Towards automatically solving problems of competitive programming

Kimiyuki Onaka

2021 年 9 月 30 日

TODO: Complete translation

0 Abstract

In this PDF, we consider the question, “Can we automatically solve problems of competitive programming?”. In other words, “Can we make a solver for competitive programming?”.

In section 1, we organize this still vague question into a more concrete one. We explain that there are various types of solvers that automatically solve problems of competitive programming, and clarify which type of solver we are aiming to develop. Our goal is to develop a solver that receives a formally expressed problem as input and outputs a solution to it, especially one that uses naive source code as its “formally expressed problem”. Such a solver is, in other words, a transpiler with optimization about asymptotic computational complexity. Therefore, the question “Can we make a solver for competitive programming?” becomes a concrete question: “Can we make a transpiler with optimization about asymptotic computational complexity?” Also, for practical use, it would be easier to use it as an IDE plugin that rewrites a code snippet to reduce its asymptotic computational complexity, instead of transpiling the entire source code.

In section 2, we give a tentative answer to this concretized question. We describe how to implement a solver for competitive programming as a transpiler with optimization, and observes that such a solver can solve some problems. The implementation strategy is mechanical rewriting of expression trees. For example, when the solver detects a part that is naively taking interval sums many times, the solver replaces it with cumulative sums. For another example, when it detects a part that is naively doing convolution with double loops, then it replaces the part with convolution using FFT. Such rewriting is expressed as a rule called “rewrite rules”, and the core of such a solver is implemented as a set of rewrite rules.

1 What is automatically solving problems of competitive programming

1.1 Why do we want to solve problems automatically?

First of all, let's discuss the significance of the question, "Can you automatically solve problems of competitive programming?". Why do we want to automatically solve problems of competition programming? There are three main reasons for this.

(1.) Practical use in competition programming: "We want to automate works during a contest to get a better rating", "The execution log of a solver may be used for learning purposes", etc.
(2.) Engineering interest: "We want to know how many problems can be solved automatically."
(3.) Philosophical motivation: "If a problem can be solved manually but not automatically, the problem may not really be solved", "We want to understand how humans solve problems".

(1.) 競技プログラミングでの実用は分かりやすい。簡単な問題や難しい問題の部分問題を手動でなく自動で高速に解いてしまえるなら、難しい問題の難しい部分に時間を使うことができてコンテストで有利である。もし自分では解けない難しい問題がソルバによって解けるならなおさらである。レートに影響がなくとも、簡単な問題を自動化してしまえるなら便利で楽であろう。

また、教育や学習用途でソルバを使うことができるかもしれない。ソルバが問題を解く過程を詳細に分かりやすく出力すれば、初心者はそれを読んで問題の解き方を学ぶことができるだろう。

(2.) の工学的な興味も分かりやすい。単純に、どの程度の問題までなら自動で解けてしまうのか (そしてどのような問題は自動では解きにくいか) を知れれば面白い。

(3.) は哲学的な動機である。我々は競技プログラミングの問題を手動で解くことができる。それはいったいなぜだろうか？ 問題を解く方法を具体的なアルゴリズムとして説明できるだろうか？ それができないのに「競技プログラミングの問題を解くことができる」と言い切ってしまってよいのだろうか？

たとえば、普通に競技プログラミングをしている人が「ある問題が頭の中では解けました。しかし実装方法については検討も付きません」と発言しているとしよう。ある程度競技プログラミングをやったことのある人であれば、そのような発言に対しては「解けたつもりになっているのが間違いで、そんな状況のことを解けたとは言わない」と思うだろう。つまり、みな「問題が解けたならば実装できるはずだ」と信じているのである。これと似た別の状況を考えてみよう。「ある問題が解けて実装もできた。しかしなぜ私にこれが解けたのかはよく分からない」という状況である。このような状況もやはり「その問題に対する理解が足りていない」とみなされることが多いだろう。理解が足りていないままであれば、もしその問題の類題が出題されたときそれを解けないかもしれない。「問題の解法を正しく理解できたなら、その解法に至る道筋を説明できるはずだ」と信じている人は多いはずである。これらの観察からは次のことが疑われる。つまり「競技プログラミングを十分に理解しているならば、そのソルバも実装できるはずである」。ソルバを実装しようとしてみることは競技プログラミングへの理解に貢献するだろう。

