot, allowing the second set of distances to be computed (all items against the second pivot) and put into the matrix. The process continues by selecting as the $i$-th pivot the (not previously selected) item that is cumulatively furthest from the previous $i - 1$ pivots (this requires a running accumulator for each item). Once $m$ pivots have been selected, the construction is complete and has taken $O(mn)$ time and space.

4.6.2 LAESA search

The LAESA was introduced specifically in the context of nearest neighbor (1-nn) queries, although the search algorithm is trivially modified to handle general $k$-nn searches with a threshold.

Searching the matrix begins by computing the distance between the query
item and one of the base set members. Using the triangle inequality, a lower
bound on the distance between every other item in \( P \) and the query item is
computed and stored in a lower bounds vector \( G \). At this point, database
items may be eliminated by comparing the lower bound of each item with
the current threshold \( d^* \) (which is fixed for radius queries, and is strictly
decreasing by being equal to the current best \( k \)-th nearest neighbor for \( k\)-
nn queries). Any item in the set \( P \setminus B \) with a corresponding value in \( G \)
lower than \( d^* \) can be eliminated promptly from the search. Members of
\( B \) satisfying such criteria however, may or may not be eliminated. This
exception will be covered in section 4.6.3. For the remainder of this section,
it will be assumed for simplicity that no base set member can be eliminated
via triangle inequality (i.e. they must be compared with the query).

After the previous round of elimination has taken place, the next base
set member for comparison is selected as the one with the least lower bound
(as stored in \( G \)). The distance between this item and the query is once
again computed, the result of which can be used to update \( G \) (if some newly
computed lower bound is greater than a previous value in \( G \) for some database
item), while possibly also decreasing \( d^* \). Elimination of each remaining data
item in \( P \setminus B \) with a value in \( G \) less than \( d^* \) again takes place, and the loop
is repeated until all base set members have been exhausted.

When no members of \( B \) are left, the remaining database items are ordered
by their \( G \) values (with the least lower bound first) and compared with the
query. After each comparison, no further updating of \( G \) is possible (since the
remaining items store no information about their relative distances to other

60
items, i.e. they are not pivots). Nevertheless, further elimination is possible since each distance computation may update $d^*$, and after each round of elimination the loop continues with the next remaining data item in the sorted list. Once all database items have either been examined (via direct distance computation with the query item) or eliminated, the algorithm is complete.

4.6.3 Base set member elimination and LAESA tuning

Besides the query parameters themselves (i.e. the query item, $k$, and $\epsilon$) the search procedure takes a value called the elimination condition value ($ec$), which is typically an integer. This value is used to determine if base set members should be eliminated prior to being compared with the query during the search.

Members of $B$ can be eliminated only if the number of previously examined items ($nc$) is greater than $m/ec$. The primary disadvantage with eliminating base set elements is that they immediately become unavailable for use to further update the remaining items’ lower bounds. On the other hand, the advantage to eliminating such items is to potentially reduce the total cost of the search (especially if the eliminated base set members would not make any significant updates to $G$). Note that if $ec = 1$, every base set member will be examined, creating a lower bound of $m$ on the possible performance cost of each search. On the other hand, if $ec = \infty$, it is possible for only a single element of $B$ to be picked.
The inclusion of such a parameter forces the need to tune the LAESA matrix for data sets. In particular, \( m \) must correspond to the \( ec \) value that will be used during the search. For \( ec = 1 \) searches, it is desirable to have a lower \( m \) than with \( ec = 2 \) searches, for example. Moreover, choosing the “optimal” \( m \) requires examining the data set as well (as it will vary quite a bit between data sets, even with a fixed \( ec \) strategy).

It is shown in the original LAESA work [61], and is verified by experiments done in conjunction with this work, that the average search performances for varying \( ec \) values when tuned to their optimal \( m \) do not deviate far from each other. This implies that selecting the \( ec = 1 \) strategy will offer performance that is as competitive as any other \( ec \) value, while requiring less storage and construction cost (a lower \( m \)). For this reason, most of the remaining experiments on the LAESA adopt this \( ec = 1 \) search strategy.

4.6.4 LAESA performance

One of the greatest advantages of the LAESA (and its predecessor the AESA) is that they offer nearly constant-time performance for nearest-neighbor queries as the dataset size grows. This is a unique result that is not generally seen in any Metric Tree. Figure 4.3 illustrates the nearly constant performance, and a similar phenomenon can be seen in figure 4.4 (by focusing directly on the LAESA performance).
4.7 Space/performance tradeoffs

It has already been observed with VP-Trees (in section 4.5.4) that the storage space to search performance ratio is hardly one-to-one. Since one of the easiest criticisms to use against matrix indices is the possibly high storage costs, it is natural to extend the mentioned cost ratio analysis to the LAESA. This will begin to provide the insight needed in making an informed decision on when to choose one indexing solution over another.

4.7.1 Optimal $m$ a function of dimensionality

As mentioned, the LAESA requires $O(nm)$ stored distance computations, and a dataset-specific $m$ must be found ahead of time. Experimental results indicate that the optimal $m$ for a dataset remains relatively constant as the number of items grow (provided the growth follows the same distribution in space). Rather than the database size, it is apparent that the intrinsic entropy of the database is a better indicator of the optimal $m$. The is best observed when tuning the LAESA for datasets of i.i.d. (identically and independently distributed) vectors in an $L^p$-space. As the dimensionality increases, the optimal $m$ increases (indicating that as the dataset increasingly suffers from the Curse of Dimensionality [15, 41], the ability to perform eliminations from the triangle inequality in general drops).

This property of the LAESA allows for $m$-tuning via sampling techniques prior to full index construction. A test index can be built on a subset of the full dataset, and tuning can rapidly be done on that subset. Even so, the
need to progress through any procedure at all to select $m$ may be seen as a drawback to structures like VP-Trees, where the storage can be accurately estimated from the dataset prior to any testing.

4.7.2 LAESA vs VP-Trees

In order to adequately compare performance of VP-Trees and LAESA matrices, the memory consumption for each structure (as measured by stored exact distance computations) must be as close to each other as possible. The results of such experiments can be observed in figure 4.4, where Tight-Historical VP-Trees with $l = 2$ are compared with LAESA methods (where $m = 10$ is chosen to mirror the memory consumption of the VP-Tree as closely as possible). The plots illustrate that provided with such memory, the performance of Tight-Historical VP-Trees — while performing better than those without history — do not rival that of the LAESA. Note especially in figure 4.4 that the $m$ is chosen due to memory considerations, while experiments shown in figure 4.3 use a near-optimal $m$. In particular the 9-dimensional random vector dataset experiment use $m = 10$ while the optimal $m$ is 35 for $ec = 1$ searches. The performance for such an optimally tuned LAESA is indeed even better than the that displayed.

Such results, combined with the observation of nearly constant search performance with increasing dataset size, certainly adds to the argument that if ample memory is available to select a “good” $m$ (it may not even need to be optimal, as illustrated in figure 4.4), the LAESA certainly is an
attractive alternative to Metric Trees in general.
Figure 4.2: Plot showing average 1-nn query performance (dashed lines and
left Y-axis) for Tight-Historical VP-Trees with various fanout levels. Storage
requirements are indicated by solid lines (and the right Y-axis). Note that
the query performance for each tree remains very similar while the number
of stored computations for each vary. The same 9d image data set is used
across each different tree.
Figure 4.3: Average LAESA 1-NN query performance for three different datasets, with each LAESA matrix tuned to a “good” $m$. Note how performance remains nearly constant for increasing database size. The growth of the database is illustrated by the curved line entitled “brute force” (depicting the upper bound on search cost).
Figure 4.4: Average 1-NN query performance and storage for both Tight-Historical VP-Trees and LAESA matrices. Solid lines (and the right Y-axis) indicate storage while dashed lines (and the left Y-axis) indicate performance. Note how given enough memory to match that of a Tight-Historical VP-Tree the LAESA significantly outperforms the former. In particular, when applied over a random i.i.d. vector space (the right plot), the difference is quite drastic.
Chapter 5

Distance-Based Indexing for String Proximity Searching

5.1 Introduction

In many database applications, including those in computational genomics and proteomics, text and audio processing, and computational finance, it is common to have proximity queries; asking for data items that are similar to a query item (i.e. near neighbor search), or are most similar to a query item (i.e. nearest neighbor search). String similarity search is one of the most important such applications, considering that it is at the heart of computational genomics where similarity between biological sequences are of key importance. The majority of the work here deals with indexing for such strings (biosequences), although the methods described are applicable to all forms of string data – and some observations even extend to general search
techniques that are independent of the application domain.

5.1.1 Biosequence Searching

Biological sequence data is at the heart of modern biochemistry and genomics research. The rapid growth of sequence databases has driven the need for improved search methods, as sequence similarity searches are the primary method for locating and accessing sequence information. Below are some examples of applications where efficient similarity searching is key.

- Sequence similarity often implies functional homology and evolutionary relationships. It is possible to improve the understanding of the workings of newly identified sequences by searching for similar sequences whose functions have already been documented.

- Identification of mechanisms for structural rearrangement in the human genome – particularly genomic duplications – is important to understanding the causes of several genomic diseases; this requires genome wide sequence similarity search and discovery.

- Complex methods involving genome sequences such as (i) approximation algorithms [86] and popular heuristics [82] for the multiple sequence alignment problem, and (ii) distance based phylogenetic tree construction methods [31] for analyzing the evolutionary history of genomic sequences, require iterative identification of the closest pair of sequences. (Large scale pairwise sequence comparison is the main bottleneck in both methods.)
• Genome sequencing itself (especially whole-genome shotgun assemblies [33]) requires discovery of pairs of sequences which have long suffix/prefix similarity.

In order to facilitate searching for such applications, two key issues must be addressed. First, a distance function which captures the notion of sequence similarity for the desired application must be identified. Secondly, an efficient data structure/indexing technique which works well with the chosen distance function must be employed.

5.1.2 Sequence Distance Functions

Sequences are simply strings (often of nucleotides or amino acids pulled from a limited alphabet), and are thus compared using edit distance techniques. Traditionally, such comparisons were done with respect to SNPs (single nucleotide polymorphisms) almost exclusively. Recently however, the notion of segmental modifications in the genome have been applied to measuring sequence similarity.

Character edit distances are sufficiently powerful for measuring dissimilarity between strings that have diverged via SNPs only. It has been observed that many genomes (including human) consist of many segments which are repeated within a small (< 10%) divergence rate (i.e. small normalized edit distance) [5]. Within the genome, pairs of sufficiently long repeat segments (> 100bp) are more likely to be results of genome duplication than chance.

\(^1\)Consider for example a 3·10^9bp (which is approximately the size of the human genome)
Thus long sequences with high similarity are very likely to have a common ancestor in their evolutionary history.

Recent studies predict that more that 60% of the human genome is duplicated [49]; i.e. more than 60% of the nucleotides in the human genome are located in 1Kb segments which have duplicates within a divergence rate of less than 30%. Most of these duplications are common repeat segments [50] which are usually short and tend to repeat themselves several thousands of times throughout the genome (for example, the well known \textit{alu} segment consists of approximately 300 nucleotides and is repeated along the genome for more than $10^6$ times within a divergence rate of 5% – 15%). For capturing similarities between these short common repeats, distance functions which only allow single nucleotide (character) edit operations are sufficiently powerful.

As the human genome project gets close to completion, it is becoming apparent that the genome consists of much longer segments comprising portions of genes and even entire gene segments which are duplicated with very high similarity (low divergence) [49]. Some of those long duplicative segments are known to cause recurrent \textit{structural rearrangements} (i.e. edit operations long sequence, generated by an i.i.d. random source over the four letter DNA alphabet. The probability that two specific 100bp long substrings have a hamming distance of $\leq 5\%$ is $\leq 2^{-150}$ and the probability that at least one pair of 100bp long substrings with hamming distance of $\leq 5\%$ exists in the whole genome is $\leq 2^{-75}$ (practically nil). Work has also been done on estimating the probabilities of alignments that involve insertions and deletions in addition to replacements [55].
on whole segments/blocks) in chromosomes, such as segmental deletions, translocations, and reversals, as well as local duplications. A main mechanism that causes such rearrangements (particularly segmental deletions and repeats) is unequal crossover during cell replication [76]. Other mechanisms that cause such structural rearrangements include replication slippage and retrotransposition [50]. These segmental rearrangements may occur hierarchically and in multiple layers. (For example, a long block of tandemly repeated DNA may have a complete duplicate elsewhere; this may indicate that the tandem duplication event is evolutionarily older than the long segmental duplication.) Some of these rearrangements occur frequently, and may result in the deletion or duplication of a developmentally important gene segment, causing a genetic defect [51]. Such defects (often called genomic disorders) occur at a rate of approximately 1 in every 1000 births. Several birth defects are a result of these disorders, including spina bifida and cleft lip/cleft palate, as well as a number of adult diseases such as cardiovascular disease and osteoporosis. Sequence search tools that capture block operations may help improve methods for identification, diagnosis and, indirectly, treatment of such genomic disorders.

Character Edit Distances Applied to Biosequences

There are several commonly used methods to measure the dissimilarity (i.e. distance) between pairs of genome sequences. If the goal is to capture the evolutionary relationships between “short” sequences on which only point mutations have been applied (under “neutral hypothesis”), one can use character
edit distances such as the the Levenshtein edit distance [57] (or its weighted forms [73, 77], which can be defined as the minimum number of character replacements, insertions and deletions to transform one sequence to the other). These measurements have been used for decades and are still the most widely used methods for general string comparison. Because mutations are believed to occur at a fixed rate per nucleotide per year, higher similarity score between such sequences usually implies a shorter evolutionary distance.

Fastest known algorithms for computing any character edit distance between two sequences are dynamic programming based and run in time quadratic with the sequence size [73]. To compute the near neighbors (or the nearest neighbor) of a query sequence among \( n \) sequences, each of length \( O(l) \), one can compute the character edit distance between every sequence in the database and the query (i.e. the brute force method). This sequential approach takes \( O(l^2 \cdot n) \) time however, and is not efficient for large sequence databases (hence the need for improved access methods on this type of data).

**Block Edit Distances Applied to Biosequences**

While the character edit distances are well known and commonly used in many string-based applications, block edit distances have emerged as attractive alternatives to the norm for some uses. The transformation distance of Varre et. al. [85] and its closely related companion, the compression distance of Li et. al. [58, 30, 59] are recent examples that illustrate the power of such a measure. Rather than focusing on single character edit operations, block edit distances deal with long sequences that have been subject to a broader
set of alterations that include block operations (such as block copies, translo-
cations, deletions, and reversals). As previously mentioned, such operations
have direct analogues as evolutionary operations that often occur during cell
replication.

The most general block edit distance can be defined as the minimum num-
ber of these block operations and character edits to transform one sequence
to another. Block edit distances, in particular the compression distance, pro-
vide practical upper bounds to the algorithmic complexity among strings (i.e.
the Kolmogorov complexity of a string when an external string is available
for support [58, 59]). Although recently introduced, the compression dis-
tance attracted considerable attention from general scientific community [8]
due to its demonstrated success in capturing the evolutionary relationships
between world languages (through comparing “declaration of human rights”
in various languages) [30], DNA and RNA sequences 2, and identification
of authorship for works of literature [30]. Unlike character edit distances,
the block edit distance between two sequences is NP-hard to compute [60]
3. Another approach to examining genomic phylogenies with respect to relative information content has recently been introduced [21], as well. Further
studies on block edit operations and their applicability to comparing strings
and sequences are also available [32].

2Known mitochondrial DNA sequences of various species [58] and families of retro-
transposons [85].

3In fact, any edit distance measure that involves the block copy operation is NP-hard
to compute.
5.1.3 Indexing Strings for Similarity Searching

Indexing strings (of all types) for similarity searching has been an ongoing area of research for many years. There is a plethora of indexing schemes for searching through strings for matching subsequences, and most of these revolve around the general notion of suffix trees/arrays. These structures are built up from characters that are found in the strings, and are thus tightly bound to a particular distance measure. Such methods cannot easily adapt to new measures of similarity (such as block edit distances). Instead, one must look towards more general indexing schemes.

