Synthesizing Geometry Constructions

Sumit Gulwani
Microsoft Research
Redmond, WA, USA
sumitg@microsoft.com

Vijay Anand Korthikanti *
UIUC
Urbana Champaign, IL, USA
vkortho2@uiuc.edu

Ashish Tiwari †
SRI International
Menlo Park, CA, USA
tiwari@csl.sri.com

Abstract

In this paper, we study the problem of automatically solving ruler/compass based geometry construction problems. We first introduce a logic and a programming language for describing such constructions and then phrase the automation problem as a program synthesis problem. We then describe a new program synthesis technique based on three key insights: (i) reduction of symbolic reasoning to concrete reasoning (based on a deep theoretical result that reduces verification to random testing), (ii) extending the instruction set of the programming language with higher level primitives (representing basic constructions found in textbook chapters, inspired by how humans use their experience and knowledge gained from chapters to perform complicated constructions), and (iii) pruning the forward exhaustive search using a goal-directed heuristic (simulating backward reasoning performed by humans). Our tool can successfully synthesize constructions for various geometry problems picked up from high-school textbooks and examination papers in a reasonable amount of time. This opens up an amazing set of possibilities in the context of making classroom teaching interactive.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Program Synthesis; K.3.1 [*Computers and Education*]: Computer uses in Education

General Terms Algorithms, Theory

Keywords Program synthesis, Ruler-Compass geometry constructions, Abstraction, Forward and Backward Analysis

1. Introduction

Program Synthesis is the task of automatically synthesizing a program in some *underlying language* from a given *specification* using some *search technique* [12]. It has been used for a wide variety of applications targeted towards various classes of users.

- Discovery of new algorithms for *Algorithm Designers*: Bit-vector algorithms [14], Mutual exclusion algorithms [3, 21], Concurrent algorithms [34, 39], parameters of embedded systems [6, 37].
- General purpose programming assistance for *Software Developers*: Partial programming [4, 23, 33], Template based programming [35, 36], Automated debugging [40], Program understanding [18].
- Automating repetitive tasks for *End-users*: Programming by Demonstration systems [9, 26], Programming by Example systems [13, 15], shell scripts [25].

However, we feel that the most revolutionary application of program synthesis technology can be in K-12 education. The potential impact size of target population is simply mind boggling: *billion* students and teachers (compared to say *ten thousand* algorithmic designers, *million* software developers, and *hundred million* end-users).

Program synthesis technology can be applied to *automating constructions* for various domains in *high-school curriculum*. Examples of such constructions include (i) ruler/compass based geometric constructions in mathematics (ii) constructions of chemical compounds or biological proteins using (bio)chemical reactions under appropriate conditions governed by catalysts, temperature and pressure in chemistry and biology, (iii) constructions of electrical circuits using series and parallel composition of resistances and capacitances in physics, and so on.

One might wonder what is the connection between program synthesis and automating constructions. Programming languages allow us to formally describe these constructions. Take the example of geometric constructions that involve constructing a set of objects O with desired properties ϕ_2 from an initial set of objects I with certain properties ϕ_1 using a series of steps S, where each step involves using ruler or compass to create new objects. The objects I and O are like program variables and S is the program whose instruction set includes ruler/compass operations.[1] The formulas ϕ_1 and ϕ_2 play the role of precondition and postcondition respectively. Checking the correctness of a given geometric construction is like checking the validity of the Hoare triple $\langle \phi_1, S, \phi_2 \rangle$, which is a program verification problem. In fact, this verification problem has been studied intensively in the context of geometric constructions. Synthesizing the geometric construction S, given ϕ_1 and ϕ_2, is the program synthesis problem, which is what we address in this paper in the context of geometric constructions.

We focus on automating geometry constructions because geometry is regarded to be one of the most difficult as well as important subjects in high-school curriculum. Geometry education is supposed to help exercise logical abilities of the left-brain, visualization abilities of the right-brain, and hence enables students to make the two connect and work together as one. Geometric con-

* Work done while visiting SRI International.

† Research supported in part by NSF grants CSR-EHCS-0834810, CSR-0917398 and SHF:CSR-1017483.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

[1] In fact, a programming language for drawing such geometric objects is used to motivate freshman and non-major students to take up programming lang. course at UCSD [29].

structions have a close connection to axiomatic logic. The skills needed to figure out how to construct, say, a square without a protractor, are closely related to the thinking skills needed to prove theorems about squares [1]. The visual nature of geometry makes it initially more accessible than other parts of mathematics, such as algebra or number theory. "Construction can reinforce proof and lend visual clarity to many geometric relationships." [30] "They give the secondary school student, starved for a Piagetian concrete-operational experience, something tangible." [28].

One might also wonder why we insist on high-school curriculum as opposed to undergraduate/graduate curriculum for our success metric. We want to build tools that are scalable, predictive in their power and can have a real practical impact. Given scalability limitations of program synthesis techniques in terms of being able to automatically synthesize only relatively small snippets of code in a reasonable amount of time, high-school problem solving (where solution size is relatively small enough and are less involved compared to undergraduate/graduate problems) is an excellent fit.

We present a new program synthesis technique for automating geometry constructions. Our program synthesis technique is based on exhaustive search, but exploits three key ideas, inspired by human experience and intelligence, to make it scalable.

Idea 1: Reducing symbolic reasoning to concrete First, we reduce the problem of symbolic reasoning to concrete reasoning. We reduce the problem of "obtaining a construction that can transform input objects I with precondition ϕ_1 to output objects O with postcondition ϕ_2" to that of "obtaining a construction that can perform the transformation on a randomly chosen concrete model for I and O that satisfies constraints ϕ_1 and ϕ_2". In our implementation, we generate such a random model using a numerical (multivariate) function minimization procedure. The probabilistic soundness of this reduction relies on a deep theoretical result that is an extension of randomized polynomial identity testing theorem. (This result is analogous to reducing verification of a straight-line program to testing on a random input.) This reduction is critically important because symbolic reasoning would not scale - it would make our technique multiple orders of magnitude slower, and hence useless in an interactive setting. This reduction is inspired by how humans also work out geometric constructions by visually working out the construction on a randomly chosen configuration.

Idea 2: Using extended set of common constructions When humans typically solve a construction problem, they don't attempt to construct everything from first principles. They think in terms of some key higher level abstractions, where they exploit their knowledge of how to perform some key multi-step constructions. Our technique, which is based on exhaustive search, exploits this observation by working with an extended set of common constructions found in textbook chapters (Table 2) as opposed to simply working with a basic minimal set of constructions (Table 1). This can also be thought of as giving preference to certain combinations of basic construction steps, which is related to the notion of using priors in the machine learning community. This allows for transforming the search process with small width and large depth to one with large width and small depth.

Idea 3: Performing goal-directed search We prune our forward exhaustive search by applying a construction (from the extended set of constructions) only when the *goodness measure* function suggests that it may be useful to apply that construction. For example, the goodness measure function might suggest that if a construction results in a line L_1 that passes through a given output point P, then it may be useful to apply that construction (since we might be able to construct another line L_2 in the future that also passes through P, and hence intersection of L_1 and L_2 can yield desired point P). This is inspired by backward reasoning performed by humans in doing such constructions. This can also be thought of as pruning the forward search by integrating it with backward search.

Based on these ideas, we built a tool for automating geometry constructions. Our tool could solve most standard geometry construction problems in less than a second. Surprisingly, it could also solve some nontrivial problems whose solution is not immediately clear, such as, constructing a square whose (extended) sides pass through four given points. While our focus is on synthesizing geometry constructions, the basic concepts used here could be applicable in other domains as well.[2]

Another interesting application of our algorithm is in the context of dynamic geometry, or creating animations. Suppose we can find a model for declarative constraints using numerical methods in 1 second, and it takes 5 seconds to synthesize an equivalent construction for the declarative problem specification. The importance of having the construction (besides education purposes) is that running that construction will take micro-seconds. So, if the goal is to quickly re-compute coordinates of various points after assigning a new value to the free variables, running the construction will be faster by orders of magnitude compared to using numerical methods.

This paper makes the following key contributions.

