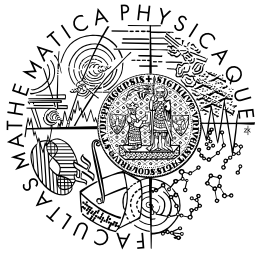


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jaroslav Macke

Learning to solve geometric construction problems from images

Department of Software and Computer Science Education

Supervisor of the master thesis: Dr. Ing. Josef Šivic
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisors Dr. Ing. Josef Šivic and Mgr. Jiří Sedlář Ph.D. This thesis would not be possible without theirs guidance, patience and their enthusiasm.
Also I would like to thank Mgr. Miroslav Olšák Ph.D. for sharing and explanation of his Euclida code.
Most importantly I would like to thank my parents and family for their support.

Title: Learning to solve geometric construction problems from images

Author: Bc. Jaroslav Macke

Department: Department of Software and Computer Science Education

Supervisor: Dr. Ing. Josef Šivic, Czech Institute of Informatics, Robotics and Cybernetics (CIIRC CTU)

Consultants: Mgr. Jiří Sedlář, Ph.D., Mgr. Miroslav Olšák, Ph.D., Mgr. Josef Urban, Ph.D.

Abstract: Geometric constructions using ruler and compass are being solved for thousands of years. Humans are capable of solving these problems without explicit knowledge of the analytical models of geometric primitives present in the scene. On the other hand, most methods for solving these problems on a computer require an analytical model. In this thesis, we introduce a method for solving geometrical constructions with access only to the image of the given geometric construction. The method utilizes Mask R-CNN, a convolutional neural network for detection and segmentation of objects in images and videos. Outputs of the Mask R-CNN are masks and bounding boxes with class labels of detected objects in the input image. In this work, we employ and adapt the Mask R-CNN architecture to solve geometric construction problems from image input. We create a process for computing geometric construction steps from masks obtained from Mask R-CNN and describe how to train the Mask R-CNN model to solve geometric construction problems. However, solving geometric problems this way is challenging, as we have to deal with object detection and construction ambiguity. There is possibly an infinite number of ways to solve a geometric construction problem. Furthermore, the method should be able to solve problems not seen during the training. To solve unseen construction problems, we develop a tree search procedure that searches the space of hypotheses provided by the Mask R-CNN model. We describe multiple components of this model and experimentally demonstrate their benefits. As experiments show, our method can learn constructions of multiple problems with high accuracy. When the geometric problem is seen at training time, the proposed approach learns to solve all 68 geometric construction problems from the first six level packs of the geometric game Euclidea with an average accuracy of 92%. The proposed approach can also solve new geometric problems unseen at training. In this significantly harder set-up, it successfully solves 31 out of these 68 geometric problems. The implementation of our method is available at `https://github.com/mackej/Learning-to-solve-geometric-construction-problems-from-images`

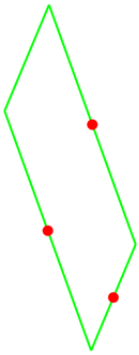
Keywords: computer vision, visual recognition, automatic geometric reasoning, solving geometric construction problems

Contents

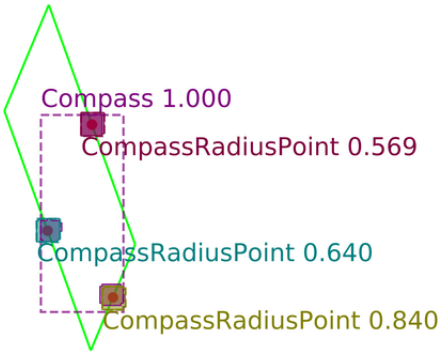
1. Introduction

1.1 Goal

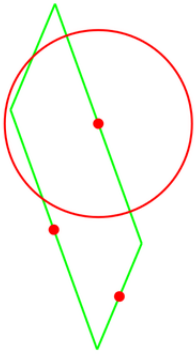
The goal of this thesis is to create a solver for geometric construction problems. The solver may use only a ruler, compass and tools representing a sequence of ruler and compass usages, such as the Perpendicular Bisector tool or the Parallel tool. The goal is to develop a machine learning model based on a convolutional neural network that will be able to predict the next step of the construction. Input for this model is an image of the scene containing the initial state and the goal. Outputs of this model are masks and bounding boxes of detected objects. Those masks have to be processed into actions representing construction steps. We can see examples of inputs and outputs in Table 1.1. This model should be able to solve as many geometric construction problems as possible. Furthermore, the model should also be able to solve problems for which it was not trained. Finally, the goal is to test the model accuracy on the problems that were both seen and unseen during the training. To test and train our model, we use geometric problems from Euclidea, which is an online construction game.



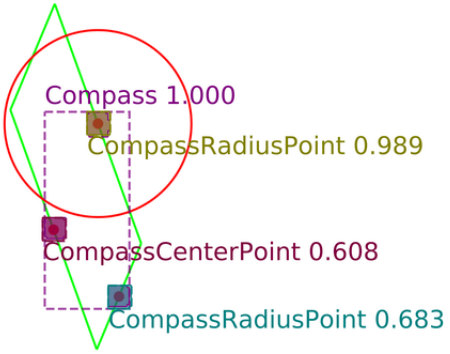
a) Level definition: Construct a parallelogram given three of the midpoints. The red denotes the current state of the construction and the green color denotes the remaining goal.



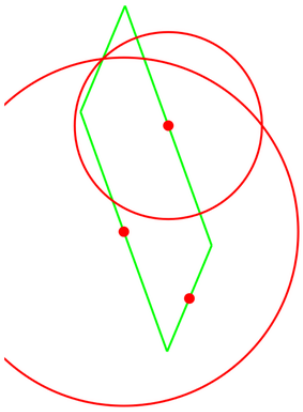
b) Prediction of the Mask R-CNN model for step 1. In the figure, we can see a prediction of the Compass tool. Each detection has a bounding box highlighted with a dashed rectangle and mask, which is filled with the same color as the corresponding bounding box. Note that the Compass tool should have two radius points and one center point but there is missing detection of the center point. In this situation can be any radius point used as the center point. Hence it is randomly chosen. The green color denotes the goal and the red denotes the current state of the construction.



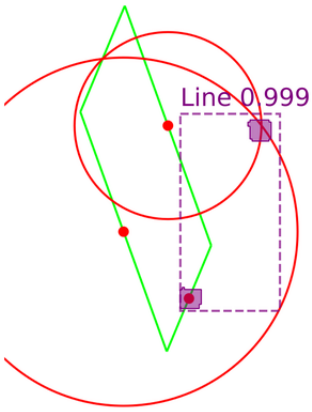
c) Step 1. In the figure, we can see that the Compass tool created a circle.



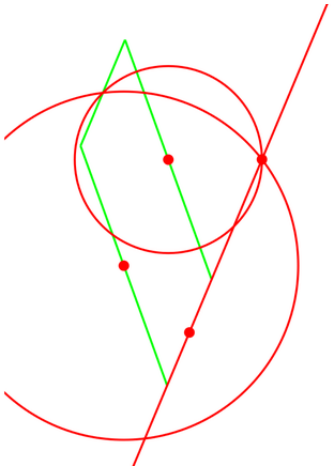
d) Prediction for step 2. We can see another detection of the Compass tool. However, the difference is that the circle center is now specified.



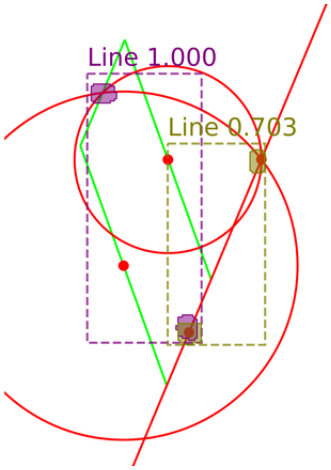
e) Step 2.



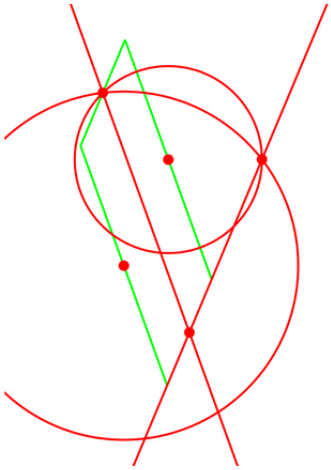
f) Prediction for step 3. Based on this prediction a line will be constructed.



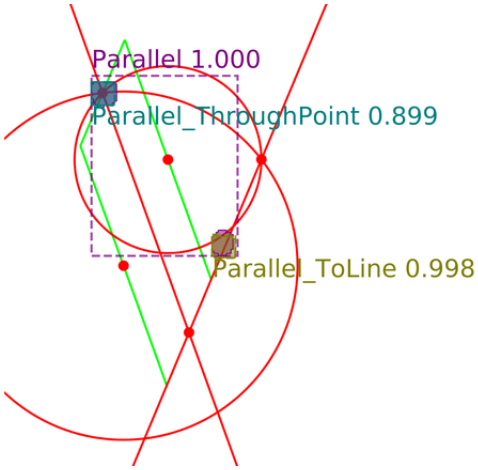
g) Step 3. The line constructed part of the goal, one side of the parallelogram.



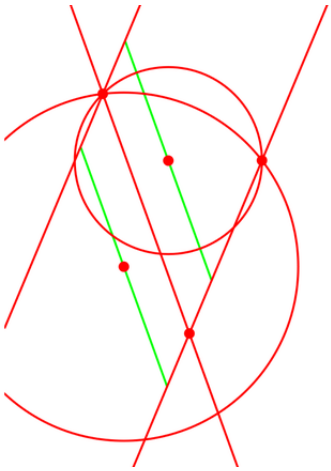
h) Prediction for step 4. Based on this prediction, a line will be constructed. Note that there are two line tool predictions. However, one of these prediction has a significantly lower Mask R-CNN score.



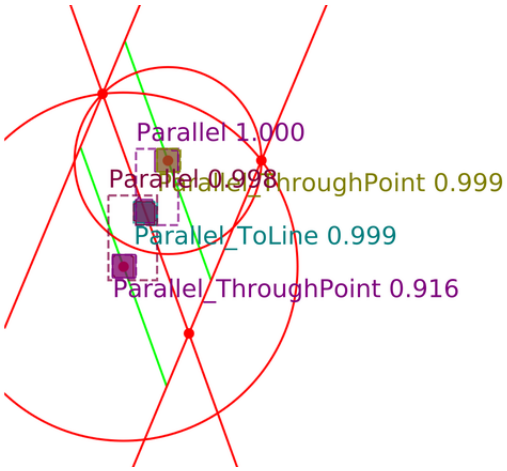
i) Step 4.



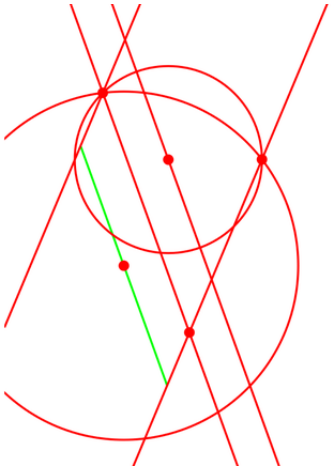
j) Prediction for step 5. Based on this prediction a parallel line will be constructed.

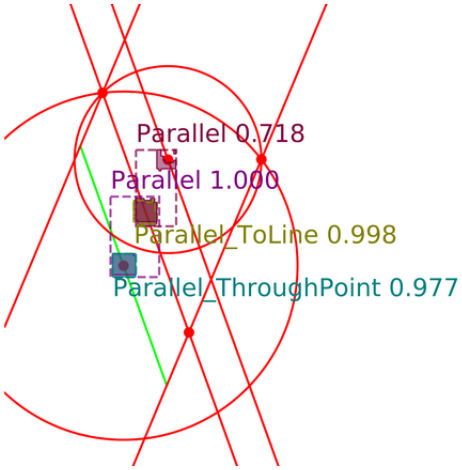


k) Step 5. The parallel line constructed part of the goal, another side of the parallelogram.

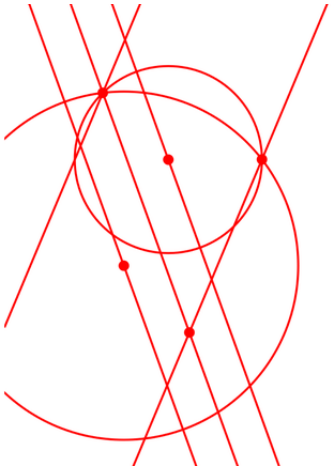


l) Prediction for step 6. Based on this prediction, a parallel line will be constructed. Note that there are two predictions of the parallel tool. Both of these parallels have to be constructed in order to finish the goal. In this step, detection with a higher Mask R-CNN score will be constructed.





m) Step 6. The parallel line constructed another part of the goal, one side of the parallelogram.



o) step 7. The parallel line constructed the last part of the goal. Hence the geometric problem is successfully constructed.

n) Prediction for step 7. Based on this prediction, a parallel line will be constructed. Both predictions are still present. However, the prediction used in the previous step has a lower Mask R-CNN score. Hence, it is not used now.

Table 1.1: Example construction of level *Zeta-12*: Parallelogram by 3 midpoints. The table contains 7 steps of the construction. Each step contains current progress on the left and Mask R-CNN prediction for a new step on the right. The green color denotes the goal and the red denotes the current state of the construction. Other colors mark prediction masks, bounding boxes, classes and scores for each detected object.

1.2 Motivation

The first motivation for this work is to answer whether geometric constructions can be solved given only the image input. Most existing methods solve geometric constructions with knowledge of the analytical equitations behind each geometric primitive in the geometric problem.

The second motivation is to develop a method that can solve geometric construction problems similar to how humans solve them. Since humans are capable of solving geometric problems without knowledge of the analytical model. Also there might be a situation, when we have only a sketch that is an approximation of a geometrical problem. In this situation, we are forced to use only image data about the problem.

The third motivation is a step towards developing an automatic geometric theorem prover. We will show that our learned recognizer is, to some extent, transferable across different constructions, and hence opens up the possibility of using similar models as hypothesis generators. This hypothesis generator can be used as an assistant for a human solver or a possible reinforcement learning method

that will learn to combine proposed hypotheses.

1.3 Why is it difficult?

In the course of the thesis we have to deal with four major problems: problem variability, construction ambiguity, object detection and training data generation. Figure 1.1 is an illustration of the problem variability. The same geometric problem can have different variants, e.g. different scale, rotation or different position of individual geometric primitives. Furthermore, each problem can be solved in an infinite number of ways. The most straightforward way to solve these problems is an exhaustive search of the construction space. The majority of existing methods use an analytical model to generate the next steps quickly. However, even with smart heuristics, a complete search of more complex problems can take up to days or even months of searching. Nevertheless, our goal is to play construction game Euclidea, so we aim to have a reasonable reaction time, but exhaustive search is often too slow.

On top of that, we have to deal with object detection because only data about the geometric problem our method receives as input is an image. To recognize individual geometric primitives and their relative positions, part of our approach is object detection architecture. However, object detection architecture, such as Mask R-CNN, requires large amount of annotated data and computing time to be properly trained. Furthermore, training data have to be constructions that can be solved based on the image data. To train a model that can solve a geometric problem, we have to generate training data, which are constructions of multiple different configurations of the given problem. While creating new configurations of a problem, we have to create well-defined variants that are solvable. Problem variability, detection and data generation can be challenging apart, but we have to deal with them at once.

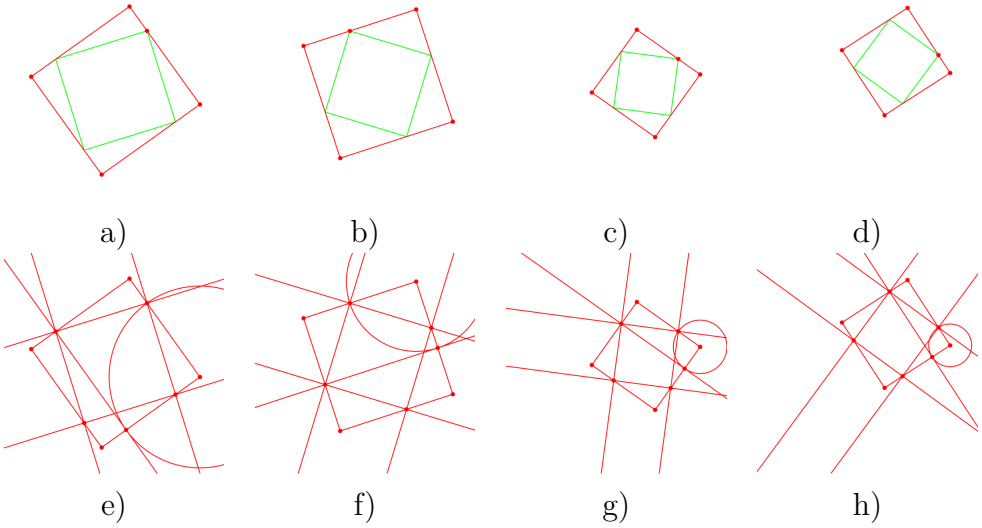


Figure 1.1: Four instances of the same geometric problem: Inscribe a square in the square when a vertex is given. For each instance of the problem the figure contains the definition (top row) and the corresponding solution (bottom row). Red color indicates the current state and green the remaining goal.

1.4 Related work

Most geometric construction solvers utilize the analytical model for solving the problems. In our setup, we aim to find a solution to problems that we know are solvable. However, the focus of the research is also to prove or disprove the existence of a solution to a given problem. This proving process is also known as automated theorem proving (ATP). Methods of proving can be an exhaustive search for a solution [1], by deriving new facts based on well-know properties in the problem or a combination of both [2]. ATPs can also be solved with resolution theorem provers and a coherent logic theorem provers [3]. Alternatively, there are ATPs that can answer SAT-questions about geometric problems [4]. For example, a question like: Based on known facts, are given lines perpendicular or not? On the other hand, some complex problems, such as Kepler conjecture, require assistance from ATP to be properly proven [5, 6].
In this thesis we combine ATP with an object detection architecture. One of the well-known architectures is the Mask R-CNN [7], which detects mask and bounding boxes in images and videos. This architecture is also used for pose estimation [8, 9] or to estimate 3D motion and forces [10].

1.5 Outline

In Chapter 2, we describe our implementation of Euclidea, an online construction game with an interface matching the desired agent. This chapter also describes how to generate new configurations of geometric problems used as training data for supervised learning. Then in Chapter 3, we analyze the difficulty of the problem by estimating the branching factor of the exhaustive tree search. In Chapter 4, we introduce a new approach based on Mask R-CNN as our primary model for supervised learning approach for the geometric construction problems.

Then in Chapter 5, we analyze the performance of Mask R-CNN models on the problems that were not seen during the training. To do so, we also describe the hypothesis tree search to search hypotheses obtained from multiple Mask R-CNN models. In the last Chapter 6, we describe multiple components of this model and experimentally demonstrate their benefits. Then we analyze the results of our best model on levels seen during the training. Then we analyze the accuracy of models for each level pack on unseen geometric problems with leave-one-out evaluation. Finally, we present multiple example solutions of Euclidea geometric problems.

2. Euclidea environment for solving geometric construction problems

In this chapter, we describe Euclidea, an online geometric construction game. The game is played with construction tools, which are used to complete various geometric problems. We will then present our version of Euclidea and describe how to apply random transformations to existing levels to create new variants of the same level.

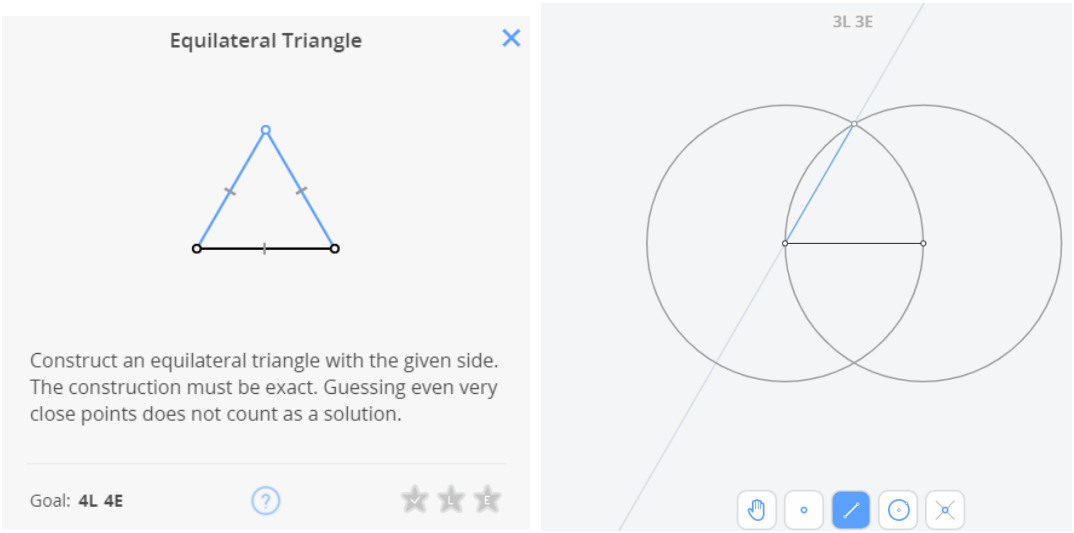


Figure 2.1: Screenshot from Euclidea [11]. The goal of this level is to construct an equilateral triangle with the given side. The figure contains the goal description on the left and the construction on the right. The current state of the construction cost is 3L and 3E. Each usage of a tool costs pre-defined L cost and E cost. To get to the current state 3 tools were used: 2x Circle tool and 1x Line tool. Each tool costs 1L and 1E (see 2.1).

