Automation of Triangle Straightedge-and-Compass Constructions Using Automated Planning

Milan Banković[1*]

[1*]Faculty of Mathematics, University of Belgrade, Studentski Trg 16, Belgrade, 11000, Serbia.

Corresponding author(s). E-mail(s): milan.bankovic@matf.bg.ac.rs;

Abstract

In this paper, we consider automated solving of triangle straightedge-and-compass construction problems by reducing them to automated planning. We consider the problems from the Wernick's list, where each problem assumes that locations of three significant points of a triangle are given, and the goal is to construct all three vertices of the triangle. We develop two different models of the corresponding planning problem. In the first model, the planning problem is described using the PDDL language, which is suitable for solving with dedicated automated planners. The second model assumes that the planning problem is first expressed as a finite-domain constraint satisfaction problem, and then encoded in the MiniZinc language. Such model is suitable for solving with constraint solvers. In both cases, we employ existing artificial intelligence tools in search for a solution of our construction problem. The main benefit of using the existing tools for such purpose, instead of developing dedicated tools, is that we can rely on the efficient search that is already implemented within the tool, enabling us to focus on geometric aspects of the problem. We evaluate our approach on 74 solvable problems from the Wernick's list, and compare it to the dedicated triangle construction solver ArgoTriCS. The results show that our approach tends to be superior to dedicated tools in terms of efficiency, while it requires much less effort to implement. Also, we are often able to find shorter construction plans, thanks to the optimization capabilities offered by the modern planners and constraint solvers. The presented approach is only a search method and does not address proving the correctness of the obtained constructions and discussing when solutions exist, leaving these tasks to other tools. Although the paper focuses on a specific set of construction problems, the approach can be generalized to other classes of problems, which will be explored in future work.

Keywords: Construction problems, Automated planning, Constraint solving, SMT solving

1 Introduction

Construction problems are among the oldest and the most studied problems in geometry. Such a problem is usually defined as follows: given some elements of a figure (such as triangle), the goal is to find a sequence of steps to construct the remaining elements of the figure using the available tools – typically a *straightedge* and a *compass*. Such a sequence is called a *construction plan*, and the obtained figure is called a *solution* of that construction problem. Solving construction problems usually requires specific, and often very deep geometric knowledge. Many construction problems turned out to be very challenging.

From the algorithmic point of view, the construction problems are *search* problems, and the search space is usually very large: at each step we have to choose between many different types of elementary construction steps and many possible objects that can be constructed, and the number of steps required to solve a specific problem can be quite large. There are two possible approaches here: one is to develop a specific search algorithm in some programming language with required geometric knowledge compiled into it, and the other is to use existing artificial intelligence tools that are good in solving search problems in general. In the second case, one should only specify the problem and the required knowledge using an appropriate input language and then leave the search to an appropriate tool.

The first approach is considered, for instance, by Marinković [12]. In this paper, we consider the second approach. More specifically, we show how construction problems can be expressed as *planning problems* [9], and then solved by tools that are already available for such problems. We use two types of tools for that purpose. The first type of tools are dedicated solvers for planning problems (known as *planners*). For such tools, we develop a model of our problem in PDDL language [1], understood by most modern planners. The second type of tools are constraint solvers, such as finite-domain solvers [17], or SAT/SMT solvers [4]. In order to use these tools, we have to express our planning problem as a *constraint satisfaction problem* (CSP), and then encode it in a suitable language (we use the MiniZinc [14] constraint modeling language for that purpose).

The main benefit of using off-the-shelf tools as search procedures is that we may focus on geometric aspects of the problem and on modeling the geometric knowledge required for its solving, and leave the search to the tool that is good at it. That way, we save our time and effort. Moreover, we can expect that the search will be less time-consuming, since we are using well-tuned and highly optimized tools for that purpose.

A single construction problem instance may be solved using different construction plans, as the same object can often be constructed in different ways, relying on different fragment of our geometric knowledge. Among these construction plans, we prefer

shorter ones (i.e. plans involving fewer construction steps), because they are easier to follow and understand and tend to have simpler visual presentations. To achieve this, we must incorporate optimization capabilities into our search procedure with the goal of minimizing the construction plan length. Such capabilities already exist in state-of-the-art tools, and we would need to implement them manually if we developed our own search tool from scratch. This is another benefit of using existing tools as search procedures.

Note that solving construction problems involves more than just the search. After a construction plan is found, one must *prove* that the obtained solution is *correct*, i.e. that the constructed objects indeed satisfy the specification of the problem. Moreover, a *discussion* on the conditions for the existence of a solution is typically the final phase of solving a construction problem instance. Having this in mind, we must stress that the method we are considering in this paper is purely a search method, and it does not deal with the remaining phases of solving construction problems. The goal of the proposed method is to isolate the search component of the problem and solve it in a simple and efficient way, using off-the-shelf solvers. The obtained construction plans can then be fed into other tools (such as *theorem provers*) to handle the rest of the solving process.

In this paper, we consider a specific class of construction problems – the straightedge-and-compass triangle construction problems from the Wernick's set [21]. The problems from this set assume that *locations* of three significant points of a triangle (such as the centroid, the orthocenter, the feet of the altitudes, etc.) are given in a plane, and the goal is to construct the vertices of the triangle. The set consists of 139 problems, 74 of which are proven to be solvable by a straightedge and a compass. We evaluate our approach on these 74 solvable instances, and compare it to the state-of-the-art tool for automated generation of triangle constructions ArgoTriCS [12], developed in the Prolog programming language, also primarily targeting the problems from the Wernick's list. The comparison is made in terms of efficiency (measured by solving time) and the quality of the obtained construction plans (measured by the plan length). Note that ArgoTriCS is not just a search procedure. It can detect redundant, symmetric and locus dependent problems. It can also generate data needed for the proving and the discussion phases. However, since the focus of this paper is on the search (and our method does only the search part), we do not consider those other functionalities of ArgoTriCS in the comparison.

Although the work presented in this paper can be viewed as a case study of a particular class of problems, the method could be generalized to other classes of construction problems. The general idea can be summarized as follows:

- fix a class of construction problems that we want to solve. In our case, we consider the Wernick's set of problems.
- specify a set of *construction step types* that are allowed to be used in the constructions. In our case, we use a set of construction step types that can be performed by a straightedge and a compass (virtually the same set as the one used by ArgoTriCS [12]), but this can be generalized to other tools and the step types that can be performed by such tools (e.g. tools offered by some *dynamic geometry* software).

- identify the geometric knowledge needed for solving the problems from the chosen class. Such knowledge is represented by a set of relevant *objects* (such as points, lines, or circles) and the set of *facts* about these objects (given by a set of *definitions* and *lemmas*). In our work, we rely on the knowledge presented in [12], on which ArgoTriCS is also based.
- compile the identified knowledge into the model (in the form of *relations* between the objects).
- incorporate the chosen set of construction step types into the model (in the form of *actions* that can be executed in our plans).
- generate a set of problem instances (in our case, we generate the set of 74 solvable Wernick's instances)
- run a chosen of-the-shelf solver on the generated instances
- collect the obtained construction plans and fed them into other tools that will handle other phases of the solving process (proving correctness, discussion, visualization, etc.).

In this work, we cover all these points, except the last one, for the set of Wernick's problems. We hope that the success of this case study will motivate other researchers to apply the same approach to other classes of construction problems.

Some of the results shown in this paper were preliminarily presented at the *ADG 2023 conference* [13]. This paper extends those results in the following ways:

- The MiniZinc model is optimized by removing some symmetries that we noticed. This significantly improved the results obtained by finite-domain constraint solvers that were presented in [13].
- We present the results obtained by SMT solvers (applied to the MiniZinc model). SMT solvers were not used in experiments in [13].
- The PDDL model and the usage of PDDL-based automated planners were not considered at all in [13], so it is a completely new contribution of this paper.

The rest of this paper is organized as follows. In Section 2, we introduce needed concepts and notation used in the rest of this paper. In Section 3 we provide a brief overview of the geometric knowledge needed for solving the problems from Wernick's set and discuss the representation of that knowledge in our models. In Section 4 we describe our models (both PDDL and MiniZinc). Section 5 provides a detailed evaluation of the approach. Finally, in Section 6, we give some conclusions and mention some directions of the further work.

2 Background

2.1 Straightedge-and-Compass Triangle Constructions

In straightedge-and-compass triangle construction problems, the goal is to construct all three vertices of a triangle (usually denoted as A, B and C), assuming that some elements of the triangle (such as points, lines or angles) are given in advance. A *construction plan* consists of a sequence of steps, where in each step some new objects (points, lines, angles or circles) are constructed based on the objects constructed in

previous steps. Constructions performed in each of the steps are usually *elementary* ones, such as constructing the line passing through two given points, or the point that is the intersection of two given lines, or the circle centered at a given point that contains another given point. However, in order to simplify the description of a triangle construction, some higher-level construction steps are also considered, such as constructing the tangents to a given circle from a given point, or the line perpendicular or parallel to a given line and passing through a given point, etc. Such higher-level constructions are called *compound* constructions, since they can be easily decomposed into sequences of elementary construction steps.

In this paper, we focus on triangle construction problems from the Wernick's list [21]. Each problem from the list assumes that locations of three different points from the following set are given: the triangle vertices A, B, C, the circumcenter O, the incenter I, the orthocenter H, the centroid G, the feet of the altitudes H_a, H_b, H_c, the feet of the internal angles bisectors T_a, T_b, T_c and the midpoints of the triangle sides M_a, M_b, M_c. The goal is to construct the vertices[1] A, B and C of the triangle (if some of them are given, the goal is to construct the remaining ones).

