From Program Synthesis to Optimal Program Synthesis

Joaquin Reyna

Department of Computer Science
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA
magitek7@gmail.com
1. Need for Data Processing: A Brief Reminder

- One of the main objectives of science is to describe the world.
- This means that we know the values of different physical quantities.
- There are many physical characteristics which are difficult to measure directly.
- E.g.: distance to a star, amount of oil in a well.
- We can measure them indirectly:
  - we measure the values of related easier-to-measure characteristics $x_1, \ldots, x_n$,
  - find (and list) all possible relations between these characteristics $x_i$ and the desired quantity $y$;
  - based on these relations, we design an algorithm that estimates $y$ based the $x_1, \ldots, x_n$: $y = f(x_1, \ldots, x_n)$. 

2. Other Problems for Which Data Processing Is Needed

- An important goal of science is to *predict* the future values $y$ based on the current values $x_1, \ldots, x_n$.

- In many cases, we only know some relations between $y$, $x_i$, and maybe some auxiliary characteristics.

- Our objective is to design an algorithm that transforms the value $x_1, \ldots, x_n$ into the desired prediction $y$.

- In engineering, we want to design an object with given characteristics $x_1, \ldots, x_n$.

- In many cases, we only know the relations between the design parameters $y$ and the values $x_i$.

- We need to design an algorithm producing $y$ based on the $x_i$. 
3. Program Synthesis

- A general problem:
  - we know the values of some of the quantities $x_1, \ldots, x_n$, and
  - we know the relations between these quantities, the desired quantity $y$, and some other quantities.

- Our objective is to use the known values and the known relations to find the values of the desired quantities.

- Traditionally, practitioners use their own creativity to come up with an algorithm.

- It turns out that we can have a system that automatically synthesizes the desired program.
4. Toy Example

- A triangle is described by its angles $A, B, C$ and side lengths $a, b, c$.

- We know the following relations between them:
  - $A + B + C = \pi$ (the sum of the angles is $180^\circ$, or $\pi$ radians),
  - $a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos(C) = c^2$ and similar expressions for $a$ and $b$ (cosine theorem), and
  - $\frac{a}{\sin(A)} = \frac{b}{\sin(B)} = \frac{c}{\sin(C)}$ (sine theorem).

- Typical questions:
  - If we know $a, b$ and $c$, can we determine $A$? and if yes, how?
  - If we know $a, b$ and $A$, how to compute $B$? and if yes, how?
5. Preparing for the Program Synthesis: Idea

- First, we analyze which quantities are directly computable from which.
- Suppose that we have a relation \( F(x, y, z) = 0 \).
- If we know all of these values but one, then we have an equation with one unknown.
- So, if we already know \( x \) and \( y \), then we are able to compute \( z \).
- We will describe this implication as \( x, y \rightarrow z \).
- Similarly, if we know \( x \) and \( z \), then we can compute \( y \), and from \( y \) and \( z \) we can compute \( x \).
- So each equation leads to as many computability relations as there are quantities in it.
- In our case we get three computability relations:
  \[
  x, y \rightarrow z; \quad x, z \rightarrow y; \quad y, z \rightarrow x.
  \]
6. Example

In the triangle case, the relations turn into the following formulas:

- \( A, B \rightarrow C; \ B, C \rightarrow A; \ A, C \rightarrow B; \)
  (these three come from the equation \( A + B + C = \pi \))

- \( A, a, b \rightarrow B; \ A, a, B \rightarrow b; \ldots \)
  (from sine theorem), and

- \( a, b, C \rightarrow c; \ a, b, c \rightarrow C; \ a, c, C \rightarrow b; \ b, c, C \rightarrow a; \ldots \)
  (from cosine theorem).
7. Wave Algorithm for Program Synthesis

- Algorithm: we first mark the variables that we know.
- Then, repeatedly:
  - we find the rules for which all conditions are marked and the conclusion is not,
  - and mark the conclusion.
- We stop when no new marks are assigned.
- If the desired quantity $y$ is not marked, then we cannot compute this quantity.
- If $y$ is marked, we can combine the corresponding algorithms into a program for computing $y$ based on $x_i$. 
8. Triangle Example

