university of groningen

faculty of science and engineering

MASTERS THESIS

From Knowledge Graph to Cognitive Model: A Method for Identifying Task Skills

Author:
Ivana Akrum, BSc

Supervisors:
C. Hoekstra, MSc
Prof. Dr. N.A. Taatgen

A thesis submitted in fulfilment of the requirements for the degree of Master of Science in

Computational Cognitive Science

July 18, 2022

UNIVERSITY OF GRONINGEN

Abstract

From Knowledge Graph to Cognitive Model:
A Method for Identifying Task Skills
by Ivana Akrum, BSc

When we learn new tasks, rather than starting from scratch, we often reuse skills that we have learned previously. By integrating these previously learned skills in a new way, we can learn how to do new tasks with little effort. In this research, I test a method aimed at identifying the skills reused between tasks. More specifically, I use a knowledge graph as a tool for identifying reused skills. From this knowledge graph, I built a cognitive model that shows how the identified skills can be integrated to solve a new task. The final cognitive model could successfully solve a variety of related but distinct tasks. This shows it is possible to use knowledge graphs to identify the skills reused between tasks. This ability could have far-reaching implications: Knowing, in advance, the skills needed to successfully complete a new task may allow us to learn said task in an easier, more focused manner. Although the validity of the described method should be further examined, the method provides a promising step towards revolutionising education and how we approach learning.

Acknowledgements

Generally, I am a very independent person. I do my work, and when I come across a pitfall, I try to solve the problem by myself first and foremost. In a project such as this, where you are expected to make your own decisions on how you tackle your work, proper guidance is nonetheless unmissable. Even as an independent worker, I would not have been able to produce this thesis without the help of my supervisors. I would like to thank Niels Taatgen primarily for always being available to brainstorm with me. His quick correspondence was well-appreciated and although our meetings often went overtime, Niels made the time for me. His input was invaluable and I appreciate it greatly.

I furthermore want to thank my second supervisor Corné Hoekstra for being there to offer a fresh point of view. This was especially important in the cases where Niels and I were so well-aligned, we could not see what the project would like from an outside point of view. Explaining my project and my decisions to Corné helped me tremendously in making sense of it all. Importantly, it helped me make sure my reasoning was sound and that my explanations were accessible.

Finally, I would like to thank my boyfriend for his support throughout my project. When things got tough, he helped me centre myself. Thank you.

Contents

1 Introduction

Life is a perpetual cycle of growth. Each time we learn something new, we are guided by our experiences of the past (Middleton & Baartman, 2013; National Research Council, 2000; Thorndike, 1914/1999). The new experience, in turn, moves on to be a building stone for yet another learning opportunity, and in this way, our minds continue to expand for the majority of our lives (Anokhin et al., 1996; Costandi, 2016; Kempermann, 2006).

To optimise the learning process, it would be helpful to know which of a person's previous experiences are relevant when learning a new task. With a wide array of previous experiences, it is unlikely that all will be relevant for any one task. If, however, it were possible to somehow identify the skills that are being reused across tasks, this would greatly simplify the learning process. Learning a new task would become a matter of comparing the skills one already possesses from previous experiences to those they have yet to learn.

To that end, this thesis proposes a new method for identifying the reused skills underlying tasks. In particular the method proposed in this thesis relies on using knowledge graphs as a tool for identifying the skills reused between tasks. By building a cognitive model from the skills identified by the knowledge graph, this thesis aims to answer the research question: "Can knowledge graphs be used to identify the skills reused between tasks?"

If a cognitive model can successfully be built on the basis of the skills identified by the knowledge graph, it is suggested that indeed, knowledge graphs can be used to identify the skills reused between tasks.

In the rest of this thesis, the full method is outlined for using knowledge graphs in this novel way. Particularly, the thesis will begin with an overview of the theoretical background. As will be shown, this theory neatly aligns with the proposed method in this thesis, while also highlighting the importance of furthering our understanding of skill reuse between tasks.

After the theoretical background leading up to this thesis has been established, the rest of the thesis will focus on using knowledge graphs as a tool for identifying task skills. In Section 3 *Problem Context*, the data set that was used in this thesis is given. An overview is given of what the data looked like, and it is explained how the data was processed for the subsequent data analysis. The data analysis procedure is discussed further in Section 4. This section outlines how the knowledge graph was created and what the final knowledge graph looked like. Section 5 goes on to explain how the information from the knowledge graph was used to create a set of two cognitive models. After an analysis of the results of the cognitive models, the *Discussion* couples these results back to the research question proposed here. At the end of this section, some concluding remarks are given about the quality of the thesis and the future of the proposed method therein.

2 Theoretical Background

2.1 What Is a Skill?

If the aim of this thesis is to identify the skills underlying tasks, then it is imperative we begin by establishing a sound and concise definition of what a skill is. In *The skill approach in education: From theory to practice*, Güneş (2018) gives one such definition. She explains how the definition of a skill has changed over time from a "collection of behaviours" to "a collection of knowledge and cognitive processes" (p. 2). This shift in the concept of what a skill is closely aligns with the equivalent shift in the field of psychology as a whole, which moved away from behaviourism in the late 1900s in favour of the cognitive movement (Miller, 2003). Following this shift in its definition, Güneş (2018) concludes that a skill can be defined as "the ability to transfer knowledge into practice to perform a task or duty" (p. 4).

This definition touches upon some interesting characteristics of a skill. Firstly, it highlights that there is some knowledge component to a skill, but that this knowledge component on its own does not define a skill. In fact, Güneş is neither the first nor the last to propose this distinction between knowledge and skills.

In long-term memory, for example, it has long been proposed that factual knowledge is stored separately from procedural knowledge in different memory systems (Marilee, 1999). A distinction is made between semantic memory, the general information one knows about the world, and procedural memory, memories about sequences of actions that work towards some goal. Equating this to the definition of a skill as given by Güneş (2018), the idea of semantic memory matches more closely to Güneş' definition of knowledge, while the idea of procedural memory aligns better with her definition of a skill. Not only does the definition of procedural memory align with Güneş' definition of a skill, it also concretizes this definition further. It moves away from a vague sense of "some ability" to a concrete set of actions: procedures. Procedures are instructions on how to turn theoretical knowledge into practice, and as such, it is possible to define skills in terms of procedural knowledge.

In fact, many cognitive architectures (frameworks for unified theories of cognition; Newell, 1990) take precisely this approach to defining skills. From SOAR (Newell, 1990) to ACT-R (Anderson et al., 2004) to EPIC (Kieras & Meyer, 1997), cognitive models written in these architectures have the same commonality: they all express skills through procedures or *production rules* (Chong et al., 2007; Kotseruba & Tsotsos, 2020). Such production rules typically consist of conditions and actions. They trigger when their conditions are met, and once triggered, they execute their associated actions.

Yet despite of the fact that many cognitive architectures thus implicitly define skills through production rules, those same architectures do not make this definition explicit. There is no explicit mention of 'skills' at all in either ACT-R, SOAR, or EPIC (Anderson et al., 2004; Kieras & Meyer, 1997; Newell, 1990). Rather, the production rules work directly towards achieving the desired task, and there is no need for defining how the production rules relate to the concept of skills. For the development of these cognitive architectures, that is arguably a worry less, but for defining what a skill is, it is clearly a cause for concern.

Luckily, there are cognitive architectures that do explicitly define skills. Two examples of such architectures are PRIMs (Taatgen, 2013) and ICARUS (Langley & Choi, 2006). In PRIMs, skills are defined by a set of production rules (which are called *operators* in the PRIMs terminology). Listing 1 shows an example of a PRIMs skills. Specifically, the example is showing the iterate skill, which is one of four skills that make up a counting model (a model that can count from 1 up to 10). The skill is intentionally general, so that it is also applicable outside the counting model it was designed for. In the listing, the content of the operators is intentionally left out, but each operator consists of a set of conditions and a set of actions (as per the description of a production rule). This explicit skill definition re-iterates the idea of defining skills through procedural knowledge, but it is not the only way skills can be defined.

Listing 1: Schematic of an example skill in the PRIMs architecture, adapted from Taatgen (2022).

```
define skill iterate {
// Start iteration
  operator start-iteration {
  ...
   }

// Do something with the current items
  operator do-sub-skill {
  ...
  }

// Iterate as long as the goal is not met
  operator retrieve-next {
  ...
  }

// Stop when reaching final match
  operator final {
  ...
  }

// If the iteration fails, do something else
  operator final-fail {
  ...
  }
}
```

To give an alternative, In ICARUS, skills are characterised by their objective (Langley & Choi, 2006). Once an ICARUS skill has been executed, it will have had a predefined effect on the state of its model. Through that effect, the skill achieves its objective. However, not all ICARUS skills can be executed. ICARUS makes a crucial distinction between primitive and non-primitive skills. Only the primitive skills can be executed, and they are defined by a set of actions that occur once their conditions are met (and are thus comparable to production rules).

The non-primitive skills cannot be executed. Rather than actions, these skills are defined through sub-goals that must be achieved in a certain order to actualize the skill's objective. Each of the sub-goals, in turn, corresponds to the objective of a different, predefined skill. The objective of a non-primitive skill can thus be achieved by going through the skills that can realize its sub-goals until a primitive skill is reached that can be executed.

The distinction made between primitive and non-primitive skills is particularly interesting because it addresses the problem of scope. It is useful to be able to define a skill through procedures. However, in defining skills in this way, another question becomes evident. Namely: How do we know how many procedures go into one skill? Which procedures belong to which skills, and, more generally, how can the scope of a skill be determined? ICARUS tackles this problem by making a distinction between primitive and non-primitive skills. PRIMs uses a different approach.

Originally, PRIMs was designed to address the phenomenon of skill transference (Taatgen, 2013). It has been hypothesised since the times of Ancient Greece that there is some type of transfer that occurs between tasks. Plato (257BC/2003, as cited by Inglis & Attridge, 2017) surmised that the study of mathematics also helps students develop their general thinking abilities. Later on, Thorndike (1914/1999) insisted that such transfer only occurs if the knowledge elements between the different tasks are identical. In his work, Thorndike gives various examples of situations that would and would not incur transfer, according to this theory of identical elements. Singley & Anderson (1985) further investigated transfer in the context of text-editing in the hopes of better understanding the identical elements proposed by Thorndike. They postulate that production rules form this identical element, and that transfer occurs when two tasks rely on the same set of production rules. Finally, we return to Taatgen (2013), which argues that the production rules are too specific to form Thorndike's identical element. Cognitive architectures that rely solely on production rules, Taatgen argues, are proficient at showing the cognitive processes behind single tasks but are not designed to show how skills can be reused between tasks. Taatgen himself proposes the *primitive information processing elements* (PRIMs) as the unit of cognitive skills and designed the PRIMs architecture specifically to study skill transfer in more detail.

Regardless of what the identical element is that causes transfer to happen between tasks, what all these theories have in common is that they show, in the first place, that there is such a thing as transfer between tasks. Although the exact make-up of a skill (down to its smallest element) is still under debate, what is unequivocal is that skills *are* reused between tasks. Not only that, there is also a consensus in the literature that skills, at some dimension, consist of procedures. As such, this thesis will not attempt to define skills down to their smallest unit. Instead, a skill is defined as **the largest unit of procedural knowledge that can be reused between tasks** (Hoekstra et al., 2020). This definition of skill will be upheld throughout this thesis.

2.2 Why Identify Reused Skills?

Having defined what a skill is, the next question to address is: How can we identify which skills are reused between tasks? This is an important question to tackle, because research shows that skill transfer does not always have a positive effect on the main task.

A famous example of this is the Stroop effect. By presenting participants with two concurrent but conflicting stimuli, Stroop (1935) showed how two tasks (identifying a colour and reading a word) can effect each other negatively. There exist a variety of theories as for why this negative effect occurs (see e.g. Stirling, 1979, or Cohen et al., 1990).

One interesting theory is given by Besner et al. (1997). Although the main purpose of Besner et al. was to contest the theory of automaticity as an explanation for the Stroop effect, in their concluding remarks, they state that, evidently, the literal context of the task is more powerful than the instructions the participants were given. Potentially attributing the Stroop effect to the transfer appropriate processing theory (Morris et al., 1977), Besner et al. hypothesise that the task context of the Stroop task activates the "mental set" for reading rather than for identifying colours. Of course, they concede that this theory begets more questions than it does answers, on accounts of the need to then identify exactly what such a mental set might look like.

This thesis argues that mental set is characterised (at least partially) by skills. Evidence for this theory comes from Hoekstra et al. (2020), who investigated skill reuse within the attentional blink paradigm. The attentional blink is a phenomenon that has been observed to occur in Rapid Serial Visual Presentation (RSVP) tasks. Raymond et al. (1992) showed that participants were likely to miss a probe if it was presented between 180 to 450 milliseconds after a target stimuli within a visual stream of distractors. This phenomenon was dubbed the 'attentional blink'.

In lieu of the discovery of the attentional blink, Ferlazzo et al. (2007) found that the phenomenon could be mitigated through strategy. By manipulating the experimental instructions participants were given, they showed (over a series of three experiments) that the attentional blink could be reduced by promoting participants to use one strategy over another. In this way, the occurrence of the attentional blink can be likened to the Stroop effect, in that both phenomena are evidently influenced by the task context.

The question that naturally arises from this influence of the task context is 'why?' Why does a change in instructions have an effect on whether negative transfer does or does not occur? As per Hoekstra et al. (2020), one explanation is that instructions influence the set of skills that participants choose to use. Through a cognitive model built in PRIMs, Hoekstra et al. show that the occurrence of the attentional blink is dependent on the skills the model reuses from other tasks. Thus, negative transfer could be explained by people reusing skills that are not beneficial for the new task context.

Following this possible explanation, it becomes evident that it is quite important to know which skills people reuse between tasks. After all, the choice of reused skills could make the difference between succeeding or failing at a task.

Since skills are fundamentally a cognitive phenomenon (or, a collection of knowledge and cognitive processes, as per Güneş, 2018), it makes sense that they are often identified through cognitive models. However, as was explained in the previous subsection, most cognitive models do not make the skills that go into a task explicit. They rely rather on sets of instructions like procedures to explain how tasks can be completed. Although there are such cognitive architectures that do make skills explicit (and in doing so, make it easier to model transfer), there is at least one limitation to identifying skills through cognitive modelling alone.

The work of Hoekstra et al. (2020) highlights this limitation well. The attentional blink model Hoekstra et al. create is made up of four general skills. These general skills were identified on the basis of previous work done on the attentional blink paradigm. Since the attentional blink is a well-known phenomenon, there is a lot of literature that dissects the cognitive and neuronal basis of the attentional blink, and as such, the model built by Hoekstra et al. has a strong theoretical basis. Yet what of tasks that lack such a strong body of literature? In those cases, and from a general perspective as well, modellers will have to make careful considerations about what skills they include in their cognitive models, and what theories they base these skills on.

Mainly, cognitive models are examples of deductive research. They are built off existing theories about how certain cognitive phenomena work. Through the cognitive model, the existing theories can be tested by comparing the performance of the model against human performance. In this way, cognitive models (that simulate human intelligence) are designed to test existing theories, rather than generate new ones. What this means within the context of skills is that a cognitive modeller does not know in advance what skills should go into their model. The purpose of the model is rather to test a set of skills, once they have been identified through a different method.

Finding a method for identifying reused skills is thus not only useful from a learning perspective, it could also play an important role for skill-based cognitive modelling.

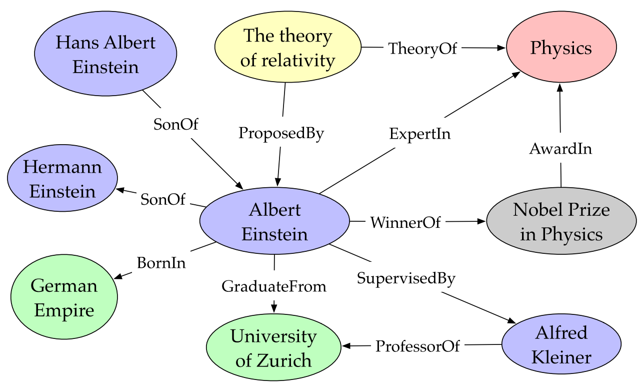
2.3 How to Identify Reused Skills?

If cognitive models are not an appropriate tool for identifying skills, then what other methods could be used for this purpose? Returning to the definition of a skill, it was claimed that a skill could be defined through procedural knowledge. While procedural knowledge can indeed be modelled through cognitive models, knowledge overall can also be represented by *knowledge graphs*.

Knowledge graphs, or knowledge bases (depending on whether there is a graphical or textual representation), are commonly-used methods for providing structure to knowledge (Ji et al., 2022). Figure 1 show how these methods achieve that. Factual knowledge is broken down into a subject, an object, and a predicate that connects these two. Knowledge bases then represent this knowledge in the form of triplets, while knowledge graphs use a visual representation that is shaped like a graph structure.

(Albert Einstein, **BornIn**, German Empire)
(Albert Einstein, **SonOf**, Hermann Einstein)
(Albert Einstein, **GraduateFrom**, University of Zurich)
(Albert Einstein, **WinnerOf**, Nobel Prize in Physics)
(Albert Einstein, **ExpertIn**, Physics)
(Nobel Prize in Physics, **AwardIn**, Physics)
(The theory of relativity, **TheoryOf**, Physics)
(Albert Einstein, **SupervisedBy**, Alfred Kleiner)
(Alfred Kleiner, **ProfessorOf**, University of Zurich)
(The theory of relativity, **ProposedBy**, Albert Einstein)
(Hans Albert Einstein, **SonOf**, Albert Einstein)

(a) Knowledge base.



(b) Knowledge graph.

Figure 1: Generic examples of knowledge representation through knowledge graphs/bases (Ji et al., 2022). These examples represent factual knowledge on Albert Einstein.

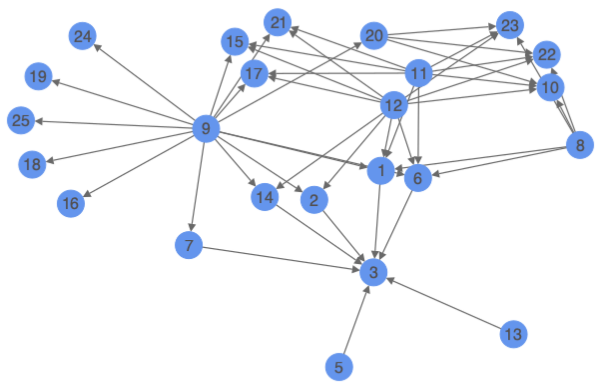
Although Figure 1 thus shows that knowledge graphs can be used to represent factual knowledge, the interest of this thesis is in procedural knowledge rather than factual knowledge. Thankfully, Falmagne et al. (1990) show that knowledge graphs can also be used to represent *knowledge states*. Falmagne et al. describe a knowledge state as the set of problems an individual is capable of solving. Following this definition, the knowledge space of a specific individual is defined as the collection of all their knowledge states. Although Falmagne at al. do not attach a cognitive interpretation to their concepts of knowledge space and knowledge states, they do concede that it is possible to interpret a knowledge space in terms of its underlying skills.

In this interpretation, it is assumed that each problem has an underlying set of skills that are required to solve said problem. This interpretation gives way to the idea of a hierarchy among the problems. If an individual has all the skills to solve a problem p, and a problem p' relies on that same set of skills, then observing that the individual can solve the problem p will automatically imply that they should also be able to solve the problem p'.

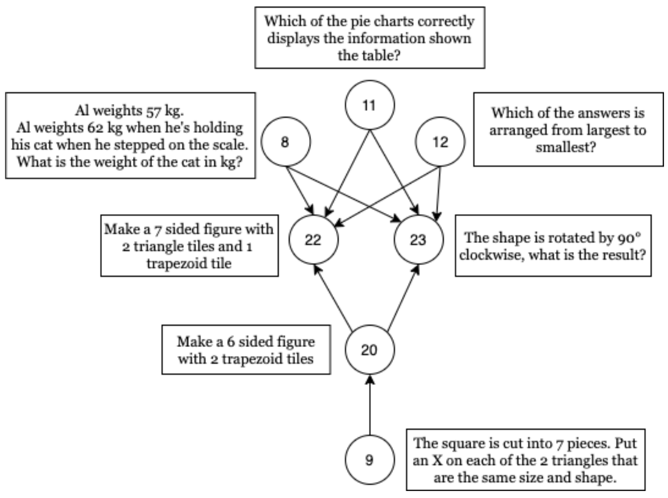
If the concept of a 'problem' is replaced by that of a task, then it is evident from this interpretation of the knowledge space that identifying the skills reused between tasks requires one to first identify the relationship between the tasks themselves. In other words, it is necessary to first build the knowledge space, before it is possible to make inferences about the skills underlying the knowledge space.

Falmagne et al. (1990) propose that the knowledge space can be built by a combination of expert knowledge and empirical evidence. For the expert knowledge, they propose that an expert answers the question: Does failing task t entail that an individual will also fail task t'? In lieu of (or in addition to) an expert, the empirical evidence can be used to answer this question. Performance data can be analysed to see which tasks are often solved together and which tasks do not appear to be related.

Rozestraten (2021) achieves precisely this. Rozestraten creates an algorithm that uses accuracy data to determine the hierarchy between tasks. Specifically, she looks at three different data sets, each of which contains a variety of mathematical problems. For each data set, Rozestraten generates a knowledge graph that represents the hierarchy between the problems.



(a) The knowledge graph Rozestraten (2021) generated on the basis of the TIMSS data set (International Association for the Evaluation of Educational Achievement (IEA), 2008).


Which of the pie charts correctly displays the information shown the table?
Al weights 57 kg. Al weights 62 kg when he's holding his cat when he stepped on the scale. What is the weight of the cat in kg?
8
11
12
Which of the answers is arranged from largest to smallest?
Make a 7 sided figure with 2 triangle tiles and 1 trapezoid tile
22
23
The shape is rotated by 90° clockwise, what is the result?
Make a 6 sided figure with 2 trapezoid tiles
20
9
The square is cut into 7 pieces. Put an X on each of the 2 triangles that are the same size and shape.

(b) A subset of some of the problems in the knowledge graph (2a) and their relationships.

Figure 2: Some results from Rozestraten (2021).

Figure 2 shows the results for one of the three data sets Rozestraten (2021) analysed. Figure 2a shows the knowledge graph that was generated on the basis of the TIMSS data set (International Association for the Evaluation of Educational Achievement (IEA), 2008). Each node represents a problem in the data. Figure 2b takes a closer look at a specific subset of the problems and includes the actual problems students saw for reference.

While Figure 2a clearly demonstrates the hierarchy between the questions through directed edges, Figure 2b is especially useful for the interpretation of this hierarchy. It is visible that question 9 is required to solve question 20, and that question 20, in turn, is required to solve questions 22 and 23. Other questions that contribute to questions 22 and 23 are questions 8, 11, and 12. From a skill perspective, this means that some of the skills from questions 20, 8, 11, and 12 are all reused to solve questions 22 and 23.

The hierarchical property as explained by Falmagne et al. (1990) is also visible in the results from Rozestraten (2021). After all, it can be concluded from Figure 2b that someone that could e.g. solve question 20 could also solve question 9. The underlying explanation for this is that question 9 evidently relies on a subset of the skills that question 20 relies on.

Rozenstraten's analysis already goes a long way in achieving the outcome desired from this thesis. Figure 2 shows that knowledge graphs can be used to a) create a knowledge space and b) show how tasks reuse skills. Rozestraten herself concludes, however, that knowledge graphs are not sufficient for identifying the skills reused between tasks. They can show the task hierarchy, but from this, concrete skills cannot be identified (Rozestraten, 2021).

In this thesis, I propose that this limitation of knowledge graphs might be overcome by combining the knowledge graph approach with cognitive modelling. If skills can be identified on the basis of knowledge graphs generated from performance data, then cognitive models may be used to validate these skills. In this way, it might be possible to use knowledge graphs after all to identify the skills reused between tasks.

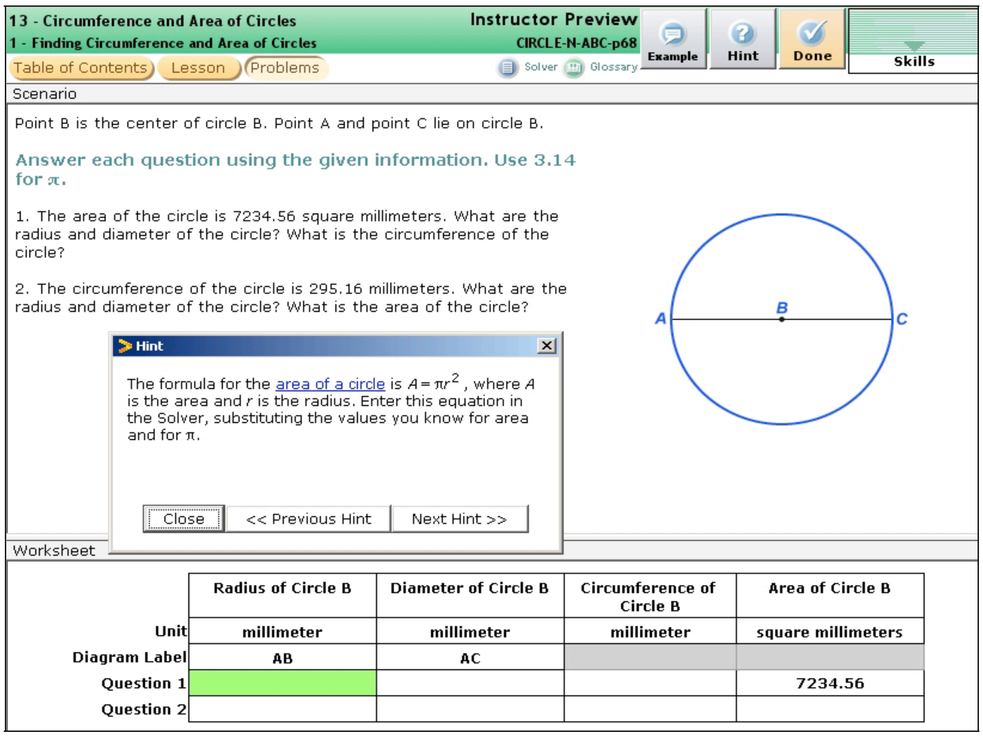
To answer the research question "Can knowledge graphs be used to identify the skills reused between tasks?", I will build onto the work done by Rozestraten (2021). A new version is created of the knowledge graph algorithm (the algorithm that generates the knowledge graph), which is subsequently tested with a new data set. Skills will be identified on the basis of the generated knowledge graph (if possible), and they will be validated using cognitive models. If the method proposed in this thesis shows that knowledge graphs can be used as effective tools to identify the skills reused between tasks, this would not only be beneficial for how people approach learning, it would also help guide cognitive modellers in their task of building skill-based cognitive models.

3 Problem Context

3.1 Description of the Dataset

As was explained in the *Theoretical Background* transfer happens between a variety of tasks, both in positive and negative contexts. Given this property of transfer, the hope is that the method described in this thesis will be applicable for a variety of task domains. Nonetheless, for the creation and testing of the method, a specific task domain must be chosen. Ideally, the chosen task domain should be small in scope with a very specific focus. It is hypothesised that tasks in more focused task domains reuse a smaller number of skills between them, making those skills easier to identify. Since the skills identified by the knowledge graph must also be validated through cognitive models, it would be useful to only tackle a small set of skills for the sake of feasibility. If the method proposed in this thesis is shown to work well for a small task domain that contains relatively little skills, future work could investigate its use for larger task domains as well.

Given the desire to first test the described method on a smaller task domain, this thesis focuses on the domain of geometry. Specifically, the knowledge graph method is tested using the 'Geometry Area (1996-97)' data set, which is available for public use via DataShop (Koedinger et al., 2010). The data set contains data from the area unit of a Geometry Cognitive Tutor course that was given to students in the school year 1996 to 1997. Figure 3 shows a more recent version of that Geometry Cognitive Tutor course. The program offers students an environment for solving various geometry problems, while simultaneously keeping track of their cognitive progress (i.e. what factual/procedural knowledge they possess at any given point in time).



Point B is the center of circle B. Point A and point C lie on circle B.

Answer each question using the given information. Use 3.14 for π.

1. The area of the circle is 7234.56 square millimeters. What are the radius and diameter of the circle? What is the circumference of the circle?

2. The circumference of the circle is 295.16 millimeters. What are the radius and diameter of the circle? What is the area of the circle?

The formula for the area of a circle is $A = \pi r^2$, where A is the area and r is the radius. Enter this equation in the Solver, substituting the values you know for area and for π.

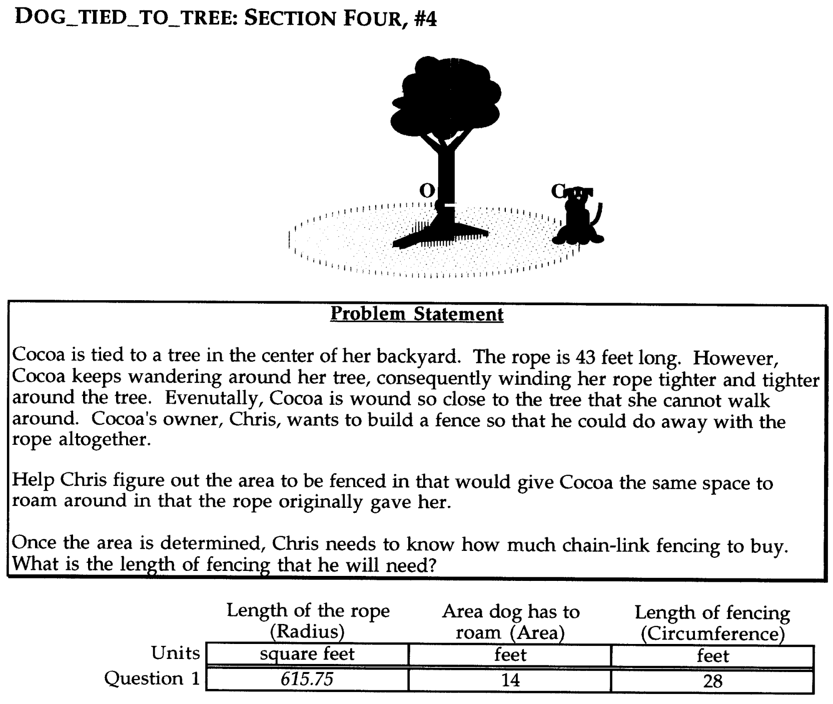
	Radius of Circle B	Diameter of Circle B	Circumference of Circle B	Area of Circle B
Unit	millimeter	millimeter	millimeter	square millimeters
Diagram Label	AB	AC		
Question 1	🟩			7234.56
Question 2				

Figure 3: A screenshot of (a more recent version) of the Geometry Cognitive Tutor course that was used to generate the Geometry Area (1996-97) data set (Koedinger et al., 2012).

