ADAPTIVE GENERATION OF PROGRAMMING PUZZLES

Anonymous authors
Paper under double-blind review

ABSTRACT

AI today is far from being able to write complex programs. What type of problems would be best for computers to learn to program, and how should such problems be generated? To answer the first question, we suggest programming *puzzles* as a domain for teaching computers programming. A programming puzzle consists of a short program for a Boolean function $f(x)$ and the goal is, given the source code, to find an input that makes f return `True`. Puzzles are objective in that one can easily test the correctness of a given solution x by seeing whether it satisfies f, unlike the most common representations for program synthesis: given input-output pairs or an English problem description, the correctness of a given solution is not determined and is debatable. To address the second question of automatic puzzle generation, we suggest a GAN-like generation algorithm called "Troublemaker" which can generate puzzles targeted at any given puzzle-solver. The main innovation is that it *adapts* to one or more given puzzle-solvers: rather than generating a single dataset of puzzles, Troublemaker generates a diverse set of puzzles that are difficult for those solvers.

1 INTRODUCTION

Computers still perform worse than humans at many elementary programming and mathematical reasoning problems, let alone the AI grand challenge of designing, analyzing, and implementing sophisticated algorithms. For example, many Python and symbolic mathematical interpreters fail or struggle to evaluate the trivial expression $10^{10^{10}} - 10^{10^{10}}$ to 0 or solve $n^{n^n} = 10^{10^{10}}$ for n. While such programs can be taught to immediately answer such puzzles, it points to a fundamental lack of understanding that limits their ability to perform complex programming and mathematics.

The first question faced considering the daunting task of teaching computers to actually program is, *how should one represent programming problems to teach computers?* The ideal representation may be quite different from what suits humans. To see why, consider how a question like "What number logically follows the sequence 1-2-1-4-1-6-1?" (Barrett et al., 2018; Saxton et al., 2019), requires: (a) understanding some English and (b) guessing which sequence the test-writer feels is the most natural among the many consistent sequences. We refer to these biases as *human priors*. Program synthesis problems involve human priors as well. Computers are generally given programming problems in English[1] and/or by examples (Gulwani et al., 2012) such as *"Extract the area code from the phone number"* and/or "`(212)123-4567`" → "`212`". These involve human prior knowledge about phone numbers, which may impede computers from learning the fundamentals of programming and math.

For the purposes of advancing artificial reasoning, the ideal programming problem representation would:

1. Be *objective* in that a candidate solution should be clearly correct or incorrect based on whether or not it satisfies the problem specification. In particular, a problem-solving algorithm should be able to trivially determine correctness without requiring knowledge of English or consulting an answer key.
2. Capture a rich range of useful programming problems from easy to hard.
3. Capture problems that human programmers can solve easily but which foil computers.

[1] After creating NAPS (Zavershynskyi et al., 2018), a "Natural Programming Synthesis" dataset of programming contest problems with English descriptions and input-output examples, one of the authors later conceded that "ML, specifically natural language understanding, is not there yet." (Polosukhin, 2018)

```python
# find an integer n whose square begins with 123456789
def f1(n: int, prefix=123456789):
    return str(n*n).startswith(str(prefix))

# find a set S of powers of 10 summing to 11010010
def f2(S: Set[int], n=11010010):
    return sum({10**i for i in S}) == n

# find a string s with 1,000 As but no two consecutive As
def f3(s: str):
    return s.count("A")==1000 and s.count("AA")==1000

# solve a classic Boolean SAT CNF formula
def f4(x: List[Boolean]):
    return (x[1] or x[2]) and (not x[1] or not x[2])

# find a nontrivial integer factor of 10987654321
def f5(m: int, n=10987654321):
    return 1 < m < n and n % m == 0
```

Figure 1: Sample puzzle problems with possible answers $n = 111111111$ (Python: `int("1"*9)`), $S = \{1, 4, 6, 7\}$, a concatenation of 1000 copies of "AB" (Python: `s="AB"*1000`), `x=[True, False]`, and $m = 7$ (call `f5` repeatedly to factor). Problems range from easy to very hard to unsolvable.

To this end, we propose *Programming Puzzles* in which one is give the source code of a function f in a fixed programming language, such as Python, and the goal is to find an input x such that $f(x)$ returns the Boolean True. As illustrated in Figure 1, short puzzles can capture extremely difficult problems such as factoring or subset-sum, but they can also represent easy questions such as: what list x when reversed is `[1,2,3,4,5]`, or what number x satisfies $(x+1)^{x+1} == 100^{100}$? And such puzzles are objective in that a candidate solution can easily be evaluated for correctness.

