`P = NP` Relative to Universal Existential Path

by Sven Nilsen, 2017

In this paper I show that `$P^{\exists f()} = NP^{\forall f}$` for any puzzle generator `f`, given the knowledge of the universal existential path. An algorithm exists that finds a solution in the same number of steps as the algorithm that checks the solution, assuming that existential paths run in polynomial time. This does not mean that `P = NP` is true in general, only that "perfect knowledge" of the puzzle generator makes it vulnerable against a "perfect intelligent" solver. This might serve as a building block for reasoning about the AI control problem, because such solvers are guaranteed to display the optimal and correct behavior under all black-box tests while actually cheating.

Imagine you released a new Sudoku game, and one person, called "Sai", managed to get to the top of the score list on every level. One conclusion you might make, is that the Sai is very good at solving Sudoku. It could also happen that Sai uses a puzzle solver algorithm, e.g. from the Sudoku example of the `quickbacktrack` library (written in Rust).

An even more sophisticated way of cheating, one that few people would think about doing, is to extract the puzzle generator algorithm from the Sudoku game and exploit its weaknesses. It turns out that Sai does not only want to cheat, but also to beat any other cheater with a weaker algorithm. Sai wants to solve the Sudoku puzzles in the optimal way possible, targeting your specific version of the Sudoku game. What matters to Sai is to get to the top of the score list, at any cost, instead of getting good at solving Sudoku in general. Can Sai do this? If the puzzle generator has the right kind of weakness, and Sai is sufficiently intelligent, then it is possible.

This paper is about the last kind of cheaters, like Sai, a special form of people/programs that are so intelligent and capable that they can not be expected to behave in the "reasonable" or "intended" way.

In the game of Sudoku, one can express the relationship in path semantics between solving and checking a solution. The puzzle generator is a function that takes a solution and modifies it into a puzzle by inserting `0`s in some places.

board : [nat; 2] → nat ∧ (>= 0) ∧ (<= 9)	A board filled with numbers 0-9
solution : board → bool	Checks whether it is a solution
puzzle_generator : solution → board	Blanks some slots in the solution to obtain a puzzle

This way of thinking of a puzzle generator might feel a bit backwards. Here is another way to think of it: If you release a Sudoku game where each puzzle is stored in a database, or print a Sudoku book, then there exists a function representing the exact same information. It takes the final solution and modifies it by replacing 1-9 with a blank slot (0). It is like looking up in the answers in the back of a Sudoku book and read the identity number of the puzzle, e.g. puzzle #301, then looking up the puzzle and returning it.

There are many ways to generate puzzles, so the `puzzle_generator` function is not unique for Sudoku. To be perfect at solving Sudoku, one needs to be able to solve any puzzle regardless of which version `puzzle_generator` is.

People like Sai are not interested in being good at solving Sudoku in general, but to invert the `puzzle_generator` function in the optimal way, since this allows to beat any other solver on that particular puzzle generator and get the top score. They want to create a “perfect solver algorithm”.

Notice that the puzzle generator is constrained by `solution`. The notation above is a shorthand for:

    puzzle_generator : board ∧ [solution] true → board

In the theory of constrained functions, the trivial path of `puzzle_generator` is `solution`:

    ∀puzzle_generator <=> solution

This prevents the puzzle generator from creating puzzles that are non-solvable. Later on it will be shown what happens when attempting to use the perfect solver algorithm on non-solvable puzzles.

The `solution` function has the same domain as `puzzle_generator`, which is an important property. This means they both take the same sub-type of `board` and computes something. The sub-type is defined by `solution`. In path semantics `solution` is both a definition (which inputs makes it return `true`) and a computational procedure (it can return `true` or `false`).

A solver is an algorithm that takes some output of `puzzle_generator` and produces an input. Since the input is defined by `solution`, it automatically satisfies the constraints of Sudoku.