From these 16 characteristic points of a triangle, we can form $\binom{16}{3} = 560$ different point triples that can be considered. However, only 139 among them represent significantly different problems (that is, mutually non-symmetric). Among these, only 74 problems are proven to be solvable by a straightedge and a compass (others either contain redundant points, or are undetermined, i.e. may have infinitely many solutions, or are proven to be unsolvable [18]). In our work, we consider only these 74 solvable problems from the Wernick's list.

For instance, assume that the locations of the vertex A, the orthocenter H and the centroid G of a triangle are given. The goal is to construct the remaining vertices B and C. This can be achieved by the following construction plan:

1. construct the line through points A and H (the altitude h_a)
2. construct the point X such that $\overrightarrow{AX} = \frac{3}{2} \cdot \overrightarrow{AG}$ (this is the midpoint M_a of the side BC).
3. construct the point Y such that $\overrightarrow{HY} = \frac{3}{2} \cdot \overrightarrow{HG}$ (this is the circumcenter O)
4. construct the line through M_a perpendicular to the line h_a (this is the line a containing the side BC)
5. construct the circle with the center O that contains the point A (this is the circumcircle $k(O, A)$ of the triangle)
6. construct both intersections of the circumcircle with the line a (these are the vertices B and C).

The corresponding solution is illustrated in Figure 1. Note that we have to be equipped with some geometric knowledge in order to come up with a construction plan like this one (and also to prove its correctness). For instance, we have to know that the orthocenter H is the intersection of the altitudes (so it belongs to h_a), that the centroid G divides the segment AM_a such that $\overrightarrow{AG} = 2 \cdot \overrightarrow{GM_a}$, that the circumcenter O, the centroid G and the orthocenter H are colinear and that $\overrightarrow{HG} = 2 \cdot \overrightarrow{GO}$, etc. Such

[1] Note that the construction of the triangle sides is not required. Of course, this can be easily achieved once the locations of the vertices are known.

geometric knowledge can be expressed by a finite set of definitions (that introduce new objects by relating them to already known objects) and lemmas (proven statements about the properties of the objects that follow from the definitions and the axioms of the Euclidean geometry). A detailed list of definitions and lemmas needed for solving the problems from the Wernick's list is given in [12].

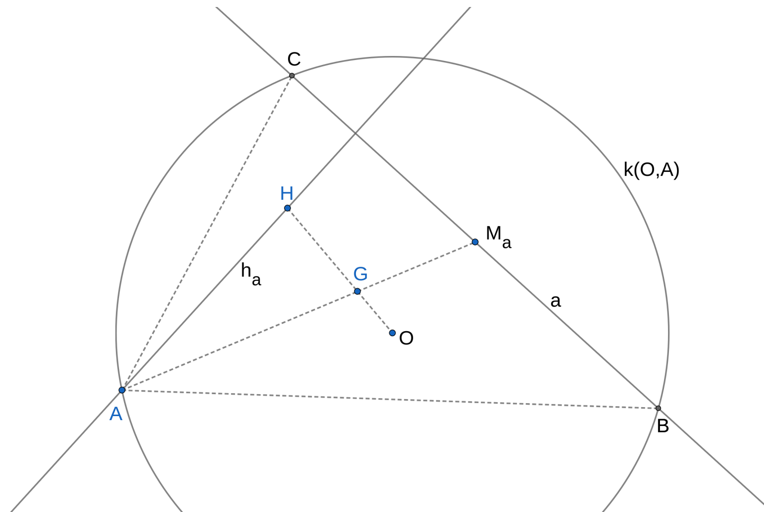

Fig. 1 An illustration of a solution for the problem (A, G, H). Blue-colored points are given, and other elements denoted in black are constructed. It holds $\overrightarrow{HO} = \frac{3}{2} \cdot \overrightarrow{HG}$ and $\overrightarrow{AM_a} = \frac{3}{2} \cdot \overrightarrow{AG}$. The dashed segments are not constructed; they are depicted only to aid visualization.

Solutions of triangle construction problems may include constructions of many auxiliary objects (points, lines, circles and angles). For such constructions to be useful for achieving our final goal (construction of the vertices A, B and C), the constructed objects must be related to other triangle objects through definitions and lemmas that form the foundation of our geometric knowledge. In other words, only the objects that we know something about make sense to be constructed. Given a fixed knowledge base (expressed by a finite set of definitions and lemmas), we obtain a finite set of objects that may be considered in each step of our construction plan. This gives as a finite (although quite large) search space when looking for a construction plan. Figure 2 shows some of the objects that may be considered when solving problems from the Wernick's list. The exact set of objects and the geometric knowledge that we assume in this paper is presented by Marinković in [12]. We also assume the set of elementary and compound construction steps used in that work.

As said earlier, solving a construction problem does not involve only the search for a construction plan, it also involves proving that the obtained plan is *correct*, which means that the constructed objects satisfy the specification of the problem. In our previous example, one must prove that the points G and H are indeed the centroid and the orthocenter of the constructed triangle ABC, assuming the construction sequence given above. In order to prove the correctness, we rely on the same set of lemmas and

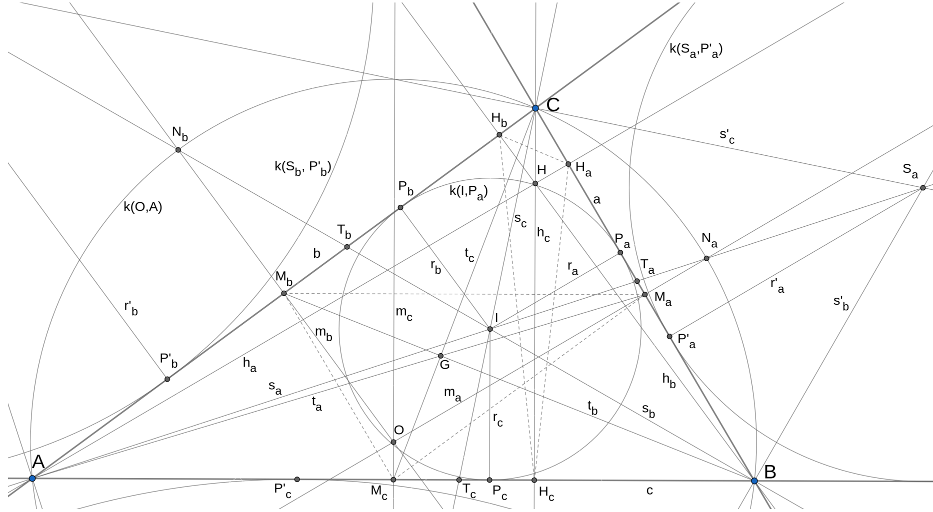

Fig. 2 A triangle and (some of) its significant objects (points, lines and circles).

definitions that we used to obtain the above construction plan. In our example, one possible proof might be the following:

- by construction, it follows that the point O is the circumcenter of the triangle ABC.
- since $\overrightarrow{AM_a} = \frac{3}{2} \cdot \overrightarrow{AG}$ and $\overrightarrow{HO} = \frac{3}{2} \cdot \overrightarrow{HG}$ (by construction), we prove that $\overrightarrow{AH} = 2 \cdot \overrightarrow{OM_a}$ (by simple vector arithmetics), so the line OM_a is parallel to the line h_a.
- since the line a is perpendicular to the line h_a (by construction), it follows that the line OM_a is also perpendicular to the line a. Since it contains the circumcenter O, the line OM_a is the perpendicular bisector of the side BC, so the point M_a is its midpoint.
- since $\overrightarrow{AM_a} = \frac{3}{2} \cdot \overrightarrow{AG}$ (by construction), and M_a is the midpoint of BC, the point G is indeed the centroid of the triangle ABC.
- since $\overrightarrow{HO} = \frac{3}{2}\overrightarrow{HG}$ (by construction), the point H is indeed the orthocenter of the triangle ABC.

A proof like this can be generated by automated theorem provers or manually derived using interactive theorem provers. To assist with the proof, a procedure that generates the construction plan may also export a list of used lemmas and definitions, which can be fed into the theorem prover. For example, such an export is performed by ArgoTriCS [12]. In our work, we do not consider correctness proofs, so our method currently does not export this information along with the obtained construction plan.

The final phase of solving construction problems is a *discussion* on when (and how many) solutions exist and under which conditions. These conditions are obtained by analyzing each of the steps in the obtained construction plan. Namely, each step may have its *non-degeneracy conditions* (i.e. when the objects constructed by the step exist), and its *determination conditions* (i.e. when the objects constructed by the step are uniquely determined). For instance, we can always construct a line through two points (there are no non-degeneracy conditions), but the points must be distinct

for the line to be uniquely determined. On the other hand, if we want to construct the intersection of two given lines, then the non-degeneracy condition is that the lines are not parallel, and the determination condition is that the lines are distinct. The conditions that a solution obtained by the construction plan exists (is uniquely determined) are obtained simply by conjoining the non-degeneracy (determination) conditions of the individual steps. In the above example, a solution will exist whenever the circle $k(O, A)$ and the line a have two points in common, and will be unique if the points A and H are distinct. Note that the non-degeneracy and determination conditions are not considered during the search phase, that is, they are discussed only after the construction plan is obtained. Since our work deals only with the search, we do not consider the discussion phase in this paper.