Figure 3 furthermore shows the cognitive tutor offers a hints system. In the generation of the Geometry Area (1996-97) data set, this hint function was not used. Instead, students would be shown the correct answer if they answered a problem incorrectly. They would then get another attempt at the problem at a later point in time.

In total, the data set contains performance data of 59 students on 40 problems. Each problem consists of one or multiple questions. These questions, in turn, consist of one or multiple steps, as depicted in Figure 3 in the worksheet (the table at the bottom of the figure). Each column in the worksheet corresponds to one step in the question. For some of the steps, the answer is already given in the problem description. In that case, filling in the step is only a matter of copying the provided information. The data set only contains performance data for those steps that students had to solve themselves. In total, the data set contained 78 steps.

These 78 steps were not unique to the questions or the problems. For example, Question 1 in Figure 3 is asking students to calculate the circumference of the depicted circle. The associated step, according to the naming conventions used in the data set, would be "Circumference Question 1". However, "Circumference Question 2" and "Circumference Question 3" also exist in the data, showing that the circumference step is not unique to question 1. Similarly, the "Circumference Question 1" step can be asked for the problem `CIRCLE-N-ABC` (depicted in Figure 3), but it could also occur for a different circle problem, such as the `DOG_TIED_TO_TREE` problem shown in Figure 4. In this way, the steps are not unique to the problems either.



DOG_TIED_TO_TREE: SECTION FOUR, #4

Problem Statement

Cocoa is tied to a tree in the center of her backyard. The rope is 43 feet long. However, Cocoa keeps wandering around her tree, consequently winding her rope tighter and tighter around the tree. Evenutally, Cocoa is wound so close to the tree that she cannot walk around. Cocoa's owner, Chris, wants to build a fence so that he could do away with the rope altogether.

Help Chris figure out the area to be fenced in that would give Cocoa the same space to roam around in that the rope originally gave her.

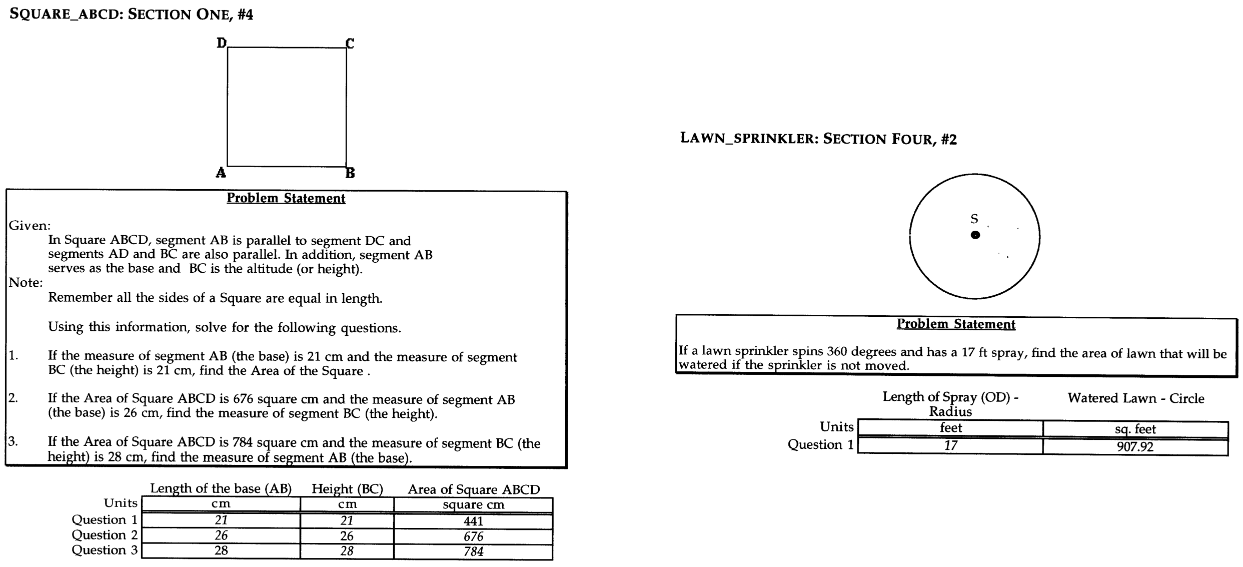
Once the area is determined, Chris needs to know how much chain-link fencing to buy. What is the length of fencing that he will need?

	Length of the rope (Radius)	Area dog has to roam (Area)	Length of fencing (Circumference)
Units	square feet	feet	feet
Question 1	*615.75*	14	28

Figure 4: The `DOG_TIED_TO_TREE` problem in the Geometry Area (1996-97) data set.

Although the steps are not unique to the problems, there are still some constraints to the possible problem-step combinations. Figure 5, for example, shows a square and a circle problem from the data set. It is visible that each shape has its own applicable steps. A student solving a square problem would never be asked to solve a circumference step. Vice versa, a square-area step would never occur for a circle problem. Clearly, some problem-step combinations are valid, while others are not.

In total, there are 139 valid problem-step combinations[1]. While all of these problem-step combinations are performed by at least one student, there is no such student that has answered all 139 problem-step combinations, nor is there a student that has answered all 40 problems. The data is therefore unbalanced. Additionally, the order of the problems is randomised for each student, which is a characteristic of the data set that may also cause problems in its subsequent analysis.



SQUARE_ABCD: SECTION ONE, #4

Problem Statement

Given:
In Square ABCD, segment AB is parallel to segment DC and segments AD and BC are also parallel. In addition, segment AB serves as the base and BC is the altitude (or height).

Note:
Remember all the sides of a Square are equal in length.

Using this information, solve for the following questions.

1. If the measure of segment AB (the base) is 21 cm and the measure of segment BC (the height) is 21 cm, find the Area of the Square .
2. If the Area of Square ABCD is 676 square cm and the measure of segment AB (the base) is 26 cm, find the measure of segment BC (the height).
3. If the Area of Square ABCD is 784 square cm and the measure of segment BC (the height) is 28 cm, find the measure of segment AB (the base).

	Length of the base (AB)	Height (BC)	Area of Square ABCD
Units	cm	cm	square cm
Question 1	*21*	*21*	441
Question 2	*26*	26	*676*
Question 3	28	*28*	*784*

LAWN_SPRINKLER: SECTION FOUR, #2

Problem Statement

If a lawn sprinkler spins 360 degrees and has a 17 ft spray, find the area of lawn that will be watered if the sprinkler is not moved.

	Length of Spray (OD) - Radius	Watered Lawn - Circle
Units	feet	sq. feet
Question 1	*17*	907.92

Figure 5: The SQUARE_ABCD and LAWN_SPRINKLER problems from the Geometry Area (1996-97) data set.

Regardless of the quality of the data, in order for it to be used as input for the knowledge graph algorithm (which will be described in more detail in the subsequent section), it first needs to be transformed into a usable format. The next section will describe how the original data set was transformed into a format that the knowledge graph algorithm knows how to process.

3.2 Pre-Processing

As a first step, the original data set must be pruned for relevant and irrelevant data. This pruning (and other subsequent transformations of the data) was done using R version 4.1.1 (R Core Team, 2021). The raw data set, as loaded into R, contains 6778 observations over 191 variables. Clearly, not all these variables will contain information that is relevant for the knowledge graph algorithm. For example, of the 191 variables, the last 161 correspond to previous analyses done by other people. Each of the columns represents its own analysis, and the observations in each column correspond to the knowledge components these analyses identified for each of the problem-steps. Since the methods behind these analyses cannot be verified, all 161 of these variables are discarded.

[1]The full problem set (i.e. all problem-step combinations) can be downloaded via https://pslcdatashop.web.cmu.edu/DatasetInfo?datasetId=76. Note that an account is needed to access this information.

Of the remaining 27 variables, 8 contain no observations and are filled with NAs. Five contain no unique values, with each observation corresponding to the same value. An example of such a variable is the *Level..Unit* variable. This variable always corresponds to *Area*, as this data set contains only the area unit of the Geometry Cognitive Tutor course. There are two variables that contain the same information, namely *Step.Name* and *Selection*. Of these, only one needs to remain in the data set. Finally, there are six variables that are deemed uninformative with an eye on the research question. These are administrative variables, such as the session IDs of the student's interaction with the cognitive tutor, as well as information pertaining to time (e.g. time stamps and response times). The latter variables are considered uninformative because, according to the description on DataShop, the values for these variables are simulated rather than measured. With all of the above-mentioned variables removed, the data set contains only 6 variables of interest. These variables are: *Anon.Student.Id*, *Step.Name*, *Problem.Name*, *Problem.View*, *Attempt.At.Step*, and *Outcome*.

For some of these variables, the name is sufficient to convey its meaning. The *Anon.Student.Id* variable, for example, is an anonymized identifier. It is a repeating variable, where each of the 50 students has one unique identifier tied to all their observations. The *Step.Name* and *Problem.Name* variables represent the step and problem respectively, as they are explained in the *Description of the Dataset* (i.e. *Problem.Name* represents one of the 40 problems, while *Step.Name* represents a step within one of the problem questions). The *Outcome* variable holds the actual performance of the students. Students can either have solved a problem-step correctly or incorrectly.

The two variables that require some additional explanation are the *Problem.View* and *Attempt.At.Step* variable. Since the problems in the data set were presented to students in a cognitive tutor, students would not only be presented with the same problem multiple times, they would also be given the opportunity to try and solve said problem multiple times. Given this characteristic of the cognitive tutor, the *Problem.View* variable keeps track of how often a student has been presented with a problem. The *Attempt.At.Step* variable, in turn, tracks how often a student has attempted to solve a problem. The latter variable is dependent on the former, in the way demonstrated in Table 1. Each time a student viewed a problem, they were allowed multiple attempts to solve it before moving on.

Problem Name	Step Name	Problem View	Attempt at Step	Outcome
`TRAPEZOID_AREA`	`(AREA QUESTION 1)`	1	1	INCORRECT
`TRAPEZOID_AREA`	`(AREA QUESTION 1)`	1	2	CORRECT
`TRAPEZOID_AREA`	`(AREA QUESTION 1)`	2	1	CORRECT
`TRAPEZOID_AREA`	`(AREA QUESTION 1)`	3	1	INCORRECT
`TRAPEZOID_AREA`	`(AREA QUESTION 1)`	3	2	CORRECT
`TRAPEZOID_AREA`	`(AREA QUESTION 1)`	4	1	CORRECT
`TRAPEZOID_AREA`	`(AREA QUESTION 1)`	5	1	CORRECT

Table 1: One student's attempts to solve the `TRAPEZOID_AREA` problem.

Next to the relationship between the *Problem.View* and *Attempt.At.Step* variables, Table 1 also clearly demonstrates the effects of these variables. Cognitive tutors are meant to help students improve their skills, and as such, students are evidently shown a problem until it is clear that they have learned the skill underlying the problem (as evidenced by multiple, subsequent correct attempts). For identifying the skills re-used between problems, this setup is less than ideal.

The idea behind the creation of the knowledge space (as explained in the *Theoretical Background*) is that the relationship between the problems can be identified by comparing the performance across them. If it is observed that failing one problem p often leads to failing another problem p', this indicates that these problems likely rely on the same (or a similar) set of skills. If students have learned all skills (and thus solve all problems correctly) by the end of their interaction with the cognitive tutor, this information is lost. As such, it is important that only the first *Problem.View* and *Attempt.At.Step* are considered for the identification of the skills underlying the problems. A subset `first.attempts` is created, which contains, for each student, only their first attempts at each problem-step.

Furthermore, to simplify the data set, the *Problem.Name* and *Step.Name* variables are combined into one variable called *Problem.Step* in the *first.attempts* subset. At the end of the pruning, the original data set of 191 variables has been reduced to a data set that contains only 3 variables: the *Problem.Step*, the *Anon.Student.ID*, and the *Outcome*.

These remaining 3 variables are transformed into a data frame that has the problem-steps as its columns and the students as its rows. Each square in the data frame contains the outcome of a student x for a problem-step y. The outcome is encoded into 1s and 0s, where a 1 indicates a student had a problem-step correct, while a 0 indicates the problem-step was incorrect. From this point forward, this data frame will be referred to as the *cleaned data*.

3.3 Subsets of the Data

Using the cleaned data directly for the knowledge graph algorithm proved too computationally expensive. As a solution, only a subset of the cleaned data was used. To be more precise, the knowledge graph algorithm was tested with a variety of subsets drawn from the cleaned data. These subsets were:

- **Dataset 1**: The 69 problem-steps that contained the most observations. This data set contains the information of 57 students, meaning 2 students are excluded from this data set for not having enough data.
- **Dataset 2**: The first questions only of each problem. These questions correspond to 66 of the original 139 problem-steps. No students are excluded from this data set, meaning it has observations for 59 students.
- **Dataset 3**: The first questions only of a subset of the problems. The subset of the problems was chosen so that there were no duplicate problems. That is to say, there are such problems in the data set that consist of the same or near identical problem-steps. Given their similarity, some learning could take place between these problems. As such, for each of these duplicate problems, only the problem that contained the most observations was saved. Some additional problematic problems were also removed, such as those that contained confusing/erroneous documentation. The final data set contained observations for 59 students over 46 problem-steps.
- **Dataset 4**: A subset of the problems only. This data set contains the same subset of problems as Dataset 3. However, in this data set, no distinction is made between the different steps. Looking only at question 1, a problem is considered correct if all of its steps are done correctly.

Otherwise, it is considered incorrect. This data set contains 20 (of the original 40) problems and has observations for all 59 students.

Regardless of which of the four data sets was used, at the start of the knowledge graph algorithm, an additional pruning is performed where all problems/problem-steps that contain less than 10 observations are removed. The data frame is transformed into a transposed matrix, so that the columns represent the students and the rows represent the problems. For simplicity's sake, 'problems' is used here (and from hereon) as an encompassing term for either the overarching problems or the problem-steps, depending on the data set used (see explanation above on the content of each data set). After transforming the chosen data set into a transposed matrix, the data can be used directly by the knowledge graph algorithm.

4 Knowledge Graph

The largest contribution of this thesis is the knowledge graph algorithm. It is a newer version of the algorithm used by Rozestraten (2021) (though it shares little overlap with hers). As with her algorithm, it was built in collaboration with N.A. Taatgen. The purpose of the algorithm is to build a knowledge space from performance data. Here, the term knowledge space is used according to the definition from Falmagne et al. (1990). The knowledge space shows sets of tasks that are solved together (i.e. when one task is solved correctly, so is another). In the knowledge graph, this is visualised by having each set in its own node. It is assumed that tasks within one node have a shared, underlying skill set, and that skill reuse occurs both between the tasks within one node and across nodes. Given this assumption, it is investigated whether knowledge graphs can be used to identify the reused skills.

If skills can be identified on the basis of the knowledge graph, cognitive models will be built to evaluate these skills. The models will hopefully show how the identified skills integrate with each other to correctly solve a variety of tasks, thereby answering the research question. The cognitive models will be discussed in further detail in Section 5. In the current section, it is explained how the knowledge graph itself was generated.

The explanation is broken down into four parts. First, the original knowledge graph algorithm is explained as it was used to generate the knowledge graph. After some intermediate results, it will become clear that the original knowledge graph algorithm is insufficient for identifying the skills reused between tasks. Thus, on the basis of these intermediate results, an alteration is made to the original algorithm. New results are generated with the altered algorithm, and at the end, the final knowledge graph is shown.

4.1 Original Knowledge Graph Algorithm

4.1.1 The Logic Behind the Algorithm

A knowledge graph consists of nodes and edges. In this thesis, the idea is to populate the nodes of the knowledge graph with the problems from the geometry data set, so that the skill hierarchy between the problems becomes clear. To be more concrete, the final knowledge graph should encapsulate all the problems within its nodes, while adhering to the following constraints:

1. Problems that rely on the same skills should be put in the same node.
2. If problem P_x relies on a superset of the skills of problems P_y, then problem P_x should be lower in the knowledge graph than problem P_y.
3. Vice versa, if problem P_y relies on a subset of the skills of problem P_x, then problem P_y should be higher in the knowledge graph.
4. Finally, if two problems rely on the same number of skills, but the skills relied on are different, these problems should be on the same level in the knowledge graph but inside different nodes.

From these written rules, it is not necessarily clear what such a knowledge graph may look like. For this reason, an example is given. Table 2 shows some hypothetical outcomes for four students across four problems. Here, a 1 indicates a problem was performed correctly, while a 0 indicates that a problem was solved incorrectly. In terms of skills, it is assumed that a student who solves a problem correctly possesses all the skills that underlie said problem.

Given this assumption, Table 2 shows that the first three students possess the skills to solve both the `RECTANGLE_ABCD` problem and the `SQUARE_ABCD` problem. The last student can solve neither of the problems, and therefore possesses none of the skills they require. This observation can be interpreted in two ways. The first interpretation assumes that `RECTANGLE_ABCD` and `SQUARE_ABCD` rely on different skills. In this case, it is simply a coincidence that students 1 to 3 possess both the skills for the `RECTANGLE_ABCD` problem and those for the `SQUARE_ABCD` problem, while student 4 possesses none of the required skills. With a data set consisting of four students, this interpretation seems quite plausible.

PROBLEM NAME	**S1**	**S2**	**S3**	**S4**
RECTANGLE_ABCD	1	1	1	0
SQUARE_ABCD	1	1	1	0
ONE_CIRCLE_IN_SQUARE	0	1	0	1
TWO_CIRCLES_IN_SQUARE	1	1	0	0

Table 2: Example outcomes for four students across four problems in the data.

However, what if Table 2 held the outcomes of 100 students, and across all these students, the same pattern is observed? If across 100 students, a student can either solve both problems correctly or solve neither of them, then suddenly, chance feels like a less likely explanation. A more likely hypothesis, in such a case, is that the `RECTANGLE_ABCD` and `SQUARE_ABCD` problems rely on the same skill set. From that interpretation, it would be easy to explain why students can either do both problems or can do neither. In the former situation, the student possesses the skills that are reused across the two problems, while in the latter, they do not. Assuming the second hypothesis holds true, then the `RECTANGLE_ABCD` and `SQUARE_ABCD` problems should be put in the same node in the knowledge graph, as per the constraints given prior.

Looking at the two remaining problems, `ONE_CIRCLE_IN_SQUARE` and `TWO_CIRCLES_IN_SQUARE`, Table 2 shows that neither of these problems follow the same pattern as observed for the `RECTANGLE_ABCD` and `SQUARE_ABCD` problems. From this observation, it follows that the circle-in-square problems must rely on different skills than the rectangle and square problems. If they did not, the performance across these two sets of problems should have been the same.

Furthermore, it is visible that the circle-in-square problems have a lower performance compared to the `RECTANGLE_ABCD` and `SQUARE_ABCD` problems. This indicates that the circle-in-square problems have a higher difficulty. Worded differently, a student that has the skills to solve the `RECTANGLE_ABCD` and `SQUARE_ABCD` problems does not have the skills to solve the circle-in-square problems. The circle-in-square problems are thus thought to require a superset of the rectangle and square skills. Given this property, the constraints given prior state that both circle-in-square problems should be placed lower in the knowledge graph than the rectangle and square problems.

When comparing the `ONE_CIRCLE_IN_SQUARE` and `TWO_CIRCLES_IN_SQUARE` problems to each other, there is no one problem that outperforms the other. Although the problems both demonstrate a 50% accuracy, the set of students that got each problem correct differs. From this, it can be concluded that these two problems must rely on different skills. If they relied on the same skills, then students who had one problem correct should also have had the other problem correct (and vice versa with incorrectness). From this, it can be concluded that the `ONE_CIRCLE_IN_SQUARE` and `TWO_CIRCLE_IN_SQUARE` problems should be placed in different nodes of the knowledge graph, but that these different nodes should still be on the same level of the graph.

Figure 6 shows the knowledge graph that Table 2 would result in. The graph consists of three nodes and is two levels in its depth. The first node contains the `RECTANGLE_ABCD` and `SQUARE_ABCD` problems. The second and third node contain the circle-in-square problems. It is visible that the size of the node reflects the number of problems inside said node. This visual hint is intentional, and it will be upheld by the algorithm as a guideline for the design of the final knowledge graph. This will be discussed further in Section 4.1.4.

What is furthermore visible is that the top node is connected to the lower nodes by edges. The edges in the knowledge graph indicate a subset/superset relationship between the nodes. The lower nodes in the graph are thought to be supersets of the higher nodes they are connected to. In this case, the knowledge graph indicates that the circle-in-square nodes are supersets of the rectangle and square node.

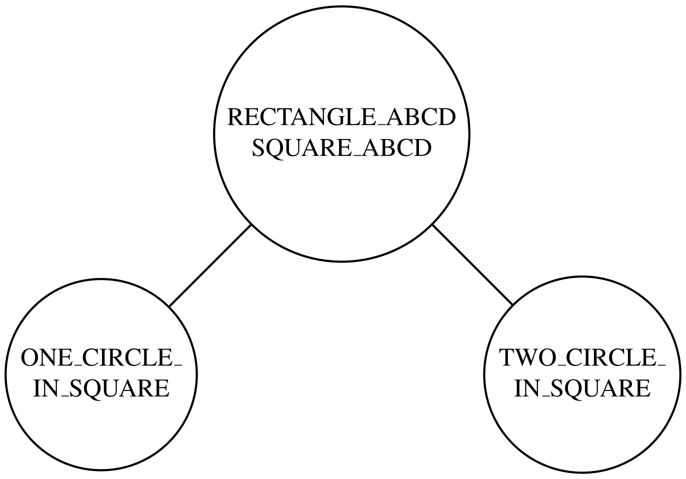


Figure 6: The knowledge graph that can be derived from Table 2.

Overall, the transformation from Table 2 to Figure 6 roughly shows the task the knowledge graph algorithm has to accomplish. In the subsequent subsections, it is explained in more detail how exactly the knowledge graph algorithm goes about accomplishing this task.

4.1.2 Its Core Functions

The example shown in the previous section gives some idea of the task the knowledge graph needs to accomplish. Of course, the actual process of generating the knowledge graph from the performance data is not as deterministic as implied above.

For one thing, the real-life performance data is much more noisy than the hypothetical data shown in Table 2. Students can answer questions incorrectly, even when they do theoretically possess the skills required to solve a problem. This can happen, for example, when a question is interpreted incorrectly or a given number is not copied correctly. In general, the patterns in the real-life data will be a lot less clear-cut and easy-to-interpret as in Table 2. As such, the knowledge graph algorithm will have to find a way to find the relationships between the problems, even when there is some noise involved in the data.

Additionally, it is important to point out that Figure 6 assumes, based on the data, that there are three skills underlying the four problems. It is possible this is not a correct assumption, and it is therefore vital that the number of skills underlying a data set are determined prior to generating the full knowledge graph. The knowledge graph algorithm determines the number of skills underlying a data set through an empirical process. To understand this process, it is necessary to first understand the core functions of the knowledge graph algorithm. These functions are explained in this subsection.

As in the example shown, the knowledge graph algorithm begins by comparing the problems from the chosen data set. It compares the problems on three dimensions: *sameness*, *moreness*, and *lessness*. Each problem is compared with each other problem, which results in three comparison matrices (one for each dimension).

S1	**S2**	**S3**	**S4**
0	1	0	1
1	1	0	*NA*

↓ results in

0	1	1	*NA*

(a) **`same.as`**

S1	**S2**	**S3**	**S4**
0	1	*NA*	1
1	1	0	0

↓ results in

0	0	*NA*	1

(b) **`more.than`**

S1	**S2**	**S3**	**S4**
0	*NA*	0	1
1	*NA*	0	0

↓ results in

1	*NA*	0	0

(c) **`less.than`**

Table 3: The results of comparing problems on the dimensions of sameness, moreness, and lessness (in that order).

The comparison matrices are generated through three functions named `same.as`, `more.than`, and `less.than`. Each function corresponds to one of the three dimensions mentioned prior and is used to compare a row_i with a row_j on said dimension. This comparison, when applied to two rows, results in a vector of 1s and 0s of length C (where C is the number of columns, or students, in the data set). Each entry in this vector corresponds to the result of one specific student. A 1 in the vector means that the relevant function returned as "true" for that student for the two rows compared. A 0 means the function returned as "false" for said student.

What this indication of true or false means depends on the function. The `same.as` function returns true when the two compared performances are the same, meaning they are both 1s or both 0s. The `more.than` function returns true when the performance from row_i is more than the performance from row_j. In other words: It returns true when row_i is correct and row_j is incorrect. The `less.than` function does the opposite of the `more.than` function. It returns true when row_i is incorrect, but row_j is correct. All functions return as *NA* (not applicable) if either row_i or row_j has a missing value (which means a student did not do the corresponding problem).

Table 3 shows a visual summation of these rules by comparing two rows of hypothetical outcomes. By comparing all rows in a data set, the N-by-N comparison matrices are created. Here, N equals the number of rows, or problems, in the data set. An example of such a comparison matrix is shown in Table 4.

Table 4 specifically shows the sameness matrix (identifiable by a problem always being 100% the same as itself). However, the moreness and lessness matrix use the same schema and thus look similar to Table 4. Notice that a square in a comparison matrix indicates the proportion of students for which the relevant function (i.e. `same.as`, `more.than`, or

	P_1	P_2	$P_{...}$	P_N
P_1	1	0	...	...
P_2	0	1	...	...
$P_{...}$	...	...	...	...
P_N	0.25	0.64	...	1

Table 4: The schema of the sameness matrix filled with some arbitrary numbers.

`less.than`) returned true. This proportion is given
on a scale of 0 to 1, with a 0 indicating the function was not true for any of the students and a 1 indicating the function was true for all of the students.

The comparison matrices are not used directly by the algorithm. Rather, they are used to create the *penalty matrices*. The penalty matrices serve to determine the *fit* of the knowledge graph generated by the knowledge graph algorithm. That is to say, they evaluate how well the final knowledge graph fits the data. There are four penalty matrices, each bound to a specific condition:

- **p.S**: this penalty matrix is applied when two problems P_x and P_y are put in the same node. It is equal to the sum of the moreness and lessness matrices (i.e. all the cases where P_x and P_y do not have the same performance).
- **p.L**: this penalty matrix is applied when a problem P_x is thought to rely on a superset of the skills of a problem P_y (and thus be a more difficult problem). It is equal to the moreness matrix (i.e. all the cases P_x was performed better than P_y).
- **p.M**: this penalty matrix is applied when a problem P_x is thought to rely on a subset of the skills of a problem P_y (and thus be an easier problem). It is equal to the lessness matrix (i.e. all the cases P_x was performed worse than P_y).
- **p.D**: this penalty matrix is applied when two problems are considered to belong to different nodes of the same level. It is equal to the sameness matrix (i.e. all the cases where P_x and P_y have the same performance).

If these conditions sound familiar, it is because they directly incorporate the constraints given in *The Logic Behind the Algorithm*. In fact, the knowledge graph algorithm can be summarised as such: The algorithm tries to adhere to the constraints given in *The Logic Behind the Algorithm*. Each time it puts a problem P_z in one of its nodes, it evaluates how well it is matching these constraints through a calculation of its fit. This fit is determined by penalties that, at their core, reflect the proportion of cases where the algorithm is wrong about its chosen node. In simple terms, the penalty matrices say: according to this proportion, P_z should not be in that node at all. Of course, this description is a simplification of the algorithm's full workings. Listing 2 shows the function through which the fit of the algorithm is actually determined.

Listing 2 shows that, rather than calculating the fit of one problem only, the `calc.fit` function always calculates the total fit. The total fit is calculated by comparing each problem to each other problem and then summing up each of the individual penalties. The problem comparison is done through the use of for-loops in line 4 and 5 of the listing.

The `nodes` vector used in these for-loops contains the nodes the algorithm has placed each problem in. The position (or index) of the vector identifies the problem, while the actual value gives its node. For example, `nodes[3] = 5` says that the third problem in the data set is put in node 5 of the knowledge graph.

Each problem comparison begins by confirming that i and j are not the same value. This ensures that a problem is not compared with itself. If the two problems that are being compared are unique, the algorithm considers the four conditions given prior through if-statements. The first condition is easy

Listing 2: The `calc.fit` function.

```
1  calc.fit <- function(nodes, p.S, p.L, p.M, p.D) {
2    nodes <- as.integer(nodes)
3    fit = 0
4    for (i in 1:(length(nodes))) {
5      for (j in (1:length(nodes))) {
6        if (i != j) {
7          if (nodes[i] == nodes[j]) {
8            fit = fit + p.S[i,j]
9          } else if (bitwOr(nodes[i], bitwNot(nodes[j])) == -1) {
10           fit = fit + p.L[i,j] * hamming.distance(intToBits(nodes[i]),
                 intToBits(nodes[j])
11         } else if (bitwOr(nodes[j], bitwNot(nodes[i])) == -1) {
12           fit = fit + p.M[i,j] * hamming.distance(intToBits(nodes[i]),
                 intToBits(nodes[j]))
13         } else {
14           fit = fit + p.D[i,j]
15         }}
16     }
17   }
18   return(fit)
19 }
```

to check through a logical equals comparison. The comparison checks if two values, in this case two nodes, are the same. Similarly, the last condition, if two nodes are different, is also easy to check. This condition is considered to be true if none of the other conditions were (line 13 and 14 of the listing).

The second and third condition (lines 9 to 13) are more difficult to check. They try to determine whether one node relies on the superset (or subset) of another node's skills. To do this, they rely on a fundamental property of the knowledge graph: Each node has a unique numerical value, and that decimal value has a unique binary counterpart. The binary representation of the node 5 given earlier, for example, would be `0101` (using 4 bits) [2]. In this thesis, each bit of this binary representation is interpreted to correspond to one skill in the data set. According to this interpretation, node 5 thus represents two skills: one at position 0 and one at position 2 of the binary representation.

This property makes it possible to determine whether one node relies on the superset (or subset) of another node's skills by comparing the binary representations. If the binary representation of node *j* contains at least 1 more 1-bit than that of node *i*, then node *j* relies on a superset of node *i*'s skills (and vice versa). The expression `bitwOr(nodes[i], bitwNot(nodes[j])) == -1` in line 9 and 11 of Listing 2 is used to do this check. For these two conditions, a multiplier is applied to the penalty matrix that is equal to the number of 1-bits that node *i* and *j* differ in. Thus, if node *j* is thought to rely on a much bigger skill set than node *i* (as in line 11), the penalty for all the cases where this is false is heavier.

[2] See Dube (2022) for a refresher on reading binary.

Given that the total fit is determined by the penalties and their weights, a lower fit score is, in the case of the `calc.fit` function, considered to represent a better fit to the data. A fit of 0 would (in theory) mean that no penalties were applied, and that the generated knowledge graph thus perfectly fits the data. To generate a good knowledge graph, the `calc.fit` function and the penalty matrices are therefore crucial aspects of the knowledge graph algorithm.

