Solving Geometry Problems using a Combination of Symbolic and Numerical Reasoning

Shachar Itzhaky[1], Sumit Gulwani[2], Neil Immerman[3], and Mooly Sagiv[1]

[1] Tel Aviv University, Israel
[2] Microsoft Research, Redmond, WA, USA
[3] University of Massachusetts, Amherst, MA, USA

Abstract. We describe a framework that combines deductive, numeric, and inductive reasoning to solve geometric problems. Applications include the generation of geometric models and animations, as well as problem solving in the context of intelligent tutoring systems.
Our novel methodology uses (i) deductive reasoning to generate a partial program from logical constraints, (ii) numerical methods to evaluate the partial program, thus creating geometric models which are solutions to the original problem, and (iii) inductive synthesis to read off new constraints that are then applied to one more round of deductive reasoning leading to the desired deterministic program. By the combination of methods we were able to solve problems that each of the methods was not able to solve by itself.
The number of nondeterministic choices in a partial program provides a measure of how close a problem is to being solved and can thus be used in the educational context for grading and providing hints.
We have successfully evaluated our methodology on 18 Scholastic Aptitude Test geometry problems, and 11 ruler/compass-based geometry construction problems. Our tool solved these problems using an average of a few seconds per problem.

Keywords: geometry, reasoning, synthesis

1 Introduction

We describe a framework for solving geometry problems, which are specified as a tuple of inputs, outputs, and constraints between them. The perfect solution to a geometry problem consists of a constructive model generation procedure along with a proof of its correctness. The synthesized procedure can allow models to be constructed in real time, within an interactive environment, as the input points are moved — this has applications in both dynamic geometry environments [WCY05] and animations.

This class of problems is a subset of CLP(R) [JMSY92] — Constraint Logic Programming with Real variables. Current implementation of CLP(R) in Prolog has limited support for non-linear constraints [swi]. Gröbner bases suggest

a technique for solving ruler-and-compass construction problems, but this technique relies on expressing the constraints using polynomials [Buc98]. This is insufficient for our target domain, since problems typically contain numerical data in the form of both angles and length, requiring some use of trigonometry.

Our solver starts out by constructing a model of inputs and outputs that satisfy the constraints using a combination of symbolic and numeric reasoning. To bridge the gap between the two techniques, we use the notion of *partial programs*. A partial program is one that contains "choice" statements, meaning that certain output objects need to be chosen nondeterministically from certain loci. To evaluate these programs in practice, we use numerical methods for minimizing a non-negative function that has the value 0 iff the relevant constraints are met. These methods typically perform well when the number of dimensions is low (up to 2), so a considerable effort is invested in decreasing the search dimension. More specifically, the solver has a built-in knowledge base of geometric theorems, written as a set of Datalog rules. Given an input problem specification, the algorithm tries to identify small search spaces and splits the problem into individual search invocations of low dimension. Once all the output objects are found we have a solution to the given problem. Constructing the instance suffices to solve geometry problems from SAT exams, etc. (This typically requires computing the value of some quantity such as length, angle, area, etc, which can be read off from the model).

Perhaps more interestingly, the solver goes beyond the construction of the model in the following two ways. First, if numeric reasoning was required to constructing the model, then the solver attempts to eliminate this need in an attempt to decrease running time. The procedure works as follows: it constructs a second model for another instance of the problem in which the positions of the inputs have been perturbed. The solver next searches for equalities between distances and angles that occur in both of the constructed models, but were not mentioned in the input specification. According to theorems shown in [Hon86,GKT11], the probability that such equalities are incidental approaches zero. The solver adds these new equalities to the input constraints and solves the new problem. This elimination of numeric search produces a more efficient program, making future evaluations of instance of the same problem much faster. By an *instance* of a given problem we mean the same constraints with different values for the inputs (e.g. different lengths of segments or positions of points). Furthermore, the resulting program provides a complete, constructive solution rather than a numeric approximation. Second, once a deterministic program has been synthesized, our solver generates a proof that the program always constructs a model satisfying the constraints. Thus the correctness of the construction is automatically proved. We view a total program as a perfect solution for a given geometric problem, whereas a partial program searches for the answer. The dimension of the search spaces provides an estimation for the run-time cost of the search.

In the future we plan to use our geometric solver as a helper and tutor for geometry students. The above metric for partial programs will be useful for

measuring how far a student is from a solution, and gauging the "size of hints" that students need to help them solve a given problem.

The main contributions of this paper are the following:

1. Our solver for geometric programs shows how we can combine the complementary strengths of symbolic, numeric, and inductive reasoning.
2. We introduce a non-deterministic language of partial programs for capturing partial insights about geometric constructions. Such programs have an underlying cost corresponding to the size and number of loci that must be searched numerically. This language is useful both as an intermediate data-structure for our solver, and for the user to communicate insights.
3. We provide a substantial experimental evaluation that demonstrates the efficacy of our solver. Out of the 21 questions in SAT practice tests we found freely available on the Internet, we were able to automatically solve 18. The only questions we were not able to solve are those when the size of the problem is part of the input or output (e.g. when the user is asked to determine the number of sides a given polygon has). In 6 of the problems we tested it on, the solver was able to eliminate all of the numeric search steps, thus synthesizing a very efficient program that solves a general version of the given problem.