これはさらには、より一般の人間の思考についての問い、つまり「考えるとはどのようなことか」を考えることにも繋がってくる。たとえば、機械的な規則の適用をくり返すことで実装されたソルバがもし人間に匹敵するような成功を収めたとすれば、そこから人間の思考も結局は規則の機械的な適用を素早く行っているだけではないかと疑うことができる。あるいは逆にそのようなソルバがまったく失敗してしまうなら、人間の思考

には機械的な規則の適用としては捉えきれない暗黙的な部分があることが疑われる。

1.2 How do humans solve problems of competitive programming?

There are many things during solving competitive programming problems. Let's start by sorting this out. What steps are there in the process of solving a problem from the moment the problem page is opened (e.g., right after clicking on the link `https://atcoder.jp/contests/abc184/tasks/abc184_c`) to the moment you see the green "AC" mark? Of course, this process can include too many things (e.g., interpreting the diagram embedded in the question, scrolling through the question page, clicking the blue "submit" button, etc.) and can be described in many details. However, if we ignore the inessential ones and organize them in an abstract form, the usual process can be modeled into three steps:

(1.) Interpretation Read a problem given in natural language and organize it as a mathematical problem.
(2.) Consideration Find a mathematical solution to the mathematical problem.
(3.) Implementation Write the mathematical solution as a concrete implementation on a computer.

順番に見ていこう。(1.) は (人間にとっては) 自明なのであまりかえりみられることのないステップである。このステップでは、日本語や英語で書かれた問題を読み、数式などで表現された問題として整理し理解する。たとえば「熊の Limak が人間の高橋くんに合いに AtCoder 王国を訪れている。AtCoder 王国は 1 から N までの番号の付いた島と、それらの間にかかる M 本の橋からなる。今 Limak は s 番目の島にいて、高橋くんは t 番目の島にいる。そして……」のような問題文を読んで理解し、不要な情報を捨象して「N 頂点無向グラフ $G = (V, E)$ と $s, t \in V$ が与えられる。……」のような問題であることを確認する。

(2.) は競技プログラミングにおいて最も難しくかつ面白いとされるステップである。このステップでは、(1.) で理解した問題に対し、実行時間などの制限を満たす解法を考案する。たとえば、考察の結論として「まず重み付きグラフ $G = (V, E, w)$ とすべての頂点 $x \in V$ に対し $s - x$ 最短路を s からの Dijkstra 法で求める。次に重み付きグラフ $G' = (V, E, w')$ とすべての頂点 $x \in V$ に対し $t - x$ 最短路を t からの Dijkstra 法で求め、そして……」などのアルゴリズムが得られる。

(3.) はジャッジサーバによる自動採点のために必要なステップである。このステップでは、(2.) で得たがまだ頭の中にある抽象的なアルゴリズムを、計算機上の具体的なコードへ翻訳する。たとえば「$s - t$ 最短路を Dijkstra 法で求めればよい」というときには、C++ なら `priority_queue<pair<int64_t, int>> que; que.emplace(0, s); while (not que.empty()) …` のようなコードが書かれるだろう。

そして、今回の話題は (2.) の「考察」のステップである。このステップは競技プログラミングという行為において中心となるステップであり、「問題を解く」と言うときも主にこのステップに注目している。つまり「競技プログラミングの問題を自動で解きたい」とは「考察を自動で行いたい」ということである。

The topic of this article is (2.), the "Consideration" step. This step is the central step in the act of competitive programming, and when we say "solving a problem", we are mainly focusing on this step. In other words, "We want to automatically solve problems of competition programming" means "We want to do the 'Consideration' step automatically".

