A Code Search Engine for Go

Harrison Brewton
Advised by Aws Albarghouthi
May, 2020

Creating indexes to aid search through structured data is a finicky business, and it is especially hard to make these indexes achieve sub-polynomial search time. This paper looks into problem of automatically creating an index for arbitrary user defined data types. It details a procedure for the creation of a metric from data types. It then shows how to use this metric to create an index that provides $O(\log(n))$ bounded search. The modularity of this system allowed the creation of a package *type2met*, which automatically creates metrics from given types. We then use this module to create a fast search engine for Go functions based on their signature. We demonstrate its use on a number of packages from the Go standard library, showing that we get good search results and we get strong speed up over linear search.

1 Background

1.1 Metric

Metrics provide a sensible definition of distance: the distance from an object to itself is none, the distance between and object and another is same in either direction, and the distance between an object and another is the shortest way to get there. Metrics can be formalized simply:

Definition 1 (Metric)**.** *Let (X, d) space, such that $d : X \times X \to \mathbb{R}^{\geq 0}$. Then d forms a metric over X if three properties hold: symmetry or $d(a, b) = d(b, a)$, identity of indiscernibles or $d(a, b) = 0 \iff a = b$, and sub-additivity $d(a, b) \leq d(a, c) + d(c, b)$ for all c.*

Because of this, they are often used for searching for the nearest objects to some query. They've been used for image processing, computational biology, computer vision, melody search, amongst others [1]. While useful, metrics for particular problem spaces can be difficult to create especially when dealing with complicated data types commonly found in modern databases. The first goal of this paper is to provide an automatic mechanism which allows a user to provide a type definition, and from this definition produce a metric which allow the user to compute similarity. This metric can then be used for further applications such as threshold search, repair, and possibly synthesis. We implement this in a tool called Type2Metric.

1.2 Code Search

Code reuse has long been a goal for tool designers. It allows engineers to collectively work on performance, security, robustness, and further reuse. One part of code reuse is being

able to search through a large code base and find functions that have already been created. Previous work [2] has looked as using raw source code. Other work [3] has instead used an ad-hoc rewriting scheme to find the difference between pieces of code, requiring linear time to search through the code. Our second goal is to create a code search engine for Go. We use the aforementioned metric generation to create a metric on type signatures, then we can use this metric to create an efficient data structure for search. This data structure both provides the logarithmic search complexity of [2] but with the richer type information of [3]. We implement this search engine in a tool called Axe.

1.3 Implementation and Evaluation

We have implemented our Type2Metric and Axe in Go. Type2Metric allows users to use normal structures defined in Go to define their data objects, built in Go annotations to provide numeric adjustments in the conversion, and a single function call to automatically create metrics that operate over all non-function types in Go. We evaluate this converter by its ability to be used in a real world search situation. We use Type2Metric to automatically create a metric between Go type signatures themselves, these signatures are parsed out of Go code using Go build tools. This metric is then used to guide Axe to organize fragments of Go code. We evaluate Axe by creating indexes over most of the packages in the Go standard library, and performing random queries on this data. We check for both performance and good answers.

1.4 Contributions

Our contributions are

- **The Metric Definition Problem:** We describe a novel problem of automatically creating metrics for arbitrary data with solutions applicable to search, machine learning, and program synthesis.
- **Automatic Metric Creation:** We describe a solution to the previous problem that uses vanilla type objects to guide in the automatic creation of a metric for product, sum, list, map, channel, and primitive types.
- **Application:** We apply this metric to speed up code search.
- **Evaluation:** We give a discussion of our implementation and show the improvements that can be gained through using a metric to improve complex data search.

2 Example

In this section we present an example of creating a metric from a data type.

2.1 Example

Suppose we want to measure how similar two pieces of code are. Suppose we limit ourselves to the following language, with an obvious interpretation.

$$\begin{aligned} \text{exp} &\rightarrow \text{exp} \quad \text{biop} \quad \text{exp} | \text{singop} \quad \text{exp} | \text{variable} \\ \text{biop} &\rightarrow \text{“}+\text{”} \,|\, \text{“}*\text{”} \\ \text{singop} &\rightarrow \text{“}-\text{”} \,|\, \text{“cos”} \,|\, \text{“sin”} \end{aligned}$$

There are many ways to do that we might do this, we might try to just use a simple edit distance or we might might try to train from a corpus of expressions. Edit distance requires that we treat our expressions as just linear strings of characters. The second option requires us to measure dependent on occurrence which is never exactly what we want to measure by.

One place we can start is with the two sets of terminal objects: biop and singop. Within each class we can use a discrete metric to measure similarity. That is, if the two objects are different, we simply say they are some fixed κ apart. This seems as good as anything, though we might want to say that cos and sin are somewhat more related, so we give them a $\frac{1}{2}\kappa$ distance. This is somewhat arbitrary, and setting constants is an art, but this does provide a good start.