In the following we define the format of geometry problems that we consider. We present our solver in detail. Finally we report on our experimental results.

2 Geometric Construction Problems

We begin by describing how a geometric construction problem is specified. We also define the three components of the solution to that problem, namely the model, the drawing program, and the proof that the program is correct.

The same formalism also applies to another subclass of problems, which we refer to as *measurement* problems, where a student is required to calculate some value, for example, an angle or an area.

Problem Specification; A geometry construction problem is a CSP — constraint satisfaction problem — consisting of a set $\mathcal{V}$ of variables and a set $\mathcal{C}$ of constraints. Each variable, $v \in \mathcal{V}$, denotes a real number, point, line, or circle. For pure construction problems, the variables are partitioned into input variables $\mathcal{I}$ (thought of as given with the problem) and output variables $\mathcal{O}$ (to be constructed).

For measurement problems, the distinction between inputs and outputs is not significant; instead, a set of query expressions $\mathcal{Q}$ is given, and the output is a numeric value for each such term.

Solution; The solution consists of a **model**, a **drawing program**, and a **proof of correctness**. The model is an assignment to the variables that satisfies all of the constraints $\mathcal{C}$. The drawing program is a sequence of computations. The program is proved correct for all inputs that satisfy their constraints.

3 Partial Programs

We now describe the language of partial programs, which combines imperative and declarative constructs. The solver's first main step will be to construct a partial program that is used to build the desired model.

A partial program is a sequence of instructions. Some of the construction steps require numeric search to find the relevant objects. The language of partial programs is defined by the BNF grammar shown in Figure 1. The scheme is generic, in the sense that it allows for domain-specific predicate and function symbols, denoted by P and F respectively in the grammar.

$$
\begin{array}{rrcl}
\text{Program} & S & ::= & A_1; \ldots; A_n; \\
\text{Statement} & A & ::= & v := F(v_1, \ldots, v_n) \;\mid\; p :\in R \;\mid\; \texttt{Assert}\ \varphi \\
\text{Range} & R & ::= & G(v_1, \ldots, v_n) \;\mid\; R_1 \cap R_2 \\
\text{Constraint} & \varphi & ::= & \gamma_1 \wedge \ldots \wedge \gamma_n \\
\text{Atom} & \gamma & ::= & P(v_1, \ldots, v_n)
\end{array}
$$

Fig. 1. A language for partial programs.

For geometry, We used the set of symbols shown in Table 1. These predicates and functions are very natural for two-dimensional Euclidean geometry. The functions $\texttt{line}(\ell)$, $\texttt{ray}(p, \boldsymbol{u})$, $\texttt{segment}(a, b)$, $\texttt{circle}(p, r)$, and $\texttt{disc}(p, r)$ are *primitive*, in the sense they are internally recognized by the system; the others are just names to use in logical inference rules (see 4.1 below). To re-target the framework to another domain, such as three-dimensional space, a designer may introduce other symbols, but the discussion of this goes beyond the scope of this paper.

Intuitively, the reader may find it useful to think of a partial program as a representation of partial insight into the problem, an algorithm for solving it but with a few "holes".

Example 1. A simple partial program.

1: $a := \langle 0, 0 \rangle$ // a is the origin
2: $b :\in \texttt{circle}(a, 10)$ // b is on circle of given center, radius
3: $c :=$ MIDDLE(a, b) // c is midpoint of segment $\overline{ab}$
4: $\texttt{Assert}\ c.y = 4$ // the y value of c is 4

This program looks for a point b of distance 10 from the origin, such that the midpoint of the line segment from the origin to b has height 4 above the x axis.

The first three statements specify a range of possible values for the objects to be found (in this case, the three points a, b, and c), and the assertion specifies a constraint. An assertion is different from an assignment, in the sense that it constrains properties of objects that have already been assigned.

To evaluate this program, one should search across points on the circle of radius 10 around the origin, for a point b such that $c.y = 4$ holds.

Program variables and functions are typed, and assignments must be properly typed. Thus, in a statement $v := F(v_1, .., v_n)$, if the type of v is T, then F should be a function returning an object of type T, and in a statement $v :\in G(v_1, .., v_n)$, G should return a set $S \subseteq T$. E.g., if v is a point ($T = R^2$), we require that $G(v_1, .., v_n) \subseteq R^2$.

The `Assert` φ statement initiates a numeric search over variables provided in ranges above, but not yet fixed. A successful completion of the search assigns fixed values to some of these variables. Each constraint is translated to the numeric requirement that some necessarily non-negative value be minimized. For example, the constraint $c.y = 4$ is translated to "minimize $(c.y - 4)^2$".