4.1.3 The Main Workings of the Algorithm

The knowledge graph algorithm generates its knowledge graph through the `improve.fit` function shown in Listing 3.

Listing 3: The `improve.fit` function.

```r
improve.fit <- function(nodes, oldfit, p.S, p.L, p.M, p.D, n.skills, n.
    cycles = 500) {
  for (i in 1:n.cycles) { # run for n.cycles or until break
    bestfit = oldfit
    bestnode = -1
    bestproblem = -1

    # go through each problem in the nodes vector
    for (problem in 1:length(nodes)) {
      oldnode = nodes[problem]

      # try to put that problem in each possible node
      for (node in 0:(2^n.skills-1)) {
        nodes[problem] = node
        newfit = calc.fit(nodes, p.S, p.L, p.M, p.D)

        # move the problem to the new node if it improves the fit
        if (newfit < bestfit) {
          bestfit = newfit
          bestnode = node
          bestproblem = problem
        }
      }
      nodes[problem] = oldnode # reset problem to its original node
    }

    # move the problem to the node that resulted in the best fit
    if (bestnode != -1) {
      nodes[bestproblem] = bestnode
      oldfit = bestfit
    } else {
      break # break if local optimum is found
    }
  }
  return(nodes)
}
```

Although the 35 lines of code may give the opposite impression, the `improve.fit` function is actually quite simple in its functionality in that it resembles a Hill-Climbing algorithm (Russel & Norvig, 2003). In principle, it consists of a loop in which the function continually looks for the move that results in the best improvement to the knowledge graph fit. Once it finds that optimal move, it breaks out of its loop and executes it. It then returns the distribution of problems into nodes that resulted in the best fit according to the best move.

To describe the function in more detail, its input variables are explained first. First and foremost, the `improve.fit` function requires a `nodes` vector, which contains the initial division of problems into nodes. This is the same `nodes` vector required by the `calc.fit` function. Before running the `improve.fit` function, the knowledge algorithm sets this vector to zero, meaning it places all the problems into a node 0. Most likely, this initial vector will not result in a great fit. Its fit is calculated through `calc.fit` and stored in the $oldfit$ variable. The `improve.fit` function will, as its name implies, try to improve upon this $oldfit$ variable.

To improve upon the $oldfit$, the `improve.fit` function moves the problems around to alternative nodes. After moving one problem to one new node, the function will re-calculate the fit through the `calc.fit` function. For this, the `improve.fit` function requires the penalty matrices *p.S*, *p.L*, *p.M*, and *p.D* as input as well.

The two final variables the `improve.fit` function needs to know are the number of skills in the data set (*n.skills*) and the number of cycles it should maximally run (*n.cycles*). The `improve.fit` function will move problems around until it reaches a local optimum, meaning there is no problem left it can move to a new position to get a better fit, or until it reaches the *n.cyles* variable. This variable is set to 500 by default.

With its input variables clear, the `improve.fit` function goes through all the problems in the `nodes` vector and finds, for each problem, the node that results in the best fit (lines 8 to 24 of Listing 3). The nodes it can choose from are given in line 12. The range goes from 0 to maximally 2^*n.skills*-1. Since each node has a corresponding binary representation, the maximum value given here corresponds to the highest possible binary value (i.e., all bits are 1) given a certain number of skills. To give an example, if a data set consists of 5 skills in total, then the highest binary value would be `1111`, which corresponds to 31 in the decimal system. Within this range, there are 32 possible nodes to place a problem in (including 0), and each node corresponds to its own unique combination of 1s and 0s in the 5-bit binary string.

Clearly, to use this range, the total number of skills in the data set must be known prior to running the `improve.fit` function. This is precisely what the algorithm does: It determines the total number of skills empirically before it runs the `improve.fit` function. A range is checked of 1 to maximally 32 skills (capped at 32 for computational reasons). For each *n.skills* value, the `improve.fit` function is run to get the best fit for that number of skills. Once the fit plateaus (meaning it no longer improves even if the numbers of skills are increased), that number of skills is chosen as the correct number of skills for the chosen data set. Figure 7 shows this empirical determination of the correct number of skills for Dataset 1.

Given the full range of nodes available, in each cycle of the `improve.fit` function, the best node is determined for each of the N problems. However, not every problem is moved to its best node. Only

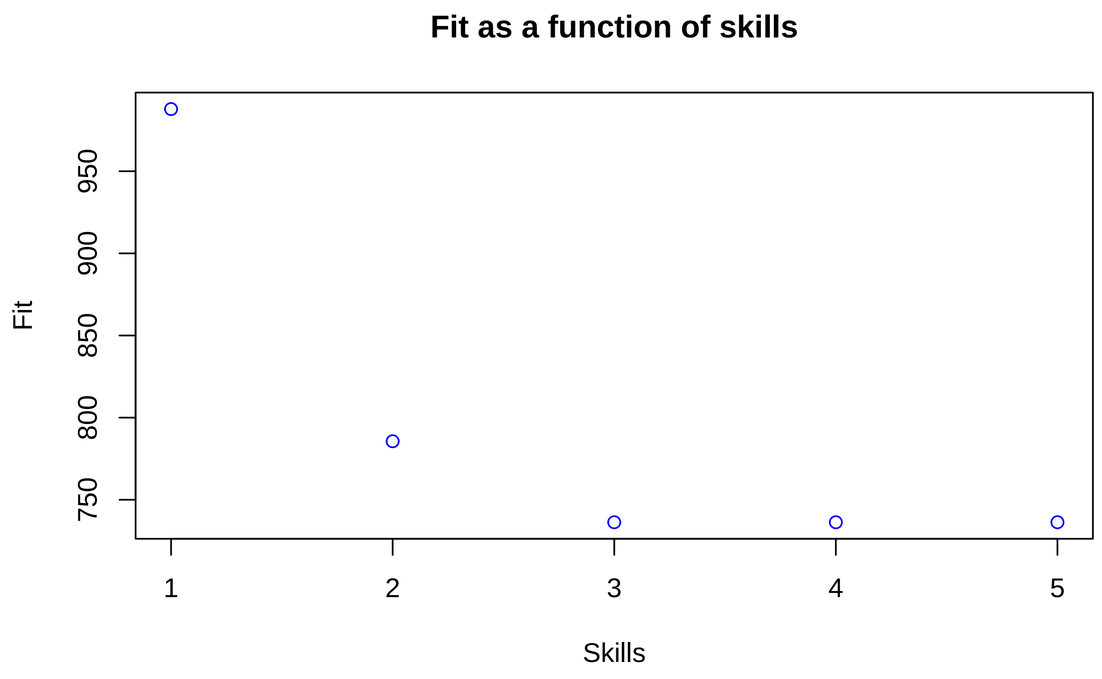


Figure 7: The fit as a function of the number of skills for Dataset 1. This data set is thought to have 3 underlying skills.

the problem that resulted in the highest fit is moved to its new best node. Then, a new cycle starts, and the `improve.fit` function goes through all of the problems again to find the next best problem to move. As was said prior, this goes on until the 500th cycle is reached, or until there is no move that will result in an improvement in the fit.

By improving the fit in this way, the algorithm generates a final knowledge graph that best fits the data according to a local optimum. Since the algorithm never makes a move that results in a higher fit, it is possible it misses a global optimum. Sometimes making a worse move in one cycle can result in an overall better fit in the long run. To try and find a global optimum, *simulated annealing* is used as per the `GenSA` function from version 1.1.7 of the equivalent GenSA package (Xiang et al., 2013) in R. The function is called as:

```
GenSA(nodes, p.S, p.L, p.M, p.D, fn = calc.fit, lower=minval,upper = maxval,
control=list(max.time=600*n.skills,verbose=FALSE))
```

In this function call, *minval* is the same as the initial `nodes` vector. Namely, a vector of size N that consists of all zeroes. The *maxval* variable holds the opposite vector, which puts each problem in the maximum node possible (where the maximum node possible is again given by 2ˆ*n.skills*-1). The `calc.fit` function is passed to the `GenSA` function to optimise, and the function is given 600**n.skills* seconds to find the optimum (i.e. 10 minutes per skill).

If given enough time, the generalised simulated annealing will find a global optimum through the use of an artificially induced temperature variable T. This temperature variable controls the probability that the simulated annealing algorithm will allow for a “bad move” (i.e., a move that increases the fit). The temperature starts off high, but becomes lower over time. Thus, the probability of accepting a bad move decreases over time. In practice, this means that, once the algorithm is close to finding a global optimum, it is unlikely to allow for a bad move. While it still has a lot of space to find the optimum, however, it is much more likely to allow for a bad move.

Through empirical testing, it was found that, if the *max.time* variable was set high enough, the `GenSA` function always resulted in the same or a better fit than the `improve.fit` function. Nonetheless, it cannot be guaranteed that the fit returned by the `GenSA` function is the global optimum, since it is not known in advance what value of *max.time* is "enough" to guarantee the global optimum has been found. The comparison between the `improve.fit` and `GenSA` results do imply that a *max.time* of 10 minutes per skill is an appropriate rule of thumb, *if the value of* n.skills *is set.*

When the value of *n.skills* is variable, such as when the optimal number of skills is being determined, then having the *max.time* variable depend on the number of skills is not appropriate. Since in that case, the differences between the fits in each cycle could be attributed to the differences in the *max.time*, rather than the difference in *n.skills*.

Rather, the *max.time* variable should be kept constant when the number of skills is being determined. Additionally, the value chosen for *max.time* should be large enough so that it is enough time for the algorithm to find (or approach) the global optimum both for the smallest number of skills, as for the largest number of skills. If the 10 minutes per skill rule of thumb is adhered to, that means the *max.time* should be set to a constant value of 320 minutes (with 32 being the maximum number of skills that is tested for) when determining the optimal number of skills. This approximates to about 5 hours. Clearly, running the `GenSA` algorithm for 5 hours for each value of *n.skills* is inefficient.

Consequently, the decision was made to use the `improve.fit` function for determining the optimal number of skills. After the optimal number of skills have been determined, two node distributions are generated. One by the `improve.fit` algorithm, and one by the `GenSA` algorithm. The node distribution that resulted in the lowest fit is chosen as the best node distribution.

4.1.4 Drawing the Graph

Once the best node distribution has been established, either by the generalised simulated annealing or by the `improve.fit` function, the distribution is used to draw a graph. The graph is drawn through version 1.2.6 of the igraph package (Csardi & Nepusz, 2006).

In order to draw a graph, the final `nodes` vector is first transformed to a table. This table gives all the unique nodes used (not every node in the full range will necessarily have been used) and the number of problems in each node. The length of this table L is used to initially set up a graph with L nodes and no edges.

After, the algorithm loops through all the nodes and compares them with each other to determine if edges need to be drawn to connect certain nodes. This comparison works the same as the second and third condition of the `calc.fit` function. That is to say, if $node_i$ has more 1-bits than $node_j$ in its binary representation, then an edge is drawn between those nodes. An average fit is calculated which compares all the problems in $node_i$ with those in $node_j$, and this average fit is used as a label for the edge.

After the edges are determined, the graph is plotted with its determined nodes and edges. As was also mentioned in *The Logic Behind the Algorithm*, the nodes are drawn in size to be proportional to the number of problems in said node.

The graph at this step is not yet the final graph, as there is still a pruning step that is performed. The pruning step removes all superfluous edges. To understand this pruning step, it is important to realise that the nodes in the knowledge graph are transitive. Accordingly, if $node_i$ is a subset of the skills used by $node_j$, and $node_j$ is, again, a subset of the skills used by $node_k$, then $node_i$ is, through its transitive nature, also a subset of the skills in $node_k$. The graph that is drawn initially will have explicit edges between $node_i$ and $node_j$, $node_j$ and $node_k$ *and* $node_i$ and $node_k$. The edges like the one between $node_i$ and $node_k$ are superfluous, since the connection between $node_i$ and $node_j$ and $node_j$ and $node_k$ already establishes that $node_i$ is a subset of $node_k$. After the previous step, there are quite a few of these superfluous edges left, and it is these edges that are removed in the pruning step to end with a clean, clear final graph. No example is given here, as the next section will show the final knowledge graphs generated by the knowledge graph algorithm for the various data sets.

4.2 Intermediate Results

4.2.1 Dataset 1

Dataset 1 consisted of the 69 problem-steps from the 'Geometry Area (1996-97)' data set that contained the most observations. According to the skill optimisation procedure (shown in Figure 7), this data set contained three underlying skills. Figure 8 shows the accompanying knowledge graph.

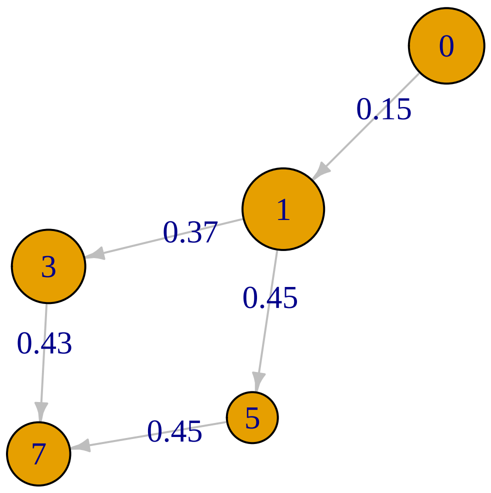


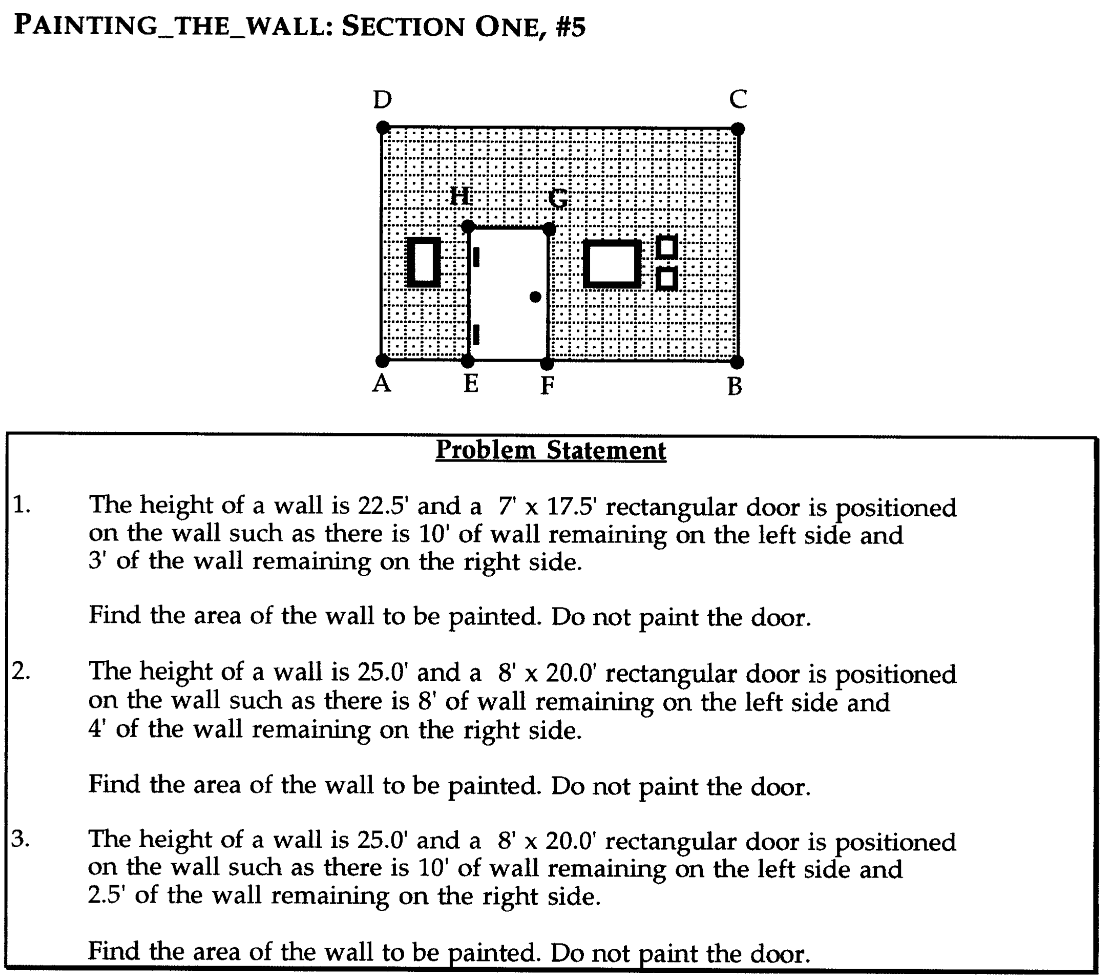
Figure 8: The knowledge graph generated from Dataset 1.

The knowledge graph in Figure 8 consists of 5 nodes. Node 0 (which corresponds to the bit-string of `000`) contains all the problems that students could solve by relying on their prior knowledge only. After Node 0, each node at depth $d+1$ in the knowledge graph requires one skill more than the node at depth d. Thus, Node 1 contains the problems that require one skill, Node 3 and 5 those that require two, and Node 7 contains the problems that require all three of the skills that underlie this problem set.

Although the average fit shown on the labelled edges of the knowledge graph gives some indication of how well this knowledge graph fits the data (lower is, in theory, better), to fully understand what the knowledge graph is showing, the problems within each of the nodes must be analysed.

The difficulty in analysing the problems within each node is that there is no 'answer-sheet' for the correct distribution. There is no ground truth to compare the knowledge graph against, because the skills that underlie the problem set are not known in advance. It is, however, possible to assess the knowledge graph qualitatively on the basis of its intuitive correctness.

Intuition would state that, if the knowledge graph is correct in its distribution of problems across its nodes, then problems that are fundamentally the same should be in the same node. A problem is considered the same as another problem if it consists of the same steps within the same shape.


PAINTING_THE_WALL: SECTION ONE, #5

Problem Statement

1. The height of a wall is 22.5' and a 7' x 17.5' rectangular door is positioned on the wall such as there is 10' of wall remaining on the left side and 3' of the wall remaining on the right side.

   Find the area of the wall to be painted. Do not paint the door.

2. The height of a wall is 25.0' and a 8' x 20.0' rectangular door is positioned on the wall such as there is 8' of wall remaining on the left side and 4' of the wall remaining on the right side.

   Find the area of the wall to be painted. Do not paint the door.

3. The height of a wall is 25.0' and a 8' x 20.0' rectangular door is positioned on the wall such as there is 10' of wall remaining on the left side and 2.5' of the wall remaining on the right side.

   Find the area of the wall to be painted. Do not paint the door.

Figure 9: The PAINTING_THE_WALL problem in Dataset 1.

Figure 28 shows an example of such identical problems. Each question of the PAINTING_THE_WALL problem is considered its own problem in Dataset 1. Clearly, each of these questions is the same, save from different values used for the various variables. The questions consist of the same steps, which are shown in Table 6. The table also shows which node each step was placed in according to the colour scheme shown in Table 5.

Nodes	0	1	3	5	7

Table 5: Legend for the colour coding of each node in Figure 8.

Table 6 shows that most of the steps from the PAINTING_THE_WALL problem are placed in Node 0. This would match with our intuition, which expects each identical step to be placed within the same node. There are two steps, however, that do not match this intuition. Calculating the area of the wall is placed in Node 1 for Question 1, and calculating the shaded region (the area to be painted) is placed in Node 3. If the knowledge graph was accurate in its depiction of the skills underlying the various steps, then these two steps should have been placed in Node 0 as well, *assuming* that the steps do not rely on any of the skills that underlie this data set.

	Area of Wall ABCD	Area of Door EFGH	Area of shaded region
Units	sq. feet	sq. feet	sq.feet
Question 1	450	122.5	327.5
Question 2	500	160	340
Question 3	512.5	160	352.5

Table 6: The steps of the three questions of PAINTING_THE_WALL, plus their correct answers.

If this step distribution is further analysed, it actually seems, intuitively, that the distribution of steps in Question 1 makes more sense than the distribution of the steps in Question 2 and 3. Calculating the area of the door is arguably the easiest step in the PAINTING_THE_WALL problem, since all the information necessary to complete this step is given. Calculating the area of the wall, by contrast, requires a student to first calculate the base of the wall. It makes sense that this additional step would make calculating the area of the wall a more complicated cognitive undertaking than calculating the area of the door. The same holds for calculating the area of the shaded region, a step that requires, at minimum, that the other two steps are completed first.

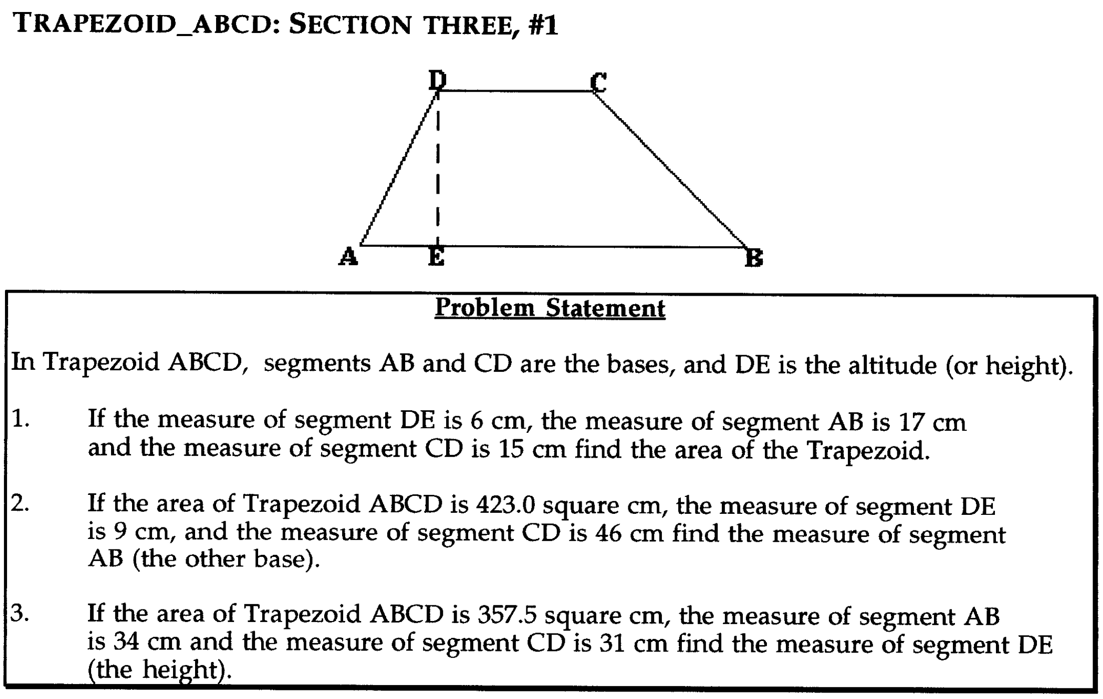
As with the idea that identical steps should belong in the same node, intuition also states that more complicated steps (i.e. steps that consist of multiple sub-steps) should be placed in lower levels of the knowledge graph. Thus, the distribution of steps in Question 1 matches well with what feels, intuitively, correct, but the distributions of Question 2 and 3 do not.

That is, *unless* there is some element of learning that takes place here. The knowledge graph is generated based entirely on student performance expressed by their accuracy. It makes sense that, after correctly performing Question 1, students know how to perform its steps by Question 2 and 3. This especially holds because of the cognitive tutor's behaviour, since it will show students the correct answer to the steps they had wrong, until they get these steps correct. Clearly, there is a strong learning effect at play here, which causes the knowledge graph to change how it classifies certain steps over time.

Of course, making this conclusion on the basis of only one problem is rather premature. In **Appendix A**, another example is given that is similar to the PAINTING_THE_WALL problem, where the same step is classified into a different node for each subsequent question.

Next to identical steps within the same problem, the learning effect is also seen for identical steps across problems. Figure 10 shows the TRAPEZOID_ABCD problem, which asks students to calculate the area, the other base, and the height of a trapezoid in question 1, 2, and 3 respectively.

In addition to this TRAPEZOID_ABCD problem, Dataset 1 contains the problems TRAPEZOID_AREA, TRAPEZOID_BASE, and TRAPEZOID_HEIGHT (shown in Figure 11). Clearly, the latter three problems correspond directly to question 1, 2, and 3 of the TRAPEZOID_ABCD problem, though, as with the three questions in PAINTING_THE_WALL, they use different values for their variables.



TRAPEZOID_ABCD: SECTION THREE, #1

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the measure of segment DE is 6 cm, the measure of segment AB is 17 cm and the measure of segment CD is 15 cm find the area of the Trapezoid.

2. If the area of Trapezoid ABCD is 423.0 square cm, the measure of segment DE is 9 cm, and the measure of segment CD is 46 cm find the measure of segment AB (the other base).

3. If the area of Trapezoid ABCD is 357.5 square cm, the measure of segment AB is 34 cm and the measure of segment CD is 31 cm find the measure of segment DE (the height).

Figure 10: The TRAPEZOID_ABCD problem, which consists of three questions.

Yet despite of the overlap between the question of TRAPEZOID_ABCD and the problems in Figure 11, each problem is classified differently in terms of its node distribution. Table 7 shows the node distribution of each of the trapezoid problems. For TRAPEZOID_ABCD, there are three questions that each correspond to one of three possible steps: trapezoid area, trapezoid longer base, and trapezoid height. The remaining three problems all require students to perform only one of those three steps.

It is visible that for TRAPEZOID_ABCD, each step is placed in Node 7. For the TRAPEZOID_AREA problem, however, the trapezoid area step is placed in Node 1 instead. Similarly, the trapezoid longer base step is placed in Node 3 for the TRAPEZOID_BASE problem, while it is placed in Node 7 for TRAPEZOID_ABCD. The only step that is classified consistently is the trapezoid height step.

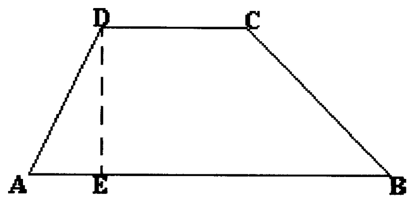
With the different classifications, it is difficult to say which nodes the three steps most accurately belong in (where accuracy is defined by the knowledge graph's fit to the data). If it is assumed that students typically see TRAPEZOID_ABCD before they see the other three problems, however, then these

	Trapezoid ABCD	Trapezoid Area	Trapezoid Base	Trapezoid Height
Question 1	$96cm^2$	$224cm^2$		
Question 2	$7cm$		$58cm$	
Question 3	$11cm$			$16cm$

Table 7: The node distribution of the steps for each trapezoid problem. Each step is represented by one of the questions of TRAPEZOID_ABCD. Steps that are not applicable for a problem are greyed out.

results can, once again, be explained by a learning effect. If TRAPEZOID_AREA and TRAPEZOID_BASE are shown to students after the TRAPEZOID_ABCD problem, then students will already know how to solve these problems, and they will receive higher accuracies on them.

TRAPEZOID_AREA: SECTION THREE, REMEDIAL

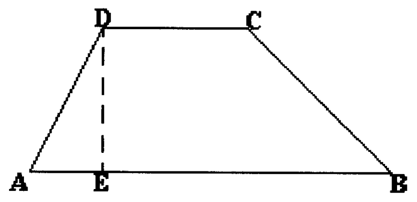


Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the measure of segment DE is 7 cm, the measure of segment AB is 33 cm and the measure of segment CD is 31 cm find the area of the Trapezoid.

TRAPEZOID_BASE: SECTION THREE, REMEDIAL

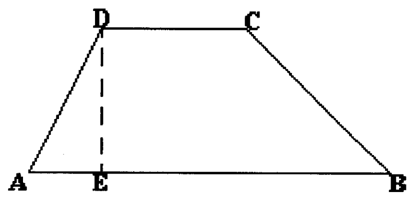


Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the area of Trapezoid ABCD is 1425.0 square cm, the measure of segment DE is 25 cm, and the measure of segment CD is 56 cm find the measure of segment AB (the other base).

TRAPEZOID_HEIGHT: SECTION THREE, REMEDIAL



Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the area of Trapezoid ABCD is 472.0 square cm, the measure of segment AB is 31 cm and the measure of segment CD is 28 cm find the measure of segment DE (the height).

Figure 11: Three Trapezoid problems that each represent one question of the TRAPEZOID_ABCD problem.

Of course, this analysis only holds if TRAPEZOID_ABCD was shown to students prior to the other trapezoid problems. The fact that the trapezoid height step was classified consistently into Node 7 counteracts this assumption. If TRAPEZOID_ABCD was shown to students before TRAPEZOID_HEIGHT, then the trapezoid height step should have been placed in a higher node in the knowledge graph for the TRAPEZOID_HEIGHT problem, as also occurs for the trapezoid area and trapezoid longer base steps.

In the *Description of the Dataset*, it is mentioned that the order of the problems was, in fact, randomised across students. Table 8 shows this as well for three chosen students. It is visible that some students saw TRAPEZOID_ABCD before the other trapezoid problems, while others saw the problems in the direct opposite order. These students saw the other trapezoid problems before the TRAPEZOID_ABCD problem. Additionally, $S3$ (who only did the TRAPEZOID_ABCD problem) shows there are those students that did not see all four problems at all. This means that some students had the opportunity to learn from TRAPEZOID_ABCD (or vice versa), while others did not. The learning effect across problems is thus much more difficult to quantify. Nonetheless, the trapezoid problems do show that there is a possibility the node distribution is influenced by the order of the problems.

S1	S3	S8
TRAPEZOID_AREA	TRAPEZOID_ABCD	TRAPEZOID_ABCD
TRAPEZOID_HEIGHT		TRAPEZOID_AREA
TRAPEZOID_BASE		TRAPEZOID_HEIGHT
TRAPEZOID_ABCD		TRAPEZOID_BASE

Table 8: The order of the trapezoid problems for three chosen students.

Clearly, there is a strong learning effect across questions and a possible learning effect across problems (see **Appendix A** for another example of possible learning across problems). This learning effect can cause the students' performance to change over time. The original knowledge graph algorithm, which generates its knowledge graph based on said performance, is not equipped to incorporate the learning effect. Crucially, it assumes that the accuracies directly indicate the possession of the underlying skills in the data set (i.e., a 1 means a student possesses a skill, a 0 means they do not). The algorithm furthermore assumes that this skill possession does not change across data points.

Following these assumptions and the results of Dataset 1, it is clear that the learning effect has to be mitigated for the knowledge graph algorithm to accurately identify the skills underlying the various problems. As a first attempt at mitigating the learning effect, Dataset 2, 3, and 4 were created.

4.2.2 The Remaining Three Datasets

Dataset 2 contained only the first questions of each problem. After the pre-processing, this data consisted of 60 problem-steps for 59 students. By removing the other questions for the relevant problems (i.e., those with more than one question), the learning effect across questions is eliminated.