We could do a similar approach with variables, but we don't want to say that $counterVariable$ and $countrVariable$ are further apart than any two other words, and we do not want to give a specific distance for every two words, so instead we can use some edit distance that measures how close the words themselves are.

For the two productions that use "exp" recursively, we can use the above approaches inductively to find distance between two members of the language that at the top are one of these expressions. Here's how: first find the distance between left node of the two members, then the right, then the operator used, and then we can simply use a linear combination of these distances to find our result. In this way we can find the distance between any two members of the language. There is one final question about what we ought to do if the two words are different at the top level. For now we will say that we apply a fixed cost.

This metric was created through a set of intuitions but is not yet something formal. In the next section we will discuss how to formalize these intuitions and others into an algorithm for creating metrics from types.

3 Type2Metric

Definition 2 (Type To Metric Problem)**.** *Let $\mathcal{T}$ be some type theory. The type to metric problem asks to take some $T \in \mathcal{T}$, and create a metric d_T. d_T should satisfy all of the metric properties between any two objects x and y of type T.*

In the following sections we will use this problem definition to show several solutions to the above problem for different theories. After seeing these different theories will see how these can be combined to create a complete solution for a reasonably complex type system – something very similar to Go.

3.1 Levenshtein Combinator

Suppose we have two lists $[1, 2, 1, 1]$ and $[1, 3, 2]$. We want to find the distance between these two lists. This is a somewhat ill defined question, but we can divine a reasonable approach to this question. We can ask what is the minimum cost to change the first list into the second list. Cost, for this example, can be defined as the fewest insertions, substitutions, and removals required to convert between the two lists. There are two things to note here, first different substitutions are more expensive than others, for example $1000 \mapsto 1$ is definitely more expensive than $2 \mapsto 1$. For simplicity, we will say a substitution is as expensive as the difference between the two numbers are. With this in mind we can set a substitution or removal to have some finite cost. Why not 3? We choose this number somewhat flippantly, but there is a bit of an art to it. It's worth noting that this constant sets an upper limit to the substitution cost. This is because any substitution can be achieved with an insertion and a deletion. So, in this particular example the substitution between any two numbers is really the minimum of their difference and 6 (the cost of a deletion followed by an insertion).

The Levenshtein edit distance is a commonly used metric for strings. It gives the number of substitutions, deletions, or additions to edit one string into another string [4]. It is well known that the normal Levenshtein distance on strings satisfies the properties of a metric. There are a couple of assumptions that the Levenshtein metric makes: that all substitutions are of the same cost, that deletions and additions are of the same cost and the same cost as a substitution. While this is a safe assumption for strings of characters, it is not general, as there might be better substitutions than others. For example, it is clear that in a real sense $[1, 2, 3]$ is closer to $[2, 2, 3]$ than it is to $[10, 2, 3]$, as a 1 is closer to a 2 than a 10; however, the Levenshtein edit distance would be the same from the first string to the second string (1). To that end we define the Levenshtein Combinator.

Definition 3 (Levenshtein Combinator)**.** *Assume that $d : (T \times T) \to \mathbb{R}^{\geq 0}$ is a metric. Suppose further that κ is any positive real. Then we say the Levenshtein Combinator of d is $Lev_d : (List \quad T \times List \quad T) \to \mathbb{R}^{\geq 0}$. Such that it is defined on lists of T. We define the combinator as:*

$$Lev_d(a, b) = \begin{cases} \kappa|a[1:]| & |b| = 0 \\ \kappa|b[1:]| & |a| = 0 \\ min \begin{cases} Lev_d(a[1:], b) + \kappa \\ Lev_d(a, b[1:]) + \kappa \\ Lev_d(a[1:], b[1:]) + d(a[0], b[0]) \end{cases} & otherwise. \end{cases}$$

The Levenshtein Combinator runs almost the same as the traditional Levenshtein edit distance. The only differences are the generalizations of the κ from 1, and the addition of the metric between elements of the list. In fact, the original Levenshtein distance can be recovered by substituting 1 for κ, and the characteristic function $1_{a \neq b}$ as the d metric. We will now show that the Levenshtein combinator of a metric is itself a metric.

Identity of indiscernibles and symmetry are pretty obvious, and so are omitted. The triangle inequality also follows in a similar manner as the proof of traditional Levenshtein [5]. We first make a simple optimality argument of the Levenshtein distance, arguing that if

there were a shorter distance between two strings it would have been found. Suppose we are considering a transformation between lists X and Z, that is a series of insertion, deletions, and substitutions. A transformation from X to some Y followed by a transformation from X to some Z is a transformation from X to Z. As the transformation from X to Z is the minimal transformation, it follows $d(X, Z) \leq d(X, Y) + d(Y, Z)$. For more detail on this proof see [4].