Table 1. Notation for function and predicate symbols used for geometry

`circle`(O,r)	the circle centered at O with radius r
`linetru`(A,B)	the line through A and B
`raythru`(A,B)	the ray whose origin is A and goes through B
`ray`(A,u)	the ray whose origin is A with direction u
`segment`(A,B)	the line segment connecting A and B
Dist(A,B)	distance between points A and b
$\angle$(A,B,C)	the (smaller) angle $\angle ABC$
$\angle_{ccw}$(A,B,C)	the angle $\angle ABC$, measured counterclockwise
Middle(A,B)	the mid-point of the segment AB
Circumf(R)	the circumference of the circle R
ArcDist(O,A,B)	the length of the arc $\overset{\frown}{AB}$ on the circle centered at O
Diameter(R,AB)	**true** iff AB is a diameter in circle R
IntersectSegments(A,B,C,D)	**true** iff AB intersects CD
Colinear(A,B,C)	**true** iff A, B, and C are on the same line

Running example, part I Partial program to generate a regular hexagon.

The following partial program generates a regular hexagon *abcdef* given side *ab*. It first chooses a point o on the perpedicular bisector of *ab*. Next it draws c such that *cob* makes the same angle as *aob* and $oc = ob$. Next draw d such that *dob* makes the same angle as *aob* and $od = ob$, and so on, until point f is drawn. The user then asserts that $\angle foa = \angle aob$.

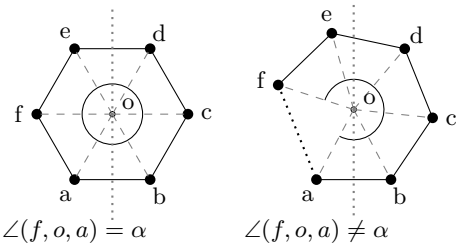


Fig. 2. A hexagon — drawn around its circumcenter; A different choice of o leads to a sub-optimal run

1: $o :\in \textsc{Perp-Bisect}(a, b)$
2: $r := |\overline{ob}|$
3: $\alpha := \angle(a, o, b)$
4: $c :\in \texttt{circle}(o, r) \cap \texttt{ray}(o, \textsc{Rotate}(b - o, \alpha))$
5: $d :\in \texttt{circle}(o, r) \cap \texttt{ray}(o, \textsc{Rotate}(c - o, \alpha))$
6: $e :\in \texttt{circle}(o, r) \cap \texttt{ray}(o, \textsc{Rotate}(d - o, \alpha))$
7: $f :\in \texttt{circle}(o, r) \cap \texttt{ray}(o, \textsc{Rotate}(e - o, \alpha))$
8: `Assert` $\angle(f, o, a) = \alpha$

This partial program relies on the insight that all sides subtend the same angle with the circumcenter of the regular hexagon, as illustrated by Figure 2. Other insights, e.g. that triangle $\triangle abo$ is equilateral, would generate simpler partial programs (see part V of this running example).

3.1 Operational Semantics

The partial program interpreter visits each non-deterministic assignment ($p :\in R$) and attempts to choose a value for p that satisfies the assertions.

To be able to use numeric methods, we interpret each assertion as a non-negative expression that is zero iff the assertion is true. We then choose those points that minimize the sum of these expressions.

For example, the assertion that two real scalar values x, y are equal is translated to the expression $(x-y)^2$ and the assertion that two vectors $\mathbf{u}, \mathbf{v} \in \mathbb{R}^2$ are perpendicular is translated to the square of their inner product, $(\mathbf{u} \cdot \mathbf{v})^2$.

Example 2. Consider the following partial program:
1: $a := (0, 10)$
2: $b := (40, 0)$
3: $c :\in \texttt{segment}(a, b)$
4: `Assert` $|\overline{ac}| = 2|\overline{bc}|$

In the assertion, $|\overline{xy}|$ denotes the distance function. We use a standard hill-climbing algorithm to find the value of c in the segment ab that minimizes the expression $|\overline{ac}| - 2|\overline{bc}|$.

Our hill-climbing procedure discretizes the search space. It partitions it into a finite number of sub-spaces and minimizes the expression among the division points. It then recursively descends to the chosen sub-space. The coarser the discretization factor, the faster the search, but the greater the chances of the search getting stuck in non-optimal local minima and thus requiring random restarts.

The interpreter is implemented using a sequential pass that keeps track of the variables that are not yet determined. It processes each `Assert` statement in turn by invoking numeric search. If the dimension of the combined search space is 1 at that point (space is isomorphic to $\mathbb{R}$), numeric search is done by hill-climbing. If it is 2 or more, we use nested hill-climbing, such that for every value of the first variable that has to be evaluated, we perform hill-climbing on

the second variable and determine an optimum with respect to the value set for the first variable.

The model generation algorithm uses a heuristic for avoiding multi-dimensional search where possible: it iterates the variables (in the order they are defined in the program), fixing them one by one to the minimum obtained from hill-climbing. If at some point, however, the procedure encounters a non-model (the minimum of the target function is not 0), it back-tracks and try different minima for variables that have already been set.

3.2 Cost Metric for Partial Programs

We define a metric to approximate performance of partial programs. The deductive algorithm that creates the partial program tries to construct a minimal one via this metric. As part of this effort, we will consider 3 compile-time criteria:

- Combined dimension of loci being searched;
- Number of choice statements ($v :\in R$) in the program;
- Distance from a choice to its corresponding `Assert`.