Efficient indexing methods compute the near neighbors of a query item by iteratively pruning subsets of potential answers to a query (desirably a large fraction of them) by partitioning the data set and checking out to which partition the query belongs. For vector spaces (where dimensions are well defined), this pruning can be done by focusing on a small number of dimensions (preferably one) at a time. Spatial index structures such as R-trees [40], R+-trees [74], R*-trees [10], quad-trees [35], X-trees [14], SR-trees [54], A-trees [72], and others are examples that employ this general strategy. These data structures work quite well for vector spaces with small dimensionality; however as the number of dimensions grow, their performance drops significantly, possibly to the level of the brute-force search (as a result of the commonly referenced curse of dimensionality [23, 15, 87]).

To overcome the curse of dimensionality in vector spaces, several dimension reduction techniques, such as those based on random projections, sam-
pling, etc., have been demonstrated to be effective [4, 19, 37, 48, 56]. For example, one can estimate the Hamming distance between two data items within some $(1 + \epsilon)$ approximation factor by comparing a small random sample of their attributes (i.e. dimensions). Clearly this general approach requires that the data items have well defined dimensions, i.e. a priori knowledge of which attribute of one data point is compared with which attribute of another. Strings under edit distances have no clear notion of a dimension: even a single character insertion or deletion may alter which attribute (i.e. character/block) of one string needs to be aligned and compared to which attribute of the other string. Thus because of their complex nature, no result even for character edit distances (the simpler of the two presented) even approximate or probabilistic guarantees.

There are some other indexing methods on near(est) neighbor search for vector spaces only, especially for Euclidean distances [37, 56, 47, 45, 46] which, (with high probability), guarantee to reduce the number of comparisons from linear to poly-logarithmic with respect to database size. Some of these methods were later employed in sequence near/nearest neighbor search (e.g. [19, 18]). Unfortunately these methods perform similarity search under hamming distance, which allows character replacements only (excluding even character insertions and deletions). Thus the applicability of these methods are limited to very short sequences and for specific applications such as discovering short motifs that regulate gene expression or very short proteins [65, 73].

Dimension reduction techniques for strings under various edit distances
rather “reduce” a long query string to one or more of its substrings (usually called q-grams) [39, 52, 75, 43] and identify all strings in the database which also include such substrings (q-grams). Many of these methods are designed for substring similarity search problems in which the goal is to find the most similar substrings of a long text string to a query string. When applied to similarity search problems involving full strings (which are the main focus of our work), these methods usually provide a tradeoff between the false positive rate and the false negative rate by adjusting the length of the q-grams searched. With proper calibration, such methods can prune out ≥ 50% of the strings in certain data sets, however, for many other data sets their performance drops to the level of brute force search.

An alternative method for dimension reduction for strings was presented in [62] which provides an involved mapping of strings under certain block edit distances (such as the transformation distance [85]) to binary vectors under the Hamming distance. Although the number of dimensions in these binary vectors are high, they can later be decreased substantially through random projections [19, 37, 48, 56]. Unfortunately this approach imposes an approximation factor of $O(\log n \log^* n)$ that must be tolerated, which makes it unsuitable for several applications. Moreover, it is also not appropriate for any character edit distance.
5.1.4 Motivation for this Application

Here the main premise is that string similarity search can be performed efficiently through distance-based indexing [20, 84, 17]. Originally designed for metric spaces, vantage point trees (VP-trees) [88], their variant MVP-trees [16], and M-trees [27] are examples of distance-based indexing methods. The focus in this paper is on VP-Trees which iteratively partition a given data set by defining “spheres” around select “vantage” points in the input space. Pruning is achieved if the “query sphere” falls completely within or outside of the vantage point’s sphere; this is possible due to triangular inequality, a necessary condition for a space to be metric.

Unfortunately many string distance measures do not satisfy the triangular inequality; examples include the weighted character edit distance [73, 77] and the compression distance [58, 30, 59]. In order to use distance based indexing techniques under such measures the following strategy is employed.

1. It is first shown that the compression distance and the weighted character edit distance are almost metrics, a notion introduced [70] to describe distance measures which are reflexive, symmetric, and satisfy the triangular inequality within a constant factor.

2. It is then shown how to update pruning conditions on distance based indexing methods if the distance measure used is an almost metric. The constant factor in the triangular inequality satisfied by the distance function has an important role in the performance of this modified index.
3. In high dimensional spaces, many indexing methods perform poorly if the data is uniformly distributed over the input space [13, 15]; however not much is known for other data distributions. Observations reveal however, that many applications involve string data with high levels of clustering. Such examples include all known protein strings that are active in the human brain and other organisms, and declaration of human rights in 52 Euro-Asian languages. For both data sets $f(r)$ is plotted as the number of pairs of strings whose distance is less than some radius $r$ against $r$; $f(r)$ illustrates that the first data set exhibits a polynomial distribution (a power law) whereas the second exhibits an exponential distribution. This bias can be exploited to improve the performance of distance based indexing methods (in particular VP-trees) by providing a tradeoff between query time and space for both polynomial and exponential distributions.

4. Theoretical results are verified on near(est) neighbor search via updated VP-trees using various data sets under several distance functions. One such data set is synthetic, which allows one to better understand the search performance dependency to specific parameters of the data set. Other, real, data is also used and good performance is obtained. For example, $\geq 90\%$ pruning is observed in some nearest neighbor searches in a database of all well-known proteins that are active in mammalian brains.

The applicability of the compression distance is also demonstrated by
comparing the presented method with an existing efficient implementation [62], and measuring both in terms of accuracy in approximating the block edit distance. This alternative approach provides verification of the usefulness of the compression distance by using a distance measurement technique that is far removed from the edit distance genre, yet providing nearly identical relative results.

5.2 Computing Evolutionary Similarity Between Genomic Sequences

A fundamental goal of this work is to efficiently search among genomic sequences, those which can be transformed into a query sequence through the fewest number of edit operations that emulate known evolutionary mechanisms. Thus it is of key importance to define which edit operations are sufficient to emulate the complete set of known evolutionary mechanisms [50] and how they can be used to define distances between sequences.

5.2.1 Notation

Let the following table illustrate the notations used throughout much of the remainder of this work.
5.2.2 Evolution as Sequence Transformations

Given two sequences $R$ and $S$, consider the process of transforming one into another (e.g. $R$ into $S$) through a set of edit operations. These operations must capture the full set of known evolutionary mechanisms which results in microdeletions, replications, relocations, and point mutations. A sequence transformation of biological interest must include both character edit and block edit operations. In particular, the character edit operations must include:

1. character insertion,

2. character deletion, and

3. character replacement;

For all example sequences given here, the DNA alphabet ($\{A, C, G, T\}$) is used.
and the block (i.e. substring) edit operations must include:

1. block copying,

2. block deletion, and

3. block relocation.

Another block operation of interest is block reversals. The methods described here are modifiable to handle transformations that also allow block reversals, though the focus remains on transformations which do not allow block reversals (as the reversals case adds complexity that often deems it impracticable).

**5.2.3 Distances Between Sequences**

Given two sequences $R$ and $S$, one can describe a transformation of one sequence into the other through a set of specified edit operations. Let the distance $d(R \rightarrow S)$ from $R$ to $S$ be defined as the minimum number of edit operations to transform $R$ to $S$. Such a distance $d()$ is not necessarily symmetric, i.e. there may be sequences $S$ and $R$ for which $d(R \rightarrow S) \neq d(S \rightarrow R)^4$. Given a transformation, the *symmetric version* of the distance $d()$ implied by this transformation can be defined as $d(S, R) = \frac{1}{2} \cdot (d(S \rightarrow R) + d(R \rightarrow S)) = d(R, S)$. Symmetry is a necessary attribute of *metric* distance functions, permitting an entire class of indexing methods that will be described shortly. For the remainder of this work, symmetric versions of distance functions are always used, unless stated otherwise explicitly.

---

4e.g. consider transforming an empty sequence to a long sequence with character insertions, and then consider the reverse transformation with block deletions allowed.
5.3 Sequence Transformation Classifications

Transformations can be classified based the permitted edit operations. Such classifications are necessary for understanding mathematical properties of various transformations. Let each edit operation have a source and a destination. Transformations then can have the properties described below.

5.3.1 External vs Internal Transformations

Definition 25 (External Transformations). External transformations iteratively construct $S$ from an initially empty sequence $S'$ by performing edit operations whose source is $R$ and whose destination is $S'$; thus $R$ does not change during the transformation.


1. copy block $A, T$ from $R$ to the right end of $S$ to obtain $S = A, T$,
2. copy block $C, T, G$ from $R$ to obtain $S = A, T, C, T, G$, and

Definition 26 (Internal Transformations). Internal transformations construct $S$ by iteratively applying edit operations to sequence $R'$ which is initially set to $R$. 

84

1. move the rightmost \( A \) to the right end of \( S \) to obtain \( S = A, C, T, G, T, G, A \), and
2. copy \( T \) to second position in \( S \) to obtain \( S = A, T, C, T, G, T, G, A \).

5.3.2 Marginal vs Unrestricted Transformations

Definition 27 (Marginal Transformations). Marginal transformations allow edit operations whose destinations are either a prefix or a suffix of the sequence that evolves into \( S \).

Definition 28 (Unrestricted Transformations). Unrestricted transformations allow edit operations whose destinations can be any arbitrary location on the sequence that evolves into \( S \).

5.3.3 Block Edit Distances and Uses

Based on the model above, a general distance measure between sequences to capture their evolutionary relationships can be defined; such a distance should allow all block and character edit operations without any restrictions on their locations.
Definition 29 (Block Edit Distance). Let $b(R \rightarrow S)$ be defined as the minimum number of block edit (translocations, copies, and deletions) and character edit operations (insertions, deletions, and replacements) to transform $R$ to $S$ in the unrestricted internal model. Let $b(R, S)$ be the symmetric version of this transformation, called the block edit distance.

A previously defined block edit distance [62] does not allow deletions of arbitrary blocks (which the above distance does). This other measure will be referred to as the limited block edit distance for the remainder of this document.

Lemma 13. The block edit distance provides a metric between sequences.

Proof. One must show that the block edit distance is reflexive, symmetric, non-negative, and triangularly inequal. Reflexivity is shown via $b(R, R) = b(R \rightarrow R) + b(R \rightarrow R) = 0$. Symmetry is shown via $b(R, S) = \frac{1}{2}(b(R \rightarrow S) + b(S \rightarrow R)) = b(S, R)$. A negative count of operations is clearly impossible, thus non-negativity is satisfied trivially. Finally, the triangle inequality is shown via the following argument. It is not possible to have $b(R \rightarrow S) > \frac{1}{2}(b(R \rightarrow Q) + b(Q \rightarrow S))$ as one could first transform $R$ into $Q$ and then $Q$ into $S$. Thus, all properties of a metric are satisfied. \qed

Lemma 14. The block edit distance computation is NP-hard.

Proof. Follows from [60]. \qed

Recently, block edit operations and distances based on these operations have received considerable attention [8], especially in the context of evolu-
tionary analysis of world languages and genome strings (e.g. mitochondrial DNA) from various species. Given two strings $R$ and $S$, the \textit{transformation distance} \cite{85} is defined to be the minimum number of block relocations, copies, and deletions as well as single character insertions, deletions and replacements to transform one string to another in the \textit{internal, unrestricted} model. Using this method, string $R$ is transformed into $S$ by assigning $S' = R$ and applying available edit operations directly on $S'$ to transform it to $S$. The transformation distance is identical to the the block edit distance of \cite{62}; because of its generality, it provides a lower bound to any distance based on block edits.

Although the transformation distance (a.k.a. the block edit distance) includes all edit operations responsible from genome sequence or natural language evolution, it is NP-hard to compute \cite{60}. For heuristically approximating the transformation distance \cite{85} it is possible to put a restriction on the edit operations by employing the \textit{external, 1-sided marginal} model. In other words one can start with an empty string $S'$ and transform it to $S$ by (externally) copying blocks of $R$ or inserting single characters to the right end of $S'$ (hence it is one-sided as additions to $S'$ are always made to the right end). The number of such edit operations performed gives an upper bound on the transformation distance.

Such limitations on the transformation distance can be relaxed by allowing internal copying of blocks of $S'$ to its right end as well. The resulting distance is called the \textit{compression distance}. Introduced in \cite{58, 30, 59}, the compression distance basically measures the number of Lempel-Ziv-77
phrases obtained during the compression of a string $S$ when another string $R$ is used as part of the dictionary. By the use of suffix trees the compression distance can be computed in time linear with the lengths of the strings [66].

5.3.4 Summary of Remaining Arguments

Although the character edit distance and the transformation distance are metric distances, their more useful siblings, the weighted character edit distance and the compression distance are not. It is shown here that the compression distance and the weighted edit distance (under certain conditions on operation costs) are almost metrics, i.e. they are symmetric, reflexive and that satisfy the triangular inequality within a constant factor.

A bottom-up approach is employed to show these desirable properties of the compression distance, starting with properties of external transformations, followed by the introduction of a number of intermediate distance measures for utility purposes. More specifically it is first demonstrated that any distance in the external model can be approximated by a very limited and easy to compute distance $gem_1(R, S)$, the greedy one sided external marginal distance. $gem_1(R, S)$ is then updated only slightly to obtain the compression distance $c(R, S)$, and as per $gem_1(R, S)$, it is shown that $c(R, S)$ provides a good approximation to any distance in the internal model. To accomplish this, a relaxed compression distance $rc(R, S)$ is defined as a utility distance measure. It is also demonstrated that the greedy version of $c(R, S)$, denoted $gc(R, S)$ is as powerful as $c(R, S)$ and thus $c(R, S)$ can be computed in linear
5.3.5 External Transformations

External transformations and their properties are studied first. These will be useful in subsequent proofs related to the compression distance. The following lemma demonstrates that without loss of generality, one can focus only on character insertions and block replacements (while not considering any other block edit operation) in the external model.

**Lemma 15.** Let $e(R \rightarrow S)$ be a distance from $R$ to $S$ in the external model, defined as the minimum number of character insertions, deletions, and replacements, as well as block copies, relocations, and deletions (in the marginal or unrestricted model) to transform $R$ to $S$; let $e(R, S)$ denote its symmetric version. Let $e'(R \rightarrow S)$ be a restricted version of $e(R \rightarrow S)$, in which only character insertions and block copies are allowed; let $e'(R, S)$ be its symmetric version. Then, $e(R \rightarrow S) = e'(R \rightarrow S)$, and thus $e(R, S) = e'(R, S)$.

**Proof.** The external model does not allow any block relocation operations as either it involves copying a block from $S$ to $S$ or $R$ to $R$, which are both prohibited in the external model, or it involves copying a block from $R$ to $S$, and then deleting the original copy in $R$, which is not permitted either. Consider now a deletion performed on block $B$ (which can be a single character) of $S'$; $B$ should have been inserted in $S'$ earlier. Note that $e(R, S)$ is the minimum number of edit operations to transform $R$ into $S$; if the copy operations involving substrings of $B$ excluded those substrings, there would
be no need to delete \( B \) later. The new set of operations can outnumber the original ones by at most one, which would be compensated by the deletion of \( B \). Thus one can achieve the same number of edit operations to transform \( R \) into \( S \) if no deletions were performed. Because a character replacement is a deletion of a character followed by an insertion of another one, there is no need to perform any replacements either. The proof follows. \( \square \)

Thus from this point on only character insertions and block replacements are allowed for any distance measure in the external model. It is now shown that in the external marginal model, performing operations on only one end of a given sequence is as powerful as allowing operations on both ends.

**Lemma 16.** Let \( \text{em}_1(R \rightarrow S) \) be the distance from \( R \) to \( S \) in the external marginal model in which edit operations are performed only on one end of \( S' \) and let \( \text{em}_1(R, S) \) be its symmetric version. Let \( \text{em}_2(R \rightarrow S) \) be the distance from \( R \) to \( S \) in the external marginal model in which edit operations are performed on both ends and let \( \text{em}_2(R, S) \) be its symmetric version. Then, \( \text{em}_1(R \rightarrow S) = \text{em}_2(R \rightarrow S) \), and thus \( \text{em}_1(R, S) = \text{em}_2(R, S) \).