We use existing techniques for learning to solve such problems – certainly innovation is needed but is left for future work. Instead, our contribution is a "Troublemaker" (TM) algorithm that automatically generates puzzles in an *adaptive* manner to challenge any given puzzle-solving system. A primitive system may be given a set of easy puzzles, while an advanced system may be given hard puzzles (or easy puzzles if a certain weakness is identified). Automatic problem generation has been applied to program synthesis (Balog et al., 2016; Christakopoulou & Kalai, 2018) and numerous related domains such as mathematical problems (Saxton et al., 2019) and IQ test problems (Barrett et al., 2018). However, to the best of our knowledge this prior work generated problems in a non-adaptive fashion, independent of which solver would be used to solve them.

TM *searches* problems that are hard for the solver. Given a *trainable* solver, this results in a GAN-like setup where TM finds increasingly hard problems as the training progresses. In this work, we employ TM to output *solvable* puzzle problems along with a host of symbolic and neural solvers. Troublemaker uses a neural network to identify problems.

The contributions of this paper are: (1) introducing programming *puzzles*, a class of objective problems that can be used to evaluate and improve the reasoning ability of existing AI systems; (2) introducing an *adaptive* puzzle generator that can be targeted at any give puzzle solver(s). Note that this paper does *not* set out to introduce novel approaches for writing puzzle solvers – we use existing approaches including solvers based on existing approaches to solve puzzles. We focus on adaptive problem generation and establishing the puzzle domain as especially suitable for learning about programming, leaving the many interesting questions about how to best solve puzzles to future work.

It is important to note that not all programming problems can be nicely defined as puzzles. In some problems, writing the puzzle itself can be as much work as solving the problem. Examples of this nature include adding two numbers (without using a builtin addition routine) or computing the probability of getting a total of 30 when rolling 10 dice. Other problems involve human knowledge

```
bool equation := term == term
float term := term + term | term * term
           | term - term | term / term
           // evaluates to 1.0 if equality holds, 0.0 otherwise
           | _float(term == 0)
           // various floating point constants
           | π | 0.0 | e
           // variable
           | x
```

Figure 2: Simple grammar that defines a family of PSAT problems with floating point solutions.

and data, such as alphabetizing a list of names by last names, where the syntax rules determining how to tell whether *Mary De Leon* is sorted under *D* or *L*. However, the formal theoretical basis for the class of problems whose solutions can be quickly checked (in polynomial time) is NP, which contains P and a great many problems ranging from easy to hard, including shortest-path, factoring, and finding a neural network that achieves a training loss less than a given threshold.

2 DEFINITIONS

We assume a language of *programs* L defined as a Context Free Grammar (CFG). Each syntax tree $T \in L$ describes a program $p_T : \mathcal{X} \to \mathcal{X}$, where for simplicity, we assume inputs and outputs belong to a common set $\mathcal{X}$. When T is clear from context, we write p instead of p_T. Both L and $\mathcal{X}$ could simply be strings or more generally $\mathcal{X}$ may include other types such as numbers, arrays, errors, and so forth (or a standard object representation such as JSON may be used). We make two assumptions on $\mathcal{X}$: (1) $L \subseteq \mathcal{X}$ so programs themselves can be inputs/outputs, and (2) there are two special elements $\top, \bot \in \mathcal{X}$ respectively representing the Booleans *True* and *False*. We further denote by $\text{time}(p, x) > 0$ the (possibly infinite) amount of time that program p takes to run on input x.

A programming puzzle or just *puzzle* is simply a program p defined as a syntax tree $T \in L$, and a *solution* $x \in \mathcal{X}$ is an input that *satisfies* $p(x) = \top$. A problem is *solvable* if it has any solution x. Good "problems" are typically short, instructive, and solvable "snippets," but none of these are formal requirements.

A *puzzle solver* or just *solver* is a procedure $S : L \to \mathcal{X}$ and is said to solve problem p_T within a time budget B if $p_T(S(T)) = \top$ with $\text{time}(S, T) \leq B$. In the programming contest analogy, the program p is a scoring function which is made transparent.

