Efficient Sets

by Sven Nilsen, 2017

In this paper I represent a new concept, called “efficient sets”, which is inspired by recent advancements in computer science such as meta-tracing Just-In-Time (JIT) compilers and Sets Of Real Numbers (SORN). Efficient sets allows composing approximations of the universal existential path, a computational oracle, to make qualitative leaps in the fields of supercomputing and artificial intelligence, where SORN is a special case for projected real numbers.

Background

In mathematics there exists finite and infinite sets, sets of low and high complexity, and even sets with infinite complexity such as those studied in fractal geometry. The question is: Why are some sets easier to use in practical computing than others?

Traditional mathematical set theory does not make any distinctions between classes of set-membership. In complexity theory, this distinction is vital to our understanding of algorithms and problem solving. This bizarre conceptual gap leads to problems when analyzing sets in terms of complexity, or complexity in terms of sets.

A staggering demonstration of this missing fundamental understanding, is the famous `P = NP` problem which is considered one of the major open problems in computer science. These two complexity classes are currently unknown to be the same or separate.

This uncertainty is due to the fundamental nature of sets: Either a problem belongs to the set of problems defined by `P`, or it does not belong to that set. `P = NP` if and only if both sets have the same members. Therefore, `P = NP` seems like it should be a statement that is either true or false.

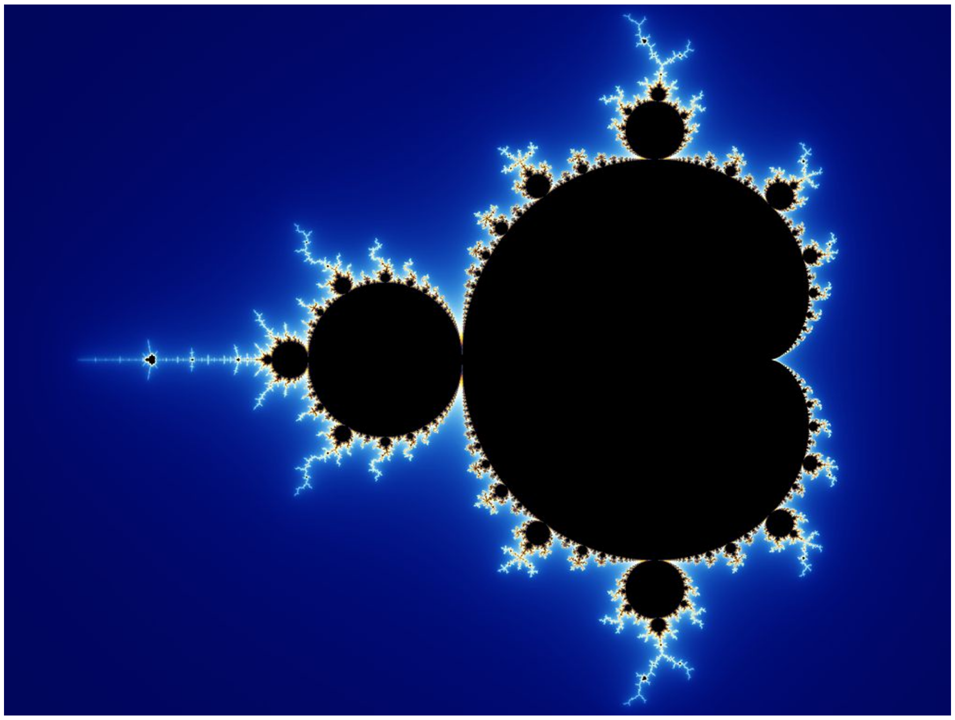
The set `P` defines problems that are “quickly solvable”, while `NP` defines problems that are “easy to verify”. When running an algorithm for all inputs, one gets the answer whether the algorithm is “quickly solvable” or “easy to verify” by how long time this computation process takes. Since one can pick an arbitrary algorithm, the computation process can take arbitrary long time. As long the algorithm that is picked solves the same problem, the algorithm that takes the shortest amount of time at either task is used to classify the problem. So, the question whether some problem belongs to `P` is defined whether there exists an algorithm over all possible algorithms which this specific characteristic of spending less than polynomial time relative to the size of input type.

The set `P` is defined by how hard something is to compute, which is a much more complex concept than merely defining whether some object belongs to a collection defining a set, for example, whether `2` is in the list `[1, 2, 3]`. Clearly, there are different classes of set-membership by how hard or easy it is to compute. `P = NP` is a meta-question that asks whether some minimum hardness predicts actual set-membership. That is, whether two classes of problems are equal or not, simply by looking at the time it takes to solve and verify the problems.

Motivation

With all the conceptual problems of `P = NP`, it is desirable to understand why sometimes things are easy to tell whether they belong to a set, and why sometimes it is hard. This issue must probably be solved before any meta-question about this nature can be answered.