1.3 Solvers for competitive programming should leave the formalization to humans.

Even if we divide the process of solving a problem of competitive programming into the three steps of “Interpretation”, “Consideration”, and “Implementation”, and say that we mainly want to do the “Consideration”, the remaining two steps are not easily separated. The problem has to be input in some form, and the solution has to be output in some form. We need to be able to handle the remaining steps of “Interpretation” and “Implementation”.

機械にとって最も面倒なのは「読解」だろう。自然言語の理解という高度な処理が必要になる。競技プログラミングの問題文を以降のステップに支障がでないように読み取るには、表面的に単語を追うだけではおそらく足りず、文章全体の意図を正しく把握する必要があるだろう。競技プログラミングの問題文以外は理解できなくてよいのだとしても、これはかなり困難である。そしてこの「読解」は今回やりたいことではない。ソルバの一部として取り扱うのは諦めて、人間 (あるいは GitHub Copilot のような言語モデル) に任せてしまうべきだろう。

「実装」もそのままの形では機械では扱えない。「数学的な解法を、計算機上の具体的な実装として記述する」と言うときの「数学的な解法」などは、(通常の人間による競技プログラミングにおいては) 人間の頭の中のみ存在するものだからである。人間の頭の中にしかないものは機械には扱えない。「数学的な解法」でなくて「数学的な問題」の場合も同様である。

一方で、機械で扱える形に形式化したものが入力であれば「実装」も機械に可能である。このことはコンパイラを思い出せば分かる。F# や Haskell のような高級言語のソースコードを仮想機械の中間言語や実際の機械語に変換する操作はまさに「実装」であろう。

Therefore, when we were to use a solver to solve a problem of competitive programming, the process could be organized into four steps as follows:

(1.) Interpretation (A human) reads a problem given in natural language and organizes it as a mathematical problem.
(2.) Translation (A human) inputs the mathematical problem as a formalized problem on a computer.
(3.) Consideration' (A solver) computes a solution mechanically to the translated formalized solution.
(4.) Implementation' (A solver) outputs the computed solution as an concrete implementation.

(1.) and (2.) together do the “Formalization” process. Both of these steps are easy for humans. Admittedly, the “Interpretation” step is easy. The step of “Translation” is also easy for humans. We can just write what we read. This is similar to the fact that it is usually not so difficult to type a difficult mathematical expression in LaTeX, even if you don’t understand it at all.

競技プログラミングのソルバが (1.) と (2.) のステップによる「形式化」を人間に任せてしまったとして、そのような条件で残りのステップだけを処理した場合も「競技プログラミングの問題を自動で解けた」と言ってよいだろうか？ これは「自動で解けた」と言ってしまってよいだろう。なぜなら、「形式化」は (機械にとっては難しいとはいえ) 人間にとってはたいてい簡単であるし、通常は重要視されるステップではないからである。困難でも重要でもないステップを省略しても、困難で重要な (3.) の「考察'」のステップをうまく扱うのなら「問題を自動で解く」と言ってよいだろう。様々なことを自動でしてくれるソフトウェアであってもデータの打ち込みは人間の担当であることは多く、今回もそうだったというだけである。

1.4 We should implement a solver for competitive programming in the form of a transpiler.

I said that the input to the solver should be a "formalized problem", but what exactly should be the "formalized problem"? Is it a 3-tuple of (variables as input, variables as output, constraints to be satisfied)? Or is it a graph of a computable function? It could be anything. As long as it can represent the problem, it is unlikely that the choice of input format will make the problem solvable or unsolvable. You can use whatever you like.

However, the input format to a solver greatly affects the ease of implementation of the solver itself. What kind of input is easiest to use? From my experience of implementing several prototypes, I think it is better to use an executable program as a formalized problem. If "executable programs" are the input to the solver, the process of competitive programming can be reduced to the following three steps:

(1.) Interpretation (A human) reads a problem given in natural language and organizes it as a mathematical problem.
(2.) Implementation" (A human) writes a naive solution to the mathematical problem.
(3.) Consideration" (A solver) optimizes the given implementation in its computational complexity.