**Proof.** Let \( E = e(1), e(2), \ldots, e(k) \) be the minimum size sequence of block copy and character insert operations in the external model in which edit operations can be applied to both ends of an initially empty \( S' \); thus \( e(R \rightarrow S) = k \). Among these edit operations let \( E_L = e_L(1), e_L(2), \ldots, e_L(k_L) \) be those which are applied on the left end of \( S' \), maintaining the original order in \( E \); i.e. for any \( i > j \), if \( e_L(i) = e(i') \) and \( e_L(j) = e(j') \), then \( i' > j' \). Similarly, let \( E_R = e_R(1), e_R(2), \ldots, e_R(k_R) \) be those operations
which are applied to the right end of $S'$, maintaining the original order in in $E$. Notice that $k = k_R + k_L$. Because the insertion operations applied on the left end of $S'$ do not have an effect on the right end (and vice versa), any permutation of edit operations in $E$ in which the order of edit operations in $E_L$ and $E_R$ are maintained transforms an initially empty $S'$ into $S$. Now consider $e_L(1), \ldots, e_L(k_L), e_R(1), \ldots, e_R(k_R)$, which is one such permutation in which all $e_L(i)$s are applied on the left end and all $e_R(j)$s are applied on the right end. Then $E' = e_L(k_L), e_L(k_L - 1), \ldots, e_L(1), e_R(1), e_R(2), \ldots, e_R(k_R)$ in which all operations are applied on the right end of an initially empty $S'$ would construct $S$ as well.

Finally it is shown that no distance measure in the external model provides a metric. Internal transformations are later addressed that retain similarity with external transformations, but are metrics.

**Lemma 17.** No distance measure in the external model satisfies the triangular inequality and thus no distance measure in the external model is a metric.

**Proof.** Consider $R = x$, $S = (x, y)^i$, and $Q = x, y$. Now for any distance $u(R \rightarrow S)$ in the external unrestricted model, $u(R \rightarrow S) = 2 \cdot i$, $u(R \rightarrow Q) = 2$, and $u(Q \rightarrow S) = i$. Thus $u(R \rightarrow Q) + u(Q \rightarrow S) = 2 + i < 2 \cdot i = u(R \rightarrow S)$. Because $u(S \rightarrow R) = 1$, $u(Q \rightarrow R) = 1$, and $u(S \rightarrow Q) = 1$, $u(R, Q) + u(R, Q) = 2 + i/2 < i + 1/2 = u(R, S)$ for any $i > 2$. $\square$

**Definition 30** (The Greedy External Model Transformation/Distance). Let the greedy version of $em_1$ transformation from $R$ to $S$ (and the associated
greedy $e_m$ distance, denoted by $gem_1(R,S)$) be defined as follows. In any step of this transformation, let $S'$ be the already constructed prefix of $S$ and let $S = S'S''$. Then the transformation copies the longest substring of $R$ which is a prefix of $S''$ to the right end of $S'$. If no such prefix exists, this transformation inserts $S''[1]$ to the right end of $S'$.

Example 8. Consider the sequences $R = A,C,T,A,G,T,A,T$, and $S = A,G,T,C,T,A,A,T$. One can compute the $gem_1(R \rightarrow S)$ as follows. The longest prefix of $S$ which exists as a substring in $R$ is $S[1 : 3] = A,G,T$; this provides the first partition of $S$, leaving us with the task of processing $S[4 : 8]$. The longest prefix of $S[4 : 8]$ that exists in $R$ is $S[4 : 6] = C,T,A$, which provides the second partition of $S$, leaving us with $S[7 : 8] = A,T$, which also exists in $R$. Thus we can partition $S$ into 3 blocks, which means that $gem_1(R \rightarrow S) = 3$. The can verify that $gem_1(S \rightarrow R) = 4$, thus $gem_1(S,R) = gem_1(R,S) = 7/2$.

It is now shown that $gem_1(R,S)$, which is very simple to compute is equal to $e_m(R,S)$. Since $e_m(R,S)$ provides a close approximation to any distance in the external model, $gem_1(R,S)$ is (almost) as powerful as any distance measure in the external model.

Lemma 18. For any distance $e(R,S)$ in the external model, $gem_1(R,S) = e_m(R,S) \leq 2 \cdot e(R,S)$. The approximation factor of 2 is tight.

Proof. The proof consists of 3 steps:

1. $gem_1(R,S) = e_m(R,S)$: let $S_{gem}(k)'$ be a prefix of $S$ which is the state of $S'$ after the first $k$ edit operations associated with $gem_1(R \rightarrow$
Similarly let $S_{em}(k)'$ be a prefix of $S$ which is the state of $S'$ after the first $k$ edit operations associated with $em_1(R \rightarrow S)$. Because $gem_1(R \rightarrow S)$ picks the longest substring of $R$ to extend $S'$, it is easy to see that for any $k$ if $|S_{gem}(k)'| \geq |S_{em}(k)'|$, then $|S_{gem}(k+1)'| \geq |S_{em}(k+1)'|$. The proof follows inductively.

2. $em_1(R, S) \leq 2 \cdot e(R, S)$: because $e(R, S) \geq eu(R, S)$, it is sufficient to show that $em_1(R, S) \leq 2 \cdot eu(R, S)$. The proof follows from the fact that the insertion (of a block or a single character) operations in $S'$ can be reordered from left to right in a way that each insertion of a block $B$ by the transformation associated with $eu(R \rightarrow S)$, which splits an existing block $B'$ into two portions $B'_1$ and $B'_2$, can be handled by the transformation associated with $gem_1(R \rightarrow S)$ by at most two block insertions, one for $B$ itself and another for $B_2$.

3. There exists sequences $R$ and $S$ for which $gem_1(R, S) = 2 \cdot eu(R, S)$: consider $R = x, y$ and $S = x^i, y^i$. Clearly $gem_1(R \rightarrow S) = 2i$ and $eu(R \rightarrow S) = i$ (by consecutively inserting $R$ in the middle of $S'$). On the other hand, $gem_1(S \rightarrow R) = eu(S \rightarrow R) = 1$. Then $gem_1(R, S) = \frac{1}{2}(2i + 1)$ approaches twice $eu(R, S) = \frac{1}{2}(i + 1)$.

\[\square\]

### 5.3.6 Internal Transformations

The focus now turns on generalizing some of the external transformations to their internal counterparts, leading to a new set of properties to be explored.
Lemma 18 it is shown that $gem_1(R, S)$ is a very simple but powerful distance that approximates all possible external distances within a factor of 2. Here, this distance is modified slightly into the compression distance which is shown to have similar properties in approximating any distance in the internal model including the block edit distance.

**Definition 31** (Compression Transformation/Distance). Consider the following internal transformation from $R$ to $S$ and its associated symmetric distance $c(R, S)$, which is called the compression transformation and compression distance respectively. The transformation (and the associated distance $c(R, S)$) is a generalization of the $em_1$ transformation (and respectively $em_1(R, S)$) as follows:

1. $R$ is transformed into $S$ by starting from $R' = R$ and iteratively copying any substring of $R'$ or inserting any single character to its right end.

2. As a final operation, deletion of the original version of $R$ from the left end of $R'$ is permitted.

It is now shown that the greedy version of the compression distance is equal to the compression distance, making the job of computing the compression distance very easy.

**Lemma 19.** Consider the greedy version of the compression transformation between $R$ and $S$, where the block copy operations are chosen greedily as per the greedy $em_1$ transformation (Definition 30). Let $gc(R, S)$ be the symmetric distance associated with greedy compression transformation. Then $gc(R, S) = c(R, S)$.  

94
Proof. Follows immediately from the proof of Lemma 18.

Example 9. Consider the sequences \( R = A, C, T, A, G, T, A, T \) and \( S = A, G, T, C, T, A, A, T \) again. One can compute the \( c(R \to S) \) as follows. Begin with \( S' = R = A, C, T, A, G, T, A, T \). The longest prefix of \( S \) which exists as a substring in \( S' \) is \( S'[4 : 6] = A, G, T \); \( S' \) is updated to \( S' = A, C, T, A, G, T, A, T, A, G, T, A, T \). In the remainder of \( S \) (which is \( S[4 : 8] \)) the longest prefix which exists in \( S' \) is \( S'[2 : 4] = C, T, A \); so \( S' \) is updated to \( S' = A, C, T, A, G, T, A, T, A, G, T, C, T, A \). The remaining portion of \( S \) exists as \( S'[7 : 8] = A, T \), and thus another update yields \( S' = A, C, T, A, G, T, A, T, A, G, T, C, T, A, A, T \). The final operation is the deletion of the original version of \( R \) as a prefix of \( S' \), leaving \( S' = S = A, G, T, C, T, A, A, T \). This implies that \( c(R \to S) = 3 \). The reader can verify that \( c(S \to R) = 4 \), thus \( c(S, R) = c(R, S) = 7/2 \). Note that for this example \( c(R, S) = gem_1(R, S) \), which is not the case for all \( R \) and \( S \).

Lemma 20. The compression distance \( c(R, S) \) between two sequences \( R \) and \( S \) can be computed in optimal \( O(|R| + |S|) \) time.

Proof. By a simple application of the \([66]\) on-line suffix tree construction method, the computation of \( Q \), the longest prefix of the unconstructed portion of \( S \) can be performed in time \( O(|Q|) \) time. The proof follows.

Although the compression distance is efficiently computable, it does not satisfy the triangular inequality and hence is not a metric.

Lemma 21. The compression distance \( c() \) does not satisfy the triangular inequality and thus is not a metric.
Proof. Consider $R = v, w, x, y, z, Q = z, y, x, w, v$ and $S = w, v, x, w, y, x, z, y, y, x, w, x, w, v, z, y, x, w, v$. One can verify that $c(R \rightarrow Q) = 6 = c(Q \rightarrow R)$, $c(Q \rightarrow S) = 10$, $c(S \rightarrow Q) = 2$, $c(R \rightarrow S) = 19$ and $c(S \rightarrow R) = 6$. Thus $c(R, S) = \frac{6+19}{2} = 12.5 \geq c(R, Q) + c(Q, S) = \frac{6+6}{2} + \frac{10+2}{2} = 12$.

Lemma 21 demonstrates that in order to construct a sequence $S$ from sub-strings of another sequence $R$, it may be helpful to construct an intermediate sequence $Q$. Because the compression transformation does not allow deletion of a substring except $R$ itself, $Q$ cannot be incorporated in the construction. It is possible to relax this deletion constraint to obtain a metric distance. Because this relaxed version approximates the compression distance well, it will help show that compression distance is an almost metric.

Definition 32 (Relaxed Compression Transformation/Distance). Consider relaxing the compression transformation from $R$ to $S$ by allowing deletion of any prefix of $R'$ as a final operation. This transformation is called the relaxed compression transformation and its associated distance the relaxed compression distance, denoted $rc(R, S)$.

Lemma 22. The relaxed compression distance, $rc(R, S)$, provides a metric.

Proof. One must show that the compression distance is reflexive, symmetric, non-negative, and triangularly unequal. Reflexivity is shown via $rc(S, S) = [rc(S \rightarrow S) + rc(S \rightarrow S)]/2 = 0$. Symmetry is shown via $rc(R, S) = [rc(R \rightarrow S) + rc(S \rightarrow R)]/2 = [rc(S \rightarrow R) + rc(R \rightarrow S)]/2 = rc(R, S)$. A negative
count of operations is clearly impossible, thus non-negativity is satisfied trivially. Finally, the triangle inequality is shown via the following argument. Consider sequences $R, S, Q$. It is not possible that $rc(R \rightarrow S) > rc(R \rightarrow Q) + rc(Q \rightarrow S)$ as one can simply transform $R$ to $S$ by first transforming $RQ$ via at most $rc(R \rightarrow Q)$ edit operations and then transforming $RQ$ to $S$ via $rc(Q \rightarrow S)$ edit operations. The proof follows through a symmetric argument on $rc(S \rightarrow R)$.

Unfortunately the relaxation makes the distance much harder to compute, as seen in the Lemma 23.

**Lemma 23.** Computing $rc(R \rightarrow S)$, the relaxed compression distance between $R$ and $S$, is NP-hard.

**Proof.** Follows from setting $R$ to the empty sequence and by using the proof in [79] stating that the optimal dictionary selection problem is NP-hard. 

Nevertheless, the compression distance (which is easy to compute) provides a tight approximation to the relaxed compression distance. Because relaxed compression distance is a metric, this implies that the compression distance is an *almost metric*, a notion defined below.

**Definition 33** (Almost Metrics). A distance function $f$ provides an almost metric for space $S$, if it is symmetric, reflexive and satisfies the triangular inequality within a multiplicative constant $k$; i.e. for all $S, R, Q \in S$, $f(S, R) \leq k \cdot [f(S, Q) + f(Q, R)]$. 

97
Now it is shown that the compression distance provides a constant approximation to its relaxed version, and hence is an almost metric.

**Theorem 4.** $rc(R, S) \leq c(R, S) \leq 3 \cdot rc(R, S)$.

**Proof.** In the relaxed compression transformation from $R$ to $S$ which gives $rc(R, S)$, suppose that $R' = RQS$ right before the final prefix deletion operation. Now consider the set of extended compression transformations where deletion of a prefix of $R'$ longer than the original $R$ may be permitted. Let $ec(R \rightarrow S[\max(R')])$ denote the minimum number of operations to transform $R$ into $S$ via an extended compression transformation where $\max(R')$ is the maximum extent of sequence $R'$. Thus $ec(R \rightarrow S[RS])$ is equal to $c(R \rightarrow S)$ as the maximum extent of $R'$ in the compression transformation is the concatenation of $R$ and $S$. Similarly $ec(R \rightarrow S[RQS])$ is equal to $rc(R \rightarrow S)$.

**Claim 1**
For any sequence $P$, $c(R \rightarrow S) \leq em_1(R \rightarrow P) + 3 \cdot c(RP \rightarrow S)$.

**Proof**
Let $S = S_1, \ldots, S_k$, where $S_i$’s are the substrings of $S$ used in the compression transformation of $RP$ into $S$. Similarly let $P = P_1, \ldots, P_l$, where $P_i$’s are the substrings of $P$ used in the $em_1$ transformation of $R$ into $P$. Consider the process of emulating the compression transformation from $RP$ to $S$ via the compression transformation from $R$ to $S$. Inductively assume that the claim is correct for $S_1 \ldots S_{i-1}$. The lemma must be proven then for $S_1 \ldots S_i$. If $S_i$
is a copy of a substring in $S_1 \ldots S_{i-1}$ or in $R$, the compression transformation from $R$ to $S$ can perform the copying of $S_i$ to the end of $R'$ in a single copy operation as well. If, however, a given $S_i$ is a copy of a substring of $P$, then let $S_i = P'_j, P_{j+1}, P_{j+2}, \ldots, P_{j+\ell-1}, P'_{j+\ell}$, where $P'_k$ is a suffix of $P_k$ and $P'_{j+\ell}$ is a prefix of $P_{j+\ell}$. Now if $\ell \ell \leq 2$ then the claim is trivially proven as for every copy operation in $c(RP \rightarrow S)$, one is allowed to perform three copy operations in $c(R \rightarrow S)$. If, on the other hand, $\ell \geq 3$, then (1) three copying operations are charged in $c(R \rightarrow S)$ to the copying operation of $S_i$ in $c(RP \rightarrow S)$, (2) and the rest of the copying operations are charged in $c(R \rightarrow S)$ to the borders in between $P_{j+1}, \ldots, P_{j+\ell-1}$ - each of which is a result of a copy operation in $em_1(R \rightarrow P)$; if a border between $P_h$ and $P_{h+1}$ is already been charged, it means that $P_hP_{h+1}$ exists as a substring in $S_1 \ldots S_{i-1}$ and thus can be copied in a single shot - thus there is no need for a second charging.

Claim 2

For any sequence $P$, $c(R \rightarrow S) \leq c(R \rightarrow P) + 3 \cdot c(RP \rightarrow S)$.