A *learning solver* $\ell : L^{\times n} \to (L \to \mathcal{X})$ is a program that takes multiple training problems $T_1, \ldots, T_n$ as input and outputs a solver program $S \in L$. Answers to the training problems, if teacher chooses to make training answers available, may be provided for example directly in comments in the code in T_i. We do not formally make this a requirement of the model since some problems may be unsolvable while other problems may have multiple solutions that may be beneficial to share in the training data.

The language of Program Puzzles are known to be trivially NP-complete when given a time limit expressed in unary and the programming language is that of a Universal Turing Machine. This problem is sometimes called the Bounded Halting Problem (e.g. Goldreich, 2010, p. 102).

A natural generalization is to consider optimization programs that output a number $p(x) \in [0, 1]$ representing a score (any $t(x) \notin [0, 1]$ or error can be taken to be a score of 0). In this paper we focus on Boolean puzzles, and through the standard optimization/feasibility reduction any such real-valued optimization problem can be reduced to solving a sequence of puzzles where $p_\theta(x)$ tests whether or not $p(x) \leq \theta$ for various thresholds $\theta \in [0, 1]$ and then grid or binary search can be used on θ. However, just as linear programming (optimization) algorithms employ different implementations than their feasibility counterparts, the white-box program optimization problem is worth independent consideration.

```
lhs == rhs → p**lhs == p**rhs
lhs == rhs → p+lhs == p+rhs
```

Figure 3: Tree rewrite rules for `float` puzzles

3 PROGRAMMING PUZZLES

Recall that we defined a puzzle p to be described by a syntax tree $T \in L$. To be able to generate puzzles that involve complex reasoning patterns, the language needs to be expressive enough to represent a wide range of puzzles. For example, the grammar shown in Fig 2 can represent a wide range of equations with a single variable. Further, note that such a grammar can be trivially extended to be able to represent complicated equations (for *e.g.* involving trignometric ratios, exponentiation, *etc*). Randomly sampling programs from this language is not useful as it is possible to generate potentially unsolvable problems *i.e.* there is no $x \in \mathcal{X}$ s.t. $p(x) = \top$.

Generating Solvable Programs. In order to generate solvable puzzles, we use domain-specific knowledge to *convert* a puzzle sampled from L to a solvable one. To continue with the `float`-puzzle example, given an equation $a(x) == b(x)$ where both $a, b \in L_{term}$, we can convert it to a *solvable* puzzle at some arbitrarily chosen x_0 by modifying the program to be $a(x) = b(x) - k$ where $k = a(x_0) - b(x_0)$. Note that a solver that indicates *unsatisfiability* can be trivially used to filter sampled programs for solvability; however, such a solver may not be efficient find for the broad class of programming puzzles we are interested [2].

Generating Complex Puzzles. A standard strategy employed by experts for posing mathematical problems is *chaining* (Silver, 1994). Chaining expands on an existing problem (and solution) such that finding the solution for the modified problem requires solving the original problem as an intermediate step. For instance, in the case of our `float`-puzzle problems, exponentiating both sides of the equation is a good example of *chaining* – while preserving the *solvability*, it requires solving the original problem as an intermediate step (after taking log on both sides). In this work, we incorporate this strategy to generate *complex* puzzles that potentially require multiple steps of reasoning. In our setting of generating programming puzzles, we incorporate this problem posing strategy via tree-rewriting rules that transform an existing tree to another tree in the language – formally, a tree-rewriting rule $R : L \rightarrow L$ For the example of `float`-puzzles, Fig. 8 provides a bank of tree-rewrite rules that can be used to produce complex solvable puzzles.

Further, this approach is general enough that it can be easily extended to other domains like `int`-puzzles, `set`-puzzles, *etc*. For example, consider the sub-set sum problem – find set $\mathcal{B}, \mathcal{B} \subseteq \mathcal{A}$ such that the sum of elements in $\mathcal{B}$ is some intenger K. If this `set`-puzzle is solvable, chaining can be used to produce more complex puzzles by replacing the ground-set with $\mathcal{A} \cup \mathcal{C}$ for some non-empty integer set $\mathcal{C}$. While this transformation still preserves the solvability of the puzzle, it also makes it harder for a solver that depends on the size of the ground set.