2.2 Automated Planning

In our approach, triangle construction problems are considered as problems of *automated planning* [9]. An automated planning problem consists of the following:

- a set $\mathcal{S}$ of possible *states*. Each state is represented by a subset of a given set of *facts* $\mathcal{F}$ that are considered to be true in that particular state. These facts are usually described in a first-order fashion – as *predicates* applied to *objects* (such as $is_open(door)$ or $holds(monkey, banana)$). One distinguished state $S_0 \in \mathcal{S}$ is the *initial state*.
- a set of *actions* (or *operators*) A, where each action $a \in A$ consists of a *precondition* C_a describing the conditions (in terms of facts from $\mathcal{F}$) that must be satisfied in the current state in order to apply the action, and a set of *effects* E_a describing how the current state is changed when the action a is applied to it (this may include both introducing new facts from $\mathcal{F}$ into the state, and removing some existing facts from the current state). For instance, we may have an action with the precondition $holds(monkey, banana)$ and the effects $\{+eaten(banana), -hungry(monkey)\}$, meaning that the first fact is added to the state, and the second fact is removed from the state. The state obtained by applying an action a to some state S is denoted by $a(S)$.
- a goal G, describing the conditions (again, given in terms of the facts from $\mathcal{F}$) that must be satisfied in the final state.

The objective of automated planning is to find *a plan*, that is, a finite sequence of actions $a_1, \ldots, a_n$ from A that can be successively applied to the initial state S_0 (i.e. for each $i \in \{1, \ldots, n\}$, we have $S_i = a_i(S_{i-1})$, and the state S_{i-1} satisfies the precondition C_{a_i}) producing the final state S_n satisfying the goal G. The number n of actions used in a plan is called the *length* of the plan.

In order to solve a planning problem, we first have to model it using a suitable language. Two most popular languages for modeling planning problems are STRIPS [8] and PDDL [1]. In our work, we use PDDL for modeling planning problems. PDDL enables expressing the planning problems in a very natural way, since it directly supports declaring objects, predicates, actions (with preconditions) and goals. The modeling is usually divided into two parts. The first part is the *model* itself, which defines the general class of problems, in terms of supported object types, available

predicates, and actions whose preconditions and effects are parameterized with variables that can be instantiated with arbitrary objects. The second part considers a particular *problem*, given by a concrete set of objects, the initial state (given as a set of facts), and the goal.

After the problem is described in a modeling language such as PDDL, it can be automatically solved by an appropriate tool. Such tools are known as *AI planners* (or just *planners*, for short). Modern planners are usually based on some advanced search algorithm (such as A^*), equipped with heuristics that are well-tuned for planning problems. Another approach is to reduce the planning problem to constraint solving, which will be considered in the next section.

2.3 Constraint Solving

One way of solving planning problems is to reduce them to *constraint satisfaction problems* [17]. A finite-domain *constraint satisfaction problem* (CSP) consists of a finite set of *variables* $\mathcal{X} = \{x_1, \ldots, x_n\}$, each taking values from its given finite domain $D_i = D(x_i)$, and a finite set of *constraints* $\mathcal{C} = \{C_1, \ldots, C_m\}$, which are relations over subsets of these variables. A *solution* of a CSP is an assignment $(x_1 = d_1, \ldots, x_n = d_n)$ of values to variables ($d_i \in D_i$), such that all the constraints of that CSP are satisfied. A CSP is *satisfiable* if it has at least one solution, otherwise is *unsatisfiable*. The optimization version of CSP, known as a *constrained optimization problem* (COP) additionally assumes a function f over the variables of the problem that should be minimized (or maximized), with respect to the constraints from $\mathcal{C}$.

Tools that implement procedures for solving CSPs (and COPs) are called *constraint solvers*. They are usually based on a combination of a backtrack-based search and constraint propagation [17]. Constraint solvers have been successfully used for solving many real-world problems in many fields, such as scheduling, timetabling, resource allocation, combinatorial design, and so on.

An important step in using constraint solvers is *constraint modeling*, that is, representing a real-world problem in terms of variables and constraints. A constraint model is described using an appropriate modeling language. One such language supported by many modern constraint solvers is *MiniZinc* [14], which we use in our work. This language offers a very flexible high-level environment for modeling different kinds of constraints, enabling a compact and elegant way to represent some very complex problems. Examples of some high level language elements include tuples, multi-dimensional arrays, sets, aggregate functions, finite quantification and so on. Since most of these high level constructs are not supported by backend solvers, each MiniZinc model must be translated into an equivalent *FlatZinc* form, containing only primitive language constructs and constraints supported by a chosen backend solver. MiniZinc supports modeling of both CSPs and COPs.

In MiniZinc, we distinguish *variables* from *parameters*. MiniZinc variables correspond to the variables of a CSP, i.e. we declare their domains and expect from the solver to find their values satisfying the constraints. On the other hand, parameters are just named constants, and their values must be known when the model is translated to the FlatZinc form (i.e. before the solving starts). Parameters are the language's construct that allow us to specify a general model for a class of problems, and then to

choose a specific instance of the problem by fixing the values of the model's parameters. Parameter values are usually provided in separate files (called *data files*), so that we can easily combine the same model with different data.

A planning problem for a fixed plan length n can be reduced to a CSP problem [9, 16]. The plan of a minimal possible length can be found by successively checking for existence of plans of lengths $n = 1, 2, 3, \ldots$, that is, by solving the corresponding sequence of CSPs until a satisfiable one is encountered. Another approach is to consider n as a CSP variable, and try to find a solution that minimizes n (that is, to solve a COP). We consider both approaches in the context of solving construction problems (Section 5).

2.4 SAT and SMT Solving

SAT [4] and SMT [3] solvers are typically used to solve problems that arise in verification of formal systems, such as hardware and software, but are also used in other areas, such as constraint solving, combinatorial design, cryptanalysis, theorem proving, optimization, program synthesis, etc. In SAT, a problem instance is expressed as a propositional formula in conjunctive normal form (CNF) which is tested for satisfiability. On the other hand, SMT problems are problems of satisfiability of first-order formulae with respect to given theories. Theories that are usually supported by modern SMT solvers are the theory of equality with uninterpreted functions, real and integer arithmetics, the theory of arrays, the theory of finite-size bitvectors, etc. [3]. Standard input languages used by SAT and SMT solvers are DIMACS CNF[2] and SMT-LIB [2], respectively.

SAT and SMT solvers are commonly used as constraint solvers [20], [19], [5], [7]. Although CSP problems might be encoded directly in the languages supported by SAT and SMT solvers, this is usually inconvenient. Fortunately, there are tools that can translate inputs given in some constraint modeling languages to the languages native for SAT and SMT. In case of the MiniZinc modeling language, there are tools that can convert a FlatZinc input into SMT-LIB format. One such tool is provided by the SMT solver MathSAT [7]. Its optimization version OptiMathSAT can understand inputs given in FlatZinc and automatically solve CSP problems expressed in such format. It is also capable of just translating the given FlatZinc instance to the SMT-LIB language, without solving. The obtained SMT-LIB output can be then given to any other SMT solver that is available, in order to solve the problem. On the other hand, there is no publicly available tool for translation of inputs in MiniZinc/FlatZinc into DIMACS CNF, to the author's knowledge.

Since planning problems can be reduced to CSPs and COPs, we can also use SAT and SMT solvers to tackle planning problems [16]. In fact, there are tools that automatically convert planning problems described in STRIPS or PDDL languages into SAT or SMT instances, and then solve them using SAT/SMT solvers [11], [22], [6]. In our work, we do not use such tools, and we develop our own CSP model of our specific planning problem using the MiniZinc language instead. This enables us to use both finite-domain solvers and SMT solvers (by translating FlatZinc to SMT-LIB, as previously described) to solve our problem.

[2] https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html

3 Geometric Setup

In this section, we consider the geometric knowledge that we rely on in this work. Most of it resembles the knowledge described in [12], so we refer the interested reader to that work. Here we deal mainly with the representation of that knowledge in the form that is suitable for expressing the construction problems as planning problems, and that will be assumed in the further text.

3.1 Representation of Geometric Knowledge

The geometric knowledge needed for solving construction problems considered in this paper can be represented by a set of geometric objects and relations between them. We consider four types of objects: points, lines, circles and angles (or, more precisely, angle measures). The sets of objects of these types that we consider in our constructions will be denoted respectively by $\mathcal{P}$, $\mathcal{L}$, $\mathcal{C}$ and $\mathcal{A}$. These sets represent the significant objects of a triangle, covered by our geometric knowledge. They are finite and their elements are enumerated in advance (i.e. the sets are fixed). The enumerated objects are the only objects that can be used in constructions. In total, we consider 31 points, 42 lines, 50 circles and 9 angles.

As we said above, the elements of $\mathcal{A}$ are not individual angles (in the sense of two half-lines with a common endpoint), but rather angle *measures*, i.e. classes of mutually congruent angles. Note that if we have constructed an angle of some measure ϕ, we can construct angles whose measures are expressible[3] in terms of ϕ. For instance, we can construct, by a straightedge and a compass, an angle whose measure is equal $\phi/2$ or $\phi + \pi/2$. This means that an element $\phi \in \mathcal{A}$ does not denote only that particular angle measure ϕ, but the *class* of measures containing all angle measures ψ such that ψ is *constructible* (by a straightedge and a compass) from ϕ and vice-versa. We denote such class of angle measures as $[\phi]$. We consider an angle measure $\phi \in \mathcal{A}$ as constructed if and only if an angle of any measure $\psi \in [\phi]$ is constructed.

The relations between the relevant geometric objects are expressed as finite relations over the sets $\mathcal{P}$, $\mathcal{L}$, $\mathcal{C}$ and $\mathcal{A}$. These relations contain only the information relevant for the search. Namely, during the search, we do not have to know exact vector ratios, homothety coefficients, or expressions relating angle measures from the same class. For the purpose of the search, we only need to know that this information is available to the user (or tool) that will execute the construction sequence once it has been found.