Proof

Let $P = P_1, P_2, \ldots, P_k$ for which the transformation from $R$ to $RP$ only performs copying from prefix $RP_1, \ldots, P_k$ for any block in $P_{k+1}$. Also let $S_i$ be the collection of blocks in $S$ which are copied from $P_i$ during the optimal compression transformation from $RP$ to $S$. Then by Claim 1, $c(R \rightarrow S_i) = c(RP_1 \ldots P_{i-1} \rightarrow S_i) \leq em_1(RP_1, \ldots, P_{i-1} \rightarrow P_i) + 3 \cdot c(RP \rightarrow S_i)$. Similarly, one can verify for all $i$ that $c(R \rightarrow S_i) = c(RP_1 \ldots P_{i-1} \rightarrow S_i) \leq 99$.
\[ \text{em}_1(RP_1, \ldots, P_{i-1} \rightarrow P_i) + 3 \cdot c(RP_1 \ldots \rightarrow S_i). \]

Notice that
\[ \sum_{i=1}^{t} c(R \rightarrow S_i) = c(R \rightarrow S) \leq \sum_{i=1}^{t} \text{em}_1(RP_1, \ldots, P_{i-1} \rightarrow P_i) + 3 \cdot c(RP \rightarrow S_i) = c(R \rightarrow P) + 3 \cdot c(RP \rightarrow S). \]

The proof follows.

What follows in the final observation is that the block edit distance is closely approximated by the relaxed compression distance, and as per Theorem 4 by the compression distance itself.

**Lemma 24.** \( b(R, S) \leq rc(R, S) \leq 4 \cdot b(R, S) + 1. \)

**Proof.** Consider the block edit transformation from \( R \) to \( S \). Let \( R_0 = R \); the result of the \( i^{th} \) block edit operation gives the sequence \( R_i \); thus if \( b(R \rightarrow S) = k \), then \( R_k = S \). One can emulate each operation performed on \( R \) during the block edit transformation via at most 4 operations permitted in the relaxed compression transformation from \( R \) to \( S \). Again start with \( R'_0 = R \) and iteratively construct \( R'_i = R'_{i-1}R_i = R_0, R_1, \ldots, R_i \).

1. If the \( i^{th} \) operation performed by the block edit transformation is *deletion* of block \( R_i[k : l] \), then the relaxed compression transformation copies the blocks \( R'_i[1 : k-1] \) and \( R'_i[l+1 : |R'_i|] \) from \( R'_i \) to its end so as to obtain \( R'_i = R'_i, R_{i+1} \).

2. Similarly, if the \( i^{th} \) operation is *copying* block \( R'_i[l : h] \) in between \( R'_i[k-1] \) and \( R'_i[k] \), then the relaxed compression transformation copies the blocks \( R'_i[1 : k] \), \( R'_i[l : h] \) and \( R'_i[k+1 : |R'_i|] \).
3. Finally, if the $i^{th}$ operation is relocating block $R'_i[l : h]$ in between $R'_i[k - 1]$ and $R'_i[k]$ (w.l.o.g., assume that $k < l$) then the relaxed compression transformation copies the blocks $R'_i[1 : k], R'_i[l : h], R'_i[k + 1 : l - 1]$ and $R'_i[h + 1 : |R'_i|]$.

5.3.7 The Weighted Edit Distance is an Almost Metric

Weighted edit distances are commonly used for protein and genome string comparisons. Here each operation has a certain cost, determined in general by the log of the probability of that edit operation occurring in one of the two copies of a string during its evolution. Let $h$ be the highest cost of any operation among the ones allowed and $l$ be the lowest. Then $h$ times the character edit distance between any two strings $R$ and $S$ will provide an upper bound on their weighted edit distance. Similarly $l$ times the edit distance will provide a lower bound, which implies that the weighted edit distance satisfies the triangular inequality within a factor of $h/l$.

5.4 Distance Based Indexing for Sequences via Compression Distance

There are a number of distance based indexing structures in the literature including [84, 88, 28, 17, 16, 27, 91], which are potentially applicable to sequence proximity search. One such distance based indexing method, the
MVP trees, are here modified to accommodate *almost metrics*, and in particular to the compression distance. Note that it will be immediately clear the it is possible to modify other distance based techniques for almost metrics as well. However, the goal here is to simply show that being almost metric is a property for a distance measure that can be employed to perform efficient proximity search.

For completeness, the main idea behind distance based index structures and MVP trees is described briefly below. It is later shown how these structures can be generalized so as to perform efficient search under almost metrics.

### 5.4.1 Distance based indexing

The general task of similarity search is a well studied problem in database research. The types of queries can vary, but most can be reduced to one of the two basic forms: the near neighbor query (i.e. range query), and the nearest neighbor query.

More formally, a near neighbor query on a given set of data elements \( X = \{x_1, x_2, \ldots, x_n\} \) asks to retrieve all data elements that are within some specified distance \( r \) of a given query point \( q \); i.e. the task is to return \( X' = \{x_i | x_i \in X \& d(x_i, q) \leq r\} \). The nearest neighbor query, on the other hand, asks for the closest element in the data set to the query element. Other types of queries include k-nearest neighbors, k-farthest neighbors, etc.

Fundamental to the similarity search problem is the nature of the search
space and the distance function. When the distance function and search space can convey spatial information (as is the case of Euclidean distances), a wide variety of spatial index structures can be used to efficiently answer the queries. These methods include R-trees [40], R+-trees [74], R*-trees [10], quad-trees [35], X-trees [14], SR-trees [54], A-trees [72], and others. An excellent survey of popular methods, along with some insight into the general problems of searching in metric spaces is recently available [42]. Such index structures take advantage of the fact that each data element has a well-defined location in the search space. The distances of these elements to arbitrary points in the search space can be computed, and using this property the search space is often broken up hierarchically into partitions that can be later pruned by exploiting properties of the particular nature of the partitioning method. Because of the complexity of space partitioning in multi-dimensional spaces, such methods are particularly useful for spaces with small dimensionality.

Distance-based index structures are inherently different than these spatial index structures [20, 84, 17]. Here, only the relative distances of data elements are needed for index construction and search, i.e., no spatial information on the data elements are utilized. The key principle is again eliminating a subset of the search space (through hierarchical partitioning) that can be proven to not contain any of the data points in the answer of the query. This pruning is often done using the triangle inequality, a feature of metric spaces, which provides a necessary condition on the search space. Vantage point trees (VP-trees) [88], their variant MVP-trees [16], and M-trees [27]
are examples of distance-based indexing. Because dimensionality is not a key issue for the pruning technique they employ, such distance based methods are considered to be more effective for high dimensional data than the above alternatives.

For certain types of data it is possible to obtain specialized indices [39, 2], optimized (and often designed exclusively) for operation under a unique distance (such as character edit distances for textual data). Unfortunately, not all distance measures provide such specialized structure as per our problem. Because of the inherent high dimensionality of the data considered here, distance-based methods are studied exclusively.

A primary difficulty is that available distance based methods require the distance measure used to be a metric, which is not satisfied by the compression distance. Metric trees can be modified however, so that it enables efficient proximity search even under almost metrics.

5.4.2 Updating VP trees for almost metrics

In this section it is shown how to update vantage point (and VP) trees so that they could be effectively used in proximity search in almost-metric spaces. As described earlier for vantage point trees, portions of the search space can be eliminated when certain conditions (the distance from the query element to the vantage point and the radius of the query) hold. This elimination is the primary goal and measure of effectiveness of the vantage point tree, as each branch eliminated from search reduces the number of the candidates for
answering the query. Using the definition of an almost metric, it is now shown how one can eliminate branches in the binary vantage point tree search.

Let $Q$ be the query element, $r$ be the query range, $X_v$ be the vantage point accessed during the search, and $M$ be the median distance value for $X_v$. Given an almost metric distance $d()$ which satisfies the triangular inequality within a factor of 3, it is first shown that if $d(X_v, Q) + r < M/3$ one does not have to search the right branch. Then, it is shown that if $d(X_v, Q) - 3r > 3M$ then one does not have to search the left branch.

Let $Z$ denote any data element indexed in the right branch, and $Y$ any data element indexed in the left branch.

1a $d(Z, X_v) \geq M$
1b $M/3 > d(X_v, Q) + r$ (hypothesis)
1c $d(X_v, Q) + d(Q, X) \geq d(X_v, Z)/3$ (triangular inequality)
1d $d(Q, Z) > r$ (summation of 1a, 1b, and 1c)
2a $d(Y, X_v) \leq M$
2b $3M < d(X_v, Q) - 3r$ (hypothesis)
2c $d(X_v, Y) + d(Y, Q) \geq d(X_v, Q)/3$ (triangular inequality)
2d $d(Y, Q) > r$ (summation of 2a, 2b, and 2c)

One method for performing a near/nearest neighbor search is to pick a "good" value of $r$ (possibly using a heuristic method) to reduce the search
space and then perform each actual distance computation in the reduced $r$-radius set to pick the correct data point. Vantage point trees can also be used though to help reduce the final search space without the need to pick a radius $r$. In this case, only the right branches of the search can be eliminated.

It is now shown that if $d(X_v, Q) < M/6$, the right branch can be eliminated from the search. Let $Z$ again denote any data element indexed in the right branch.

3a $M/6 > d(X_v, Q)$ (hypothesis)

3b $d(X, Q) > d(X_v, Q)$ (summation of 1a, 3a, and 1c)

5.5 An alternative to compression distance: signatures

In this section it is shown how to compute a respective signature for each sequence in a data set to approximate the block edit distance among them [62]. A performance comparison of compression distance and the [62] method for various data sets and under multiple settings is provided in Section 5.6 A high agreement between the two methods for approximating block edit distances is observed, with the compression distance ultimately showing less variance in its approximations.

Given two sequences $S$ and $Q$, the [62] method computes bit strings $T(S)$ and $T(Q)$ as the signatures of $S$ and $Q$. The main result of [62] is that the hamming distance between $T(S)$ and $T(Q)$ approximates a limited version of
the block edit distance (where arbitrary deletions are not allowed) between
the original sequences $S$ and $Q$.

The signature computation is done as follows: the first step is the com-
putation of the so called core blocks/substrings of $S$ and $Q$; the signature of
$S$ is simply a string of bit-flags, one for each possible core block, indicating
whether that core block exists in $S$ or not. Obtaining these core blocks is
described below. Note that this implementation provides practical improve-
ment over [62] and thus is one of the modest contributions of our paper.

1. The core blocks of $S$ are computed hierarchically. At the base level
every substring of the form $10^k1$ and $01^l0$ for all $k > 1$, $l > 2$ is a
core block. Core blocks at the next level are concatenation of a few
successive core blocks at the current level - this is done iteratively
for each level through the so called locally consistent parsing (LCP)
(see [69, 62] for details). A practical implementation of LCP (which is
another of contribution of this work) is described in the appendix. The
key features of the core blocks are:

- there are $O(\log |S|)$ levels of core blocks,
- if $V$ and $W$ are two identical substrings of $S$, and $V$ is a core
  block, then $W$ is necessarily a core block as well,
- core blocks at a given level cover the whole sequence $S$, however,
- core blocks do not partition $S$ as there may be overlaps between
  successive core blocks.
Each unique core block has a unique label. When processing a set of sequences, identical core-blocks get identical labels no matter in which sequence they occur at.

2. Now the signatures must be computed. Consider the set of unique core blocks in all sequences, \( \{C_1, C_2, \ldots, C_k\} \). Then the signature \( T(S) \) is simply the string of bit flags where the \( i^{th} \) bit is 1 if \( C_i \) is a core block of \( S \) and is 0 if \( C_i \) is not a core block of \( S \). For improving space efficiency, \( T(S) \) is denoted as a list labels of all core blocks in \( S \).

The space efficient representation of signature \( T(S) \) can be computed in time \( O(|S| \log |S|) \), and requires \( O(|S| \log |S|) \) space. Notice that the space requirement is larger than the original representation of \( S \). This should not come as a surprise: the signatures are helpful in approximating the block edit distance not because they reduce the dimensionality but simply because they map the block edit distance to hamming distance.

It is proven in [62] that the hamming distance between the signatures of two sequences; i.e. the signature distance, provides an approximation to (the limited version of) the block edit distance. More specifically, \( \Omega(d(Q, S))/ \log^* (|Q| + |S|) = h[(T(Q), T(S)] = O(d'(Q, S) \log(|Q| + |S|) \log^* (|Q| + |S|)) \) where \( h[T(Q), T(S)] \) is the hamming distance between \( T(Q) \) and \( T(S) \) which is equal to \( \sum_i T(Q[i]) \oplus T(S[i]) \).

In their version of the block edit distance, denoted \( d'(S, Q) \), arbitrary deletions of blocks are not allowed; it is possible to delete a block only if another copy of it exists elsewhere. Another important restriction on this
version of block edit distance is that no two edit operations can overlap; i.e. if a character was involved in any edit operation, it can not be touched again. This especially is an undesirable limitation on the power of their method as such restrictions can not be imposed on biochemical mechanisms that result in the structural rearrangements of interest in the human genome. However, because hamming distance provides a metric, the signatures are well suited to be used with distance based indexing methods.

While the block edit distance can be embedded into the Hamming Distance quite efficiently, the same is not true for arbitrary character edit distances. Recent results [3], [71] show that such a mapping cannot be achieved without tolerating an approximation factor of at least $\frac{3}{2}$, as well as illustrating a few other limitations to the Hamming Distance embedding approach.

### 5.6 Results on the Comparisons of the Approximations of Signatures and Compression Distances

A generated data set is used here to test the accuracy of the two presented approximation methods. The goal with the generated sequences is to emulate evolutionary alterations to genome segments (i.e. character and block edits). This is implemented as follows.

1. Start by “randomly” generating a single sequence which eventually serves as the query sequence.
2. The characters of the query sequence is selected independently and uniformly at random from the DNA alphabet.

All of the other sequences for a particular data set are generated through controlled alterations (block and character edits) to the query sequence. The number of such alterations are increased both linearly and geometrically. Given the number of edits (imposing a distance between the query and the sequence), each edit operation is picked among the character and block edits independently and uniformly at random. The lengths of the blocks to be operated on are determined through a binomial distribution; i.e. by starting with block length 1 and increasing it iteratively by 1 with probability \(1 - 2^i\) for \(i \in \{1, \ldots, 5\}\). The starting location of each edit operation was also picked uniformly at random for unclustered sequences generated with unclustered edits. Experiments were also done with sequences with varying degrees of clustering. In such data sets block edit operations were performed in clusters of 3 or 5 with distance between the locations of adjacent block operations fixed as \(2^{i+1}\), twice the expected length of a block operation.

### 5.6.1 Generated Ordering vs True Ordering

In both of the models for generating random sequences, the number of mutations is taken to be the actual distance (either character edit distance or block edit distance). The lengths of the original sequences, the rates of growth of the number of mutations, and the number of mutated sequences were all chosen such that the chance of overlapping operations was low. If too many
operations overlap, the measure of the actual distance through this production method begins to fall off, as the results of \( n \) overlapping operation events may in the best case be achievable with fewer than \( n \) mutations. By choosing the parameters carefully though, the number of mutation events that take place when generating the sequences can be taken as a close indicator to the minimal corresponding edit distance.

### 5.6.2 Accuracy Tests on Generated Data

Figures 5.1, 5.2, and 5.3 illustrate tests of the accuracy of both the compression distance and the hamming distance between signatures. Not only are the numerical values of the estimated distances checked, but also whether the order of sequences in terms of their distance to the query are preserved. One can observe that although signatures preserve the order of sequences against the query well, the compression distance does a better job. In terms of numerical values of block edit distance approximations, the compression distance performs much better, providing an approximation factor of approximately 1.2 for sequences under non-clustered block operations; this is much better than what the theoretical results suggest. It is interesting that both block distance estimators become more accurate as more block edit operations on longer blocks are performed on the sequences. Also note that for clustered operations, the estimated distances are smaller than the “correct” block edit distances. This is due to overlaps between such edit operations: because of such overlaps it becomes possible to perform fewer number of block edits to
transform one sequence to another than the operations actually perform.