4 TROUBLEMAKER

In this section, we present *TroubleMaker*, a procedure to generate programming puzzles given access to a solver – in a manner that progressively finds puzzles harder for the solver as the name suggests. While Section 3 defines the grammar and tree transformations to generate PSAT problems, randomly sampling productions and tree transofrmatoins may not lead to the generation of *interesting* problems. For the purpose of this paper, we assume interesting problems are *hard* puzzles whose solution a solver cannot obtain within the given time budget *i.e.* $\texttt{time}(S, p_T) > B$). However, note that the TM procedure itself is agnostic to this assumptiion and can be trivially modified to generate problems for different notions of interesting-ness. For example, we might want to generate instructional problems solving which helps in solving a target problem – for *e.g.* learning to solve $10^2 - 10^2$ might be helpful in solving $10^{10^{10}} - 10^{10^{10}}$ by creating the necessary bias that any number subtracted from itself is always 0.

[2]Recall that programming puzzles are in NP and hence not being able to find a solution does not imply unsolvability.

```
bool equation (1) := term == term
float term := (0.1) term + term | (0.2) term * term
           | (0.05) term - term | (0.05) term / term
           // evaluates to 1.0 if equality holds, 0.0 otherwise
           | (0.15) _float(term == 0)
           // various floating point constants
           | (0.05) π | (0.05) 0.0 | (0.05) e
           // variable
           | (0.3) x
```

Figure 4: Simple Grammar shown with associated probabilistic weights. As comapred the grammar in Fig 2, programs sampled from this grammar include complex terms due to the increased preference for multiplication of terms.

We now detail the working of the TM procedure and explain two strategies to generate hard problems for a given solver.

Generation with Probabilistic Grammars. The first *troublemaker* strategy we consider to improve the generation based on the solver is to use a probabilistic grammar as shown in 4. As can be seen, depending on the solver, different rules can be encouraged to bias the production of problems. For instance, the weights in Fig 4 display a higher preference for multiplication of terms – this lends itself conveniently to the generation of puzzles (equations) higher degree polynomials. Note that the probability of outputting a program p_T is the product of individual rules in it *i.e.* $\prod_{r \in T} \Pr(r)$. Further, as mentioned we are trying to produce *hard* problems or in other words, maximize $\log \texttt{time}(S, T)$. It is now easy to see that the weights θ of the probabilistic grammar can be learnt via REINFORCE Williams (1992) – specifically maximizing $E_{p_T \sim L_\theta}[\log \texttt{time}(S, p_T)]$.

While this system allows us to tailor puzzle generation for a given solver, it only allows us to tune global preferences (*i.e.* encourage certain rules *always*) as opposed to making context-dependent changes (*i.e.* rule A is preferred conditioned on the current partial tree). Factoring the tree produced so far before predicting the next rule to use gives us fine-grained control over the generation process – allowing us to tune it towards producing targeted puzzles for a given solver.

Neural Guided Generation. Improving from the probabilistic grammar based generation scheme for *troublemakers*, we now motivate a neural network based generation scheme that outputs the probability of the next rule conditioned on the partial tree produced so far. Given $\phi(\tilde{T})$, an encoding of the partial tree $\tilde{T}$, the neural network outputs the probabilities of choosing the next rule. Similar to the previous version of TM, REINFORCE is used to learn the parameters of the network.

To obtain a suitable representation for the tree $T \in L$, we consider two approaches in this work. The first representation is naive, encoding a in-tree traversal of the tree via an LSTM network (Hochreiter & Schmidhuber, 1997). The second encoding scheme uses GNN-FiLM (Brockschmidt, 2019), a more recent Graph Neural Network architecture that allows for feature-wise linear modulations, achieving performance improvements on standard graph-based tasks like node classification.

TroubleMakers with Trainable Solver. While solvers (*e.g.* symbolic solvers) are static, neural network based solvers are trainable and can be improved with more data. Incorporating TM with such a trainable solver leads to an *adversarial* setting where the weights of the puzzle generator are updated to produce harder problems for the solver and similarly, the solver's weights are updated based on the newly generated *hard* problems (as shown in Alg. 1).

In this work, we study two variants of solvers:

1. Induction Solver (IS): This neural solver directly outputs the solution of the puzzle given a representation of the puzzle. In the case of structured outputs (like sets), the solver sequentially outputs elements along with a stop token to signal the end of generation.

2. Synthesis Solver (SS): This solver is similar to the neural generator – given a grammar and a bank of constants, it uses an encoding of the puzzle and the partial tree (traversed so far) to select the next rule – choosing to go down the most likely branch.