For example, in the Mandelbrot set pictured below, the black points belong to the set and the other colors are used to tell how fast it takes to determine non-membership.



https://en.wikipedia.org/wiki/Mandelbrot_set#/media/File:Mandel_zoom_00_mandelbrot_set.jpg

The point at `(0, 0)` in the Mandelbrot set is very easy to tell that it belongs to the set. If one takes an arbitrary point with infinite precision, it could happen that it requires infinite time, since the Mandelbrot set has infinite complexity.

Using a combination of fast programs, it is possible to construct efficient sets that are easy to compute, but it is impossible to construct any set that is equal to the Mandelbrot set. The Mandelbrot set contains points that take infinite time. The efficient sets one can construct by combining algorithms that are easy to compute, have no points that take infinite time. Therefore, these two classes of sets can not give the same results. Minimum hardness predicts actual set-membership, the same motivation as in `P = NP`.

Now, assume that a magical computer program existed that determined instantly whether any point belongs to the Mandelbrot set. This program, and together with other conventional programs, could be used to construct more complex sets that would be efficient.

Traditional mathematics ignores minimum hardness, because it does not know which functions are efficient, and is therefore too weak to reason about some actual set-memberships in the real world. It makes no general assumptions about which sets are efficient and which are not.

Definition

An efficient set in path semantics is the Boolean algebra of sub-types defined by functions such that:

$$∃^+f\{g_i(x^i)\} \iff g_n(f(x_i))$$

Where `f` is some function for computation and `$g_{in}$` defines a dependent product from a family of set-membership functions.

There following law of computation on sets works by destructing and restructuring a canonized form of Boolean algebra expression:

$$∃^+f\{\sum i \{ \prod n \{ g_{in} \} \}, \sum j \{ \prod m \{ g_{jm} \} \}, \ldots\} \iff \sum i, j \{ \prod n, m \{ ∃^+f\{g_{in}, g_{jm}, \ldots\} \} \}$$

The law of computation on sets defines which sets are efficiently computable in which context of available algorithms and knowledge, and requires the actual construction of these functions and the knowledge about them to be actually efficient. Sets that are not describable in this form are considered inefficient.

The rest of this paper is about explaining what the definition of an efficient set means. The law of computation on sets is not treated here because it is a very heavy and complex topic in itself, which requires multiple papers going into details of a larger toolbox.

Constructing Sets in Boolean Algebra of Sub-Types

An arbitrary set can be constructed using the equality operator alone.

For example, to construct the set of numbers 1-4, one can write down 4 equations:

$$(x = 1) \lor (x = 2) \lor (x = 3) \lor (x = 4)$$

In path semantics one can write this as an expression of a sub-type defined by functions:

$$x : (= 1) \lor (= 2) \lor (= 3) \lor (= 4)$$

The logical disjunction operator `∨` (OR) is less common in simple proofs of path semantics than the logical conjunction operator `∧` (AND), because one can just split it up into cases and solve them separately.

The important thing here is that a single function `(= k)` is all that is needed to describe a set, but it requires listing all the members.

Infinite Sets

In the previous example, I showed that one can construct arbitrary sets with the `(= k)` function:

x : (= 1) ∨ (= 2) ∨ (= 3) ∨ (= 4)

For infinite sets, this is not possible to do using a computer with finite memory. It does not matter what algorithms one uses, because the set is inefficient simply by not be able to exist in the first place. In technical language one says that infinite sets are inefficient with respect to a family of functions consisting of only one member, `(= k)`, parameterized by `k`.

Computing With Infinite Sets Using Finite Sets of Symbols

There might be sets that are finite, but not possible to compute with for some specific operations. These sets are inefficient with respect to these operations, but not for all algorithms since they fit in memory.

Here are two thumb rules for remembering when a set is considered to be efficient:

1. The set must fit inside a finite description of symbols
2. Given a set of input values for a function, the set of output values must be computable, but only up to the accuracy available by the available symbols to describe the set

For example, all natural numbers larger than 2 is an infinite set:

x : (> 2)

Even the set is infinite, it requires only a finite set of symbols to describe it.

Now, if one has two sets:

x : (> a)
y : (> b)

What is the set of `x + y`? It is not hard to notice that:

x + y : (> a + b)

In path semantics, one can write the following:

∃add{(> a), (> b)} <=> (> a + b)

The `∃` symbol is called “existential path” and tells whether the output of that function returns something. The set of outputs is the same as the values for which the existential path returns `true`.