• Suppose that we know $A$, $B$, we want to find $C$, $a$.
• Then, we first mark $A$ and $B$.
• There is only one rule whose conditions are marked: the rule $A, B \rightarrow C$. So, we mark $C$.
• On the second iteration, we find three rules whose conditions are marked:
  \[ A, B \rightarrow C; \quad B, C \rightarrow A; \quad A, C \rightarrow B. \]
• Their conclusions are already marked; so we stop.
• $C$ is marked, so we can compute $C$.
• $C$ was obtained from a rule $A, B \rightarrow C$ that stems from $A + B + C = \pi$.
• So, we find $C$ from the equation $A + B + C = \pi$
• As for $a$, it is not marked, and so, cannot be computed.
9. Boolean Logic Interpretation

• We can define a boolean variable $X$ meaning “we can compute the quantity $x$”.

• Each rule $x, y \rightarrow z$ turns into a formula $X \land Y \rightarrow Z$.

• Original problem: can $y$ be computed?

• In logical terms: is $Y$ deducible from the rules-related formulas?

• Triangle example:
  
  – we have a knowledge base $A \land B \rightarrow C$; $B \land C \rightarrow A$;
  
  …; $A$; $B$, and

  – we want to know whether $C$ and $a$ follow from these formulas.

• Application: automatic program synthesis in space missions, e.g., in the NASA Cassini mission to Saturn.
10. Limitations of the Boolean Logic Approach

- **Situation:** we know two relations between the unknowns $y_1$ and $y_2$:
  
  $$y_1 + y_2 - 1 = 0 \text{ and } y_1 - y_2 - 2 = 0.$$ 

- From this system of 2 linear equations with 2 unknowns, we can determine both $y_1$ and $y_2$.

- However, the Boolean-logic approach does not work:
  - rules lead to formulas $Y_1 \rightarrow Y_2$ and $Y_2 \rightarrow Y_1$;
  - from these formulas, we cannot conclude $Y_1$.

- Therefore, we cannot conclude that $y_i$ are computable.

- It was shown that such examples can be handled
  - if we replace the original Boolean logic
  - with a more complex fuzzy-type logic.
11. Remaining Problem

- Triangle example: we know $A$, $B$, $a$, $b$, and $c$, and we want to compute $C$.

- First way: use the rule $A, B \rightarrow C$ corresponding to the relation $A + B + C = \pi$, and find $C$ as $\pi - A - B$.

- Also: we can use the sine rule $A, a, c \rightarrow C$ corr. to $\frac{a}{\sin(A)} = \frac{c}{\sin(C)}$, and compute $C = \arcsin\left(\frac{c \cdot \sin(A)}{a}\right)$.

- In such situations, it is desirable to come up with the fastest program – or, more generally, optimal program.

- We may need to take into account resources needed to measure the input characteristics $x_i$.

- In this paper, we show how fuzzy-type logic can help in this optimal program synthesis.
12. Optimal Program Synthesis: Formulation of the Problem

• **We know:**
  
  – rules $r$ of the type $a, b \rightarrow c$;
  
  – for each rule $r$, its *weight* $w(r)$ – the amount of resources needed to compute $c$ from $a$ and $b$.

• **Objective:** to select:
  
  – among all paths that lead to the desired quantity $y$,
  
  – the path with the shortest overall weight.

• **Comment:** to take into account resources needed for measuring each quantity $a$, we add rules of the type $0 \rightarrow a$. 
13. Logical Interpretation of Weights

- **Interpretation:** we can view the weights \( w(r) \) as degrees in the style of fuzzy logic:
  - if the derivation takes a long time,
  - this means that we should not be using this rule \( r \) unless it is absolutely necessary.

- **Difference:**
  - in the standard fuzzy logic, degrees are from the interval \([0, 1]\);
  - weights can be larger than 1: e.g., \( w(r) = 5 \text{ sec.} \)

- **Solution:** to get values \( \in [0, 1] \), we can normalize degrees, i.e., take \( \mu(r) = w(r)/W \) for an appropriate \( W \).