In this work, both solvers use a GNN-FiLM representation of the puzzle p_T.

Algorithm 1 TroubleMakers with Trainable Solver

Result: neural generator weights, θ_g^K
given: θ_g^0, θ_s^0, L
for k=*1,..,K* **do**
  Obtain D_k, puzzles generated using θ_g^{k-1}
  Update θ_s^k based on D_k
  Update θ_g^k based on new solver weights
end

5 EXPERIMENTS

In this section, we present results for the TroubleMaker approach to generate hard problems for a given solver. We consider `float`, `int` and `intset` puzzles in this work to demonstrate the flexibility of the proposed puzzle generation framework. We provide the full grammar and tree-rewrites in Appendix. Before discussing these results, we discuss the details of the puzzle generators, solvers and metrics used.

Puzzle Generators. As discussed earlier, we study both probabilistic grammar based and neural-guided problem generators. Further, in the grammars used, we allow a new *copy* operator that can copy a node of type τ from the partial tree generated so far as opposed to generating a tree from L_{NT} where NT is of type τ. The introduction of this operator increases the likelihood of generating puzzles with recursive structure (*e.g.* $10^{10} - x^{x^x} = 0$) – a desirable quality that often results in interesting reasoning patterns. As discussed in Sec. 4, both generators are trained using REINFORCE. Finally, note that the recursion depth was capped at 10 for the production of all puzzles (from the grammar and including tree rewrite rules).

Solvers. In this work, we consider multiple solvers and in fact, use TM to produce programs that are in general *hard* for all solvers. The solvers used are as follows:

1. Enumerative Solver (ES): Given a grammar to generate the solution, this solver performs enumerative search to find a satisfying solution. For the sake of simplicity, we provide the solver with the same grammar used to generate a puzzle (of the same solution type). However, a significant difference is that the bank of constants available to the solver is *large* and corresponds to all the unique constants generated so far.
2. Guided Solver (GS): This solver builds on top of the enumerative solver by using a neural network to guide the search process conditioned on the puzzle. By learning heuristics from data, this solver learns to accelerate enumerative search by pruning away large portions of the search space. Unlike other solvers, being *eps* close to a satisfying solution is sufficient for the grid solver. (We set $\epsilon = 10^{-10}$ in our experiments)
3. Grid Search (GrS): This solver, as the name suggests, searches in the solution space, narrowing down to a satisfying solution based on comparisons. This solver is not a general-purpose puzzle solver and is used only in the context of floating point and integer puzzles.
4. Sympy[3]: Sympy is a python library for symbolic mathematics and similar to GrS, it is used to only solve floating point and integer problems.

Finally, note that apart from GS, the rest are not trainable solvers and so, cannot be used as part of adversarial method described in Alg 1.

Metrics. To evaluate the performance of the generators, we use the following two metrics:

[3]https://www.sympy.org

Puzzle Generators	Enumerative Solver	Guided Solver	Grid Solver	Sympy
float-puzzles				
Random	1.21	0.92	1.5	0.05
Prob	2.47	1.95	2.22	1.02
Guided	2.78	1.99	2.34	1.45
int-puzzles				
Random	3.06	2.93	3.78	4.21
Prob	3.22	2.95	3.82	4.33
Guided	3.45	3.15	3.85	4.54
int-set-puzzles				
Random	4.12	3.83	-	-
Prob	4.33	4.02	-	-
Guided	4.51	4.19	-	-

Table 1: As can be seen from the table, both generators – probabilistic grammar based (Prob) and neural guided (Guided) lead to improvements (over randomly sampled puzzles) in the **time** required for the solvers, with the guided solver showing the highest gains. Note that randomly sampled puzzles can sometimes be unsolvable – in which case the solver just times out. Further, all values are measured in seconds and solvers have a time bound $B = 5s$.

Puzzle Generators	Enumerative Solver	Guided Solver	Grid Solver	Sympy
float-puzzles				
Random	5.42	5.43	5.39	5.42
Prob	4.84	4.29	3.95	4.11
Guided	4.81	4.45	3.67	4.07

Table 2: Observe that unique-rules metric for both probabilistic grammar based and Guided search is only slightly lesser than random sampling – indicating that the outputted programs are sufficiently "diverse", employing different strategies to "trouble" solvers.