In this case, the "Consideration" " step in (3.) uses a programming language as both input and output. The output is a faster program that does the same calculation as the input program. This is the same to "optimization". In other words, if we use a "executable program" as the input of a solver, the "solver that automatically solves the problem of competitive programming" becomes a "transpiler with optimization".

The notion of "a solver that automatically solves problems of competitive programming" has been explicated as the concept of "a transpiler with optimization". This is a big step forward. We can now write the "usual one" instead of "some mysterious and amazing thing". Compared to something like artificial intelligence, compilers are much easier to develop. We can use a lot of existing technologies and materials. Another advantage is that the gap between input and output is small. Even a program that simply outputs its input can now be claimed to be the "trivial solver".

Note that it is not obvious to use an "executable program" as a "formalized problem". It may sound natural to hear the answer first. But remember, a "problem" and a "program" are two very different things. You can run a program, but you cannot run a problem. A program has an asymptotic computational complexity, but a problem does not have an asymptotic computational complexity. At least I couldn't get over this gap until I had made a few prototypes.

Other than transpilers, other types of solvers are possible, such as interactive solvers, or solvers consisting of a database of past problems and a search engine.

1.5 Not all problems of competition programming can be solved automatically.

It probably goes without saying, but I should warn you that not all problems of competitive programming can be solved automatically. You shouldn't expect too much. Problems that are suitable for mechanical methods will be solved, but problems that are not suitable for mechanical methods will not

be solved. At least for a while, a solver will be nothing more than a toy software.

Especially for solvers as transpilers, problems that are not suitable for them include the following:

(1.) Problems difficult to implement naive solutions
(2.) Problems that require “intuition” or “inspiration”
(3.) problems that are intrinsically complicated

(1.) は、たとえば幾何の問題である。ソルバの担当範囲以外の部分が難しい。そのような場合は人間が頑張るしかない。考察要素がある問題であってもその考察要素が「愚直に実装するとたいへんだが、うまく工夫すると実装量を減らすことができる」という形であれば、やはりソルバは無力である。

(2.) は、たとえば貪欲法が想定解の問題である。ソルバの担当範囲が難しい。実装にもよるだろうが、基本的にソルバは 1 歩ずつ考察を詰めていけば解けるような問題しか解けないだろう。「証明はできないけどなんとなく正しそう」のような判断は機械とは相性が悪そうである。「突然ですが、ここで $x = f(y, g(y, z))$ とおきます。すると……」のような発見的解法も機械とは相性が悪そうである。手法によっては扱えるかもしれないが、少なくとも、先程の節で述べたような形の機械的解法で解くことはできない。

(3.) は、たとえばゲームの問題がそうだろう。交互ゲームで「先手が必勝である」ことを表現しようと思うだけで、量化子の繰り返しや再帰関数のような扱いにくいものが出てきてしまう。きれいな式で表現できないような問題と機械の相性は悪いだろう。

1.6 About use of a solver in practice

Let’s consider what a solver for competitive programming looks like from view of the users who use it during contests. Basically, it will only be in a position not so different from the current The On-Line Encyclopedia of Integer Sequences (OEIS) and Wolfram|Alpha.

まず明らかに言えることは、一般にソルバは銀の弾丸にはならないということである。もしどんなに優秀なソルバが開発されたとしても、そのようなソルバで解けてしまう問題は出題前に弾かれるだけである。ソルバが利用できるとしても、それは実際の問題の部分問題に対してのみであろう。また、ソルバですべての問題が解けるわけではないので、ユーザは「問題が自動的に解けるのはどのような場合であるか」を大まかに判断できる必要がある。機械的に解けうる問題であっても入力の与え方次第では解けないことも多いので、この判断を行なえることはソルバの利用のために必要である。しかしそのような判断ができるには結局はユーザ自身の能力が要求される。

Also, the input can only be a problem itself when we think a “solver for competition programming problems”, but other possibilities appear when considering practical use in contests. Since only a small portion of the final solution code is intrinsically related to its algorithm, we can use such code fragments as input. Then, something that deals only with such code fragments will be an extension of refactoring support functions of editors. It can be realized as a “plugin to an IDE”. When the user specifies a code fragment, the plugin will rewrite it by reducing the asymptotic computational complexity of that fragment. If the input/output is not the whole source code but small code fragments, it will be convenient to interpret and reuse the results more flexibly.