2.1 Euclidea

Euclidea is an online geometric construction game in 2-dimensional Euclidean space. In Euclidea we use the following terms:

- **Geometric primitives** in Eculidea are points, lines and circles.
- **Level** in Euclidea is a geometric problem. Each level starts with an initial configuration, and the goal is to get to a target configuration.
- **Target configuration** denotes a state of a level that has all goals constructed, e.g. it is done.

- **Goal description** is a definition of the remaining goals. In Euclidea, it is a visual information on how the remaining goal looks like. In our visualization marked as green lines, circles or points.
- **Current configuration** denotes a current state of construction, e.g it is initial configuration plus all constructed primitives with tools.
- **Initial configuration** denotes the first state of a level, and it also contains a goal description.
- **Scene**, **level instance** is generated by applying random transformations to the one predefined level template.

The main goal is to find a sequence of construction steps leading from an initial configuration of objects to a given target configuration. The construction steps utilize a set of straightedge and compass based tools (see Section 2.3). The game has two additional goals (L and E) and one hidden goal (V). The additional goals are to minimize L or E costs. The L cost is 1 for each tool used, with the exception of Point, Move, and Intersection tools, which have zero cost. The E cost equals the number of lines and circles necessary to construct the tool. The hidden goal is to find all possible solutions to a level and is thus available only for levels with multiple solutions.

Every tool takes up to 3 arguments with values specified by coordinates of clicks on the image of the scene, for example, circle(A, B), where A, B are points on the image of the scene. An exception is the Move tool, which is realized by dragging points to another place, although it can be represented also by a translation vector, defined by two click coordinates.

Euclidea is divided into 15 level packs (Alpha, Beta, Gamma, ..., Omicron) with increasing difficulty. Each level pack contains around 10 levels with a similar focus. Description of levels from Alpha to Zeta can be found in Appendix A.1 (see Table A.1).

2.2 Precision and goal evaluation

In Euclidea, each level has its analytical model, which is projected on an image canvas. Each pixel in the image thus corresponds to a point in the analytical model. However, the analytical model points have float coordinates, making it impossible to find out precise coordinates of a point in the analytical model based on image data. Euclidea therefore provides certain tolerance for clicks and automatically finds the nearest geometric primitive corresponding to the click coordinates.

Euclidea checks the goal on the analytical level, so the player cannot cheat by drawing a similar goal instead. The player has to construct the target configuration to ensure that the result is the same as the goal.

In practice, the float parameters of two same geometric primitives stored in different objects cannot be the same, so even this comparison has some tolerance.

We have to keep this float tolerance in mind because it might cause issues, as described in the chapter about data generation (see Section 2.6).

2.3 Tools

This section describes the tools available in Euclidea. They can be divided into 3 categories: Tools creating points, construction tools for lines and circles and the Move tool.

2.3.1 Point and Intersection tools

The Point tool takes one argument and creates a point in the desired location. However, in accordance with the precision approach in Euclidea (see Section 2.2), this tool also finds all line and circle primitives within a small neighborhood of the click coordinates and creates a point using the first applicable rule from the following:

1. Create a point on the closest intersection of primitives if there are any.
2. Create a point on the closest geometric primitive if there are any.
3. Create a point at the exact coordinates given by the argument.

The Intersection tool creates points on all intersections of the two geometric primitives given in arguments. Both tools have E and L costs equal to 0.

2.3.2 Construction tools

The following tools take several arguments. Before a tool is executed, each click coordinates in the arguments are assigned the nearest geometric primitive matching the argument type.

Tool	Arguments	Description	L	E
Line	(point, point)	Draw a line passing through the given points.	1	1
Circle	(point, point)	Draw a circle centered at the first point with a radius marked by the second point.	1	1
Perpendicular Bisector	(point, point)	Draw the perpendicular bisector of two given points.	1	3
Angle Bisector	(point, point, point)	Draw the axis of an angle, where the second point marks the vertex and the first and the third points lie on its rays.	1	4
Perpendicular	(line, point)	Draw a line perpendicular to the line passing through the point.	1	3
Parallel	(line, point)	Draw a line parallel to the line passing through the point.	1	4
Compass	(point, point, point)	Draw a circle with a center in the third point and a radius given by the distance between the first two points.	1	4

Table 2.1: Tools available for construction steps in Euclidea. L and E denote the tool costs (see Section 2.1).

2.3.3 Move tool

The Move tool does not add any new geometric primitive but instead moves one primitive elsewhere and then recomputes the whole analytical model, if necessary. For example, if we move a line, the Move tool must also move all points on that line. Furthermore, it also has to adjust all the primitives intersecting those points, and so on. In Euclidea, the Move tool is used for exploring and understanding a given level. However, each level specifies the set of movable primitives, so not every primitive can be moved.

If a level configuration has multiple solutions, moving the primitives might remove some of the solutions. We will discuss this problem in data generation (see Section 2.6).

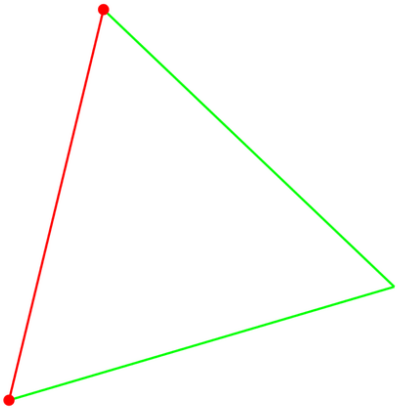
2.4 Our Euclidea-like environment

In this thesis, we use our python version of Euclidea based on [12], which contains every level and implements every tool.

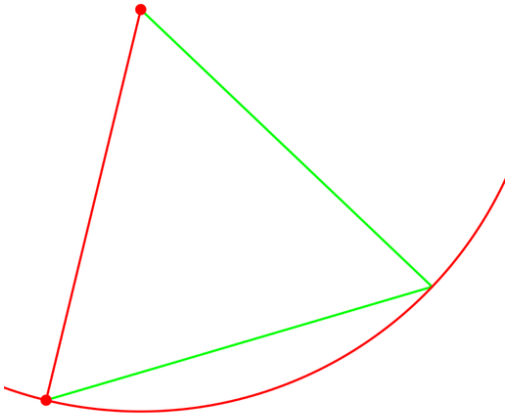
2.5 Example construction

Each state in our environment is represented by two gray-scale images: the current configuration and the target configuration. The two images are stacked into an RGB image, where the red channel is the current state, and in the green are remaining goals, and the blue channel is filled with zeros.

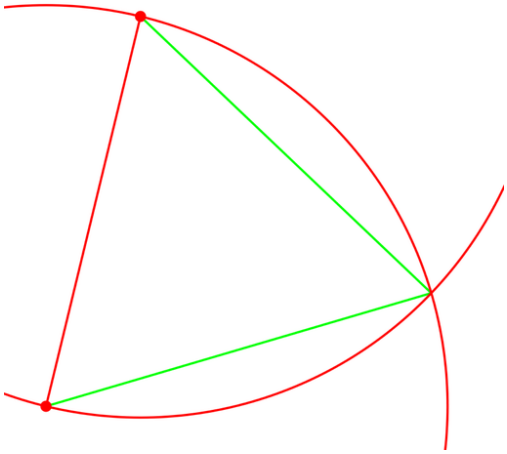
Figure 2.2 shows an example of a level in our environment and the construction steps. The task of the level is to construct an equilateral triangle given by one side.



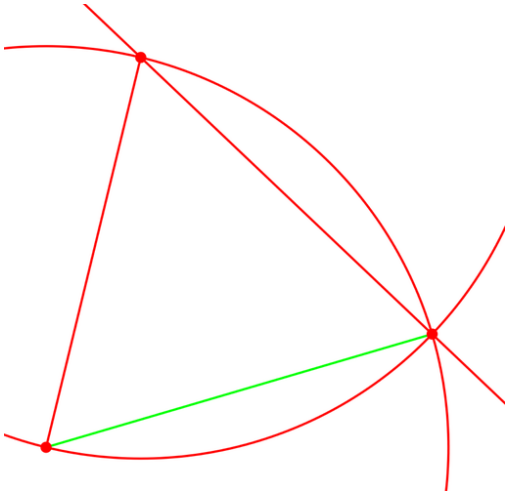
a) Initial configuration



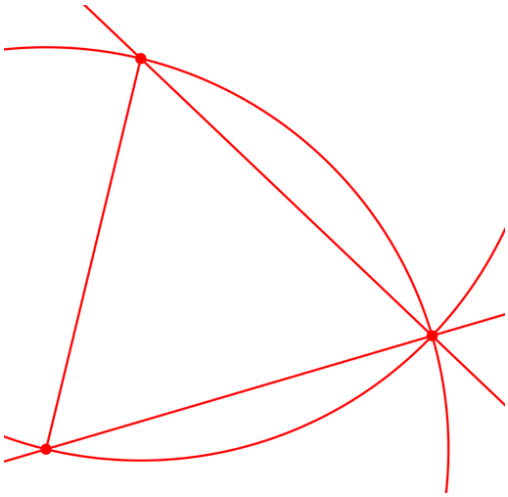
b) Construction step: 1



c) Construction step: 2



d) Construction step: 3



e) Construction step: 4 - level finished

Figure 2.2: Construction steps in our version of Euclidea for level *Alpha-05*: Equilateral Triangle. Current state (red) and remaining goals (green). The first state is the level definition. In each step, we use one tool to construct the goal. Note that in Euclidea, we can see the goal, but we cannot use it for construction.

2.6 Generation of level instances

To train an automatic recognizer, we add a generator of new level instances to our version of Euclidea. The new scenes are based on predefined ones. Each level has one predefined scene and goal definition. In order to generate a new scene, we transform the predefined scene using the Move tool (see Section 2.3). Generated scenes can be either valid or degenerated. Degenerated scenes cannot be solved due to precision problems or other construction problems. We will describe how to recognize degenerated scenes in the rest of Section 2.6.

We use the following steps to generate training data:

1. Load predefined scene.
2. Apply random rotation.
3. Randomly move all movable points.
4. Apply random scale.
5. Check scene validity / degeneration.
6. Apply random translation.

There are two types of geometric primitives in the predefined scenes: constrained and movable. Constrained objects are constrained to another geometric primitive in the scene. We denote a construction of each constrained object as a "step". Objects constructed within one step can be constrained to any movable object or to an object constructed in previous steps. An example of a constrained object is: "point on a line", "line perpendicular to another line" etc. Movable objects on the other hand are not constrained to any other object, so they can be moved, but their movement also moves all other objects constrained to them. Our version of Euclidea contains only movable points. Circles and lines may be constrained to other points; note that these points can be hidden from the visualization of the level.

The generation process can lead to degenerated scenes, i.e. scenes that cannot be solved based on image information, mainly because certain parts are too close to each other and cannot be distinguished. Additionally, during re-scaling, we compute a minimal possible scale to avoid small scenes degenerated due to a small size.

2.6.1 Degeneration criteria

Degeneration criteria are a set of rules that attempt to determine whether a scene is degenerated. We use the following four rules:

1. Different points cannot be too close to each other, measured in pixels.
2. Circles cannot have a too small radius, measured in pixels.

3. Two lines with a similar cannot be too close. Normal vectors compared in the analytical model.

4. Intersections of geometric primitives can not be too close to points used in the construction, measured in pixels.

The eule 4. was added later as it is mainly used in the advanced Euclidea levels. Rules number 1-3. were at first only applied to geometric primitives that were in the level definition. However, the degeneration status of a level definition does not include the degeneration of construction. So construction degeneration has to be checked as well. There is possibly an infinite number of different constructions, and we cannot make them all valid.

Without rule 4. the degeneration criteria had two significant problems that led to degeneration:

- Levels with multiple solutions can have configurations that make those solutions coalesce into each other. Probability of this generation goes to 0, but those solutions can be so close together that we cannot differ one from another based on visual information.

- During construction, we create many intersections between geometric primitives. Although we force points created during construction to be reasonable by previous rules, we force only those points that are necessary for the construction. Sometimes a point not necessary for the construction can be too close to a construction point.

The second problem is more general than the first one, and the solution to it also solves the first problem. To realize rule 4. we go through all intersections between all pairs of primitives and check if they are far enough from important points.

As we found out, for some levels these degeneration rules do not ensure validity and we need to define level-specific degeneration criteria as described next.

2.6.2 Level-specific degeneration criteria

Certain levels require degeneration rules that are not general. Some definitions of the levels also have "additional degeneration" which is again a set of rules applied only for instances of that level. The most frequent use of those additional degeneration rules is that specific angles cannot be too small or obtuse. It is also used to add additional constraints between objects of the scene that are level-specific. For example, in one level that contains two squares, those squares should not overlap.

2.6.3 Precision problems

While generating data, especially the constructions, we also have to deal with precision problems tied with scene re-scaling. Generally, when we create a construction, we have to check whether two primitives are the same. Each primitive

is defined by a number of normalized arguments. An object with parameters $args_1$ is identical to an object with parameters $args_2$ when:

$$|args_1 - args_2| < \epsilon \tag{2.1}$$

However, the difference can surpass ϵ when the objects are upscaled. Then objects are considered the same, but they are not. This can also occur the other way around. This may generate different construction then then desired. This precision problem has to be dealt with during the construction creation. However, it is only present when a level can have multiple solutions that can be potentially very close.

3. Complexity of exhaustive search for constructions

This chapter analyzes the difficulty of exhaustive search for geometric constructions in Euclidea. We analyze the exhaustive search by computing the branching factor of the tree search. After defining our choice of tree search, we analyze the branching factors in the actual tree search.

The Euclidean space has an infinite number of constructions. Hence the tree search for possible solutions has to have a large branching factor, and the search problem is unsolvable within a reasonable time to play the Euclidea game. According to [1], exhaustive search can be done within days on a standard computer. This chapter is an illustration of how difficult the problem is.

3.1 Euclidea tools for tree search

Before we decide which variant of the tree search to use, we will describe how to generate new nodes. We will use geometric primitives instead of click coordinates as the tool arguments. All tools can work with geometric primitives. The only exception is the Point tool. As a reminder, the Point tool creates points on intersections or primitives or points in open space (see Section 2.3.1). For search purposes, we can use the Intersection tool for finding intersections instead. The creation of a point on a geometric primitive is even more straightforward with the primitive given as the argument. However, there is possibly an infinite number of choices to create a point on the geometric primitive. We will only create random points that are "reasonable". A reasonable point is a point that is not too close to any other point nearby. We will omit the function for open space point creation of the Point tool and not use it at all. That function may be needed to construct some more advanced levels in Euclidea, but in this chapter, we will investigate the complexity of solving only simple levels of the Euclidea game.

3.2 Estimate of the branching factor

A complete tree search in the worst case has to go through b^n possibilities, where b is the branching factor, and n is the minimal depth of the solution. We can thus analyze the difficulty of the search problem by estimating the branching factor.

For this purpose, we have to define the number of degrees of freedom (DOF) of each tool. DOF is determined by the number of different tool outputs given by permutations of the arguments. For example the Line tool has DOF = 1, since $line(A, B) = line(B, A)$, whereas the Circle tool has different outputs $circle(A, B) \neq circle(B, A)$ hence circle has DOF equal to 2. The highest value of DOF is 3 (Compass and Angle Bisector tools).

For the purposes of the estimate, let us define G as the number of geomet-

ric primitives in the current scene. Then we can divide the tools into 3 groups, according to the number of arguments and the DOF:

- **Line, Perpendicular Bisector, and Intersection** tools have 2 arguments and a sing1e degree of freedom. Each tool in this group adds the same number of branches, which can be computed as follows:

$$G + \binom{G}{2} = \frac{1}{2}G^2 + \frac{1}{2}G, \tag{3.1}$$

  G branches for each tool usage like line(A, A) and $\binom{G}{2}$ for each tool usage like line(A, B), where A,B are unique combinations.

- **Circle, Perpendicular, and Parallel** tools have 2 degrees of freedom and 2 arguments. Each tool in this group adds the same number of branches, which can be computed as follows:

$$G + 2\binom{G}{2} = G^2. \tag{3.2}$$

  This group has maximal possible DOF for its number of arguments, therefore G^2.

- **Angle Bisector and Compass** tools have 3 degrees of freedom and 3 arguments. Each tool in this group adds the same number of branches, which can be computed as follows:

$$G + 4\binom{G}{2} + 3\binom{G}{3} = \frac{1}{2}G^3 + \frac{1}{2}G^2, \tag{3.3}$$

  G branches for each tool usage like compass(A, A, A). $4\binom{G}{2}$ counts the number of each tool usage like compass(B, A, A) and $\binom{G}{2}$ gives the number of combinations we can pick in 2 ways which argument is used twice an in 2 ways we can order arguments (DOF = 3, but -1 since two parameters are same). $3\binom{G}{3}$ branches for each tool usage like compass(A, B, C), times 3 because DOF is equal to 3.

The worst case happens when all tools are allowed. The branching factor is then the sum of branches of each tool:

$$b = G^3 + \frac{11}{2}G^2 + \frac{3}{2}G. \tag{3.4}$$

Note that the number of geometric primitives grows with every successful use of any tool by at least +1. Furthermore, for the Intersection tool we can create 2 points if we use the tool to find the intersection of two circles.

If we assume that every tool is allowed and there is one primitive at the beginning, then when we add 1 primitive at each step, the branching factor estimate grows as follows (see Table 3.1).

# of primitives	1	2	3	4	5	6	7	8	9
branching factor estimate	**8**	**33**	**81**	**158**	**270**	**423**	**623**	**876**	**1188**

Table 3.1: Growth of the branching factor estimate (see Section 3.2). Each step adds one geometric primitive.

However, these branching factors estimates represent the worst-case scenario and some of the actions can be invalid in the Euclidea environment. We can decrease the branching factor by simple heuristics. We will describe these heuristics in the next section.

3.3 Tree search over known primitives

With changes to tools in the previous section (see Section 3.1), we can use any type of tree search on our problem. Although we omit creating random points in space, we can solve several levels, especially in the first level pack of Euclidea.

In theory, we can use any tree search algorithm. However, the iterative deepening makes the most sense for its memory usage. Additionally, the levels construction length is known, so the initial depth can be set to the length of the construction, effectively transforming iterative deepening to depth-first search.

To reduce the branching factor of the tree search, we use several heuristics. Amongst them are 2 heuristics preventing action repeats, but most notably, the following two heuristics:

- **Reward cutting**: It is beneficial to know the effect of an action to decide which action will be used first. To get the results of actions, we execute each action and then reverse it. This can reveal errors that can occur during action execution. Most importantly, it allows us to check if an action completes a part of the goal. If it does, we assume this action is the only action in the current node of the search.
- **Goal construct-ability**: Since we use iterative deepening, the maximal depth is equal to d. If we are in depth $d-i$ and there are still k parts of the goal to complete, and $k > i$, we can cut the branch since we cannot finish the goal in i steps.

We can use the search to analyze the complexity of geometric construction problems. However, the search often runs longer than desired to play the game.

In the easier levels, the heuristics allow us to reduce the branching factor significantly. However, for more complex levels, the measured branching factor approaches the estimate given in Section 3.2.