To further clarify this, recall the example given in Section 2.1 (the Wernick's problem (A, G, H)). The presented construction plan for this problem includes the construction of the point M_a (midpoint of the triangle side BC) based on the fact that $\overrightarrow{AM_a} = \frac{3}{2} \cdot \overrightarrow{AG}$. The search procedure does not have to know that the exact ratio is $\frac{3}{2}$. It only has to know that the point M_a can be constructed if the points A and G are given. This enables the search procedure to include such construction step in the construction plan (i.e. the step that says "construct the point M_a from the points A and G using the known vector ratio"). On the other hand, the entity (whether a

[3] Note that not all angle measures that are algebraically expressible in terms of ϕ can be constructed from ϕ by a straightedge and a compass. For instance, an angle whose measure is $\phi/3$ is not constructible in general case.

human user or a tool) that executes the generated construction plan must know that the exact ratio is $\frac{3}{2}$ in order to apply that construction step.

We consider the following relations between objects:

- *inc_point_line* $\subseteq \mathcal{P} \times \mathcal{L}$ and *inc_point_circle* $\subseteq \mathcal{P} \times \mathcal{C}$, representing the information about which points are incident with which lines and circles, respectively
- *parallel_lines* $\subseteq \mathcal{L} \times \mathcal{L}$ and *perp_lines* $\subseteq \mathcal{L} \times \mathcal{L}$, representing the information about pairs of lines that are parallel or perpendicular, respectively
- *tangent_line* $\subseteq \mathcal{C} \times \mathcal{L}$, representing the information about lines that are tangents of the related circles
- *circle_diameter* $\subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{C}$, representing the information about segments (i.e. pairs of points) that are diameters of the related circles
- *circle_center* $\subseteq \mathcal{P} \times \mathcal{C}$, representing the information about points that are centers of the related circles
- *known_ratio_triples* $\subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{P}$, representing the information about the triples of collinear points (X, Y, Z) such that the ratio $\overrightarrow{XY}/\overrightarrow{YZ}$ is known. Similarly, the relation *known_ratio_quadruples* $\subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{P} \times \mathcal{P}$ represents the information about quadruples of points (X, Y, Z, W) such that the ratio $\overrightarrow{XY}/\overrightarrow{ZW}$ is known. The exact value of the ratio is not stored in the knowledge base, since it is not relevant for the search.
- *angle_def* $\subseteq \mathcal{L} \times \mathcal{L} \times \mathcal{A}$, representing the information about the measures of the angles between the lines. A triple $(p, q, \phi) \in$ *angle_def* means that the measure ψ of the angle formed by the lines p and q is in $[\phi]$. The exact expression that relates ψ and ϕ is not relevant for the search.
- *perp_bisector* $\subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{L}$, representing the information about the perpendicular bisectors of line segments, which are given as pairs of their endpoints.
- *harmonic_quadruples* $\subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{P} \times \mathcal{P}$, containing the quadruples of points (X, Y, Z, W) such that the points X and Y are harmonic conjugates of each other with respect to the pair (Z, W).
- *locus_def* $\subseteq \mathcal{P} \times \mathcal{P} \times \mathcal{A} \times \mathcal{C}$, where a tuple $(X, Y, \phi, c) \in$ *locus_def* means that the locus of points such that the segment XY is seen at an angle whose measure ψ is in $[\phi]$ is an arc of the circle c. Again, the exact expression that relates ψ and ϕ is not relevant for the search.
- *homothety_triples* $\subseteq \mathcal{P} \times \mathcal{L} \times \mathcal{L}$, where a triple $(X, p, q) \in$ *homothety_triples* encodes that the line q is the image of the line p by homothety centered in the point X (the homothety coefficient is not relevant for the search, so it is not represented in the knowledge base).

The tuples composing the above relations are enumerated in advance, based on our knowledge about the geometric properties of our objects. Namely, since our knowledge is expressed by a finite and fixed set of definitions and lemmas (listed in [12]), our relations are also finite and fixed. The knowledge base is generated manually, by carefully replicating the knowledge given in [12]. We took a special care to ensure the consistency of the knowledge base. However, we must stress that our knowledge base is not formally verified by any means. It is only carefully reviewed by hand for errors and inconsistencies, and we strongly believe that it is error-free.

Note that our geometric knowledge establishes many different relationships between our objects, which means that most objects belong to multiple tuples in our relations. As a consequence, we usually have multiple ways to construct the same object. For instance, if the vertices A and B and the centroid G of a triangle are given, then we can construct the vertex C in (at least) two different ways:

- we can construct the point M_c (the midpoint of the side AB), and then construct the point C, having in mind that $\overrightarrow{M_cC} = 3 \cdot \overrightarrow{M_cG}$ (an illustration is given in Figure 3).
- we can construct the point M_a (the midpoint of the side BC) since we know that $\overrightarrow{AM_a} = \frac{3}{2} \cdot \overrightarrow{AG}$. Then we can construct the vertex C, having in mind that $\overrightarrow{BC} = 2 \cdot \overrightarrow{BM_a}$ (an illustration is given in Figure 4).

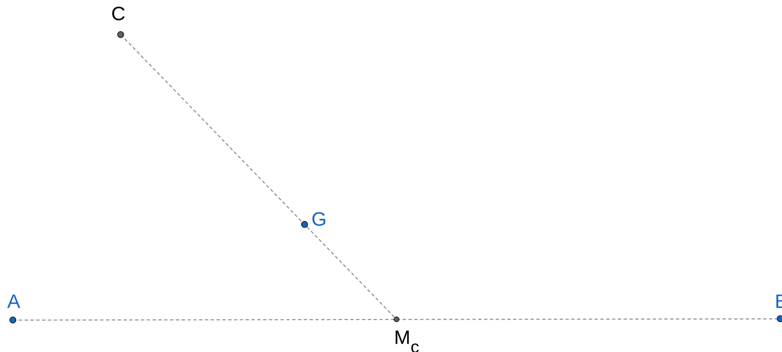

Fig. 3 The first way to construct the vertex C, given the vertices A and B and the centroid G. It holds $\overrightarrow{AM_c} = \frac{1}{2} \cdot \overrightarrow{AB}$ and $\overrightarrow{M_cC} = 3 \cdot \overrightarrow{M_cG}$.

These multiple possibilities in our constructions lead to different construction plans for the same problem.

3.2 Construction Steps

In this section, we describe types of construction steps that can be used in constructions. They are mostly the same as those used in [12], with few technical differences. In the following list, when we say that some object is *given*, we mean that it is either given initially, or it is constructed in some of the previous steps. Each construction step is also accompanied by *preconditions* specifying when it can be applied. The preconditions are expressed in terms of the relations within our knowledge base, and are used to guide the search (that is, to determine which significant objects of the triangle can be constructed from the already constructed ones, based on our knowledge).

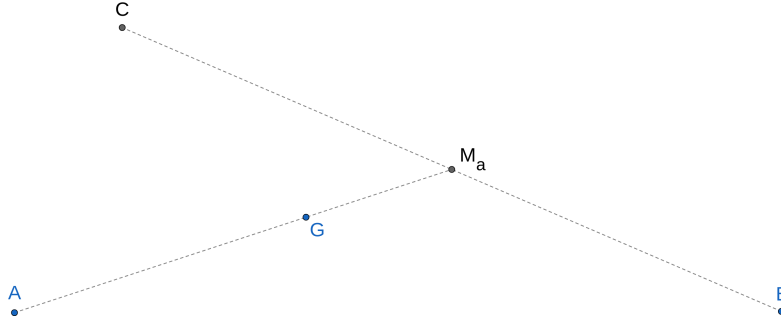

Fig. 4 The second way to construct the vertex C, given the vertices A and B and the centroid G. It holds $\overrightarrow{AM_a} = \frac{3}{2} \cdot \overrightarrow{AG}$ and $\overrightarrow{BC} = 2 \cdot \overrightarrow{BM_a}$.

In this work, the following construction step types are supported:

- *construction of the line $l \in \mathcal{L}$ through two given points $P, Q \in \mathcal{P}$*: in order to apply this step, the points P and Q must represent distinct objects from our object set[4], and the line l must be incident with both points, according to our knowledge base.
- *construction of the point $P \in \mathcal{P}$ that is the intersection of two given lines $t, s \in \mathcal{L}$*: in order to apply this step, the lines t and s must represent distinct objects from our object set, and the point P must be incident with both lines, according to our knowledge base.
- *construction of the points $P, Q \in \mathcal{P}$ that are the intersections of a given line $l \in \mathcal{L}$ and a given circle $c \in \mathcal{C}$*: in order to apply this step, the points P and Q must be incident with both the line l and the circle c, according to our knowledge base. If one of the intersections is already constructed, we can construct the other intersection.
- *construction of the circle $c \in \mathcal{C}$ whose center is a given point $P \in \mathcal{P}$, and that contains a given point $Q \in \mathcal{P}$*: in order to apply this step, the circle c must have P as its center, and must be incident with Q, according to our knowledge base.
- *construction of the points $P, Q \in \mathcal{P}$ that are the intersections of two given circles $c_1, c_2 \in \mathcal{C}$*: in order to apply this step, the points P and Q must be incident with both circles, according to our knowledge base. Again, if one of the intersections is already constructed, we can construct the other intersection.
- *construction of the circle $c \in \mathcal{C}$ over the segment PQ as its diameter, where $P, Q \in \mathcal{P}$ are given points*: in order to apply this step, the segment PQ must be a diameter of c, according to our knowledge base.

[4]This requirement is here to forbid the application of such step to construct, for instance, the line a from the points B and B, which would not make sense, since we know only one point incident with the line.