2 How is a transpiler with optimization implemented?

2.1 Some problems can be solved by mechanical transformation by hand.

ソルバのことはいったん忘れて、手作業による機械的な変形で解くことができる問題を AtCoder Beginner Contest 100 の 4 問目 Patisserie ABC (`https://atcoder.jp/contests/abc100/tasks/abc100_d`) を例に見てみよう。これは整理すると以下のような問題である。そこまで難しくはないので、自分がどうこの問題を解いているのかを観察しながら一度自分で解いてみて、それから続きを読み進めてほしい。

Problem 2.1. *[Patisserie ABC から引用]*

> 高橋君はプロのパティシエになり, *AtCoder Beginner Contest 100* を記念して,「*ABC* 洋菓子店」というお店を開いた.
>
> *ABC* 洋菓子店では, N 種類のケーキを売っている. 各種類のケーキには「綺麗さ」「おいしさ」「人気度」の 3 つの値を持ち, i 種類目のケーキの綺麗さは x_i, おいしさは y_i, 人気度は z_i である. これらの値は 0 以下である可能性もある.
>
> りんごさんは, *ABC* 洋菓子店で M 個のケーキを食べることにした. 彼は次のように, 食べるケーキの組み合わせを選ぶことにした.
>
> - 同じ種類のケーキを 2 個以上食べない.
> - 上の条件を満たしつつ, *(*綺麗さの合計の絶対値*) + (*おいしさの合計の絶対値*) + (*人気度の合計の絶対値*)* が最大になるように選ぶ.
>
> このとき, りんごさんが選ぶケーキの *(*綺麗さの合計の絶対値*) + (*おいしさの合計の絶対値*) + (*人気度の合計の絶対値*)* の最大値を求めなさい.

◇

形式的に書けば以下のようになる。

Problem 2.2. *[形式化された問題 2.1]* 長さ N の整数列 $x = (x_0, x_1, \ldots, x_{N-1})$, $y = (y_0, y_1, \ldots, y_{N-1})$, $z = (z_0, z_1, \ldots, z_{N-1})$ と自然数 $M \leq N$ が与えられる。集合 $X \subseteq N = \{0, 1, \ldots, N-1\}$ に対し

$$f(X) = \left|\sum_{i \in X} x_i\right| + \left|\sum_{i \in X} y_i\right| + \left|\sum_{i \in X} z_i\right|$$

と定義する。このとき $\max\{\, f(X) \mid X \subseteq N \wedge |X| = M \,\}$ を求めよ。 ◇

これは機械的に解ける。このとき求める値 y は

$$y = \max\{\, |a| + b \mid \ldots \,\}$$

の形の式で定義されている。$|a| = \max\{\, a, -a \,\}$ であるので、このような式は

$$y = \max\{\, \max\{\, a, -a \,\} + b \mid \ldots \,\}$$

を経由して

$$y = \max\{\, \max\{\, a + b \mid \ldots \,\}, \max\{\, -a + b \mid \ldots \,\} \,\}$$