3.2 Linear Combinator

We may want to use multiple definitions of distance between some object. A simple example is finding the taxi cab distance between two points, which is the sum of the distances between two points along any finite number of axes. We may also want to weigh different distances different values. This is reasonable if we consider the taxi cab analogy where blocks might be longer north-south than they are east-west. We can apply this idea generally to any product type, where we want to define the metric of that type as a linear combination of the metrics of each of the fields.

We will now briefly show that the linear combination of metrics is a metric. Suppose we have a possible metric $d(a, b) = \sum c_i d_i(a, b)$, where c_i is a positive real number, and d_i is a metric. Symmetry follows clearly. Identity of indiscernibles follows as each term goes to zero, leaving the sum at zero. The triangle inequality follows $d(a, b) = \sum c_i d_i(a, b) \leq \sum c_i d_i(a, c) + c_i d_i(a, b) = d(a, c) + d(b, c)$. Thus, we can safely take a linear combination of metrics and arrive still at a metric.

3.3 Alteration

Sum types appear rather often. As a simple example, consider an option type where a value might be present and have some value or might not be present and thus have no value. How might we define a metric for two objects of this type? Well if both of the objects have values present, then we can simply use the metric for their values (assuming they exist). If neither exist, then the two objects are exactly the same, and so should have zero distance. However, what do we do if one exists and the other does not. One option, and the one taken here, is to simply give them some fixed value. This choice is clean, but does pin us in a bit. Consider what happens if the distance between two present values is greater than twice this constant. We would have broken our metric definition by allowing a shorter route from the first object through an objects of a different type in the alteration back to the second than the given route between the two objects. So we revise earlier, and say that the distance between objects of the same type in the alternation is the min of their types metric result and the change type constant. We formalize this below.

Suppose that we have a tuple (n, v), where n is some natural number less than some N, and a mapping $f : n \to (v \times v \to \mathbb{R})$. Then we can define the alteration combinator on these tuples.

Definition 4 (Alteration Combinator)**.** *Let $T = E \times V$, such that E is some finite set of objects, and V is any other set of objects. Let $f : E \to (V \times V \to \mathbb{R})$ be a relation, such that for all e in E, we have (fe) restricted to $\{(d, v) \in T : d = e\}$ is a metric on that restriction.*

Furthermore, let $\kappa \in \mathbb{R}^+$. *Then we define the alteration combinator* $Alt : f \to T \times T \to \mathbb{R}^{\geq 0}$

$$Alt(f, ((e_1, v_1), (e_2, v_2))) = \begin{cases} \min([fe_1](v_1, v_2), \kappa) & e_1 = e_2 \\ \kappa & otherwise. \end{cases}$$

Symmetry, and identity of indiscernibles is pretty straightforward. The triangle inequality for $Alt(f)$, is a little trickier. We consider four cases. Let A, B, and C all be in E. We can consider the triangle inequality as triple of three of these types, and ask where it will hold. If all three are of the same type (A, A, A), then the inequality is inherited by that of (fA). Consider (A, A, B), then the inequality can be seen $(fA)(A, A) \leq \kappa \leq (fA)(A, A) + \kappa \leq (fA)(A, A) + Alt(f)(A, B)$ Consider (A, B, B), then the inequality can be seen $(fA)(A, B) = \kappa \leq \kappa + (fB)(B, B)$. Consider (A, B, C), then the inequality obviously holds.

3.4 From Types To Metrics

In Figure 1`null` we provide a formalization of rules for generating a metric from a type. They are largely the same as discussed in the previous subsections, but they have two additions which are worht noting. First, we have added maps, a common language type. Largely, these are treated like lists; however to do so we first have to convert them to lists of sorted pairs of keys and values and then we sort them by their key element. At this point we simply plug them into the Levenshtein distance. Second, we have added pointers. These just behave as wrappers of their types. While we are able to prevent some types of pointer cycles by testing if pointers are equal, this will not terminate for all objects such as two linked lists with one element missing. More work needs to be done to resolve this.

There is a cheap proof that needs to be made where we show that all of the distance functions created here are in fact metrics. As a base case, it is clear that numbers are metrics. We have sufficiently shown that all other combinations are metrics in the previous. So, by induction these are all metrics. We have implemented these rules in a tool called type to metric which we will discuss later.