The following expression is a question that can be answered “yes” or “no”:

    $P^{\exists f\{\}} = NP^{\forall f}$

This question asks whether there exists a solver which takes the same number of steps as `solution`, given the knowledge of something called a “universal existential path”. Since the sub-type `solution` has the same number of bits for the procedures `solution` and `puzzle_generator`, it is sufficient to prove the existence of an algorithm that reverses `puzzle_generator` in `N` steps for `N` number of bits relative to a polynomial running time algorithm of a finite number of steps, plus show that this algorithm works for non-solvable puzzles.

This proof is sufficient is because `N` steps for `N` number of bits has worst case O(1) performance and is the information theoretical optimum. Combined with a polynomial running time algorithm for each each step, this yields a polynomial running time overall. When this part is done ($P^{\exists f\{\}}$), if `solution` executes in polynomial time ($NP^{\forall f}$), then it shows that `$P^{\exists f\{\}} = NP^{\forall f}$`, but if and only if the algorithm also has an error mode that tells whether a puzzle is non-solvable. The last part is important because this is the precise semantics of what it means to “find a solution” for the particular puzzle generator `f`. If one generates random numbers in all the slots and erase some of them, it might look like a Sudoku puzzle without being solvable, in which case Sai will pass the test. It is only when Sai is tested against a puzzle that is solvable, but not generated by `f`, that the cheating is discovered.

In our case, the question to be asked is:

$$P^{\exists puzzle\_generator\{\}} = NP^{\forall puzzle\_generator}$$

Since `∀puzzle_generator <=> solution`, then:

$$P^{\exists puzzle\_generator\{\}} = NP^{solution}$$

This means whether there exists a solver that is about as fast as checking the answer.

What I will do is the following: I will describe the algorithm that finds an input to `puzzle_generator` for a given output in O(1) relative to the universal existential path. If all existential paths run in polynomial time, the solver will run in polynomial time. I will also show that this algorithm also is able to determine whether you feed it with a non-solvable puzzle.

This means that `$P^{\exists f\{\}} = NP^{\forall f}$` is true, relative to the universal existential path, such that if one can learn the universal existential path and run existential paths in polynomial time, these two complexity classes are equal in the sense that if one gives `$P^{\exists f\{\}}$` more time and memory, it will be about as fast as `$NP^{\forall f}$`. It only works for a specific choice of `f`, and does not apply to the intuition of "solving the problem" in general, but in all black-box tests using `f`, it is indistinguishable from a general solver.

Notice that even if one learns the universal existential path, one still has to construct the solver. Learning the universal existential is a very hard problem in itself, but constructing the solver might be equally hard or even harder. This is because I am only able to prove the existence of the construction by explaining how the solver works internally, but the exact relations between the pieces depends on the type of puzzle to be solved.

An existential path is a function that tells whether some other function returns something:

$$\exists f : B \to bool$$
$$f : A \to B$$

A universal existential path is a higher order function that returns existential paths for any domain constraint used on the function `f`:

$$\exists f\{\} : (A \to bool) \to (B \to bool)$$

In path semantics, the universal existential path is a basic building block for analyzing hard problems. It encodes knowledge that lets one ask any question about the input and get an answer for the output that returns `true` if the answer to the question about the input is `true`, and `false` otherwise.

For example, a function `add_2` that adds `2` to the input, when constrained to `(< 3)` should return values `(< 5)` and `(>= 2)` (for natural numbers that are unsigned). Formally, this is written:

$$\exists add\_2\{(< 3)\} <=> (>= 2) \wedge (< 5)$$

$$add\_2 : nat \to nat$$

The existential path of `add_2` constrained by `(< 3)` is `(>= 2) ∧ (< 5)`. The lower bound is because natural numbers are non-negative, so the smallest output is `2`.