5.6.3 Genome Sequences and Agreement Tests

Figure 5.4 illustrates the results of a comparison of two block edit distance estimators. The compression distance and the signature/Hamming distance methods were both used to generate orderings of long, duplication-rich, segments from the chromosome 16 of the human genome that are known to evolutionarily linked. Because such sequences are generated in an uncontrolled fashion one cannot test how well the implementations estimate the block edit distance between pairs of sequences. One can, however, test how well the two implementations agree on pairs of sequences. To achieve this tests were performed on 16 pairs of evolutionarily related sequences (complete or draft BAC clones) each of size approximately 20kb, comprising of duplicate genes some of which are silenced. For each sequence $S$ in the data set, its compression distance is computed against the rest of the sequences as well as the hamming distance between the signature of $S$ and that of the others. For a given sequence $S$ the ordering (sorting) of the sequences according to their compression distance against $S$ and compare this order with that obtained through the hamming distances of signatures. The plots shown provide some of the best and worst agreeing orderings are demonstrated. One can observe that for most sequences the signatures and compression distance agree quite well\textsuperscript{5}.

\textsuperscript{5}Test were also performed to see how these two approaches compare to character edit distances; significant discrepancies were observed due to the fact that the sequences involve
5.7 Properties of String Spaces for Proximity Search Applications

There are two important issues that determine whether proximity searches could be performed efficiently by distance based indexing.

1. The distance measure used has to be a metric or an almost metric - this issue was addressed in the earlier sections of the paper.

2. The distribution of data points should be “suitable” for distance based indexing - which is the focus of this section.

Some earlier works [13, 15] consider especially metric spaces where data is distributed uniformly and identically over the input space. They show that for any given point \( p \), as the dimensionality increases the difference between the nearest and the farthest neighbors of a point diminishes. This suggests that nearest neighbor searches in high dimensional spaces may not give meaningful answers for uniformly distributed data. However, biased/clustered distributions of data have not been thoroughly investigated under efficiency considerations for distance based indexing.
5.7.1 Characteristics of Data Sets that Contain Evolutionarily-Related Members

The primary concern here is to understand how typical string data from applications in computational linguistics and computational proteomics is distributed over the input space. The two test data sets used here are listed below.

1. Declaration of human rights in 52 Eurasian languages. This is the same data set used by [58, 30] for demonstrating the power of compression distance.

2. The complete set of protein strings that are known to be active in the brain cells of humans and other organisms from the SwissProt database (93 proteins).

Given these data sets a natural question to be asked is: how does \( f(r) \), the number of pairs of strings whose distance is at most \( r \), behave against \( r \)? The performed tests give the following answers:

1. The linguistic data under compression distance seems to be exponentially distributed over the input space. More specifically, the function \( f(r) \) seems to satisfy \( f(r) = k \cdot c^r \) for some constants \( c \) and \( k \) as the plot of \( \log f(r) \) against \( r \) is quite close to a straight line. Clearly \( \log f(r) = \log k \cdot c^r \) gives \( \log f(r) = \log k + r \cdot \log c \) which enables the computation of constants \( c \) and \( k \) to be approximately \( 2^{1/400} \) and \( 2^{-2.2} \) respectively. Considering that the number of characters (and hence
the potential dimensions) in each string is in the order of a few thousand; these constants are very small and indicate some clustering (non-uniformity) in the data set. This does not come as a surprise as these languages are evolutionarily related to each other [30].

2. The protein data under both the character edit distance and the compression distance gives a polynomial distribution over the input space (i.e. satisfies a power law). In other words the function $f(r)$ seems to satisfy $f(r) = k \cdot r^c$ for some constants $c$ and $k$: this can be observed through the log-log plots of $f(r)$ against $r$ which are quite close to straight lines. If $\log f(r) = \log(k \cdot r^c)$, then $\log f(r) = \log k + c \cdot \log r$.

The constants $k$ and $c$ vary only slightly among the two distances; they imply some significant non-uniformity in the data set. Again this should not come as a surprise as these proteins are also evolutionarily related.

### 5.7.2 Performance of VP Trees Under Polynomial and Exponential Pairwise Distance Distributions

In this section a model is developed for analyzing the performance of the VP trees under polynomial and exponential pairwise distance distributions. Two assumptions are used here:

1. the distribution of the distances between a typical data point to other points in the data set resembles the overall pairwise distance distribution, and
2. the distribution of query points in the input space resemble the distribution of the data points.

Based on this model it becomes possible to calculate bounds on constants $c$ and $k$ under which the VP trees perform nearest neighbor searches well. The analysis below assumes that the distance measure $dist()$ provides a metric; however, it can easily be generalized to almost metrics as well.

**Nearest Neighbor Search under Exponential Distributions**

Let a data set contain $m$ points. Given any typical query point $q$ the number of points observed at distance $\leq r$ obeys $f(r) = k \cdot c^r$. One can easily compute the distance between $q$ and its nearest neighbor $nn_1(q)$, namely $dist(q, nn_1(q))$ as follows. By definition $f(dist(q, nn_1(q))) = 1$ and thus $dist(q, nn_1(q)) = \log_c 1/k$.

Let $p$ be the topmost vantage point in the VP tree. It is possible to compute the distance between $p$ and its $m/\ell$’th nearest neighbor for some constant $\ell \geq 1$ which will be determined soon: $f(dist(p, nn_{m/\ell}(p))) = m/\ell$ and thus $dist(p, nn_{m/\ell}(p)) = \log_c m/k\ell$. The number of points that are within distance $dist(p, nn_{m/\ell}(p)) + dist(p, nn_1(p))$ from $p$ are $f(dist(p, nn_{m/\ell}(p)) + dist(p, nn_1(p))) = k \cdot c^{\log_c 1/k} \cdot c^{\log_c m/k\ell} = \frac{m}{k\ell}$. Let the VP tree be built in a way that once the vantage point $p$ is determined, it partitions the data set into an (1) inner partition containing the nearest $m/k\ell$ points to $p$ and (2) the outer partition containing the remaining points.

When searching for the neighbors of $q$ within distance $\epsilon = dist(q, nn_1(q))$
= \log_e 1/k \text{ (i.e. the nearest neighbors) the first step we perform is to compute }\ \text{dist}(q, p)\ \text{and take the following action according to the result.}

1. If \text{dist}(p, q) \leq \text{dist}(p, \text{nn}_{m/\ell}(p)) = \log_e m/\ell \text{ then eliminate the outer partition and iteratively perform the search on the inner partition. Because the query points are distributed similar to the data points, the probability of this case is } 1/\ell \text{.}

2. If \text{dist}(p, q) > \log_e m/\ell + \log_e 1/k \text{ then eliminate the inner partition and iteratively perform the search on the outer partition. Because the query points are distributed similar to the data points, the probability of this case can be calculated as } 1 - 1/\ell k^2 \text{.}

3. Otherwise both the inner and the outer partitions need to be searched.

If \ell = 1/2 \text{ as per standard VP trees (so that the cardinality of the inner and outer partitions are equal), the probability of case (2) is non-zero if } 1/\ell \text{, the probability of case (1) is } > 1. \text{ Thus one can ignore case (2) and obtain the following recurrence relation for the query time: } T(m) \leq 1 + 2k \cdot T(m/2) + (1 - 2k) \cdot 2 \cdot T(m/2). \text{ This recursive equation has a solution at } T(m) \leq m^{\log 2 - k/2} \text{. The space requirements of this implementation will be a small factor of } m, \text{ much smaller than the data set itself (typical strings of interest are several hundreds or thousands of bytes long).}

An Alternative Approach

It is possible to modify the standard vantage point trees so that the search time could be substantially improved for the exponential data distribution
model. In this version of the VP trees the data set is not partitioned into two by a vantage point. Rather, the following actions after computing $\text{dist}(p, q)$:

1. If $\text{dist}(p, q) \leq \text{dist}(p, \text{nn}_{m/k\ell}(p)) = \log_c m/k\ell$, then iteratively perform the search within the inner partition again. This case will again occur with probability $1/\ell$.

2. If on the other hand $\text{dist}(p, q) > \log_c m/k\ell$, then perform the search once more on the whole data set, this time using another vantage point.

To limit the space complexity one does at most some $j \cdot \ell$ comparisons of $q$ against vantage points; after that the search will be performed separately on both the inner and the outer partitions. The probability of failing in all $j \cdot \ell$ vantage points is only $(1 - 1/\ell)^{j \ell} \leq 1/e^j$.

The running time of this modification can be obtained as follows: $T(m) \leq \ell + (1 - 1/e^j) \cdot T(m/k\ell) + 1/e^j \cdot T(m/k\ell) + 1/e^j \cdot T(m(1 - 1/k\ell))$. For $\ell = 2/k$ as per the standard VP, $T(m) \leq 2/k + (1 - 1/e^j) \cdot T(m/2) + 2/e^j T(m/2)$ and thus $T(m) \leq 2/k \cdot (1 + 1/e^j)^{\log m} \cdot T(m/2)$ which implies $T(m) \leq 2/k \cdot (1 + 1/e^j)^{\log m} = O(2/k \cdot m^{\log 1+1/e^j})$.

The space requirement of this version of the updated VP tree is proportional to the total number of vantage points picked. There are $2/k$ vantage points at level 1 and in level $i$ there are $(2/k)^i$ vantage points. Because the number of levels is $\log m$, the total space requirement will be $O((2/k)^{\log m})$. 
5.7.3 Nearest Neighbor Search with Polynomial Distribution

Given any typical query point $q$ the number of points observed at distance $\leq r$ obeys $f(r) = k \cdot r^c$ for some $c > 1$. By definition $f(dist(q, nn_1(q))) = 1$ and thus $dist(q, nn_1(q)) = (1/k)^{1/c}$.

Similarly, given $p$ as the vantage point, $f(dist(p, nn_{m/\ell}(p))) = m/\ell$ thus $dist(p, nn_{m/\ell}(p)) = (m/k\ell)^{1/c}$. It is easy to verify that the number of points that are within distance $dist(p, nn_{m/\ell}(p)) + dist(p, nn_1(p))$ from $p$ is approximately $m/\ell$. Thus for $\ell = 2$ as per the standard VP tree (which sets the sizes of partitions to be $m/2$ each), the search time becomes $T(m) = 1 + 3/2 \cdot T(m/2)$ and hence $T(m) = O((3/2)^{\log m}) = O(m^{\log 3/2}) = O(m^{0.58})$. The space complexity of this standard VP tree implementation is again $O(m)$.

An Alternative Approach

A modification for improving the performance of the above VP tree can again be obtained for the polynomial distribution as per the modification for the exponential distribution. For $\ell = 2$ and letting at most $j$ vantage points per level, one can get $T(m) \leq 2 + (1 - 1/2^j)T(m/2) + 2/2^jT(m/2)$. Solving the recurrence relation gives $T(m) \leq 2 \cdot m^{\log(1+1/2^j)}$. By repeating the analysis performed for the exponential distribution, one can show the space complexity to be $O(j^{\log m}) = O(m^{\log j})$. By picking $j = 4$ one can achieve $O(n^{1/11})$ search time (which will be a small constant for all practical data sets) by using space $O(m^2)$.
5.7.4 Experimental Evaluation of VP Trees

The theoretical results indicate that the VP trees and the suggested updates would perform well especially under polynomial distance distributions. VP Tree implementations are first tested on synthetic data that exhibits some high degree polynomial distance distribution. The distance function chosen is the compression distance. The implementations are then tested for searches on the complete human proteome from the Celera databases [33]. The distance used for this application is the character edit distance.

Performance of VP-Tree on Synthetic Data

Figure 5.7 illustrates the results of tests where \( \approx 2000 \) (polynomially distributed) random strings are generated, among which the nearest 5 strings were searched for a given query string. The 2000 strings are generated via the query string, which itself is generated uniformly at random. A single “nearest neighbor” is generated by applying 5 random block edit operations on the query string; 50 of the remaining strings are generated by applying 15 operations; all the rest of the strings are generated by applying 45 operations. Experiments were performed on three different VP Trees on the same sized data set, with each VP Tree dataset using a different query string “seed” for generation.

Near neighbor searches (with radius \( \epsilon = 15 \)) were performed using compression distance using a \( O(m) \) space VP tree implementation with the best and worst possible constants in the triangular inequality (recall the constant
used in almost metrics). As described earlier, this constant can be empirically computed to ensure a most efficient implementation of the modified VP tree. Two two sets of experiments depended on the constant $k$ used which for any three of the strings $R, S, Q$ in the data set, $c(R, S) \leq k \cdot [c(R, Q) + c(Q, S)]$.

The first experiment assumes the worst possible constant $k = 3$. The second experiment assumes the best possible constant $k = 1$. It is then verified that the data set indeed satisfies the best case constant $k = 1$, but the experimental results for $k = 3$ are still provided for comparison purposes.

In the worst case, the VP tree was able to eliminate only $33 - 45\%$ of the strings in the data set to return the closest 5 strings to the query. This provides the worst case performance of the data set if the data set satisfies the triangular inequality within a constant factor of $k = 3$. In reality, the data set satisfies the triangular inequality with $k = 1$. By using this fact, the index was able to eliminate $90\%$ of the strings in the data set, resulting in significant savings in near neighbor search. A further study is to be performed examining closely the tradeoffs between different $k$ values and the percent data set elimination.

**Experimental Results on Protein Data**

(Figure 5.8) Along with the synthetic data, query experiments were performed on actual biosequence data, the results of which are illustrated in figure 5.8. The first real data set includes all active and potential proteins derived from the complete human genome sequence database, obtained by Celera [33]. The second data set is a collection of the coding DNA sequences
for a complete yeast proteome sequenced through public efforts. The human proteome consists of 38,051 sequences, while the yeast database held 6356. For each test query sequence, (character edit) range searches were performed with increasing radii to observe the pruning performance. The test query sequences were actual sequences randomly picked from the database itself (thus, one is always guaranteed to find at least one hit, regardless of query range). In the figures shown, the vertical axis represents the number of distance computations performed during the search. The horizontal line across the top of the graphs indicates 0% pruning, as the query sequence at that point has been compared with all other sequences in the database. The horizontal axis shows the ration of range to query sequence length. (Some lines do not extend to the far right of the figure as they represent long sequences, and the range was not tested to long extremes).

In the tests of the human proteome index, the plots show that the performance varies considerably, although for ranges less that 15% of the query string length, pruning is never below 50%. The best performing queries are those near the bottom of the graph, which are in general, longer that the other query sequences. Similar results are observed when querying the yeast database, although there is far less variance.
Figure 5.1: **Accuracy test: linearly increasing distances.** Testing the accuracy of approximation by the compression distance and the signatures to block edit distance on sequences of size 10Kb. Both the distance measurements and the relative ordering of the sequences against the query sequence are shown. On a pseudorandomly generated query sequence, linearly increasing number of block edit operations were applied uniformly at random to obtain the data set. Block lengths were determined by starting with length 1 and iteratively extending with probability $1 - 1/2^i$ until failure.
Compression distance based ordering - geometric growth

Signature based ordering - geometric growth

Compression distance values - geometric growth

Signature based estimates - geometric growth

Figure 5.2: **Accuracy test: geometrically increasing distances.** Testing the accuracy of approximation by the compression distance and the signatures to block edit distance on sequences of size 10Kb. Both the distance measurements and the relative ordering of the sequences against the query sequence are shown. On a pseudorandomly generated query sequence, geometrically increasing number of block edit operations were applied uniformly at random to obtain the data set. Block lengths were determined by starting with length 1 and iteratively extending with probability $1 - 1/2^i$ until failure.
Figure 5.3: **Accuracy test: clustered edit operations.** Testing the accuracy of approximation by the compression distance and the signatures to block edit distance on sequences of size 1Kb. Both the distance measurements and the relative ordering of the sequences against the query sequence are shown. On a pseudorandomly generated query sequence, linearly increasing number of block edit operations were applied to obtain the data set. The edit operations were performed in clusters of size 1, 3 and 5. Block lengths were determined by starting with length 1 and iteratively extending with probability $1 - 1/2^i$ until failure.
Figure 5.4: Agreement test. Comparing the accuracy of compression distance against signature distance on 17 genomic sequences of length 20Kb from human chromosome 16, which are known to be evolutionarily close. The relative ordering of sequences against select few sequences acting as the query sequence are plotted. Plots demonstrate both the best and the worst agreements between the two methods.
Figure 5.5: Pairwise distance distribution (compression distance): declaration of human rights in 52 Eurasian languages.