3.5 Example: Go Code Look Up

In order to test our conversion technique we applied the converter to Go type signatures and then created an index over this data. Go type signatures create a recursive type, a fragment of which we provide here for understanding:

$$\begin{aligned} \mathcal{T} = &\text{Primative} \\ &|\text{Slice } \mathcal{T} \\ &|\text{Channel } \mathcal{T} \\ &|\text{Map } \mathcal{T} \quad \mathcal{T} \\ &|\text{Tuple (List} \quad \mathcal{T}) \\ &|\text{Function (List} \quad \mathcal{T})(\text{List} \quad \mathcal{T}) \end{aligned}$$

$$\frac{\Gamma \vdash x, y : \text{List} \quad T}{d(x,y) = \text{Lev}_T(x,y)} \textsc{Lists}$$

$$\frac{\Gamma \vdash x, y : (T_1, T_2, \ldots, T_n) \qquad \Gamma \vdash c_i = \text{const } ((T_1, T_2, \ldots, T_n))}{d(x,y) = \sum_{i=0}^{n} c_i d_{T_i}(x_i, y_i)} \textsc{Products}$$

$$\frac{\Gamma \vdash x, y : T_1 | T_2 | \ldots | T_n \qquad \Gamma \vdash \kappa = \text{const } ((T_1 | T_2 | \ldots | T_n))}{d(x,y) = 1_{\text{type}(x)=\text{type}(y)} \min(\kappa, d_{\text{type}(x)}(x,y)) + 1_{\text{type}(x)\neq\text{type}(y)}\kappa} \textsc{Sums}$$

$$\frac{\Gamma \vdash x, y : \text{Map} \quad T_1 \quad T_2}{d(x,y) = \text{Lev}_T(\text{sortedPairs}(x), \text{sortedPairs}(x))} \textsc{Map}$$

$$\frac{\Gamma \vdash x, y : *(T), x = y}{d(x,y) = 0} \textsc{SameAddressPointer}$$

$$\frac{\Gamma \vdash x, y : *(T), x \neq y}{d(x,y) = d_T(*(x), *(y))} \textsc{DifferentAddressPointer}$$

$$\frac{\Gamma \vdash x, y : T \qquad \Gamma T <: \text{Number}}{d(x,y) = |y - x|} \textsc{Number}$$

Figure 1: Rules for converting types to metrics. The const function simply provides a user provided (or default) cost for various alterations and products. $d(\cdot, \cdot)$ gives us a distance between two objects, and d_T is a metric for some object of type T.

We can then simply apply our converter to this datatructure to create a metric. We can then use this metric, and metric trees (which are discussed) below to create Axe, a prototype go search engine.

3.5.1 Metric Trees

Metric trees are an efficient way to search for object based on some metric. We will give a quicker overview of our implementation of a VP-Tree[1] [5], a particular favor of metric tree that is convenient because of its not dependence on the dimensionality of a vector space. This is useful because it allows for metric spaces with no dimension such as that of type signatures.

The goal of a metric tree is to take some point p, and some radius, and return all the points that are within the radius away from p. The point p and the radius can together be thought of as a target ball, and the goal of a metric tree is then to return all the points in the metric space that reside in that ball. The idea of the VP-Tree is split the metric space recursively into interiors and exteriors. An interior is a metric ball, that is a point in the space with some radius. An exterior is just the complement of some interior. By splitting a space up this way, we can save ourselves a substantial amount of effort by only search the interior if the target ball intersects the interior, and only searching the exteriors if the target ball extends beyond an interiors. A VP-Tree is defined as a recursive data type parameterized over the type of the metric, denoted X here:

$$\begin{aligned}
&\text{VP-Tree } X = \{ \\
&\quad \mathit{interior} : \text{VP-Tree } X, \\
&\quad \mathit{exterior} : \text{VP-Tree } X, \\
&\quad \mathit{self} : X, \\
&\quad \mathit{radius} : \mathbb{R}^{\geq 0} \\
&\quad \}
\end{aligned}$$

In this definition we note a VP-Tree X can be `null`

To aid in understanding it is worth noting the similarity between a VP-Tree and normal binary search tree. In a normal binary search tree, if we were to do a range query, we explore the left child of a node if the node's value is within our range, and same with the right child. As discussed before, in a VP-Tree we search the interior child of a node n if our target ball intersects the ball $(n.self, n.radius)$, and we search the exterior child of a node n if our target ball is at all outside the ball $(n.self, n.radius)$. We formalize this with the below method:

[1] VP-Tree stands for vantage point tree

$$query(v : VP - TreeX, target : X, cutoff : \mathbb{R}^{\geq 0}) \to 2^X =$$

$$\begin{cases} \emptyset & v = \texttt{null} \\ \bigcup \left\{ \begin{array}{ll} query(v.interior) & d_X(v.self, target) \leq v.radius + cutoff \\ query(v.exterior) & d_X(v.self, target) > v.radius - cutoff \\ \{v.self\} & d_X(v.self, target) < cutoff \end{array} \right\} & \text{otherwise.} \end{cases}$$