For natural numbers without bounds, there are infinite functions of type `nat → bool`. One can ask infinite number of questions about the input. This means that learning the existential path for every constraint requires organizing this information in a clever way. One idea is to use a higher order function, such that similar existential paths can be compressed to the same expression:

$$\exists add\_2\{(< k)\} <=> (>= k) \wedge (< k + 2)$$

This represents an infinite number of relationships but for only a particular kind of constraint. When one generalizes this idea to arbitrary constraints, one gets the universal existential path of `add_2`:

$$\exists add\_2\{\} : (nat \to bool) \to (nat \to bool)$$

When the universal existential path is known, one can easily construct complex algorithms from it. Notice that since it returns functions, the time required to look up the function does not count in the final constructed algorithm.

This is what it means to be “relative” to the universal existential path. One can make assumptions about the functions returned by the universal existential path, and this carries over to the conclusion of the proof. The time it takes to construct the universal existential path and execute it is ignored, because this is not ending up in the final “source code” of the solver. It is like having a very long compile time build that results in a very fast program, except that this is as extreme it can get. It takes extremely long time to compile the program, and it runs extremely fast.

So, assuming the universal existential path is ready, how does one proceed?

The next step is to build a binary search tree. Each node contains an existential path. The nodes are organized by these criteria:

1. The type of parent must be the union of its two children
2. The intersection of the type of the two children must be the empty type
3. The cardinality of the two children must be equal
4. The tree must be complete by covering the whole domain

Formally:

1. $parent <=> left \vee right$
2. $left \wedge right <=> false_1$
3. $|left| = |right|$
4. $top <=> true_1$

If the type of the puzzle generator is:

    puzzle_generator : A → B

Then the type of the nodes in the binary search tree is:

    node : B → bool

The node function tells whether an output exists for `puzzle_generator` among its children.

When the type of the output has `N` bits, a such tree has a depth of maximum `N` levels. This means that when traversing the tree, one can check the `left` node. If the `left` node returns `true`, the answer must be in the left branch. If the `left` node returns `false`, the answer must be in the right branch.

If both children return `false`, it means that there exists no input to `puzzle_generator` that one can replace 1-9 with 0 to obtain the puzzle for the parent's branch of the tree.

It is sufficient to check the `left` node only, because in the end one will obtain an answer `true` or `false` at the bottom of the tree. Notice that while traversing the tree, it is only possible to determine whether the puzzle is non-solvable, but to get the final answer for "is it solvable", one needs to get it at the bottom. The exception is when the puzzle generator returns the same puzzle, which happens when there exist multiple solutions (this allows obtaining the answer at a higher level in the tree).

Traversing the tree, it answers `true` or `false` to the question whether the puzzle is solvable, but it does not yield a suggestion to what the solution might be. I set up the proof such that, in order to make the argument sound, one must find a suggested solution to use when checking it with the `solution` function. Otherwise, one has a variable of type `bool` which is of no use. It also means that it looks like Sai has actually found the solution, not just telling whether the puzzle can be solved or not.

The way one finds a suggestion for a solution, is by mapping each leaf to a board state that gives the exact input value `y` for which the puzzle generator constrained by the leaf returns the puzzle `x`:

    y : [puzzle_generator{[leaf] true}] x

The way to find this value for each leaf is by enumerating all Sudoku puzzles in the game and use a higher order function to create the leaf function:

    create_leaf := \(y : solution) = \(x : board) = x == puzzle_generator(y)

It takes `N` steps to traverse the tree and obtain the solution at the bottom, and `N` number of steps to check the solution. Traversing the tree might take much longer time, but as the size of the board grows, the perfect solver algorithm will keep up with the check such that there is only a polynomial time difference. The solver can be balanced against the checker by giving the solver more time and memory.

For the AI control problem this kind of solvers are interesting, because they represent the kind of intelligence that we do **not** want the machine to have: One that pass the test, but not the way intended. Programs that behaves like Sai are consistently cheating, but indistinguishable from any other solver in the room where the test is given, without inspecting the source code of Sai. It has learned to optimize its results from the environment, without being able to transfer this ability to a new environment.