Figure 5.6: Pairwise distance distribution: proteins that are active in the brain.
Figure 5.7: Pruning efficiency test on synthetic data. The number of string comparisons made vs the size of the data set.
Figure 5.8: Pruning efficiency tests on the complete human and yeast proteomes.
Chapter 6

Probabilistic Approaches to Distance-Based Indexing

6.1 probabilistic analysis of a metric tree

The task of analyzing the performance of a metric tree directly from properties of the data is now considered.

6.1.1 distance distributions

The work here is centered around the general notion of exploiting the information provided by distance distributions over the data set. In non-vector spaces, the distributions of distances between points is one of the only means for characterizing the data set as a whole. For clarity’s sake, VP-Trees will be used as the presented model for descriptions, although other metric trees
can be analyzed using precisely the same methods.

Consider a search space $S$ and a region of this space $R \subseteq S$, and let $R$ be indexed by a vantage point $v \in S$. Let $H_{vp}(v, R)$ be the set of distances from every data point in $R$ to $v$ ($H_{vp}(v, R) = \{d(x, v) | x \in R, x \neq v\}$). A histogram describing the number of points found at any distance away from the vantage point is formed from $H_{vp}(v, R)$. This histogram can be made as granular as desired, but can be appropriately scaled (by $1/(|R| - 1)$) and viewed as a coarse version of a continuous probability density function, and for the remainder of the analysis this latter notion will be used\(^1\). The corresponding probability density function will then be called $P_{vp}(v, R, r)$, where $r$ is the distance away from $v$ (i.e. the radius of the ball centered at $v$), and the image of the triple $(v, R, r)$ under $P_{vp}$ is then the probability of finding a point in $R$ at distance $r$ from $v$. Thus, $P_{vp}(v, R, r)$ gives a concise view of the region $R$ from the perspective of $v$.

Another view of a region of the data space as a whole is obtained from the complete pairwise distance distribution. In this case, the process of acquiring $P_{vp}(v, R, r)$ is repeated for all points in $R$. That is, the distance between every pair of data points in $R$ is computed and the resulting histogram (call this set of distances $H_q(R)$) is created. This histogram can then to be scaled and viewed as another continuous probability density function, which will be called $P_q(R, r)$. (The subscript $q$ notation used is justified in section 6.1.2,\(^1\)

---

\(^1\)Since the cardinalities of the data spaces will often be enormous, performing the analysis on continuous probability density functions (as opposed to discrete probability density functions) is quite natural.
as this density function will be used to predict ranges of queries.

The importance of these two distributions relies heavily on single assumption: the distribution of query items on average follows that of the data items in the search space. This is a safe assumption on which the entire realm of improving distance-based indexing relies upon. It is easily justified by making the information theoretic argument that the query items are generated by the same process that generates the database. For example, if evolution constructs those items in a biosequence database, then this same process is assumed to generate those biosequence queries typically worked with.

Given this, one can examine any single data point in a region $R$ (and consider it representative of a single typical query point) to find its histogram of distances to all the other points in $R$. This process repeated over the entire set of data points in $R$ (plus the appropriate scaling) gives precisely $P_q(R, r)$. Thus, $P_q(R, r)$ gives the view of the region $R$ from the perspective of a typical query point $q$, on average (thus explaining the notation chosen).

**tails of the distance distributions**

The tails of the distance distributions discussed here have some importance. In particular the tails of $P_q(R, r)$ play an important logical role. replace the appropriate 0’s and ∞’s with the tails once good symbols are defined.
6.1.2 predicting query ranges for $k$-nn search

An important part of acquiring the expected performance of a metric tree is knowing apriori the expected query parameters. In particular, one should know what sort of queries will be asked, and what the expected limits are on these queries. Generally, a domain expert organizing the database will expect similarity queries with some maximum range (anything beyond that range is usually a “don’t care”). Given the distribution of the data items, this can be translated also into an expected maximum $k$ for $k$ nearest neighbor queries. Ultimately, this information will be used to tune the search tree during construction to provide the best performance possible. Since $k$ nearest neighbor queries can be reduced to range queries (as shown in lemma 3.5.2), the anticipated search range is used as the tuning parameter. So, for data sets where only range queries are expected, the remainder of this section is unneeded.

For instances where $k$ nearest neighbor queries are expected though, the expected $k$ needs to translated into an expected range. This expected distance to the $k$-th nearest neighbor for a typical query will be denoted $E(d(q, NN_k(q)))$, and can be found through:

$$n \int_0^{E(d(q, NN_k(q)))} P_q(S, r)dr = k. \quad (6.1)$$

While this method does provide the expected search radii, it does not provide any information on how much this expected value varies. Another way to compute this expected value is to take the subset (call it $H_\epsilon(k, S)$) of $H_q(S)$ that represents each data point’s distance to its $k$-th nearest neigh-
bor \( H_\epsilon(k, R) = \{d(u, NN_k(u))|u \in R\} \). Once again, a histogram can be created, scaled, and viewed as a probability density function (in this case \( P_\epsilon(k, R, r) \)). Using this method, the expected search radius computed as:

\[
E(d(q, NN_k(q))) = \int_0^\infty rP_\epsilon(k, S, r)dr. \tag{6.2}
\]

Of course, the advantage to this method over the previous one is that both the probability and variance statistics of the \( k \)-th nearest neighbor being at any particular distance can be computed for a typical query point.

For the remainder of the work, it is assumed that when analyzing performance, an expected search radius (\( \epsilon \)) will be given and fixed (as if the method described by equation 6.1 was chosen). The analysis can be extended mathematically to consider the variance of the computed \( \epsilon \) value, but this adds unneeded complexity for this presentation.

### 6.1.3 search events

Now that the search radius is available, a discussion on the probability of pruning is given. Fundamentally, a tree’s performance is simply a function of the probabilities that certain branches are traversed during a search.

To simplify the presentation of the analysis, the trees considered are space-covering and data items are only indexed under a single node at each level\(^2\). These are not a needed assumptions, but they greatly simplify the

\(^2\)For trees that have mutually exclusive subregions this is not a concern, but for trees that can have region overlaps this states that a data item falling in this intersection is only a child of one node in the tree.
mathematics. It will become clear to the reader while continuing through this section that the presentation can be extended to trees that do not guarantee these restrictions.

the simple case: analyzing a single internal node

To see the general strategy, one should first consider the search process at a single internal node itself. Once the search has proceeded to any internal node (including the root), there is a limited set of search events that can take place relative to that node. A search event is defined as a possible outcome of the search after the pruning strategy has been employed. This is best illustrated through an example. Consider an internal node with three children (each representing a subregion), indexed by a vantage point $v$. The entire set of possible outcomes is that only a single child is traversed (which can happen three different ways), two children are traversed (which, for a VP-Tree can happen only two different ways), or all three children are traversed.

Associated with each event is its particular probability of occurring, which can be computed from the distance distributions\(^3\). Each event’s probability becomes simply a function of the distance between $v$ and the query item $q$. Also associated with each event is the amount of pruning achieved by eliminating traversal through any children (if any is achieved at all). For the analysis of a single node, this pruning can be measured as the fraction of the data items indexed by that node that are not visited. This can also be computed from the distance distributions. The expected amount of pruning

---

\(^3\)Recall the assumption that query items distribute similarly to data items.
that an internal node will provide during a search can then be computed by
taking the product of each event’s probability of occurring and the amount
of pruning achieved for that event, and summing these products over all
possible events. An example best illustrates this.

**Example 10.** Consider an internal node in a VP-Tree (space-covering) rep-
resenting a region indexed by some vantage point $v$ with three children, and
let $m_0$ and $m_1$ (for medians) represent the distance from $v$ to the region
boundaries. (Also, for simplicity, let no data item be directly on a region
boundary.) Now, let the sub-region closest to $v$ be called $R_0$, and the region
farthest from $v$ be called $R_2$, while the middle region is called $R_1$. (For all
points $x, y, z$ such that $x \in R_0$, $y \in R_1$, and $z \in R_2$, the following holds:
$d(v, x) < d(v, y) < d(v, z)$. This can also be represented as $d(v, x) < m_0$,
$m_0 < d(v, y) < m_1$, $d(v, z) > m_1$, and $m_0 < m_1$.)

Once a search path has proceeded to this region, exactly six possible search
events can occur. In one case the search proceeds into only $R_0$, and simi-
larly two more cases for $R_1$ and $R_2$ each. Two cases involve proceeding into
only two regions (either $R_0$ and $R_1$ or $R_1$ and $R_2$). (Note that the case of
proceeding into only two regions being $R_0$ and $R_2$ is not possible.) The sixth
case is when the search proceeds into all three children and no pruning is
achieved. The notation adopted to represent the probability of each event is
$P\{X\}$, where $X$ is the union of the regions included in that search event.
(So $P\{R_0 \cup R_1\}$ is the probability of the search traversing into both $R_0$ and
$R_1$’s representative nodes.)
Given the fixed $\epsilon$, the notation $R_\epsilon$ will be used to indicate a region in a tree that includes all data items present in $R$, as well as those data items that are within distance $\epsilon$ of the boundary of $R$. Recall the assumption that query items distribute similarly to the data items in the search space. If a search has proceeded into some region $R$, it is because pruning (further up in the tree) of that region was not possible due to a region intersection between $R$ and the $q$-$\epsilon$-ball. This indicates that the query item may be inside $R$’s boundary, or within $\epsilon$ of $R$’s boundary. To compute the expected distance between $R$’s pivot $v$ and $q$, one must then examine the probability distribution $P_{vp}(R_\epsilon, v, r)$. Existing methods that look at statistical properties of regions neglect this fact and predict the query’s distance from the pivot using $P_{vp}(R, v, r)$ only, which is not accurate. Refer to figure 6.1 to see this visually.

Computing the expected pruning amount of the node then becomes a matter of identifying mathematically from the distributions when each search
event is possible, and how much pruning is achieved for each event. Let $\Pr(R, v, \epsilon)^4$ denote the measured pruning amount as a fraction of region $R$’s total data item cardinality, when using vantage point $v$ and a search range of $\epsilon$. Thus $E(\Pr(R, v, \epsilon))$ is the expected amount of pruning achieved by this node. The resulting equations can be found in figure 6.2\(^5\).

Note that for other sorts of metric trees, and even for VP-Tree nodes with a different number of children, equation 6.9 will be different, but of precisely the same flavor.

the extended case: analyzing the entire tree

The analysis of an entire vantage point tree’s performance is of the exact same flavor as that for an internal node. Instead of considering only the possible search events at each node of the tree, one examines all the possible search events for a complete search throughout the tree. Such an event is called a tree search event.

One additional component is needed though (as compared to analyzing a single node): a measurement of the amount of space that each region

\(^4\)Note the difference between $\Pr()$, $P()$, and prune():

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pr()$</td>
<td>the amount of pruning achieved</td>
</tr>
<tr>
<td>$P()$</td>
<td>probability distribution function</td>
</tr>
<tr>
<td>$P{}$</td>
<td>probability of a search event</td>
</tr>
<tr>
<td>prune()</td>
<td>the prune value of a region and a query item</td>
</tr>
</tbody>
</table>

\(^5\)Note that the amount of pruning when all three regions are searched is zero, and that case is not shown
\begin{align*}
P\{R_0\} &= \begin{cases} 
\int_0^{m_0-\epsilon} P(R, v, r)dr & \text{if } m_0 > \epsilon \\
0 & \text{if } m_0 < \epsilon 
\end{cases} \quad (6.3) \\
P\{R_1\} &= \begin{cases} 
\int_{m_0+\epsilon}^{m_1-\epsilon} P(R, v, r)dr & \text{if } 2\epsilon < m_1 - m_0 \\
\int_{m_0+\epsilon}^\infty P(R, v, r)dr & \text{if } 2\epsilon > m_1 - m_0 
\end{cases} \quad (6.4) \\
P\{R_2\} &= \int_{m_1+\epsilon}^\infty P(R, v, r)dr \quad \text{always} \quad (6.5) \\
P\{R_0 \cup R_1\} &= \begin{cases} 
\int_{\max\{0, m_0-\epsilon\}}^{m_1-\epsilon} P(R, v, r)dr & \text{if } 2\epsilon > m_1 - m_0 \\
\int_{\max\{0, m_0-\epsilon\}}^{m_0+\epsilon} P(R, v, r)dr & \text{if } 2\epsilon < m_1 - m_0 
\end{cases} \quad (6.6) \\
P\{R_1 \cup R_2\} &= \begin{cases} 
\int_{m_1-\epsilon}^{m_1+\epsilon} P(R, v, r)dr & \text{if } 2\epsilon > m_1 - m_0 \\
\int_{m_1-\epsilon}^{m_0+\epsilon} P(R, v, r)dr & \text{if } 2\epsilon < m_1 - m_0 
\end{cases} \quad (6.7) \\
P\{R_0 \cup R_1 \cup R_2\} &= \begin{cases} 
\int_{\max\{0, m_1-\epsilon\}}^{m_0+\epsilon} P(R, v, r)dr & \text{if } 2\epsilon > m_1 - m_0 \\
0 & \text{if } 2\epsilon < m_1 - m_0 
\end{cases} \quad (6.8) \\

E(Pr(R, v, \epsilon)) &= P\{R_0\} \cdot \int_{m_0}^\infty P(R, v, r)dr + \\
P\{R_1\} \cdot \left( \int_0^{m_0} P(R, v, r)dr + \int_{m_1}^\infty P(R, v, r)dr \right) + \\
P\{R_2\} \cdot \int_0^{m_1} P(R, v, r)dr + P\{R_0 \cup R_1\} \cdot \int_{m_1}^\infty P(R, v, r)dr + \\
P\{R_1 \cup R_2\} \cdot \int_0^{m_0} P(R, v, r)dr + P\{R_0 \cup R_1 \cup R_2\} \cdot 0 \quad (6.9)
\end{align*}

Figure 6.2: Equations for example 10.
represents. The notation $\Psi(R)$ will be used to represent the fraction of all the data items in the entire search space $S$ that region $R$’s node indexes. (Thus, $\Psi(S) = 1$.) Revisiting the previous example, if one were to find out how much pruning relative to the entire search space is achieved, the result of equation 6.9 would be multiplied by $\Psi(R)$ (giving its weighted expected pruning amount).

The following lemmas are useful in understanding how to work with tree search events.

**Lemma 25.** If an internal node represented by region $R$ of a space-covering tree has been included in a search path, then at least one leaf node descendant of the internal node must also be included in the search path. (Alternatively said, no search path can terminate at an internal node.)

**Proof.** Assume that a search path has reached an internal node representing region $A$ (thus the $q$-$\epsilon$-ball must intersect with $A$). Furthermore, assume the search path has terminated at this node. This implies that the $q$-$\epsilon$-ball does not intersect with any of $A$’s sub-regions (children of the node). Since the tree is space-covering, this implies that the $q$-$\epsilon$-ball does not intersect with region $A$. Thus the search would never have reach $A$’s node, and a contradiction is reached.

**Lemma 26.** Given a space-covering metric tree with mutually exclusive sub-regions, every tree search event includes visiting some collection of leaf nodes, although not every collection of leaf nodes needs to corresponds to a tree search event.
Proof. Since search paths cannot terminate at an internal node (lemma 25), it directly follows that a collection of leaf nodes will be reached for every unique tree search event.

For the second part of the lemma, consider an internal node with three children in a VP-Tree (and the same notation will be used from example 10). Let the search path reach this node (region $R$). Also let the search path reach a descendant of $R_0$ and $R_2$, but not $R_1$. This implies that $d(v, q) + \epsilon < m_0$, $d(v, q) - \epsilon > m_1$, and that $2\epsilon > m_1 - m_0$, which is clearly impossible since $\epsilon > 0$ and $m_0 < m_1$. A contradiction is thus formed.

Given a tree search event, its probability of occurring can be computed by examining each internal node’s local search event that contributes to the tree. Basically, every tree search event results in a “coloring” of a tree, where each visited internal node is colored. Associated with each colored internal node is then its own local search event probability (one just needs to look at which of its children are colored to determine the event). Taking the product of the search event probabilities of all the colored nodes then gives the probability of the full tree search event.

The amount of pruning achieved can be found by examining how many leaves are not visited in the tree search event\(^6\). Taking the product of these two values (the amount of pruning provided by, and the probability of that tree search event) and summing over all possible tree search events results in

\(^6\)Look at all internal nodes (regions) that are parents of leaf nodes (data items), and sum the number of data items contained in the regions never visited during the tree traversal.
the expected pruning for a given tree\textsuperscript{7}.