Now, in order to use this data structure we need to first construct it. To do this we follow a similar to path to that of quick sort. We will choose a random point, and we will find the distance from it to all other points. We will then partition the points by those closer to median and further than the median. Those closer than the median will form the interior and those further than the median will form the exterior of a VP-Tree. We then create a VP-Tree node by setting its self to the random point, interior and exterior as above, and radius as distance to the median. We will formalize this with the below function. As a ease of notation, let $r_{\text{med}}(S)$ simply be an approximate the distance from the random point to the median of S found with arbitrary precision in linear time[2].

$$\begin{aligned} index(\{random\} \cup rest : 2^X) \to VP - TreeX = \\ \text{VPTree}\{&interior : index(\{x \in rest : d_X(x, random) < r_{\text{med}}(S)\}), \\ &exterior : index(\{x \in rest : d_X(x, random) \geq r_{\text{med}}(S), \}), \\ &self : random, \\ &radius : r_{\text{med}}(S)\} \end{aligned}$$

4 Implementation

We implemented the automatic type to metric converter, vantage point tree, and Go type extractor in Go [**?**]. We seek to answer the following two research questions:

RQ1 Can we automatically create metrics from given types?

RQ2 Are these metrics effective in reducing time complexity of problems?

RQ3 Do we get adequate results from our query to axe?

4.1 Experimental Setup

To test both Type2Metric and Axe we use the Go standard library. We parsed most of the functions, type aliases, struct definitions, constants, and methods. Though some modules such as parts of `net`, the network module, dropped to either assembly or foreign calls. These packages were elided because they contained functions for which we could not gather

[2]this can be done with median of medians approach

types. We also elided all testing libraries as they are not used by users of Go. We then used the go compiler to extract various fragments of code, and from the compiler data we extract the types of all of these fragments. We then encoded these types into an Axe specific representation, with the intention of extensibility to other languages. We also gathered any comment block associated with methods, structures, and any of their fields. This took relatively little time (less than ten seconds) to collect. In total we gathered 120 packages, and each of these packages contained on average 94 fragments. For each package we ran a series of random queries with two cutoffs. We recorded the results in **??**. We also ran queries over all the packages with several different cutoffs and we plotted the result in 4.2.2`null`.

4.1.1 Experimental Setup

Our performance experiments all revolved around querying data and checking performance gains by using a VPTree. We queried for objects similar to members of the tree at different ratios. After indexing, we followed the following testing procedure

- For each package we randomly chose 10 fragments in those packages using clock time to provide randomness.
- We then iterated through various cutoffs, for individual packages we used $1, 100, 1000$. For the set of all packages we also tested $0, 2, 3, 5$.
- For a each cutoff we ran each query 10 times over 10 queries and then found the averaged the total time for lookup. We reported this as time for any given benchmark.

For all packages not containing foreign code, this querying and indexing worked.

4.2 Evaluate

4.2.1 Adequate Results

One question we have is does search even work? That is, if we query do we actually get reasonable results? There are not yet great ways to evaluate search appropriateness Especially when evaluating type search, a new feature to Go. That said, we present the following argument: a query gives a good result. So, a query that we ran was for all types matching. F_{64} represents a 64-bit floating point and ?? a hole that matches all types. This query seems like a reasonable query if we consider a Developer who wants to possible raise some number to a power

$$(F_{64}, ??) \to F_{64}$$

We get the following results:

- **math/Max** : $(F_{64}, F_{64}) \to F_{64}$ takes the max of two numbers
- **math/Ldexp** : $(F_{64}, \mathbb{Z} \to F_{64})$ multiples the first argument by two the power of the second

- **math/Dim** : $(F_{64}, F_{64}) \to F_{64}$ takes the max of the difference of the two numbers and zero

- **math/Mod** : $(F_{64}, F_{64}) \to F_{64}$ takes the floating-point remainder of x/y

- **math/Min** : $(F_{64}, F_{64}) \to F_{64}$ takes the min of two numbers

- **math/Nextafter** : $(F_{64}, F_{64}) \to F_{64}$ returns the next floating point after the first argument in the direction of the second.

- **math/Pow** : $(F_{64}, F_{64}) \to F_{64}$ returns the first argument raised to the second

- **math/Remainder** : $(F_{64}, F_{64}) \to F_{64}$ returns the IEEE 754 floating-point remainder of x/y

- **math/Hypot** : $(F_{64}, F_{64}) \to F_{64}$ pythagorean theorem solver

- **math/remainder** : $(F_{64}, F_{64}) \to F_{64}$ internal remainder function

Looking at the results, these seem pretty good. However, in future work we'd like to develop a better way of gaging good results

4.2.2 Performance

Over a linear search, almost all search methods perform worse in indexing. We therefore provide a few quick summarative statistics concerning indexing. It took 15 minutes and 41 seconds to do a cold index of the considered Go standard library. The `log` package, fairly representative in terms of amount fragments contained, took 423 milliseconds to index. It is likely that the steady state of such a search engine in the wild would index the standard library and each individual package. This is because users will often want to query things such has how to convert some data type into another, often a package specific query. So, while certainly longer than indexing by trigrams, it seems that times for indexing are not too high.