Unfortunately, the learning effect across problems persists in the results of Dataset 2. The data set is determined to have 4 underlying skills, which are divided across 6 nodes as per Figure 12.

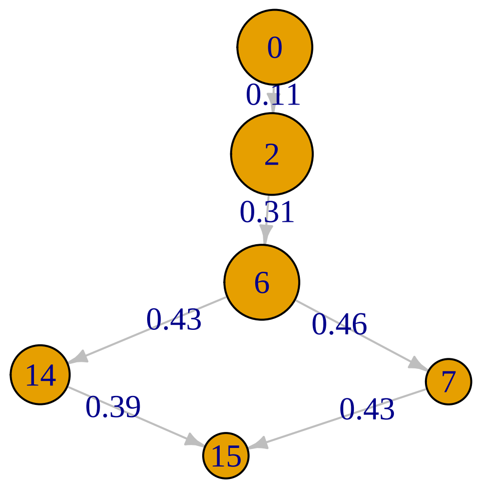


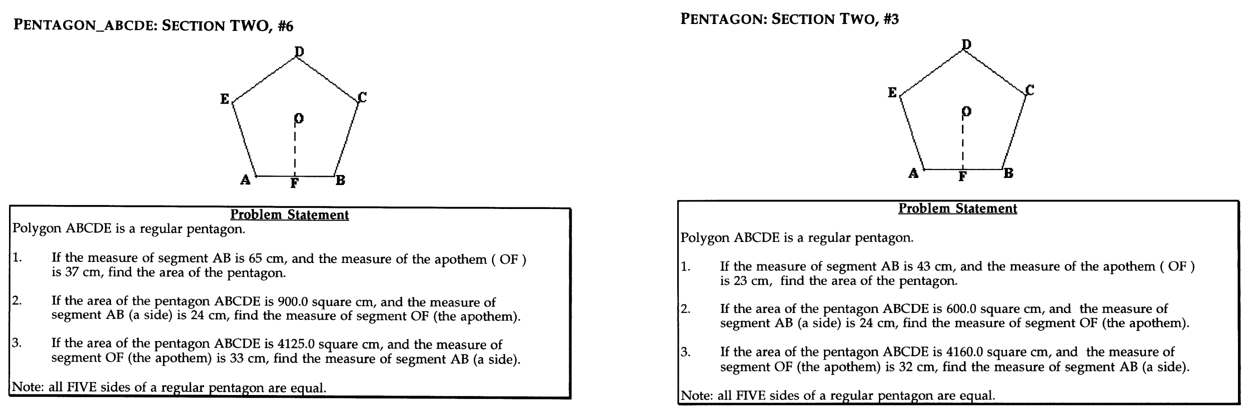
Figure 12: The knowledge graph generated from Dataset 2.

To investigate the learning effect, the trapezoid problems are analysed once more. Dataset 2 contains only (question 1 of) the `TRAPEZOID_ABCD` problem and the `TRAPEZOID_AREA` problem. Since these two problems still overlap, it is expected that the area step of these questions is placed in the same node. This is not the case, however. For the `TRAPEZOID_ABCD` problem, the area step is placed in Node 15. For the `TRAPEZOID_AREA` problem, on the other hand, the same step is placed in Node 6. Dataset 2 thus follows the same pattern as Dataset 1 did for the trapezoid problems.

For completeness sake, two additional problems are analysed, namely `PENTAGON` and `PENTAGON_ABCDE` (depicted in Figure 13). In Dataset 2, only the first question of each problem is included in the data. This question asks for the area of the pentagon and is identical across the two problems (save for different numerical values). Yet as with the trapezoid problems, the area step of these two problems are not placed in the same node. The area step of the `PENTAGON` problem is placed in Node 14, while that same step is placed in Node 7 for the `PENTAGON_ABCDE` problem.

Although Dataset 2 thus succeeds in eliminating the learning effect, it does not remove the learning effect across problems. Dataset 3 goes one step further in its restrictions: problems that are similar to each other (like `PENTAGON_ABCDE` and `PENTAGON`) are removed from the data set. Dataset 3 contained the first questions of each problem only and additionally contained but one problem for each basic shape (i.e., one circle problem, one square problem, one triangle problem, et cetera). The data set did still include the more complex problems like `PAINTING_THE_WALL` (which combine multiple shapes).

Dataset 3 resulted in a knowledge graph that contained 3 skills like Dataset 1, but with different nodes (e.g. Dataset 3 contained a Node 6 in its final knowledge graph while Dataset 1 did not). The data set further contained 43 problem-steps after pre-processing for 59 students. The amount of data that the


D
E
C
O
A
F
B
D
E
C
O
A
F
B

PENTAGON_ABCDE: SECTION TWO, #6

Problem Statement

Polygon ABCDE is a regular pentagon.

1. If the measure of segment AB is 65 cm, and the measure of the apothem (OF) is 37 cm, find the area of the pentagon.
2. If the area of the pentagon ABCDE is 900.0 square cm, and the measure of segment AB (a side) is 24 cm, find the measure of segment OF (the apothem).
3. If the area of the pentagon ABCDE is 4125.0 square cm, and the measure of segment OF (the apothem) is 33 cm, find the measure of segment AB (a side).

Note: all FIVE sides of a regular pentagon are equal.

PENTAGON: SECTION TWO, #3

Problem Statement

Polygon ABCDE is a regular pentagon.

1. If the measure of segment AB is 43 cm, and the measure of the apothem (OF) is 23 cm, find the area of the pentagon.
2. If the area of the pentagon ABCDE is 600.0 square cm, and the measure of segment AB (a side) is 24 cm, find the measure of segment OF (the apothem).
3. If the area of the pentagon ABCDE is 4160.0 square cm, and the measure of segment OF (the apothem) is 32 cm, find the measure of segment AB (a side).

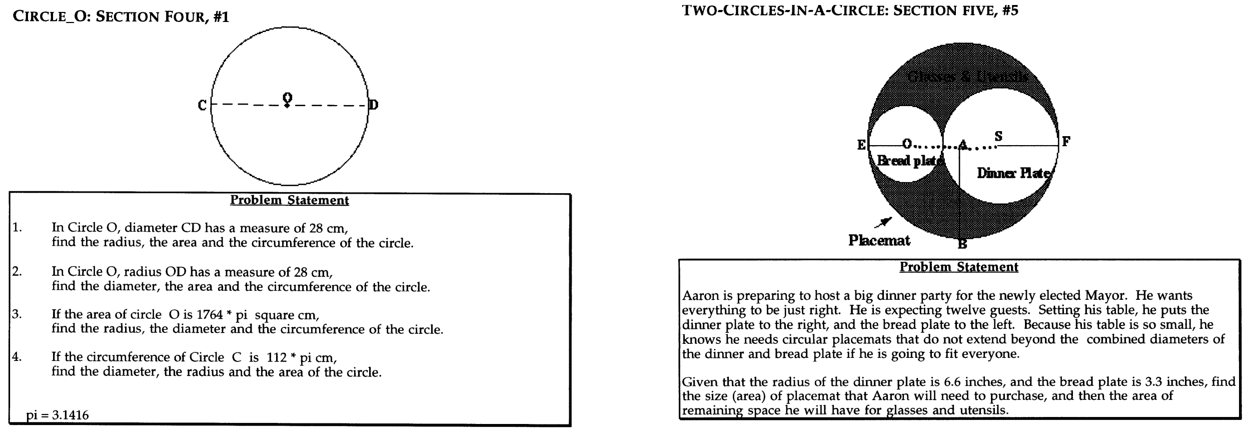
Note: all FIVE sides of a regular pentagon are equal.

Figure 13: The pentagon problems from the Geometry Area (1996-97) data set.

knowledge graph algorithm can use to generate its knowledge graph noticeably decreases with each subsequent data set.

Removing the duplicate problems from Dataset 3 ensured that learning as it seemed to occur across the pentagon and trapezoid problems could no longer occur for this data set. Nonetheless, some odd results (that contradicted intuition) persisted. Specifically, problems that appeared to be more difficult were placed in lower nodes than their simpler counterparts.

To illustrate, Figure 14 shows the `CIRCLE_O` problem (a simple problem that consists of only one shape) in comparison with the `TWO-CIRCLES-IN-A-CIRCLE` problem (which consists of two shapes within another shape). Dataset 3 contained only the first question of `CIRCLE_O`, which asks students to calculate the radius, the area, and the circumference of the circle, given the diameter. The `TWO-CIRCLES-IN-A-CIRCLE` problem consists of one question with many steps, and it involves (amongst other things) calculating the area of the bread plate (circle O) and dinner plate (circle S).


CIRCLE_O: SECTION FOUR, #1
C
O
D
TWO-CIRCLES-IN-A-CIRCLE: SECTION FIVE, #5
Glasses & Utensils
E
O
A
S
F
Bread plate
Dinner Plate
Placemat
B

Problem Statement

1. In Circle O, diameter CD has a measure of 28 cm, find the radius, the area and the circumference of the circle.
2. In Circle O, radius OD has a measure of 28 cm, find the diameter, the area and the circumference of the circle.
3. If the area of circle O is 1764 * pi square cm, find the radius, the diameter and the circumference of the circle.
4. If the circumference of Circle C is 112 * pi cm, find the diameter, the radius and the area of the circle.

pi = 3.1416

Problem Statement

Aaron is preparing to host a big dinner party for the newly elected Mayor. He wants everything to be just right. He is expecting twelve guests. Setting his table, he puts the dinner plate to the right, and the bread plate to the left. Because his table is so small, he knows he needs circular placemats that do not extend beyond the combined diameters of the dinner and bread plate if he is going to fit everyone.

Given that the radius of the dinner plate is 6.6 inches, and the bread plate is 3.3 inches, find the size (area) of placemat that Aaron will need to purchase, and then the area of remaining space he will have for glasses and utensils.

Figure 14: A basic circle problem versus a more complex problem with two circles inside a larger circle.

Intuition would not only assume that the `CIRCLE_O` problem is a simpler problem by comparison, it would also assume that the skills necessary for `CIRCLE_O` would be prerequisites for solving `TWO-CIRCLES-IN-A-CIRCLE`. At best, one could argue that both situations require solving the area of a circle, and that these steps should thus be inside the same node. This is not the case. The knowledge graph algorithm places the area step of the `CIRCLE_O` problem in Node 7 (the highest node of Dataset 3's knowledge graph), while the area step for circle S and circle O are placed in Node 0 and Node 2 respectively.

A telling difference between the two problems is that students start off with the diameter in the `CIRCLE_O` problem, while they are given the radius of the circle S and circle O of the `TWO-CIRCLES-IN-A-CIRCLE` problem. This could explain why calculating the areas of circle S and circle O is easier for students than calculating the area for the `CIRCLE_O` problem. However, the radius step of the `CIRCLE_O` problem is placed in Node 2. Thus, if finding the radius from the diameter is not so difficult (and requires only 1 skill; given that the bit-string of Node 2 is `010`), this difference does not explain why the area step of the `CIRCLE_O` problem is thought to require all three skills in the data set, while the area steps of circle S and O are thought to require no skills and 1 skill respectively.

Clearly, something is still not right with the classification of the problems into their respective nodes. As a final attempt to mitigate any learning that may occur across problems, Dataset 4 looked only at the problems (rather than the problem-steps) and again, removed any duplicate problems (for the basic shapes). This data set resulted in the node classification shown in Table 9.

Problem	Node	Accuracy
PARALLELOGRAM_ABDE	0	0.93
RECTANGLE_ABCD	0	0.91
SQUARE_ABCD	0	1
TRIANGLE_ABC	0	0.85
BUILDING_A_SIDEWALK	1	0.70
DOG_ON_A_ROPE	1	0.63
LAWN_SPRINKLER	1	0.63
TRIANGLE_TRIANGLE	1	0.61
LAWN_SPRINKLER_2	3	0.47
ONE_CIRCLE_IN_SQUARE	3	0.5
PAINTING_THE_WALL	3	0.44
WATERING_VEGGIES	3	0.5
COVERING_POOL	5	0.35
TWO_CIRCLES_IN_SQUARE	5	0.28
CIRCLE_O	7	0.32
ONE_CIRCLE_IN_CIRCLE	7	0.37
PENTAGON	7	0.41
TRAPEZOID_ABCD	7	0.30
TRIANGLE_RECTANGLE	7	0.3
TWO_CIRCLES_IN_CIRCLE	7	0.31

Table 9: The node classification of Dataset 4, plus the average accuracies (rounded to 2 decimals) for each problem.

Some of the nodes in Table 9 match well with what would intuitively feel like a correct classification. Very simple shapes like rectangles, squares, and triangles are all placed in Node 0. It would not be strange to conceive that students already have some prior knowledge on how to solve for these shapes and would thus not require the underlying skills of Dataset 4 to solve these problems. Generally, the progression of the nodes for these problems makes some sense (e.g. that the `LAWN_SPRINKLER_2` problem relies on skills needed for the `LAWN_SPRINKLER` problem or that pentagon and trapezoid shapes come later in the knowledge graph than the triangle and rectangle shapes).

However, there are also still classifications that make less sense, such as why the `CIRCLE_O` problem is lower in the knowledge graph than the `ONE_CIRCLE_IN_SQUARE` and `TWO_CIRCLES_IN_SQUARE` problems (intuition would dictate that *both* the square problem and the circle problem would be prerequisites for these problems). With information about the steps removed through aggregation, the node classifications are harder to analyse. Information is missing about what exactly (i.e. which steps) cause a problem to be perceived as more difficult (or, requiring more skills) by the knowledge graph algorithm. Additionally, the data set consists of only 20 problems, meaning some information may be lost between problems. Overall, aggregating over the steps does not seem like the best solution to the learning effect problem.

In fact, none of the data sets seem to succeed entirely in mitigating the learning effect. Even if the learning effect is less pronounced for some data sets compared to others, it is clear that the order of the problems and the questions does influence the final knowledge graph that is generated by the knowledge graph algorithm. The best solution for this problem, as this subsection demonstrated, is not to further subset the original data, but to adjust the knowledge graph algorithm to compensate for order.

4.3 Compensating for Order

When compensating for the order of the problems, the one property of the data that comes back is that the order is randomised across students. It is thus not possible to compensate for a global order; the compensation has to be done for each student separately.

First, the order each student did each problem in is recorded in a separate data frame. This data frame has the problems as its columns and the students as its rows. Each cell contains an index that represents when a student S did a problem P. The index was determined through the `match` function in base R and compared the problem names with the student data. If a student did not do a certain problem, an *NA* is recorded for that student and problem combination. Table 10 provides a visual aid for what this *order* data frame looked like.

	P_1	P_2	$P_{...}$	P_N
S_1	1	2	...	N
S_2	*NA*	4	...	21
$S_{...}$	...	...	...	...
S_M	2	1	...	*NA*

Table 10: The schema of the *order* data frame filled with some arbitrary numbers.

Once created, the order data frame is utilised in the knowledge graph algorithm to compensate for the learning effect. Specifically, the order is considered in the problem comparison on the dimensions of sameness, moreness, and lessness. New rules are applied for determining when two problems are the same, when one problem is more than another, or when one problem is less than another. These rules are:

1. Two problems P_x and P_y are the same only if they are both incorrect. When the problems are both correct, it is possible that P_x has been learned since the occurrence of P_y, or vice versa.

2. A problem P_x is considered more than a problem P_y when P_x is correct and P_y incorrect. This only holds if P_x occurred before P_y. Otherwise, it is possible that P_x has been learned since the occurrence of P_y.

3. A problem P_x is considered less than a problem P_y when P_x is incorrect and P_y correct. This only holds if P_x occurred after P_y. Otherwise, it is possible P_y has been learned since the occurrence of P_x.

The rules remove the effect of learning by taking into account that, if a problem occurred later in time, it could have since been learned. Table 11 shows that the new rules reduce the data the comparison matrices can account for from 100% to about 35.8%. With the original algorithm, all problems were either classified as being the same, being more, or being less than each other. With the new algorithm, 65.2% of the problems cannot be classified into any of the comparison matrices. Consequently, the algorithm cannot take these problems into account when creating its knowledge graph.

	`M`(sameness)	`M`(moreness)	`M`(lessness)	Total Sum
original algorithm	0.574	0.213	0.213	1
updated algorithm	0.130	0.114	0.114	0.358

Table 11: The mean (`M`) of each comparison matrix and their addition.

While the method for compensating the order is thus simple in its ease of implementation, it is also very extreme in the problems it discards due to a possible learning effect. In reality, it is likely learning does not occur between all problems. Rather, it seems more probable that learning would earlier occur between problems that are very similar than between problems that are not alike at all. However, to determine the similarity between problems, some new metric would have to be introduced that, like the goodness of fit, has no ground truth. Although it seems likely that problems like `CIRCLE_0` and `TWO-CIRCLES-IN-A-CIRCLE` would exhibit more overlap and result in more learning between them, it is equally possible that practicing multiplication and division for squares and rectangles makes it easier to solve circle problems as well. Since it is difficult to determine how much learning occurs between tasks, the simple method implemented by the above rules serves as a good method for mitigating any learning that may occur between problems, even if much of the data gets disregarded as a result.

The new rules are implemented in the `same.as`, `more.than`, and `less.than` functions. Since these functions lie at the core of the knowledge graph algorithm, the change propagates through to the comparison matrices, the penalty matrices, and subsequently, to the goodness of fit that determines the final knowledge graph that comes out of the algorithm. With this change implemented, it is possible to produce a new knowledge graph for Dataset 1 with the updated knowledge graph algorithm.

4.4 The Final Knowledge Graph

The updated knowledge graph algorithm generated the knowledge graph shown in Figure 15 for Dataset 1. It is immediately obvious that this knowledge graph looks quite different from the one shown in Figure 8 (which was generated by the original knowledge graph algorithm). The new knowledge graph contains 7 skills, rather than 3, and consists of 11 nodes. It also shows much lower average fits between the nodes, although, as was stated in the original analysis, this residual goodness of fit is not a sufficient metric on its own for determining how well the knowledge graph has identified the skills underlying the data set.

To get a better idea of how 'good' this new knowledge graph is, the problems that showed problematic results initially are re-examined. For the `PAINTING_THE_WALL` problem, all steps of all questions are placed in Node 0. Improving on the knowledge graph from Figure 8, the steps are now all placed in the same node. This matches the intuition that steps which are the same should be in the same node.

Similarly, the three questions from `TRAPEZOID_ABCD` are placed in the same node (Node 8) in the new knowledge graph as the `TRAPEZOID_AREA`, `TRAPEZOID_BASE`, and `TRAPEZOID_HEIGHT` problems. To some degree, updating the knowledge graph algorithm thus seems to have mitigated some of the learning effect.

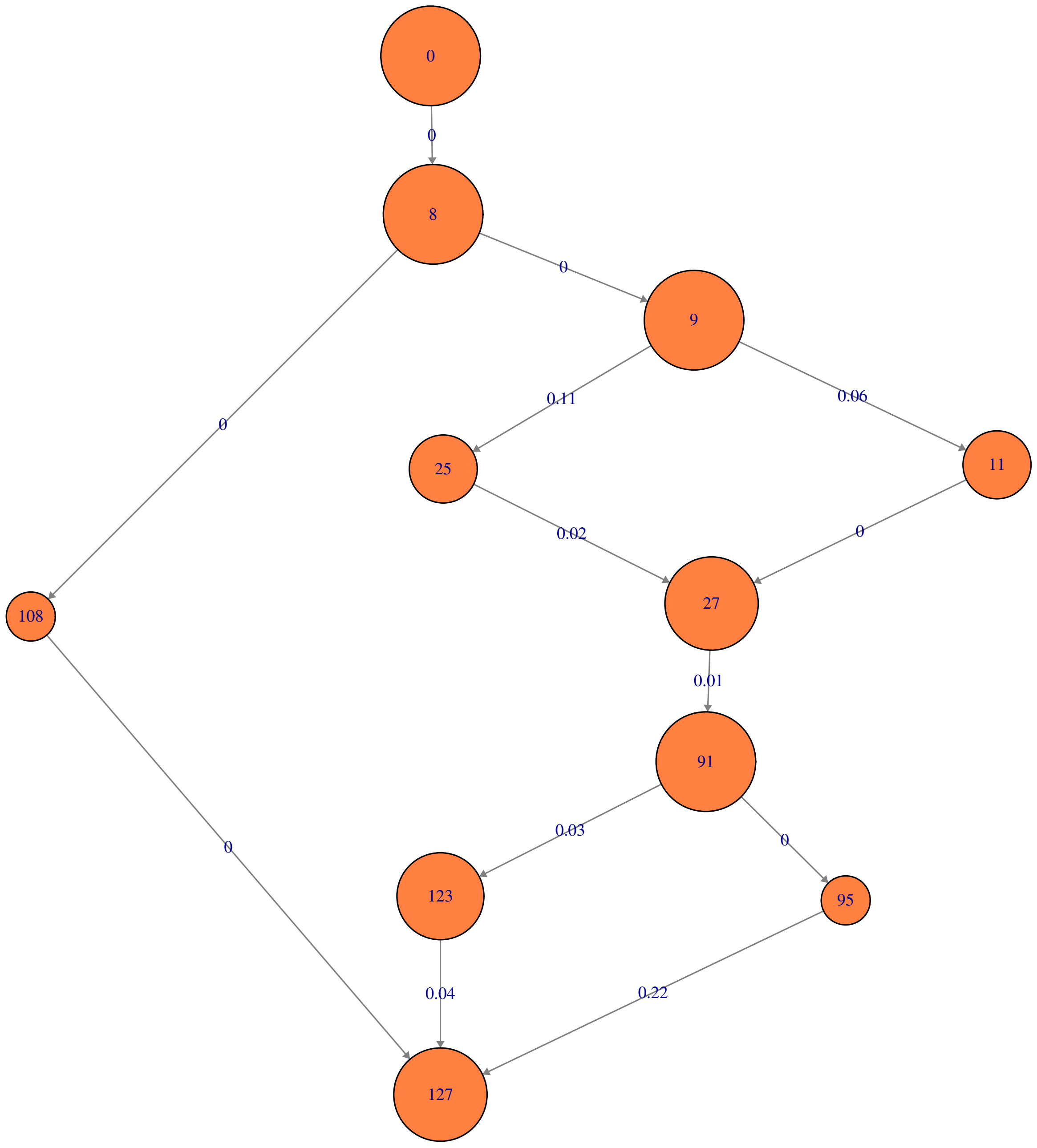
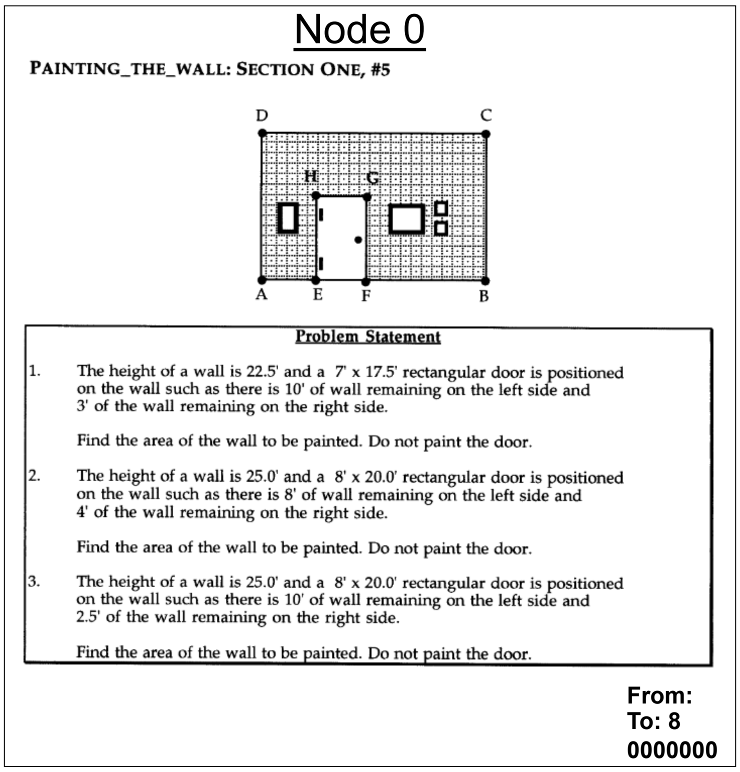


Figure 15: The knowledge graph generated from Dataset 1 with the updated knowledge graph algorithm.

However, there are still classifications that make less sense intuitively. The pentagon problems from Figure 13 are still classified into different nodes (`PENTAGON_ABCDE` has steps in Node 25 and 27, while the `PENTAGON` steps are all in Node 9). Similarly, the classification of the additional problems discussed in **Appendix A** continues to be inconsistent, with identical questions being spread across different nodes.

Thus, the updated knowledge graph algorithm seems to have improved the classification of some but not all problems in terms of the learning effect. For the situations that did not improve, it is difficult to identify a cause. The updates in the knowledge graph algorithm at least rule out learning as a possible cause, but there are many other reasons why certain contradictions to intuition might be observed. To give just one: it is possible that the order randomisation is still having an effect on the algorithm. With the updated algorithm, it becomes possible that two of the same problems are classified differently between two different students, if those students did said problems in a different order. Thus, a step like area for the `PENTAGON_ABCDE` problem might be more than that same step for the `PENTAGON` problem for a student S_x, but not more for a different student S_y. Again, since there is no clear ground truth for the skills underlying the data set, it is hard to say what is happening cognitively when students solve the various problems.

When looking at intuition, the least that can be said is that the knowledge graph from Figure 15 appears to improve on the one from Figure 8. It notably removes the learning effect observed in some problems, and it also shows a good progression of the problem difficulty. Unlike the knowledge graph generated from Dataset 4, the knowledge graph in Figure 15 places most of the basic shape problems before their more complex variants that include shapes within shapes. For example, all basic circle problems are placed in nodes higher in the knowledge graph than the `TWO-CIRCLES-IN-SQUARE` and `TWO-CIRCLES-IN-CIRCLE` problems (which are in Node 127).



Node 0

PAINTING_THE_WALL: SECTION ONE, #5

Problem Statement

1. The height of a wall is 22.5' and a 7' x 17.5' rectangular door is positioned on the wall such as there is 10' of wall remaining on the left side and 3' of the wall remaining on the right side.

   Find the area of the wall to be painted. Do not paint the door.

2. The height of a wall is 25.0' and a 8' x 20.0' rectangular door is positioned on the wall such as there is 8' of wall remaining on the left side and 4' of the wall remaining on the right side.

   Find the area of the wall to be painted. Do not paint the door.

3. The height of a wall is 25.0' and a 8' x 20.0' rectangular door is positioned on the wall such as there is 10' of wall remaining on the left side and 2.5' of the wall remaining on the right side.

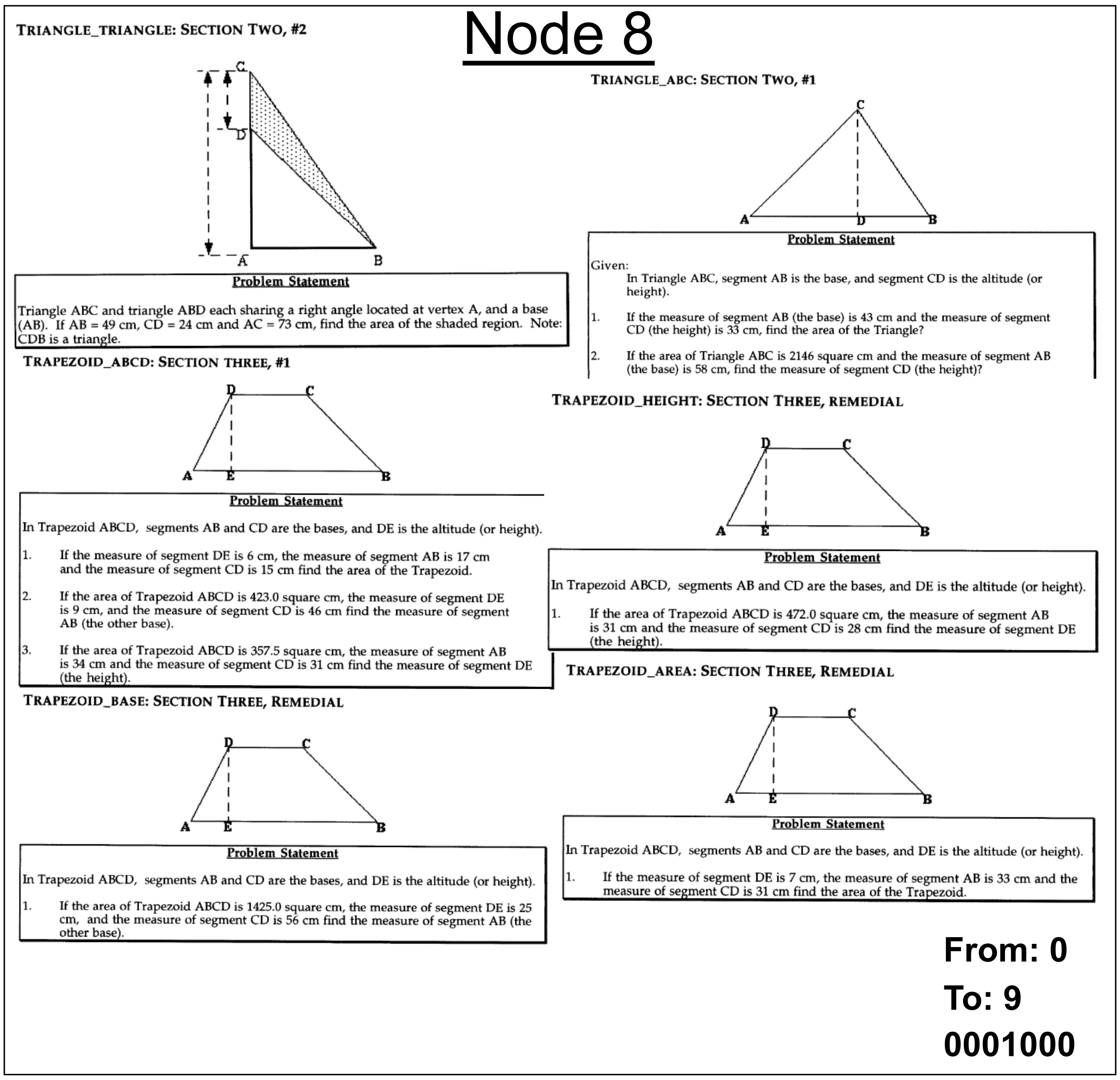
   Find the area of the wall to be painted. Do not paint the door.

From:
To: 8
0000000

Figure 16: Node 0 of the knowledge graph contains all steps of the `PAINTING_THE_WALL` problem.

Given this apparent improvement (visible also in the lower fit scores), the knowledge graph generated from Dataset 1 with the updated knowledge graph algorithm is considered the final knowledge graph. Although it still shows some discrepancies when observed intuitively, it also shows the best fit to date. Furthermore, since a definite cause behind the remaining discrepancies cannot be identified, there are no straightforward ways left to improve the knowledge graph algorithm (from this available data).