\textbf{in terms of distance computations: comparing trees and nodes without experiments}

Using the analysis described above it is clear that one can obtain, for every instance of a metric tree, a quantitative value that is the expected amount of pruning the tree will provide on average. Of course, all of this pruning comes at a cost: each internal node visited results in a distance computation (unless pivots are \textit{recycled} and used again elsewhere in the tree). When computing the performance one can expect for a node, the pruning provided by each event can be scaled by the number of distance computations that the event results in at the next level of the search (i.e. divide by the number of children traversed in that search event).

With this, given two vantage point trees (and their respective average pruning levels) that index the same search space, one can decide which of the two offers better performance without running a single experiment. Unfortunately, performing an exact comparison is most often not feasible. For one, the number of possible tree search events is quite large. Even for VP-Trees

\textsuperscript{7}The reason for the space-covering and mutually exclusive subregion restrictions should now be clear to the reader. Without the space-covering assumption, searches can terminate at internal nodes, adding to the number of possible tree search events to consider. Without the disjoint set-membership assumption, computing the amount of pruned data for each internal node requires examining if any of the data items are present under any other node, which may require a separate lookup table for each data item.
with many restrictions (such as the tree being balanced with fixed fanout for internal nodes) the number of possible tree search events is exponential in the size of the data.

In addition to the astronomical number of tree search events, computing the expected pruning amount exactly requires the storage of all distance computations performed during construction of the metric tree. Consider the primary advantage of a metric tree: (hopefully) sub-linear query times while remaining linear in storage space. Using the vantage point tree example, the storage required would be at least $O(n \log n)$ (where $n$ is the number of data items in the search space). If this much memory is readily available, other methods (like LAESA [61]) are readily available.

Given this Catch-22 (the only way to absolutely decide which of two trees is better is to have enough information such that the trees would not be needed), one should focus on the fact that it is possible to compare two trees analytically. That this comparison is difficult is a separate notion. Sampling can then be used as a means for approximate expected performance computations.

---

8In fact, it requires more than this, to account for the $P_{vp}(R, v, r)$ distribution (as described in the example 10)

9Note that most of the analysis done here for metric trees can be extended to studying metric arrays. Such extensions are currently being investigated.
6.1.4 optimal metric trees

The conclusion of the previous section (that two trees are directly comparable without experiments) has a logical next step: there exists an optimal tree for any search space. Thus the problem of index creation can be worked on as an optimization problem, albeit a complex one. The complexity becomes apparent even when considering a small section of a tree.

For example, consider a single internal node in a metric tree and the method for computing the expected performance (as in example 10, in particular the parameters used in equation 6.9). The selection of the representative pivot for that node is one factor that influences the performance of search at that node. Yet another factor is the number of children. Furthermore, the cardinality of data items indexed by each child also plays a role in the computation of the expected pruning achieved at this single internal node.

Note that if a pivot appears as a descendant of itself, it is recycled (a common technique to save distance computations [63]) and the distance computations can be saved from the first use and referenced later (although this once again imposes the need for extra memory during search and an external lookup table keeping track of such occurrences). This adds further difficulty in trying to find an optimal node (as one must keep in mind that a recycled descendant will be “free” if the correct previous distance computations are saved).

One can see immediately that measuring the performance of a node is impacted by the performance of each descendant of that node (all the way
down to the leaves). This tends to point towards a possible dynamic programming approach, but the complexity is still too great to compute directly. Instead, the goal of acquiring better trees relies on heuristics (as all existing methods do), although now the use of a probabilistic analysis can improve this process.

6.2 PVP-Trees

Directly using the topics covered in the previous section, and in particular the example given, the Probabilistic Vantage-Point (PVP) Tree is now presented. Rather than selecting pivots through the “second moment about the median” strategy employed in traditional VP-Trees, PVP-Trees rely on a probabilistic analysis to select pivots that will decrease the expected search cost for expected queries\(^{10}\). Moreover, the PVP-Tree differs from VP-Trees also in the method for choosing the region boundaries, and somewhat unbalanced trees are not uncommon as a result.

6.2.1 PVP-Tree construction

The PVP-Tree construction algorithm is founded on two optimization problems, one nested in another. Similar to the VP-Tree, the selection of a “good”

\(^{10}\)It is a standard assumption in multidimensional indexing that over time query items distribute identically to the dataset being searched. Thus, \textit{expected} query properties (such as the expected distance to the nearest neighbor) can be determined by a statistical analysis of the dataset.
pivot amongst a set of candidates is one of the optimization problems (the outer one). The nested problem selects “good” region divisions for each candidate pivot. Note that with traditional VP-Trees, the region divisions for a node are selected such that the cardinality of each child region is equal (or approximately so), resulting in a balanced tree.

Both the selection of pivots and the accompanying region division values require a “tuning factor” of $\epsilon$, which will be the expected search radius (assumed to be known, or at least upper-bounded, at construction time). A similar tuning approach has led to other improvements on VP-Trees [89, 90]. Note that even for $k$-NN queries such a parameter is available after sampling pairwise distances between items in the dataset.

**Selection of region boundaries**

Given a candidate pivot object $p$ in the current region $R$, the task of the inner problem is to identify $l - 1$ region boundaries (for a node with $l$ children) that will be used to partition the children nodes that index (collectively) $R \setminus \{p\}$. Recent results [81] describe a probability density function (PDF) $f : R, S^*, [0, \infty) \mapsto [0, 1]$ which is defined by $f(p, R, r)$ representing the probability of finding an item in $R$ that is distance $r$ away from $p$ (and thus provides a picture of the density of the region $R$ from $p$’s perspective). In the same work, a “search event” at an internal node is described as any possible coloring of the children of that node during a search. Assuming the region boundaries have been provided, the function $f$ is then used in the following manner:
1. The probability of the query being at any distance from \( p \) can be computed from \( f(p, R, r)^{11} \).

2. All possible search events (e.g. for a tight binary tree, searching the left child only, the right child only, both children, or neither children) then have a probability of occurring based on the location of the query interval (the \( 2\epsilon \)-wide region centered at the query’s distance to \( p \)). An example of a search event and its corresponding probability of occurring is given in figure 6.3. Each search event also offers some pruning count (i.e. the total number of items indexed under each non-searched child).

3. The probability of each search event is multiplied by the pruning count of each search event, and the results are accumulated. This final value is the expected amount of pruning achievable using \( p \) as the pivot with the given region boundaries.

Since this is the inner optimization problem, \( p \) is fixed and thus one must simply compare (via the steps just listed) various candidate locations for region boundaries. Sampling is used to obtain a number of candidate configurations for the node, and each is measured to obtain the expected pruning value. The configuration with the greatest such value is then selected and returned (along with that prune value) to the outer optimization problem.

\[ \text{11In fact this is more accurately computed using the distances between } p \text{ and all items in } R, \text{ as well as a few items that are just outside of } R. \text{ For the sake of simplicity, this is omitted here (as it does not seem to make a significant change in the result). A better description of this is available in a previous work [81].} \]
In order to compute the expected amount of pruning a node can provide, one must compute the probability of each possible search event. This probability can be computed based off of the probability density function \( f(p, R, r) \), where \( R^\epsilon \) represents all the data objects in the current region \( R \) plus all of the objects that are within \( \epsilon \) of \( R \)'s boundary. The additional elements are needed to accurately predict the distribution of expected queries to \( p \), since a search will progress into the node for \( R \) if the query object is within \( \epsilon \) of \( R \).

The remainder of this section relies on a few terms which will now be
defined. Any two nodes in a VP-Tree that have the same parent are called *siblings*. An internal has at least one child node, with multiple children being separated by boundary values (as described section 4.5.1). These boundaries are all real values falling in the interval $[0, \infty)$.

**Lemma 27.** If a search progresses into more than one child of an internal node, all of the children must be adjacent.

*Proof.*

**Lemma 28.** For an internal node indexing region $R$ with $k$ children (existing as loose subregions), there are exactly $\sum_{i=1}^{k} i$ search events.

*Proof.* Since the children subregions are loose, a search that has progressed to the internal node cannot terminate at that node. This means that the search must progress into at least one child node. There are $k$ children nodes, so if the search progresses into exactly one of them, there are $k$ possible search events. If the search progresses into exactly two children, they must be adjacent (as shown by lemma 27). This means there are exactly $k - 1$ possible two-child events. The same principle can be applied to show that there exist exactly $k - 2$ possible three-child search events, and thus in general exactly $k - l$ possible $l + 1$-child search events. There is then exactly 1 possible $k$-child search event. The summation over all possible search events then is $\sum_{i=1}^{k} i$. □

If $k$ is large then, there can be quite a few search events. Luckily, there exist exactly twelve cases than can be used to compute the probability for
each possible search event (no matter how many search events there are).
The principle here is that some search events can be generalized by others.
Each of the twelve search events are now described, with the assumption that
the children regions are loose. Thus the division values for the $k$ children are
represented as $k - 1$ values $v_1, v_2, v_3, \ldots, v_{k-1}$ (where the leftmost child is
bounded then by the interval $[0, v_1)$, the next child to the right by $[v_1, v_2)$,
and so on until the rightmost child is bounded by $[v_{k-1}, \infty)$). Children nodes
are considered adjacent if they share a boundary value.

**Definition 34** (search event L1). The $L1$ search event represents the case
where the search progresses into exactly one child, and that child has only a
right sibling. Letting the right boundary of this child be $v_1$, the probability of
this search event can be modeled as:

$$P\{L1\} = \int_{0}^{v_1-\epsilon} f(p, R^\epsilon, r) dr.$$  \hspace{1cm} (6.10)

**Definition 35** (search event M1). The $M1$ search event represent the case
where the search progresses into exactly one child, and that child has both a
left and a right sibling. Letting the left and right boundaries for this child be
$v_i$ and $v_{i+1}$ respectively, the probability of this search event can be modeled
as:

$$P\{M1\} = \int_{v_i+\epsilon}^{v_{i+1}-\epsilon} f(p, R^\epsilon, r) dr.$$  \hspace{1cm} (6.11)

**Definition 36** (search event R1). The $M1$ search event represent the case
where the search progresses into exactly one child, and that child has only a
left sibling. Letting the right boundary for this child be $v_{k-1}$, the probability
of this search event can be modeled as:

\[ P\{R1\} = \int_{v_{k-1}+\epsilon}^{\infty} f(p, R^c, r)dr. \]  \hspace{1cm} (6.12)

**Definition 37** (search event LMR1). The *LMR*1 search event represents the case where the search progresses into exactly one child, and that child has no left or right siblings. This is a trivial case, as it represents a situation that can only arise when a node has exactly one child. In this case, the search must always progress into this node, and thus the probability of this trivial search event is always 1.

\[ P\{LMR1\} = 1 \]  \hspace{1cm} (6.13)

**Definition 38** (search event L2). The *L*2 search event represents the case where the search progresses into exactly two adjacent children, and that the leftmost of these children has no left sibling, while the rightmost of these children does have a right sibling. Letting \( v_1 \) and \( v_2 \) represent the right boundaries for the leftmost child and the rightmost child respectively, the probability of this search event can be modeled as:

\[
P\{L2\} = \begin{cases} 
\int_{v_1}^{v_2} f(p, R^c, r)dr & \text{if } \epsilon \geq v_1 \text{ and } 2\epsilon \leq v_2 - v_1 \\
\int_{v_1+\epsilon}^{v_2+\epsilon} f(p, R^c, r)dr & \text{if } \epsilon \geq v_1 \text{ and } 2\epsilon > v_2 - v_1 \\
\int_{v_1-\epsilon}^{v_2-\epsilon} f(p, R^c, r)dr & \epsilon < v_1 \text{ and } 2\epsilon > v_2 - v_1 \\
\int_{v_1+\epsilon}^{v_2+\epsilon} f(p, R^c, r)dr & \epsilon < v_1 \text{ and } 2\epsilon \leq v_2 - v_1 
\end{cases} \]  \hspace{1cm} (6.14)

**Definition 39** (search event M2). The *M*2 search event represents the case where the search progresses into exactly two adjacent children defined by the boundaries \( v_i, v_{i+1}, \) and \( v_{i+2}, \) and both children have another adjacent sibling.
that is not one another. Given those boundaries, the probability of this search event can be modeled as:

\[
P\{M2\} = \begin{cases} 
0 & \text{if } 2\epsilon > v_{i+2} - v_i \\
\int_{v_{i}+\epsilon}^{v_{i+2}-\epsilon} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon \geq v_{i+1} - v_i \text{ and } 2\epsilon \geq v_{i+2} - v_{i+1} \\
\int_{v_{i+1}+\epsilon}^{v_{i+2}} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon \geq v_{i+1} - v_i \text{ and } 2\epsilon < v_{i+2} - v_{i+1} \\
\int_{v_{i+1}+\epsilon}^{v_{i+2}-\epsilon} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon < v_{i+1} - v_i \text{ and } 2\epsilon \geq v_{i+2} - v_{i+1} \\
\int_{v_{i+1}+\epsilon}^{v_{i+2}} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon < v_{i+1} - v_i \text{ and } 2\epsilon < v_{i+2} - v_{i+1} 
\end{cases}
\]

(6.15)

**Definition 40** (search event R2). The L2 search event represents the case where the search progresses into exactly two adjacent children, and that the rightmost of these children has no right sibling, while the leftmost of these children does have a left sibling. Letting \(v_i\) and \(v_{i+1}\) represent the left boundaries for the leftmost child and the rightmost child respectively, the probability of this search event can be modeled as:

\[
P\{R2\} = \begin{cases} 
\int_{v_{i+1}+\epsilon}^{v_{i+2}-\epsilon} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon \geq v_{i+1} - v_i \\
\int_{v_{i+1}+\epsilon}^{v_{i+2}} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon < v_{i+1} - v_i 
\end{cases}
\]

(6.16)

**Definition 41** (search event LMR2). The LMR2 search event represents the case where the search progresses into exactly two adjacent children, neither of those children have another sibling (other than each other). Thus the LMR2 case can exist only when there are exactly two children. Letting \(v_1\) be the shared boundary between the two children, the probability of this search event can be modeled as:

\[
P\{LMR2\} = \begin{cases} 
\int_{0}^{v_{1}+\epsilon} f(p, R^\epsilon, r) dr & \text{if } \epsilon \geq v_1 \\
\int_{v_{1}-\epsilon}^{v_{1}+\epsilon} f(p, R^\epsilon, r) dr & \text{if } \epsilon < v_1 
\end{cases}
\]

(6.17)
Definition 42 (search event L3). The L3 search search event represents the case where the search progresses into three children (with the rightmost and the leftmost child both being adjacent to the middle child), and the leftmost child has no left sibling while the rightmost child does have some other sibling. Letting \( v_1, v_2, \) and \( v_3 \) be right boundaries for each child node respectively, the probability of this search event can be modeled as:

\[
P\{L3\} = \begin{cases} 
0 & \text{if } 2\epsilon < v_2 - v_1 \\
0 & \text{if } \epsilon > v_3 \\
\int_{v_1+\epsilon}^{v_1-\epsilon} f(p, R^\epsilon, r) dr & \text{if } \epsilon \geq v_2 \text{ and } 2\epsilon \geq v_3 - v_1 \\
\int_{v_2-\epsilon}^{v_2+\epsilon} f(p, R^\epsilon, r) dr & \text{if } \epsilon < v_2 \text{ and } 2\epsilon \geq v_3 - v_1 \\
\int_{v_3-\epsilon}^{v_3+\epsilon} f(p, R^\epsilon, r) dr & \text{if } \epsilon < v_2 \text{ and } 2\epsilon < v_3 - v_1 \\
\int_{v_2-\epsilon}^{v_2+\epsilon} f(p, R^\epsilon, r) dr & \text{if } \epsilon < v_2 \text{ and } 2\epsilon < v_3 - v_1 \\
\end{cases}.
\] (6.18)