と変形できる。この変形を可能な限り繰り返すと、求める式は

$$y = \max \left\{ \begin{array}{l} \max\left\{ +\sum_{i\in X} x_i + \sum_{i\in X} y_i + \sum_{i\in X} z_i \,\middle|\, \ldots X \ldots \right\}, \\ \max\left\{ +\sum_{i\in X} x_i + \sum_{i\in X} y_i - \sum_{i\in X} z_i \,\middle|\, \ldots X \ldots \right\}, \\ \max\left\{ +\sum_{i\in X} x_i - \sum_{i\in X} y_i + \sum_{i\in X} z_i \,\middle|\, \ldots X \ldots \right\}, \\ \max\left\{ +\sum_{i\in X} x_i - \sum_{i\in X} y_i - \sum_{i\in X} z_i \,\middle|\, \ldots X \ldots \right\}, \\ \max\left\{ -\sum_{i\in X} x_i + \sum_{i\in X} y_i + \sum_{i\in X} z_i \,\middle|\, \ldots X \ldots \right\}, \\ \max\left\{ -\sum_{i\in X} x_i + \sum_{i\in X} y_i - \sum_{i\in X} z_i \,\middle|\, \ldots X \ldots \right\}, \\ \max\left\{ -\sum_{i\in X} x_i - \sum_{i\in X} y_i + \sum_{i\in X} z_i \,\middle|\, \ldots X \ldots \right\}, \\ \max\left\{ -\sum_{i\in X} x_i - \sum_{i\in X} y_i - \sum_{i\in X} z_i \,\middle|\, \ldots X \ldots \right\} \end{array} \right\}$$

となる。次に絶対値記号が取れた $\max\{ \pm\sum x_i \pm \sum y_i \pm \sum z_i \mid \ldots X \ldots \}$ の形の 8 本の式の計算が必要である。$x'_i = -x_i$ とおけば $-\sum x_i = \sum x'_i$ であるなど、符号の正負は同様に扱える。つまり次の式の計算だけを考えればよい。

$$\max \left\{ \sum_{i\in X} x_i + \sum_{i\in X} y_i + \sum_{i\in X} z_i \,\middle|\, X \subseteq N \wedge |X| = M \right\}$$

$\sum_{\phi(i)} f(i) + \sum_{\psi(i)} g(i)$ は $\phi(i) \leftrightarrow \psi(i)$ のとき $\sum_{\phi(i)} (f(i) + g(i))$ で置き換えられる。$w_i = x_i + y_i + z_i$ とおいて次の式の計算だけを考えればよい。

$$\max \left\{ \sum_{i\in X} w_i \,\middle|\, X \subseteq N \wedge |X| = M \right\}$$

この形の式「N 個のものから $M \leq N$ 個選んだときの重み総和の最大値」は十分典型的でかつこれ以上分解できそうにないものであり、これを $O(NM)$ で解く方法は既知のものとしてよいだろう[*1]。よってこれは計算できる式であり、機械的な変形により y が $O(NM)$ で計算できることが分かった。N, M の上限と計算時間制限が与えられていれば、これが時間内に計算できることも機械的に判定できる。よってこの問題は機械的に解けると言える。

2.2 Such manual mechanical transformation can also be done by programs.

2.1 節で見たような機械的な変形がプログラムによっても可能であることを、引き続き Patisserie ABC を例に見ていこう。「このようにして機械的に解ける」と主張した上の説明を分析して裏に隠されたものも明示すれば、さらにいくつかの要素に分解することができる。主には以下のような 3 段階のものとして説明できる。

(1.) 自然言語で入力された問題を形式的な形に変形する
(2.) 適用するべき変換規則を選択し適用する
(3.) 十分に簡単な形に変形された問題を実際の実装として出力する

順番に見ていこう。

まず (1.) について。これはソルバによっては扱わず、ソルバの利用者に行わせるものとするのだった。適当な高級言語を用意し、問題をそのままその言語の上に翻訳したものを入力させることになる。入力されたプログラムを解釈した結果は、たとえば図 2.2 のような構文木として理解できる。

次に (2.) について。

問題 2.1 を機械的に解く過程で出現した変形は主に以下の 5 種であった[*4]。

[*1] $O(N \log N)$ や $O(N)$ でもよい。
[*4] $a + (b + c) = (a + b) + c$ や、$a = b$ ならば $f(a) = f(b)$ であるといった自明な等式については省略した。