Having generated a final knowledge graph that shows a decent fit to both the data and intuition, the knowledge graph will next be used to create a set of cognitive models. The cognitive models are meant to make the skills identified by the knowledge graph concrete, as well as to show how these skills integrate with one another to solve the relevant problems. The next section will explain in more detail how the cognitive models are created from the knowledge graph given here. For time reasons, the cognitive models will focus on Node 0 and 8 from the knowledge graph. The content of these nodes can be seen in Figures 16 and 17 respectively. The full distribution of problems into nodes for the final knowledge graph can be found in **Appendix B**.


Node 8

TRIANGLE_TRIANGLE: SECTION TWO, #2

Problem Statement

Triangle ABC and triangle ABD each sharing a right angle located at vertex A, and a base (AB). If AB = 49 cm, CD = 24 cm and AC = 73 cm, find the area of the shaded region. Note: CDB is a triangle.

TRIANGLE_ABC: SECTION TWO, #1

Problem Statement

Given:
In Triangle ABC, segment AB is the base, and segment CD is the altitude (or height).

1. If the measure of segment AB (the base) is 43 cm and the measure of segment CD (the height) is 33 cm, find the area of the Triangle?

2. If the area of Triangle ABC is 2146 square cm and the measure of segment AB (the base) is 58 cm, find the measure of segment CD (the height)?

TRAPEZOID_ABCD: SECTION THREE, #1

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the measure of segment DE is 6 cm, the measure of segment AB is 17 cm and the measure of segment CD is 15 cm find the area of the Trapezoid.

2. If the area of Trapezoid ABCD is 423.0 square cm, the measure of segment DE is 9 cm, and the measure of segment CD is 46 cm find the measure of segment AB (the other base).

3. If the area of Trapezoid ABCD is 357.5 square cm, the measure of segment AB is 34 cm and the measure of segment CD is 31 cm find the measure of segment DE (the height).

TRAPEZOID_HEIGHT: SECTION THREE, REMEDIAL

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the area of Trapezoid ABCD is 472.0 square cm, the measure of segment AB is 31 cm and the measure of segment CD is 28 cm find the measure of segment DE (the height).

TRAPEZOID_BASE: SECTION THREE, REMEDIAL

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the area of Trapezoid ABCD is 1425.0 square cm, the measure of segment DE is 25 cm, and the measure of segment CD is 56 cm find the measure of segment AB (the other base).

TRAPEZOID_AREA: SECTION THREE, REMEDIAL

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the measure of segment DE is 7 cm, the measure of segment AB is 33 cm and the measure of segment CD is 31 cm find the area of the Trapezoid.

From: 0
To: 9
0001000

Figure 17: Node 8 contains 4 trapezoid problems and 2 triangle problems.

5 Cognitive Modelling

Two cognitive models were made on the basis of the final knowledge graph from Section 4. The models were made in the PRIMs cognitive architecture (Taatgen, 2013). At its core, PRIMs works by defining the skills that make up a task. Each skill consists of its own operators (as also explained in the *Theoretical Background*), and each tasks consists of various skills. A model in PRIMs is always meant to reproduce the cognition behind one task, but the skills defined in each task are meant to be general enough for reuse between different tasks.

When a cognitive model in PRIMs is run, it continuously checks its internal state to see which skills it can execute. By executing its skills in the correct order, the model can successfully complete its tasks. It is also possible for the model to fail to complete its task. This can happen when the model executes its skills in an incorrect order, when it fails to successfully execute one of its skills, or when it is missing a skill that is required to successfully complete its task.

Given the inner workings of the PRIMs cognitive architecture, it is a well-suited architecture for creating the cognitive models in this thesis. These models after all, serve two critical purposes. Firstly, they are meant to make the identified skills from the knowledge graph concrete. PRIMs allows for this through its explicit skill definition. Secondly, by creating general skills as PRIMs intends, the models can be used to show how these skills can be reused between different nodes (which each contain different tasks).

Since the models are thus meant to show how skills can be reused between tasks, rather than one cognitive model, two cognitive models are created in this section. The first model identifies the skills necessary to solve the problems from Node 0. It is assumed Node 0 does not contain any of the underlying skills in the data set (i.e., the associated binary string is `0000000`), and as such, the skills in this first model are considered the students' prior knowledge.

The second model, on the other hand, will contain one of the skills underlying the data set. Specifically, it will contain the skill `0001000` (read: skill 8), which lies at the core of Node 8. The purpose of this second model is thus to solve all the problems in Node 8 by successfully identifying this skill. Since the knowledge graph shows a one skill difference between Node 0 and Node 8, the model of Node 8 cannot contain any other additional skills than skill `0001000`. All the other skills it may need, it should be able to reuse from Node 0.

In this section, first it will be explained how the model for Node 0 was set up. This first model went through two iterations, and both of its versions will be explained in the relevant subsections. After describing the model for Node 0, the model for Node 8 will be given. An important subsection is dedicated to identifying skill `0001000`, before the model of Node 8 can be explained.

After both models have been described, some results are given that show how the models performed. Importantly, the model performance is expected to provide answers for the research question: "Can knowledge graphs be used to identify the skills reused between tasks?". If the model performance is sufficient (i.e., the models can each solve their respective problems), this supports the use of knowledge graphs as tools for identifying the skills reused between tasks. If the model performance is insufficient (the models fail to solve one of the problems in their respective nodes), this would indicate the skills identified by the knowledge graph were not accurate enough for the successful completion of the relevant tasks.

5.1 Representing the Problems

The tasks the models will set out to complete are the problem-steps in their respective nodes. In order to solve these tasks, both Model 0 and Model 8 require an internal representation of these tasks. This internal representation can take different forms, but it will always be in the *visual buffer* of the model. The visual buffer is how the model sees the world.

In simple situations, a visual cue for a subject can be stored in a single *chunk* in the visual buffer. This chunk will have various, numbered slots, so (with an eye on Figure 16) $V1$ might be set to the outer wall of the `PAINTING_THE_WALL` problem, while $V2$ might contain a representation of the door. Each slot of the visual chunk represents a single piece of information, akin to a single fact (or a single *string* in programming lingo). Together, the various slots can be used to represent a whole visual input.

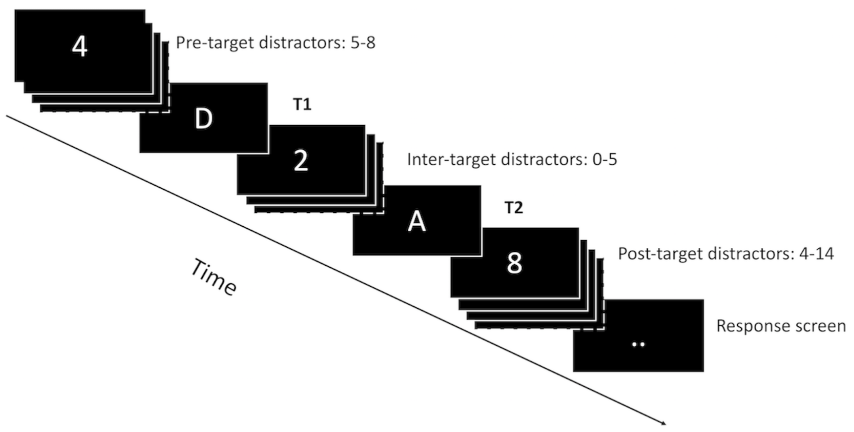

4
Pre-target distractors: 5-8
D
T1
2
Inter-target distractors: 0-5
A
T2
8
Post-target distractors: 4-14
Response screen
..
Time

Figure 18: The set-up of Experiment 1 in Hoekstra et al. (2020). This experiment has a simple visual input.

As was said, if the visual input is simple, then a single chunk is sufficient to represent this input. A paradigm like the attentional-blink, for example, (discussed in the *Theoretical Background* in the context of Hoekstra et al., 2020) can be represented by a single chunk, because its visual cues (shown in Figure 18) can be as simple as single letters and digits. This visual input could thus be captured by a single chunk, which furthermore would only need to consist of a single slot.

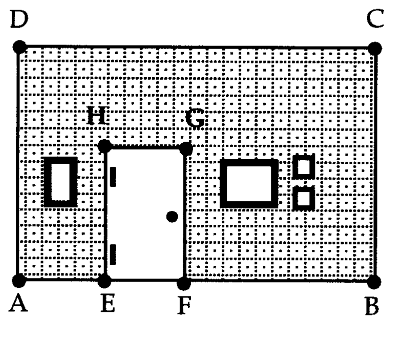


Figure 19: The visual cue that needs to be internalised by the model for the `PAINTING_THE_WALL` problem.

The issue that comes up with representing the problems of Dataset 1, however, is that their associated visual input is never particularly simple. Each problem in the data set has a corresponding shape that students are given as a visual aid, and each shape consists, at the very least, of multiple segments (where a segment is a line denoted by two or more letters). Some problems can even consist of multiple shapes within each other, and each of these shapes will again consist of its own segments. Figure 19 shows how complex such a visual input can be.

At the top level, Figure 19 consists of a rectangle with a base made up by AB and a height of AD. Within that rectangle, there is another rectangle with the base EF and the height EH. Clearly, there is a hierarchy to this visual input. If each segment is then stored in its own slot, information about this hierarchy is lost. Ideally, the visual cue would thus be internalised by a representation that can capture its hierarchy.

In PRIMs, capturing this hierarchy is possible by defining a visual input that consists of multiple chunks. Each chunk represents a different item in the hierarchy of the visual input, as shown in Figure 20. The highest level of this hierarchy is always a screen, as defined in Listing 4.

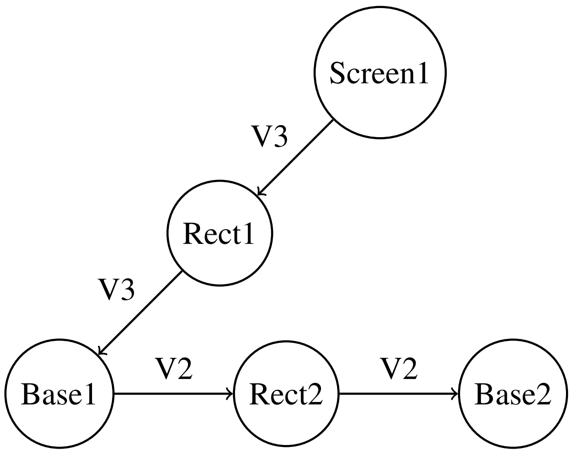


Figure 20: The hierarchy of the visual cue shown in Figure 19.

Generally, Listing 4 shows the definition of the visual input via multiple visual chunks. Each visual chunk is proceeded by a unique identifier. After this identifier, slot $V1$ holds information about what the chunk is representing. In the case of Figure 19, a chunk can be either a screen, a rectangle, or a base. With an eye on problems other than the `PAINTING_THE_WALL` problem, visual chunks can also represent different shapes (like triangles and trapezoids) and heights (on top of bases).

Listing 4: An example that shows how the visual hierarchy from Figure 20 can be defined in the cognitive model.

```
define visual {
   // Screen for question 1 of the painting-the-wall problem
   (screen1 screen nil rect1 shaded-area)
   (rect1 rectangle nil base1 nil 22.5 nil)
   (base1 base rect2 nil 3 nil)
   (rect2 rectangle base2 nil 7 17.5 nil)
   (base2 base nil nil 10 nil)
}
```

Beyond the $V1$ slot, the $V2$ and $V3$ slots hold information about the hierarchy of the items in the visual input. The $V2$ slot holds the next item on the same level of the hierarchy, as shown between e.g. Base1 and Rect2 in Figure 20. The $V3$ slot holds the next item one-level down from the current item, like with Screen1 and Rect1. Whether to the next item of the same level or of a level down, the >> operator allows for the cognitive model to shift its visual focus to the next item in the hierarchy. Vice versa, the << operator allows the visual focus to be brought back to the top-level item (i.e., the item that sprouted the current level of the hierarchy).

While the $V2$ and $V3$ slots thus control the hierarchy of the visual input, the remaining slots of each visual chunk define it. The number of slots and their content depend crucially on what each chunk is representing (the $V1$ slot). The $V4$ slot of a screen, for example, holds the goal for that screen. In the case of the `PAINTING_THE_WALL` problem, the goal is to find the shaded region, which is the area to be painted. Bases and heights will have a base and height respectively (with the inapplicable slot set to *nil*). Shapes, like the rectangles of Figure 19, will have a base, a height, and an area. Some shapes, like trapezoids, can have two bases. In these cases, each base is represented by its own chunk and connected to its main shape through the $V3$ slot.

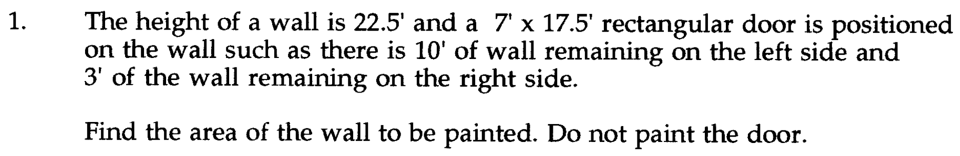
If information is missing (as e.g. the full base of the wall is for the `PAINTING_THE_WALL` questions), it is set to *nil* in the visual chunk until the student solves for that problem-step. Once a student has written down a solution, the visual chunk is updated to reflect this. Additionally, a slot is also set to *nil* if it is not applicable for a certain item (e.g. when it is the last item of a hierarchy or the height slot of a base).

Having defined the structure of the various chunks in Listing 4, the visual input defined therein can now be coupled back to Figure 19. The height of the wall DA is set in slot $V5$ of Rect1 (and equal to 22.5 feet for question 1 of the `PAINTING_THE_WALL` problem). The full base of the wall (AB) is not given in question 1 (and thus set to *nil* in Listing 4), but it is visible that this base consists of the base of the door plus segments AE and FB. Segments AE and FB are given by Base1 and Base2 respectively. The base of the door is given by slot $V4$ of Rect2. Its height is stored in its slot $V5$. Finally, the task assigned by question 1 (to find the shaded area) is encapsulated by the $V4$ slot of the screen chunk.

Separately, each of the chunks in Listing 4 represents only one part of the visual input. By looking at these visual chunks as a holistic whole, however, the chunks can effectively be used to represent the problems of Dataset 1. Given this problem representation, the next subsections tackle how the two models use this problem representation to solve the problems of their respective nodes.

5.2 Model 0

Model 0 represents the cognition behind the problems in Node 0. As Figure 16 shows, this node contains only one problem: the `PAINTING_THE_WALL` problem. In Dataset 1, this problem is distinguished by three steps: finding the area of the door, finding the area of the wall, and finding the area of the shaded region of Figure 19. The latter area is the area that needs to be painted, according to the problem statement of the questions.



1. The height of a wall is 22.5' and a 7' x 17.5' rectangular door is positioned on the wall such as there is 10' of wall remaining on the left side and 3' of the wall remaining on the right side.

   Find the area of the wall to be painted. Do not paint the door.

Figure 21: Question 1 of the `PAINTING_THE_WALL` problem.

Figure 21 shows the first question of the the `PAINTING_THE_WALL` problem. Question 2 and question 3 are identical to this question, save from that they contain different values for the various variables. Each question consists of the same three steps mentioned in the previous paragraph, and each question-step combination has its own entry in Dataset 1.

The model for Node 0 must thus be able to perform these three steps, and it must be able to do so regardless of the values of the variables.

Version 1 of Model 0 approaches the steps directly. The model is created so that it can exactly reproduce all questions of the `PAINTING_THE_WALL` problem. Version 2 of Model 0, however, is created to serve as a basis for Model 8. It is more general in its skills and its overall approach. The following two subsections will explain Version 1 and Version 2 of Model 0 respectively.

Regardless of the version, both models start with the same task definition, which sets the model's initial goal and its parameters. The task definition is given by Listing 5 for reproducibility. For a full explanation of the different parameters, consult Taatgen (2022).

Listing 5: The task definition of Model 0.

```
define task shaded-area {
  initial-goals: (read-task) // the skill the model starts with
  default-activation: 1.0 // All chunks defined in this model receive a
     fixed baselevel activation of 1.0
  ol: t // optimised-learning
  rt: -2.0 // retrieval-threshold
  lf: 0.2 // latency-factor
  default-operator-self-assoc: 0.0
  egs: 0.05 // utility noise
  retrieval-reinforces: t
}
```

5.2.1 Version 1

Version 1 of Model 0 is designed specifically to solve the questions of the `PAINTING_THE_WALL` problem. It does so in a direct way and is designed entirely by looking at what skills would be required to solve the questions.

Before any questions can be solved, it is necessary for the model to read the questions and from them, gain an understanding of its task. For this reason, Model 0 (regardless of its version) is always initialised with the `read-task` skill (through the `initial-goals` variable in Listing 5). Through this initialisation, it is guaranteed that the `read-task` skill will always execute first at the start of a model run.

The `read-task` skill is defined by one, relatively simple, operator, as shown in Listing 6. The operator reads the task out of the $V4$ slot of the screen and places it in the model's $G1$ slot. The $G1$ slot contains the model's current goal. It is originally set to `read-task` (by the task definition), but will be set to `shaded-area` (the content of slot $V4$ as shown in Listing 4) after the `read-task` skill has executed successfully. The `shaded-area` goal tells the model that its overall task is to find the area of the shaded region of the wall shown in Figure 19.

Listing 6: The first version of the read-task skill.

```
define goal read-task {
   operator read {
      V1 = screen
      V4 <> nil
   ==>
      V4 -> G1
      >>V3 // shift focus to the first shape
   }
}
```

The goal of the model guides it through its skill execution. By changing the goal, the model can be nudged towards certain skills over others. This 'nudging' consists of two elements. Firstly, skills can be bound to specific goals. When the model has to execute its next skill, the goal binding limits the skills it can choose from. Skills with goals that do no match its current goal cannot be executed successfully. In this way, the model is encouraged towards skills that either match its current goal or to those that do not have goal restrictions.

Secondly, when a skill's name matches the model's current goal, the skill receives a *spreading activation* from the goal. Without going into the concept of activation in too much detail, in simple terms, the spreading activation makes it more likely for the model to choose a skill whose name matches with its current goal. By changing the goal to `shaded-area`, it is thus very likely the model will execute the `shaded-area` skill after it finishes reading its task.

The `shaded-area` skill is a high-level skill. It tells the model exactly how to calculate the shaded area. It does so in three parts. First, it tells the model which information it needs to calculate the shaded area. If the model has this information, or can read it off the screen, the skill next tells the model what to do with said information. If the model successfully completes the required actions, it will end up with the calculation of the shaded area and it will thereby complete its task. The `shaded-area` skill also tells the model what to do if it is missing the information it needs. In that case, the skill guides the model to other skills, which can be used to find the missing information.

Listing 7: The operators of the shaded-area skill.

```
// Find a shaded area, i.e. the area of a shape minus the other areas
   within that shape
define goal shaded-area {
   // Find the area of the top shape
   operator top-area {
    ...
   }

   // If an area is missing, find that first
   operator missing-area {
   ...
   }

   // Subtract intermediate areas from the top area
   operator subtract-area {
   ...
   }

   // Move on to the next shape within the top shape
   operator next-shape {
    ...
   }

   // This runs if the next V2-level item is not a shape but e.g. a base
      or height
   operator not-a-shape {
    ...
   }

   // Keep WM1 as the total of the shaded-area calculations
   operator update-shaded-area {
    ...
   }

   // Finish the shaded area calculation and signal the writing action
   operator end-shaded-area {
   ...
   }
}
```

Listing 7 shows the exact operators of the `shaded-area` skill. The skill consists of seven operators in total. Each operator represents a different part of the process of finding the area of a shaded region.

It is assumed a shaded region highlights one part of a top shape. Other shapes within that shape must be retracted from the top shape to find the shaded region.

As such, the `shaded-area` skill begins by looking at the area of the top shape. If that area is given in the visual input, then the skill can move on to the next shape within that top shape (through the `next-shape` operator). The skill iterates over all shapes and subtracts the area of said shape from the area of the top shape. When there are no more shapes to iterate over, the model has finished calculating the shaded area. The `end-shaded-area` operator triggers and the shaded area is written down through the `writing` action.

If the area of any of the shapes is missing, however, the `missing-area` operator will trigger. This operator changes the goal from `shaded-area` to `area`. As with the `shaded-area` skill, there exists an `area` skill that triggers when the goal is set to `area`.

The `area` skill works similarly to the `shaded-area` skill in that it has a success scenario and a failure scenario. In the success scenario, a `base-times-height` operator executes successfully. It takes the base and a height stored in the $V4$ and $V5$ slots of its current visual focus and multiplies them. For the multiplication (as for the subtraction in the `shaded-area` skill), a separate `math` skill is used. The `math` skill is triggered by setting a `fact-type` variable to `multiply-fact` and by storing the base and height in the first and second slot of the *working memory buffer* (which represents the brain's short-term, or, working memory) respectively.

The `math` skill is in charge of actually performing mathematical operations. In this model, said performance is simplified to "remembering" the correct answer. It is assumed that students were able to use some type of aid (like a calculator) to solve the math problems. This assumption was made because no clear difference was found between easier and more difficult calculations in the node analysis. Additionally, cognitive tutors try to limit students from performing cognitive actions that the tutor cannot keep track of (i.e. something the student does without interacting with the tutor). Thus, it is unlikely students were expected to solve the mathematical equations themselves.

As such, in the cognitive model, all answers to the mathematical equations are stored in the *declarative memory* of the model (the model's long-term memory). The `math` skill simply asks the model to remember the answer to a specified equation. Given the base-level activation of these memories (specified in the task definition as 1.0) and the time the model runs, it is very unlikely it will not know the answer to any of the mathematical equations it needs to solve.

After the `math` skill has found the answer, an `end-area` operator is triggered in the `area` skill which instructs the model to write down the area it has found. The `writing` action is executed which results in a change in the visual input. This concludes a successful scenario for the `area` skill.

In the failure scenario, the `base-times-height` operator in the `area` skill cannot execute correctly. In the context of the `PAINTING_THE_WALL` problem, this occurs once for each question, since the base of the wall is always missing. Without a base and a height, the `base-times-height` skill will not execute. In this case, there is a `missing-base` operator in the `area` skill that will execute instead. This operator passes the baton along (once more) to a `base` skill.

The `base` skill is used to calculate a base from various segments. The skill iterates over the items on the $V2$ level of the visual hierarchy. If those items have a base (which will be stored in the $V4$ slot), then it adds these bases up to a total base. This total base is assumed to be the base of the top shape. Once there are no more items on the $V2$ level to iterate over, the `end-base` operator is executed. It triggers the `writing` action, and the found base is written down (with its associated update of the visual input).

After an answer has been written down, the visual focus of the model is reset to the top shape. The model starts at the `shaded-area` skill, sees if it now has the information it needs, and if it does not, it follows the same path it did before. It tries the `area` skill to find the area, and if that fails, it tries the `base` skill to find its missing base. Figure 22 gives a schematic overview of this strategy.

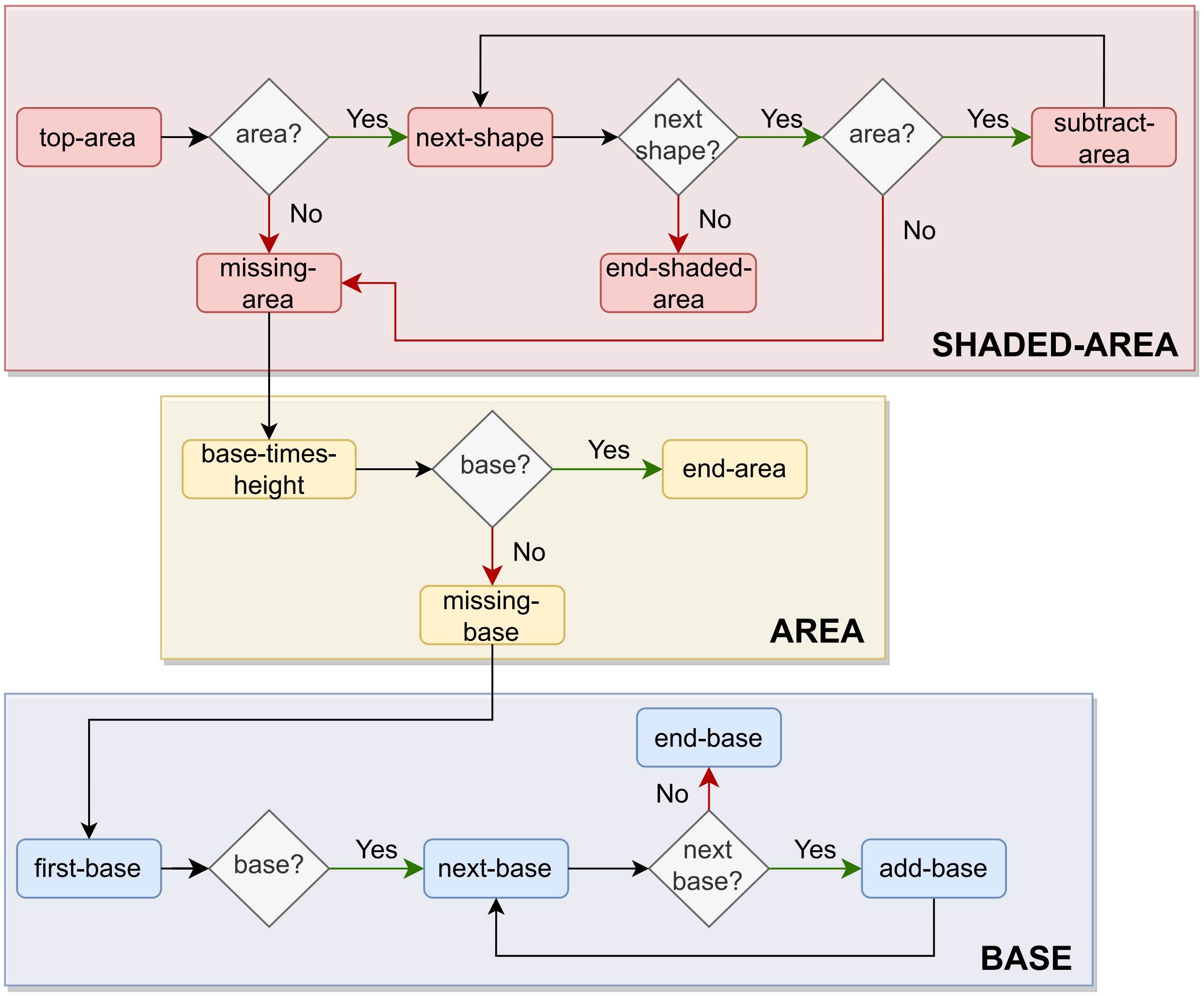


Figure 22: A schematic overview of the workings of version 1 of Model 0. The rectangles represent operators, while the diamonds represent variable checks (is variable X available, yes or no?).

By writing down each intermediate answer (and thereby updating the visual input), the model will eventually be able to successfully execute the `shaded-area` skill. In this way, it can solve the overarching task, which boils down to solving all three questions of the `PAINTING_THE_WALL` problem.

The strength of version 1 of Model 0 is that it aligns well with how the students solve the problems in the cognitive tutor. There, students were also allowed (and requested) to write down intermediate answers (in the worksheet shown in Figure 3). By writing down their intermediate answers, the cognitive load of each problem is lowered, because students have less information they need to remember at any one point in time.

The weakness of this first version of Model 0 is that it is very specific. The `area` skill has an operator for finding a base, but it does not have an operator for finding a height. It does not have this operator because this scenario does not occur for the `PAINTING_THE_WALL` problem. Additionally, the `shaded-area` and `base` skills do essentially the same thing: they iterate over items in the visual hierarchy and do some mathematical operation with these items to find a total value.

In PRIMs, skills are meant to represent the largest unit of procedural knowledge that can be reused between tasks. As such, they rely on a delicate balance between being specific enough to solve a current task but general enough to also apply to other tasks. From this viewpoint, the skills defined in version 1 of Model 0 are too specific. This version of the model can certainly be used to solve all three questions of the `PAINTING_THE_WALL` problem. It cannot do much else. Hence, a second version of Model 0 was created which aimed to be overall more general, in the hopes of it becoming a building block for Model 8. The full code of version 1 of Model 0 can be found in **Appendix C.1**.

5.2.2 Version 2

Version 2 of Model 0 was created with two purposes. Firstly, it was meant to be a more general model. Secondly, it was meant to serve as a building block for Model 8. At times, these two purposes would overlap, and making the model more general would also make it a better building block for Model 8.

What is meant by a more general model is a model that is able to solve for a wider breath of tasks, without changing the purpose of the model. To be more precise, Version 1 of Model 0 could only solve the full `PAINTING_THE_WALL` problem. It stands to reason, however, that the model should be able to solve for its intermediary steps as well. If asked to calculate only the area of the door, Model 0 should be able to do so, since it has the `area` skill. Despite of this, version 1 of Model 0 cannot do this task.

The reason for this lies in the `read-task` skill. In version 1 of Model 0, the `read-task` skill reads its task off the $V4$ slot of the screen chunk in the visual input. It assumes this task is bound to the top shape in the visual hierarchy. If the goal is changed to `area`, the model would thus try to calculate the area of the wall, rather than that of the door.

An easy solution to this problem is to add an optional value to a new slot ($V5$) in the screen chunk. This slot could, in theory, be used to specify a shape or segment within the visual hierarchy that the task is bound to. Of course, adding this slot means nothing if the model does not know how to process said slot. For version 2 of Model 0, it is therefore necessary to adjust the `read-task` skill to allow for more general task assignments.

The `read-task` skill is given two additional operators in version 2 of Model 0. They can be found in Listing 8. The original `read` operator shown in Listing 6 is renamed to `read-simple-goal` for this second iteration of the `read-task` skill.

Listing 8: The two additional operators of the read-task skill, version 2.

```
// A complex goal has a shape specified as focus (e.g. area rect5 asks to
    find the area of rect5 in the visicon)
operator read-complex-goal {
   G1 = read-task
   V1 = screen
   V4 <> side
   V5 <> nil
==>
   search-visual -> G1  // Before we get to the top goal, we first need
      to find the specified shape
   V4 -> G2 // The top goal is stored in G2
   V5 -> WM1   // The specified shape gets placed in WM1 so that we can
      iterate until it's found
   >>V3 // shift focus to the first shape
}

// If the goal is to find a side, V5 will specify which side we are
   interested in
operator read-side {
   G1 = read-task
   V1 = screen
   V4 = side
   V5 <> nil
==>
   V4 -> G1
   V5 -> *side // save the specified side as a variable
   >>V3 // shift focus to the first shape
}
```

The new operators of the `read-task` skill allow the model to solve two additional task types. It can either solve for the `area` of a specified shape, or it can solve for the `side` of the top shape. The `side` task must be further specified to refer to either a base or a height. These two additional task types allow the model to complete all the intermediary steps of the `PAINTING_THE_WALL` problem (namely, calculating the area of the door or calculating the base of the wall), when prompted.