For some functions, the existential path can get very complex, for example for natural numbers:

$$\exists \mathrm{mul}\{(> 1), (> 1)\} \iff \mathrm{composite}$$

$$\mathrm{composite} \iff (\neg= 0) \land (\neg= 1) \land \neg\mathrm{prime}$$

The function for determining the set of composite number is just as complex as the function for determining the set of prime numbers, so this set is less efficient than by making an approximation:

$$\exists^{+}\mathrm{mul}\{(> 1), (> 1)\} \iff (> 2)$$

This means one can trade accuracy with performance, but if one uses the right tricks and algorithms, the accuracy can be recovered, and therefore the set is still efficient in that sense. However, since one can have many kind of inputs, it is not efficient for `(> k)`, but only for `(> 1)` and `(> 2)` specifically.

To make it efficient for all parameters, one would need a slightly worse approximation:

$$\exists^{+}\mathrm{mul}\{(> a), (> b)\} \iff (> a \cdot b)$$
$$\exists^{+}\mathrm{mul}\{(> 1), (> 1)\} \iff (> 1)$$

This is because in the definition of efficient sets, the same function `f` appears on both sides:

$$\exists^{+}f\{g_i(x^i)\} \iff g_n(f(x_i))$$

The `∃⁺` symbols means it is a heuristic of the existential path, which might return `true` for some values that are not actually returned by the function. Heuristic accuracy is determined by which family of functions that are used to describe the set, because some Boolean algebra expression of these functions yield the best description of the set.

Perhaps some might think this is to restrictive. However, there are two reasons for this.

The first reason is a very nice property that makes computing on sets a superset of normal computation:

$$\exists f\{(= x_0), (= x_1), \ldots\} \iff (= f(x_0, x_1, \ldots))$$
$$\exists \mathrm{mul}\{(= a), (= b)\} \iff (= \mathrm{mul}(a, b))$$

In the case of the equality function, computing with sets is just normal computation!

The second reason is that for other functions, one can reuse the circuitry for normal computation while re-interpreting the output as a higher order parameter for a partial set.

$$\exists^{+}\mathrm{mul}\{g_0(a), g_1(b)\} \iff g_2(\mathrm{mul}(a, b))$$

This is particularly useful because it puts an upper boundary on the complexity of computation, while pushing extra complexity to the Boolean algebra of sub-types. If one wants to increase accuracy beyond what is possible for a single member of the function family, then one needs to build increasingly complex expressions, such as `(> 2) ∧ (< 7) ∨ (> 10)` and then break it up using the law of computation on sets. The definition of efficient sets gives a precise way to reason about trade-offs in complexity.

If the function family for `mul` contains equality `(= k)`, then one can split the computation into every pair of natural numbers larger than 1, and collect the outputs to create the set of composite numbers.

```
2 · 2 = 4        adding `4` to the set
2 · 3 = 6        adding `6` to the set
3 · 2 = 6        already got 6
2 · 4 = 8        adding `8` to the set
4 · 2 = 8        already got 8
…
```

However, this is inefficient because there are infinite composite numbers!

It is much cheaper to say that composite numbers are a subset of numbers greater than `1`:

$$\text{composite} \subset (> 1)$$

This is an efficient set when using addition and multiplication, but it is not an accurate set.

In many cases one is not interested in whether the output has a complex property, e.g. whether a number is composite or not, so the heuristic works just fine. The whole point is to get high quality answers about desirable properties that are often inaccurate, not to describe every complex property of the answer in full detail.

Tracing Normal Computation

Any function in the function family determining set-membership must depend only on the value from the normal computation, such that it can not make any better predictions than the information content by tracing normal computation.

For example:

```
x := a · b
if x > 3 {
    x + 4
} else {
    x - 4
}
```

Assume `a = 1` and `b = 2`. The trace would be the following:

```
x := a · b      // 2
x – 4           // -2
```

When computing on sets, the input is re-interpreted as `$g_0(1)$` and `$g_1(2)$`.

```
x := a · b      // ∃⁺mul{g₀(1), g₁(2)} <=> g₂(2)
x – 4           // ∃⁺sub{g₂(2), (= 4)} <=> g₃(-2)
```

Notice that the heuristic is determined by values of normal computation, but not vice versa.

In the application of meta-tracing JIT compilers, one can write an interpreter in some programming language with some extra hints and then generate a program that contains an automatic JIT optimizer for the interpreted language.

Since computation on efficient sets only depends on the trace, it might be possible generate JIT compilers for computation on sets!

Side by side with the normal computation circuitry, some lookup tables can be generated from the function family. If some lookup depends on the intermediate computed value, the value is passed from normal computation to function computation by adding extra runtime instructions. This is similar to guards in normal tracing JITs, except that computation can continue afterwards.

Therefore, the complexity of computing on sets can be added to the meta-tracing compiler, which can include instructions that supports set computing in the normal trace, and generate interfaces that lets programmers invoke a function call a computation on sets instead of values.

This might make it possible to construct a whole programming language that supports computation on sets derived from the source, without the need to design specific formats for every type. However, this might require extensive research.