- Using the specific example of geometry constructions, the paper points out the role that programming languages, logic, and program synthesis can play in the area of building automated tutoring systems, which is traditionally considered to be a subfield of AI.
- We present a novel search algorithm that combines three key ideas picked up from different research areas: property testing (theoretical computer science), higher-level abstractions (programming languages), and goal-directed reasoning (AI). We believe that the principles underlying our search algorithm are general enough to be applicable to other automated problem solving domains, and other program synthesis applications.
- We report on a successful experimental prototype thereby establishing the feasibility of building automated tutoring systems around the important high-school subject of ruler-compass based geometry constructions.

2. Overview of our Approach

Consider the problem: *Construct a triangle, given its base, a base angle and sum of the other two sides.* We wish to synthesize a program S that performs the above construction using ruler and compass instructions. Before we can discover S, we first need to formally state its inputs $\vec{I}$, output $\vec{O}$, its precondition $\phi_{\texttt{pre}}$ and its postcondition $\phi_{\texttt{post}}$.

The inputs $\vec{I}$ consist of a line segment (points p_1, p_2, and a line $L = \texttt{Line}(p_1, p_2)$) that defines the base of the desired triangle, a length r and an angle a.[3] The desired output $\vec{O}$ consists of a single point p.

The precondition $\phi_{\texttt{pre}}$ arises from the triangle inequality as

$$r > \texttt{Length}(p_1, p_2) \tag{1}$$

[2] The idea of representing knowledge as a library and then constructing new structures by composing elements of the library can be viewed as one way of interpreting (Turing award winner) Ed Feigenbaum's recent remarks in an interview in ACM Communications, where he calls out for having a way for computer to read books and learn to solve problems after that, as opposed to building domain specific tools [32].

[3] In our formalization of geometry constructions, we distinguish between the cases when we have just two points on a plane versus when we also have the line segment connecting those two points explicitly drawn on the plane.

The postcondition $\phi_{\texttt{post}}$ can be stated as,

$$\texttt{Angle}(p, p_1, p_2) = a \ \wedge \ \texttt{Length}(p, p_1) + \texttt{Length}(p, p_2) = r \quad (2)$$

After we have a formal description of the inputs $\vec{I}$, outputs $\vec{O}$, precondition $\phi_{\texttt{pre}}$ and postcondition $\phi_{\texttt{post}}$, we next find a (random) concrete input-output pair that is consistent with the pre/post specification (Idea 1). In other words, we need to find coordinates of the input points p_1, p_2, the distance r, the angle a, and the coordinates of the output point p such that the conditions in Equation 1 and Equation 2 are satisfied. These conditions are (multivariate) nonlinear constraints over the reals. We translate these nonlinear constraints into a multivariate nonlinear optimization problem (as described in Section 6.2) and then use a (numerical) multivariate nonlinear optimization routine to find a concrete input-output pair. During this process, we ensure that the concrete input-output pair is picked sufficiently randomly. Let us say that we find the following concrete input-output pair:

$$\begin{array}{rclcrcl} L & = & \texttt{Line}(\langle 81.62, 99.62\rangle, \langle 99.62, 83.62\rangle) & & & & \\ r & = & 88.07 & & a & = & 0.81 \text{ radians} \\ p & = & \langle 131.72, 103.59\rangle & & & & \end{array}$$

Let $(\vec{I}_c, \vec{O}_c)$ denote these concrete input-output objects. (Here all numerals are truncated to 2 digits after decimal, but the implementation uses a much higher precision).

Now we have a concrete input-output pair, $(\vec{I}_c, \vec{O}_c)$, for the problem. The geometry synthesis problem is also given an executable library of functions that should be used to synthesize S. For example, assume we have a library that implements the "ruler-compass" functions in Table 1 (the functions take concrete inputs and output concrete objects).

Our tool now searches for a program S *that works correctly on this one input-output pair* $(\vec{I}_c, \vec{O}_c)$. It searches for S by enumerating programs up to a certain length. However, to reduce the size of the search space, rather than using the basic library in Table 1, we use an extended library that additionally implements functions in Table 2 (Idea 2). Furthermore, we use a heuristic goodness metric to make the search for S goal-directed (Idea 3).

When searching for S, we generate several partial programs. These partial programs will, when given the input $\vec{I}_c$, generate different geometric objects like points, lines and circles - all of which are represented using points. Figure 1 shows all these intermediate points in two different settings. On the left, we show the points generated when we use the goodness metric to prune the search space, and on the right, we show the points generated when we do not use the goodness metric. It is clearly evident that when we do not use a goodness metric to prune the search space, then the approach does not scale.

Once we find a program S that works on input $\vec{I}_c$, we return it if it also works correctly on a second randomly sampled concrete input. The program synthesized by our tool for the above problem (using the extended library in Table 2) is:

```
Test10(p1,p2,L,r,a):
 L1 := ConstructLineGivenAngleLinePoint(L,a,p1);
 C1 := ConstructCircleGivenPointLength(p1,r);
 (p3,p4) := LineCircleIntersection(L1,C1);
 L2 := PerpendicularBisector2Points(p2,p3);
 p5 := LineLineIntersection(L1,L2);
 return p5;
```

The line `L1` is good because the output point p lies on it. The circle `C1` is good because it intersects an existing object (namely, line `L1`) at a good point `p3`: `p3` is good because it is equidistant from the output point p and the input point p_2. Similarly, we can establish goodness of all the other intermediate objects. Without using goodness, the search for S fails to terminate in allocated time. Fig. 1(right) shows the points constructed during a truncated search.

The above program was synthesized in 3.5 seconds. When rewritten in terms of the basic library, the above program expands to a program with 17 instructions. We remark that our implementation distinguishes between lines and rays, but we do not show this distinction here for simplicity.

3. Geometry Programming Language

We describe a programming language for geometric constructions. We will later synthesize programs in this programming language starting from a given specification.

Traditional (imperative) programming languages manipulate *values* of *variables*. These *values* are typically integers, floats, Booleans, or addresses (pointers). The entities that our programming language for geometry manipulates are *objects*, such as *points*, *lines* and *circles*. Specifically, our programming language for geometry has the following set of object types:

Point. A point is a primitive object in our language. It is represented using Cartesian coordinates. In this paper, we restrict ourselves to 2-dimensional geometry and a point is represented using its x- and y-coordinates that remain hidden.

Line. A line is represented by a pair of two distinct points that lie on it. The constructor $\texttt{Line}(p_1, p_2)$ returns a line object that is defined by the points p_1 and p_2.

Angle. An angle is a number in the range $[0, 2 * \pi)$. An angle is represented using three points. The function ExplodeAngle returns these three points.

Length. A length is a number in the range $[0, \infty)$. The constructor $\texttt{Length}(\texttt{p}_1, \texttt{p}_2)$ returns a length object denoting the distance between points p_1 and p_2.

Circle. A circle is represented as a pair of a point and a length. For a point p and length l, the constructor $\texttt{Circle}(p, l)$ returns the circle with center p and radius l. The radius of a circle is always a nonzero positive number.

Program variables are typed and are given one of the five types defined above. A *program* takes some objects as inputs, and using a predefined library `Lib` of functions $F_1, \ldots, F_k$, it constructs new objects and outputs one or more of these objects. Thus, a program takes as input $\vec{I}$ and uses (tuples of) temporary variables $\vec{\texttt{t}}_i$ to compute the output $\vec{O}$ as follows:

$$\begin{array}{l} \underline{\texttt{geoProgram}(\vec{I})\texttt{:}} \\ \quad \vec{\texttt{t}}_1 := F_{\pi_1}(\vec{V}_1); \\ \qquad \vdots \\ \quad \vec{\texttt{t}}_n := F_{\pi_n}(\vec{V}_n); \\ \quad \vec{O} := \vec{V}_{n+1}; \\ \quad \texttt{return } \vec{O}; \end{array}$$

where

- each variable in $\vec{V}_i$ is either an input variable from $\vec{I}$, or a temporary variable from $\vec{\texttt{t}}_j$ such that $j < i$, and
- $1 \leq \pi_i \leq k$.