1. Time taken by the solvers (time): This is computed as the *average* (log) time taken to solve a set of 1000 generated problems. This metric (higher is better) corresponds to the "hardness" of the produced solutions.
2. Number of Unique Rules used (unique-rules): This is computed as the number of unique rules present in a problem, averaged over a set of 1000 generated problems. While not something we optimize for, an increase in this metric corresponds to an increase in the diversity of the problems generated.

Both solvers and puzzle generators, when trainable, were optimized using Adam (Kingma & Ba, 2014) with a learning rate of 10^{-4}. All the LSTM networks used a hidden size of 256 and in the case of GNN-FiLM architectures, 3 propagation steps and a node representation size of 256 were used.

Results. As can be seen from Tab. 1, the guided generator finds puzzles with the highest (average) time required across all solvers. Note that in the case of the guided solver, the only trainable solver, the numbers shown are after $K = 100$ iterations with each iteration sampling ~ 2000 puzzles. From Tab 2, we can see that the puzzles produced by both probabilistic and neural-guided generators are reasonably diverse across all solvers – as the proposed generators obtain a score slightly lesser than randomly generated puzzles. Further, the generators are relatively less diverse when trained against the grid solver – from a qualitative inspection of the results, this is because of the frequent usage of the rule $term := \texttt{_float}(term == 0)$ (see Fig 2), that is hard for a grid solver when the solutions are random floating point numbers.

```
# float puzzle
def f1(x: float):
    return ((x + x) + (float(x == x / (x + 2.)))) == -0.1474945082398796

# int puzzle
def f2(x: int):
    return 2**(x**2)) == 16

# int-set puzzle
def f3(x: set):
    return sum(x) == 47 and x in union({-2,-1,0}, range(40,103))
```

s

Figure 5: Sample generations corresponding to each type considered in this work. The `float` sample uses tree-rewrites to substitute a constant in the equation with another term and a constant. Similarly, the `int` puzzle uses exponentiation and `int-set` example applies a tree rewrite of replacing the ground set with a union of itself and another set.

Further, we observe that induction solvers that directly regress to the output are significantly worse off (solving less than a fifth of the problems generated) – TM does not lead to improvements as even small improvements overwhelm the solver. Regarding synthesis based solvers, we observe that GNN-FiLM architecture outperforms LSTM-based solvers. Finally, we provide some examples of generated problems in Fig 5.

6 Comparison to Related Work

The prior work on Program Synthesis is too large to surveyed here (see the survey by Gulwani et al., 2017). As mentioned, prior work on program synthesis largely works from input-output examples x_i, y_i where the goal is to find a "natural" function f such that $f(x_i) = y_i$. As discussed, this representation is *subjective* – the issue is not that there are multiple correct programs but rather that the correctness of an given solution f is debatable and cannot be readily determined from the examples alone. Similarly, the other body of work also covered by Gulwani et al. (2017) describes problems in a human language such as English, for which correctness is even more debatable. From the point of view of expressing the intent of an end user, these representations may be quite natural and the ambiguity may be inherent to the problem. (Evaluation may be made objective using a hidden test set of input-output pairs.) However, for the purpose of teaching computers programming and reasoning basics, such ambiguity impedes learning.

An exception is closely-related work that inspired this paper, where Christakopoulou & Kalai (2018) considered a problem that is essentially a harder special case of puzzles, but in which the specification was a program that took a whole problem-solving program as input. For instance, say the goal was to design an algorithm for finding a repeated number in a list. Their specification would take as input a function that should find duplicate numbers in any list with duplicates. The specification would be a program that would generate a number of random lists with duplicates in them and ensure that provided function succeeded in finding all the duplicates. Like our work, this specification in the form of source code is given to the problem solver.

There is a dual view of puzzles in that their solutions are programs, and the puzzle itself serves as a very simple *formal specification*. For example the puzzle `f3` of Figure 1 has a solution of 1,000 copies of the string `'AB'` which can be written in Python as a trivial program that takes no input and outputs `'AB'*1000`. An older literature on *deductive* program synthesis (e.g., Manna & Waldinger, 1980) defined the goal by a formal specification and aimed to find a program that satisfied the specification. The greater expressive power in some languages inherently means that it is not always trivial to determine whether a solution satisfies the specification.

Prior work has also automatically generated programming problems (Balog et al., 2016; Christakopoulou & Kalai, 2018). However, the way they generated problems was non-adaptive, a dataset was generated independent of the solver.