- *construction of the line $l \in \mathcal{L}$ passing through a given point $P \in \mathcal{P}$ that is perpendicular to a given line $t \in \mathcal{L}$*: in order to apply this step, the line l must be incident with the point P and perpendicular to the line t, according to our knowledge base.
- *construction of the circle $c \in \mathcal{C}$ whose center is a given point $P \in \mathcal{P}$ and a given line $l \in \mathcal{L}$ is its tangent*: in order to apply this step, the point P must be the center of the circle c, and the line l must be a tangent of c, according to our knowledge base.
- *construction of the lines $t_1, t_2 \in \mathcal{L}$ that are the tangents to a given circle $c \in \mathcal{C}$ from a given point $P \in \mathcal{P}$*: in order to apply this step, the lines t_1 and t_2 must be tangents of the circle c and incident with the given point P, according to our knowledge base. If one of the tangents is already constructed, we can construct the other tangent.
- *construction of the line $l \in \mathcal{L}$ that is the perpendicular bisector of a segment PQ, where $P, Q \in \mathcal{P}$ are given points*: in order to apply this step, the line l must be the perpendicular bisector of the segment PQ, according to our knowledge base.
- *construction of the line $l \in \mathcal{L}$ that is the homothetic image of a given line $t \in \mathcal{L}$, with a given point $P \in \mathcal{P}$ as the center of the homothety*: in order to apply this step, the line l must be the homothetic image of the line t with the point P as the center of the homothety, according to our knowledge base.
- *construction of the line $l \in \mathcal{L}$ that is parallel to a given line $t \in \mathcal{L}$ and contains a given point $P \in \mathcal{P}$*: in order to apply this step, the line l must be parallel to t and incident with P, according to our knowledge base.
- *construction of the line $l \in \mathcal{L}$ that is a side of an angle with the measure constructible from a given measure $\phi \in \mathcal{A}$, where the other side of the angle is a given line $t \in \mathcal{L}$, and a given point $P \in \mathcal{P}$ is the vertex of the angle*: in order to apply this step, the lines l and t must be incident with the point P, and the angle between l and t must be in $[\phi]$, according to our knowledge base.
- *construction of the angle measure $\phi \in \mathcal{A}$ determined by the angle between two given lines $s, t \in \mathcal{L}$*: in order to apply this step, the measure of the angle formed by the lines s and t must be in class $[\phi]$, according to our knowledge base.
- *construction of the point $X \in \mathcal{P}$ determined by two given points $Y, Z \in \mathcal{P}$ and a known vector ratio*: in order to apply this step, the vector ratio $\overrightarrow{XY}/\overrightarrow{YZ}$ must be known, according to our knowledge base.
- *construction of the point $X \in \mathcal{P}$ determined by three given points $Y, Z, W \in \mathcal{P}$ and a known vector ratio*: in order to apply this step, the vector ratio $\overrightarrow{XY}/\overrightarrow{ZW}$ must be known, according to our knowledge base.
- *construction of the point $X \in \mathcal{P}$ that is the harmonic conjugate of a given point $Y \in \mathcal{P}$ with respect to two other given points $Z, W \in \mathcal{P}$*: in order to apply this step, the points X and Y must be harmonic conjugates with respect to Z and W, according to our knowledge base.
- *construction of the circle $c \in \mathcal{C}$ containing the locus of points from which a given segment PQ ($P, Q \in \mathcal{P}$) is seen at the angle of a measure constructible from a given measure $\phi \in \mathcal{A}$*: in order to apply this step, the circle c must contain the arc from which the segment PQ is seen at the angle of a measure in $[\phi]$, according to our knowledge base.
- *construction of the point $P \in \mathcal{P}$ that is the center of the circle containing three given points $S, R, T \in \mathcal{P}$*: in order to apply this step, there must exist a circle $c \in \mathcal{C}$

incident with the points S, R, T, such that P is the center of c, according to our knowledge base (note that we do not have to construct c).

- *construction of the point $P \in \mathcal{P}$ that is the center of the circle containing two given points $R, S \in \mathcal{P}$, with a given line $l \in \mathcal{L}$ incident with P*: in order to apply this step, there must exist a circle $c \in \mathcal{C}$ incident with the points R and S, the point P must be the center of c and must be incident with l, and the line l must not be the perpendicular bisector of the segment RS, according to our knowledge base (again, we do not construct the circle c itself).

Note that the preconditions assigned to some construction step type only tell us when it is possible to apply such a step. They do not guarantee that the objects constructed by that step will indeed exist in a specific situation in a plane. That is, these preconditions should not be confused with the non-degeneracy or determination conditions for that construction step (discussed in Section 2.1). For instance, given the circumcircle kO and the line a, we want to apply the step that constructs the intersections of kO and a (the points B and C). According to the preconditions, such a step can be applied, since from our knowledge base we know that the points B and C are incident with both kO and a. On the other hand, the non-degeneracy condition is that kO and a indeed intersect, which depends on a specific situation in a plane. As said earlier, such non-degeneracy conditions are not considered during the search and are analyzed only after the construction plan is obtained (and we do not discuss them in our work).

4 Construction Problems as Planning Problems

The triangle construction problems that we consider in this paper can be naturally described as problems of automated planning:

- states correspond to the sets of constructed objects, and the initial state is the set consisting of given elements of a triangle that we want to construct (three points in case of Wernick's problems).
- actions correspond to the construction steps; the precondition for each action is that objects used as inputs in the corresponding construction step are already constructed (i.e. belong to the current state), and that the preconditions of the corresponding construction step are met (as defined in the previous section); the effect of each action is the addition of the objects constructed by the corresponding construction step to the current state.
- the goal condition is that the triangle vertices A, B and C belong to the final state.

All we need to do is to encode such a planning problem using some modeling language and then use an appropriate off-the-shelf tool to solve it. We provide two such models. The first model is developed in PDDL language. In the second model, we assume a fixed plan length, and encode the planning problem as a CSP using the MiniZinc language.[5] In the rest of this section, we describe the two models in more details.

[5]Both models are available at: https://github.com/milanbankovic/constructions/.

4.1 PDDL Model

The encoding of our planning problem in the PDDL language is straightforward. First, we have to define types of objects that we use in our model:

```
(:types
    point line circle angle - object
)
```

That is, we have four types of objects, and all are subtypes of the most general PDDL type – `object`. The objects themselves are defined by enumerating them. For instance:

```
(:objects
   ; points
   A B C O I G H Ma Mb Mc Ta Tb Tc Ha Hb Hc
   Pa Pb Pc Na Nb Nc Ppa Ppb Ppc Sa Sb Sc Tpa Tpb Tpc - point

   ; lines
   la lb lc lma lmb lmc lsa lsb lsc lha lhb lhc ... - line

   ; circles
   kO kI kMa kMb kMc ... - circle

   ; angles
   Alpha Beta Gamma ... - angle
)
```

Second, we have to define types of predicates that can be used to express facts. In our case, predicates directly correspond to our relations described in Section 3. For instance, the predicate:

```
(incident_point_circle ?p - point ?c - circle)
```

corresponds to our relation $inc_point_circle \subseteq \mathcal{P} \times \mathcal{C}$. Similar predicates (with appropriate argument types) are defined for other relations. Facts are now obtained by applying predicates to objects. For instance:

```
(incident_point_circle A kO)
```

represents the fact that the triangle vertex A is incident with the circumcircle of the triangle, denoted by kO. In addition, there is a predicate:

```
(constructed ?o - object)
```

which is applicable to all objects and is used to express the fact that some object is already constructed, or initially given.

In PDDL, states are represented as sets of facts. As we said earlier, we want our states to correspond to the sets of constructed objects, i.e. to the sets of facts of the

form `(constructed o)`, where o is an object of any of our four types. On the other hand, the facts that represent our knowledge (i.e. the tuples of our relations) must hold permanently, in all the states. The simplest way to do this is to include all these tuples in the initial state, and then to keep them in all subsequent states (i.e. not to remove any of them by actions that are applied to change the state).

The actions directly correspond to our construction steps described in Section 3. For instance:

```
(:action line_intersection
     :parameters (?l1 - line ?l2 - line ?p - point)
     :precondition
        (and
          (constructed ?l1)
          (constructed ?l2)
          (not (= ?l1 ?l2))
          (incident_point_line ?p ?l1)
          (incident_point_line ?p ?l2)
          (not (constructed ?p))
       )
     :effect
       (constructed ?p)
)
```

This action allows us to construct the point P, provided that it is the intersection of two distinct, already constructed lines l_1 and l_2. Note that we insist that the point P is not already constructed, otherwise it would be possible to apply this action even if the point P is already in the state, without any effect.

The initial state contains the facts representing the relation tuples (which are the same for any instance of Wernick's problem) and the facts representing the given three points (which depend on the specific instance of the Wernick's problem). For instance, for the problem (A, G, H) discussed in the example in Section 2.1, we would have the following encoding of the initial state:

```
(:init
  (and
    ; relation tuples
    (incident_point_line A lb)
    (incident_point_line A lc)
    ;... other relation tuples go here

    ; given points
    (constructed A)
    (constructed G)
    (constructed H)
  )
)
```

The goal is given by requiring that the vertices A, B and C belong to the final state:

```
(:goal
  (and
    (constructed A)
    (constructed B)
    (constructed C)
  )
)
```

A solution plan is represented as a sequence of actions. For instance, the output that corresponds to the solution plan for the problem (A, G, H) presented in Section 2.1 might look like this:

```
(line_through A H lha)
(known_ratio3 A H Ma)
(perpendicular_line lha Ma la)
(known_ratio3 G H O)
(circle_center_point O A kO)
(line_circle_intersections la kO B C)
```

In each action, the last argument (or the last two arguments in case of the last action) represents the constructed object that is added to the current state, and the remaining arguments are inputs.