Tables 3.2 and 3.3 show the branching factors of level packs Alpha and Gamma,

Alpha levels	Successful search	Branching factor (estimate)
01 T1 line	True	1.0 - 1.0 - 1.0 (6 - 10 - 15)
02 T2 circle	True	1.0 (4)
03 T3 point	True	1.0 - 1.0 - 1.0 (6 - 10 - 15)
04 TIntersect	True	1.0 (6)
05 TEquilateral	True	11.0 - 22.0 - 37.0 - 0.1 - 1.00 (21 - 36 - 55 - 109 - 136)
06 Angle60	True	5.0 - 21.0 - 48.0 - 67.5 - 0.1 (10 - 36 - 78 - 105 - 176)
07 PerpBisector	True	11.0 - 22.0 - 37.0 - 70.00 - 0.03 (21 - 36 - 55 - 105 - 175)
08 TPerpBisector	True	1.0 (6)
09 Midpoint	True	4.00 - 14.0 - 0.03 (13 - 27 - 66)
10 CircleInSquare	False	128.0 - 166.0 - 209.0 - 282.4 (172 - 216 - 265 - 359)
11 RhombusInRect	False	128.0 - 165.0 - 232.1 - 0.1 (172 - 216 - 304 - 433)
12 CircleCenter	True	3.00 - 27.0 - 60.0 - 105.0 - 154.9 - 0.0 (4 - 46 - 99 - 172 - 250 - 358)
13 SquareInCircle	False	15.0 - 29.0 - 49.0 - 93.0 - 155.5 (27 - 46 - 70 - 133 - 221)

Table 3.2: Branching factors of Alpha levels, first 10k nodes visited. The first column show Euclidea level. The second column indicates whether the search completed the construction successfully (True/False). The third column gives the average branching factor (top) at each depth of the search and its estimate in the parenthesis (bottom).

respectively. Branching factors of the other level packs can be found in Appendix A.2.

Gamma levels	Successful search	Branching factor (estimate)
01 ChordMidpoint	True	27.0 - 0.1 (48 - 128)
02 ATrByOrthocenter	False	71.0 - 189.0 - 394.0 - 691.7 (90 - 231 - 468 - 794)
03 AtrByCircumcenter	False	70.0 - 188.0 - 396.5 (90 - 231 - 468)
04 AEqualSegments1	False	71.0 - 189.0 - 393.0 - 690.5 - 0.0 (90 - 231 - 468 - 794 - 1459)
05 CircleTangentPL	False	30.0 - 66.0 - 212.8 (48 - 90 - 255)
06 TrapezoidCut	False	416.0 - 570.0 - 757.0 - 1120.2 (468 - 630 - 825 - 1211)
07 Angle45	True	13.0 - 65.0 - 180.0 - 0.0 (21 - 90 - 231 - 507)
08 Lozenge	False	30.0 - 66.0 - 120.0 - 277.0 - 528.2 - 0.0 (48 - 90 - 150 - 336 - 619 - 1214)
09 CentroidOfQuadrilateral	False	416.0 - 570.0 - 757.0 - 977.0 - 1238.0 - 1751.1 (468 - 630 - 825 - 1056 - 1326 - 1868)

Table 3.3: Branching factors of Gamma levels, first 10k nodes visited. The first column show Euclidea level. The second column indicates whether the search completed the construction successfully (True/False). The third column gives the average branching factor (top) at each depth of the search and its estimate in the parenthesis (bottom).

3.4 Tree search over automatically recognized primitives

The search described in Section 3.3 assumed that the geometrical primitives in the environment were known. This assumed having access to the environment. Since this thesis aims to construct geometric constructions from image data, we also modified the iterative deepening to automatically detect and recognize geometric primitives in the image and then proceed with the search with these detected primitives. We use the Mask R-CNN network trained to recognize all geometric primitive in the scene. To train the network, we use Alpha levels, where each target is a mask of all geometric primitives in the scene. We discuss more in-depth details of the Mask R-CNN object detector in the next chapter. This approach rapidly slows the deepening because we have to generate an image of the scene and then run the CNN detection in each node. This can be further optimized to run the detector only once at the beginning of the search, and then newly constructed geometric primitives can be derived based on the difference between the previous state image and the current state image. Overall, this approach is a slower variant of the previous tree search, but it fits the theme of the thesis.

4. Supervised learning approach

In this section we introduce Mask R-CNN [7] as our primary supervised model for solving Euclidean construction problems, describe how to train it and how to obtain Euclidea actions from Mask R-CNN model predictions.

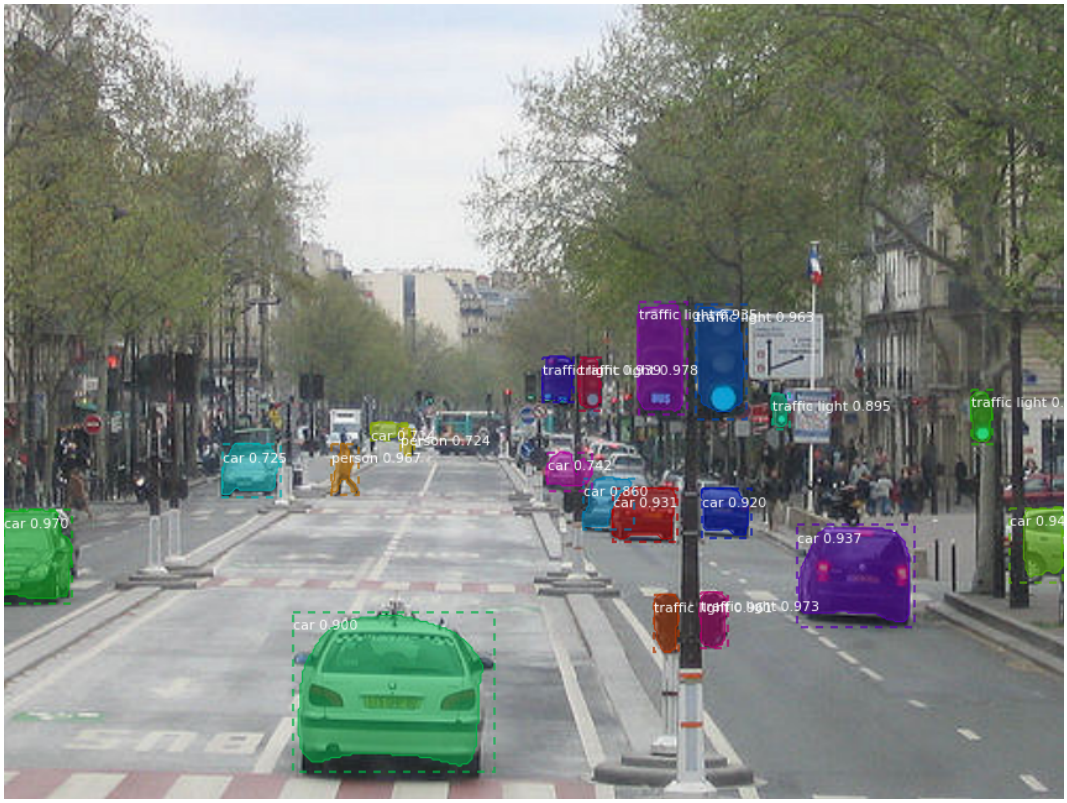


Figure 4.1: Example of a detection with the Mask R-CNN model. The figure contains multiple detected objects: cars, traffic lights. Each object has the mask, bounding box, and label with the score. Source: [13]

4.1 Mask R-CNN review

Mask R-CNN is a deep neural network used for, detection, classification and segmentation of objects in images and videos. The input is an image and the output is a set of bounding boxes, segmentation masks and class labels for each detected object instance in the image. Example output is shown in Figure 4.2

Mask R-CNN first computes image features with a convolutional backbone, usually the ResNet backbone. Followed by two stages of Mask R-CNN. The first stage is a deep convolutional network with Region Proposal Network, which proposes regions of interest from the features computed by the backbone. The second stage uses the ROI pooling layer and predicts class, bounding box and mask for each ROI.

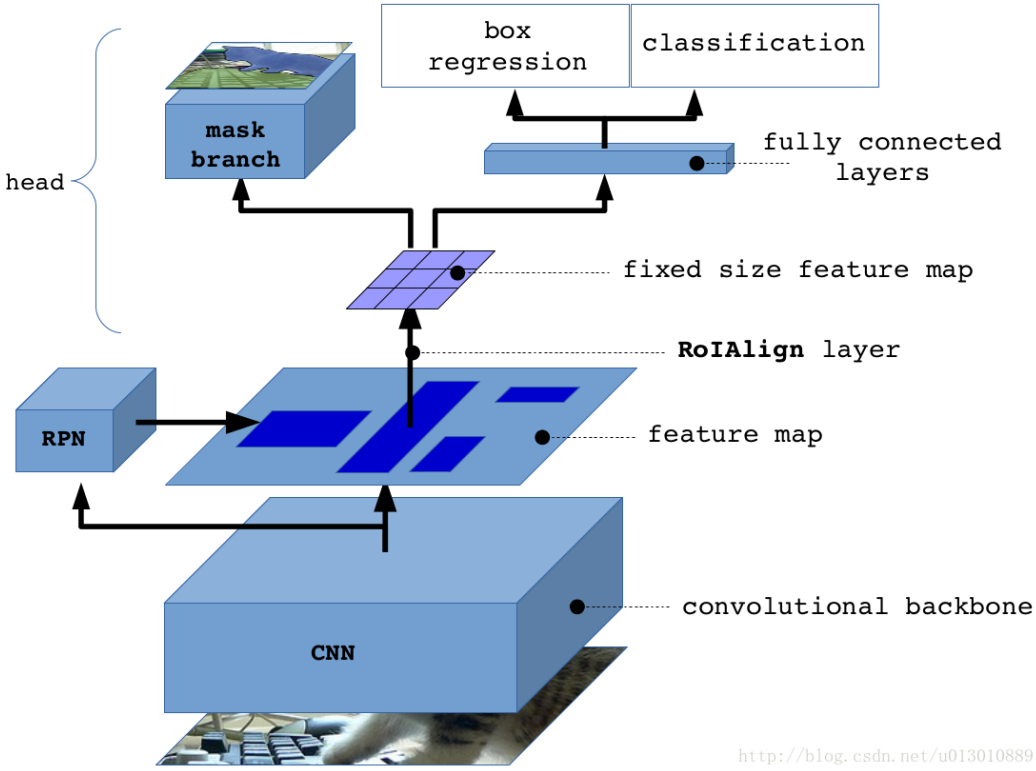


Figure 4.2: Diagram of the Mask R-CNN model, the CNN backbone, the ROI proposal network and head mask layers. Source: [14]

4.2 Mask R-CNN for solving geometric constructions

In Figure 4.3 we can see the schema of our approach. We generate the training data as we go through Euclidea levels. We solve levels by following a predefined construction that is recomputed to a current instance in the generation process (see Section 2.6.1). Each application of a Euclidea tool corresponds to one sample in the training data. To train Mask R-CNN to solve geometric constructions, we have to create training data that represent tool usage, and we have to adjust outputs of the network to work with the Euclidea-like environment.

We denote each application of a tool in our environment as an "action". For this purpose, we assign each tool an index, e.g. 1 for Line tool, 2 for Circle tool, etc. An action is then represented by the index of the tool and the corresponding number of click coordinates (see Section 2.3). For example, the Line tool needs two action clicks, which represent two points on the line. Next, we will describe how to generate training data for Mask R-CNN and how to infer actions from those masks.

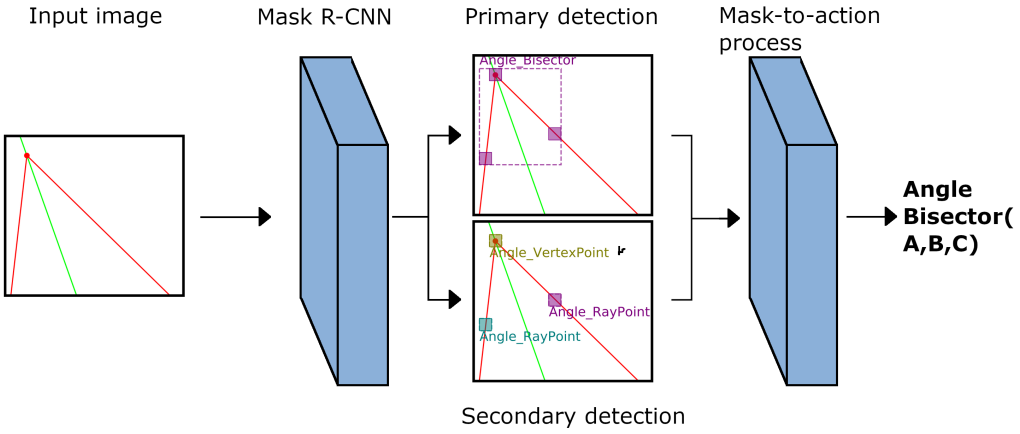
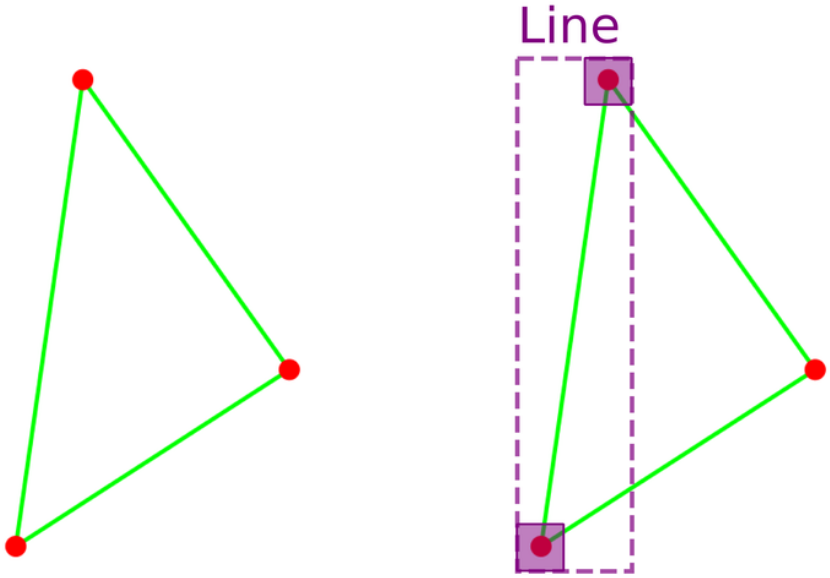


Figure 4.3: Diagram of our approach. The input is an image of the scene, which is run through Mask R-CNN. The input image contains an RGB image with the current state of the construction in the red channel and the remaining goal in the green channel. The results are primary and secondary detections, which are then used to obtain an action tool for Euclidea. Note that there is no head for predicting primary and secondary detections. Instead, both are predicted with the same head and sorted to primary and secondary detections by the prediction id. How the Mask-to-action process works is described Sections 4.2.1 and 4.2.2

4.2.1 Action to mask

We represent the input of the Mask R-CNN as an image of the scene with current state in the red channel and the remaining goal in the green channel. In our experiments, we also add extra channels representing history (see Section 6.1.2). The n-th history channel represents the state of the scene n steps before the current state.

A target is a pair of an object type and its location, represented as a mask of the object. Object type corresponds to the tool that is used in the step. The target mask is the mask of each point click contained in the step. These sub-masks are squares around the click location. Also, some tools have a line as its argument; passing a line argument can be a mask of a single click on the line or a mask of the whole line. Figure 4.4 shows an example of input and output.



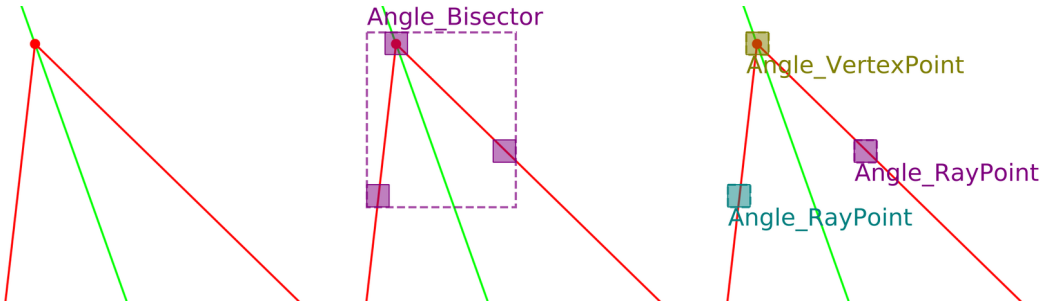
a) Input b) Primary detection

Figure 4.4: A sample from training data for level *Alpha-01*: Tutorial for the Line tool, connects points. Both parts contain the current state (red), remaining goal (green). Input for the model is on the left and target mask (purple in this case) for the Line tool with respective bounding box on the right. The mask contains two areas for each endpoint of the triangle side, representing click coordinates for the tool. The Line tool does not have position-dependent parameters. Hence the secondary detection is empty and not shown.

4.2.2 Tools with position-dependent parameters

Encoding clicks like in the previous subsection is not sufficient for most tools where tool parameters are position-dependent. For example, the Circle tool has two parameters: a circle center and a point on the circle. For such tools, we also have to distinguish these points. Therefore we have to add an additional target. For the Circle tool, for example, we also detect the circle center and the circle radius-point. In this thesis, we call the Circle tool detection the primary detection, and the detection intended for parameter order the secondary detection. We can see an example of primary and secondary detections in Figure 4.5.

Another special case is the Compass tool. We know that $\text{compass}(A, B, C) = \text{compass}(B, A, C)$ where A, B, and C are any valid inputs from the definition of the Compass tool. This is related to the number of degrees of freedom for each tool discussed in Section 3.2. If a tool has this property, we can use the same object type for A and B. This will reduce the number of object classes and also improve the problem trainability. The reason for it is that there can be multiple very similar scenes in training data that have different permutations of such points that can be switched. Then we could get to a situation where the points may be indistinguishable.



a) Input b) Primary detection c) Secondary detection

Figure 4.5: Sample from training data for level *Beta-02*. All three parts contain the current state (red), the remaining goal (green). Subfigure b) is a primary detection of Angle Bisector tool (purple), c) are three secondary detections: 2x angle ray point (purple, turquoise) and 1x angle vertex point (dark-yellow).

4.2.3 Reducing ambiguity

An ambiguity in a construction occurs when the next step is, for example, a random point on a given line. We would like to have every possible point on the line that does not lead to a degenerated construction in the training data. However, computing areas where points could be is extremely time-consuming. To check all degeneration criteria for a single example, we would have to check $O(n^2)$ rules, where n is the number of points in the scene. Furthermore, those areas where we cloud create a point are not even continuous, so we would have to test many points in space. On top of that, when adding multiple random points, we would have to solve whether a pair, triplet, or n-tuple of random points lead to valid construction, which would lead to exponential numbers of degeneration checks.

Therefore, we reduce the ambiguity as much as we can. For example, in level *Alpha-12* the goal is to find the center of a given circle. The order of the construction is:

1. Angle bisector between 2 random points on the circle.
2. Another angle bisector between 2 random points on the circle.
3. Intersection of those angle bisectors is the circle center.

When we want to minimize ambiguity, we need just 3 random points. It is also beneficial to fix the relative positions of these points. We choose to fix those 3 random points to sections that correspond to points $(1, 0)$, $(-1, 0)$, $(0, 1)$ on the unit circle. If we do not address these ambiguities, predictions can look like in Figure 4.6, where we can see many point candidates. Having that many point candidates significantly lowers the inference accuracy.

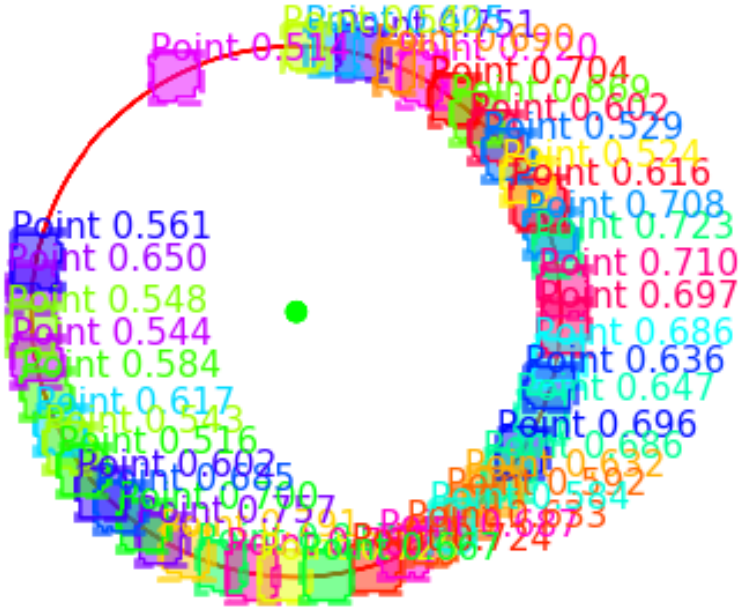


Figure 4.6: Level *Alpha-12*: Find the center of a given circle. In the construction of this level, we have to create three random points on the circle, but the Mask R-CNN proposes many points due to ambiguity in training data used for training of the model. Due to the number of predicted points, we cannot use prediction like this in inference.

4.3 Solving geometric constructions

In this section we describe how we obtain actions for our Euclidea environment from the Mask R-CNN model.