REFERENCES

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

David GT Barrett, Felix Hill, Adam Santoro, Ari S Morcos, and Timothy Lillicrap. Measuring abstract reasoning in neural networks. *arXiv preprint arXiv:1807.04225*, 2018.

Marc Brockschmidt. Gnn-film: Graph neural networks with feature-wise linear modulation. *arXiv preprint arXiv:1906.12192*, 2019.

Konstantina Christakopoulou and Adam Tauman Kalai. Glass-box program synthesis: A machine learning approach. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

O. Goldreich. *P, NP, and NP-Completeness: The Basics of Computational Complexity*. Cambridge University Press, 2010. ISBN 9781139490092. URL https://books.google.com/books?id=rej7fcC7ed4C.

Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *TOPLAS*, 2(1): 90–121, 1980.

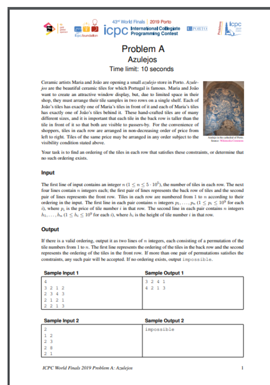
Illia Polosukhin. NEAR.ai → NEAR Protocol, August 2018. https://nearprotocol.com/blog/near-ai-near-protocol/, Last accessed on 2019-09-24.

David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. *CoRR*, abs/1904.01557, 2019. URL http://arxiv.org/abs/1904.01557.

Edward A Silver. On mathematical problem posing. *For the learning of mathematics*, 14(1):19–28, 1994.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Maksym Zavershynskyi, Alexander Skidanov, and Illia Polosukhin. NAPS: natural program synthesis dataset. *CoRR*, abs/1807.03168, 2018. URL http://arxiv.org/abs/1807.03168.



43rd World Finals | 2019 Porto

icpc International Collegiate Programming Contest

Problem A

Azulejos

Time limit: 10 seconds

Ceramic artists Maria and João are opening a small *azulejo* store in Porto. *Azulejos* are the beautiful ceramic tiles for which Portugal is famous. Maria and João want to create an attractive window display, but, due to limited space in their shop, they must arrange their tile samples in two rows on a single shelf. Each of João's tiles has exactly one of Maria's tiles in front of it and each of Maria's tiles has exactly one of João's tiles behind it. These hand-crafted tiles are of many different sizes, and it is important that each tile in the back row is taller than the tile in front of it so that both are visible to passers-by. For the convenience of shoppers, tiles in each row are arranged in non-decreasing order of price from left to right. Tiles of the same price may be arranged in any order subject to the visibility condition stated above.

Azulejo in the cathedral of Porto. Source: Wikimedia Commons

Your task is to find an ordering of the tiles in each row that satisfies these constraints, or determine that no such ordering exists.

Input

The first line of input contains an integer n ($1 \leq n \leq 5 \cdot 10^5$), the number of tiles in each row. The next four lines contain n integers each; the first pair of lines represents the back row of tiles and the second pair of lines represents the front row. Tiles in each row are numbered from 1 to n according to their ordering in the input. The first line in each pair contains n integers $p_1, \ldots, p_n$ ($1 \leq p_i \leq 10^9$ for each i), where p_i is the price of tile number i in that row. The second line in each pair contains n integers $h_1, \ldots, h_n$ ($1 \leq h_i \leq 10^9$ for each i), where h_i is the height of tile number i in that row.

Output

If there is a valid ordering, output it as two lines of n integers, each consisting of a permutation of the tile numbers from 1 to n. The first line represents the ordering of the tiles in the back row and the second represents the ordering of the tiles in the front row. If more than one pair of permutations satisfies the constraints, any such pair will be accepted. If no ordering exists, output `impossible`.