3.1 The Expression Language

There is only one kind of expression in our geometry programming language, namely a function call $F_i(\vec{V})$, where F_i is a function from a given library. The library for "ruler-and-compass" constructions is shown in Table 1. It consists of functions for constructing lines, circles, and lengths from points, and functions for constructing points from line-line intersection, line-circle intersection, and

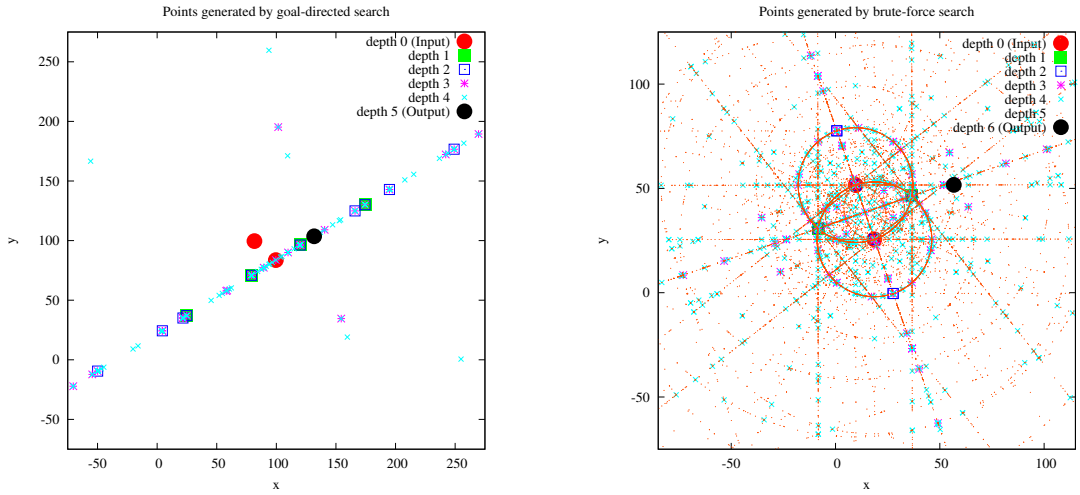

Figure 1. Points visited in a goal-directed search (left) and a brute-force search (right). The concrete inputs and output were picked independently in the two runs. The output point was not reached in the brute-force search and the search had to be truncated.

Function	Description
`L = Line(p1,p2)`	L is the line joining p1 and p2 (provided p1 $\neq$ p2).
`C = Circle(p,r)`	C is the circle with center p and radius r.
`r = Length(p1,p2)`	r is the length of the segment from p1 to p2 (provided p1 $\neq$ p2).
`p = LineLineXn(L1,L2)`	p is the point that lies at the intersection of L1 and L2 (provided L1, L2 are not parallel).
$\vec{p}$ `= LineCircleXn(L1,C1)`	$\vec{p}$ is the vector containing (1 or 2) points that lie on both L1, C1 (provided they intersect).
$\vec{p}$ `= CircleCircleXn(C1,C2)`	$\vec{p}$ is the vector containing (1 or 2) points that lie on both C1, C2 (provided they intersect).
$\vec{p}$ `= ExplodeAngle(a)`	$\vec{p}$ is the vector of (three) points that define angle a.

Table 1. Library functions in the geometry programming language: the type of $p1, p2, \ldots$ is Point, $\vec{p}$ is a vector of points, L is a line, C a circle, a an angle, and r a length.

circle-circle intersection. It also consists of a function to expose points that define a given angle. It is not difficult to observe that programs in our geometry programming language correspond directly to "ruler-and-compass" constructions on paper.

However, the library shown in Table 1 contains only the primitive "ruler-and-compass" operations. It can be extended by including new (derived) functions such as those in Table 2. We later show that such an extension enables synthesis of nontrivial geometry programs.

3.2 The Specification Language

The specification language is used to write the precondition and the postcondition of a geometry program.

A specification is given as a *conjunction* of atomic facts. An *atomic fact* is an equality or inequality between two arithmetic expressions. An arithmetic expression is built using the standard arithmetic operations, $+, -, *$, applied on numeral objects, the `Length` and `Angle` functions, and the functions:

- `distL(p,L)` returns distance between point p and line L.
- `slope(L)` returns the slope of line L.

We found that these functions were sufficient to describe almost all high-school geometry problems. For example, note that we can specify that "point p lies on line joining points p_1 and p_2" as an equality between two slopes,

$$\texttt{slope}(p, p_1) = \texttt{slope}(p_1, p_2),$$

and we can specify that $\texttt{Line}(p_1, p_2)$ and $\texttt{Line}(p_3, p_4)$ are perpendicular as an equality between arithmetic expressions:

$$\texttt{slope}(p_1, p_2) * \texttt{slope}(p_3, p_4) = -1$$

4. Problem Definition

Our goal is to synthesize programs in our geometry programming language given their specification in the form of precondition and postcondition. Here we will formally define the synthesis problem for the geometry domain.

The synthesis problem is given the specification of the desired program and also the executable code implementing a given library. The goal of the geometry synthesis problem is to discover a program in the geometry programming language that correctly implements the specification.

Formally, the geometry synthesis problem requires the user to provide the following:

- A *specification* $\langle \vec{I}, \vec{O}, \phi_{\texttt{pre}}(\vec{I}), \phi_{\texttt{post}}(\vec{I}, \vec{O}) \rangle$ of the desired program, which includes
  - a tuple of typed input variables $\vec{I}$ and output variables $\vec{O}$.
  - a formula $\phi_{\texttt{pre}}(\vec{I})$ that specifies the precondition and a formula $\phi_{\texttt{post}}(\vec{I}, \vec{O})$ that specifies the postcondition.
- A *library* of executable specifications

$$\texttt{Lib} := \{\langle \vec{I}_i, \vec{O}_i, F_i(\vec{I}_i) \rangle \mid i = 1, \ldots, k\},$$

where F_i is an executable implementation of the i-th library function.

The *geometry program synthesis problem* seeks to synthesize a geometry program S – a composition of the primitive operations from `Lib` – that implements the given specification; that is, the following correctness criterion holds:

$$\forall \vec{I}, \vec{O} : (\phi_{\texttt{pre}}(\vec{I}) \ \wedge \ \vec{O} = f_S(\vec{I})) \quad \Rightarrow \quad \phi_{\texttt{post}}(\vec{I}, \vec{O}) \quad (3)$$

where f_S is the function computed by the program S.

We note here that the problem definition above does not require a formal specification of the library functions, but only an executable specification. This is one important point that distinguishes our work from other work on synthesis; see also Section 7.

The synthesis procedure needs to find a program from the huge space of all possible programs that works correctly *for all* concrete inputs.

5. The Synthesis Procedure

In this section, we present our synthesis approach for automatically discovering geometry programs that meet some given specification.

As a first approach, one can perform an exhaustive search on the space of all possible programs (of some fixed length). In other words, one could enumerate all possible compositions of library functions and then verify if any one implements the given specification. This is, however, not feasible for two reasons. First, the number of possible programs (even of a fixed depth) is huge, and expensive to exhaustively enumerate. Second, verifying whether a program implements a given specification is a hard "geometry theorem proving" problem. Specifically, it involves checking the validity of a nonlinear formula in the theory of reals. While this theory is decidable [38], decision procedures are not practical in our setting since solving a single synthesis problem will involve making a large number of verification queries.

We use a new and different approach for synthesis that completely avoids all symbolic reasoning and significantly prunes the search space of all programs. Figure 2 presents the pseudocode of our synthesis procedure. The program `GeoSynth` takes two inputs: a specification $(\phi_{\texttt{pre}}, \phi_{\texttt{post}})$ of the desired program and a library `Lib` of available functions. It first generates a concrete input-output pair $(\vec{I}_c, \vec{O}_c)$ that satisfies the given specification. It then calls the recursive function `GeoSynthRec` with the arguments $\vec{I}_c, \vec{O}_c$, `Lib` and an empty program P. If the concrete output objects $\vec{O}_c$ are already contained in the available objects $\vec{I}_c$, then there is nothing to synthesize and the function `GeoSynthRec` returns the program P (after checking that it works on a second concrete input-output pair). If not, then the synthesis procedure generates new objects from the available objects and adds them to the set of available objects if they are new and *good*, and recursively calls itself.

The procedure in Figure 2 for synthesis completely avoids all symbolic reasoning and significantly prunes the search space of all programs. It achieves this using three key ideas:

Concretization: From Symbolic to Numeric We discover the program by finding a program that works for a concrete input-output pair. By using a concrete input-output pair, we avoid doing any symbolic reasoning on nonlinear constraints. However, we still need to solve symbolic nonlinear constraints to generate a concrete input-output pair. This we do by using numerical techniques for nonlinear optimization.