4.2 MiniZinc Model

Modeling planning problems in MiniZinc requires a little more effort, since we have to manually encode the states and the transitions between them (using CSP variables and constraints). Let us first consider the encoding of the geometric knowledge. Each type of objects is encoded as an *enumeration type* in MiniZinc (`Point`, `Line`, `Circle` and `Angle`, respectively), and each object is represented by one enumerator of the corresponding type, e.g.:

```
Point = { A, B, C, O, I, G, H, Ma, Mb, Mc, Ta, Tb, Tc, Ha, Hb, Hc, Pa,
          Pb, Pc, Na, Nb, Nc, Ppa, Ppb, Ppc, Sa, Sb, Sc, Tpa, Tpb, Tpc
        };
```

Relations between the enumerated objects are encoded by the parameters of the model, using MiniZinc's arrays, sets and tuples. We define the following parameters in our MiniZinc model:

- incidence relations are represented by two arrays of sets, `inc_lines` and `inc_circles`, indexed by points. The set `inc_lines`[p] contains the lines incident with the point p, and the set `inc_circles`[p] contains the circles incident with the point p.

- the relations between lines are represented by two arrays of sets, `perp_lines` and `paralell_lines`, indexed by lines. The set `perp_lines`$[l]$ contains the lines perpendicular to the line l, and the set `parallel_lines`$[l]$ contains the lines parallel to the line l.
- the array of points `circle_center` indexed by circles contains information about circle centers; the array `circle_diameter` of point pairs indexed by circles contains information about circle diameters; the array `tangent_lines` of sets of lines is indexed by circles, and the set `tangent_lines`$[c]$ contains the lines that are tangents of the circle c.
- the array `known_ratio_triples` (`known_ratio_quadruples`) of point triples (quadruples) stores the information about the known vector ratios.
- the array `angle_defs` of `Line` $\times$ `Line` $\times$ `Angle` triples encodes the information about the angles between the lines.
- the array `perp_bisectors` of `Point` $\times$ `Point` $\times$ `Line` triples encodes the information about the perpendicular bisectors of line segments.
- the array `harmonic_quadruples` of point quadruples contains the harmonic conjugates.
- the array `locus_defs` of `Point` $\times$ `Point` $\times$ `Angle` $\times$ `Circle` tuples stores the information about the loci of points.
- the array `homothety_triples` of `Point` $\times$ `Line` $\times$ `Line` triples stores the information about lines that are homothetic images of one another.

In order to encode the planning problem as a CSP, we must fix the plan length n (or, at least, its upper limit) in advance. Recall that the plan length n corresponds to the number of construction steps. Let S_0 be the initial state, and let S_i be the state after the ith step ($i \in \{1, \ldots, n\}$). This means that we have $n + 1$ states in total. To encode these states, we introduce arrays of set variables `known_points`, `known_lines`, `known_circles` and `known_angles`, where, for instance, `known_points`$[i]$ ($i \in \{0, \ldots, n\}$) denotes the set of points belonging to the state S_i (similarly for other arrays). The initial state S_0 is given by appropriate constraints (for instance `known_points`$[0] = \{A, G, H\}$, if we want to solve the Wernick's problem (A, G, H), discussed in the example in Section 2.1).

To encode the plan, we define the enumeration type `ConsType`, with one enumerator for each supported construction step type:

```
enum ConsType =  { LineThrough,
                   LineIntersect,
                   ...
                   Locus,
                   CenterThreePoints,
                   CenterTwoPointsAndLine
                 };
```

We also define the array `construct` of variables of type `ConsType` (with indices in $\{1, \ldots, n\}$) encoding actions used in each step. For each step, we also need additional information to fully determine the actual construction (for instance, if we choose to

construct the intersection of two lines, we must also choose the lines that we want to intersect). For this reason, we also introduce additional two-dimensional arrays of variables: for instance, `points`$[i][j]$ denotes the jth point used in the ith construction step (similarly we have `lines`$[i][j]$, `circles`$[i][j]$ and `angles`$[i][j]$).

Finally, to glue the whole plan together, we must add the constraints that connect the state variables in the successive states, depending on the chosen action in the corresponding step. This must be done for each $i \in \{1, \ldots, n\}$, and we use MiniZinc's finite universal quantification for that purpose:

```
constraint forall(i in 1..n)
(
   construct[i] = LineIntersect ->
       % Precondition
       (lines[i,1] in known_lines[i-1] /\
        lines[i,2] in known_lines[i-1] /\
        lines[i,1] != lines[i,2] /\
        not (lines[i,1] in parallel_lines[lines[i,2]]) /\
        lines[i,1] in inc_lines[points[i,1]] /\
        lines[i,2] in inc_lines[points[i,1]] /\
        not (points[i,1] in known_points[i-1]) /\
       % Effects
        known_points[i] = known_points[i-1] union { points[i,1] } /\
        known_lines[i] = known_lines[i-1] /\
        known_circles[i] = known_circles[i-1] /\
        known_angles[i] = known_angles[i - 1]
       )
);
```

That is, for all $i \in \{1, \ldots, n\}$, if the chosen operator is `LineIntersect` (constructing the intersection of two lines), then the chosen two lines `lines`$[i, 1]$ and `lines`$[i, 2]$ must belong to the current state S_{i-1} (i.e. they must have been already constructed), they must be distinct and not parallel. Also, the chosen point `points`$[i, 1]$ must belong to both chosen lines (i.e. it must be their intersection), and it must not belong to the current state (we do not want to construct a point that is already constructed). If all these preconditions are met, then the effect is that the set `known_points`$[i]$ is obtained by adding the intersection point `points`$[i, 1]$ to the set `known_points`$[i - 1]$ (the sets of lines, circles and angles remain the same). Similar constraints are defined for all other types of construction steps.

The goal is encoded simply by adding the constraint that requires that the vertices A, B and C belong to the set `known_points`$[n]$:

```
{ A, B, C } subset known_points[n];
```

The output that corresponds to the solution plan for the problem (A, G, H) presented in Section 2.1 might look like this:

```
Solved in 6 steps;
1. line through points A and H --> ha
2. point with known ratio (3 points): A, G --> Ma
3. point with known ratio (3 points): G, H --> O
4. line perpendicular to line ha through point Ma --> a
5. circle with center O through point A --> kO
6. both intersects of line a and circle kO --> B,C
```

As it can be seen, the output may be fairly descriptive, since MiniZinc allows a user to program an arbitrary output format.

Note that n is also a parameter of the MiniZinc model. As said before, we can search for a plan of the minimal possible length by successively solving the CSPs for $n = 1, 2, \ldots$ until one of them is satisfiable. Another option is to consider n as a variable (with some finite domain, given in advance), and to solve one COP that minimizes n. We discuss both approaches in the next section.

5 Evaluation

The models described in the previous section are evaluated on 74 solvable instances from Wernick's set [21]. The experiments were performed on a computer with 3.1GHz processor and 8Gb of RAM. The goal was to find construction plans of minimal possible lengths for each of the Wernick's instances.

For evaluation of the PDDL model, we used `fast-downward`[6] planning system. The system allows different algorithm configurations, and we used the one based on the A^* algorithm with the *landmark-cut* heuristic [10]. The system finds the plans of minimal possible lengths by itself, that is, we rely on its built-in optimization capabilities, without any additional effort on our side.

When the MiniZinc model is concerned, we used official MiniZinc distribution[7] (version 2.7.2) for developing and testing the model, and also for translating the problem instances to FlatZinc. For solving the obtained FlatZinc instances, we used the following solvers:

- *finite-domain constraint solvers*: we experimented with multiple constraint solvers provided within MiniZinc distribution. By far the best results were obtained by the solver `chuffed`[8] (version 0.11.0). It is a finite-domain constraint solver based on so-called *lazy clause generation* [15] (that is, using a SAT solver as the search engine). Therefore, we present only the results obtained by `chuffed`.
- *SMT solvers*: we used `optimathsat`[9] SMT solver (version 1.7.3) with optimization capabilities. This solver provides direct support for FlatZinc inputs, which are internally converted to SMT formulae and solved. A FlatZinc instance can be converted to an SMT problem either in the theory of linear arithmetics, or in the theory of finite-length bitvectors. Our preliminary experiments showed a clear superiority of the bitvector theory, so we present only the results obtained in that way. We have

[6] https://www.fast-downward.org/
[7] https://www.minizinc.org/software.html
[8] https://github.com/chuffed/chuffed
[9] https://optimathsat.disi.unitn.it/

also experimented with `Z3`[10] SMT solver (version 4.13.0), also with optimization capabilities. Since it does not support FlatZinc inputs directly, we used `fzn2omt`[11] helper tool that utilizes `optimathsat` to convert FlatZinc instances to SMT-LIB format (again in the theory of bitvectors). However, the results obtained by `Z3` were significantly worse than those obtained by `optimathsat`, so we will not present them here.