What immediately stands out about the new operators is that they are visibly more complex than the original `read` operator in Listing 6. They have more conditions (lines before the arrow), and some new results (lines after the arrow) that have not been used prior.

In terms of conditions, a goal slot is now specified to streamline how the skills are setup (recall that the `shaded-area` skill also had a goal slot specification) and to further ensure that the `read-task` skill is executed anytime the student returns to the screen chunk. Additionally, the conditions now check the new $V5$ slot of the the screen chunk, which will hold either a target shape or a base or height specification. The existence of the $V5$ slot is what sets the conditions of the `read-complex-goal` and `read-side` operators apart from the `read-simple-goal` operator.

In the new operator results, the read-complex-goal operator shows the use of two goal slots, rather than a single one. The $G1$ slot holds the immediate goal, while the $G2$ slot holds the overarching goal. When a task is read that specifies a certain target (but not a side), then the immediate goal becomes to find that target. The target can be found with a new search-visual skill.

The search-visual skill consists of two operators: one for when the desired target is found, and one for when a visual item is not the target. When the target is found, the overarching goal that was temporarily placed in the $G2$ slot is moved to the $G1$ slot. The $G2$ slot is then cleared, and the model continues with its overarching goal. The use of two goal slots is unique to version 2 of Model 0, but it is only used in this instance: to process more complex goals.

The read-side operator shows another mechanic unique to version 2 of Model 0. In its results, the content of the $V5$ slot (which will specify whether the task is to find a height or a base) is passed on to a variable called *side. It is clear from the asterisk that this variable is different from constants like screen and side, which the conditions check for. In fact, the asterisk denotes that this variable is a *binding*. Bindings are, as their name imply, bound to a certain value. In this case the *side binding is bound to the contents of the $V5$ slot.

The use of bindings allows for the creation of more general models. The *side binding makes it possible to use one skill (called side) for finding either a base or a height. Within the skill, different operators can be defined for dealing with the different values of the *side variable. This obviously allows for more versatile skills, but it is fair to wonder if the use of bindings is cognitively sound.

To answer that question, the side skill is analysed in more detail. The side skill is very simple in version 2 of Model 0, because the only way to find a side in this model is through segments. There is no evidence in the PAINTING_THE_WALL problem alone that students could also use e.g. the area to find a side. Thus, the only proven way students can determine sides (if they can solve Node 0), is by adding segments together. As such, the side skill has but one operator (side-by-segments) which changes the goal to segments.

The segments skill in version 2 of Model 0 corresponds to the base skill of version 1. Unlike its version 1 iteration, however, the segments skill can be used to find either a base or a height. Its only requirement is that the relevant side can be found by adding segments together. Listing 9 shows the two operators of the segments skill that encapsulate the difference between finding a height and finding a base. It is a matter of looking in a different visual slot ($V5$ for a base, $V6$ for a height). Beyond that, the operators are the same, and the segments skill goes about finding a base in the exact same way as it goes about finding a height.

If the *side binding had not existed, each operator in the segments skill would have needed two variants: one for when the task was to find a height, and one for when the task was to find a base. Alternatively, it would be possible to create two separate skills, one for finding a base and one for finding a height. These two scenarios are identical in that they make little sense. The process for finding a base is the same as the process for finding a height. The only difference, as Listing 9 indicates, is where a student needs to look to read the height or base off their visual input. Given this small difference, it makes no sense to have separate operators or even separate skills for this identical process. The *side binding (and all other bindings used in version 2) allows the operators to be used in a more versatile way, without them being bound to very specific constants.

Listing 9: The operators that allow the segments skill to differentiate between finding a base and finding a height.

```
// If we're trying to find a base, read the width of a segment/shape
operator read-width {
   G1 = segments
   *side = base
   *item = none
   *done-iterate = no
   V5 <> nil
==>
   V5 -> *item
}

// If we're trying to find a height, read the height of a segment/shape
operator read-height {
   G1 = segments
   *side = height
   *item = none
   *done-iterate = no
   V6 <> nil
==>
   V6 -> *item
}
```

Together, bindings and multiple goal slots allow version 2 of Model 0 to be much more generic. The repetition of steps seen in version 1 of the `shaded-area` and `base` skills is solved by creating a more generic `iterate-over` skill that can only work because of the use of bindings. Overall, version 2 of Model 0 allows the model to do more, not by creating additional skills, but by breaking down the skills there are into their specific and generic components.

Ultimately, such a generic version of the model is more cognitively sound precisely because it relies on the reuse of (component) skills between tasks. To draw a comparison, say there was a table with oranges on one side and apples on the other. Version 1 of Model 0 can be thought of as a model created to count oranges. It could count the oranges, but it could not do anything with the apples, even though it knows how to count. It is so specific it can only work in the exact conditions for which it was created. Version 2 of the model is then a version that can count irrespective of the item it needs to count. Version 2 of the model is still made to solve the `PAINTING_THE_WALL` problem (which encompasses the full content of Node 0). The difference is, it can also solve problems that rely on the same skills it needs for the `PAINTING_THE_WALL` problem (like calculating a rectangle area, adding two bases, et cetera). It is this quality to apply the same skills to different scenarios that make the second version of the model better aligned with human cognition.

While thus being an improvement over version 1 in terms of generality, there is another requirement for version 2 of Model 0. It also needs to serve as a building block for Model 8. The driving power behind the knowledge graph approach is that it identifies the skills underlying a data set. The final knowledge graph in the *Knowledge Graph* section is very clear in its assessment that Node 8 differs from Node 0 by only one skill. If this assessment is adhered to strictly, then it means the model of Node 0 should be written in such a way that the Model of Node 8 can solve all its problems by reusing the skills defined in Model 0, in combination with its one additional skill.

In many ways, making version 2 of Model 0 more generic already helped in improving its function as a building block for Model 8. The `search-visual` and `side` skills are skills that were added to Model 0 in part because Model 8 would need them. Similarly, the additional operators in the `read-task` skill were added for that same reason. The addition of the skills and operators was justified, because it could be shown that Model 0 could also already do these things, as evidenced by the `PAINTING_THE_WALL` problem. Furthermore, by virtue of Node 0 representing the prior knowledge of a student coming into the data set, it was possible to fine-tune the necessary skill set without worrying about restrictions regarding the number of skills.

Following this line of thinking, three more skills were added to the model. These skills were addition, multiplication, and subtraction; all skills which the `PAINTING_THE_WALL` problem indicates students possess if they can correctly solve said problem. The addition, multiplication, and subtraction skills could be thought of as higher levels of the math skill. In functionality, they are all the same in that they set the `fact-type` binding to their associated values (e.g. the `multiplication` skill sets the `fact-type` to `multiply-fact`). After setting the `fact-type`, the rest of the mathematical operation is done by the `math` skill (which remains virtually the same as its version 1 variant). As with version 1, all math is still simplified as a remembering process.

Separating the different mathematical operations from the higher-level `shaded-area`, `area`, and `base` skills make sense in the same way it makes sense that the `iterate-over` skill is not tied to the `shaded-area` and `base` skills. The mathematical operations should be independent skills that are not tied to specific instances of use. While separating them was thus done because it would prove necessary for Model 8, it again aligns with making the model more generic and through that, more human.

Listing 10: The full content of the missing-area operator (a part of the shaded-area skill).

```
// If an area is missing, find that first
operator missing-area {
   G1 = shaded-area
   V1 = rectangle
    V6 = nil
==>
   area -> G1
   nil -> WM0
}
```

The final change that was made to Model 0 to establish it as a building block for Model 8 was a change to its visual input. Node 8 contains trapezoids and triangles, but Model 0 was designed to function only with rectangles. Listing 10 shows that the model uses the rectangle constant in its conditions to determine whether it is dealing with a missing area or whether the current item does not have an area at all (like a screen or a base). To make sure the model could also deal with other shapes, the visual input was changed according to Listing 11. Items were categorised as either shapes or segments.

Listing 11: An updated version of the visual input used for version 2 of Model 0 and for Model 8.

```
// Define the visual with a graph representation. V2 holds same-level
   items. V3 holds lower-level items
// V4 is a further specification of the V1 type. V5 is width, V6 is
   height. V7 is area for shapes
define visual {
   // Screen for question 1
   (screen1 screen nil rect1 shaded-area)
   (rect1 shape nil base1 rectangle nil 22.5 nil)
   (base1 segment rect2 nil base 3 nil)
   (rect2 shape base2 nil rectangle 7 17.5 nil)
   (base2 segment nil nil base 10 nil)
```

After this update to the visual input, skills like the `shaded-area` skill check whether an item is a shape, rather than specifically a rectangle. Although Model 0 only has rectangle shapes and base segments, the adaptation of a two-type specification for the items makes a wider variety of shape and segment definitions possible.

With that final update to version 2 of Model 0, Model 0 is complete. This is a version of Model 0 that is both generic and capable of serving as a building block for Model 8. Its full code can be found in **Appendix C.2**. As a visual aid, Figure 23 shows a schematic (partial) overview of the model's operators and skills, as discussed in this section.

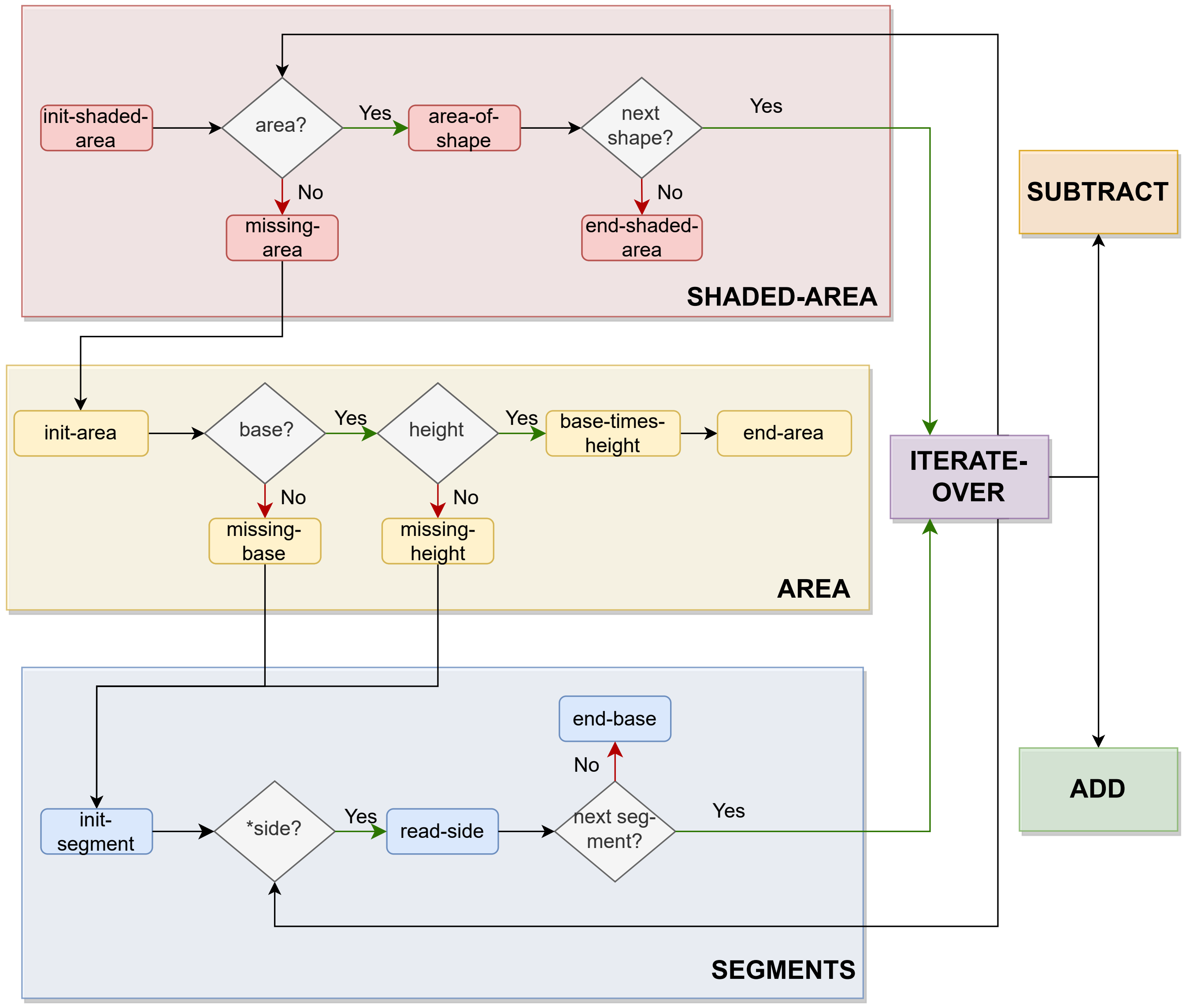
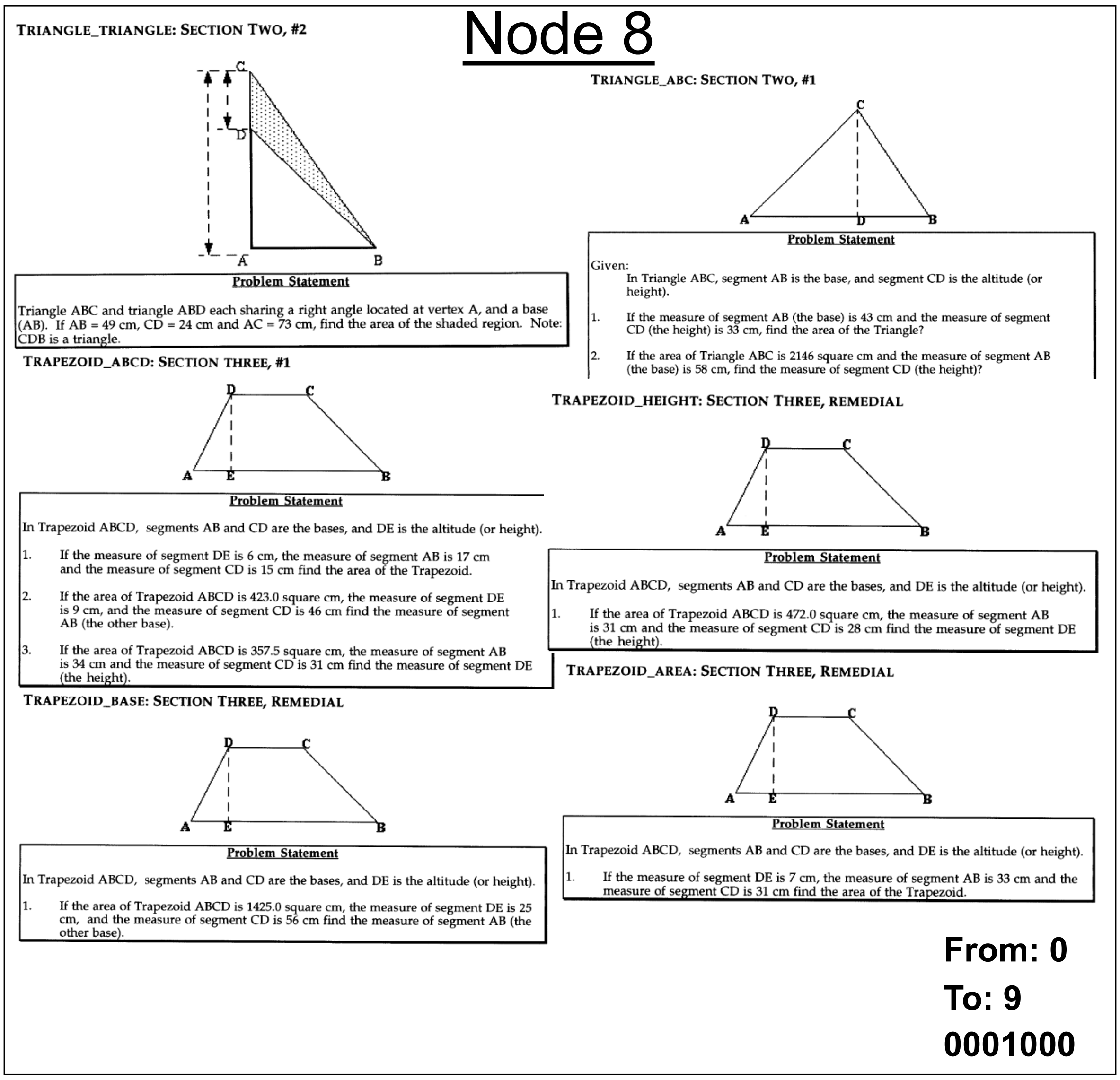


Figure 23: A schematic overview of the workings of version 2 of Model 0. Rounded rectangles represent operators, square rectangles represent skills, and diamonds represent variable checks (is variable *X* available, yes or no?). Starred variables represent bindings, which can be set to specific values to check for.

5.3 Model 8

Similar to Model 0, Model 8 was built to represent the cognition behind the problems in Node 8. For simplicity's sake, these problems are repeated in Figure 24. It is visible that Node 8 contains six problems. Of these six problems, four are trapezoid problems. Specifically, they are the `TRAPEZOID_ABCD`, `TRAPEZOID_HEIGHT`, `TRAPEZOID_BASE`, and `TRAPEZOID_AREA` problems. These problems should be familiar, since they were also discussed in the *Knowledge Graph* Section.

The two new problems in Node 8 are `TRIANGLE_TRIANGLE` and `TRIANGLE_ABC` (at the top of Figure 24). The former problem asks students to find the area of the shaded region; it consists of a triangle shape within a larger triangle. For the latter problem, only the first two questions are included in Node 8. These questions ask for the area of the triangle, given its base and height, and for the height, given the base and the area.



Node 8

TRIANGLE_TRIANGLE: SECTION TWO, #2

Problem Statement

Triangle ABC and triangle ABD each sharing a right angle located at vertex A, and a base (AB). If AB = 49 cm, CD = 24 cm and AC = 73 cm, find the area of the shaded region. Note: CDB is a triangle.

TRIANGLE_ABC: SECTION TWO, #1

Problem Statement

Given:
In Triangle ABC, segment AB is the base, and segment CD is the altitude (or height).

1. If the measure of segment AB (the base) is 43 cm and the measure of segment CD (the height) is 33 cm, find the area of the Triangle?
2. If the area of Triangle ABC is 2146 square cm and the measure of segment AB (the base) is 58 cm, find the measure of segment CD (the height)?

TRAPEZOID_ABCD: SECTION THREE, #1

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the measure of segment DE is 6 cm, the measure of segment AB is 17 cm and the measure of segment CD is 15 cm find the area of the Trapezoid.
2. If the area of Trapezoid ABCD is 423.0 square cm, the measure of segment DE is 9 cm, and the measure of segment CD is 46 cm find the measure of segment AB (the other base).
3. If the area of Trapezoid ABCD is 357.5 square cm, the measure of segment AB is 34 cm and the measure of segment CD is 31 cm find the measure of segment DE (the height).

TRAPEZOID_HEIGHT: SECTION THREE, REMEDIAL

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the area of Trapezoid ABCD is 472.0 square cm, the measure of segment AB is 31 cm and the measure of segment CD is 28 cm find the measure of segment DE (the height).

TRAPEZOID_BASE: SECTION THREE, REMEDIAL

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the area of Trapezoid ABCD is 1425.0 square cm, the measure of segment DE is 25 cm, and the measure of segment CD is 56 cm find the measure of segment AB (the other base).

TRAPEZOID_AREA: SECTION THREE, REMEDIAL

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the measure of segment DE is 7 cm, the measure of segment AB is 33 cm and the measure of segment CD is 31 cm find the area of the Trapezoid.

From: 0

To: 9

0001000

Figure 24: The problems in Node 8 (repetition of Figure 17).

The goal of Model 8 is to solve all six of these problems. To accomplish this, the model will be given one additional skill compared to Model 0. The rest of the skills it may need to solve these problems, it will have to reuse from Model 0. This setup ensures Model 8 adheres to the data analysis and the conclusions from the knowledge graph.

Although it is simple to say Model 8 will be given only one additional skill, it is much more difficult to determine what that skill should be. By analyzing the skills Model 0 already possesses, Section 5.3.1 will try to determine what the skill is that underlies Node 8. Having determined skill `0001000`, Section 5.3.2 then uses this skill to build Model 8.

5.3.1 Finding Skill 8

To determine the skill Model 8 requires to solve the problems of Node 8, it is important to map the capabilities Model 0 already possesses. Although Model 0 does not know how to solve for triangle or trapezoid shapes, there are many aspects of the problems in Node 8 that Model 0 can already tackle.

For example, the `TRIANGLE_TRIANGLE` problem asks for the area of a shaded region. This is precisely what Model 0 needs to do for the `PAINTING_THE_WALL` problem. Thus, while the model does not know how to calculate the area of a triangle, it does know how to subtract two areas from each other to find a shaded region.

In fact, the `shaded-area` skill is not the only skill Model 0 can reuse to solve the problems of Node 8. Specifically, it can also apply the `segments` skill to all questions that involve trapezoids. Trapezoids are characterised by having two bases: a long base (AB) and a short base (DC). Even if both bases are given, to calculate anything for a trapezoid (a height or an area), the bases will have to be added first to get the trapezoid's total base. Through the `segments` (and `add`) skill, Model 0 is already capable of performing this addition.

From these observations, it is clear Model 0 has some skills that are both useful and necessary for solving the problems in Node 8. Unfortunately, Model 0's biggest shortcoming is that it only knows how to do calculations for rectangle shapes. The primary reason for this is that it has only one operator for calculating the area of a shape: the `base-times-height` operator. This operator works specifically by multiplying a shape's base with its height. Through that multiplication, it determines the area of the shape. For rectangles, that is all there is to it. For triangle and trapezoids problems, this multiplication of the base times the height is insufficient.

Although insufficient, it is not wrong. That is to say, to calculate the area of a triangle or a trapezoid, the base and height must still be multiplied with each other. For triangles, the area that comes out of this multiplication must next be divided by two. For trapezoids, the same applies, but its base is made up of two bases that must be added together first (before the total base can be multiplied with the height). In simple terms, calculating the area of a triangle and a trapezoid is simply calculating the area of a rectangle with some additional steps.

For the purpose of identifying skill `0001000`, this is a good discovery. If the process of identifying the area were entirely different for the different shapes, then it would stand to reason that each shape thus has a unique skill for calculating its area. While this is theoretically still possible, the data analysis done indicates this is not the case, since it proposes there is but a singular skill that differentiates Node

0 from Node 8. If this data analysis is adhered to, then it would not be correct to add two new skills to Model 8: one for calculating the area of a triangle, and one for calculating the area of a trapezoid.

Following this reasoning, it is good that there are steps that clearly repeat when calculating the area of rectangles, triangles, and trapezoids. Since there is such overlap, these steps could be considered as new operators for the area skill, rather than separate skills on their own. In this case, the skill is the same `area` skill, but its new operators teach the model how to apply that skill in new circumstances.

By updating the `area` skill with new operators, the model should be able to calculate the areas of triangles and trapezoids. However, there remains one obstacle for the model to be able to succeed in this calculation: Model 0 does not know how to do division! Recall that for triangles and trapezoids, the intermediate area must be divided by two for the final area. Since the `PAINTING_THE_WALL` problem does not require the use of division, Model 0 has no skill for this.

The model does have skills for multiplication, addition, and subtraction. These are mathematical operations that are clearly necessary for the `PAINTING_THE_WALL` problem, and thus Model 0 was given these skills to be able to solve the problems in Node 0. Since division is now clearly necessary to solve the problems of Node 8, Model 8 will therefore need a division skill.

Such a division skill will not only allow Model 8 to calculate the area of trapezoids and triangles, it will also allow the model to solve the other remaining questions of Node 8. These other questions ask students to solve for a side (a height or a base), given an area and the other relevant side. Clearly, if the model knows how to do division, it can also solve these types of questions.

To conclude, if the skills from Model 0 are accessible to Model 8 as well, then the only truly new skill Model 8 needs to solve the problems of Node 8 is the division skill. Although some skills from Model 0 will need to be adjusted to teach Model 8 how to handle new circumstances, these adjustments can be implemented through additional operators. In this way, the data analysis is adhered to and Model 0 and 8 only differ in one skill.

Listing 12: The divide skill added to Model 8.

```
// This skill has the knowledge of how to do division
define goal divide {
   operator divide-action {
      WM1 <> nil
      WM2 <> V0
      WM3 = nil
      *action = divide
      *fact-type = none
   ==>
      divide-fact -> *fact-type
   }
}
```

5.3.2 Creating the Model

Listing 12 shows the `divide` skill that is added to Model 8. Like the `multiply`, `add`, and `subtract` skills, this skill itself does not do much. It sets the `*fact-type` binding to `divide-fact`, so that the `math` skill knows what type of fact it is looking for in its memory (remember that math is simplified in this model as remembering the answer of a mathematical operation).

While the skill itself is thus not very involved, the more important changes to Model 8 lie in how this new skill is applied. The way the skill integrates with the `math`, `area`, and `side` skills is, in fact, where Model 8 differs the most from Model 0.

To begin with, Listing 13 shows how the `divide` skill was integrated with the `area` skill to calculate the area of triangles. The first operator `area-triangle` is designed to trigger after the `base-times-height` operator. This operator is not in Model 8, but the model can reuse it from Model 0.

Listing 13: The operators added to the area skill for triangles.

```
// If we're looking a triangle, divide the previously found area by 2
operator area-triangle(activation=10.0) {
   G1 = area
   WM3 <> nil
   V4 = triangle
   *action = multiply
==>
   WM3 -> WM1
   *two -> WM2
   nil -> WM3
   divide -> *action
}

// If we have the answer in WM3 and we did the area-triangle operator,
   write the answer
operator end-area-triangle(activation=10.0) {
   G1 = area
   WM3 <> nil
   V4 = triangle
   *action = divide
==>
   G1 -> *task
   WM3 -> *answer
   write -> G1
   none -> *action    // Reset the action since we are finished with the
      goal
}
```

The second operator `end-area-triangle` tells the model when it is finished with calculating the area of a triangle. This triggers after the `area-triangle` operator. Notice how both these operators have an activation specified. Activation has been mentioned twice prior in this section, once in reference to the spreading activation from the goal, and once in reference to the base-level activation.

Generally, activation determines whether the model can retrieve its chunks. In PRIMs, operators are stored as chunks in the model's declarative memory in the same way facts are. Following this design, activation plays an important role in which operator the model decides to execute at any given point in time. The conditions of the operators are meant to ensure the model executes its operators in an order that leads to the successful completion of its task. However, in a situation where the conditions of multiple operators are met, it is the activation of each operator that decides which operator will be executed. To be more precise, the model will execute the operator whose conditions are met and whose activation is the highest of all possible operators.

Activation can be increased with use. The more a chunk is recalled, the higher its activation will be (given that the `retrieval-reinforces` parameter is set to true; which it is for Model 8). This causes problems when new operators are introduced, as is the case for Model 8, because old operators are likely to have higher activation (if they have been used in the past). In other words, the operators Model 8 reuses from Model 0 are likely to have higher activation than the new operators that are being introduced for Model 8. Translating this to practice, what can happen is that the model does not calculate the area of a triangle correctly, because it decides to use the old operators it has from Model 0 instead. These operators (in version 2 of Model 0) do not mention they are specific to rectangles, because that was not necessary for Model 0. This means that the operators conditions are met for any shape, and if their activation is higher, the model will choose to execute these operators rather than its new operators. The result of which is that the model does not correctly calculate the area of a triangle (or a trapezoid).

The solution to this problem is to have the new operators start off with a higher activation (`activation = 10.0`). This is, of course, a very crude solution, because the model is, in a sense, forced to use the new operators. If approached more organically, the model would instead be allowed to make its mistakes and learn from them. It would see that each time it used the old operators, it would not succeed in its task, and over time (through another PRIMs property known as *associative learning*), it would start to use its new operators in favour of the old ones.

Although this approach would show an interesting aspect of cognition (and learning specifically), the models described in this section do not focus on the learning process itself. They simulate the brain when the desired skills are already mastered. This is also noticeable in how the models are not made to make/simulate mistakes. From the data available, it would not be possible to include this aspect of cognition, because Dataset 1 does not include the actual answers students gave; it is not known what types of mistakes they made, and as such, mistakes cannot be simulated by the models.

Without incorporating mistakes, the higher activation allows Model 8 to simulate mastery of the updated versions of Model 0's skills. Consequently, all new operators introduced in Model 8 for existing skills will have their activation set to 10, unless these operators do not need to compete with the old operators. The `side` skill, for example, does not have many operators in Model 0. Since Model 0 could not divide, the model could only find sides through the `segments` skill. As a condition, the `side-by-segments` operator postulates that there is no area available. The new operators added to Model 8, which work precisely when an area is available, do not need to compete with the `segments-by-sides` operator. These operators then do not need a higher base activation.

Before moving on to the `side` skill, there are also the trapezoid operators to consider for the `area` skill. Notably, there is a trapezoid variant for both the `area-triangle` and `end-area-triangle` operators. The `area-trapezoid` and `end-area-trapezoid` operators are identical to their triangle counterparts, save for their conditions. These ask for a trapezoid in slot $V4$, rather than a triangle. Although the operators are otherwise identical, each shape has its own set of operators because they depict different circumstances, and the model must know what to do in each unique circumstance.

To further highlight this, the `add-trapezoid-bases` and `base-of-trapezoid` operators (depicted in Listing 14) are unique to trapezoid shapes. The operators tell the model how to calculate the full base of a trapezoid (through the addition of its short and long base). They are meant to be executed before the `base-times-height` operator, since the latter only triggers if a shape's base and height are known. The circumstances surrounding these operators are very specific, and their conditions reflect as much.

Listing 14: The operators that tell the model how to determine the full base of a trapezoid shape.

```
// Since trapezoids don't have a single base, get the total base by
   adding its base1 and base2
operator add-trapezoid-bases(activation=10.0) {
   G1 = area
   V4 = trapezoid
   WM1 = nil
==>
   G1 -> G2 // Save the original goal so it can be returned to
   segments -> G1
   base -> *side  // Set the focus to base
   add -> *action
   nil -> WM0
   >>V3  // Move down to the segments that make up the trapezoid base
}

// This skill triggers after the trapezoid bases have been added
operator base-of-trapezoid(activation=10.0) {
   G1 = area
   WM1 <> nil
   *action = add
    V<<  // Returning from segments, look at the trapezoid again rather
       than the last base
   V4 = trapezoid
==>
   none -> *action
}
```

To go into further detail, the operators are designed specifically for a certain visual input. Trapezoids are represented in the visual input according to Listing 15. Their base is set to *nil* by default, and instead, they have two base segments one level down in their visual hierarchy. The `add-trapezoid-bases` operator in Listing 14 shifts the visual focus down to these segments so that the bases can be added together through the `segments` and `add` skills. Unlike other intermediate answers, the worksheet of the cognitive tutor does not give students a space to write down the trapezoid's full base. This means the students (and therefore the cognitive model) will have to remember this intermediate answer in-between calculations (in the model, it is stored in the working memory slot $WM1$). After the bases have been added together, the `base-of-trapezoid` operators shifts the focus back to the trapezoid chunk so that the rest of the area calculation can be done.