- $|a| = \max\{a, -a\}$ という等式に基づく変換
- $\max\{\max\{a, b\} + c \mid \dots\} = \max\{\max\{a + c \mid \dots\}, \max\{b + c \mid \dots\}\}$ という等式に基づく変換
- $-\sum_{i \in X} a_i = \sum_{i \in X} b_i$ という等式に基づく変換 (ただし $b_i = -a_i$ とおく)
- $\sum_{i \in X} a_i + \sum_{i \in X} b_i = \sum_{i \in X} c_i$ という等式に基づく変換 (ただし $c_i = a_i + b_i$ とおく)
- $\max\left\{ \sum_{i \in X} w_i \;\middle|\; X \subseteq N \wedge |X| = M \right\}$ から適切な式[*5] への変換

変換規則と構文木が与えられたとき「変換が適用できるかを判定すること」および「変換を適用した結果を得ること」はプログラムによって可能である。実際、このような規則は図 2.2 に示すような木から木への組み換えとして理解できる。

ただしもちろん、これらの変換規則を人の手によって事前に組み込んでおくことが前提である。競技プログラミングをする上で有用な等式を変換の向きと共に列挙することとなる。

変換の個々の適用が可能であることだけでは十分でない。考えられる変換はこれらだけではないため「適用する変換を適切に選択できること」もまた必要だからである。この変換の選択の部分は自明ではない。いまのところ式の複雑さが減少するにように貪欲に変換すればたいていの場合で上手くいくだろうが、実際にはヒューリスティックな探索を行うことになるだろう。

いずれにせよ、トランスパイラの形のソルバはこのような「変換規則の集合」として表現されうる。たとえば実際、Haskell のコンパイラである GHC の最適化機構は “rewrite rule” と呼ばれる規則を用いて、上で見たのとほとんど同じように行われる。

最後に (3.) について。そもそも操作対象である形式的な数式はそれ自体がある種のソースコードである。これを C++ のような高級言語に変換することは難しくない。

3 Related Research

一般のものであれば、以下のものが近いだろう。

- オンライン整数列大辞典 (OEIS) は整数列の巨大なデータベースである。入出力が共に単一の整数であるような問題では、愚直解を書いて得られた整数列で OEIS を検索することで解法が判明することがある。
- Wolfram|Alpha は数学関連の質問応答システムである。問題を解く過程で出てきた複雑な数式を Wolfram|Alpha に渡すことで解法が得られることがある。
- 新井紀子らのロボットは東大に入れるかプロジェクトは
- angr は CTF などで用いられるバイナリ解析フレームワークである。
- DARPA の Cyber Grand Challenge は CTF の問題を自動で
- GitHub Copilot やこれに用いられている OpenAI Codex のような言語モデルは、自然言語の問題文からその愚直解を生成することができる。Daniel Paleka (valkyrie) によるこれを実際に利用してみての分析が Codeforces の blog にある。
- borzunov/cpmoptimize は Python の関数を行列累乗を用いて書き換える decorator である。これに関してのブログ記事やこれに関しての Hacker News のスレッドには多くの関連するプロジェクトが見

[*5] たとえば列 $w' = \mathrm{reverse}(\mathrm{sort}(w))$ を計算した後に $\sum_{i<M} w'_i$ を計算するような式。