In order to look for plans of minimal possible lengths, we first had to fix an upper limit for the plan length n. We denote this limit by $maxSteps$ in further text. Such limit is usually obtained by heuristic methods, using the knowledge about the problem being solved. In our case, we already knew that the maximal length of construction sequences obtained by ArgoTriCS [12] on the Wernick's set of problems was 16. Since our models use virtually the same geometric knowledge and the same set of possible construction steps, we assumed that the maximal plan length in our case should have a similar value. Therefore, we took the value $maxSteps = 20$ as the upper limit for our experiments. We used the following two setups for finding plans of minimal lengths:

- *linear setup*: for each of the problems, we successively look for plans of length $n = 1, 2, 3, \ldots, maxSteps$, and stop when we encounter a satisfiable CSP, or when the upper limit $maxSteps$ is exceeded. Note that in this setup the value of $maxSteps$ does not affect the solving time for problems that our model can solve (that is, using a greater value of $maxSteps$ would not slow down the search). It only affects those instances that are unsolvable by our model (because it affects the number of CSPs that we have to try to solve before we give up).
- *minimization setup*: we reformulate our model such that the plan length n is not fixed. Instead, n is a variable with a domain $\{1, \ldots, maxSteps\}$ and we are trying to minimize the value of n (that is, we are solving only one constrained optimization problem).[12] In this setup, the value of $maxSteps$ influences the size of the COP instance, thus affecting the solving time for all instances, even for those that can be solved in a small number of steps.

As said earlier, our method concerns only the search, and does not handle proving the correctness of the obtained solutions. Therefore, formally speaking, we cannot claim that the obtained construction plans are correct. However, in order to further support the results of this study, we manually reviewed all the obtained construction plans and, with a fair degree of confidence, we believe that they are correct. The task of generating formal correctness proofs may be left to other tools, but this is out of the scope of this paper.

In Table 1, we provide the main results of our evaluation. The table includes results both for the PDDL model (`fast-downward`) and the MiniZinc model (both `chuffed` and `optimathsat` solvers using two described setups). We also compared our approach to the results obtained by the ArgoTriCS dedicated triangle construction

[10] https://github.com/Z3Prover/z3

[11] https://github.com/PatrickTrentin88/fzn2omt

[12] Note that most constraint solvers perform minimization by starting at the upper bound and then repeatedly tightening that bound until they encounter unsatisfiability. This means that, in some sense, this strategy is the opposite of the previously described linear setup.

solver developed by Marinković [12]. ArgoTriCS is implemented in the Prolog programming language. As we already mentioned, it uses a very similar knowledge base and an almost identical set of available construction steps.

Table 1 Overall results for different solvers and setups, compared to ArgoTriCS. Times are given in seconds

Solver	# solved	Avg. time	Med. time	Avg. on solved	Std. dev.
`fast-downward`	63	1.23	1.20	1.23	0.09
`chuffed` (lin)	63	16.05	8.58	8.09	19.79
`chuffed` (min)	63	6.65	6.51	6.73	0.69
`optimathsat` (lin)	63	53.26	26.70	28.12	67.15
`optimathsat` (min)	63	25.69	22.66	27.03	9.63
ArgoTriCS	65	54.5	21.6	54.4	140.9

Note that the choice of the solver (and the setup) does not affect how many problems from the Wernick's list will be solved, since this depends only on the geometric knowledge that is compiled into our models. In total, we managed to solve 63 of 74 problems (for the remaining 11 problems, the solvers reported unsatisfiability, i.e. nonexistence of a plan). On the other hand, ArgoTriCS solved 2 problems more (which indicates that we missed to incorporate some of the objects and lemmas known to ArgoTriCS to our models).

When the efficiency is concerned, the best results are clearly obtained by `fast-downward` planner, requiring only 1.23 seconds per instance, on average. The second best option (`chuffed` with minimization setup) was over five times slower on average. This makes PDDL-based approach the best choice, both concerning the effort needed for modeling (recall that PDDL encoding was very simple and straightforward) and the time needed for finding a solution plan.

When the MiniZinc model is concerned, the best average solving time is obtained by the `chuffed` solver. The SMT-based approach turned out to be slower, even with the bitvector arithmetics. We do not know the exact reasons for that, but we must be aware of an inherent bias in favor of the `chuffed` solver, since the MiniZinc/FlatZinc is its native language. On the other hand, SMT solvers require conversion of the FlatZinc inputs to their native language, probably losing some compactness in expressing the appropriate constraints.

Compared to ArgoTriCS, when solving times are concerned, it turned out that all variants of our approach performed significantly better (except the `optimathsat` with the linear setup, which was comparable to ArgoTriCS). This confirms our assumption that using highly optimized, state-of-the-art tools specialized in solving combinatorial search problems is often a better choice then developing our own algorithm for such purposes (at least, it turned out to be the case when solving triangle construction problems).

Let us also compare the performance of different setups for the MiniZinc-based approach. Table 1 shows that the minimization setup was a better choice in case of both the `chuffed` and the `optimathsat` solvers. However, as we mentioned earlier,

the average solving time in this setup greatly depends on the choice for the maximal possible value of n. The results shown in Table 1 are obtained for $maxSteps = 20$. We also experimented with some greater values. For instance, for $maxSteps = 30$ the average solving time was about 10 seconds for the `chuffed` solver, and about 35 seconds for the `optimathsat` solver (of course, the number of solved problems remained the same). For $maxSteps = 40$, the `chuffed` solver required about 15 seconds on average, and the `optimathsat` solver required about 46 seconds. Compared to ArgoTriCS, this is still a better performance. This means that we would obtain fair results even if we overestimated the upper bound for the plan length (at least to some extent). Note that the maximal length of all minimal plan lengths found by our model on the Wernick's set of problems was 11.

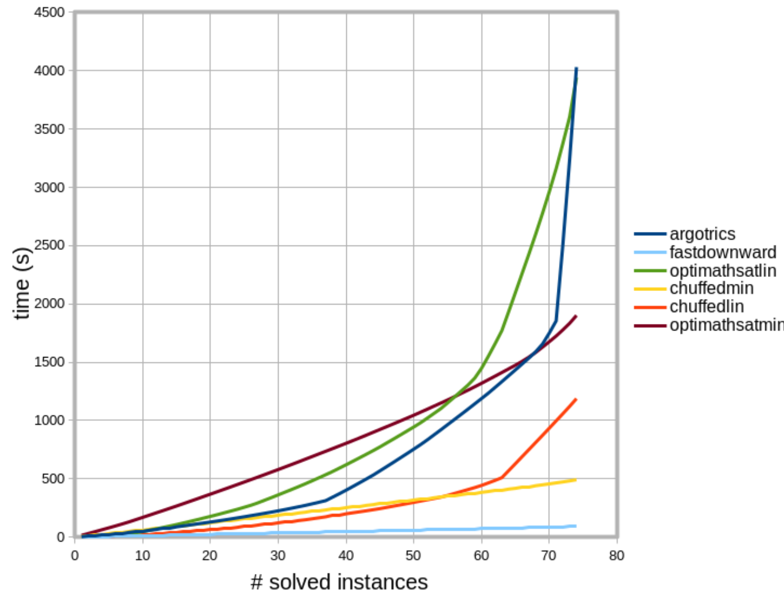

Fig. 5 Survival plot for all solvers and setups, compared to ArgoTriCS. Times are given in seconds

Figure 5, shows the survival plot for all solvers and setups, compared to ArgoTriCS. Notice that the graphs for `fast-downward` and the minimization setups are nearly linear. This indicates that most of the instances required a similar amount of time, even those that we were not able to solve. This observation is also supported by values of standard deviations (Table 1) which are significantly below the average solving times. On the other hand, for the linear setups, unsolved instances consumed much more time than average, causing the non-linearity of the survival graph (this can be confirmed by comparing the average solving times on solved instances, with the average solving times on all instances for both solvers with the linear setup in Table 1). This was expected for the linear setup, since it had to reach the upper limit $maxSteps$ in order to report the failure. In case of the ArgoTriCS solver, the non-linearity that can

be seen in the survival plot is mostly the consequence of only three instances that ArgoTriCS managed to solve, but for some reason required a lot of time (726, 660 and 789 seconds).

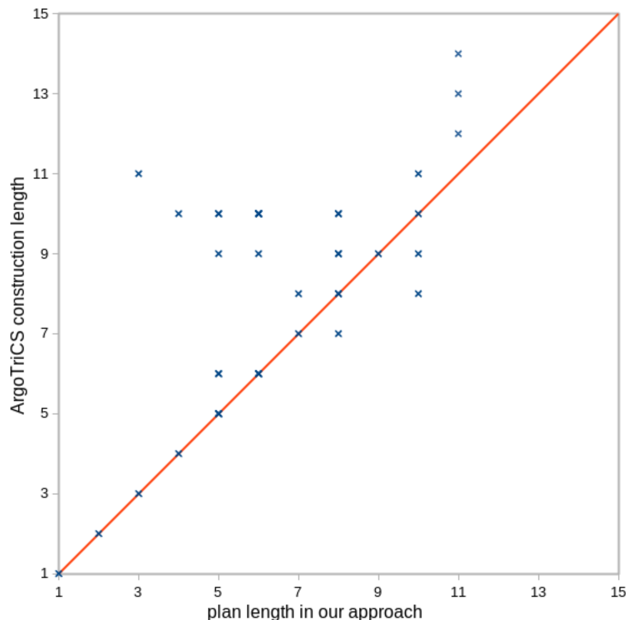

Fig. 6 A per-instance comparison of construction plan lengths between ArgoTriCS and our approach

The final comparison between ArgoTriCS and our approach concerns the lengths of the obtained construction plans. The average plan length found by our approach was 6.3, and the maximum was 11 (again, this does not depend on the chosen solver or setup). On the other hand, the average number of steps in ArgoTriCS's construction plans was 7.5 (maximum was 16). Notice that these numbers are comparable, since the sets of available construction steps in both systems are almost identical. A more detailed, per-instance comparison is shown in Figure 6. The plot clearly confirms that our approach is by far superior when finding the shortest construction plans is concerned. However, for the sake of fairness, we must stress that ArgoTriCS was not designed with that optimization in mind, that is, it does not even search for the shortest construction plans. We guess that such a capability could be integrated in ArgoTriCS, but with much more effort, since it would have to be manually implemented in Prolog (just like the search itself). On the other hand, in our approach, we rely on the built-in capabilities of automated planners and constraint solvers to solve optimization problems efficiently, imposing the minimal possible effort on our side.