Acceleration: Using Extended Library While searching for the correct program, we use a library of *high-level functions*. Each high-level function can be implemented using the basic functions in the given library. A high-level function performs a frequently-used subtask. A few calls to high-level functions often suffice to achieve what many calls to basic functions can achieve. This helps in reducing the search space of programs.

```
GeoSynth(φspec, Lib):
// Input φspec := (φpre, φpost) is the specification
// Input Lib:= ⟨(F_i)_{i=1,2,...}⟩, where F_i implements
               the i-th library function
// Output: A program or "Failure"
Ī_c := Random concrete objects s.t. φpre(Ī_c) holds
Ō_c := Concrete objects s.t. φpost(Ī_c, Ō_c) holds
return GeoSynthRec(Ī_c, Ō_c, Lib, ϵ, φspec);

GeoSynthRec(Ī_c, Ō_c, Lib, P, φspec):
// Input Ī_c, concrete objects we have constructed
// Input Ō_c, concrete objects we wish to construct
// Input P, the program (so far)
if (Ō_c are contained in Ī_c) return(Verify(P, φspec));
forall functions F_i ∈ Lib {
  forall possible choices of arguments args for F_i
         picked from Ī_c {
    newObj := F_i(args);
    if (newObj is Good and
        newObj is not already present in Ī_c) {
      newP := (P; t⃗ := F_i(args));
      newI := Ī_c ∪ {newObj};
      result := GeoSynthRec(newI,Ō_c,Lib,newP,φspec);
      if (result ≠ "Failure") return(result);
    }
  }
}
return "Failure"

Verify(P, φspec):
// Check if P works correctly on a second input
Ī_c := Random concrete objects s.t. φpre(Ī_c) holds
Ō_c := f_P(Ī_c);
if (φpost(Ī_c, Ō_c) == true) return(P);
else return "Failure";
```

Figure 2. GeoSynthesizer's Synthesis procedure.

Goal-Directed Search: Goodness Measure A naive search for the correct program based on enumerating all programs is blind to the goal. We can make the search goal-directed by developing a measure for the progress toward the goal. For any newly created object (using the basic/high-level library), the goodness function returns a measure of its goodness with respect to (achieving) the goal.

We provide more details on these three techniques below.

5.1 Concretization: Symbolic to Numeric

Recall that the goal is to *synthesize* a program P that works for all inputs; that is, it meets the given correctness criterion in Formula 3. The problem of even *verifying* that a given program satisfies the given correctness criterion is not easy. First, the set of all possible inputs is unbounded. The traditional solution is to reason about one arbitrary *symbolic* input. Formula 3 is a nonlinear formula in the theory of reals. For example, the atomic formula `Length(p1,p2)=Length(p1,p3)`, when expanded using coordinates (x_i, y_i) for `pi`, takes the form

$$(x_2 - x_1)^2 + (y_2 - y_1)^2 = (x_3 - x_1)^2 + (y_3 - y_1)^2$$

which is a quadratic formula over 6 real variables. Nonlinear formulas are hard to decide symbolically. There are some symbolic

geometry theorem proving tools that can automatically prove Formula 3, but in a synthesis procedure, one requires to check Formula 3 not once, but several times, while searching for the correct P. Hence, symbolic techniques are unlikely to scale for synthesis.

`GeoSynth` avoids all symbolic manipulation by synthesizing programs that work for some *concrete* input-output pairs. `GeoSynth` assumes that the library functions are provided as executable code that compute the concrete output objects given concrete input objects. From the descriptions of the functions in Table 1, the implementation should be immediately clear. For example, the function $\texttt{LineLineXn}(L1, L2)$ computes the x- and y-coordinates of the unique point, if any, that lies on the intersection of concrete lines $L1 = \texttt{Line}(p1, p2)$ and $L2 = \texttt{Line}(p3, p4)$ as follows:

$$\begin{aligned} x &= [(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_1 - x_2)(x_3y_4 - y_3x_4)]/D \\ y &= [(x_1y_2 - y_1x_2)(y_3 - y_4) - (y_1 - y_2)(x_3y_4 - y_3x_4)]/D \end{aligned}$$

where $D = (x_1-x_2)(y_3-y_4)-(y_1-y_2)(x_3-x_4)$, $p1 = (x_1, y_1)$, $p2 = (x_2, y_2)$, $p3 = (x_3, y_3)$, and $p4 = (x_4, y_4)$. Similarly, we can get algebraic expression that implement the other functions. The functions are implemented using floating point representation for the reals.

By using executable code for the library functions, `GeoSynth` synthesizes a program that works for one concrete input. How do we argue that the synthesized program also works for all other inputs? The key observation here is that if a geometry construction is correct for a concrete input that is randomly chosen from the space of all valid inputs (inputs that satisfy the precondition), then, with very high probability, it is correct for all valid inputs. The only assumption required is that the postcondition $\phi_{\texttt{post}}(\vec{I}, \vec{O})$ be such that for any concrete input $\vec{I}_c$, there is at most one (more generally, a constant fraction of the input space) output $\vec{O}_c$ such that $\phi_{\texttt{post}}(\vec{I}_c, \vec{O}_c)$ holds.

THEOREM 1. *Let P be a geometry program constructed using the library* `Lib` *from Table 1 and let f_P denote the function P computes. Let $(\phi_{\texttt{pre}}(\vec{I}), \phi_{\texttt{post}}(\vec{I}, \vec{O}))$ be a specification, where the number of inputs, $|\vec{I}|$, is n. Let $\vec{I}_c$ be a concrete input point obtained by sampling uniformly at random the subspace $\{\vec{c} \in \mathbb{R}^n \mid \phi_{\texttt{pre}}(\vec{c})\}$ that represents all non-degenerate inputs. If $\vec{O}_c = f_P(\vec{I}_c)$ and if $\phi_{\texttt{post}}(\vec{I}_c, \vec{O}_c)$ also holds, then with probability one it is the case that*

$$\forall \vec{I}, \vec{O} : \ (\phi_{\texttt{pre}}(\vec{I}) \wedge \vec{O} = f_P(\vec{I})) \quad \Rightarrow \quad \phi_{\texttt{post}}(\vec{I}, \vec{O})$$

PROOF: (Sketch) Assume the specification is deterministic so that there is a function f_{spec} that computes, for any given valid (non-degenerate) input, the output that is consistent with the specification $\phi_{\texttt{post}}$; that is, f_{spec} has the property that

$$\forall \vec{I}_c : \phi_{\texttt{pre}}(\vec{I}_c) \Rightarrow \phi_{\texttt{post}}(\vec{I}_c, f_{spec}(\vec{I}_c))$$

Consider the functions f_P and f_{spec}. It is well-known property of ruler-and-compass constructions that these two functions can be described using compositions of the arithmetic $+, -, *, /$ operations and the square-root operation, and hence we can argue that they are analytic over the domain of valid (non-degenerate) inputs (analytic with respect to each variable). Now consider the function $f_P - f_{spec}$. It is analytic too. Hence, unless it is identically zero, the space of zeros of this function has measure zero in the space of all inputs. Hence, for a random (valid non-degenerate) input $\vec{I}_c$, since we have $f_P(\vec{I}_c) - f_{spec}(\vec{I}_c) = 0$, it follows that, with probability one, $f_P - f_{spec}$ is identically zero and hence, the claim holds. □

The theorem is a generalization of a similar result for polynomial identity testing [31]. Similar results have been folklore in geometry theorem proving where it has been noted that geometry theorems can be proved by testing them on one random concrete instance [43]. The "probability one" claim in Theorem 1 assumes that exact computation on the reals is performed and we sample the real space. However, this is not possible and the implementation only uses finite precision floating point representation and samples over floating point numbers. These issues will slightly increase the probability of our procedure giving wrong answers, but, as one would expect, the probability will continue to remain very low.