We will give an overview of the performance of the querying of types in the below. A key evaluation is whether using a VP-Tree for code search is a true performance gain over a linear method. To test this we compared the time for a linear query (cutoff at max value 1000) under various other cutoffs (1 and 100). We plotted the speed of querying with a 1 for a cutoff over linear speed in Figure 4.2.2`null`, and similarly with a speed of 6 in Figure 4.2.2`null`. In these plots we can see that when querying with a cutoffs of 0, 117 of the 125 benchmarks saw a speed up, and just a little worse for cutoffs of 1 (115 of the 125). Additionally, for a

little over two thirds of the benchmarks we at least halved the time of a linear search. And, those benchmarks that went over did so by often very little. What accounts for the over run? Often it is packages like `math/complex` where many of the packages are very similar.

In figure 4.2.2`null` we show the performance of all the packages run together. We can see serious speed up gains for cutoffs of size less than or equal to 5, after which speed up is negligible at further distances. This likely because large swathes of objects often become within range at further distances such as functions and various constants. This means that the index of the tree becomes less useful as we as all subsections are explored as they are mostly within a ball of cutoff 10 for some query target.

We conclude that indexing can often get often order of magnitude improvements over linear searches, so **RQ1** can be answered: yes they are effective at reducing time complexity of problems.

5 Related Work

5.1 Type Based Search

Using types for search has been an idea for a long time [6, 7], though these early methods often went through the process of doing a full unification and only returning those functions which perfectly unify with the query. Hoogle [8] is a widely used search engine for Haskell functions. For a while it used this pure unification strategy. Since version 5, it has moved to ad-hoc search. It essentially does random edits on the query, each edit with a particular cost, and it then lists types based on the number of edits required to move from type to another type. This was a large improvement in query quality over previous results as it allows users to enter imperfect types. However, because it uses this ad-hoc approach it still requires linear time to the search.

5.2 Google Code searchSearch

Google code search was a popular code search engine that allowed individuals to search for code for reuse [9]. It was designed to be language agnostic, and as opposed to type based search would just use the plain text of code's source. It used an inverted look up table that would search based on various trigrams that appeared in a querier's search [2]. Using this trigram method is was able to achieve a strong index complexity; however, it did not take advantage of structural language features such as type signatures.

5.3 2Vec's

The "2vec" genre is a common genre of paper in the machine learning community [10, 11]. The general idea is to take some object such as string, molecule, or whole graph and convert those objects into a vector. These vectors are then used in various machine learning tasks such as translation and classification. This is done using domain specific knowledge about the type being converted into a vector. Type2Metric provides a somewhat similar feature, but only converts to metrics. These metrics can be used for many tasks but can not yet be