Listing 15: The definition of a trapezoid in the model's visual input.

```
(trapezoid1 shape nil base4 trapezoid nil 7 nil)
(base4 segment base5 nil base 31 nil)
(base5 segment nil nil base 33 nil)
```

Although the $V7$ slot of the trapezoid chunk can thus be updated to incorporate the calculated area, the $V4$ slot, which reflects the base, will always be *nil* for trapezoids. As a result, the `missing-base` operator from Model 0 must be adjusted to reflect this (since it normally triggers precisely when slot $V4$ is equal to *nil*). Among its operators, Model 8 is given a new version of the `missing-base` operator with updated conditions. These conditions specify that the operator should only execute if the shape is not a trapezoid.

Similarly, the `end-segments` operator of the `segments` skill must be updated to reflect that the answer is not written down in the case of trapezoid bases. As it is not possible from the segment chunks to know what the top-level shape is, a $G2$ slot is utilised to distinguish between when and when not to write down the `segments` answer. The `trapezoid-bases` operator saves the overarching `area` goal to the $G2$ slot, and a new `end-segments-alt` operator is triggered in the cases where the $G2$ slot is not *nil*. If there is some overarching goal for the `segments` skill to return to in Model 8, it will do so over writing down its intermediate answer. Like the new `area` operators, the `end-segments-alt` (and updated `end-segments`) operators are given a base activation of 10.

In addition to the `end-segments-alt` and updated `end-segments` operators, the `segments` skill in Model 8 is given two additional operators: `skip-height` and `skip-base`. These operators are not given a higher base activation, because they are added to the `segments` skill primarily so that Model 8 knows how to deal with a wide variety of circumstances. In the cases where the model is looking for a specific side, but the current item in its visual focus is missing information on this side, it will skip the item as a rule of thumb. This ensures the model does not get stuck (something that occurs when there is no operator whose conditions are being met).

To further ensure the model does not get stuck, the `side` skill from Model 0 must be updated for Model 8 as well. In this case, the added operators do not need a higher activation, because they will not be competing against older operators from Model 0's version of the `side` skill. In total, twelve new operators are added to the `side` skill for Model 8. These operators largely mirror those of the `area` skill. There are three general operators that are used by all shapes. These operators give instructions on how to read the area and the other side, which should both be given to calculate the desired side. The `area-by-other-side` operator next tells the model how to get the desired side by dividing the area with the given side.

On top of these three general operators, each shape has its own ending operator. This operator lets the model know when it is done calculating its desired side for the shape of interest.

Contrasting the other two shapes, the trapezoid shape actually has two ending operators: one for the height and one for the base. The trapezoid shape requires two operators, because finding the side of a trapezoid is a much more involved process. Where the side of a rectangle can be found through the three general operators, and the side of a triangle needs but one extra operator (`side-triangle`), the trapezoid shape requires four extra operators for calculating one of its sides. All of these extra operators deal with the fact that a trapezoid shape has two bases. They instruct the model on how to deal with this trapezoid characteristic, for example by having the model subtract its found total base by the given other base.

It is clear from this description that the `side` skill is where Model 8 and Model 0 differ the most, with the `area` skill following closely. The updates made to the `segments` skill are the least impactful.

The description of Model 8's new operators given in this section does not only show how Model 8 and Model 0 differ, it also shows how they are alike. Skills such as `search-visual` and `iterate-over` are left unaltered for Model 8. The model will use these skills as they are from Model 0 if it requires them. While the model includes new operators for three of Model 0's pre-existing skills, there is but one skill added to Model 8 that is entirely new: the `division` skill. As with the previous models, the full code for Model 8 can be found in **Appendix C** (specifically **C.3**) and Figure 25 provides a schematic overview of the Model 8 operators discussed in this section.

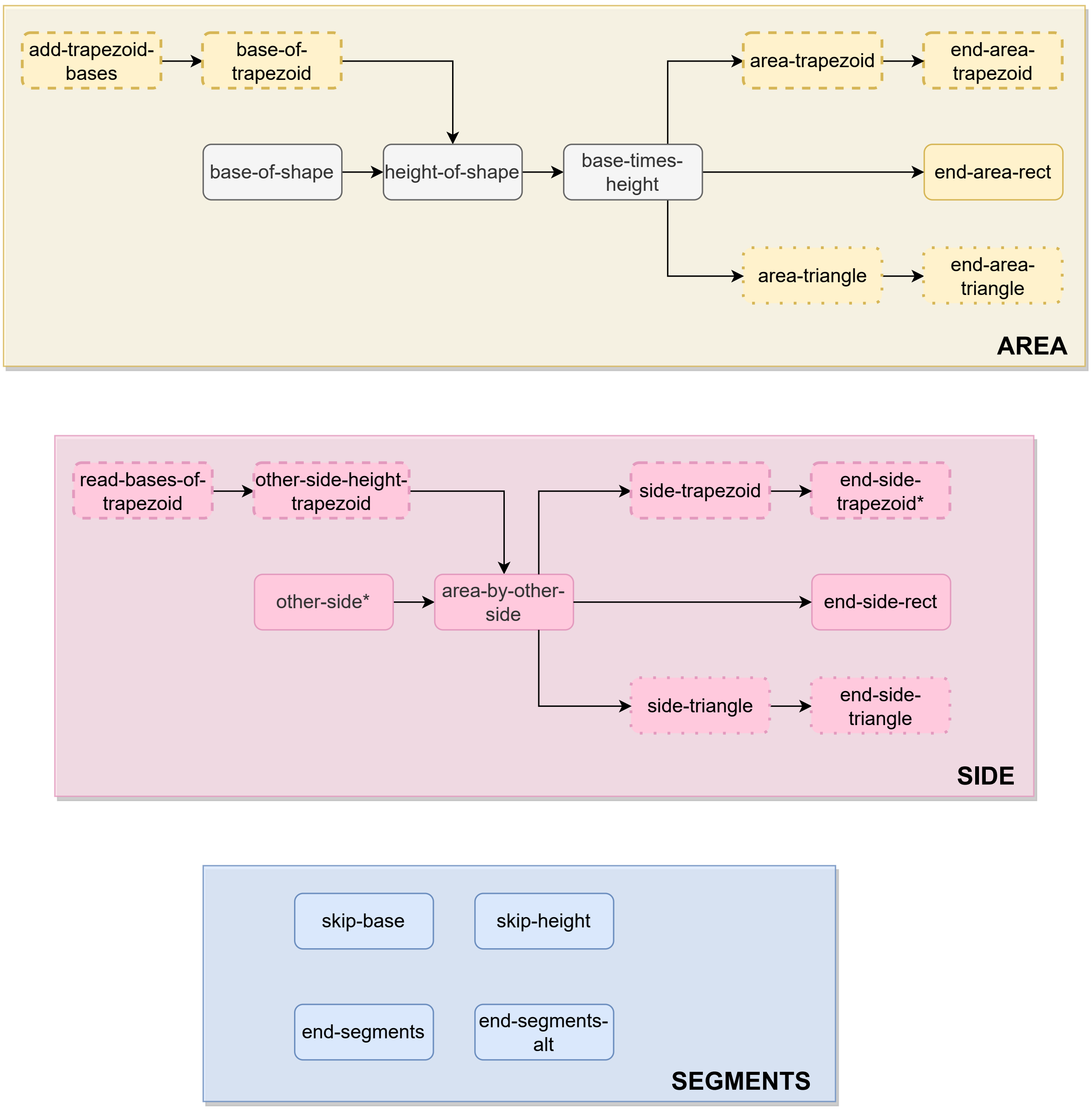


Figure 25: A schematic overview of the new operators in Model 8 (coloured; white operators are those Model 8 can reuse from Model 0). Rounded rectangles represent operators. The border of each operator is shape-dependent. Rectangles have an uninterrupted border, trapezoids a dashed border, and triangles a dotted border. The starred operators visible are binding-specific, meaning there is a version available of this operator for each binding value.

5.4 Model Performance

The previous two sections describe how two cognitive models were built on the basis of the knowledge graph that came out of the data analysis. The models, Model 0 and Model 8 respectively, were built to solve the problems of Node 0 and Node 8. The fact the models could be built in the first place already says something about the validity of the skills the knowledge graph identified. After all, if the results from the knowledge graph had been unintelligible, it would not have been possible to transform these results into functional cognitive models.

In this case, functionality is the key word. While being able to built the models in the first place is a good sign, the skills identified by the knowledge graph cannot be accepted unless they can result in the successful completion of their desired tasks (i.e. all problems in Node 0 and Node 8).

To test whether the models could successfully complete their desired tasks, multiple model runs are performed. Model 0 is checked for all three questions of the `PAINTING_THE_WALL` problem. For computational reasons, Model 8 is checked for all Node 8 problems except `TRIANGLE_TRIANGLE` and `TRAPEZOID_ABCD`. It is assumed that the model's performance on the other problems is sufficient to determine its success for the whole task (which is all of Node 8).

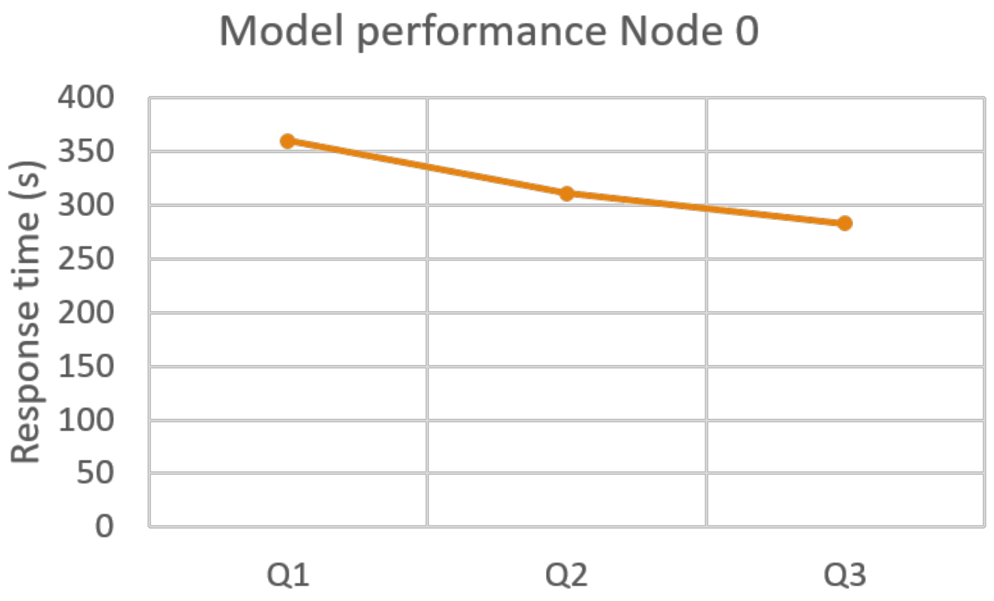


Figure 26: The performance of Model 0 in terms of response time. The three questions on the x-axis represent the three questions of the `PAINTING_THE_WALL` problem.

Figure 26 shows the model performance of Model 0. The model performance is expressed in response time, which represents how long it took the model to answer each of the three questions of the `PAINTING_THE_WALL` problem. Firstly, it is good to note that the model indeed succeeds at answering all three questions. It can be concluded that the model succeeded at its task of solving all problems in Node 0.

Another interesting observation that can be made on the basis of Figure 26 is that, although learning was not a priority of these models, it does take place. The model performance improves with each subsequent question. Under the hood, this performance increase can be explained through a PRIMs property called *production compilation*.

When a model uses its operators for the first time, it is not efficient at them yet. Its checks each condition separately and runs one result at a time. As the model executes its operators, it learns how to check multiple conditions at once and how to run results in batches. Together, the conditions and results make up the *production rules* of each operator. Through the compilation of new, more efficient production rules, the model becomes faster at using its operators.

Since Model 0 is asked to solve each of the `PAINTING_THE_WALL` questions in order, it has a chance to learn better production rules over time. Subsequently, the model is more proficient with its operators (and thereby its skills) by the time it gets to question 2, and again for question 3. Through practice, the model becomes more efficient at answering the shaded-region questions, and this is reflected in its faster response time.

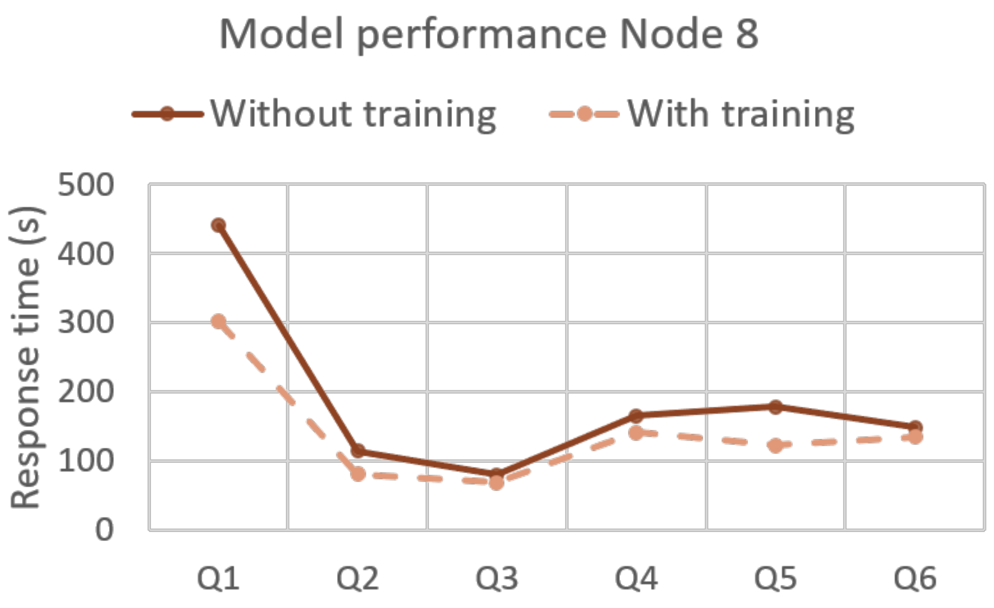


Figure 27: The performance of Model 8 in terms of response time. Question 1 on the x-axis is a repetition of question 3 of the `PAINTING_THE_WALL` problem. Question 2 and 3 represent the first two questions of the `TRIANGLE_ABC` problem. The remaining questions represent the `TRAPEZOID_HEIGHT`, `TRAPEZOID_AREA`, and `TRAPEZOID_BASE` problems (in that order).

Given this effect of practice that is observed in Model 0, the performance of Model 8 makes a distinction between the performance with- and without training. In the without training condition, Model 8 can access the Model 0 skills, but Model 0 itself is never run. In the training condition, Model 0 is run before Model 8, so that Model 8 can inherit the more efficient production rules that Model 0 has compiled from practice.

Figure 27 shows the performance of Model 8 for six questions. These questions span the `TRIANGLE_ABC`, `TRAPEZOID_HEIGHT`, `TRAPEZOID_AREA`, and `TRAPEZOID_BASE` problems. Additionally, the first question (Q1) is a repetition of question 3 of the `PAINTING_THE_WALL` problem. Since Model 8 can reuse the skills from Model 0, it should also be able to solve this question.

Indeed, Figure 27 confirms Model 8 cannot only solve problems from Node 8, it can also solve the problems from Node 0. In other words, Model 8 encompasses all the skills a student should have if they can successfully solve the Node 8 problems. While the model performance in Figure 27 does not include the `TRIANGLE_TRIANGLE` and `TRAPEZOID_ABCD` problems, it stands to reason that the model would be successful in answering these questions as well, since its success with Q1 indicates it can solve for shaded regions and its success with the other trapezoid problems indicates it could equally solve the `TRAPEZOID_ABCD` problem.

Thus, it can be concluded that Model 8, like Model 0, is successful in completing its task. However, the dark, without-training line in Figure 27 shows that Model 8 is, initially, less efficient at solving its question than Model 0. Model 0 needs less than 300 seconds to solve question 3 of the `PAINTING_THE_WALL` problem, while Model 8 needs over 400 seconds to do the same.

This is not a strange finding. Model 8 has an additional 22 operators. In practice, this means it will need more time to decide which operator to choose at any given time compared to Model 0. If Model 0 has had a chance to do a full run before Model 8 is executed, however, then Model 8 gets a head-start from Model 0's increased expertise. It knows better which operators to execute at which time to increase the chances of it successfully completing its task. This expertise results in the dashed, with-training line. With training (that is to say: if Model 0 has had a full run before Model 8 is executed), Model 8 reaches about the same performance as Model 0 near the end of its run.

An additionally interesting observation from Figure 27 is the striking difference in performance between the different questions. As with the difference between with- or without training, the difference in performance between questions is also not strange. In fact, it is visible that the model performance is directly proportionate to the number of steps a question requires. Question 3 of the `PAINTING_THE_WALL` problem is a very involved question that consists of three steps. Question 1 and 2 of the `TRIANGLE_ABC` problem, in contrast, are easier problems that require only two steps. Question 4, 5, and 6 have an increased performance time again because trapezoids have that extra requirement regarding its two bases. The performance time, in this sense, aligns well with expectations.

Unfortunately, Dataset 1 does not have any response time available for the students, so the model performance time cannot be compared to the human performance time. Nonetheless, the results shown in Figure 26 and Figure 27 are positive indications of how well the knowledge graph was able to identify the skills underlying Dataset 1.

6 Discussion

Going back all the way to Plato (257BC/2003, as cited by Inglis & Attridge, 2017), researchers (like Thorndike, 1914/1999, Singley & Anderson, 1985, and Taatgen, 2013) have found there is some type of transfer that occurs between tasks. Although a consensus has not yet been reached on what the unit of this transfer would be, this thesis proposes (in alignment with Taatgen, 2013 and Hoekstra et al., 2020) that transfer between tasks can be (partially) explained through skill reuse.

Defining skills as the largest unit of procedural knowledge that can be reused between tasks (Hoekstra et al., 2020), this thesis sets out to identify said skills. Particularly, knowledge graphs are proposed as tools for identifying the skills reused between tasks. A method is described in this thesis for generating a knowledge graph for this purpose and evaluating its proposed skills through cognitive models. The underlying research question throughout this process was: "Can knowledge graphs be used to identify the skills reused between tasks?"

The results from the cognitive models show that it is possible to extract skills from the knowledge graph that underlie a certain data set. These skills are shown to lead to the successful completion of a specific set of geometry tasks. While there is not enough evidence to conclude the identified skills are definitive and exclusive (i.e. it is possible a different set of skills could have produced the same results), the results do suggest that it is possible to use knowledge graphs to identify possible skills that are reused between tasks.

6.1 Evaluation of Proposed Method

Although the results in this thesis are positive in regards to the research question, further validation of the proposed method in this thesis is a vital constraint for its future applications. The method proposed is rather new (built off the work of Rozestraten, 2021), and if only for this reason, it still requires much work.

6.1.1 The Knowledge Graph Algorithm

The first place where the method requires more work is with the knowledge graph algorithm. The final knowledge graph that was accepted was not accepted because it was perfect, but because it was the best of the generated results.

The other generated knowledge graphs clearly showed a learning effect. Not only that, their results were so inconsistent, that it was not possible to generate cognitive models from these results. The final knowledge graph was the only graph from which skills could be extracted for the creation of the cognitive models.

Mitigating the learning effect was thus an important undertaking, and it is good that an updated version of the knowledge graph algorithm was created which takes the order of the tasks into account. Nonetheless, the final knowledge graph continued to show oddities like the ones discussed in the *Intermediate Results*. The steps of some problems, like `DESIGNING-A-QUILT`, were spread across the nodes without a discernible pattern. Others problems that should have been identical (such as `PENTAGON` and `PENTAGON_ABCDE`) were inexplicably placed in different nodes. Simply put, even the final knowledge graph did not always match with the division of problems among nodes as intuition would imply.

It is good to note here that intuition is not a ground truth. Sometimes algorithms will make choices that do not seem logical to the human eye; this does not necessarily make the algorithm wrong. Analysing the correctness of the knowledge graph through intuition implies that cognitive skills can be interpreted intuitively. This may not be the case. Since the exact nature of a skill is not known (what is its smallest unit?), it is possible that there is no straightforward, intuitive interpretation of skills. If that is the case, then evaluating the knowledge graph through intuition may not be the best approach. Generally, a better method needs to be designed for the evaluation of knowledge graphs as tools for identifying skills. Consistent measures are needed so that the evaluation of the knowledge graph is not dependent on the intuition of the programmer.

In this thesis, such measures are missing, and the intuitive approach suggests the knowledge graph algorithm used in this thesis could still be improved on. The algorithm used here is an updated version of the algorithm used in Rozestraten (2021). An interesting dichotomy between the work of Rozestraten (2021) and this work is that this work does not do additional clustering over the students and the problems. However, Rozenstraten's results indicate that this prior clustering of the data does help in generating better knowledge graphs. It is therefore possible that the data used in this thesis would have resulted in a more intuitive knowledge graph if, rather than skills across all problems, skill were identified across clusters of problems. In such a case, problems that are already similar (or near identical as `PENTAGON` and `PENTAGON_ABCDE`) could be clustered together prior to running the knowledge graph algorithm. The influence of student ability would furthermore be mitigated by clustering over students.

The reason clustering could not be applied in this thesis was that the used data set was both small and unbalanced (some problems were done by as few as four students). Generally, clustering algorithm do not know how to handle missing data, so in order for clustering to be applied to this data set, the missing values (i.e. the problems students did not do) would have to be substituted in some way. Although multiple methods exist to substitute or extrapolate missing values (e.g., Rozestraten, 2021 used a problem's mean outcome to substitute its missing values), each of these methods has its own benefits and drawbacks. The knowledge graph algorithm itself, by comparison, does not struggle with missing data, and therefore it does not requires the substitution of missing values.

Even if all missing values were substituted to make clustering possible, the clustering would still greatly reduce the available data. This could already be seen across the different data sets, where the available data decreased from 69 problem-steps in Dataset 1 to 46 in Dataset 3. While generally it may thus be beneficial to perform additional clustering, not only does this bring with it more design considerations, it may also not be ideal for all data sets.

Overall, more research should be done into the pre-clustering of the data and the effect this has on the generated knowledge graph. If the pre-clustering of the data consistently results in better knowledge graphs, it would show that the method proposed in this thesis may not be suitable for small data sets. Such a finding would match the beliefs of Falmagne et al. (1990), who already suggest that building a knowledge space requires a lot of data.

In addition to the size of the data set, the knowledge graph method should be validated further for data that has a wider variety of tasks. The 'Geometry Area (1996-97)' data set used in this thesis was chosen precisely because of its small scope. Consequently, the results shown in this thesis may not extrapolate towards data sets that are larger in scope. In general, it would be interesting to investigate how the scope of the data set affects the knowledge graph generated. It is theoretically possible that a certain degree of similarity is needed between the tasks for the knowledge graph to identify which skills are reused among these tasks. Alternatively, it is possible that for larger data sets, the knowledge graph method will only work well if clustering is also applied. The latter hypothesis aligns best with the results from Rozestraten (2021).

From the points mentioned here, it is clear there are many aspects of the knowledge graph algorithm that need further investigation. Since the method is relatively new, it needs to be tested across a variety of contexts, in order to both determine its validity as well as how different contexts and variables influence the algorithm.

6.1.2 The Cognitive Models

The same conclusion can be drawn for the cognitive models. One of the most interesting things to test for the cognitive models is whether grounding them in data did indeed reduce the influence of the modeller. Although it may be a derivative task, a lot could be learned from having another modeller set up models from the same knowledge graph generated in this thesis.

In particular, it would be interesting to see whether a different analyst would identify skill `0001000` in the same way. Some creative liberties were undoubtedly taken in the creation of the cognitive models, specifically in what skills could go into Model 0. Since Model 0 represented the prior knowledge of students going into the data set, it served as a buffer for any skills needed, so that Model 0 and Model 8 could be distinguished with only one skill.

For example, the `side` skill did not originally exist in Model 0. It was added because it would be needed for Model 8, and because an argument could be made for Model 0 possessing the skill. While Model 0 could not find sides through division, it could do so through the `segments` skill, and as such, there was evidence to support the addition of the `side` skill. Nonetheless, the primary reason for adding the `side` skill to Model 0 rather than introducing it in Model 8 was to ensure that Model 0 and Model 8 were distinguished by only one skill, as the data analysis suggested.

Of course, this interpretation assumes that the difference between the nodes in the knowledge graph is explained by a skill according to its PRIMs definition. At the start of this thesis, a skill was rather defined as the largest unit of procedural knowledge that can be reused between tasks. It is therefore possible that a PRIMs skill may not be the optimal representation of this definition. The difference between the nodes in the knowledge graph may just as well be explained by one or multiple operators, which together form the unit of procedural knowledge that is reused across nodes.

In that alternative interpretation, the conclusion that the `0001000` difference between the nodes corresponds specifically to the `division` skill may be premature. While it does match the data analysis, the idea that division is what sets Node 0 and Node 8 apart does not match with what one would expect students to know as prior knowledge. It seems unlikely that, by the time students are handling geometry problems, they do not yet know how to do division. Although it is possible that students did not need to apply their division knowledge for Node 0 but they do need it for Node 8, it is undeniable that the most important changes from Model 0 to Model 8 were implemented by updating existing skills with new operators, rather than through the `division` skill.

On the one hand, an argument can be made that perhaps the most important aspect of learning a new task lies precisely in how old skills need to change in order to incorporate new circumstances. This view matches that of Salvucci (2013), who explains that skill acquisition is a combination of skill reuse and skill *integration*. Integration, he postulates, defines "how skills are fused together to realise new task behaviours" (p. 830). The changes that needed to be made to create Model 8 strongly overlap with this idea of skill integration. In this way, the design of the cognitive models in this thesis very much supports the idea that skill integration is an important part of learning a new task.

On the other hand, highlighting division as the missing skill and incorporating the remaining necessary changes into new operators for existing skills can simply be viewed as sticking too closely to the idea that the difference between the nodes is explained by single skills as they are defined in PRIMs. It is my belief, however, that there is merit in the strictness of such an approach. If the skills identified by the knowledge graph are interpreted loosely (i.e. they could indicate multiple operators or even skill sets), then in some ways, the purpose of the data analysis is undermined. The additional constraints the data analysis places on the modelling process serve little value if they are subsequently taken as guidelines rather than proper constraints. If each modeller interprets the skills identified by the knowledge graph differently, then overall, the influence of the modeller is not reduced.

Yet though there may be a merit to it, it is very much possible that sticking so strongly to the data analysis is not the best approach, especially given that the knowledge graph algorithm still needs work. Without proper evaluation methods (which might not be so easily generated since it is not possible to establish a ground truth), it remains possible that the differences between nodes in the knowledge graph should not be explained by single skills but rather by multiple operators or through skill sets. In this interpretation, the knowledge graph is simply not precise enough to be able to distinguish between the various ways a unit of procedural knowledge may be defined.

As this discussion highlights, there is no clear right approach to interpreting the knowledge graph and generating the corresponding cognitive models. Precisely for this reason, it would be very interesting to see how another modeller might approach this dilemma.

Furthermore, it would also be interesting to see how else the models might need to be adjusted if Node 9 was included, and Node 11, and Node 25, and so forth. In theory, the accuracy of the identified skills should increase with each node, since each node creates a fuller picture of the skills needed to solve all the problems in the data set. Only by modelling all eleven nodes, would it be possible to identify all seven underlying skills. Of course, such a modelling undertaking would not be a trivial task, and it would require a large team of modellers.

This begs the question of whether the method proposed in this thesis is feasible for everyday use (such as in a classroom). As a counter-argument, it does have to be said that identifying the skills for one set of tasks only needs to be done once. After that, the underlying skills are known and are not expected to change.

There is a catch to this expectation. It creates the assumption that the skills are task-specific. This is not a complete view: Which skills are used to solve a task depend on both the task and the person undertaking the task. While a task might thus not change, having a new set of people tackle the task could result in a different set of identified skills.

This is something that the current cognitive models do not take into account. For each task, the cognitive model describes only one way to solve the task. The chosen strategy is one that is guaranteed to lead to the successful completion of the task, but there are both alternative strategies that could equally lead to this success, as there are strategies that will not lead to success.

Inadvertently, the models already showed examples of incorrect strategies leading to task failure with its transition from Model 0 to Model 8. In actuality, people's prior knowledge is much vaster than is taken into account for Model 0, and it is thus not at all unlikely a person will choose an incorrect strategies to try and solve a task it has not seen before. As was mentioned, people will likely learn from choosing a wrong strategy until eventually (if they continue to practice) they will learn a strategy that works. The models in this thesis intentionally move away from this development in learning. Nonetheless, the fact that multiple strategies may work is something that is not taken account by the cognitive models.

In fact, the existence of different strategies might also be one of the confounding factors of the knowledge graph algorithm. If multiple strategies were used by different students, neither the knowledge graph algorithm nor the cognitive models could incorporate this. In order to adjust to this possibility, more data would be required on a) the mistakes students made in solving the problems and b) the strategies they were taught/claimed to use.

The latter might be easy to obtain in the sense that many math questions ask students to show their steps. The cognitive tutor behind the 'Geometry Area (1996-97)' data set did not allow students to write down their own steps and instead forced them to follow the predefined steps by the tutor. Future cognitive tutors might benefit from asking students to show their work but allowing them a blank field to do so. This would, of course, be harder to analyse for future modellers. In this sense, this suggestion has its benefits and its drawbacks.

As a final shortcoming of the cognitive models, the visual processing must be addressed. In the models' visual representation, the different segments of the visual input are immediately coupled to their correct values. Contradicting this, reading comprehension is often considered a skill of its own, and building an internal representation of a written problem is not trivial (León & Escudero, 2017).

It is precisely because it is not trivial that it is not included in the cognitive models in this thesis. Considering the scope of reading comprehension, it would be better if a separate model for was built for this. Model 0 and Model 8 could then reuse skills from this model just as Model 8 reuses skills from Model 0.

Abstracting away from reading comprehension, while understandable because of its scope, does cause potential problems. If reading comprehension is one of the skills that underlie the data set, then the cognitive models in this thesis, by their design, could not replicate this skill. The fact that problems with more involved stories are further down in the knowledge graph does indicate reading comprehension may play a role in Dataset 1. There are thus a lot of benefits to gain from creating a cognitive model for reading comprehension.

