A Code Search Engine for Go

Harrison Brewton
Advised by Aws Albarghouthi
April, 2020

"All programming language proofs are by induction"

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 the 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 module *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 popular Go modules. We conclude with future work.

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 proprties hold: symmetry or $d(a, b) = d(b, a)$, identity of indescernables 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, mealody 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 the dream of 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 written. Previous work [2] has looked as using raw source code. Other work [3] has instead used an adhoc 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 logarithimic 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 Golang. 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 all of the packages in the Go standard library, and performing random queries on this data.w

1.4 Contributions

Our contributions are

Definition Problem We describe a novel problem of automatically creating metrics for arbitrary data with solutions applicable to search, machine learning, and program synthesis.

ic Metric Creation We descibe a solution to the previous problem that uses vanila type objects to guide in the automatic creation of a metric for product, sum, list, map, channel, and primative types, as well as version of super-typing.

Application We apply this metric to speed up code search.

Evaluation We give a discussion of our implementation and show the serious 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 interpreation.

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

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 chracters. The second option requires us to measure dependent on occurance 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 arbitrarym but seens fine.

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 fidn our result. In this way we can find the distance between any two members of the language. There is one final question abot 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 ast this often the best approach, but what we will discuss other options in future sections.

This metric was create through a set of intutions but is not yet something formal. In the next section we will discuss how to formalize these intutions 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 in 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. Specifically, we will look at lists, structs, maps, primatives and we will discuss some extensions for this soultion.

3.1 Levenshtein Combinator

Example 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 thant* $2 \mapsto 1$*. For simplicity, we will say a substiution is as expenensive 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 there 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 satifies 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 genereral, 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 recovred 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].

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 $d(X,Z) \leq d(X,Y) + d(Y,Z)$.

3.2 Linear Combinator

We will now briefly show that the linear combination of metrics is a metric. Suppose we have a 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 indescernables follows as each term goes to zero, leaving the sum at zero. The triangle ineqaulity follows $d(a,b) = \sum c_i d_i(a,b) \leq \sum c_i d_i(a,c) + c_i d_i(a,b) = d(a,c) + d(b,c)$.

3.3 Alteration

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 $Alt : f \to T \times T \to \mathbb{R}^{\geq 0}$*

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

Symmetry, and identity of indiscernables 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

$$\frac{x : \text{List}\quad T \qquad y : \text{List}\quad T}{\text{Lev}_T(x,y)}\ \textsc{Lists} \qquad \frac{x : (T_1,T_2,\ldots,T_n) \qquad y : (T_1,T_2,\ldots,T_n)}{\sum_{i=0}^{n} c_i d_{T_i}(x_i,y_i)}\ \textsc{Products}$$

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

3.5 Some Additions

3.5.1 Recursive Type Promotion

4 Example: Go Code Look Up

4.1 Generics

5 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 convient 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 ball, and the goal of a metric tree is 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 substatinal 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 reursive data type parameterized over the type of the metric, denoted X here:

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

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 $r_{\text{med}}(S)$ simpliy be an approximate the distance from the random point to the median of S found with arbitrary percision in linear time[2].

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

6 Implementation

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

RQ1 Can we automatically create metrics from given types?

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

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

7 Related Work

8 Future Work

8.1 Best Approximation

9 Conclusion

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.