Sample Input 1	Sample Output 1
4 3 2 1 2 2 3 4 3 2 1 2 1 2 2 1 3	3 2 4 1 4 2 1 3

Sample Input 2	Sample Output 2
2 1 2 2 3 2 8 2 1	impossible

ICPC World Finals 2019 Problem A: Azulejos 1

```python
def score_azulejos(s: List[int], t: List[int],
                   p1=[3, 2, 1, 2], h1=[2, 3, 4, 3],
                   p2=[2, 1, 2, 1], h2=[2, 2, 1, 3]):
    n = len(p1)
    return (
        sorted(s) == sorted(t) == list(range(n))
        and
        all(h1[s[i]]>h2[t[i]] for i in range(n))
        and
        all(p1[s[i]]<=p1[s[i+1]] for i in range(n-1))
        and
        all(p2[t[i]]<=p2[t[i+1]] for i in range(n-1))
       )
```

Figure 6: ICPC Problem 2019A in English (left) and programming puzzle (right). In the PSAT, `p1, p2` are arrays of n prices, `h1, h2` are arrays of n heights. The goal is to find `s, t`, required to be lists of integers in the function specification. They are further tested to be permutations by checking if sorting them yields `list(range(n))`, the list of numbers from 1 to `n`.

A ICPC PROBLEM

As a more complicated but illuminating example, consider the first problem of the 2019 International Collegiate Programming Contest (ICPC). At its core, the goal is, given input two matrices $P, H \in \mathbb{R}^{2\times n}$, to find a pair of permutations σ, τ on $\{1, 2, \ldots, n\}$ such that,

$$h_{1\sigma_i} < h_{2\tau_i} \text{ and } p_{1\sigma_i} \leq p_{1\sigma_{i+1}} \text{ and } p_{2\tau_i} \leq p_{2\tau_{i+1}} \text{ for all } i$$

The permutations σ and τ are to be applied to the two rows of P, which correspond to prices. When permuted, the prices are to be non-decreasing. If there are duplicate values in P, this constraint leaves flexibility in σ and τ which must be chosen so as to that when heights, encoded in H, are also permuted by σ and τ, the first row is greater than the second coordinate-wise. Figure 6 shows the problem in English and our PSAT version.

Note that there are a few important differences:

Input parsing. First, the problem like many requires reading and parsing the data from a file rather than receiving it as lists of integers. However, much prior work has successfully shown how to automatically parse such data (see the survey by Gulwani et al., 2017), so we focus on the algorithmic questions.

Solvability. Second, the problem posed in the contest asks users to print "impossible" if the problem has no solution. For such problems, the corresponding puzzle is unsolvable – there simply aren't any valid input which makes it return True. If one wanted to have an objective question where one could verify that an answer of "impossible" was correct, the puzzle would need some sort of proof that the problem is unsolvable which is much more difficult. However, the uncertainty of knowing whether an unsolved puzzle is solvable is not critical for optimization. For example, imagine a solver takes a test with 1,000 problems and knows with certainty that one solved 300 of them and didn't solve the remaining 700. Now, suppose changing a parameter of the system makes it so that it solves an additional 20 problems. This increase in objective is desirable, and even though a humans may be frustrated and even slower to solve a problem without knowing whether or not it is solvable, a computer algorithm will experience no such frustration or slowdown.

Solving instances. For people, there is a difference between submitting a program that solves a given problem with varied inputs and just submitting solutions for a number of given inputs – the latter may be easier as they may identify peculiarities in the inputs which make them easier to solve (or they may even solve them by hand). However, for computers this distinction is not crucial. Given a program that is only capable of solving individual instances, that same program can itself be submitted as a solution to the general problem. This distinction is similar to a classification problem in which a binary classifier must be returned versus one in which unlabeled test data is provided and the

```
bool equation := term == term
float term := term + term | term * term
           | term - term | term / term
           | term**term | -term
           | _float(term == 0) | _float(term != 0)
           | copy_float() // copies a float node from current parital tree
           | π | 0.0 | e
           // variable
           | x
```

Figure 7: Grammar that defines a family of PSAT problems with floating point solutions.

```
lhs == rhs → p**lhs == p**rhs
lhs == rhs → p+lhs == p+rhs
lhs == rhs→ p*lhs == p*rhs
```

Figure 8: Tree rewrite rules used for floating point puzzles.

labels alone must be submitted. Similar techniques are often used for computers to solve these two closely-related problems if the test set is sufficiently large.

B GRAMMARS AND TREE-REWRITE RULES.

We present the full grammars for the three domains used in this work. The integer rules are similar to the float domain except having additional functions – `factorial` and `modulo` operation.

```
bool int in term | sum(term) == int
set term := ∅ | {int_term} | {int_term, int_term}
                | {int_term, int_term, int_term}
                | union(term, term) | intersection(term, term) |
                  difference(term, term)
                | {e for e in range(int_term, int_term)} \\
int int_term := ... // integer grammar
```

Figure 9: Grammar for `int-set`-domain.