4.3.1 Mask to action

We cannot measure the success of the model directly by its loss function. Instead, we have to measure the accuracy of level completion in the Euclidea environment. In the experiments section (see Section 6.1.5 and Figure 6.4), we will see that a higher loss model can have better completion accuracy. However, we still have to monitor the loss since it should decrease during training regardless of its value.

To solve Euclidea levels, we have to transform Mask R-CNN output to suit our environment input. The output of the model is a mask and the object type. The final layer of the Mask R-CNN produces a heat map, which is a probability map that gives each pixel a probability whether it is part of the mask or not. The heat map is then transformed into a mask by applying a > 0.5 threshold to each element of the heat map. The heat map is used to create actions suitable for the environment.

The most straightforward actions to create are actions that correspond to the Line tool and Perpendicular Bisector tool, because these tools have a single degree of freedom. Hence we do not have to deal with the order of the arguments.

Both tools take two arguments. We take the two most probable points from the heat map that are to too close to each other. We use the same minimal distance threshold as the minimal point distance in degeneration. To find these two points, we use our version of the RANSAC algorithm [15].

Now, let us describe more complicated tools. As an example, let us consider the Angle Bisector tool, which has 3 input parameters. Detection of this tool should have 4 detection outputs from Mask R-CNN: 1 primary and 3 secondary detections. The primary detection is the detection of the angle bisector. Secondary detections are detections for the individual points: one angle vertex point and two angle ray points (see Figure 4.5).

To execute the tool, we have to determine the correspondence between the primary and the secondary detection. We can obtain 3 point coordinates from the primary detection in the same way as above with the Line tool. We can also get 3 points from 3 secondary detections, each giving us one point. Each point from the primary detection corresponds to some point in the secondary detection, but these points do not exactly align. The point correspondence is then determined by minimizing distances between the primary and the secondary points (each point has to be used exactly once).

Now we can create an agent that can solve Euclidea levels. In the previous paragraph, we have described how to get an action from a single prediction. However, Mask R-CNN can predict multiple actions. Mask R-CNN also predicts a score for each object, representing the confidence of the prediction. For now, we can use the prediction with the highest score. Detections with lower confidence may also be useful, and we will return to them in the next chapter. However, multiple detections complicate the assignment of the secondary predictions. Mask R-CNN does not connect different detections, so we can have more secondary detections than points in the primary detection. We can still use the approach we mentioned previously, just for the assignment we no longer use each point exactly once, instead once or not at all. Then the agent does the following:

Result: Test level inference: True if level completed, False otherwise.
Initialize a level;
while *level not complete* **do**
  $s \leftarrow$ current state of the level;
  $p \leftarrow model.predict(s)$;
  if *predictions p are empty* **then**
    return False;
  end
  $a \leftarrow$ action from p with highest score;
  execute a
end
return True;

Algorithm 1: Inference: Top score prediction

In Table 4.1, we can see an example solution of the geometric problem solved using top score predictions.

4.3.2 Incomplete detections

There are instances of levels with incomplete detections. A detection is incomplete if there are not enough points in the primary or the secondary detection. If there are not enough primary points, we cannot determine any action. For secondary points, we can determine as many points as possible, and then the rest of the points can be randomly assigned. Applying a random assignment is frequently used in inference, mostly for the Circle tool. Many constructions use a sequence of circle(A, B) followed by circle(B, A) to obtain equilateral triangle, perpendicular bisector, midpoint of a segment, 60° degree angle, etc. During inference of this situation based on data from the first detection, we have to construct either circle(A, B) or circle(B, A) and the second detection has to construct the other circle. When we go through Euclidea levels while creating the training data, we have to choose which step we do first. Furthermore, because we generate level instances randomly, there can be multiple similar level instances during training that choose different moves, i.e. there are two samples in training data that have the same primary detections but different secondary detections. During training, those secondary detections may cancel each other out resulting in the primary detection without the secondary one. However, as mentioned above, we can use any action defined by the detected primary actions. This effect occurs mostly for the Circle tool, but it can occur for other tools as well.

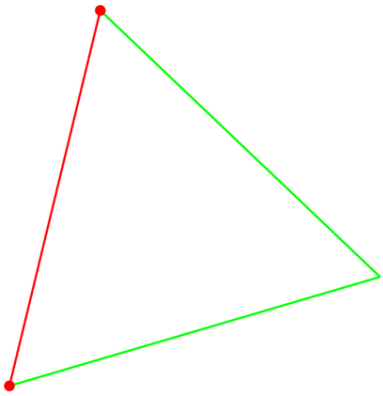
4.3.3 Opposite corner detection

Some detections may miss click coordinates even in the primary detections. However, the prediction of a bounding box can still be predicted correctly. If a tool has 2 arguments, we can find another point in this situation with point reflection. We reflect the detected point with the center of symmetry in the center of the bounding box. The bounding box for the training data is a minimal bounding box containing the mask in the training data. Hence when we expect the output to have 2 click coordinates, one point is in a corner, and the other is in the opposite corner of the bounding box.

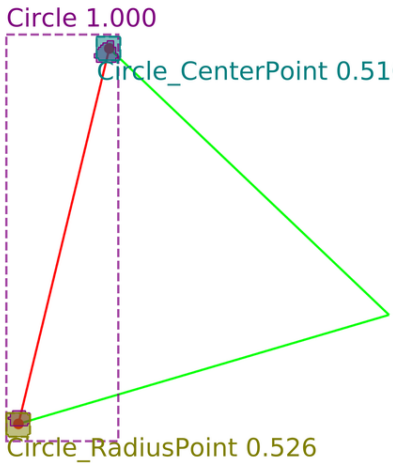
4.4 Other models

Before we experimented with Mask R-CNN, we tried models that predict action click coordinates straight from images. The output of the model were coordinates, not a mask. We used a convolutional network with few densely connected layers and two output layers: a softmax layer that predicted the tool index and a layer with 6 outputs representing x and y coordinates of 3 points. The activation function for the layer predicting coordinates was the sigmoid activation multiplied by the window size (size of the input image). Hence all predictions were valid coordinates within the image. However, this model had problems even on simple levels like *Alpha-01*. It was able to do the 1st step of *Alpha-01*, but then it was never able to predict the second step correctly. Although the model was able to detect points in Alpha-01, it could not detect already constructed lines, and thus the model was trying to construct one line repeatedly. The model had even worse results on levels that use position-dependent tools. It also produce only

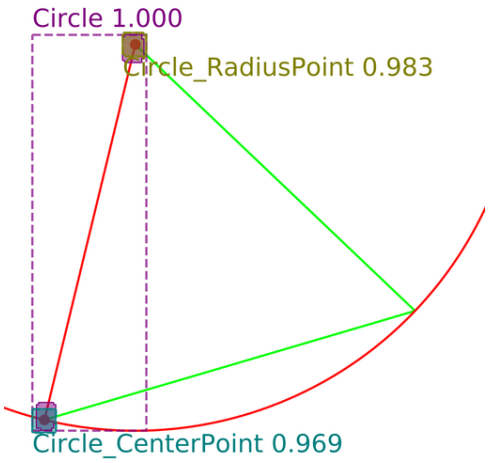
one output compared to Mask R-CNN, where we can use multiple outputs as potential moves.



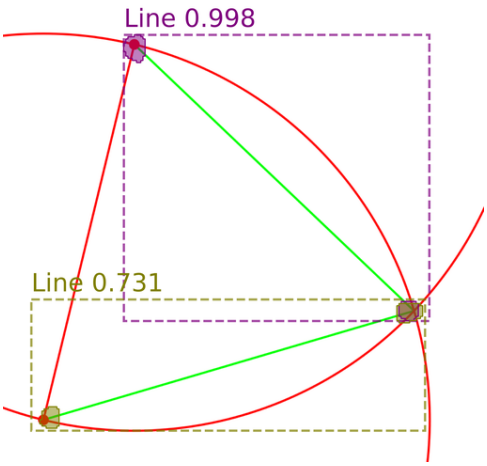
a) Initial configuration



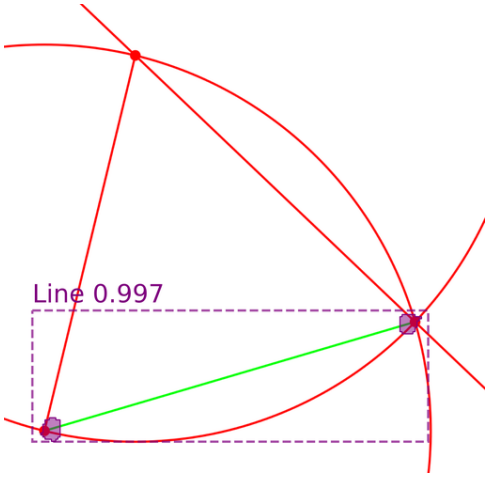
b) Construction step: 1. Based on the prediction a circle will be constructed.



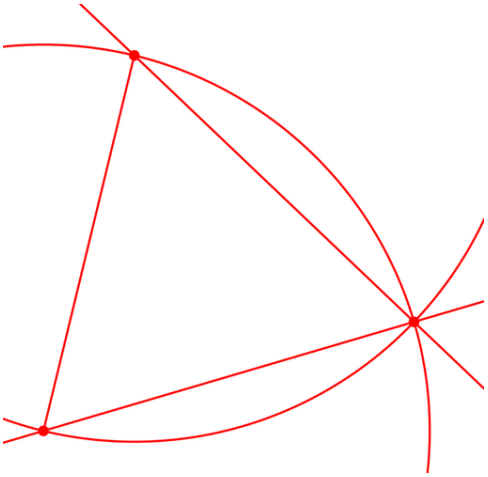
c) Construction step: 2. Based on the prediction a circle with a different center then in last step will be constructed.



d) Construction step: 3. Based on the prediction a line will be constructed.



e) Construction step: 4. Based on the prediction a line will be constructed.



f) Construction step: 5 - level finished

Table 4.1: Sequence of construction steps with predictions for the Euclidea level *Alpha-01* Equilateral Triangle. The first state is the level definition consisting of the initial configuration (red) and the remaining goal (green). In each step, our approach predicts the usage of one tool towards constructing the goal.

5. Solving unseen geometric constructions

As we have already mentioned, there are multiple predictions in Mask R-CNN models (see Section 4.3.1). In this chapter, we further examine those predictions. To do so, we create a program for the exploration of hypotheses given by multiple models. Then we introduce the hypothesis tree search for solving unseen levels. Lastly, we analyze the inference of unseen levels with the leave-one-out method.

5.1 Hypotheses generated by Mask R-CNN

Each primary detection by the Mask R-CNN model described in Chapter 4 can be transformed into an action. We denote each action, its arguments and results as a "hypothesis". The result of an action contains the reward and output geometric primitive constructed during the action execution. The reward indicates whether the output primitive is a part of the goal or not. If an action constructs part of the goal, the reward is equal to $1/n$, where n is the number of primitives in the goal, otherwise it is equal to zero. In Table 5.1 we can see a hypothesis that successfully finished one of the four goals and hence it has a reward equal to 0.25. We can extract multiple actions from the Mask R-CNN model outputs and then transform them into multiple hypotheses.

When we obtain hypotheses from the Mask R-CNN, we can explore the construction space defined by those hypotheses. Furthermore, we can also use hypotheses from multiple models trained for different tasks. However, Mask R-CNN scores across hypotheses from different models are not well calibrated. For exploration purposes, we developed an interactive program where the user can choose multiple trained models and levels for inference, and then run inference where the user can choose which hypothesis should be used at a given time. In Figure 5.1 we can see a screenshot from the program followed with hypotheses details in Table 5.1. Hypothesis visualization is based on our Euclidea visualization (see Section 2.5) with the addition of the hypothesis result object to the blue channel. The program can get hypotheses from any model, but in this thesis, we prepared models trained on the first 6 Euclidea level packs:

- 68 "level" models (one for each level).
- 6 "level-pack" models (one for each level pack).
- 1 "all" model (for level packs Alpha - Zeta).

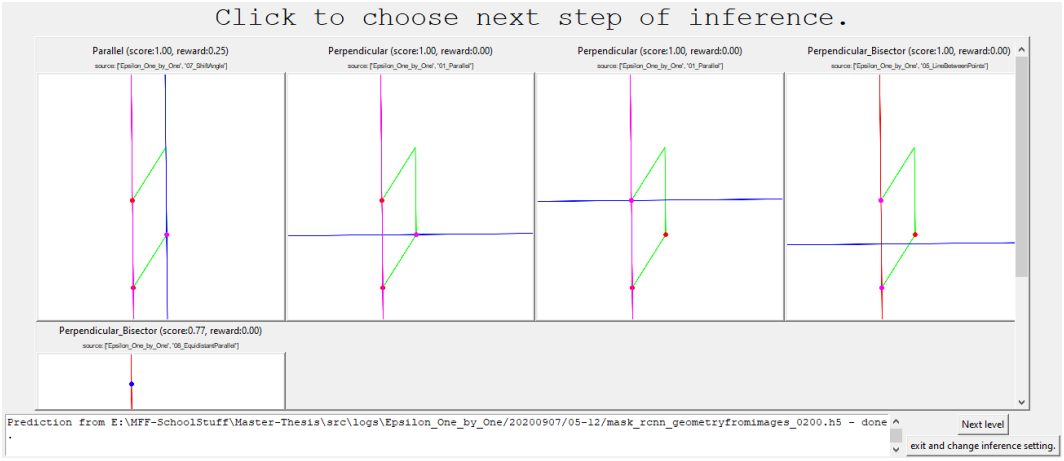
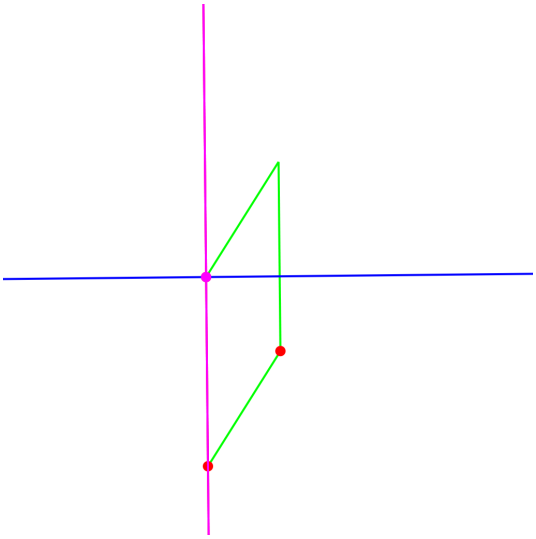
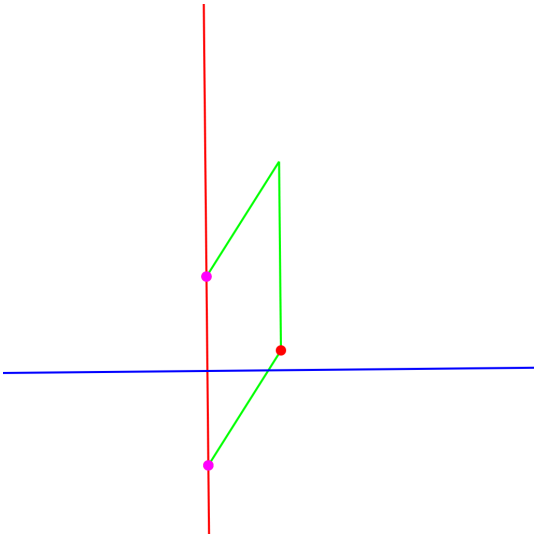


Figure 5.1: Screenshot of our hypothesis explorer program. Construction of the level *Epsilon-03*, construct a parallelogram given by 3 points. To solve this level, we use all level-specific models for Epsilon with the exception of the model for *Epsilon-03*, i.e. 11 models. In this step of construction, we have 5 different hypotheses given by 4 different models. Other models either do not give any hypotheses or give hypotheses that can be grouped with other hypothesis (see Section 5.3); we consider only a single hypothesis per group. In the program, the user chooses manually which hypothesis should be constructed. Details of each hypothesis are in Table 5.1.

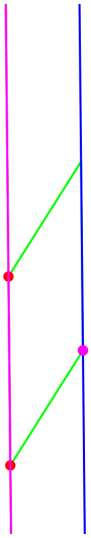
a) The current state without any hypothesis (input for the Mask R-CNN models).



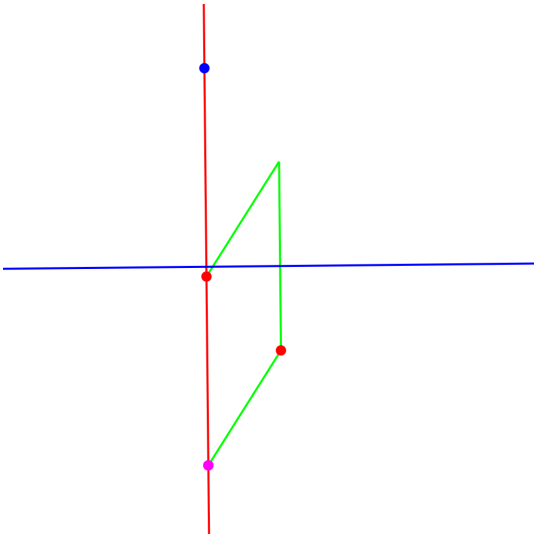
b) Source model: *Epsilon-01*, score: 0.99, tool: Perpendicular, reward: 0.0.



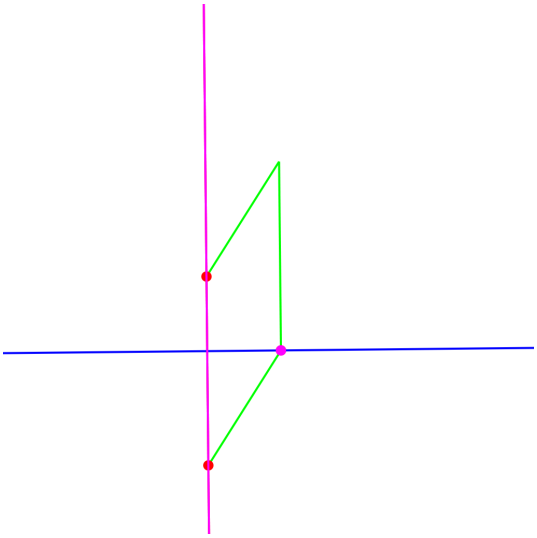
c) Source model: *Epsilon-05*, score: 0.99, tool: Perpendicular Bisector, reward: 0.0.



d) Source model: *Epsilon-07*, score: 0.99, tool: Parallel, reward: 0.25.



e) Source model: *Epsilon-08*, score 0.77, tool: Perpendicular Bisector, reward: 0.0.



f) Source model: *Epsilon-01*, score: 0.99, tool: Perpendicular, reward: 0.0.

Table 5.1: Five different hypotheses for solving the level *Epsilon-03*. This table is a detail of hypothesis from Figure 5.1. Each image contains the current state (red), remaining goal (green), hypothesis produced by Mask R-CNN (blue), and highlighted arguments of the tool (purple). We can see that hypothesis d) has a reward equal to 0.25 and thus contributes to the goal and should be picked in this step.

5.2 Inference with tree search

In the previous section, we described how to get hypotheses from multiple Mask R-CNN models. Now we will search for the target construction within these hypotheses. To compare the results of the hypothesis tree search with the exhaustive tree search from Chapter 3, we use iterative deepening. The tree search

has to build an input image and run predictions of all Mask R-CNN models in each node, which significantly increases the time spent on one node. However, the hypothesis tree search should have a significantly lower branching factor.

5.3 Reducing the number of hypotheses

Hypotheses produced by different models increase the branching factor and thus also the search time. In order to speed up the tree search, we group similar hypotheses and explore only one of them. We consider two hypotheses to be similar if they have precisely the same output geometric primitive. Note that they can have different arguments, including different tools.

5.4 Cheat moves

In Euclidea, a goal cannot be finished by simply drawing a similar line or circle (see Section 2.1). We denote such steps as cheat moves. However, a hypothesis may suggest drawing a cheat move. We can recognize such cheat move hypothesis by coordinates of its arguments. For example, one of its arguments is a point in space without any geometric primitive nearby. Note that with the mentioned approach, we cannot find every cheat move. Some levels aim to find a specific point on a line or circle, which can be guessed instead of constructed. Cheat move hypotheses can be removed as they do not lead to the goal. However, when all necessary points are constructed further in construction, this cheat move prediction becomes a legitimate move.

5.5 Leave-one-out evaluation

We use the "leave-one-out" method to evaluate performance on unseen levels. For our purposes, the method inputs are levels with corresponding level-specific models. To evaluate accuracy of a level we use all other models, i.e. models that were not trained for this level. We can also apply this on whole level pack models. Note that we should avoid a situation where two different models are trained for the same task. For example we should not combine models for *Alpha-01* level and Alpha level pack, since both models are trained for level *Alpha-01*. For inference we use the hypothesis tree search (see Section 5.2). Note that due to score misscalibration, the top score detection (see Section 4.3.1) approach cannot be used when we have multiple models. The accuracy of the leave-one-out set-up is reported in the experiments (see Section 6.3).