A program with smaller dimension will always be preferred over higher dimensions. The statement counts are considered less important.

Definition 1. *The cost of a choice statement $v :\in R$ is defined in terms of a set of* symbolic parameters*, which represent the cost of searching various kinds of spaces (that is, there is some partial ordering between them).*

- S – if R is a segment.
- C – if R is a circle.
- Y – if R is a ray.
- S · C – if R is a disc.
- L – if R is a line.

For an R that is any finite number of points, the cost is 1.

We partition the partial program into *blocks*, where a block is a sequence of statements between two assertions.

Definition 2. *For each assertion, its* cost *is the cost of the block between it and the assertion before it (or the beginning of the program, if this is the first assertion).*

Definition 3. *The* cost of a block *is the product of the costs of all the choice statements in it, and the number of variables* controlled *by the choice statements in the block. It is a* ***polynomial*** *in the symbolic parameters.*

Definition 4. *A variable v is said to be* controlled *by a choice statement iff:*

- It is on the left-hand side of a choice statement, $v :\in R$; or
- It is assigned via $v := F(v_1, .., v_n)$, and there is some v_i which is itself controlled by a choice statement.

Definition 5. *The* cost of a partial program *is the sum of the costs of all the* `Assert` *statements occurring in it.*

When we later say *dimension*, it means the degree of the cost polynomial.

Example 3. The cost of the program in Example 2 is S, because the search is over a segment, and only one variable is controlled by the choice statement.

Running example, part II Consider the partial program from part I.

The choice of o is over the perpendicular bisector of the segment ab (written PERP-BISECT(a, b)) which is a line. The choices for c, d, e, f are then over the intersections of a circle in a ray, which are at most 2 each – so they are assigned a cost of 1. The set of choice-controlled variables in the block is $\{o, c, d, e, f, r, \alpha\}$. The cost is therefore 7L.

4 Solution Generation

Figure 3 shows the phases that our geometry solver follows. The first pass of deductive reasoning produces an initial partial program. This program is run with some inputs to build a model or two. If the partial program is nondeterministic, then the models produced are studied to induce additional constraints. These constraints are then used in a second pass of deductive synthesis to construct a (lower cost) program.

Most of the computational effort goes into identifying implied constraints. Part of them are identified symbolically (4.1) and some numerically (4.3).

4.1 Deductive Reasoning

Our deductive reasoning involves standard application of logic programming with Datalog (e.g., see [AHV95,GMUW09]), which is too weak by itself to solve the problem we are targeting. Later on, we combine deductive with inductive reasoning to make the method more effective. The deductive reasoning procedure builds the partial program, by first doing a single step of preprocessing and encoding, and then running inference in a loop.

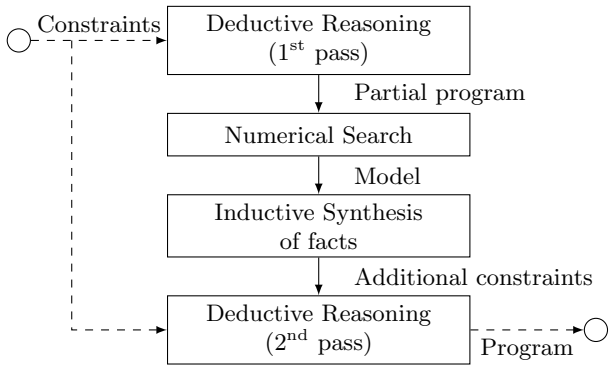


Fig. 3. Architectural diagram

Preprocessing and Encoding Each geometric axiom in our knowledge base is originally given in the form $\varphi(\boldsymbol{U}) \to \psi(\boldsymbol{U})$ where φ and ψ are conjunctions of literals with free variables $\boldsymbol{U}$. The problem specification is a conjunction of ground literals.

The main gap between the language of geometric axioms and Datalog is the presence of function symbols. From both axioms and ground facts, we replace function symbols f of arity k via relation symbols $\tilde{f}$ of arity $k+1$. In particular, we replace each term $f(t_1, t_2, \ldots, t_k)$ by a new symbol α and we assert $\tilde{f}(t_1, t_2, \ldots, t_k, \alpha)$. If the term is a ground term then α is a constant, otherwise it is a variable. As a by-product we loose the information that α is unique, but we will see that this will not keep us from proving the required properties.

This translation may introduce variables in the head of a rule that do not occur in the body. In Datalog terminology, such a rule is *unsafe*. In the next subsection we will explain how our deductions are evaluated. We will point out that since our axioms are "acyclic", deduction remains tractable and in fact bounded, even with these unsafe rules.

We must ensure that each such unsafe variable occurs in exactly one atom. We do this by rewriting each relevant conjunction as a new invented predicate symbol and adding a new rule to define it.

Inference Datalog programs can be efficiently evaluated using seminaïve evaluation as described in [AHV95,GMUW09]. A small extension of this method is needed when instantiating an unsafe rule, e.g., if the variable X_i occurs in the head but not the body of the rule $r(X_1, X_2, \ldots) \leftarrow \varphi$ [KR11].