Theorem 1 states that the probability of returning a specific wrong program P is very low. But the space of programs is large, and these probabilities could add up so that the probability of returning *some* wrong program from the space of all programs could become substantial. Hence, `GeoSynth` uses one concrete input-output pair to synthesize the program, and then another randomly sampled input-output pair to check it again (see the `Verify` function in the pseudocode). Therefore, `GeoSynth` can return a wrong answer only if the specific program P (that is synthesized using one concrete input-output pair) does not match the specification on a randomly sampled input. By Theorem 1, we know this is very low, and this establishes the probabilistic soundness of `GeoSynth`. In fact, in our experiments (reported in Section 6), the `Verify` function never returned "`Failure`". If `Verify` fails, note that the pseudocode continues its search for the correct program.

5.2 Acceleration: Extended Library

Discovering a geometry program that works correctly on even a single concrete input is not easy. This is because the search space of all possible programs is huge. We need ways to efficiently search this large state space. We use ideas from program analysis – forward and backward analysis – to significantly prune the search space. We argue that this is also the way that humans perform the search.

Consider the size of the search space of all programs. If the library has M functions that take, say, two arguments each, then the state space of all programs of length N that can be constructed using the library functions is $O((M(N + |\vec{I}|)^2)^N)$. This is super-exponential in N. This shows that we can not search for very long programs. Unfortunately, even for performing simple tasks, geometry programs P that are built using the library shown in Table 1 can be very long.

`GeoSynth` works on an extended library. Apart from the primitive functions listed in Table 1, the extended library also contains the functions listed in Table 2. Each new library function can be implemented using the primitive functions. Using the extended library, however, we can find much shorter programs P that meet the given specification. Hence, we can discover the correct P by searching for *shorter programs using the extended library*, rather than searching for *long programs using the primitive library*.

Table 2 lists the new functions in the extended library. For example, consider the function `PerpendicularBisector2Points`. Constructing the perpendicular bisector of two points will require at least three steps using the functions provided in the primitive library in Table 1. However, it is a commonly used "subroutine" when performing geometry constructions and it is useful to add such a function in the extended library, as it will likely reduce the length of the program P we are trying to discover by at least three.

The use of extended library for discovering the correct program matches the way humans synthesize programs. Rather than working with low-level primitives, complex systems are built by composing higher-level components. In geometry programs, the extended library contains high-level constructions that encode the *knowledge* the student is taught to solve more complex geometry

Function Name	Description
L = PerpendicularBisector2Points(p1,p2)	L is the perpendicular bisector of line joining p1 and p2.
p = MirrorPointLine(p1,L)	p is the reflection of p1 about line L.
$\vec{C}$ = CircleGivenChordAngle(L,a)	$\vec{C}$ is a vector of (1 or 2) circles C s.t. L is a chord of C subtending angle a.
L = ConstructLineGivenAngleLinePoint(L1,a,p)	L is at an angle a with L1 at point p (on L1).
C = ConcentricCircle(C1,r)	C is concentric to C1 and at distance r away from it.
L = PerpendicularToLineThruPoint(p,L1)	L is perpendicular to L1 and passes through p.
$\vec{L}$ = AngularBisectorLines(L1,L2)	$\vec{L}$ is the tuple of (two) lines that are angular bisectors of L1 and L2.
p = MidpointGiven2points(p1,p2)	p is the midpoint between p1 and p2.
$\vec{L}$ = TangentPointToCircle(p,C)	$\vec{L}$ is the vector of (two) lines that are tangent to C and pass through p.
L = ParallelLine(p,L1)	L is the line parallel to L1 and passing through p.
$\vec{L}$ = ParallelLineGivenLength(L1,r)	$\vec{L}$ is the vector of (two) lines that are parallel to L1 and distance r away from it.

Table 2. Extended Library

construction exercises. In fact, `GeoSynth` takes the library as an input, and hence, as the student learns more concepts, the library can be extended too, which will then enable `GeoSynth` to synthesize more complex geometry constructions.

In the experimental results in Section 6, we show the impact of using extended library by comparing it against the use of primitive library.

The use of extended library also has its analogue in the field of program analysis. It corresponds to using forward search (forward propagation) to see what search space is reachable in a few steps. The technique of *acceleration* in program analysis [5] explicitly computes high-level (multi-step transition) functions that are compositions of low-level (one-step transition) functions.

Besides efficiency, the idea of using library functions may provide another important benefit. By changing the library functions, the tool can be forced to be find constructions that use a particular primitive construction step. It is fairly common that exercises at the end of textbook chapters are framed so that their solution requires students to use the concepts introduced in that chapter. Using our tool, the same effect can be achieved. Teachers could re-inforce the learning of constructions taught in a certain chapter by just manipulating the library.

5.3 Goal-Directed Search: Goodness Measure

Introduction of an extended library reduces the depth of the programs that we need to search for finding the correct program. Despite this reduction, exhaustive enumeration of all possible programs remains infeasible. We use the third key idea – using backward analysis – to enable a more goal-directed search for the correct program.

`GeoSynth` performs a goal-directed search for the correct program by using a heuristic *goodness measure* that determines if an intermediate object could be useful in finally constructing the output objects. How to compute the goodness measure? Ideally, an intermediate object is good if an output object can be constructed using the intermediate object and the other existing objects. How can we efficiently check an intermediate object is good without continuing with the forward search? We can do so by performing some backward analysis starting from the output objects.

Performing exact backward analysis is challenging. Consider, for example, the case when the output object is a point p. Now consider one of the library functions, for example, the function `CircleCircleXn`(C_1, C_2). There are infinitely many circles (C_1, C_2) whose intersection could give p, so backward analysis would involve representing all these cases. However, if we fix C_1 to be an existing circle and we also fix the center of C_2 to be an existing point, then this fixes the circle C_2. This way of performing *partial backward analysis* can make backward analysis feasible.

Rather than explicitly performing partial backward analysis, `GeoSynth` directly tests if an intermediate object is backward reachable from the goal using multiple partial backward steps. These tests are coded using a goodness function. Specifically, the `Good` function takes as argument the set `currObjs` of objects we have already constructed, the set $\vec{O}$ of objects we need to construct, the object $\vec{\mathtt{t}}$ whose goodness we are trying to evaluate, and the library `Lib` of available functions. `Good` returns a Boolean answer – indicating if $\vec{\mathtt{t}}$ is possibly useful for eventually constructing $\vec{O}$. Depending on the type of the object $\vec{\mathtt{t}}$, `Good` performs different checks.

- A point $\vec{\mathtt{t}}$ is good if either
  - $\vec{\mathtt{t}}$ is an output point in $\vec{O}$.
  - $\vec{\mathtt{t}}$ is on an output line or circle.
  - $\vec{\mathtt{t}}$ lies on a line joining an existing point in `currObjs` with an output point in $\vec{O}$.
  - $\vec{\mathtt{t}}$ lies on a circle defined by a combination of some existing point(s) and output point(s).
- A line $\vec{\mathtt{t}}$:= `Line`(p_1, p_2) is good if either
  - $\vec{\mathtt{t}}$ is an output line in $\vec{O}$.
  - an output point in $\vec{O}$ lies on $\vec{\mathtt{t}}$.
  - $\vec{\mathtt{t}}$ is parallel or perpendicular to a line defined by output objects $\vec{O}$ and existing objects `currObjs`.
  - the intersection of the line $\vec{\mathtt{t}}$ with an existing line or circle object in `currObjs` gives a good point.
- A circle $\vec{\mathtt{t}}$:= `Circle`(p, r) is good if either
  - $\vec{\mathtt{t}}$ is an output circle in $\vec{O}$.
  - some output point in $\vec{O}$ lies on $\vec{\mathtt{t}}$ or is the center of $\vec{\mathtt{t}}$.
  - the intersection of the circle $\vec{\mathtt{t}}$ with an existing line or circle object in `currObjs` gives a good point.

These rules for goodness clearly measure the progress $\vec{\mathtt{t}}$ makes toward the goal $\vec{O}$. The rules essentially test if $\vec{\mathtt{t}}$ lies in the set of objects obtained from the goal $\vec{O}$ objects using a few selected partial backward steps.

The extended library encodes the facts a student is taught in a chapter, and the goodness function attempts to encode the intuition (search strategy) the student is expected to apply (in conjunction with the facts in the extended library) to solve the exercises at the end of the chapter.