5.6 Connections between levels

When evaluating a level using leave-one-out method, we might use hypotheses only from a fraction of level specific models. We denote that level X is "connected" to level Y, when a model trained for Y contributes with any hypothesis to a successful construction during the inference for level X. Note that the relation "connected" is not reflexive, e.g. when X is connected to Y, then Y is not

necessarily connected to X. Since we run the hypothesis tree search during the leave-one-out evaluation, we obtain connections in following way: If the search is successful, we collect all models that contributed to the solution in the final backtracking of the search.

6. Experiments

This chapter describes the components of our model and how they affect model performance on Euclidea levels (see Section 6.1). Furthermore, we analyze the performance of the best model (see Section 6.2) and the leave-one-out method performance on unseen levels (see Section 6.3). Then we show example solutions of several Euclidea levels (see Section 6.4).

6.1 Algorithm design choices

This section describes the different components of our model and the training setup. In each subsection, we describe a new component of our approach that improves accuracy. We start with our initial model that had only a low accuracy on the Alpha level pack and then add the new components that we have designed to improve the accuracy of the model.

Levels can be solved using different constructions. Some constructions can be more trainable than others, e.g. a model trained for construction A of level X might have higher accuracy than a model trained for construction Y of level X. Note that construction A and B can be completely different constructions but also can differ only in the permutation of construction steps. Also, the probability of degenerated scenes varies across different constructions. In practice, this probability should be low enough so as not to slow down the data generation. Therefore, before training a new level pack, we first ensure that all its levels are trainable by level-specific models.

Some models have difficulty deciding which tool should be used at a given state of the construction. To make it less complicated, we are using hints. To generate random instances of a level, we have to compute the level construction as well (see Section 2.6.1). Since the models aim to mimic this construction, we provide them with hints which tool should be used at a given time. The hints are not included in the Mask R-CNN input, but instead we consider only detections with a tool index corresponding to the hint. We compare accuracies with and without hints. The hints are used for our initial models that were not able to solve the most manageable levels.

6.1.1 Detection of points and intersections

In our version of Euclidea, we are limited in using tools with point arguments. Let us consider an example construction: Construction of the midpoint of a given segment AB. Assume that we have already constructed two circles, circle(A, B) and circle(B, A). The last step is to draw a line given by the two intersections of the circles. However, we cannot directly do so because intersections are not marked as points. Instead, we have to use the Point tool on each intersection, and only then create the line. To be able to draw the line without the need to mark the points, we use the automatic point detection. Realized when we execute a tool. The environment returns whether the tool was executed successfully or

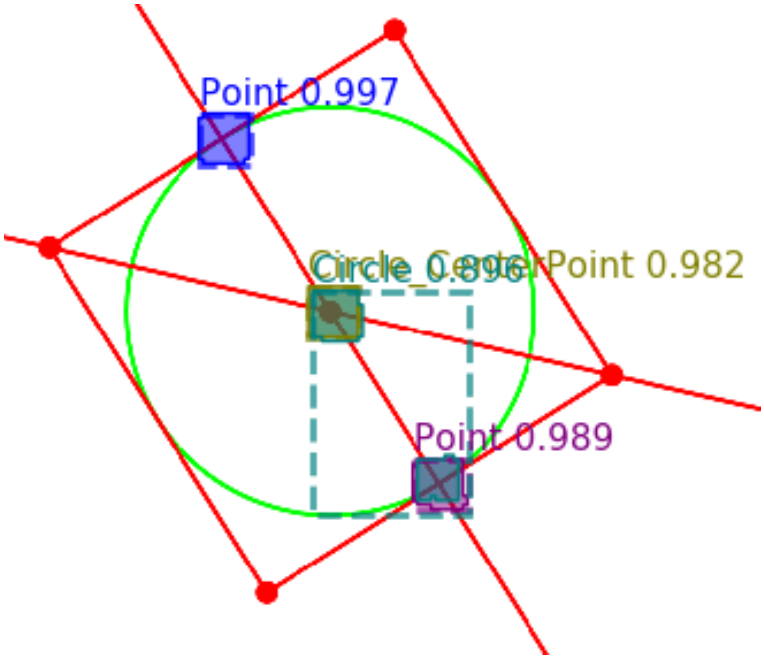


Figure 6.1: Example of a detection for level *Alpha-10*. The goal of this level is to construct an incircle of a given square. The figure shows the current state of the construction (red), the remaining goal (green) and multiple detections of the Point tool and Circle tool. The figure shows a potential problem when we do not use the automatic point detection (see Section 6.1.1). If we do not use the automatic point detection, we have to detect usages of the Point tool as well. Then the network might prefer creating many points instead of using other tools.

not. If not, we can use the Point tool on each argument to mark these points. This change allows us to reduce the usage of the Point tool and use it only if the level has a goal to find a specific point, e.g. the center of a triangle.

In Figure 6.1 we can see a potential problem if we do not use the automatic point detection. Based on the figure detection, we would be creating all points since they have a high Mask R-CNN score and we would not use the Circle tool.

From Table 6.1 it is clear that automatic point detection improves the accuracy of every level. We can also see that automatic point detection no longer needs hints, as there is little difference between hints and no hints accuracy.

Alpha levels	without automatic point detection		automatic point detection	
	Hints	No hints	Hints	No hints
01 T1 line	0.889	0.86	0.956	0.95
02 T2 circle	1	0.93	1	1
03 T3 point	1	0.84	0.99	0.98
04 TIntersect	0.941	0.908	0.97	0.968
05 TEquilateral	0.923	0.822	0.959	0.931
06 Angle60	0.962	0.774	0.982	0.992
08 TPerpBisector	0.94	0.98	0.98	0.976
09 Midpoint	0.84	0.512	0.986	0.986
10 CircleInSquare	0.386	0.256	0.736	0.768
11 RhombusInRect	0.828	0.75	0.958	0.951
12 CircleCenter	0.134	0.004	1	0.916
13 SquareInCircle	0.918	0.894	0.996	1
Average	**0.813**	**0.71**	**0.959**	**0.951**

Table 6.1: Accuracy of Alpha levels during inference for the model with and without automatic point detection, evaluation on 500 instances for each level with and without hints. Note that some accuracies with hints can be slightly higher than without hints. Different permutations of construction steps can change the predictions of the model. When we use hints, we choose a prediction with the highest score that matches a hint, but there can be a higher score that does not match the hint.

6.1.2 History channel, more data on the input

Models generally have a problem to determine the exact step of the construction. We can observe that a single detection contains predictions of actions that correspond to actions used in later stages of the construction. In most cases, actions that are supposed to be used later have a lower Mask R-CNN score. However, when this action have a higher score, then it is chosen sooner before steps that might be necessary to construct before. This situation does not lead to a successful construction when we use the top score inference (see Section 4.3.1). To avoid these situations, we add a history channel to the training data. With the current state in the first channel, the goal in the second channel, we add the previous state to the third channel. Furthermore, we can add a state 2 steps ago to the fourth channel, a state 3 steps ago to the fifth, and so on. However, we use only one history channel because the pre-trained Mask R-CNN model contain 3 channels, and increasing the number of channels would require training of the whole Mask R-CNN model, including its backbone.

In Table 6.2 we can see that the history channel improves the accuracy on the Alpha levels. Moreover, models with history were the first models that learned Beta levels with reasonable accuracy.

Alpha levels	Without history	With history	Beta levels	With history
01 T1 line	0.95	0.9406	01 BisectAngle	0.446
02 T2 circle	1	1	02 TBisectAngle	0.768
03 T3 point	0.98	1	03 Incenter	0.732
04 TIntersect	0.968	0.956	04 Angle30	0.98
05 TEquilateral	0.931	0.996	05 DoubleAngle	0.71
06 Angle60	0.992	0.998	06 CutRectangle	0.31
08 TPerpBisector	0.976	0.998	07 DropPerp	1
09 Midpoint	0.986	0.988	08 ErectPerp	0.998
10 CircleInSquare	0.768	0.914	09 TDropPerp	0.494
11 RhombusInRect	0.951	0.991	10 Tangent1	0.266
12 CircleCenter	0.916	1	11 TangentL	0.958
13 SquareInCircle	1	1	12 CircleRhombus	0.66
Average	**0.951**	**0.981**	**Average**	**0.693**

Table 6.2: Accuracy of levels for the model with and without history (see Section 6.1.2), evaluation on 500 instances for each level, for Alpha pack on the left and Beta pack on the right. Note that for Beta. The model without history was not working at all. The models utilizes automatic point detection (see Section 6.1.1).

6.1.3 Stages of Mask R-CNN training

Beta levels	Without 4+	With 4+
01 BisectAngle	0.446	0.658
02 TBisectAngle	0.768	0.73
03 Incenter	0.732	0.75
04 Angle30	0.98	0.986
05 DoubleAngle	0.71	0.894
06 CutRectangle	0.31	0.638
07 DropPerp	1	0.812
08 ErectPerp	0.998	0.996
09 TDropPerp	0.494	0.528
10 Tangent1	0.266	0.994
11 TangentL	0.958	0.988
12 CircleRhombus	0.66	0.904
Average	**0.693**	**0.823**

Table 6.3: Accuracy of levels during inference for models trained with the following setups. **Without 4+**: First 120 epochs training of head layers, then 80 epochs training the whole network. **With 4+**: First 60 epochs training of head layers, then 60 epochs of 4+ layers and then 80 epochs training the whole network. Both models use a history channel (see Section 6.1.2), and there are 500 instances of each level. Note that the model without 4+ is the same model for Beta mentioned in Section 6.2.

The training of Mask R-CNN is often divided into several stages. Each stage

can have different training parameters. For training purposes, we have a stage with head layers,i.e. layers at the on of the network that compute masks, bounding boxes and class labels from features detected in the backbone. The implementation of Mask R-CNN that we use [13] divides the backbone into 5 stages and allows to train layers from a selectd stage. We denote "4+" as training of the 4th and 5th stages of the backbone and all head layers. Until now, we were using the following training setup: First, we train only head layers of Mask R-CNN and then we train the whole network. Table 6.3 shows that 4+ training improves accuracy on the Beta level significantly. In Figure 6.2 we can also see that models with 4+ training have a smoother loss. The spikes in epochs 60 and 120 are caused by enabling more layers to be trainable: In the first 60 epoch, we train only the head layer, and then we start 4+ as well and from epoch 120 we train the whole network.

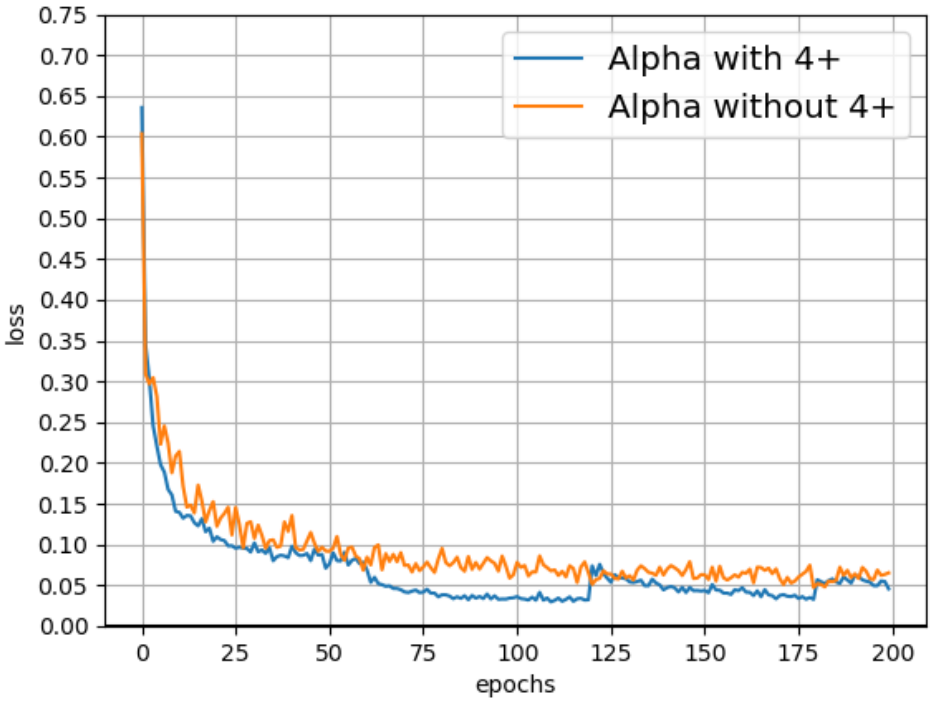
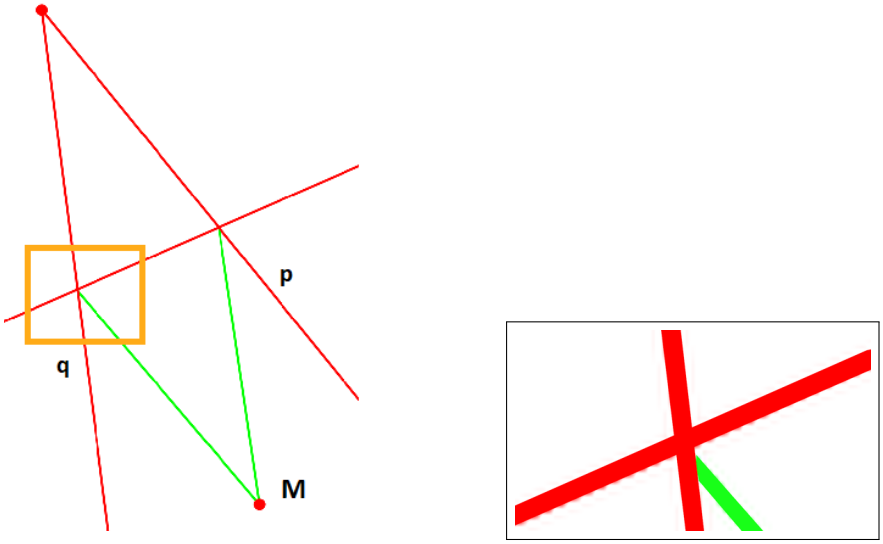


Figure 6.2: Comparison of loss functions for training with and without 4+ training setup (see Section 6.1.3). **Alpha without 4+** : Epochs 0-120 training only heads, epochs 120-200 training the whole network. **Alpha with 4+** : Epochs 0-60 training only heads, epochs 60-120 training only layers 4+ (i.e. backbone stages 4 and 5 and the heads) and epochs 120-200 training the whole network. The loss for training with 4+ layers (blue) is much smoother and less spiky than without 4+ (orange).

6.1.4 Multiple solutions problem

To improve our generation process, we added the fourth degeneration rule in Section 2.6.1: Intersections of geometric primitives can not be too close to the



a) current configuration b) detail of the orange area of subfigure a)

Figure 6.3: A degenerated instance of level *Gamma-04*, with the first step of the construction done. The goal is to find points $X \in p, Y \in q$ such that $|MX| = |MY|$. Both solutions are very close to each other, so it is hard to recognize which solution we are supposed to construct. See detail in the subfigure b. Hence the instance is degenerated. The degenerated configurations should not be in the training data nor in the test data because they are unsolvable based on image data.

construction points. The fourth rule was not applied to the data generation process while training and running inference for models in the sections above. While levels from the Alpha and Beta packs are rarely considered as degenerated by the fourth rule, levels in the Gamma pack and above are considered as degenerated by the fourth rule. Some levels in Gamma are untrainable. Around 27% of *Gamma-04* instances are considered as degenerated by the fourth rule while considered as valid by rules 1.-3. Figure 6.3 we can see an example of this. This change allowed us to train the Gamma and Delta levels. In Table 6.4 we can see results if we add the fourth rule to the data generation. The level *Delta-06*, construction of $\sqrt{2}$, has accuracy equal to 0 because it is very similar to the *Delta-07*, construction of $\sqrt{3}$, and models tend to prefer only one of these two levels.

Gamma levels	Accuracy
01 ChordMidpoint	0.998
02 ATrByOrthocenter	0.65
03 AtrByCircumcenter	0.834
04 AEqualSegments1	0.82
05 CircleTangentPL	0.868
06 TrapezoidCut	0.672
07 Angle45	1
08 Lozenge	0.606
09 CentroidOfQuadrilateral	0.744
Average	**0.799**

Delta levels	Accuracy
01 CDoubleSeq	0.944
02 Angle60Drop	0.438
03 EquilateralAboutCircle	0.952
04 EquilateralInCircle	0.994
05 CutTwoRectangles	0.534
06 Sqrt2	0
07 Sqrt3	0.9
08 Angle15	0.84
09 SquareByOppMidpoints	0.914
10 SquareByAdjMidpoints	0.8415
Average	**0.735**

Table 6.4: Accuracy of models for Gamma and Delta level packs using the fourth degeneration rule from Section 2.6.1: Intersections of geometric primitives can not be too close to the construction points. Note that the models with the fourth rule applied were the first models able to successfully solve Gamma and Delta level packs.

6.1.5 On-the-fly generation

Until now, we were using pre-generated training data that had to be stored in memory.Because the pre-generated levels have to be in memory, thus there is a limit to the size of the training data. The models described above contained 100 000 finished constructions of levels, with the average length of construction around 5 steps, 500 000 training examples had to be stored. To further increase the number of samples, we use on-the-fly generation, so each sample is seen only once during the training. Keep in mind that the history, degeneration checks, and multi-thread training have to be adjusted as well. In Figure 6.4 we compare the loss for models using pre-generated and on-the-fly generation. The loss of the model with on-the-fly generation decreases at a lower rate. The loss function is smoother, and the model achieves better accuracy during inference. The accuracy is equal to 0.991, compared to 0.735 accuracy of the model with a pre-generated dataset. Although a model with on-the-fly generation has a higher loss value, it generalizes better due to the variety of the training data, e.g. the model with pre-generated data is prone to over-fitting.

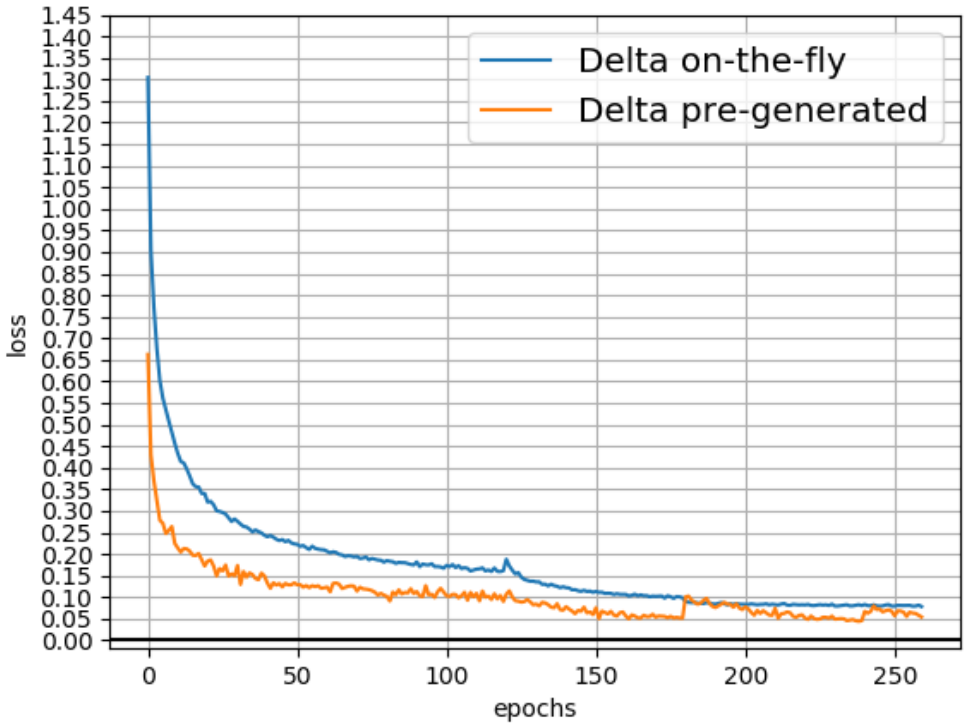


Figure 6.4: Loss of models that use 500k pre-generated training examples (Delta pre-generated) and 3.2M training examples generated on-the-fly (Delta on-the-fly). Both networks were trained in the following setup: Epochs 0-60 training only heads, epochs 60-120 training only layers 4+ (i.e. backbone stages 4, 5 and the heads) and epochs 120-200 training the whole network. Note that while the pre-generated training data are forwarded in the network with in a random order, but samples generated on-the-fly are fed to the network in the order they were created.