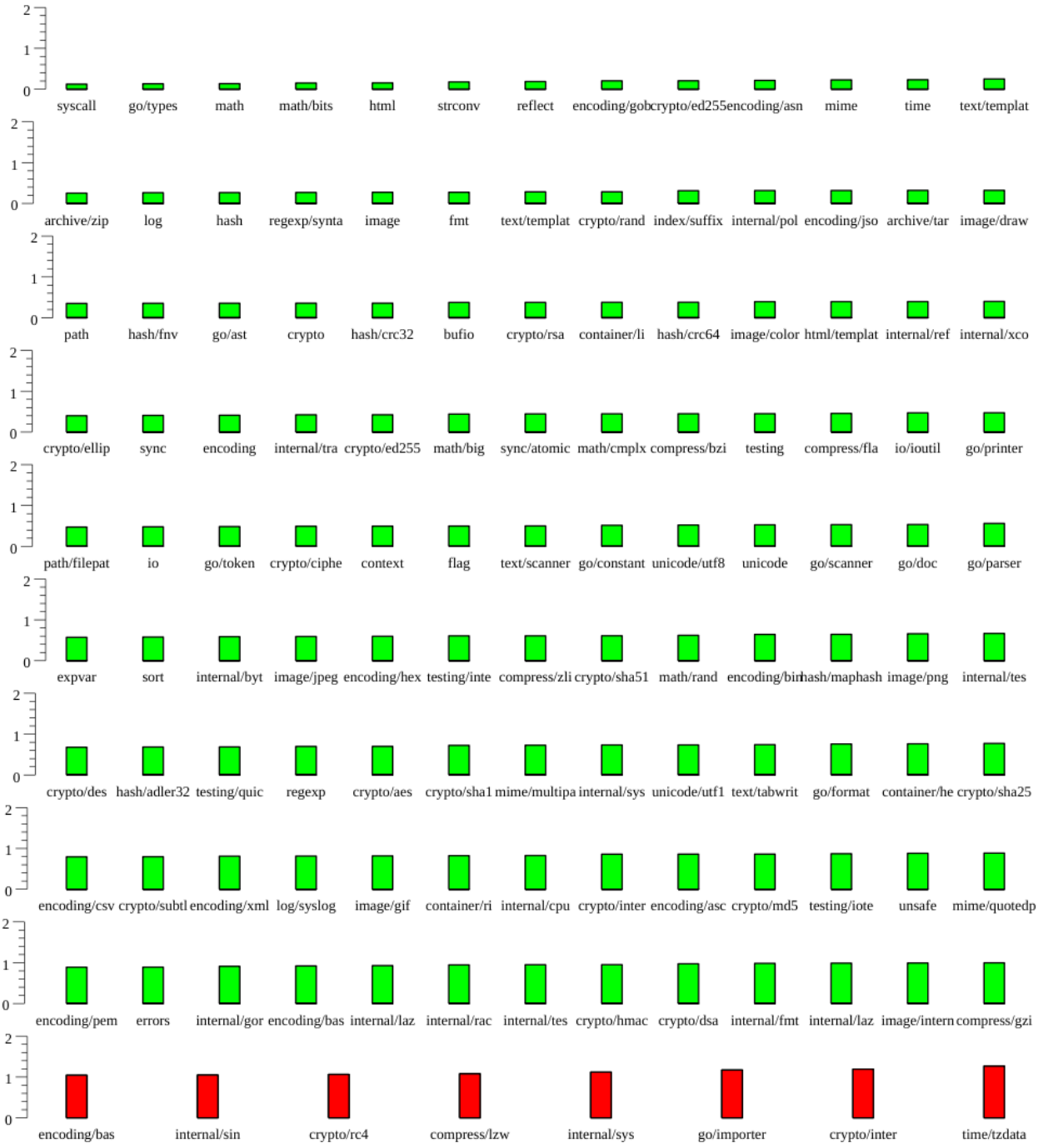

Figure 2: Each Go module was indexed and then randomly queried. Speed up over linear search for a search distance of 0 is plotted for each Go module. Green indicates search was faster, red otherwise. Names are the first twelve characters of module name. The rest are elided for readability.