`GeoSynth` uses the function `Good` to prune its search space. In the pseudocode in Figure 2, `Good` is used in the condition guarding the recursive call. A newly created object $\vec{\mathtt{t}}$ is added to the set of currently available objects only if it is `Good`. `GeoSynth` allows the user to turn-off goodness checking. In Section 6 we will compare

1. Find the circumcenter of a triangle.
2. Find the incenter of a triangle.
3. Find the orthocenter of a triangle.
4. Construct a regular hexagon inside a circle.
5. Construct length a+b given lengths a and b.
6. Construct a triangle given length of its 3 sides.
7. Construct a triangle given two sides and an included angle.
8. Construct a triangle given two angles and an included side.
9. Construct a $\triangle$ given two sides and the angle opposite one.
10. Construct a triangle, given its base, a base angle and sum of other two sides.
11. Construct a triangle, given its base, a base angle and difference of other two sides.
12. Construct a $\triangle$, given its perimeter and its two base angles.
13. Draw a pair of tangents to a circle that make an angle of 60° with each other.
14. Construct a triangle given a side and two altitudes.
15. Construct a triangle given one angle, the side opposite this angle, and the length of altitude to that side.
16. Construct a $\bigcirc$ that is inscribed in a given quadrant of a $\bigcirc$.
17. Given point A, line L not passing through A, and point B on L, construct a $\bigcirc$ passing through A that is tangent to L at B.
18. Given points A and B on the same side of a line L, find point C on L such that AC and BC make the same angle with L
19. Find the centroid of a triangle.
20. Given non-parallel lines L1 and L2 and a radius r, construct a circle of radius r that is tangent to both L1 and L2.
21. Draw arcs of radius r that are tangent to a given line and to a given circle.
22. Draw arcs of radius r that are tangent to two given circles.
23. Construct a square whose extended sides pass through 4 given points.
24. Construct a right $\triangle$ given one acute angle and sum of legs.
25. Given a circle and points D and E in its interior, construct an inscribed right triangle such that one leg contains D and other leg contains E.

Table 3. Informal description (in English) of the 25 benchmarks used to report results in this paper.

the effect of turning on-and-off the goodness check when synthesizing geometry programs.

6. Experimental Results

We implemented `GeoSynth` and tested it on a variety of high-school geometry construction exercises obtained from books and examination papers. Table 4 reports details of the runs of `GeoSynth` on selected 25 examples. As Column 4 indicates, `GeoSynth` successfully solved all the problems in a few seconds: 18 problems were solved in less than a second and only 3 problems required more than 10 seconds. We experimentally evaluated the efficacy of our two key ideas – using an extended library (E vs B) and using goal-directed search (G vs N) – by varying the inputs and command-line flags of `GeoSynth`. Specifically, Table 4 provides values of the following functions for the 25 examples:

- T_EG: time when using **E**xtended library and **G**oodness
- T_EN: time when using **E**xtended library and **N**o goodness
- T_BG: time when using **B**asic library and **G**oodness
- T_BN: time when using **B**asic library and **N**o goodness

6.1 Benchmarks

`GeoSynth` works off a fairly intuitive front-end language for writing the specification formulas describing the desired geometry construction:

`GIVEN`	$\langle$ list of input objects $\vec{I}$ $\rangle$
`that satisfy`	$\phi_{\texttt{pre}}(\vec{I})$
`CONSTRUCT`	$\langle$ list of output objects $\vec{O}$ $\rangle$
`that satisfy`	$\phi_{\texttt{post}}(\vec{I}, \vec{O})$

The formula $\phi_{\texttt{pre}}(\vec{I})$ specifies the constraints on the input objects, and the formula $\phi_{\texttt{post}}(\vec{I}, \vec{O})$ specifies the relationship between the input and output objects. The language of the formulas was described in Section 3.2.

In the selected benchmarks used for experimental study, Test1–Test9 and Test19 are standard problems in geometry, Test10–Test13 were taken from a Mathematics textbook for high-school students[4], and the rest were taken from course webpages and other websites devoted to such problems[5]. An informal description of the 25 examples is given in Table 3.

As an example of the specification of a benchmark in the above front-end language, consider our running example: *Construct a triangle given its base, base angle, and the sum of its other two sides.* This is presented to `GeoSynth` as:

```
GIVEN  Segment(p1,p2), Length(p3,p4), Angle(p5,p6,p7)
that satisfy Length(p3,p4) > Length(p1,p2)
CONSTRUCT   Point(p)
that satisfy
     Length(p,p1) + Length(p,p2) = Length(p3,p4) and
     Angle(p,p1,p2) = Angle(p5,p6,p7)
```

We note that the English text specification of a geometry construction problem is often ambiguous. In the example above, the English specification does not tell us how the sum of the two side is given or how the angle is given. The formal specification needs to eliminate this ambiguity. In the example above, the formal specification has assumed that the sum of the two sides is given as the distance between two new points $p3$ and $p4$, as opposed to, say, by point(s) that lies on the line extending the triangle's base.

6.2 Multivariate Optimizers

`GeoSynth` finds a (random) concrete input-output pair that satisfies the given precondition and postcondition using a numerical multivariate optimizer. `GeoSynth` currently uses an off-the-shelf implementation of such a function.

Recall that the precondition and the postcondition is a conjunction of nonlinear constraints. These constraints are difficult to solve symbolically. We instead use numerical techniques to get a feasible solution. First, we transform all inequalities, say $p > 0$, where p is a multivariate polynomial, to an equality constraint, namely $p - u^2 = 0$, where u is a new real-valued variable. Next, we transform a conjunction of equality constraints, say $p = 0 \wedge q = 0$, into a function minimization problem: `minimize` $(p^2 + q^2)$. The minimum value is zero iff the nonlinear constraints have a solution. Thus, using a multivariate function minimizer, `GeoSynth` finds a concrete input-output $(\vec{I_c}, \vec{O_c})$ that satisfies the given specification.

Recall that we also have to ensure that the concrete input-output pair is chosen at random. We achieve this by using heuristics to partition the set of unknowns into independent and dependent variables. Then, we randomly pick concrete values for the independent variables and fix them. We use the multivariate optimizer to find

[4] `http://ncertbooks.prashanthellina.com/class_9.Mathematics.Mathematics/chap-11\%20(02-12-2005).pdf`

[5] Test14–Test15 from `http://www-math.cudenver.edu/~wcherowi/courses/m3210/lecchap5.pdf`, Test16–Test18, Test20–Test22 from `www.geometer.org/mathcircles/construct.pdf`, Test23 from `http://puhep1.princeton.edu/~mcdonald/examples/4point.pdf` and Test24–Test25 from `http://www.misterhoffman.com/downloads/SpringProblemSets.pdf`

Test	Extended Library				Basic Library				Numerical Solving	
	LoCE	d,w	Time(T_EG)	T_EN	LoCB	#Makes	T_BG	T_BN	Mean	Min.
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)
Test1	3	2,2	0.131	0.147	7	6	NoSol	0.146	0.24	0.004
Test2	3	2,2	0.194	3.290	21	8	NoSol	Timeout	0.61	0.003
Test3	3	2,2	0.331	2.569	13	8	NoSol	Timeout	0.75	0.009
Test4	2	2,1	0.095	0.078	4	3	0.114	1.010	0.04	0.005
Test5	3	3,1	0.123	0.180	3	1	0.101	0.960	0.04	0.009
Test6	4	3,2	0.132	3.018	4	2	0.122	0.134	0.04	0.003
Test7	4	3,2	0.179	3.074	13	5	NoSol	Timeout	0.03	0.004
Test8	3	2,2	0.144	0.193	21	8	NoSol	Timeout	0.03	0.004
Test9	4	3,2	0.176	3.357	13	5	NoSol	Timeout	0.15	0.006
Test10	6	6,1	3.149	Timeout	17	8	NoSol	Timeout	0.28	0.006
Test11	6	6,1	3.550	Timeout	17	8	NoSol	Timeout	0.06	0.019
Test12	7	5,2	63.028	Timeout	45	19	NoSol	Timeout	3.91	0.555
Test13	3	3,1	0.121	0.093	4	2	0.118	0.100	1.84	0.157
Test14	5	4,2	13.684	44.198	22	14	NoSol	Timeout	0.11	0.019
Test15	4	3,2	0.150	4.185	32	20	NoSol	Timeout	1.54	0.030
Test16	6	6,1	82.613	Timeout	26	11	NoSol	Timeout	0.06	0.011
Test17	3	2,1	0.128	0.390	10	7	NoSol	Timeout	0.07	0.010
Test18	3	3,1	0.139	1.966	8	5	NoSol	Timeout	1.35	0.021
Test19	5	3,2	0.203	1.902	13	10	NoSol	Timeout	0.16	0.029
Test20	3	2,2	0.189	2.528	25	13	NoSol	Timeout	0.90	0.002
Test21	4	3,2	0.148	570.550	23	12	NoSol	Timeout	0.03	0.004
Test22	4	3,2	0.130	211.220	14	6	NoSol	Timeout	0.04	0.004
Test23	13	8,2	1.929	Timeout	25	17	NoSol	Timeout	20.35	2.616
Test24	6	6,1	2.301	Timeout	29	13	NoSol	Timeout	0.33	0.086
Test25	4	4,1	0.125	212.359	8	5	NoSol	Timeout	0.27	0.044