We instantiate such rules, with *fresh constant symbols* for the unsafe variables. Furthermore, if a constant symbol c already exists such that the corresponding head is already in the generated set, then this instance of the rule is superfluous, so it is not instantiated.

Recall that by construction each such unsafe variable occurs in exactly one atom. This ensures that the derived atom with its fresh constant symbol exactly captures the meaning of the implicit existential quantifier.

Note that the introduction of fresh constant symbols above has the effect of introducing new objects into our system. Our current set of geometry axioms is acyclic meaning that for any input problem only a bounded number of new objects can be created.

Table 2. Axioms for explaining the running example

$1\ \lvert\overline{PQ}\rvert = X \to Q \in \text{circle}(P, X)$
$2\ \lvert\overline{PQ}\rvert = \lvert\overline{SQ}\rvert \to Q \in \textsc{Perp-Bisect}(P, S)$
$3\ \angle(P, Q, S) = Y \to S \in \texttt{ray}(Q, \textsc{Rotate}(P - Q, Y))$

Running example, part III We will show how the partial program from part I might be constructed automatically using this technique.

Assume we have the following declarative specification of the regular hexagon:

$$|\overline{ao}| = |\overline{bo}| = |\overline{co}| = |\overline{do}| = |\overline{eo}| = |\overline{fo}|$$
$$\angle(a,o,b) = \angle(b,o,c) = \angle(c,o,d) = \angle(d,o,e)$$
$$= \angle(e,o,f) = \angle(f,o,a)$$

Our inference system contains the axioms shown in Table 2 (For the sake of this example only, there is an underlying assumption that $\angle$ denotes a counterclockwise angle and ROTATE performs a counterclockwise rotation. This is done to keep the example simple. In practice, we use a richer set of axioms). It produces the following atoms (among others):

$$o \in \text{PERP-BISECT}(a,b);$$
$$c,d,e,f \in \texttt{circle}(o, |\overline{ao}|)$$
$$c \in \texttt{ray}(o, \text{ROTATE}(b-o, \angle(a,o,b)))$$

4.2 Query Planning

Query planning mediates deductive reasoning and numerical search: it attempts to associate a search space with variables that have not been inferred. To this end, the query planner may choose a set of input variables $\mathcal{I}'$. Note that in the case of construction problems, after the second pass it must be that $\mathcal{I}' = \mathcal{I}$ so there is no freedom, but for the first pass we are free to choose any subset.

Locus assignment Let P be the Datalog program representing the axioms, and $\mathbf{I}$ the set of tuples from the specification. $P(\mathbf{I})$ is the result of inference, expressed as sets of ground atoms, e.g., $r(c_1,\ldots,c_k) \in P(\mathbf{I})$.

During this phase of the computation, three relation symbols become important: $\neq$ (disequality), $\in$ (set membership), and known (indicates already-computed values).

To disambiguate these symbols occurring in derived ground atoms from their common mathematical use, we surround such atoms in quotes.

Initially, known $= \mathcal{I}'$. The 'known's are then propagated according to assignments that have been inferred. For each **output symbol** s such that 'known(s)' $\notin P(\mathbf{I})$, look for the following potential search spaces:

1. l, s.t. l is a constant and '$s \in l$','known(l)' $\in P(\mathbf{I})$
2. $l_1 \cap l_2$ s.t. '$l_1 \neq l_2$' $\in P(\mathbf{I})$ and

$$\text{'}s \in l_1\text{'},\text{'}s \in l_2\text{'},\text{'known}(l_1)\text{'},\text{'known}(l_2)\text{'} \in P(\mathbf{I})$$

We choose the "best" locus based on the cost metric of 3.2. The best locus over all symbols is chosen and an assignment of the form '$s :\in R$' is emitted to the program. Then s is marked as known by adding 'known(s)' to $\mathbf{I}$. This process is repeated until all output symbols s have 'known(s)' $\in P(\mathbf{I})$.

Running example, part IV We are given one side of the hexagon, ab. We therefore introduce 'known(a)', 'known(b)'. From these we infer (by way of

deduction) that 'known(PERP-BISECT(a, b))', and the procedure will emit the choice statement '$o :\in$ PERP-BISECT(a, b)'.

As a consequence, 'known(o)' is introduced, which makes two more objects known: $\circ_1$ = '`circle`$(o, |\overline{ao}|)$' and y_1 = '`ray`$(o, \text{ROTATE}(b - o, \angle(a, o, b)))$'. Now — because both $c \in \circ_1$ and $c \in y_1$ are present, it will also emit:
'$c :\in$ `circle`$(o, |\overline{ao}|) \cup$ `ray`$(o, \text{ROTATE}(b - o, \angle(a, o, b)))$'

The other points are traced similarly leading to the program in part I.

Assertion Assignment The assigned search spaces define an over-approximation of the input–output relation. In order to generate a correct partial program, we need to add `Assert` statements. To this end, we go back to the specification, breaking it down into individual constraints. For each constraint, we identify the earliest point in the partial program at which it can be tested, that is, when all of the constraint's arguments have already been defined.