- **Result:** we can interpret \( 1 - \mu(r) \) as the “degree of confidence” in using this rule.
14. A Possible Solution: Exhaustive Search

- **Objective:** select the path with the smallest weight.
- **Toy example of a triangle:** we have few variables.
- **Conclusion:** we can simply try all possible paths.
- **Problem:** the number $N$ of possible paths grows exponentially with the number $n$ of variables $N \approx 2^n$.
- **In practical applications** (like space navigation), we have hundreds of variables.
- **Hence:** testing all $\approx 2^n$ paths will take too long.
- **For example:** for $n \approx 300$, $2^n$ computations require longer time that the lifetime of the Universe :-(
15. Simplest case: Rules of the Type $a \rightarrow b$

- **Simplest case**: all the rules have the form $a \rightarrow b$, i.e., only one input.
- This simplest case can be described by a *directed graph* in which:
  - nodes are variables, and
  - variables $a$ and $b$ are connected by an edge if there is a rule $a \rightarrow b$.
- Non-negative weights are now assigned to edges.
- In this case, estimating $y$ simply means that we have a path $0 \rightarrow a \rightarrow \ldots \rightarrow b \rightarrow y$.
- The optimal program corresponds to the *shortest path* from 0 to $y$.
- Efficient algorithms are known for computing shortest path in a graph.

- We want to find the length of the shortest path from the fixed node 0 to different nodes $y$.
- The shortest path cannot visit a node twice: otherwise, we could cut out the part between the two visits.
- Thus, on a graph with $n$ nodes, we only need to consider paths with $\leq n - 1$ edges.
- For each node $x$, let $d_k(x)$ denote the shortest of the paths from 0 to $x$ that have $\leq k$ edges ($\infty$ if none).
- Then, we have $d_0(0) = 0$ and $d_0(x) = \infty$ for all $x \neq 0$.
- For $k > 0$, the shortest path with $\leq k$ steps:
  - either has $\leq k - 1$ steps, hence length $d_{k-1}(x)$;
  - or spends $k - 1$ edges to get to some $y$; then, its length is $d_{k-1}(y) + w(y \rightarrow x)$. 
17. Shortest Path Algorithm (cont-d)

- Reminder: the shortest path with $\leq k$ steps:
  - either has $\leq k - 1$ steps, hence length $d_{k-1}(x)$;
  - or spends $k - 1$ edges to get to some $y$; then, its length is $d_{k-1}(y) + w(y \rightarrow x)$.

- Thus, the length $d_k(x)$ of the shortest path with $\leq k$ edges is equal to the smallest of these values:

$$d_k(x) = \min\left( d_{k-1}(x), \min_y (d_{k-1}(y) + w(y \rightarrow x)) \right).$$

- The length of the shortest path is $d_{n-1}(x)$.

- We can also find the actual shortest paths:
  - if the minimum of $d_k(x)$ is attained for
    $$d_{k-1}(y) + w(y \rightarrow x),$$
  - then the previous step should be the rule $y \rightarrow x$. 
18. Shortest Path Algorithm: Computation Times

- Reminder: for each $k \leq n - 1$ and each $x$, we compute
  \[ d_k(x) = \min \left( d_{k-1}(x), \min_y (d_{k-1}(y) + w(y \rightarrow x)) \right). \]

- For each $k$ and for each $x$, we need a linear time to compute this expression (by counting all $y$s).

- For each $k$, we need to repeat this computation for each of $n$ nodes $x$ – which requires $n \cdot n = O(n^2)$ time.

- We need to repeat these computations for
  \[ k = 1, \ldots, n - 1. \]

- So, the overall time is $(n - 1) \cdot O(n^2) = O(n^3)$.