Table 4. Experimental results for 25 construction exercises. Column 2–5 report results when using the extended library: Column 2 is the length of program constructed; Column 3 gives the depth(d) and width(w) of this program; Column 4-5 give the time (in sec.) taken to synthesize the program when using goodness (Col. 4) and when not using goodness (Col. 5). Column 6–9 report results when using the basic library: Column 6 is the length of program; Column 7 is the number of *constructor calls* in this program; Column 8-9 give the time (in sec.) for synthesizing the program with (Col. 8) and without (Col. 9) using goodness. Column 10–11 report time taken for finding one concrete input-output pair using nonlinear optimization: Column 10 is the mean and Column 11 is the minimum (both in sec.) of 15 runs. NoSol indicates 'procedure terminated but failed to find a solution'. Timeout indicates 'procedure did not terminate in 10 min.'.

values only for the dependent variables, *having randomly fixed the values for the independent variables*.

To illustrate the approach, consider the precondition and postcondition from the example in Section 2.

$$r > \texttt{Length}(p_1, p_2)$$
$$\texttt{Angle}(p, p_1, p_2) = a$$
$$\texttt{Length}(p, p_1) + \texttt{Length}(p, p_2) = r$$

Let x_1, y_1 (respectively x_2, y_2 and x, y) denote the coordinates of point p_1 (respectively point p_2 and point p). Thus, the above constraint has 8 variables. We identify x_1, y_1, x_2, y_2, a as the independent variables and we pick values for them randomly. We determine the values for the other three variables, x, y and r, by solving the above constraint. We find x, y, r by minimizing the following nonlinear function over these three variables and the auxiliary variable u:

$$(r - \texttt{Length}(p_1, p_2) - u^2)^2 + (\texttt{Angle}(p, p_1, p_2) - a)^2 + (\texttt{Length}(p, p_1) + \texttt{Length}(p, p_2) - r)^2$$

The time taken by the minimization routine varies across different runs, and hence, in Table 4, we report the average (Column 10) and minimum (Column 11) time spent on finding a concrete input-output pair for each benchmark. (We performed 15 different runs for this purpose.) The time spent is proportional to the complexity of the postcondition constraint; in particular, the number of output objects. (Test23 seeks to construct a square, and hence the output has 4 points, which is 8 variables.)

6.3 Extended Library vs. Basic Library

We compared the performance of `GeoSynth` when it was given the basic library with its performance when it was given the extended library. There are two cases to consider depending on whether or not we use the "goodness" heuristic.

First, consider the case when we do not use the "goodness" heuristic for pruning the search space. When using the basic library, `GeoSynth` fails to find the correct program in 10 minutes of allocated time (Col. 9). In contrast, when using the extended library, `GeoSynth` successfully finds the correct solution in 19 out of the 25 examples (Col. 5).

Second, consider the case when we do use the "goodness" heuristic. When it is given the basic library, `GeoSynth` terminates without finding the solution in 21 examples (Col. 8). This is because the "goodness" measure (incorrectly) pruned the branches that had solutions. Note that "goodness" heuristic performs only a few backward steps, and the objects created by the basic library functions fail to be "good". In contrast, the extended library can quickly (in a few steps) create "good" objects. As shown in Col. 4, when using extended library, all examples were successfully solved.

Why is using the extended library better? The reason is that when we use the extended library, we find small correct solutions.

Comparing Column 2 with Column 6, we note program lengths (lines of code) varied between 2 and 13 for the 25 benchmarks when using the extended library (Col. 2), whereas its range was 3–45 for basic library (Col. 6). Smaller programs translated to smaller search space and hence, better timing, for synthesizing the programs.

6.4 Goodness vs. No Goodness

Let us now compare the effect of using goodness. Without goodness, `GeoSynth` performs a brute-force search, whereas with goodness, it performs a goal-directed search.

There are two cases depending on whether we use the basic library or the extended library. The case when we use the basic library is uninteresting: with goodness, the search is pruned too aggressively (Col 8), without goodness, the search explodes and seldom terminates (Col 9). When we use the extended library, the use of goodness (Col. 4) significantly improves the runtime of `GeoSynth` over the case when we do not use goodness (Col. 5). The reason is clear: goal-directed search explores much fewer paths than an exhaustive search (see Figure 1).

6.5 Variations in T_EG

Looking at Column 4 of Table 4, we notice that `GeoSynth` solves 18 of the examples in less than a second, and takes more than 10 seconds on only 3 examples. Without goodness (Col. 5), `GeoSynth` fails to terminate on 6 examples. To understand the reason for this variation, we also provide the depth (d) and width (w) of the solution in Column 3. (Intuitively, whereas lines of code (LoC) is the sequential complexity, depth (d) is the parallel complexity, and width (w) is the number of parallel processors required.) The ability to find the correct solution when performing exhaustive search (Col. 5) is directly correlated with the depth of the solution. Exhaustive search fails when depth of the solution is 5 or higher. However, the depth does not explain the variation in Column 4. Certain property of Test12, Test14 and Test16 interferes with the goodness measure making it less effective in pruning the search space.

7. Related Work

Program Synthesis using Brute-force Search [12] provides a good survey of various program synthesis techniques: brute-force search, logical reasoning, probabilistic inference, and version-space algebras. Our algorithm is closely related to the brute-force search strategy. There have been few success stories of using brute-force search for program synthesis. It has been used to discover new algorithms (mutual-exclusion algorithms [3] and bitvector algorithms [11]). It has been used to search for desired small functional programs by generating a sequence of type-correct programs in a systematic and exhaustive manner and evaluating them against given specifications [20]. In comparison, our algorithm leverages some sophisticated novel concepts, essential for it to scale: obviating need for symbolic reasoning, performing a goal-directed search, and transforming the search space to one with larger width but smaller depth.

Program Synthesis using Input-Output Examples There is a lot of work on inductive program synthesis where the specification from the user comes in the form of multiple input-output examples: string manipulation macros [13], table manipulation programs [15], bit-vector algorithms [18], graph algorithms [16]. In contrast, our system accepts logical specifications, but for scalability reasons, internally transforms the logical specification into a (probabilistically equivalent) input-output example based specification.

Existing inductive program synthesis techniques are based on version space algebras [13, 15] or combinatorial search using SAT/SMT solvers [16, 18]. In case of geometry, the mathematical semantics of the operators involves higher order algebraic equations: any symbolic reasoning (including reduction to SAT using bit-vector blasting) would be prohibitively expensive, and it is noto clear how to use version space algebras in this context.

Interactive Geometry Systems There are several dynamic geometry systems such as Geometer's sketchpad [17] and Cabri Geometer [24] that allow users/students to create geometric constructions, invent conjectures, and check facts. Recent systems also permit one to build proofs (e.g., [2]), or to check facts using an automated theorem prover (e.g., Geometry Expert [10], Cinderella [22], and Geometry Explorer [41]). Automated geometry theorem proving (consisting of several techniques such as Wu's method [42], Grobner basis method [19], and angle method [7]) is one of the most successful areas of automated reasoning. In contrast, we address a technically harder problem of synthesizing constructions, as opposed to the problem of producing correctness proof of a given construction. Our scalable solution relies on a novel approach of using randomness to avoid performing symbolic reasoning. Also, another useful advantage of our framework is that teachers can configure the extended library to ensure that the tool generates small solutions that emphasize use of certain concepts (e.g., ones taught in an earlier chapter). Existing automated geometry theorem proving systems tend to produce arbitrary proofs in the underlying logical domain that may not be readable and may be beyond the vocabulary taught in the class.