Example 4. If the locus assignment generated the associations in (a) below, and if the specification has the atoms: $|\overline{ab}| = 10$ $|\overline{ac}| = 20$ $|\overline{bc}| = 15$, then knowing only a, none of the constraints can be checked. Knowing a and b allows us to check the first constraint, so an `Assert` statement is inserted after line 2. Knowing a, b, and c provides the means to check the other two constraints, so another `Assert` is added after line 3 (see (b) below).

1: $a := (10, 0)$
2: $b :\in$ `ray`$(a, (1, 1)))$
3: $c :\in$ `circle`$(a, 20)$

(a)

1: $a := (10, 0)$
2: $b :\in$ `ray`$(a, (1, 1)))$
3: `Assert` $|ab| = 20$
4: $c :\in$ `circle`$(a, 20)$
5: `Assert` $|ac| = 20 \wedge |bc| = 15$

(b)

4.3 Inductive Synthesis

In the next phase, we try to improve the efficiency of the program generated by the first pass of deductive reasoning. To do that, we attempt to learn facts that our deductive reasoning technique fell short of inferring by reading them off the model generated by the previous phase. There is an underlying assumption that since the model contains real numbers, then if we perform computations on the values and uncover an equality — with very high probability [Hon86] this equality is not coincidental, but is in fact logically implied by the partial program (hence, by the specification) that created the model in the first place.

The new facts we reveal may then be used by the same deductive reasoning mechanism, as if they were originally given as part of the specifications. Because we now have more information, there is a chance that the second run will yield a lower-cost partial program.

Running example, part V Consider the partial program for drawing the hexagon from part I. The generated model contains 7 points: 6 vertices of the hexagon (a, b, c, d, e, f) and one circumcenter (o). Among the facts learnable from

the model are $|\overline{ao}| = |\overline{ab}|$ and $|\overline{bo}| = |\overline{ab}|$. Given these two facts, the deductive reasoning engine is now able to produce the following code fragment to compute the coordinates of the point o more efficiently:

1: $o :\in \texttt{circle}(a, |\overline{ab}|) \cap \texttt{circle}(b, |\overline{ab}|)$

Replacing line 1 of the original program with this statement would then yield a program with search dimension 0 (because there are only two points in the intersection of the two circles) instead of dimension 1 (an infinite number of points lying on the perpendicular bisector).

Note. section 4 of the technical report [IGIS12] provides a much more detailed study of this example.

5 Evaluation

We consider two kinds of benchmark examples.

- Questions found in SAT practice tests.
- Construction problems, when some elements are given and you are required to draw a new shape: a regular polygon of n sides, given one of them, a square inside a given square, a rectangle inside a given square, a square inside a given triangle, a right triangle, given its circumcircle, an equilateral triangle touching 3 given parallel lines

Appendix A contains a partial listing of SAT benchmarks. A full listing of our benchmarks can be found in [IGIS12].

5.1 Generation of Partial Programs

We show that our partial program generation scheme is very effective. We evaluate this by comparing statistics about model generation for the following cases:

- Without a partial program
- Using deductive synthesis.
- Using a combination of deductive + inductive synthesis.

Table 3 contains the statistics of time taken to generate a model and the total number of dimensions that were searched (For example, the number of dimensions for a completely unknown point is 2, while the number of dimensions for an unknown point that lies on a circle is 1). The column "O" shows the original dimension of the problem, if we were to apply numerical methods to it directly.

On the first pass, the symbolic part generates a partial program (as described in 4.1, 4.2), and the numeric part generates a model via hill-climbing search based on the partial program. The running time (in seconds) of each part is provided in columns "S" (symbolic) and "N" (numeric) below "1$^{\text{st}}$ pass". The resulting dimension is shown in column "R", and "k" is the maximal dimension of the individual search space associated with each `Assert` (see 3.2). Where k is lower

than the total dimension, it means that the multi-dimensional search has been decomposed into several searches of lower dimension, improving performance considerably.

Table 3. Benchmark measurements

	1st pass					2nd pass			
	Dimension			Time (s)		Dim.		Time (s)	
#	O	R	k^*	S	N	R	k^*	S	N
1	4	1	1	0.16	0.54	0	0	0.98	0.00
2	2	0	0	0.04	0.00	0	0		
3	4	1	1	0.14	0.35	1	1	0.13	0.36
4	6	1	1	0.22	0.12	0	0	0.40	0.00
5	8	4	1	0.35	0.24	1	1	5.19	0.11
6	6	1	1	0.38	0.84	1	1	3.23	1.63
7	4	1	1	0.09	0.02	1	1	0.12	0.02
8	4	2	1	0.38	0.02	2	1	0.42	0.02
9	8	2	2	0.64	38.13	1	1	1.86	0.62
10	14	1	1	0.73	0.53	1	1	21.16	0.54
11	12	2	1	0.63	0.86	0	0	12.17	0.01
12	8	1	1	0.22	0.02	1	1	0.59	0.02
13	4	1	1	0.18	0.06	1	1	0.18	0.06
14	6	1	1	0.06	0.03	1	1	0.10	0.03
15	10	2	1	0.29	1.20	2	1	11.93	0.98
16	7	1	1	0.53	0.01	1	1	1.41	0.02
17	8	2	1	0.27	0.47	2	1	0.54	0.46
18	10	1	1	0.20	0.04	1	1	0.70	0.03
19	6	2	1	0.23	0.08	2	1	0.31	0.08
20	8	0	0	0.14	0.00	0	0		
21	11	2	1	0.11	0.26	1	1	0.85	0.03
22	4	1	1	0.08	0.01	1	1	0.09	0.01
23	6	0	0	0.56	0.00	0	0		
24	10	2	1	0.18	0.04	2	1	0.95	0.04
25	4	1	1	0.22	0.10	1	1	0.24	0.09
26	10	2	1	0.35	0.19	2	1	0.49	0.19
27	10	2	1	0.32	0.04	2	1	1.13	0.35
28	8	3	1	0.37	0.48	3	1	0.37	0.46
29	6	1	1	0.47	0.03	1	1	0.75	0.03