Definition 43 (search event M3). The M3 search event represents the case where the search progresses into three children and the leftmost and rightmost children each have adjacent siblings that are not being searched. Letting \( v_i, v_{i+1}, v_{i+2}, \) and \( v_{i+3} \) be the boundaries defining the children regions, the probability of this search event can be modeled as:

\[
P\{M3\} = \begin{cases} 
0 & \text{if } 2\epsilon > v_{i+3} - v_i \\
0 & \text{if } 2\epsilon < v_{i+2} - v_{i+1} \\
\int_{v_i+\epsilon}^{v_i+\epsilon} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon \geq v_{i+2} - v_i \text{ and } 2\epsilon \geq v_{i+3} - v_{i+1} \\
\int_{v_{i+1}-\epsilon}^{v_{i+1}+\epsilon} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon \geq v_{i+2} - v_i \text{ and } 2\epsilon < v_{i+3} - v_{i+1} \\
\int_{v_{i+2}-\epsilon}^{v_{i+2}+\epsilon} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon < v_{i+2} - v_i \text{ and } 2\epsilon \geq v_{i+3} - v_{i+1} \\
\int_{v_{i+1}-\epsilon}^{v_{i+1}+\epsilon} f(p, R^\epsilon, r) dr & \text{if } 2\epsilon < v_{i+2} - v_i \text{ and } 2\epsilon < v_{i+3} - v_{i+1} \\
\end{cases}.
\] (6.19)
**Definition 44** (search event R3). The R3 search event represents the case where the search progresses into three children and the leftmost child has a left sibling not being searched while the rightmost child has no right sibling. Letting \( v_i, v_{i+1}, v_{i+2} \) be the boundaries defining the children regions, the probability of this search event can be modeled as:

\[
P\{\text{R3}\} = \begin{cases} 
0 & \text{if } 2\epsilon < v_{i+2} - v_{i+1} \\
\int_{v_i+\epsilon}^{v_{i+1}+\epsilon} f(p, R^x, r) \, dr & \text{if } \epsilon \geq v_{i+2} - v_i \\
\int_{v_{i+2} - \epsilon}^{v_{i+1} + \epsilon} f(p, R^x, r) \, dr & \text{if } \epsilon < v_{i+2} - v_i
\end{cases}
\]  

(6.20)

**Definition 45** (search event LMR3). The LMR3 search event represents the case where a node has exactly three children and the search progresses into all three. Letting \( v_1 \) and \( v_2 \) be the boundaries defining these children regions, the probability of this search event can be modeled as:

\[
P\{\text{LMR3}\} = \begin{cases} 
0 & \text{if } 2\epsilon < v_2 - v_1 \\
\int_0^{v_1 + \epsilon} f(p, R^x, r) \, dr & \text{if } \epsilon \geq v_2 - v_1 \\
\int_{v_2 - \epsilon}^{v_1 + \epsilon} f(p, R^x, r) \, dr & \text{if } \epsilon < v_2
\end{cases}
\]  

(6.21)

**Definition 46** (search event L4+). The L4+ search event represents the case where the search has progressed into 4 or more children, and the leftmost child has no left sibling while the rightmost child does have an unsearched right adjacent sibling. The probability of this search event can be modeled using the L3 model.

**Lemma 29.** The probability of an L4+ search event can be modeled using the same equations that model the L3 search event.

154
Proof. Note that in the L3 search event, the range of the interval always span the entire middle region (i.e. the values from \(v_1\) to \(v_2\)). Similarly, in the L4+ search event, the range of the interval must always span the concatenation of the middle regions. If this were not to be the case, there would be a gap in the region coverage, which is not allowed in a VP-Tree (i.e. all covered regions by the interval must be adjacent to each other). Thus, if the L4+ event spans \(n\) regions, the \(n - 2\) regions must always be spanned, and thus can be concatenated to represent a single region. Doing this creates an L3 search event.

**Definition 47** (search event M4+). The M4+ search event represents the case where the search has progressed into 4 or more children, and the leftmost child has an left adjacent sibling that is unsearched and the rightmost child has a right adjacent sibling that is unsearched. The probability of this search event can be modeled using the M3 model.

**Lemma 30.** The probability of an M4+ search event can be modeled using the same equations that model the M3 search event.

Proof. By the same argument used for proving the L4+ search event. □

**Definition 48** (search event R4+). The R4+ search event represents the case where the search has progressed into 4 or more children, and the leftmost child has an unsearched left adjacent sibling while the rightmost child has no right siblings. The probability of this search event can be modeled using the R3 model.
Lemma 31. The probability of an $R_4+$ search event can be modeled using the same equations that model the $R_3$ search event.

Proof. By the same argument used for proving the $L_4+$ search event. ⊓⊔

Definition 49 (search event LMR4+). The LMR$4+$ search event represents the case where the search has progressed into 4 or more children, and the leftmost child has no left siblings and the rightmost child has no right siblings. The probability of this search event can be modeled using the LMR3 model.

Lemma 32. The probability of an LMR$4+$ search event can be modeled using the same equations that model the LMR3 search event.

Proof. By the same argument used for proving the $L_4+$ search event. ⊓⊔

Selection of pivots

The outer portion of the nested problem is far simpler than the inner. As with traditional VP-Trees, a set of candidate pivots is obtained from the current region, then each is evaluated (as described in section 6.2.1). The pivot selected is the one that (along with its optimal region boundary configuration) provides the greatest amount of expected pruning. Note that this provides the expected performance that will be observed at this node during any search that reaches it. All remaining items are placed into the appropriate subregions (as returned from the inner optimization problem) and the procedure is called recursively on each child node.
PVP-Tree unbalancing and storage

Trees built using the described method may be (and indeed often are) unbalanced — as the construction method only seeks to increase the expected amount of pruning achievable at each node, regardless of balancing considerations. This unbalancing may at first seem to have an impact on storage costs, but for both Tight and Tight-Historical PVP-Trees, the storage requirements are nearly identical to that of traditional VP-Trees. Thus, the only additional cost incurred for construction comes from the probabilistic analysis, and not from increased distance computations. The analysis portion of the construction cost can be kept to a reasonable level provided proper sample sizes are used.

6.2.2 PVP-Tree searching

Searching in a PVP-Tree is identical to that in all single-pivot Metric Trees. Figure 6.4 illustrates the results of searching VP-Trees and PVP-Trees over identical datasets. For both random and non-random (image) datasets, the average query performance of the probabilistic trees outperformed that of the original trees. It is expected that as datasets are found that veer away from standard unimodal distance distributions the performance difference between the two types of indices will grow larger.
6.3 Interval matrices

The comparison of VP-Trees and the LAESA matrices based on a fixed storage cost lends itself to a more in-depth probing of the indices. Rather than simply looking at how much information about pairwise distances is stored, one can look at what information is stored. The stored distance information for both Metric Trees and matrices can be extracted into a more general matrix of pairwise distance intervals, which will then allow a further comparison of competing indexing methods.

A pairwise distance interval is simply an interval over the real values describing the known information about the distance between two objects (i.e. the interval represents the lower and upper bounds of the range of possible values the pairwise distance can take on). On one end of the spectrum, if the distance between objects \( a \) and \( b \) is precisely known, the interval can be described as \( d(a, b) \in [d(a, b), d(a, b)] \). Alternatively, cases where no bounds can be made on the distance between \( a \) and \( b \) can be represented as \( d(a, b) \in [0, \infty) \). Given a dataset of objects in a metric space, any information available about the pairwise distances can be represented as an upper triangular matrix holding \( \binom{n}{2} \) intervals. Moreover, such a pairwise distance interval matrix can be extracted from the stored information in both Metric Trees and LAESA-like matrices.
### 6.3.1 Generalized matrix searches

Once the information from any distance-based index structure has been extracted into a pairwise distance interval matrix, probabilistic techniques (similar to those used in PVP-Trees) can be used to search the matrix in an attempt to take advantage of the information without the need to subscribe to any particular search technique. The GMS (Generalized Matrix Search) algorithm is now presented as such a method.

**GMS algorithm**

The GMS algorithm works through elimination cycles (iterations) in a fashion quite similar to the LAESA search. At the start of each loop, all of the remaining (i.e. not already eliminated) data items serve as candidates for selection to undergo a distance computation against the query item. The selected data item is the one that is expected to provide the greatest amount of elimination during that iteration.

Computing the expected elimination for each candidate begins by creating a histogram estimating $f$ (recall from section 6.2.1 that $f$ is the probability density function of some items relative to a pivot). This probability density function approximation step differs from that observed with PVP-Tree construction in two ways:

1. There are no “regions” (corresponding to $R$ in the function $f(p, R, r)$) with interval matrices. Instead, $R$ serves as the set of intervals for distances between the candidate and all non-eliminated items (minus
the candidate itself).

2. The histograms are built from (possibly loose) intervals rather than precise values. Once the buckets for the histogram have been identified, the intervals are placed into them as with normal histogram construction. Intervals that span multiple buckets though, are broken into pieces with area < 1 representing the portion of the interval that falls into each bucket. Such a method assumes that the true distance is equally likely to take on any value within the interval bounds, and guarantees that the total area of the histogram is only increased by 1 upon each interval addition.

The computation of the expected amount of elimination now can be done (again in a method similar to that described in section 6.2.1) via the following steps:

1. The expected query location is first identified by computing the expected value of $f$.

2. An expected query interval (window) is created by expanding the expected query location on either side by some value $\epsilon$. This expansion amount is fixed for range queries, and is variable for $k$-nn queries (being set at each step to the distance to the current best $k$-th nearest neighbor).

3. The remaining intervals are all compared one at a time with the expected query interval. If the two intervals intersect, it is expected that
no elimination will occur for the data item corresponding to that interval. On the other hand, if the intervals are disjoint, it is expected that elimination will take place, and the total expected elimination count is incremented for each such occurrence.

4. The candidate with the greatest count of expected eliminations is selected. A distance computation between this candidate and the query item takes place, the current answer set is possibly updated, and elimination follows (similarly to the LAESA search, with both $G$ and $d^*$).

5. All steps are repeated until no candidate items remain to be selected.

A visual example of this process is given in figure 6.5.

**GMS applied on Metric Tree matrices**

Pairwise distance interval matrices obtained from Metric Trees with memory (e.g. Tight-Historical VP-Trees) are clear candidates for being searched via the GMS algorithm. Normal Metric Tree searches proceed using a very restrictive approach: no node can be reached (with exception of the root) prior to examining its parent. Searching such trees with the GMS algorithm breaks this restriction and searches trees without paths. This freedom leads to improved query performance, which can be seen in figure 6.6. The GMS algorithm knows nothing about the construction method of the matrix (tree-driven in this case), and instead searches over the stored information as it is freed from the restrictions of being trapped in tree-form.
When applied to extracted interval matrices from VP-Trees without memory, the performance difference is negligible (as the matrix is far sparser and contains quite a small amount of information).

**GMS applied on LAESA-built matrices**

Unlike a Metric Tree search, the LAESA requires a search parameter \( ec \). Since the GMS algorithm has no notion of the LAESA base set \( B \), the algorithm can only be reasonably compared to a search with \( ec = \infty \). This immediately exposes a weakness of the GMS algorithm when applied to interval matrices extracted from a LAESA construction: a large \( m \) will likely be required to offer good performance (similarly to a large \( m \) being needed to offer good \( ec = \infty \) search performance). The observation is verified in figure 6.7, where a LAESA matrix tuned for an \( ec = 1 \) search is explored by the GMS algorithm. It can be seen though, that the GMS algorithm remains highly competitive with the LAESA \( ec = \infty \) search on the same matrix, validating the power of the probabilistic search method.

**GMS overhead**

The computation overhead in picking the next data item to examine directly against the query is also a limiting factor on the use of the GMS algorithm in current systems. While it has been assumed that the distance computations are costly enough that they overshadow all other costs in the system, the probabilistic analysis can possibly make that claim false. Statistical sampling methods can be employed however, to reduce the cost of the expected
elimination computations.
Figure 6.4: Comparison between VP and PVP-Trees. Left plot shows query performance on random 9d data, right plot is for 16d image data. Solid-lined boxes indicate PVP-Tree performance, and dashed-lined boxes are for VP-Trees.
Figure 6.5: Example of GMS algorithm candidate evaluation. The query interval is centered at the probability function’s mean \( (\mu_1(0) = \int_0^\infty r \cdot f(p, R, r)dr) \). Intervals with an \( \times \) through them are expected to be eliminated (they do not intersect the expected query interval).
Figure 6.6: GMS algorithm executed on extracted contents of a Tight-Historical VP-Tree, along with performance of a standard tree search with pruning. The GMS algorithm clearly beats the tree search method executed on the exact same set of stored information.
Figure 6.7: Performance results of the GMS algorithm run over a LAESA-built matrix properly tuned for an $ec = 1$ search strategy. The GMS algorithm outperforms the LAESA search with $ec = \infty$, although as predicted the search with $ec = 1$ gives better results than both of the other methods.
Chapter 7

Conclusions

7.1 Contributions

The primary contributions in this work are summarized below:

• A framework for similarity search space trees is provided, upon which a classification system on trees can be built. The usefulness of this framework lies in the ability to recognize shared properties between two seemingly different data structures. An example of this is given as the next contribution.

• Algorithm 1 is given as a single and optimal method for answering both range and $k$-nearest neighbor queries in search space trees. While a similar algorithm has already been identified for R-Trees, the ability to apply such improvements in one structure to another is a novel contribution. As an example, even recent works [26] were still employing
a non-optimal search method for $k$-nearest neighbor searches in metric
trees. The algorithm has also been written in a general enough sense to
be easily adopted to any such structure, with only a single function that
needs to be re-written for each scenario (the prune value computation
function).

• The use of distance-based indexing for string similarity search has not
been used extensively prior to this work. The primary use of distance-
based indexing methods has been for image search.

• The ability to use indexing methods on a block edit distance approxi-
mation is a novel contribution. True block edit distances are NP-Hard
to compute, and thus approximations have been used for some time.
However, at the time these methods were considered heuristics as no
tight bound on the block edit distance was proven. Here, a method
(the compression distance) is shown to have a tight bound on a true
block edit distance.

• The compression distance is shown via a series of ordering experiments
to be consistent with another, already proven block edit distance ap-
proximation. This is a first step towards making such measures actively
used in the research community. In particular, the compression dis-
tance was used to verify previous results on phylogenies that employed
slightly different block edit distance approximations.

• Block edit distance works have traditionally used smaller datasets, pri-
marily because any search operation was performed using a brute-force technique. Here, the compression distance has been shown to be *almost* metric, meaning it satisfies the triangle inequality to within some constant multiplicative factor. This allows the employment of distance-based indexing methods (with some minor alterations to handle the constant factor in the triangle inequality) to search through a space using the compression distance.

- A probabilistic analysis of metric tree nodes is shown to be possible by simply enumerating search events. This allows nodes to be qualitatively compared without the need for experiments. This analysis is also shown to be possible for entire trees, allowing entire search structures to be compared for expected performance without ever performing an experiment (although such a comparison is intractable, which is why individual nodes are compared instead).

- The probabilistic analysis leads directly to a probabilistic construction method for single-pivot metric trees (VP-Trees). This probabilistic VP-Tree (the PVP-Tree) employs a greedy algorithm and sampling to compute the expected performance of each candidate pivot and pivot configuration. Previous construction methods employed heuristics that were only shown to be optimal for distributions that were unimodal and balanced, and only for binary trees (i.e. nodes with two children). The probabilistic construction method however is more adaptable to various distance distributions and higher-order fanouts.
• For distance-based indexing solutions, matrix-based methods and metric trees are compared for the first time in both performance and space requirements. In particular, the space-performance tradeoff has been known for some time (rather intuitively), but no extensive comparison of the two methods has been performed. Here such an analysis is done, providing a search method selection strategy based on available storage space and performance requirements.

• A probabilistic search method on generalized interval matrices is introduced. This is an information-theoretic search method that can be applied to search any distance-based index (tree or matrix). It simply attempts to search based on the available pairwise distance information, and is equipped to handle an interval of possible distances for any pair of data items (i.e. it doesn’t need exact distances), and is thus particularly suitable for “fuzzy” data, where the distance between two items is known to be bounded above and below by values, while the precise value is unknown. This algorithm is then shown to outperform the optimal search algorithm on data extracted from a metric tree for the same query. Thus, learning to search the available information, without the bounds of a tree structure, is a viable alternative to tree search algorithms.
Bibliography


[56] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *STOC*