GRAMY [27] is an interesting system for proving theorems that in its search for a proof, performs some constructions to enable the application of certain postulates. All computations in GRAMY are symbolic, whereas our search is completely numeric. Our tool discovers a construction, but does not output any symbolic proof of its correctness. GRAMY can only discover proofs that do not involve arithmetic operations. For example, it can not find proofs that involve inequalities, ratios, and coincident intersections. Our tool can generate constructions whose correctness relies on arithmetic reasoning; several of our benchmark examples have this feature. GRAMY uses an exhaustive forward symbolic search, followed by a symbolic backward step to suggest a construction. The backward step enables new forward steps, and the process repeats. Unlike GRAMY, in our case, forward search is non-terminating and numeric. Our backward goal-directed strategy is also numeric, and it is used to prune, and not expand, the forward search.

Chou et. al. [8] also consider the problem of discovering geometry constructions. However, there are several differences: (i) A necessary condition for their approach to work is that the locus of every intermediate point should be a point, line or circle. We do not require this condition. For example, given points A,B, the loci of point C s.t. AC+CB=1 is an ellipse, and hence this paper can not handle such constraints (which we can handle). This severely limits the kind of examples (from highschool textbooks) that their approach can handle. (ii) Their approach does not use any backward goal-directed strategy. As we have demonstrated, forward search needs to be pruned by backward reasoning both for handling large examples, and also to make the tool more useful for pedagogical purposes. (iii) Their approach uses symbolic reasoning (theorem proving), and not numeric computation, to solve the problem. This can affect scalability by orders of magnitude. (iv) Our system is better suited for pedagogical purposes, as we can generate specific kinds of solutions that involve using certain concepts taught in a certain chapter of a certain textbook for a certain grade (by simply changing the library used by our synthesizer). This is in line with the kind of solution that the teacher expects the students to produce. Our system can also extend a partial solution of a student to produce the nearest correct solution as opposed to producing an arbitrary solution.

8. Conclusion and Future Work

We presented a novel synthesis algorithm based on combining numerical methods with symbolic methods, and apply it to synthesizing ruler-compass based geometry constructions.

It may be tempting to question the relevance of plane geometry constructions. Here one should note that plane geometry provides a good platform for teaching logical reasoning abilities while keeping it simple, visual, and fun. "They are just the rules of a game mathematicians play. There are many other ways to do constructions, but the compass and straightedge were chosen as one set of tools that make a construction challenging, by limiting what you are allowed to do, just as sports restrict what you can do (e.g. touching but not tackling, or tackling but no nuclear weapons) in order to keep a game interesting. Other tools could have been chosen instead; for example, geometric constructions can be done using origami." [1]

This paper is the first in the planned series of works on automating high-school education, and in particular, high-school geometry education. We have taken the first step towards this goal and we are in the process of building an end-to-end system that can interact with users in natural language. There is ongoing work on building a natural language front-end for our system that will translate English description of geometry problems to the logical problem representation described in this paper. There is also ongoing work on building a *paraphrasing* back-end for our system that will translate the construction in the programming language described in this paper to natural language.

The technology proposed in this paper forms the foundation for the following technical problems that we plan to tackle next, which will help take this technology to classrooms. The work done in software engineering community can play a big role in addressing some of these problems.

- Provide hints to students instead of the entire solution.
- Point out bugs in an incorrect solution of a student (taking inspiration from the work on bug localization techniques in the programming languages and software engineering community), and suggest fixes to an incorrect solution of student (perhaps using ideas from automated bug fixing techniques [40]),
- Quantify difficulty level of a given problem (by correlating various metrics of a problem, its solution, and scores of students, using ideas from work on software metrics)
- Generate problems of a measured difficulty level.

Solutions to these problems would help in realizing the goal of personalized learning or making teaching *interactive*. However, this is just a short-term goal. The real prize for investing into synthesizers for such domains lies in enabling building of an ultra-intelligent computer, and also a complete model of how the human mind works [32].

References

[1] http://mathforum.org/library/drmath/view/61335.html.

[2] J. R. Anderson, C. F. Boyle, and G. Yost. The geometry tutor. In *IJCAI*, pages 1–7, 1985.

[3] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *DISC*, 2003.

[4] S. Barmany, R. Bodik, S. Chandra, J. Galensony, D. Kimelman, C. Rodarmory, and N. Tungy. Programming with angelic nondeterm. In *POPL*, 2010.

[5] B. Boigelot. On iterating linear transformations over recognizable sets of integers. *TCS*, 309(2), 2003.

[6] S. Chaudhuri and A. Solar-Lezama. Smooth interpretation. In *PLDI*, pages 279–291, 2010.

[7] S. Chou, X. Gao, and J. Zhang. *Machine Proofs in Geometry–Automated Production of Readable Proofs for Geometric Theorems*. World Scientific, 1994.

[8] S.-C. Chou, X.-S. Gao, and J.-Z. Zhang. Automated generation of construction steps for geometric constraint problems. In *Automated reasoning and its applications: Essays in honor of Larry Wos*, pages 49–69. 1997.

[9] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky. *Watch what I do: programming by demonstration*. MIT Press, 1993.

[10] X.-S. Gao and Q. Lin. Mmp/geometer: A software package for auto. geometric reasoning. In *Automated Deduction in Geometry*, volume 2930 of *LNCS*. 2004.

[11] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. In *PLDI*, 1992.

[12] S. Gulwani. Dimensions in program synthesis (invited talk paper). In *ACM Symposium on PPDP*, 2010.

[13] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[14] S. Gulwani, S. K. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[15] B. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[16] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46, 2010.

[17] N. Jackiw. The geometer's sketchpad. Key Curriculum Press, Berkeley, 1991-1995, http://www.keypress.com.

[18] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[19] D. Kapur. Using Gröbner bases to reason about geometry problems. *J. Symb. Comput.*, 2(4), 1986.

[20] S. Katayama. Systematic search for lambda expressions. In *Trends in Func. Programming*, 2005.

[21] G. Katz and D. Peled. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *ATVA*, pages 33–47, 2008.

[22] U. Kortenkamp and J. Richter-gebert. Using auto. theorem proving to improve the usability of geometry software. In *Mathematical User Interfaces*, 2004.

[23] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, 2010.

[24] J.-M. Laborde and F. Bellemain. Cabri-geometry ii. `http://www.cabri.net` (1993-1998).

[25] T. Lau, L. Bergman, V. Castelli, and D. Oblinger. Programming shell scripts by demonstration. In *Workshop on Supervisory Control of Learning and Adaptive Systems, AAAI*, 2004.

[26] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[27] N. Matsuda and K. Vanlehn. Gramy: A geometry theorem prover capable of construction. *J. Autom. Reason.*, 32:3–33, January 2004.

[28] J. M. Robertson. *Geometric Constructions Using Hinged Mirrors*, pages 380–386. May 1996.

[29] P. Rondon. Programming lang. or how to sweet talk a computer, 2010. cseweb.ucsd.edu/∼prondon/talks/plintro.pdf.

[30] C. V. Sanders. *Sharing Teaching Ideas: Geometric Constructions: Visualizing and Understanding Geometry*, pages 554–556. October 1998.

[31] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

[32] L. Shustek. An interview with Ed Feigenbaum. *Commun. ACM*, 53(6):41–45, 2010.

[33] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[34] A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, 2008.

[35] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, 2011.

[36] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[37] A. Taly, S. Gulwani, and A. Tiwari. Synthesizing switching logic using constraint solving. In *VMCAI*, pages 305–319, 2009.

[38] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1948.

[39] M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, 2008.

[40] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

[41] S. Wilson and J. D. Fleuriot. Combining dynamic geometry, automated geometry theorem proving and diagrammatic proofs. In *UITP*, 2005.

[42] W.-T. Wu. Basic principles of mechanical theorem proving in elementary geometrics. *J. Autom. Reason.*, 2(3), 1987.

[43] J. Zhang, L. Yang, and M. Deng. The parallel numerical method of mechanical theorem proving. *Theor. Comp. Sci.*, 74:253–271, 1990.