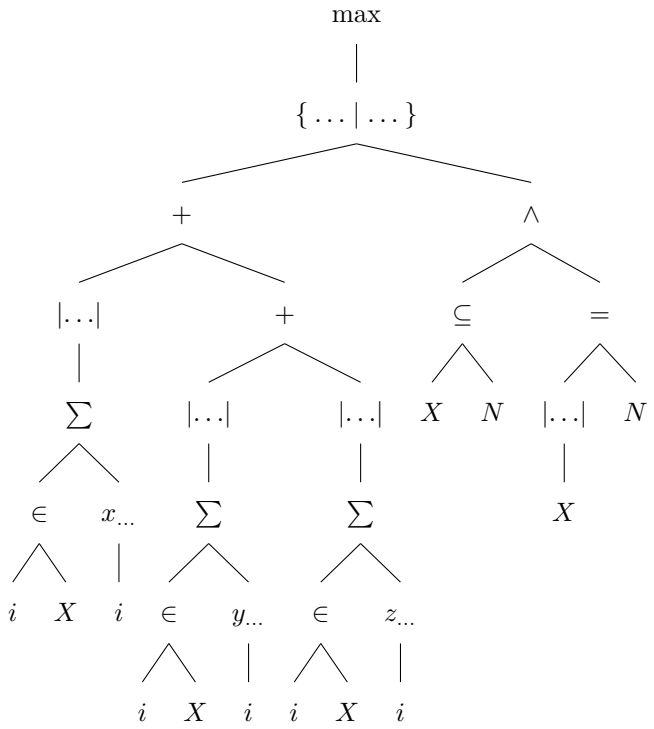

図 1　形式化された問題 2.2 を表現する構文木の一例[*3]

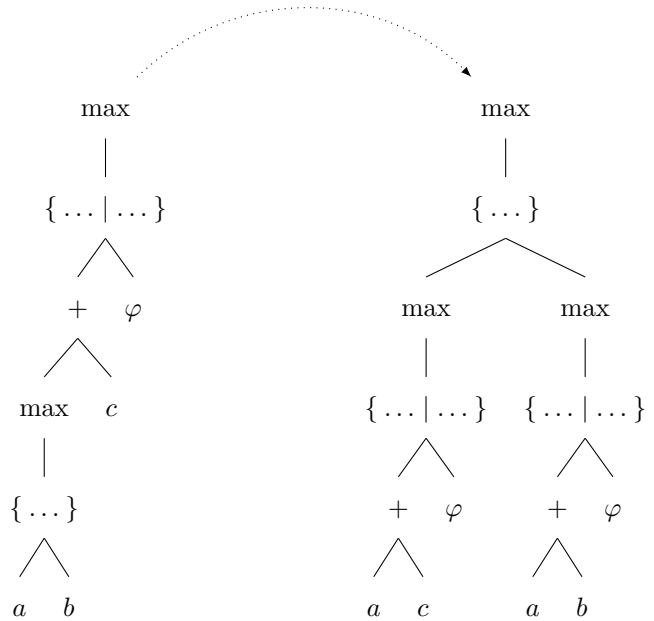

図 2　等式 $\max\{\max\{a,b\}+c\mid\dots\}=\max\{\max\{a+c\mid\dots\},\max\{b+c\mid\dots\}\}$ に基づく変換規則の、構文木の書き換えとしての表現

つかる。

競技プログラミングでの利用を意図したものに限れば、以下のような事例がある。

- 競技プログラミングの解法コードに貼り付けて利用するためのライブラリは広く開発されている。これらは最も素朴な形の「競技プログラミングの問題のソルバ」だと思うこともできるだろう。たとえば beet-aizu/library がある。
- 個別の問題のソルバとしては wata による「Σ電卓」が知られている (wata-orz/SigmaCalculator, Σ電卓 - てきとーな日記) (2009 年)。
- 実装には至っておらず構想のみであるが、一般の問題の解法の自動生成についての試みが kinaba によってなされている (SRM 531 Div2 250 - cafelier@SRM - TopCoder 部 など) (2012 年)。
- AtCoder Regular Contest のある A 問題の解法の完全自動生成が mkotha によって行われたことがある (`https://twitter.com/atcoder/status/538665089931296768`, `https://atcoder.jp/contests/arc030/submissions/286413`) (2014 年)。
- 最適化などは含まれていないが、競技プログラミングでの利用を意図して作られた言語としては laycurse による cLay がある (cLay 概要（version 20201123-1）- ゲームにっき（仮）別館（仮）) (207 年から)。
- マラソンマッチなどでの利用を意図したプログラミング言語として colun による mmlang がある (colun/mmlang) (2021 年から)。データ構造の差分の計算の自動導出機能を持つ。

一般の競技プログラミングの問題を解くソルバを目指しているものとしては、ある程度の形になっているものは見つけられていない。