- This is thus indeed a polynomial-time algorithm.
19. A Seemingly Natural Extension of This Idea to the General Case of Program Synthesis

- Let \( d_k(x) \) be the smallest amount of resources that we need to compute \( x \) by using \( \leq k \) rules.
- Similarly to the above, for \( k = 0 \), we have \( d_0(0) = 0 \) and \( d_0(x) = \infty \) for all \( x \neq 0 \).
- For \( k > 0 \), we have

\[
d_k(x) = \min \left( d_{k-1}(x), d'_k(x) \right),
\]

where

\[
d'_k(x) \overset{\text{def}}{=} \min_{a, \ldots, b \rightarrow x} \left( d_{k-1}(a) + \ldots + d_{k-1}(b) + w(a, \ldots, b \rightarrow x) \right),
\]

and the minimum is taken over all rules resulting in \( x \).
20. Counter-Example to the Seemingly Natural Approach

- **Example**: rules $0 \rightarrow a$, $a \rightarrow b$, $a \rightarrow c$, and $b, c \rightarrow d$, all with weight 1. Then:

- $d_0(0) = 0$, $d_0(a) = d_0(b) = d_0(c) = \infty$.
- $d_1(0) = 0$, $d_1(a) = 1$, $d_1(b) = d_1(c) = d_1(d) = \infty$.
- $d_2(0) = 0$, $d_2(a) = 1$, $d_2(b) = d_2(c) = 2$, $d_2(d) = \infty$.
- $d_3(0) = 0$, $d_3(a) = 1$, $d_3(b) = d_3(c) = 2$, $d_3(d) = 5$.

- After that, the values $d_k(x)$ do not change.
- We are thus tended to conclude that the length of the shortest path to $d$ is 5.
- However, we can compute $d$ by using only 4 resource units:

$$ 0 \rightarrow a; \quad a \rightarrow b; \quad a \rightarrow c; \quad b, c \rightarrow d. $$
21. Analysis of the Situation

- **Algorithm (reminder):** \( d_k(x) = \min (d_{k-1}(x), d'_k(x)) \),

\[
d'_k(x) \overset{\text{def}}{=} \min_{a',\ldots,b'\to x} \left( d_{k-1}(a') + \ldots + d_{k-1}(b') + w(a',\ldots,b' \to x) \right).
\]

- **Example:** rules \( 0 \to a \), \( a \to b \), \( a \to c \), and \( b, c \to d \), all with weight 1; we want to find \( d \).

- **We estimated:** \( d_3(d) = 5 \), but actually \( d(d) = 4 \).

- **Reason:** we added the costs of \( b \) and \( c \), but they have a common part: cost of measuring \( a \).

- **As a result:** we counted the resources needed to measure \( a \) twice:
  - once as part of estimating resources needed to compute \( b \), and
  - second time as part of estimating resources needed to compute \( c \).
22. Solution to the Problem: Idea

- Previously: for each $k$, we computed the values $w_k(a)$.
- New: compute $d_k(A)$, where $A = \{a, \ldots\}$ is a set.
- If all rules have $\leq 2$ inputs, we take $A$ s.t. $\#A \leq 2$.
- We then generate rules covering such sets.
- For example, we generate a new rule $A, \ldots, B \rightarrow X$ if every element $x \in X$ is:
  - either already in the inputs $x \in A \cup \ldots \cup B$,
  - or is covered by a rule $r_x$ of the type $C_x \rightarrow x$ with $C_x \subseteq A \cup \ldots \cup B$.
- The weight of the new rule is then defined as the sum of the weights of these original rules.
- Then, we apply the above algorithm to the nodes $A$. 
23. Example and Discussion

- **Example**: rules $0 \rightarrow a$, $a \rightarrow b$, $a \rightarrow c$, and $b, c \rightarrow d$, all with weight 1; we want to compute $d$.

- With pairs, we now have new rules:
  - $0 \rightarrow a$ (of weight 1),
  - $a \rightarrow \{b, c\}$ (of weight 2), and
  - $\{b, c\} \rightarrow d$ (of weight 1).

- Applying these rules one after another, we get the desired shortest path of weight 4.

- For each bound on the size $k$ of the set of inputs, the resulting algorithm takes $n^k$ steps.

- This time grows polynomially with $n$.

- However, it grows exponentially with $k$.

- This exponential growth is inevitable, since finding a shortest path in a hyper-graph is NP-hard.