6.2 Evaluation of supervised learning approach

With the components described in Section 6.1, we can train models that can solve entire packs or even multiple packs at once. We evaluate our models on the first six level packs (Alpha, Beta, Gamma, Delta, Epsilon, Zeta). All those packs are prepared for data generation, and thus for training.

We trained a model for each level (68) of the first 6 level packs of Euclidea. Each model is trained for 200 epochs. Each epoch consists of 16 000 training samples, e.g. 1 000 training steps of batches of size 16. The model generates data on-the-fly (see Section 6.1.5), so $3.2M$ training samples are seen during the training. In the first 120 epochs, we train only head layers, then 60 epochs of 4+ layers and the last 60 epochs train the whole network (see Section 6.1.3). We initialize the model are with weights from the COCO model, available from Matterport [13]. The trained models have an average 97.7% accuracy, evaluated on 500 instances for each level in the Alpha to Zeta packs. The results are summarized in Table 6.5.

We also trained single model for each level pack. Models for level packs are trained in the same way, and the only difference is the number of epochs. Level pack models are trained for 260 epochs: 120 heads, 60 epochs of 4+ and 80 epochs of the whole network. Level pack models have the following accuracies reported in Table 6.5.

Finally, we trained a single model for all levels. The training setup of this model is the same as for individual models and level pack models. The only difference is the number of epochs. The model was trained for 400 epochs ($6.4M$ training samples): 200 heads, 100 of 4+ and 100 of the whole network. This model has an average 91.8% accuracy and the hypothesis tree search accuracy of 92.1%. Accuracies for each level are in Appendix A.3).

Model / Level Pack	Alpha	Beta	Gamma	Delta	Epsilon	Zeta
one model per level	98.7%	98.6%	97.4%	99.3%	96.3%	96.0%
one model per pack	96.1%	96.2%	97.8%	99.1%	92.8%	95.7%
one model for all levels	90.0%	94.4%	89.4%	96.9%	88.6%	91.5%

Table 6.5: Results of our final models for individual models, level packs and all levels. Each column represents the average accuracy of the model(s) on all levels (500 instances for each level) of the respective pack. **Level models**: One model for each of the 68 levels, accuracy measured only on the corresponding level. **Level pack models**: One model for each of the 6 level packs, measured on the respective pack. **All levels model**: One model used for all levels.

6.3 Evaluation of supervised learning on unseen problems

To evaluate the performance of the model on unseen levels, we use the leave-one-out method with the hypothesis tree search (see Sections 5.2 and 5.5). We use models for all levels and all level packs from the previous chapter. In Figures 6.6 and 6.7 we can see results of the leave-one-out evaluation of the hypothesis tree search. Some levels can be solved this way, but others cannot. To solve an unseen level, parts of the construction have to be in the models we have seen in training, e.g. the first part of the construction can be done with a model one and the second part with another model.

Level	Accuracy	Connections
Alpha		
01 T1 line	0.40	06, 11, 13
02 T2 circle	0.05	05
03 T3 point	1.00	04, 09, 12
04 TIntersect	1.00	03, 09, 13
05 TEquilateral	0.50	02, 03, 06, 07, 09, 11, 12, 13
06 Angle60	0.55	01, 02, 04, 05, 06, 07, 09, 10, 11
07 PerpBisector	0.35	01, 02, 04, 05, 06, 09, 10, 11, 13
09 Midpoint	0.05	02, 10, 12, 13
11 RhombusInRect	0.25	06, 07, 08, 13
12 CircleCenter	0.10	01, 02, 04, 06, 08, 09, 13
Beta		
01 BisectAngle	0.30	04, 05, 06, 07, 08, 11
04 Angle30	0.05	01, 02, 05, 07, 08
07 DropPerp	0.10	01, 02, 04, 08, 11, 12
08 ErectPerp	0.55	01, 02, 03, 04, 06, 07, 10, 11, 12
11 TangentL	0.35	01, 02, 06, 08, 09, 12
Gamma		
01 ChordMidpoint	0.15	02, 03, 04, 05, 07, 08
03 AtrByCircumcenter	0.60	01, 02, 04, 05, 06, 07, 08
07 Angle45	0.35	01, 03, 04, 05, 06, 08, 09
Epsilon		
02 TParallel	0.35	07
03 Parallelogram3V	0.55	01, 02, 04, 05, 06, 07, 08, 10
04 LineAlongPoints	1.00	03, 05, 07
05 LineBetweenPoints	0.10	01, 02, 03, 04, 08, 10
07 ShiftAngle	0.60	01, 02, 03, 04, 05, 08, 11
09 SquareAboutCircle	0.05	02, 04, 05, 06, 08, 12
Zeta		
03 ShiftSegment	0.10	01, 02, 04, 05, 09
06 TCircleByRadius	0.85	07
07 TranslateSegment	0.65	01, 03, 04, 05, 08, 09, 11

Table 6.6: Leave-one-out evaluation of hypothesis tree search for each level pack. Model for **each level** within a pack are inputs for the method, e.g. accuracy on *Alpha-01* using the other Alpha models and so on. The tables shows 30 solved levels, remaining 38 levels were not solved. In the third column of we can see connections. In this case we have connections between level and level, e.g. which level contributed to a solution of the tested level (see Section 5.6).

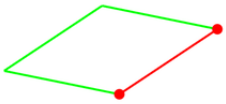
Level	Accuracy	Connections
Alpha		
01 T1 line	0.10	Beta, Delta, Epsilon
02 T2 circle	0.45	Gamma, Delta, Epsilon
03 T3 point	0.90	Beta, Delta, Zeta
04 TIntersect	1.00	Delta, Zeta
05 TEquilateral	0.70	Beta, Gamma, Delta, Epsilon, Zeta
06 Angle60	1.00	Beta, Gamma, Delta, Epsilon, Zeta
07 PerpBisector	1.00	Beta, Gamma, Delta, Epsilon, Zeta
08 TPerpBisector	1.00	Epsilon, Zeta
09 Midpoint	0.60	Beta, Gamma, Delta, Epsilon, Zeta
10 CircleInSquare	1.00	Beta, Gamma, Delta, Epsilon, Zeta
11 RhombusInRect	0.40	Beta, Gamma, Delta, Epsilon
13 SquareInCircle	0.10	Beta, Gamma, Epsilon, Zeta
Beta		
01 BisectAngle	0.80	Alpha, Gamma, Delta, Epsilon, Zeta
02 TBisectAngle	0.55	Gamma
04 Angle30	1.00	Alpha, Gamma, Delta, Epsilon, Zeta
05 DoubleAngle	0.80	Alpha, Gamma, Delta, Epsilon, Zeta
06 CutRectangle	1.00	Alpha, Gamma, Delta, Epsilon, Zeta
07 DropPerp	0.90	Alpha, Delta, Epsilon, Zeta
08 ErectPerp	0.05	Gamma, Delta, Zeta
10 Tangent1	1.00	Alpha, Gamma, Delta, Epsilon, Zeta
11 TangentL	0.65	Alpha, Gamma, Epsilon, Zeta
12 CircleRhombus	0.05	Alpha, Gamma, Zeta
Gamma		
01 ChordMidpoint	0.15	Alpha, Beta, Delta, Epsilon, Zeta
04 AEqualSegments1	0.10	Alpha, Beta, Epsilon, Zeta
06 TrapezoidCut	0.15	Alpha, Delta, Epsilon, Zeta
07 Angle45	1.00	Alpha, Beta, Delta, Epsilon, Zeta
Epsilon		
01 Parallel	0.10	Beta, Gamma, Zeta
03 Parallelogram3V	0.35	Alpha, Beta, Gamma, Delta, Zeta
04 LineAlongPoints	0.40	Alpha, Beta, Gamma, Delta, Zeta
05 LineBetweenPoints	0.25	Alpha, Beta, Gamma, Delta, Zeta
Zeta		
05 CircleByRadius	0.05	Beta, Gamma, Delta, Epsilon

Table 6.7: Leave-one-out evaluation of hypothesis tree search for each level pack. Model for **each pack** are inputs for the method, e.g. accuracy on Alpha with models for Beta-Zeta and so on. The tables shows 31 solved levels, remaining 37 levels were not solved. In the third column of we can see connections. In this case we have connections between level and pack, e.g. which pack contributed to a solution of the tested level (see Section 5.6).

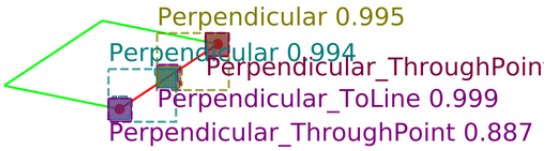
6.4 Example results

In this section, we show Tables with detailed walk-through of constructions for levels *Gamma-08* (Table 6.8), *Delta-10* (Table 6.9) and *Epsilon-12* in (Table 6.10). For each inference, we use a single model trained on all 68 levels from level packs Alpha to Zeta.

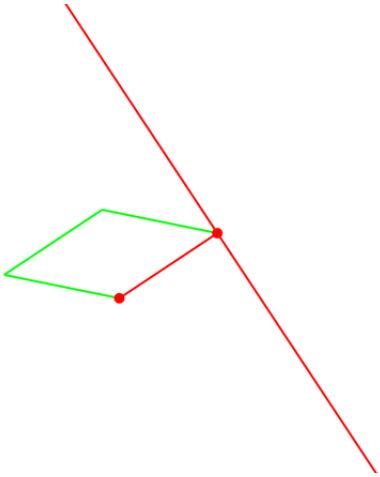
6.4.1 Gamma-08 - Lozenge



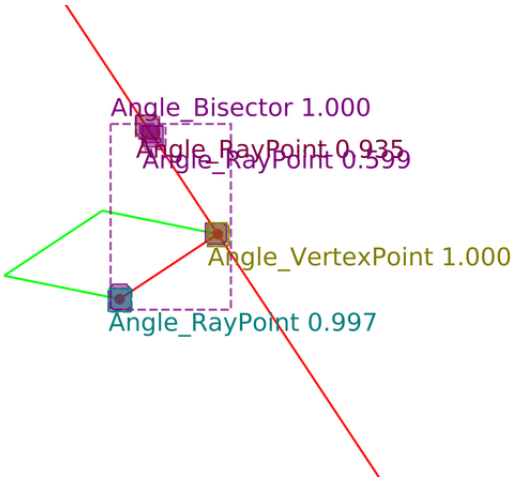
a) Level definition and first input for the network. The goal of this level is to construct a rhombus with a given side and angle of 45° in a vertex. The red denotes the current state of the construction and the green color denotes the remaining goal.



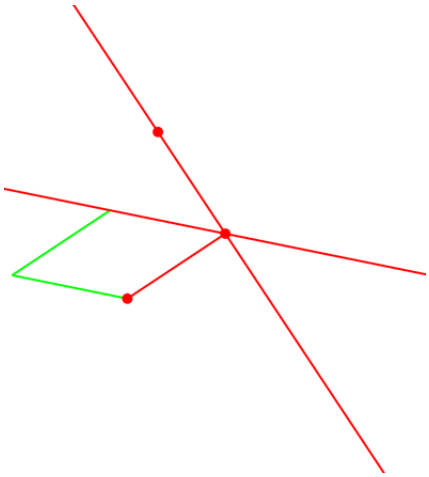
b) Prediction of the network. There are two predictions of the Perpendicular tool. Both can be used, but the top one has a slightly higher Mask R-CNN score and is thus selected.



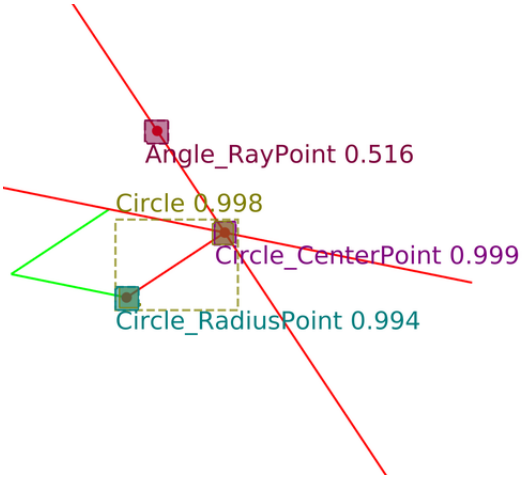
c) Step 1.



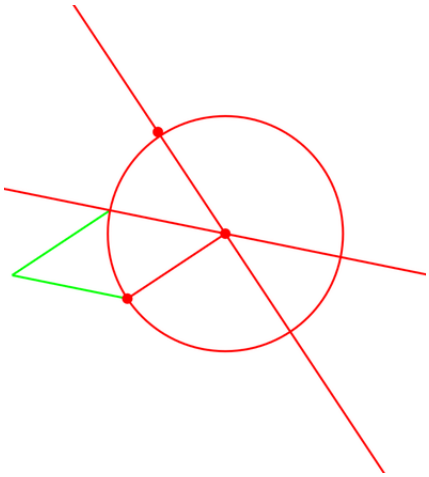
d) Prediction for step 2. Based on this prediction, an angle bisector will be constructed. Note that there are multiple angle ray points on the top ray (purple). Either of them can be used with the same result.



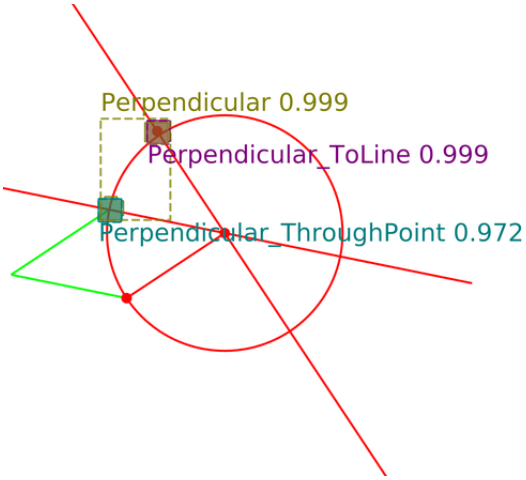
e) Step 2. Previous step constructed part of the goal, one side of the rhombus.



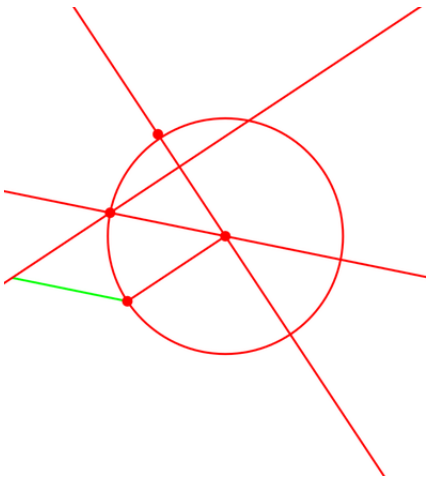
f) Prediction for step 3. Based on this prediction, a circle will be constructed.



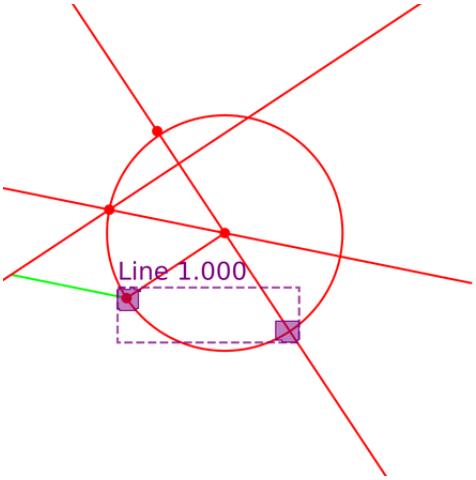
g) Step 3. Note that the point at the top does not lie on the constructed circle.



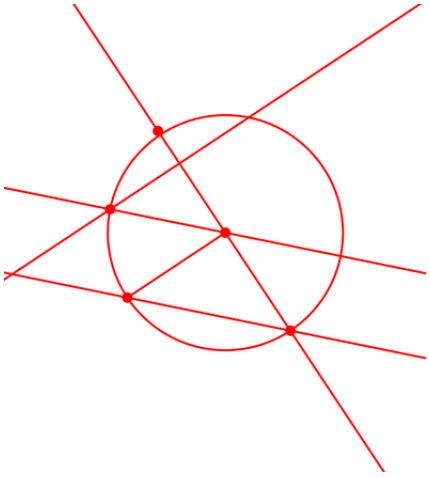
h) Prediction for step 4. Based on this prediction, a perpendicular line will be constructed.



i) Step 4. The perpendicular line constructed another part of the goal.



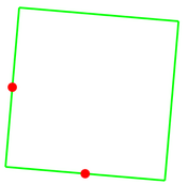
j) Prediction for step 5. Based on this prediction, a line will be constructed.



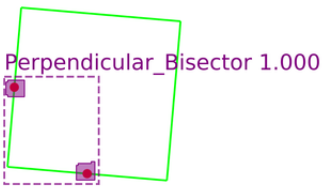
k) Step 5. Level successfully finished, the whole goal has been constructed.

Table 6.8: Example of a computed construction using the all-levels model of level *Gamma-08* (Lozenge). The goal of this level is to construct a rhombus with a given side and angle of 45° in a vertex. The table contains all construction steps of the level. Each step contains the current progress on the left and the Mask R-CNN prediction for a new step on the right. The red denotes the current state of the construction and the green color denotes the remaining goal. Other colors mark prediction masks, bounding boxes, classes and scores for each detected object.

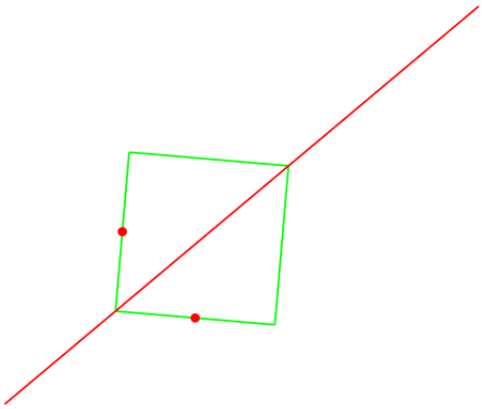
6.4.2 Delta-10 - Square by adjacent midpoints



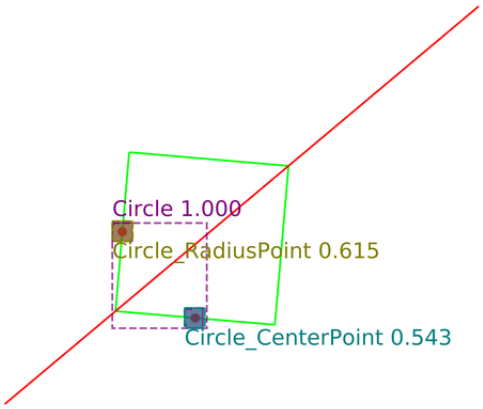
a) Level definition and first input for the network. Construct a square by adjacent side midpoints. The red denotes the current state of the construction and the green color denotes the remaining goal.



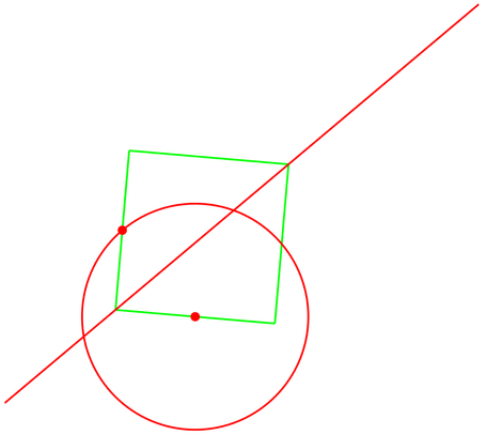
b) Prediction of the network, perpendicular bisector.



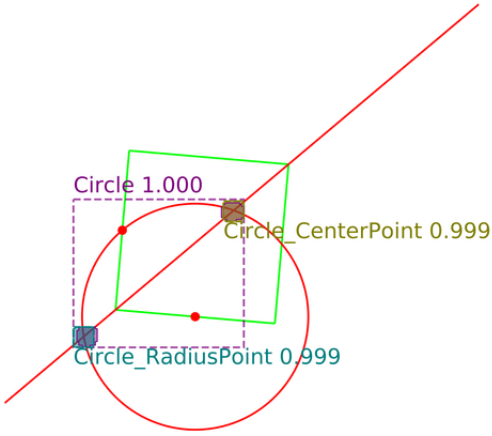
c) Step 1.



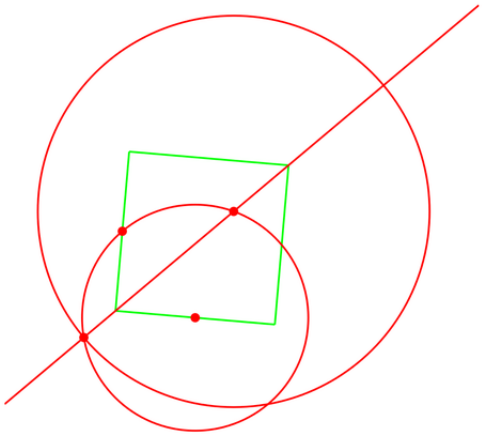
d) Prediction for step 2. Based on this prediction, a circle will be constructed.