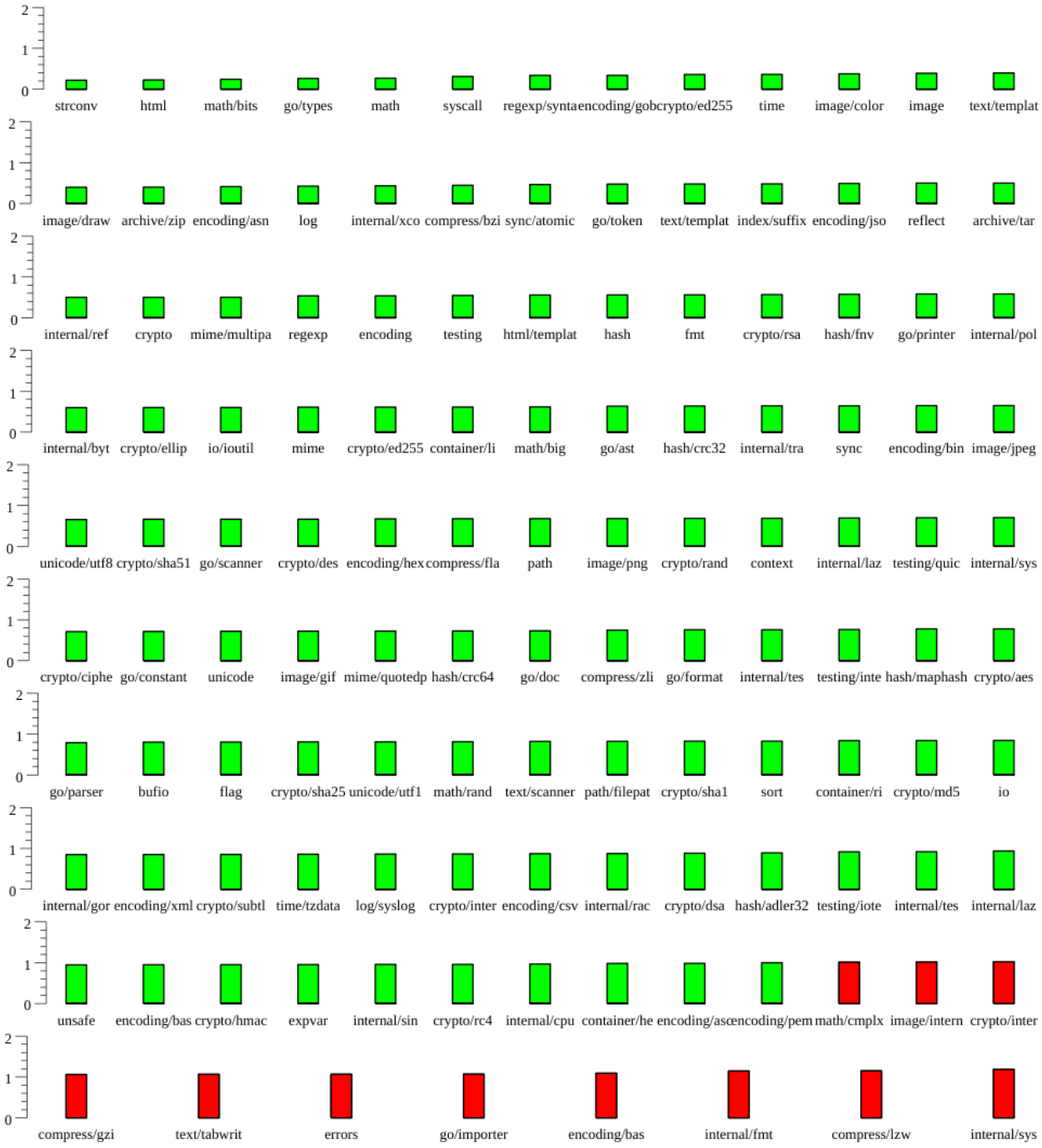

Figure 3: Each Go module was indexed and then randomly queried. Speed up over linear search for a search distance of 1 is plotted for each Go module. Green indicates search was faster, red otherwise. Names are the first twelve characters of module name. The rest are elided for readability.

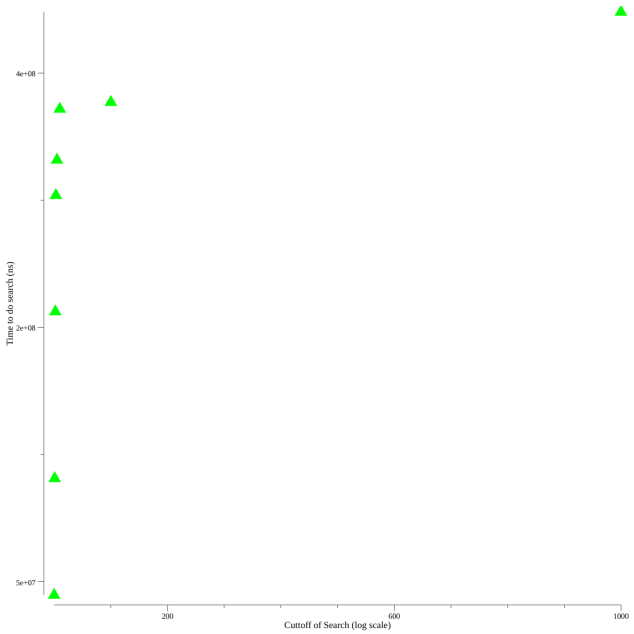

Figure 4: Query times indexing on all data

put into a machine learning model, though a machine learning model could be applied to the constants used by Type2Metric.

6 Future Work: Best Approximation

One natural extension of this work is create a best approximation of types in some euclidean space. The idea would be to find a way to embed the points of the metric space in some arbitrary euclidean space chosen by using the metric of the metric space to try to space out instances of a type. This presents its own problems particularly because the points of the metric spaces we have been dealing with deal with arbitrarily large trees of data. However, if we ware able to approximate this embedding, then we can convert arbitrary data structures into reasonable fixed dimension vectors.

7 Conclusion

We presented the problem of generating metrics based on a type. We then provided a reasonable solution to this problem, the first know to the author. We then implemented this solution in a tool called Type2Metric, which takes a type defined in GoLang and creates a metric from that type. We then made an instance of this conversion, using Type2Metric, that takes Go type signatures and creates a metrizable object. We then used this this metric to create a Go Search tool called Axe, which is the first $O(log(n))$ type based search tool known to the author.

References

[1] Ivan Chen. Vptree.

[2] Russ Cox. Regular expression matching with a trigram index or how google code search worked.

[3] Neil Mitchell. Hoogle: Finding functions from types, May 2011. Presentation from TFP 2011.

[4] Wikipedia contributors. levenshtein distance, 2020. [Online; accessed 19-April-2020].

[5] Jeremy Kun. Metrics on words, May 2014.

[6] Colin Runciman and Ian Toyn. Retrieving re-usable software components by polymorphic type. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 166–173, New York, NY, USA, 1989. Association for Computing Machinery.

[7] Mikael Rittri. Using types as search keys in function libraries. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 174–183, New York, NY, USA, 1989. Association for Computing Machinery.

[8] Neil Mitchell. Hoogle.

[9] Google. Google code search, 2012.

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[11] Sabrina Jaeger, Simone Fulle, and Samo Turk. Mol2vec: unsupervised machine learning approach with chemical intuition. *Journal of chemical information and modeling*, 58(1):27–35, 2018.