* k (the *rank*) is the maximal dimension of the search space as defined in 3.2

The results of the second pass show the effect of incorporating results of inductive synthesis, that is, facts learned by querying the model generated by the first pass. In 6 of the cases, the values in the columns of "2nd pass" exhibit lower dimensions compared to the first pass. The running time of the symbolic reasoning part is higher, due to the increase in the number of formulas to process. In most cases, however, this effort is worthwhile as it leads to a faster program, reducing the running time of the numeric part.

5.2 Proof Statistics

With the deductive inference mechanism shown earlier, the average number of steps effectively used to generate the program (not including tried and failed paths) was 51.7. We had 47 axioms; each axiom was used 31.9 times on average. The average number of statements per partial program generated was 8.2.

6 Related Work

Geometry constraint solving is a long studied problem, where the goal is to find a configuration for a set of geometric objects that satisfy a given set of constraints between the geometric elements [BFH+95]. A variety of techniques have been proposed including logical inference and term rewriting [Ald88], numerical methods [Nel85], algebraic methods [Kon92], and graph based constraint solving [BFH+95]. These

techniques either require some symbolic reasoning or some form of search. Our work is different from these works in two regards. First, we combine **both** symbolic reasoning and numerical search for model generation. Second, we deal with the more sophisticated problem of **constructive** model generation. While essentially an instance of CLP(R) [JMSY92], geometry has its own properties, which we use to create a specialized solver.

This paper is most closely related to some recent work in the area [GKT11]. Our methodology of program generation followed by model generation is similar and relies on the same theoretical result about geometry property testing. We add to it the incorporation of symbolic deduction, and the additional artifact of the partial program, which provides a more general answer to a given problem and also conveys some insight about the solution.

Our notion of partial programs, which combine imperative and declarative constructs for geometry constructions is similar to a recent proposal on doing so for a general purpose programming language [SL08]. Our interpretation of a partial program is based on use of numerical methods unlike use of SMT solvers [KKS12]. More significantly, we also automate the construction of a partial program from fully declarative specifications using deductive reasoning, and also refine a partial program into one that is more constructive using inductive synthesis techniques.

7 Conclusion and Future Work

We have presented a system that constructs geometric figures. It also allows insights from the user in the form of partial programs. In the case of end-users, this interactivity allows humans and machines to work together to solve complicated problems. In the educational domain, this interactivity allows students to express partial insights about a geometry construction problem, which the system can then extend to a complete solution, following the student's hint. In the future we will perform user studies both in the end-user setting and the classroom setting. We believe that the methodology we have introduced, combining deductive and inductive synthesis via partial programs, will find uses in many other domains.

References

AHV95. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

Ald88. Bernd Aldefeld. Variation of geometries based on a geometric-reasoning method. *Computer Aided Design*, 20(3):117–126, April 1988.

BFH+95. William Bouma, Ioannis Fudos, Christoph M. Hoffmann, Jiazhen Cai, and Robert Paige. Geometric constraint solver. *Computer-Aided Design*, 27(6):487–501, 1995.

Buc98. Bruno Buchberger. Applications of Gröbner bases in non-linear computational geometry. *Lecture Notes in Computer Science*, 296:52–80, 1998.

GKT11. Sumit Gulwani, Vijay Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *Programming Language Design and Implementation (PLDI)*, 2011.

GMUW09. Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

Hon86. Jiawei Hong. Proving by example and gap theorems. In *FOCS*, pages 107–116. IEEE Computer Society, 1986.

IGIS12. Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. Solving geometry problems using a combination of symbolic and numerical reasoning. Technical Report MSR-TR-2012-8, Microsoft Research, Jan 2012. Available from `http://www.cs.tau.ac.il/~shachar/dl/tr-2012.pdf`.

JMSY92. Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, May 1992.

KKS12. Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.

Kon92. Kunio Kondo. Algebraic method for manipulation of dimensional relationships in geometric models. *Computer-Aided Design*, 24(3):141–147, 1992.