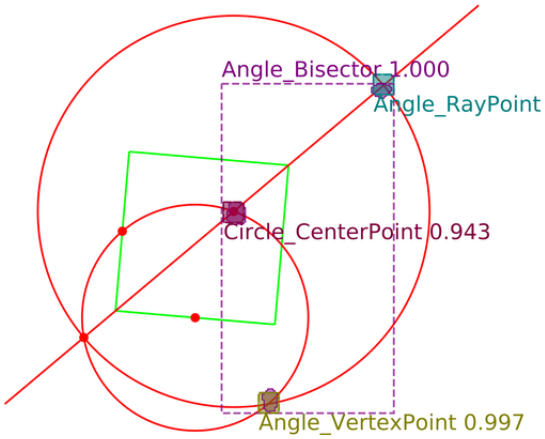


e) Step 2.



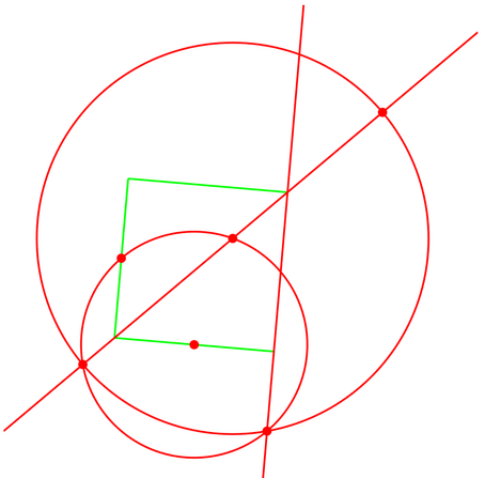
f) Prediction for step 3. Based on this prediction, a circle will be constructed.



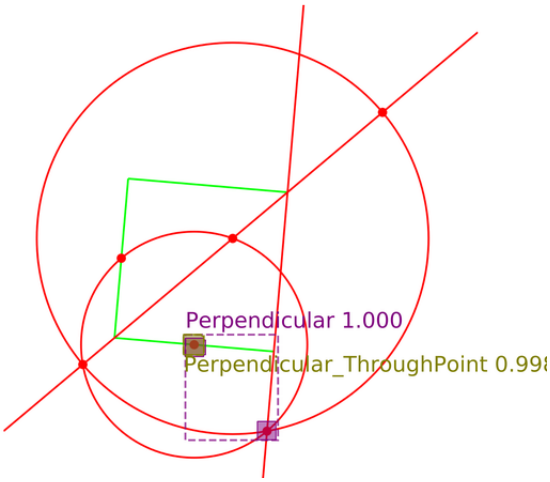


g) Step 3.

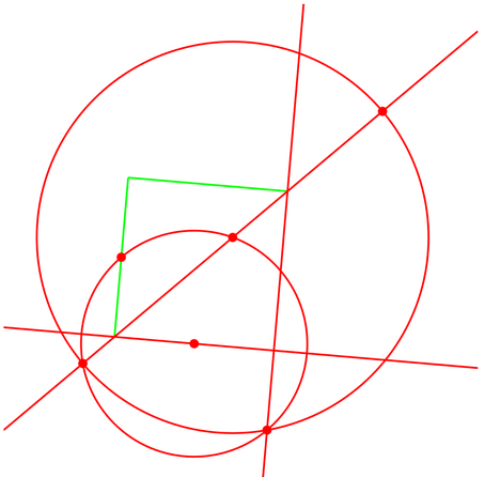
h) Prediction for step 4. Based on this prediction, an angle bisector will be constructed. Note that one ray point is not marked with detection, instead it is marked with the circle center point detection, which is a mistake. This prediction should be in the previous step. However, our model is still able to recognize all points to execute the Angle Bisector tool .



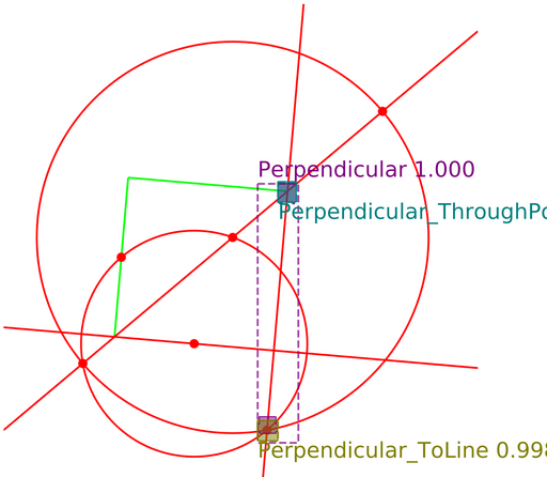
i) Step 4. The angle bisector constructed part of the goal.



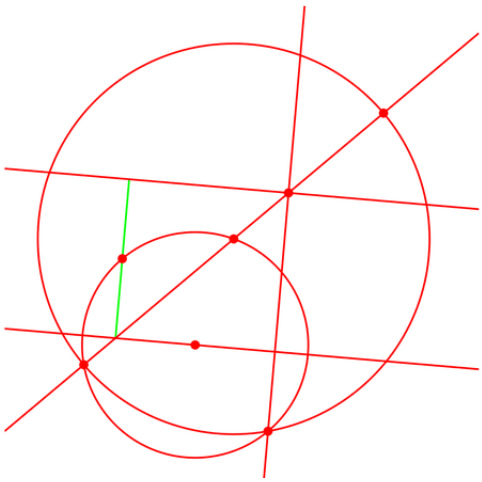
j) Prediction for step 5. Based on this prediction, a perpendicular line will be constructed.



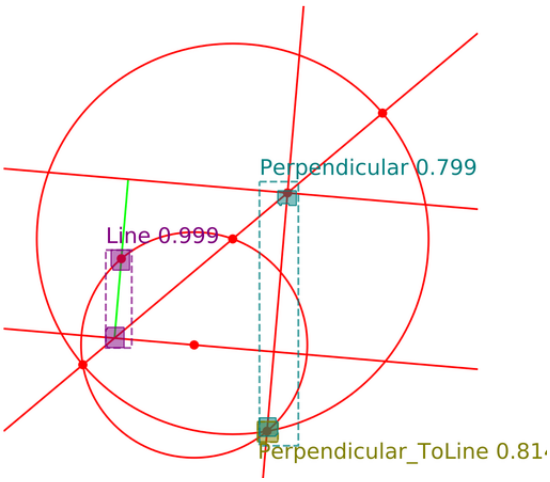
l) Step 5. The perpendicular constructed another part of the goal.



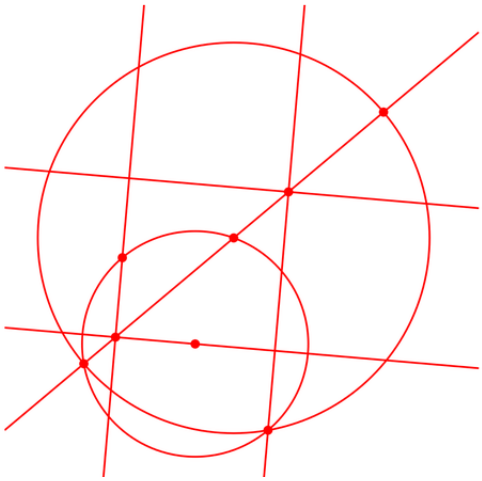
l) Prediction for step 5. Based on this prediction, a perpendicular line will be constructed.



m) Step 6. The perpendicular constructed another part of the goal.



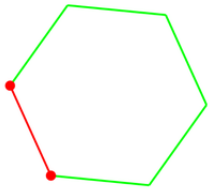
n) Prediction for step 5. Based on this prediction, a perpendicular line will be constructed.



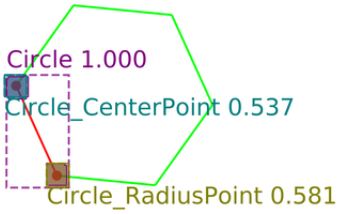
o) Step 7. - Level successfully finished, the whole goal has been constructed.

Table 6.9: Example of a computed construction using the all-levels model of level *Delta-10*: Square by adjacent side midpoints. The table contains all construction steps of the level. Each step contains current progress on the left and Mask R-CNN prediction for a new step on the right. The red denotes the current state of the construction and the green color denotes the remaining goal. Other colors mark prediction masks, bounding boxes, classes and scores for each detected object.

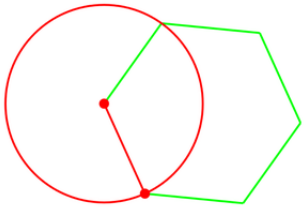
6.4.3 Epsilon-12 - Regular hexagon by the side



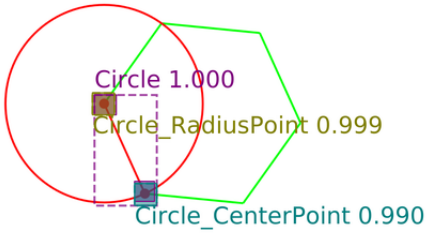
a) Level definition and first input for the network. Construct a regular hexagon given by the side. The green color denotes the goal and the red denotes the current state of the construction.



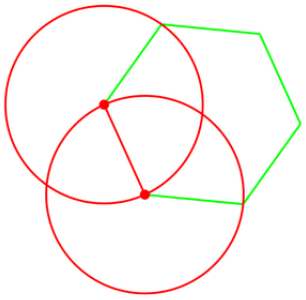
b) Prediction of the network.Based on this prediction, a circle will be constructed.



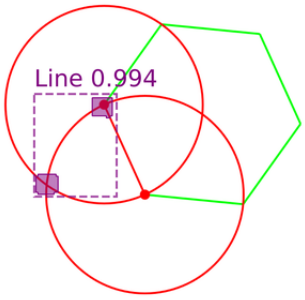
c) Step 1.



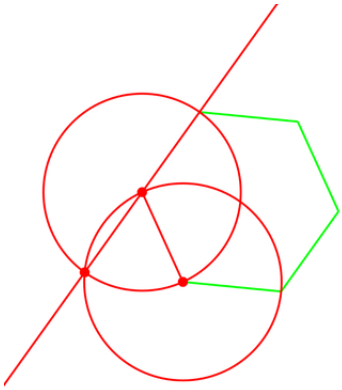
d) Prediction for step 2. Based on this prediction, a line will be constructed.



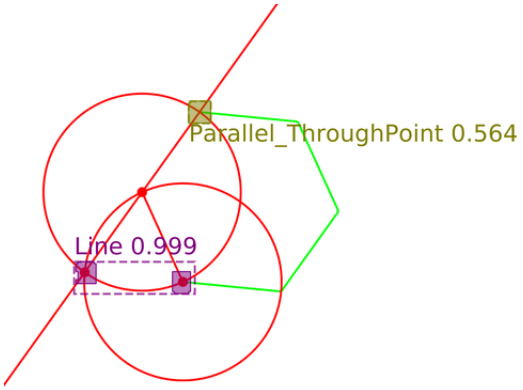
e) Step 2.



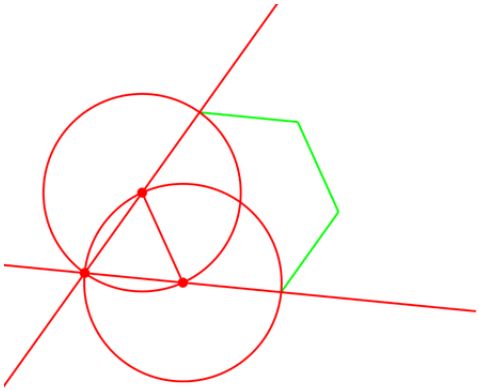
f) Prediction for step 3. Based on this prediction, a line will be constructed.



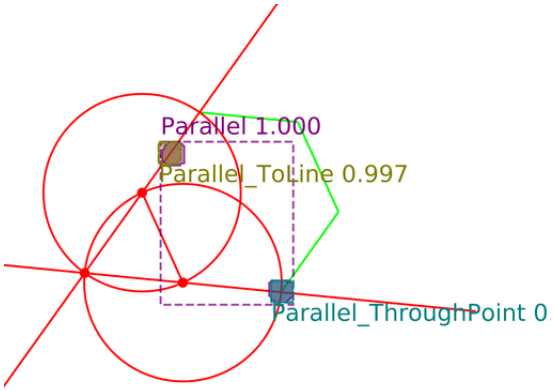
g) Step 3.



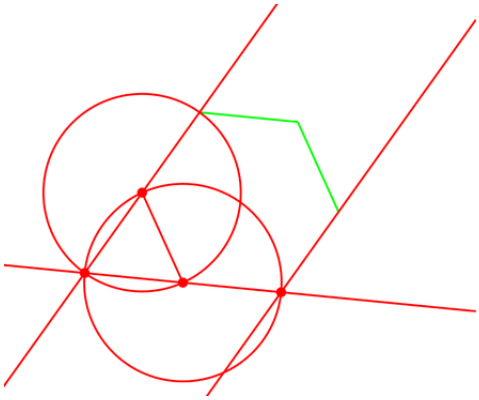
h) Prediction for step 4. Based on this prediction, an angle bisector will be constructed. Note that there is an extra detection of parallel through point. Later in construction score of this prediction will increase and the prediction will be used.



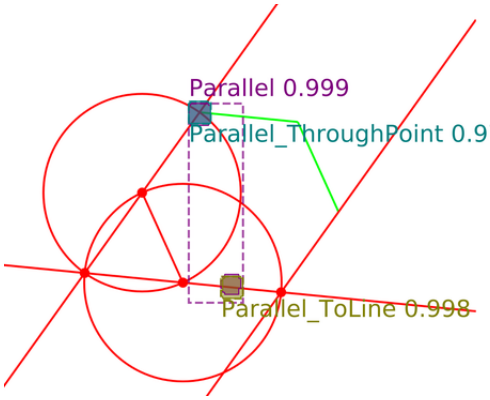
i) Step 4. The parallel line constructed part of the goal.



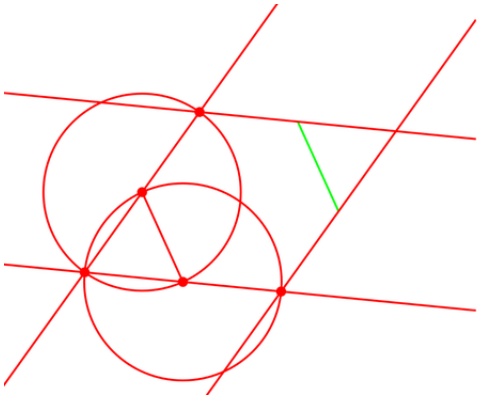
j) Prediction for step 5. Based on this prediction, a parallel line will be constructed.



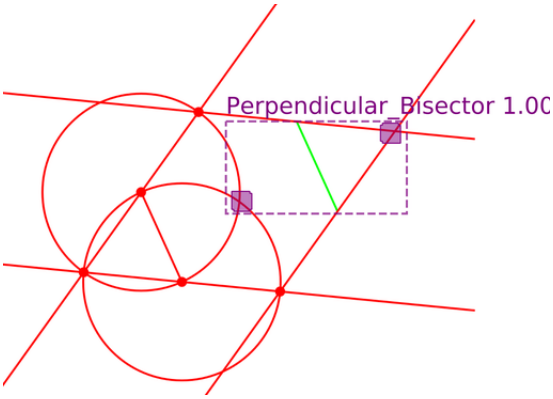
l) Step 5. The parallel line constructed another part of the goal.



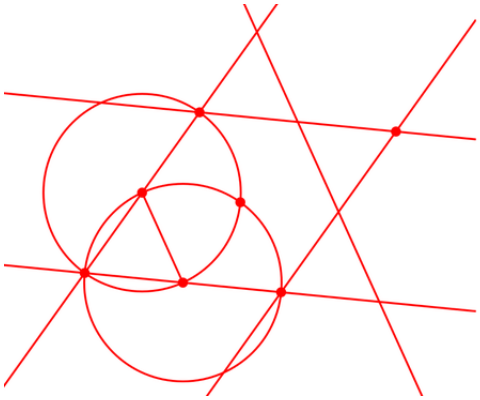
l) Prediction for step 5. Based on this prediction, a parallel line will be constructed.



m) Step 6. The perpendicular bisector constructed another part of the goal.



n) Prediction for step 5. Based on this prediction, a perpendicular bisector will be constructed.



o) Step 7 - level successfully finished, the whole goal has been constructed.

Table 6.10: Example of a computed construction using the all-levels model of level *Epsilon-12*: Regular hexagon by the side. The table contains all construction steps of the level. Each step contains current progress on the left and Mask R-CNN prediction for a new step on the right. The red denotes the current state of the construction and the green color denotes the remaining goal. Other colors mark prediction masks, bounding boxes, classes and scores for each detected object.

7. Conclusion

We developed a model for solving geometric construction problems. The model is based on Mask R-CNN. To test the model, we used Euclidea, a construction game with geometric problems levels of a various difficulties. To train the model, we created a data generator that generates new configurations of Euclidea levels. We trained a model that can solve the first 6 level packs of Euclidea, i.e. 68 different levels/problems, with 92% accuracy. We describe multiple components of this model and experimentally demonstrated their benefits. Then we analyzed accuracy of trained models on unseen levels with the leave-one-out method. Our models were able to solve 31 out of 68 levels. Our models were unable to solve 37 level, which is caused by the fact that some levels are similar, and respective models can substitute one another. Levels without similarity to other levels could not be solved. Because most of the unsolved levels require approach/construction not seen in other levels, we believe that a model trained on a more general set of levels could solve them.

7.1 Contributions of the thesis

Below we summarize the contributions of this thesis:

- We have described how to generate new level configurations that are suitable for the training of the model (see Chapter 2).
- We have described how to modify the Mask R-CNN to solve geometric problems, how to build training data and how to gather actions/clicks from the Mask R-CNN output (see Chapter 4).
- We have described hypothesis tree search, a tree search within hypotheses given by the Mask R-CNN model, and developed a program for an interactive exploration of the hypotheses (see Chapter 5).
- We have shown the results of the developed approach. We described multiple components of this model and experimentally demonstrate their benefits. In the supervised approach, we achieved an accuracy of 92% with a single model on 68 levels. In the evaluation of the unseen levels with the leave-one-out method, we were able to solve 31 out of 68 levels (see Chapter 6).
- The source code for this thesis was developed in python. Neural networks are developed using the Tensorflow package. Our source code contains training and testing scripts for the main model, an interactive program for hypothesis exploration, exhaustive tree search for constructions, and many more problem visualization scripts. The source code is avaliable at the project page [16].

7.2 Future work

In the following list, we propose possible future steps with a potential to improve the inference accuracy.

1. **More Euclidea levels**
   Prepare more packs for the training and train them. Construction for each level has to be prepared for random level generation. More levels could provide more general models for solving unseen levels. Furthermore, we could investigate the limitations of the supervised approach, how many levels could be learned at once.

2. **Tool mask as additional input**
   When we solve levels in Euclidea, we also know which tools are allowed whereas our model does not have this information. However, this information could help the model to distinguish different levels and thus improve model accuracy.

3. **Reinforcement learning**
   We experimented with the reinforcement learning (RL), but we could not learn even basic levels due to the problem complexity. The problem has ample search space, and we would also have to learn object detection, which often takes a lot of training time, and combined with RL would take even longer. However, RL methods can be applied to the output of the Mask R-CNN. Furthermore, RL methods can decrease the search time of the hypothesis search. Nevertheless, this approach cannot solve levels that miss necessary hypotheses.

4. **Allow Move tool**
   We use the Move tool only to generate new instances, but this tool could also be useful for inference. Levels that cannot be solved could be transformed to a different configuration that the model can finish. However, training of the Move tool is problematic since it is hard to define when it is beneficial to use.
   We could also solve Euclidea by finding transformation to a normalized configuration. We could have a model trained to find a sequence of point moves that transform a level instance to a normalized configuration. We could then apply these point moves and simply solve the normalized configuration, which could be memorized. Then we would apply reverse moves to obtain the original configuration of the level.

Bibliography

[1] Gergelitsová Š. and Holan T. Ancient problems vs. modern technology. *Matfyzpress*, 2019.

[2] Botana F., Hohenwarter M., and Janičić P. et al. Automated theorem proving in GeoGebra: Current achievements. *Journal of Automated Reasoning*, 2015.