The main benefit of finding shorter construction plans is that such plans tend to be simpler and easier to understand, which is especially important in educational applications. For instance, consider the instance of Wernick's problem where the vertices

A and B and the orthocenter H are given. The plan obtained by our model consists of the following construction steps:

1. construct the line through the points A and B (the line c)
2. construct the line through the points A and H (the altitude h_a)
3. construct the line through the point B perpendicular to the line h_a (this is the line a containing the side BC of the triangle)
4. construct the line through the point H perpendicular to the line c (the altitude h_c)
5. construct the point intersection of the lines a and h_c (this is the vertex C)

The construction plan consists of 5 steps. The corresponding solution is illustrated in Figure 7.

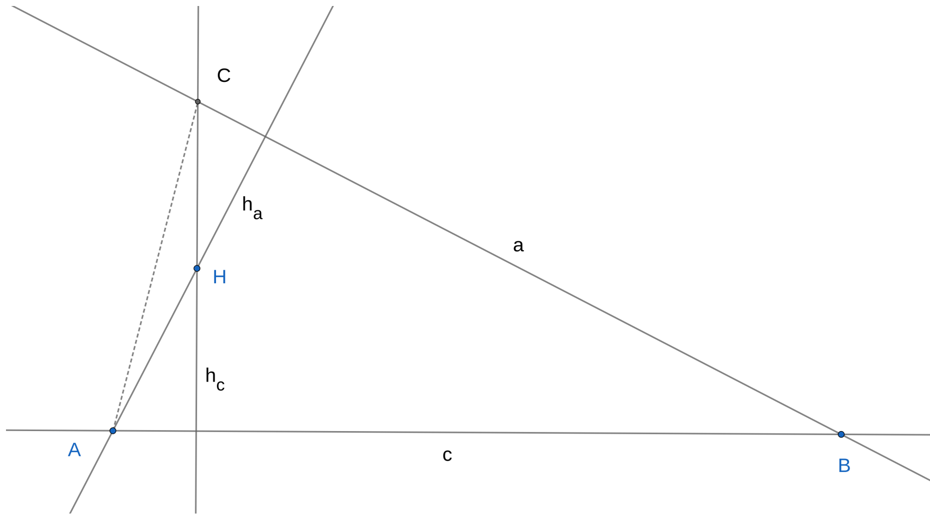

Fig. 7 An illustration of the solution for the problem (A, B, H) obtained by our model. Lines a and h_a are perpendicular, and the same holds for lines c and h_c. The dashed line is not constructed.

On the other hand, ArgoTriCS finds the following construction plan:

1. construct the midpoint of the segment AB (point M_c)
2. construct the line through the points A and H (the altitude h_a)
3. construct the line through the points B and H (the altitude h_b)
4. construct the circle $k(M_c, A)$ centered in M_c and passing through A
5. construct the other intersection (distinct from A) of the circle $k(M_c, A)$ and the line h_a (this is the foot H_a of the altitude h_a).
6. construct the line through the points B and H_a (the line a that contains the triangle side BC)
7. construct the other intersection (distinct from B) of the circle $k(M_c, A)$ and the line h_b (this is the foot H_b of the altitude h_b)
8. construct the line through the points A and H_b (the line b that contains the triangle side AC)

9. construct the point intersection of the lines a and b (this is the vertex C)

This plan consists of 9 steps. The corresponding solution is given in Figure 8. Being almost twice as long as the plan obtained by our model, this sequence is much harder to follow. Additionally, the drawing in Figure 8 is more difficult to understand, as it includes a greater number of constructed objects.

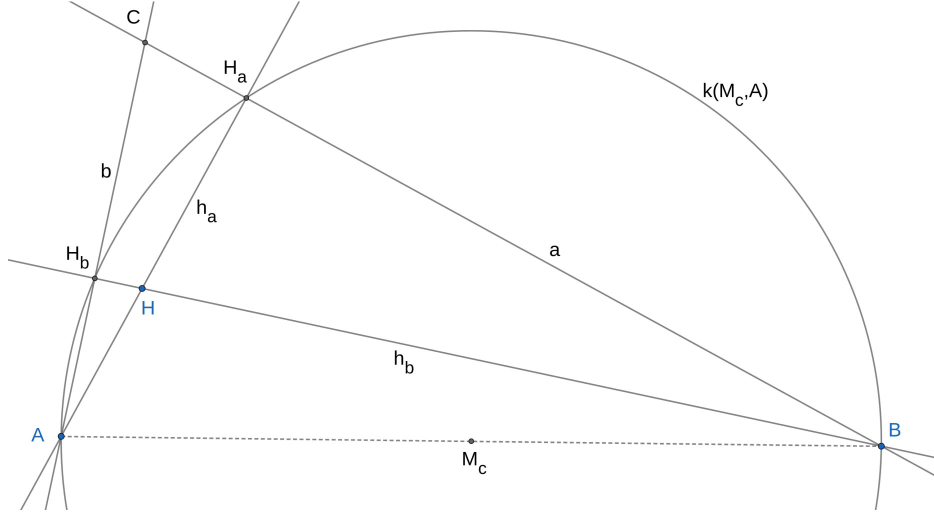

Fig. 8 An illustration of the solution for the problem (A, B, H) obtained by ArgoTriCS. The dashed line is not constructed.

6 Conclusions and Further Work

In this paper we presented and evaluated a method for automated solving of triangle construction problems based on automated planning. The first approach was to describe our planning problem in the PDDL language and solve it using an off-the-shelf automated planner. The second approach was to convert the planning problem to a CSP (or COP) problem and solve it using a constraint solver (we experimented both with finite-domain solvers and with SMT solvers). We compared our method to the state-of-the-art dedicated triangle construction solver ArgoTriCS, developed in the Prolog programming language. We advocate that our approach has two important advantages. First, our approach is much simpler to implement, since we rely on powerful state-of-the-art tools which can efficiently do the search for us, and we may focus only on modeling. On the other side, in the ArgoTriCS solver the search is implemented by hand, in more than 500 lines of code. Second, we can easily employ the built-in optimization capabilities of modern planners and solvers to search for the shortest possible construction plans, while implementing such functionality in ArgoTriCS would require much more effort.

We evaluated our approach on 74 solvable problems from the Wernick's list. The results showed that our approach is superior to ArgoTriCS when the average solving

time is concerned. In addition, our approach often finds shorter construction plans, due to built-in optimization capability which is missing in ArgoTriCS. The benefit of shorter construction plans is that they tend to be simpler and easier to follow, which is especially important in educational applications.

The work presented in this paper is mainly a case study of a particular class of construction problems. However, our approach can be adapted to support other kinds of construction problems. This can be done by identifying the relevant geometric knowledge needed for solving such construction problems and incorporating it into the model. Such research is left for further work.

Declarations

Funding

This work was partially supported by the Serbian Ministry of Science grant 174021.

Competing interests

The author has no competing interests to declare that are relevant to the content of this article.

Acknowledgements

This version of the article has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/s10472-025-09972-y. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

References

[1] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins Sri, Anthony Barrett, Dave Christianson, et al. PDDL – the planning domain definition language. *Technical Report, Tech. Rep.*, 1998.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[3] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, chapter 26, pages 825–885. IOS Press, 2009.

[4] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[5] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17(3):273–303, 2012.

[6] Michael Cashmore, Daniele Magazzeni, and Parisa Zehtabi. Planning for hybrid systems via satisfiability modulo theories. *Journal of Artificial Intelligence Research*, 67:235–283, 2020.

[7] Francesco Contaldo, Patrick Trentin, and Roberto Sebastiani. From minizinc to optimization modulo theories, and back (extended version). *arXiv preprint arXiv:1912.01476*, 2019.

[8] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[9] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[10] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: what's the difference anyway? In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 19, pages 162–169, 2009.

[11] Henry Kautz, Bart Selman, and Jörg Hoffmann. Satplan: Planning as satisfiability. In *5th international planning competition*, volume 20, page 156, 2006.

[12] Vesna Marinković. ArgoTriCS–automated triangle construction solver. *Journal of Experimental & Theoretical Artificial Intelligence*, 29(2):247–271, 2017.

[13] Milan Banković. Automation of Triangle Ruler-and-Compass Constructions Using Constraint Solvers. In Pedro Quaresma and Zoltán Kovács, editors, *Proceedings 14th International Conference on Automated Deduction in Geometry, ADG 2023, Belgrade, Serbia, 20-22th September 2023*, volume 398 of *EPTCS*, 2024.

[14] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming–CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings 13*, pages 529–543. Springer, 2007.

[15] Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[16] Jussi Rintanen. Planning and SAT. *Handbook of Satisfiability*, 185:483–504, 2009.

[17] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[18] Pascal Schreck, Pascal Mathis, Vesna Marinkovic, and Predrag Janicic. Wernick's list: a final update. In *Forum Geometricorum*, volume 16, pages 69–80, 2016.

[19] Mirko Stojadinović and Filip Marić. meSAT: multiple encodings of CSP to SAT. *Constraints*, pages 1–24, 2014.

[20] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[21] William Wernick. Triangle constructions with three located points. *Mathematics*

Magazine, 55(4):227–230, 1982.

[22] Zhao Xing, Yixin Chen, and Weixiong Zhang. Maxplan: Optimal planning by decomposed satisfiability and backward reduction. In *Proceedings of the 5th International Planning Competition, International Conference on Automated Planning and Scheduling*, pages 53–56. Citeseer, 2006.