KR11. Markus Krötzsch and Sebastian Rudolph. Extending decidable existential rules by joining acyclicity and guardedness. In Toby Walsh, editor, *IJCAI*, pages 963–968. IJCAI/AAAI, 2011.

Nel85. Greg Nelson. Juno, a constraint-based graphics system. In *SIGGRAPH*, pages 235–243, 1985.

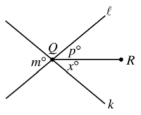
SL08. Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

swi. http://www.swi-prolog.org/man/clpqr.html.

WCY05. Wing-Kwong Wong, Bo-Yu Chan, and Sheng-Kai Yin. A dynamic geometry environment for learning theorem proving. In *Proceedings of the 5th IEEE International Conference on Advanced Learning Technologies, ICALT 2005, 05-08 July 2005, Kaohsiung, Taiwan*, pages 15–17. IEEE Computer Society, 2005.

A Examples of Benchmarks

This is a partial listing. The full list can be found in the Technical Report [IGIS12].

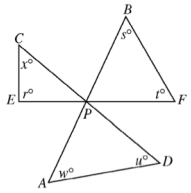


Note: Figure not drawn to scale.

4. In the figure above, lines ℓ and k intersect at point Q. If $m = 40$ and $p = 25$, what is the value of x?

(A) 15
(B) 20
(C) 25
(D) 40
(E) 65

$\text{dist}(Q, A) = 100$
$\text{dist}(Q, R) = 100$
$Q \neq L$
$\angle_{ccw}(R, Q, L) = 25$
$\text{middle}(K, B) = Q$
$\text{known}(B)$
$Q \neq B$
$\angle_{ccw}(B, Q, A) = 40$
$\text{middle}(L, A) = Q$
$\text{known}(Q)$
$?(A, R, L, K)$

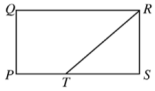


Note: Figure not drawn to scale.

7. In the figure above, $\overline{AB}$, $\overline{CD}$, and $\overline{EF}$ intersect at P. If $r = 90$, $s = 50$, $t = 60$, $u = 45$, and $w = 50$, what is the value of x?

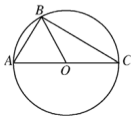
(A) 45
(B) 50
(C) 65
(D) 75
(E) It cannot be determined from the information given.

$\angle_{ccw}(D, A, B) = 50$
$\angle_{ccw}(A, B, F) = 50$
$\angle_{ccw}(F, E, C) = 90$
$\text{segment}(C, D) = CD$
$P \in CD$
$P \in EF$
$\text{known}(B)$
$\angle_{ccw}(C, D, A) = 45$
$\angle_{ccw}(B, F, E) = 60$
$\text{segment}(A, B) = AB$
$P \in AB$
$\text{segment}(E, F) = EF$
$\text{known}(A)$
$?(C, D, E, F, P)$



18. In the figure above, $PQRS$ is a rectangle. The area of $\triangle RST$ is 7 and $PT = \frac{2}{5} PS$. What is the area of $PQRS$?

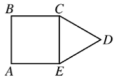
$\angle(P, S, R) = :90:$
$\angle(S, R, Q) = :90:$
$\angle(R, Q, P) = :90:$
$\angle(Q, P, S) = :90:$
$5 \cdot r = 2$
$\text{known}(P)$
$?(R, Q, T)$
$\text{segment}(P, S) = PS$
$T \in PS$
$\text{dist}(P, S) = d$
$\text{dist}(P, T) = k$
$r \cdot d = k$
$\text{known}(S)$



5. In the figure above, triangle ABC is inscribed in the circle with center O and diameter $\overline{AC}$. If $AB = AO$, what is the degree measure of $\angle ABO$?

(A) 15°
(B) 30°
(C) 45°
(D) 60°
(E) 90°

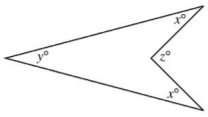
$\text{circle}(O, 75) = R$
$A \in R$
$B \in R$
$C \in R$
$\text{segment}(A, C) = AC$
$\text{dist}(B, O) = d$
$\text{known}(O)$
$A \neq B$
$A \neq C$
$B \neq C$
$O \in AC$
$\text{dist}(A, B) = d$
$?(A, B, C, R)$



4. In the figure above, CDE is an equilateral triangle and $ABCE$ is a square with an area of 1. What is the perimeter of polygon $ABCDE$?

(A) 4
(B) 5
(C) 6
(D) 7
(E) 8

$\text{square}(A, B, C, D)$
$\text{dist}(B, E) = e$
$\text{dist}(C, E) = e$
$\text{dist}(B, C) = e$
$\text{known}(A)$
$\text{known}(B)$
$?(C, D, E)$



Note: Figure not drawn to scale.

12. If $x = 20$ and $y = 30$ in the figure above, what is the value of z?

(A) 60
(B) 70
(C) 80
(D) 90
(E) 100

$\angle_{ccw}(A, B, C) = 30$
$\angle_{ccw}(B, C, D) = 20$
$\angle_{ccw}(D, A, B) = 20$
$\neg\text{colinear}(A, C, D)$
$\text{known}(A)$
$\text{known}(B)$
$?(C, D)$