[3] Stojanovic Durdevic S., Narboux J., and Janicic P. Automated Generation of Machine Verifiable and Readable Proofs: A Case Study of Tarski's Geometry. *Annals of Mathematics and Artificial Intelligence*, 73, 2015.

[4] Seo M., Hajishirzi H., Farhadi A., Etzioni O., and Malcolm C. Solving geometry problems: Combining text and diagram interpretation. Association for Computational Linguistics, 2015.

[5] Kaliszyk C. and Urban J. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 2012.

[6] Hales T., Adams M., Bauer G., Dang T. D., Harrison J., Le Truong H., Kaliszyk C., Magron V., Mclaughlin S., Tat Thang N., and et al. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 2017.

[7] He K., Gkioxari G., Dollár P., and Girshick R. Mask R-CNN. *International Conference on Computer Vision*, 2017.

[8] Girdhar R., Gkioxari G., Torresani L., Paluri M., and Tran D. Detect-and-track: Efficient pose estimation in videos. *Conference on Computer Vision and Pattern Recognition*, 2017.

[9] Girdhar R., Carreira J., Doersch C., and Zisserman A. Video action transformer network. *Computer Vision and Pattern Recognition*, 2019.

[10] Li Z., Sedlar J., Carpentier J., Laptev I., Mansard N., and Sivic J. Estimating 3d motion and forces of person-object interactions from monocular video. *Computer Vision and Pattern Recognition*, 2019.

[11] Euclidea. `https://www.euclidea.xyz`.

[12] Olsak M. Python version of the Euclidea game for possible reinforcement learning agents (https://euclidea.xyz). `https://github.com/mirefek/py_euclidea/`.

[13] Abdulla W. Mask R-CNN for object detection and instance segmentation on keras and tensorflow. `https://github.com/matterport/Mask_RCNN`, 2017.

[14] Mask R-CNN schema. `https://ronjian.github.io/assets/Mask-RCNN/mask-rcnn.png`.

[15] Fischler M. A. and Bolles R. C. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24, 1981.

[16] Macke J. Learning to solve geometric construction problems from images. https://github.com/mackej/Learning-to-solve-geometric-construction-problems-from-images, 2021.

A. Appendix

A.1 Chapter 2: Level descriptions

Level	Our Name	Level Description
Alpha 01	01 T1 line	Tutorial level for the Line tool.
Alpha 02	02 T2 circle	Tutorial level for the Circle tool.
Alpha 03	03 T3 point	Tutorial level for the Point tool.
Alpha 04	04 TIntersect	Tutorial level for the Intersection tool.
Alpha 05	05 TEquilateral	Tutorial level multiple tools. Construct an equilateral triangle.
Alpha 06	06 Angle60	Construct an angle of 60° with given side.
Alpha 07	07 PerpBisector	Construct the perpendicular bisector of the segment.
Alpha 08	08 TPerpBisector	Tutorial level for the Perpendicular bisector tool.
Alpha 09	09 Midpoint	Construct the midpoint of the segment defined by two points.
Alpha 10	10 CircleInSquare	Inscribe a circle in the square.
Alpha 11	11 RhombusInRect	Inscribe a rhombus in the rectangle so that they share a diagonal.
Alpha 12	12 CircleCenter	Construct the center of the circle
Alpha 13	13 SquareInCircle	Inscribe a square in the circle. One vertex of the square is given.
Beta 01	01 BisectAngle	Construct the line that bisects the given angle.
Beta 02	02 TBisectAngle	Tutorial level for the Perpendicular bisector tool.
Beta 03	03 Incenter	Construct the point where the angle bisectors of the triangle are intersected.
Beta 04	04 Angle30	Construct an angle of 30° with given side.
Beta 05	05 DoubleAngle	Construct an angle equal to the given one so that they share one side.
Beta 06	06 CutRectangle	Construct a line through the given point that cuts the rectangle into two parts of equal area.
Beta 07	07 DropPerp	Drop a perpendicular from the point to the line.
Beta 08	08 ErectPerp	Erect a perpendicular from the point on the line.
Beta 09	09 TDropPerp	Tutorial level for the Perpendicular tool.
Beta 10	10 Tangent1	Construct a tangent to the circle at the given point.

Level	Our Name	Level Description
Beta 11	11 TangentL	Construct a circle with the given center that is tangent to the given line.
Beta 12	12 CircleRhombus	Inscribe circle in the rhombus.
Gamma 01	01 ChordMidpoint	Construct a chord whose midpoint is given.
Gamma 02	02 ATrByOrthocenter	Construct a segment connecting the sides of the angle to get triangle whose orthocenter is in the given point.
Gamma 03	03 AtrByCircumcenter	Construct a segment connecting the sides of the angle to get triangle whose perpendicular bisectors are intersected in the given point.
Gamma 05	05 CircleTangentPL	Construct a circle through the point A that is tangent to the given line at the point B.
Gamma 06	06 TrapezoidCut	Construct a line passing through the midpoints of the trapezoid bases.
Gamma 07	07 Angle45	Construct an angle of 45° with the given side.
Gamma 08	08 Lozenge	Construct a rhombus with the given side and an angle of 45° in a vertex.
Gamma 09	09 CentroidOfQuadrilateral	Construct the midpoint of the segment that connects the midpoints of the diagonals of the quadrilateral.
Delta 01	01 CDoubleSeq	Construct a point C on the line AB such that $\|AC\| = 2\|AB\|$ using only compass.
Delta 02	02 Angle60Drop	Construct a straight line through the given point that makes angle of 60° with the given line.
Delta 03	03 EquilateralAboutCircle	Construct an equilateral triangle that is circumscribed about the circle and contains the given point.
Delta 04	04 EquilateralInCircle	Inscribe an equilateral triangle in the circle using the given point as a vertex. The center of the circle is not given.
Delta 05	05 CutTwoRectangles	Construct a line that cuts each of the rectangles into two parts of equal area.
Delta 06	06 Sqrt2	Let $\|AB\| = 1$. Construct a point C on the ray AB such that the length of AC is equal to square root of 2.
Delta 07	07 Sqrt3	Let $\|AB\| = 1$. Construct a point C on the ray AB such that the length of AC is equal to square root of 3.
Delta 08	08 Angle15	Construct an angle of 15° with the given side.
Epsilon 01	01 Parallel	Construct a line parallel to the given line through the given point.
Epsilon 02	02 TParallel	Tutorial level for the Parallel tool.
Epsilon 03	03 Parallelogram3V	Construct a parallelogram whose three of four vertices are given

Level	Our Name	Level Description
Epsilon 04	04 LineAlongPoints	Construct a line through the point C and at equal distance from the points A and B but that does not pass between them.
Epsilon 05	05 LineBetweenPoints	Construct a line through the point C that goes between the points A and B and that is at equal distance from them.
Epsilon 06	06 Hash	Construct a line through the given point on which two pairs of parallel lines cut off equal line segments.
Epsilon 07	07 ShiftAngle	Construct an angle from the given point that is equal to the given angle so that their sides are parallel.
Epsilon 08	08 EquidistantParallel	Construct a straight line parallel to the given parallel lines that lies at equal distance from them.
Epsilon 09	09 SquareAboutCircle	Circumscribe a square about the circle. Two of its sides should be parallel to the given line.
Epsilon 10	10 SquareInSquare	Inscribe a square in the square. A vertex is given.
Epsilon 11	11 CircleInOutSquare	Construct a circle that is tangent to a side of the square and goes through the vertices of the opposite side.
Epsilon 12	12 HexagonBySide	Construct a regular hexagon with the given side.
Zeta 01	01 PtSymmetry	Reflect the segment across the point.
Zeta 02	02 MirrorSeq	Reflect the segment across the line.
Zeta 03	03 ShiftSegment	Construct a segment from the given point that is equal to the given segment and lies on the same line with it.
Zeta 04	04 GivenAngleBisector	Construct two straight lines through the two given points respectively so that the given line is a bisector of the angle that they make.
Zeta 05	05 CircleByRadius	Construct a circle with the given center and the radius equal to the length of the given segment.
Zeta 06	06 TCircleByRadius	Tutorial level for the Compass tool.
Zeta 07	07 TranslateSegment	Construct a segment from the given point parallel and equal to the given segment.
Zeta 08	08 TriangleBySides	Construct a triangle with the side AB and the two other sides equal to the given segments.
Zeta 09	09 ParallelogramBySP	Construct a parallelogram with the given side an the midpoint of the opposite side in the given point.

Level	Our Name	Level Description
Zeta 10	10 9PointCircle	Construct a circle that passes through the midpoints of sides of the given acute triangle.
Zeta 11	11 4SymmetricLines	Three lines are intersected in a point. Construct a line so that the set of all 4 lines is mirror symmetric.
Zeta 12	12 ParallelogramBy3Midpoints	Construct a parallelogram given three of the midpoints.

Table A.1: Description for Alpha to Zeta levels of Euclidea. In the first column is a numbered level name, in the second column is the name used in our version of Euclidea and the third column is a level description.

A.2 Chapter 3: Branching factors

Beta levels	**Successful search**	**Branching factor (estimate)**
01 BisectAngle	True	19.00 - 49.00 - 94.00 - 0.03 (27 - 70 - 133 - 216)
02 TBisectAngle	True	1.00 (12)
03 Incenter	False	180.00 - 371.52 (195 - 389)
04 Angle30	True	11.00 - 61.00 - 174.00 - 267.50 - 0.00 (17 - 74 - 195 - 287 - 644)
05 DoubleAngle	False	34.00 - 114.00 - 269.00 - 389.00 - 752.71 (39 - 125 - 287 - 404 - 780)
06 CutRectangle	False	520.00 - 701.00 - 1032.90 (549 - 725 - 1050)
07 DropPerp	True	11.00 - 61.00 - 174.00 - 267.00 - 0.03 (17 - 74 - 195 - 287 - 578)
08 ErectPerp	True	11.00 - 1.00 (17 - 74)
09 TDropPerp	True	1.00 (4)
10 Tangent1	True	27.00 - 0.08 (48 - 128)
11 TangentL	True	13.00 - 59.50 - 0.00 (21 - 75 - 263)
12 CircleRhombus	False	416.00 - 569.00 - 755.00 - 978.00 - 1410.88 (468 - 630 - 825 - 1056 - 1515)

Table A.2: Branching factors of Beta levels, first 10k nodes visited. The first column show Euclidea level. The second column indicates whether the search completed the construction successfully (True/False). The third column gives the average branching factor (top) at each depth of the search and its estimate in the parenthesis (bottom).

Delta levels	Successful search	Branching factor (estimate)
01 CDoubleSeq	True	2.0 - 6.0 - 13.0 - 30.0 - 55.0 - 0.1 (7 - 15 - 26 - 57 - 100 - 149)
02 Angle60Drop	False	13.0 - 65.0 - 180.0 - 274.0 - 540.9 (21 - 90 - 231 - 336 - 636)
03 EquilateralAboutCircle	False	27.0 - 61.0 - 114.0 - 268.0 - 523.0 - 809.5 - 0.0 (48 - 90 - 150 - 336 - 630 - 942 - 1706)
04 EquilateralInCircle	False	11.0 - 61.0 - 174.0 - 267.0 - 388.0 - 536.0 - 719.0 - 1279.9 (21 - 90 - 231 - 336 - 468 - 630 - 825 - 1474)
05 CutTwoRectangles	False	2688.0 - 3184.0 - 3737.0 - 4348.0 - 5022.0 - 5761.0 (2856 - 3366 - 3933 - 4560 - 5250 - 6006)
06 Sqrt2	False	30.0 - 66.0 - 120.0 - 277.5 (48 - 90 - 150 - 336)
07 Sqrt3	False	30.0 - 66.0 - 120.0 - 276.0 - 533.0 (48 - 90 - 150 - 336 - 630)
08 Angle15	False	13.0 - 65.0 - 180.0 - 274.0 - 395.0 (21 - 90 - 231 - 336 - 468)
09 SquareByOppMidpoints	False	6.0 - 26.0 - 61.0 - 174.3 - 0.0 - 1.0 (21 - 48 - 90 - 231 - 532 - 468)
10 SquareByAdjMidpoints	False	6.0 - 26.0 - 61.00 - 174.00 - 375.0 - 522.0 - 704.0 - 919.0 - 1175.0 (21 - 48 - 90 - 231 - 468 - 630 - 825 - 1056 - 1326)

Table A.3: Branching factors of Delta levels, first 10k nodes visited. The first column show Euclidea level. The second column indicates whether the search completed the construction successfully (True/False). The third column gives the average branching factor (top) at each depth of the search and its estimate in the parenthesis (bottom).

Epsilon levels	**Successful search**	**Branching factor (estimate)**
01 Parallel	True	13.0 - 0.2 (21 - 79)
02 TParallel	True	1.0 (4)
03 Parallelogram3V	True	1.0 - 1.0 - 1.0 - 1.0 (57 - 106 - 175 - 267)
04 LineAlongPoints	True	21.0 - 0.1 (57 - 106)
05 LineBetweenPoints	False	21.0 - 54.0 - 129.0 (57 - 106 - 207)
06 Hash	False	166.0 - 340.0 - 612.0 (175 - 385 - 711)
07 ShiftAngle	True	1.0 - 1.0 (106 - 175)
08 EquidistantParallel	False	24.0 - 127.0 - 283.0 - 535.0 - 909.0 (25 - 175 - 385 - 711 - 1177)
09 SquareAboutCircle	False	39.0 - 121.0 - 279.0 - 399.0 - 551.0 - 733.0 - 0.0 (57 - 175 - 385 - 532 - 711 - 925 - 1773)
10 SquareInSquare	False	592.0 - 782.0 - 1008.0 - 1270.0 - 1575.0 - 1925.0 (711 - 925 - 1177 - 1470 - 1807 - 2191)
11 CircleInOutSquare	False	448.0 - 607.0 - 799.0 - 1024.0 (532 - 711 - 925 - 1177)
12 HexagonBySide	False	33.0 - 70.0 - 125.0 - 284.0 - 542.0 - 724.0 - 944.0 (57 - 106 - 175 - 385 - 711 - 925 - 1177)

Table A.4: Branching factors of Epsilon levels, first 10k nodes visited. The first column show Euclidea level. The second column indicates whether the search completed the construction successfully (True/False). The third column gives the average branching factor (top) at each depth of the search and its estimate in the parenthesis (bottom).

Zeta levels	**Successful search**	**Branching factor (estimate)**
01 PtSymmetry	False	63.0 - 190.0 - 411.0 (106 - 267 - 532)
02 MirrorSeq	False	79.0 - 212.0 - 439.0 - 787.0 - 1031.0 - 1317.0 (106 - 267 - 532 - 925 - 1177 - 1470)
03 ShiftSegment	False	63.0 - 190.0 - 411.0 - 753.0 (106 - 267 - 532 - 925)
04 GivenAngleBisector	False	33.0 - 125.0 - 300.0 - 582.0 - 788.0 - 1035.0 (57 - 175 - 385 - 711 - 925 - 1177)
05 CircleByRadius	False	63.0 - 190.0 - 411.0 - 753.0 (106 - 267 - 532 - 925)
06 TCircleByRadius	True	1.0 (28)
07 TranslateSegment	False	1.0 - 185.0 - 467.0 (134 - 230 - 539)
08 TriangleBySides	False	909.0 - 1614.0 - 2605.0 (1044 - 1793 - 2834)
09 ParallelogramBySP	False	1.0 - 185.0 - 467.0 - 939.0 (134 - 230 - 539 - 1044)
10 9PointCircle	False	324.0 - 507.0 - 743.0 - 1038.0 - 1769.0 - 2788.0 - 4141.0 - 4997.0 (363 - 539 - 764 - 1044 - 1793 - 2834 - 4215 - 5048)
11 4SymmetricLines	False	132.0 - 502.0 - 1280.0 - 2628.0 - 3952.0 - 5657.0 (134 - 539 - 1385 - 2834 - 4215 - 5984)
12 ParallelogramBy3Midpoints	False	30.0 - 90.0 - 185.0 - 321.0 - 699.0 (69 - 134 - 230 - 363 - 764)

Table A.5: Branching factors of Zeta levels, first 10k nodes visited. The first column show Euclidea level. The second column indicates whether the search completed the construction successfully (True/False). The third column gives the average branching factor (top) at each depth of the search and its estimate in the parenthesis (bottom).

A.3 Chapter 6: Inference

In the following Table A.6 we can see inference result of the all levels models on levels from Alpha to Zeta.

Level	Accuracy	Average branching factor (estimate)
Alpha		
01 T1 line	0.85	0.96 (13.00)
02 T2 circle	1.00	1.00 (4.00)
03 T3 point	1.00	1.00 (28.33)
04 TIntersect	0.99	0.99 (50.00)
05 TEquilateral	1.00	1.00 (47.50)
06 Angle60	0.94	1.16 (71.83)
07 PerpBisector	0.99	1.00 (37.33)
08 TPerpBisector	0.75	0.75 (6.00)
09 Midpoint	1.00	1.40 (39.00)
10 CircleInSquare	0.87	1.08 (298.67)
11 RhombusInRect	0.99	1.17 (416.67)
12 CircleCenter	1.00	1.00 (47.67)
13 SquareInCircle	1.00	1.00 (91.17)
Beta		
01 BisectAngle	1.00	1.19 (100.67)
02 TBisectAngle	0.99	0.99 (55.00)
03 Incenter	0.89	0.96 (358.75)
04 Angle30	1.00	1.00 (187.33)
05 DoubleAngle	0.92	0.89 (460.00)
06 CutRectangle	0.96	1.18 (1512.00)
07 DropPerp	0.98	1.30 (272.33)
08 ErectPerp	1.00	1.37 (144.00)
09 TDropPerp	0.90	0.90 (16.00)
10 Tangent1	1.00	1.00 (69.00)
11 TangentL	1.00	1.00 (120.00)
12 CircleRhombus	0.85	1.21 (1410.00)
Gamma		
01 ChordMidpoint	1.00	1.00 (120.00)
02 ATrByOrthocenter	1.00	1.29 (591.00)
03 AtrByCircumcenter	0.96	0.98 (498.00)
04 AEqualSegments1	0.96	1.31 (696.00)
05 CircleTangentPL	1.00	1.09 (255.00)
06 TrapezoidCut	0.96	1.26 (1245.00)
07 Angle45	0.99	0.99 (69.00)
08 Lozenge	0.82	1.20 (426.00)
09 CentroidOfQuadrilateral	0.95	1.49 (1292.25)
Delta		
01 CDoubleSeq	1.00	1.00 (22.00)
02 Angle60Drop	0.98	1.17 (390.00)

03 EquilateralAboutCircle	1.00	1.03 (171.00)
04 EquilateralInCircle	1.00	1.01 (220.50)
05 CutTwoRectangles	0.98	1.48 (8405.25)
06 Sqrt2	1.00	1.05 (220.50)
07 Sqrt3	1.00	1.24 (279.00)
08 Angle15	0.97	0.95 (279.00)
09 SquareByOppMidpoints	0.95	1.03 (311.00)
10 SquareByAdjMidpoints	0.93	0.97 (801.00)
Epsilon		
01 Parallel	0.70	0.90 (157.00)
02 TParallel	0.53	0.53 (16.00)
03 Parallelogram3V	0.86	1.23 (198.00)
04 LineAlongPoints	0.98	1.02 (112.67)
05 LineBetweenPoints	0.96	1.59 (198.00)
06 Hash	0.83	1.46 (963.00)
07 ShiftAngle	0.98	1.04 (394.67)
08 EquidistantParallel	0.90	2.04 (596.00)
09 SquareAboutCircle	0.94	0.98 (909.25)
10 SquareInSquare	0.98	1.59 (2441.00)
11 CircleInOutSquare	1.00	1.00 (722.67)
12 HexagonBySide	0.99	0.88 (956.00)
Zeta		
01 PtSymmetry	0.47	1.31 (443.00)
02 MirrorSeq	0.23	1.40 (1209.00)
03 ShiftSegment	0.99	1.15 (866.67)
04 GivenAngleBisector	0.96	1.43 (827.67)
05 CircleByRadius	0.92	1.39 (362.67)
06 TCircleByRadius	1.00	1.00 (28.00)
07 TranslateSegment	0.53	1.10 (406.00)
08 TriangleBySides	0.96	1.22 (4764.83)
09 ParallelogramBySP	0.88	1.74 (637.00)
10 9PointCircle	1.00	1.39 (819.00)
11 4SymmetricLines	0.95	1.08 (1764.00)
12 ParallelogramBy3Midpoints	0.77	1.87 (859.50)

Table A.6: Evaluation of the all levels model, all 68 levels Alpha to Zeta trained. The hypothesis tree search is used for evaluation. In the first column, we can see level, accuracy on that level in the second column, and in the last column the average branching factor and its estimate in the parenthesis. Note that averages under 1 mean that some instance has no hypothesis.