To summarise, the cognitive models, like the knowledge graph algorithm, require further validation.

It would be particularly interesting to test the models' reproducibility, and to see whether grounding the models in data has reduced the influence of the modeller. Other factors that need to be considered is the existence of multiple strategies to solve the same task and the large undertaking that is reading comprehension.

6.2 Main Contributions

Despite of the work that is still required both in regards to the knowledge graph algorithm and the cognitive models, the positive findings of this thesis do provide further evidence for the use of knowledge graphs as tools for identifying (reused) skills. Previously, Rozestraten (2021) found that, while knowledge graphs were good tools for identifying the hierarchy between tasks, they were insufficient for identifying the required skills to complete these tasks. In this thesis, it is shown that knowledge graphs can be used to identify skills, but that they must be combined with cognitive models to do so.

Since this is the first time (as per my knowledge) that these two tools have been combined in this way, the method proposed in this thesis must be critically validated. If the method is valid, however, it shows much promise for the future of learning.

On the one hand, the knowledge graph on its own can potentially be used in education to keep track of the students' learning and progress. By comparing the problems a student answered correctly against the "skill map" identified by the knowledge graph, it should theoretically be possible to get an overview of the skills a student does and does not yet possess. With this overview, it is possible to provide students with more tailored curricula that focus on teaching them precisely the skills they do not yet possess. Specifically in combination with a cognitive tutor system (like the one that generated the data in this thesis), the knowledge graph method could become an important tool in the educative system.

By combining the knowledge graph with the cognitive model, it is possible to make the skills identified by the knowledge graph very concrete. If the knowledge graph becomes an integrated tool in education, this will be especially useful for teachers. Having concrete skills to work with will allow teachers to understand the knowledge graph better and thereby utilise it in the most effective manner.

Although the knowledge graph thus has hopeful implications for education, this is not the only way in which the proposed method in this thesis contributes to the existing literature. Firstly, it provides additional evidence that transfer between tasks can be explained through skill reuse, a stance also adopted by Taatgen (2013), Hoekstra et al. (2020), and Salvucci (2013).

Secondly, it suggests additional characteristics of skills, beyond that they are transferable. Through the adaptation of Model 0 to Model 8, it is suggested that skills are malleable. They may have the ability to change or 'update', as they adapt to new circumstances they come across. In fact, this may be a necessary characteristic of skills, since it is shown (through the overly specific first version of Model 0) that extreme specificity is only detrimental for explaining skill reuse. This finding well aligns with the motivations behind the PRIMs cognitive architecture, which was created precisely because Taatgen (2013) found that the commonly used production rules were too specific to explain the transfer that occurs between tasks.

This problem of extreme specificity is not limited to the debate on what the unit of transfer is (ACT-

R's production rules versus PRIMs primitive units). In general, many cognitive models in this day and age are made to be very specific. They are created to explain the cognition behind a specific task or phenomenon, like the Stroop effect (Stroop, 1935), and do not generalise beyond that phenomenon.

On the one hand, this is understandable because of scope limitations. The cognitive models in this thesis are also guilty of this, in that they simplify mathematical operations and do not go into a lot of depth in terms of visual processing. To understand a specific cognitive phenomenon, it is not strange to abstract away from the parts of cognition that are less relevant for this phenomenon.

On the other hand, as Allen Newell aptly put in his *Unified Theories of Cognition* (Newell, 1990), many small, very specific theories do not the whole of cognition explain. This thesis demonstrates the use of skill-based cognitive models as a method for explaining a more unified theory of cognition (Taatgen, 2014). While one model may be specific to one situation, the skill reuse between the models allows a holistic whole to be created that can demonstrate the cognition over a wider span of tasks (just as Model 8 was capable of solving both the problems in Node 0 and those in Node 8).

In general, the proposed method here provides an innovative way of building cognitive models. Cognitive models are typically built from the expertise of the modeller (as e.g. Hoekstra et al., 2020 demonstrates). The modeller has to adhere to certain constraints, provide justification for their choices, and must evaluate their proposed model to confirm its validity. While this method is thus reliable, it does allow for a wide disparity between models. How a cognitive model turns out is largely influenced by the modeller that built it. Even in this thesis, the skills extracted from the knowledge graph were influenced and decided on by the modeller.

What this thesis does propose, however, is a method for potentially reducing this influence of the cognitive modeller. By grounding the cognitive model in a data analysis, the modeller must adhere to additional constraints. All the modelling choices made are made around a data analysis that is performed prior. In this way, the model has a more robust grounding than a typical cognitive model, which is built from a less quantifiable 'expertise from the modeller'. As with the potential use of the knowledge graph in education, further validation of the method proposed in this thesis could thus prove beneficial for the field of cognitive modelling as a whole.

6.3 Conclusion

To conclude, the work done in this thesis indicates knowledge graphs can successfully be used to identify the skills reused between tasks. Since the method proposed herein is rather novel, it requires further validation both in its data analysis and the created cognitive models. If the results from this thesis can be replicated, however, then the proposed methods could have far-reaching effects both for the future of education as well as the overall field of cognitive modelling.

References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglas, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of mind. *Pyschological review*, *111*(4), 1036–1060. https://doi.org/10.1037/0033-295X.111.4.1036

Anokhin, A. P., Birbaumer, N., Lutzenberger, W., Nikolaev, A., & Vogel, F. (1996). Age increases brain complexity. *Electroencephalography and clinical neurophysiology*, *99*(1), 63–68. https://doi.org/10.1016/0921-884x(96)95573-3

Besner, D., Stolz, J. A., & Boutilier, C. (1997). The Stroop effect and the myth of automaticity. *Psychonomic Bulletin & Review*, *4*, 221–225. https://doi.org/10.3758/BF03209396

Chong, H.-Q., Tan, A.-H., & Ng, G.-W. (2007). Integrated cognitive architectures: A survey. *Artificial Intelligence Review*, *28*, 103–130. https://doi.org/10.1007/s10462-009-9094-9

Cohen, J. D., Dunbar, K., & McClelland, J. L. (1990). On the control of automatic processes: A parallel distributed processing account of the Stroop effect. *Psychological Review*, *97*(3), 332–361. https://doi.org/10.1037/0033-295X.97.3.332

Costandi, M. (2016). *Neuroplasticity*. The MIT Press.

Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal, Complex Systems*, *1695*(5), 1–9. https://igraph.org

Dube, R. (2022, May 10). *How to read binary*. Lifewire. https://www.lifewire.com/how-to-read-binary-4692830

Falmagne, J.-C., Koppen, M., Villano, M., Doignon, J.-P., & Johannesen, L. (1990). Introduction to knowledge spaces: How to build, test, and search them. *Psychological Review*, *97*(2), 201–224. https://doi.org/10.1037/0033-295X.97.2.201

Ferlazzo, F., Lucido, S., Di Nocera, F., Fagioli, S., & Sdoia, S. (2007). Switching Between Goals Mediates the Attentional Blink Effect. *Experimental Psychology*, *54*(2), 89–98. https://doi.org/10.1027/1618-3169.54.2.89

Güneş, F. (2018). Skill based approach and teaching skill. In S. Sidekli (Ed.), *The skill approach in education: From theory to practice* (pp. 1–74). Cambridge Scholars Publisher.

Hoekstra, C., Martens, S., & Taatgen, N. A. (2020). A skill-based approach to modeling the attentional blink. *Topics in Cognitive Science*, *12*(3), 1030–1045. https://doi.org/10.1111/tops.12514

Inglis, M., & Attridge, N. (2017). *Does mathematical study develop logical thinking?: Testing the theory of formal discipline*. World Scientific.

International Association for the Evaluation of Educational Achievement (IEA). (2008). *TIMSS 2007 Technical Report*. TIMSS & PIRLS International Study Center, Boston College. https://timssandpirls.bc.edu/TIMSS2007/techreport.html

Ji, S., Pan, S., Cambria, E., Marttinen, P., & Yu, P. S. (2022). A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, *33*(2), 494–514. https://doi.org/10.1109/TNNLS.2021.3070843

Kempermann, G. (2006). Adult Neurogenesis. In P. B. Baltes, P. A. Reuter-Lorenz, & F. Rösler (Eds.), *Lifespan Development and the Brain: The Perspective of Biocultural Co-Constructivism* (pp. 82–108). Cambridge University Press. https://doi.org/10.1017/CBO9780511499722.006

Kieras, D. E., & Meyer, D. E. (1997). An Overview of the EPIC Architecture for Cognition and Performance With Application to Human-Computer Interaction. *Human-Computer Interaction*, *12*(4), 391–438.

Koedinger, K. R., de Baker, R. S. J., Cunningham, K., Skogsholm, A., Leber, B., & Stamper, J. (2010). A Data Repository for the EDM community: The PSLC DataShop. In C. Romero, S. Ventura, M. Pechenizkiy, & R. Baker (Eds.), *Handbook of Educational Data Mining*. CRC Press.

Koedinger, K. R., McLaughlin, E. A., & Stamper, J. C. (2012). *Automated Student Model Improvement* [Paper presentation]. 5th International Conference on Educational Data Mining Society (EDM), Chania, Greece. https://eric.ed.gov/?id=ED537201

Kotseruba, I., & Tsotsos, J. K. (2020). 40 years of cognitive architectures: Core cognitive abilities and practical applications. *Artificial Intelligence Review*, *53*, 17–94. https://doi.org/10.1007/s10462-018-9646-y

Langley, P., & Choi, D. (2006). A Unified Cognitive Architecture for Physical Agents. *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, 1469–1474. https://doi.org/10.5555/1597348.1597422

León, J. A., & Escudero, I. (2017). *Reading Comprehension in Educational Settings* (Vol. 16). John Benjamins Publishing Company.

Marilee, S. (1999). *Learning and Memory : The Brain in Action*. ASCD.

Middleton, H., & Baartman, L. K. J. (2013). *Transfer, Transitions and Transformations of Learning*. Brill. https://doi.org/10.1007/978-94-6209-437-6

Miller, G. A. (2003). The cognitive revolution: A historical perspective. *Trends in Cognitive Sciences*, *7*(3), 141–144. https://doi.org/10.1016/S1364-6613(03)00029-9

Morris, C. D., Bransford, J. D., & Franks, J. J. (1977). Levels of processing versus transfer appropriate processing. *Journal of Verbal Learning and Verbal Behavior*, *16*(5), 519–533. https://doi.org/10.1016/S0022-5371(77)80016-9

National Research Council. (2000). *How People Learn : Brain, Mind, Experience, and School: Expanded Edition* (Vol. 2). National Academies Press. https://doi.org/10.17226/9853

Newell, A. (1990). *Unified theories of cognition*. Harvard University Press.

R Core Team. (2021). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. Vienna, Austria. https://www.R-project.org/

Raymond, J. E., Shapiro, K. L., & Arnell, K. M. (1992). Temporary suppression of visual processing in an RSVP task: An attentional blink?. *Journal of Experimental Psychology: Human Perception and Performance*, *18*(3), 849–860.

Rozestraten, K. (2021). *Identifying Underlying Skills in Math Problems - A Data-Driven Approach* (Master's thesis). Rijksuniversiteit Groningen. http://fse.studenttheses.ub.rug.nl/id/eprint/26505

Russel, S. J., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.). Prentice Hall.

Salvucci, D. D. (2013). Integration and Reuse in Cognitive Skill Acquisition. *Cognitive Science*, *37*(5), 829–860. https://doi.org/10.1111/cogs.12032

Singley, M. K., & Anderson, J. R. (1985). The transfer of text-editing skill. *Journal of Man-Machine Interaction*, *22*, 402–423. http://act-r.psy.cmu.edu/?post_type=publications&p=13752

Stirling, N. (1979). Stroop Interference: An Input and an Output Phenomenon. *Quarterly Journal of Experimental Psychology*, *31*(1), 121–132. https://doi.org/10.1080/14640747908400712

Stroop, J. R. (1935). Studies of interference in serial verbal reactions. *Journal of Experimental Psychology*, *18*(6), 643–662. https://doi.org/10.1037/h0054651

Taatgen, N. A. (2013). The nature and transfer of cognitive skills. *Psychological Review*, *120*(3), 439–471. https://doi.org/10.1037/a0033138

Taatgen, N. A. (2014). Between architecture and model: Strategies for cognitive control. *Biologically Inspired Cognitive Architectures*, *8*, 132–139. https://doi.org/10.1016/j.bica.2014.03.010

Taatgen, N. A. (2022). *PRIMS-Tutorial*. https://github.com/ntaatgen/PRIMs-Tutorial

Thorndike, E. L. (1999). *Education Psychology: BRIEFER COURSE*. Routledge. (Original work published 1914)

Xiang, Y., Gubian, S., Suomela, B., & Hoeng, J. (2013). Generalized Simulated Annealing for Global Optimization: The GenSA Package. *The R Journal*, *5*(1), 13–28. https://doi.org/10.32614/RJ-2013-002

Appendices

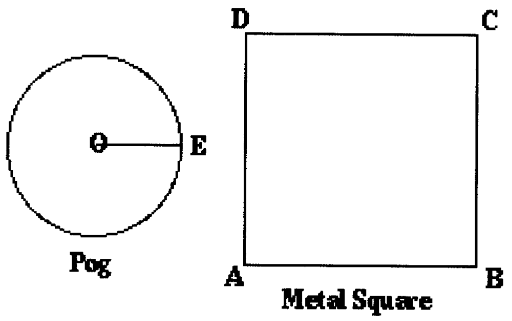
Appendix A contains additional analyses for the generated knowledge graphs. Appendix B gives the full content of the final knowledge graph nodes. Appendix C corresponds to the code for the cognitive models designed in the thesis.

A Addendum to the Intermediate Results

A.1 Problem 42: POGS

This problem shows the same learning effect that is explained in the *Intermediate Results* of Section 4. An identical step is classified into Node 7 for Question 1, Node 1 for Question 2, and Node 0 for Question 3.

POGS: SECTION FIVE, #2



Problem Statement

Pogs are circular metal disks used for recreation.

1. The radius of the pog is 2 inches. If a pog is punched out of a square piece of metal measuring 4 inches per side, find the square inches of scrap metal remaining.

2. The radius of the pog is 4 inches. If a pog is punched out of a square piece of metal measuring 8 inches per side, find the amount of scrap metal remaining.

3. The radius of the pog is 6 inches. If a pog is punched out of a square piece of metal measuring 12 inches per side, find the amount of scrap metal remaining.

NOTE: To find the area of the scrap metal remaining, you might have to first find the area of the pog, and the area of the square

pi = 3.1416

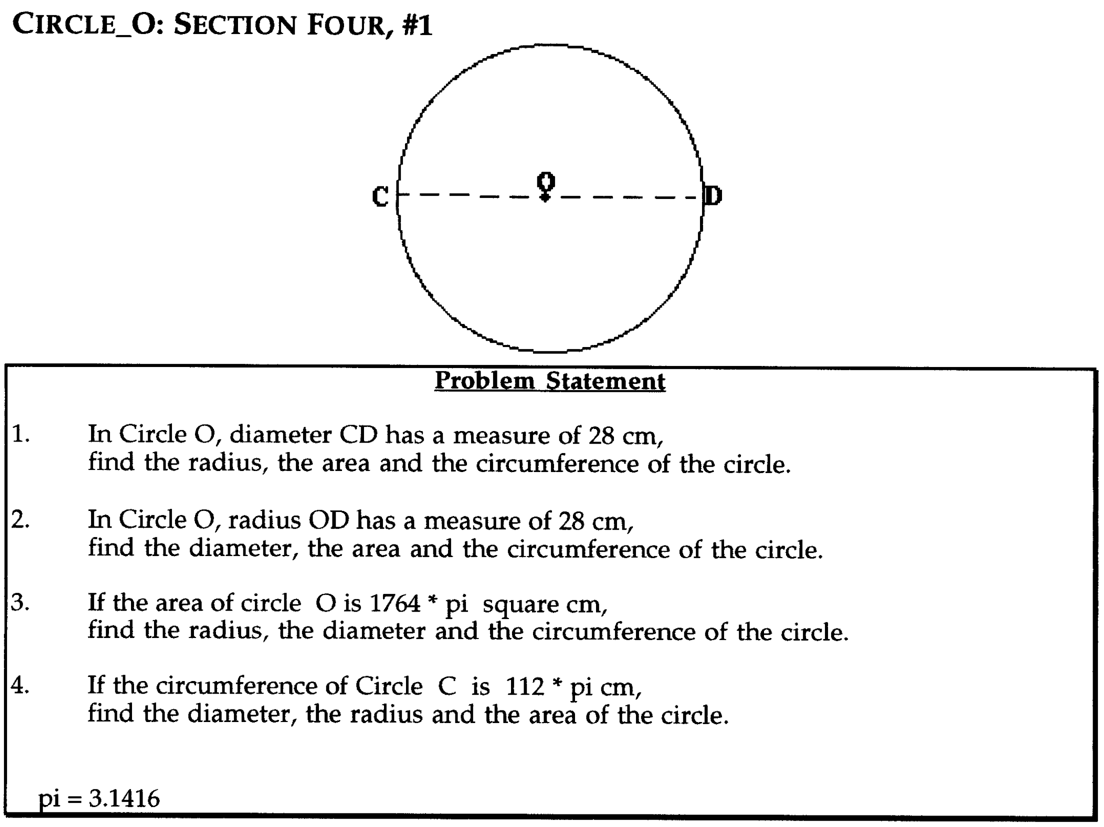
Figure 28: The POGS problem in Dataset 1.

	Area of scrap metal
Units	sq. inches
Question 1	3.43
Question 2	13.73
Question 3	30.9

Table 12: The recorded step of the three questions of POGS, with the correct answers.

A.2 The Circle Problems

The four problems in this subsection show a possible learning effect across problems, as explained in the *Intermediate Results* of Section 4. The problems `CIRCLE_DIAMETER`, `CIRCLE_AREA`, `CIRCLE_CIRCUMFERENCE` correspond to questions 1, 3, and 4 of the `CIRCLE_O` problem respectively. Since each question of `CIRCLE_O` is unique (students are given a different variable to start off with), limited learning is expected to occur across questions for this problem.



CIRCLE_O: SECTION FOUR, #1

Problem Statement

1. In Circle O, diameter CD has a measure of 28 cm,
   find the radius, the area and the circumference of the circle.
2. In Circle O, radius OD has a measure of 28 cm,
   find the diameter, the area and the circumference of the circle.
3. If the area of circle O is 1764 * pi square cm,
   find the radius, the diameter and the circumference of the circle.
4. If the circumference of Circle C is 112 * pi cm,
   find the diameter, the radius and the area of the circle.

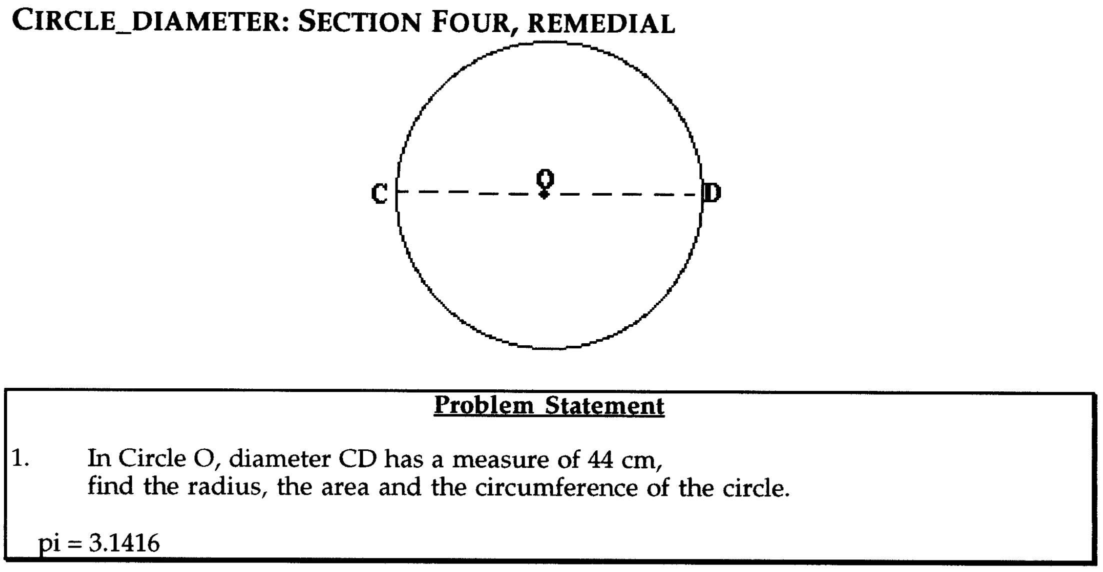
pi = 3.1416

Figure 29: The `CIRCLE_O` problem, which consists of four questions.

	Diameter (CD)	Radius (OD)	Area of Circle O	Circumference of Circle O
Units	cm	cm	sq. cm	cm
Question 1		14	615.75	87.96
Question 3	84	42		263.89
Question 4	112	56	9852.06	

Table 13: The node distribution of the steps for the `CIRCLE_O` problem. Steps that are not in Dataset 1 are greyed out.

Figure 30 shows the CIRCLE_DIAMETER problem (which corresponds to question 1 of the CIRCLE_O problem). Table 14 shows that each step for this question is classified into Node 0. This strongly contrasts the node distribution of Question 1 of the CIRCLE_O problem, where the radius step is classified as Node 1, and the area and circumference steps are both classified as Node 5. Since the equivalent steps are classified as higher in the knowledge graph for the CIRCLE_DIAMETER problem, it is feasible to believe that students saw the CIRCLE_DIAMETER problem only after the CIRCLE_O problem.



CIRCLE_DIAMETER: SECTION FOUR, REMEDIAL

Problem Statement

1. In Circle O, diameter CD has a measure of 44 cm, find the radius, the area and the circumference of the circle.

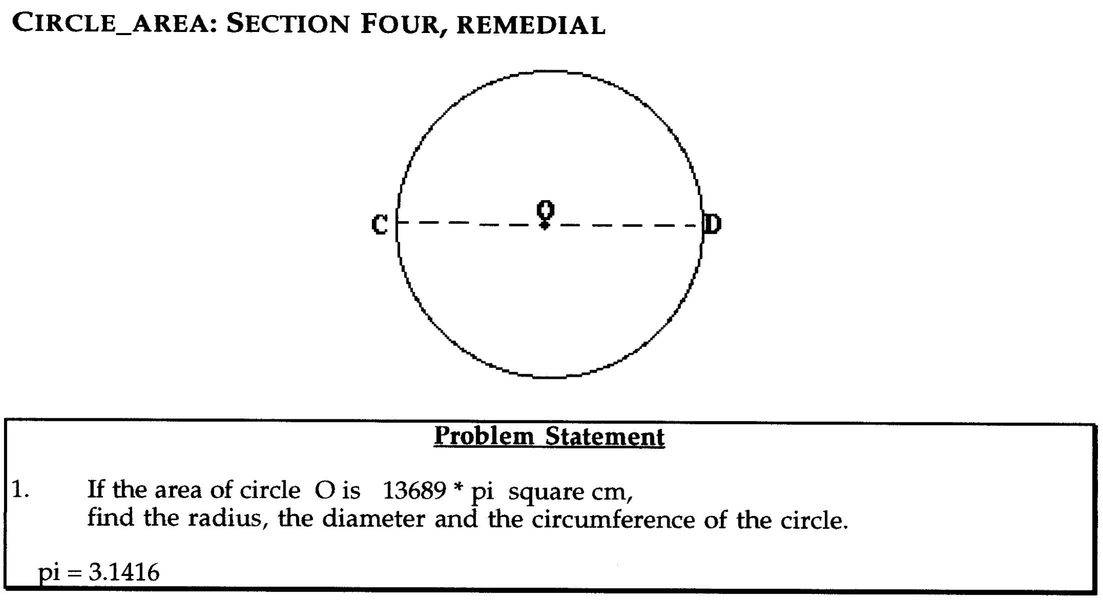
pi = 3.1416

Figure 30: The CIRCLE_DIAMETER problem, which overlaps with Question 1 of the CIRCLE_O problem.

	Diameter (CD)	Radius (OD)	Area of Circle O	Circumference of Circle O
Units	cm	cm	sq. cm	cm
Question 1	▨	22	1520.53	138.23

Table 14: The node distribution of the steps for the CIRCLE_DIAMETER problem. Steps that are not in Dataset 1 are greyed out.

For the CIRCLE_AREA problem (see Figure 31), the results are more ambiguous. In Table 13, the diameter step is classified into Node 1, and the radius and circumference step are classified into Node 3. In Table 15, however, the diameter step is perceived as more difficult (i.e. requiring more skills) and placed in Node 3. Meanwhile, the radius and circumference step are perceived as easier and are placed in Node 1 each. The perceived difficulty of the steps is, in a matter, flipped. This cannot be explained by the CIRCLE_O problem occurring before the CIRCLE_AREA problem.



CIRCLE_AREA: SECTION FOUR, REMEDIAL

Problem Statement

1. If the area of circle O is 13689 * pi square cm,
find the radius, the diameter and the circumference of the circle.

pi = 3.1416

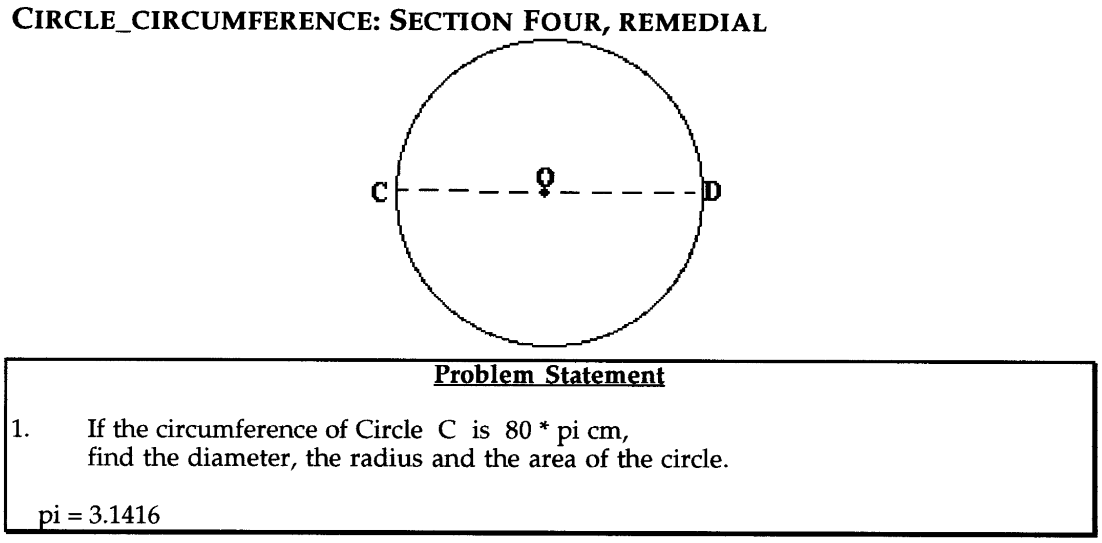
Figure 31: The CIRCLE_AREA problem, which overlaps with Question 3 of the CIRCLE_O problem.

Intuitively, neither classification necessarily makes more sense than the other. Finding the diameter *could* be more difficult, because it requires one additional step (finding the radius first - which can be determined directly from the information given). On the other hand, once the radius is determined, finding the diameter from this value is not very difficult (multiply the radius by two). Finding the radius, on the other hand, requires one to reverse the formula for determining the area of a circle, which seems like the more difficult of the two tasks. Once again, since there is no ground truth here, it is difficult to say which distribution is correct. However, what is clear is that the differences in distribution cannot be explained fully by a learning effect. For these problems, it might be more likely that students on average did not do both problems (and thus these node distributions are independent of one another).

	Diameter (CD)	Radius (OD)	Area of Circle O	Circumference of Circle O
Units	cm	cm	sq. cm	cm
Question 1	234	117		735.13

Table 15: The node distribution of the steps for the CIRCLE_AREA problem. Steps that are not in Dataset 1 are greyed out.

The final problem to analyse is CIRCLE_CIRCUMFERENCE, shown in Figure 32. In Table 16 it is apparent that not all steps of this question were in Dataset 1. The question asks for the diameter, the radius, and the area, but only the radius step is included in Dataset 1. The missing steps are, however, not so strange, since Dataset 1 contained specifically the 69 most-done problem-steps. If the other steps of the CIRCLE_CIRCUMFERENCE problem were not done with enough frequency (for whichever reason), they would not be included in Dataset 1. In any case, the radius step is classified into Node 3. This is the same node that the radius step of Question 4 from problem CIRCLE_O was classified into. These problems thus show a consistent classification, which strengthens the belief that the radius step of a circle problem should be placed into Node 3.



CIRCLE_CIRCUMFERENCE: SECTION FOUR, REMEDIAL

Problem Statement

1. If the circumference of Circle C is 80 * pi cm,
find the diameter, the radius and the area of the circle.

pi = 3.1416

Figure 32: The CIRCLE_CIRCUMFERENCE problem, which overlaps with Question 4 of the CIRCLE_O problem.

	Diameter (CD)	Radius (OD)	Area of Circle O	Circumference of Circle O
Units	cm	cm	sq. cm	cm
Question 1	▨	40	▨	▨

Table 16: The node distribution of the steps for the CIRCLE_CIRCUMFERENCE problem. Steps that are not in Dataset 1 are greyed out.

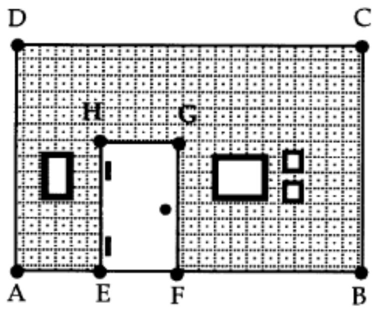
This subsection of **Addendum to the Intermediate Results** is meant to provide additional evidence for the learning observed across problems in the *Intermediate Results* of Section 4. Clear evidence of learning is shown between CIRLCE_O and CIRCLE_DIAMETER, but the other two problems paint more complex pictures. These results indicate that the effect of learning across problems is more complex than the effect of learning across questions. It is feasible to think the order the problems are shown in has some effect on their performance, but since this order is randomised across students, it is more difficult to strongly quantify this effect.

B Node Content of the Final Knowledge Graph

B.1 Node 0 (0000000)

Node 0

PAINTING_THE_WALL: SECTION ONE, #5



Problem Statement

1. The height of a wall is 22.5' and a 7' x 17.5' rectangular door is positioned on the wall such as there is 10' of wall remaining on the left side and 3' of the wall remaining on the right side.

   Find the area of the wall to be painted. Do not paint the door.

2. The height of a wall is 25.0' and a 8' x 20.0' rectangular door is positioned on the wall such as there is 8' of wall remaining on the left side and 4' of the wall remaining on the right side.

   Find the area of the wall to be painted. Do not paint the door.

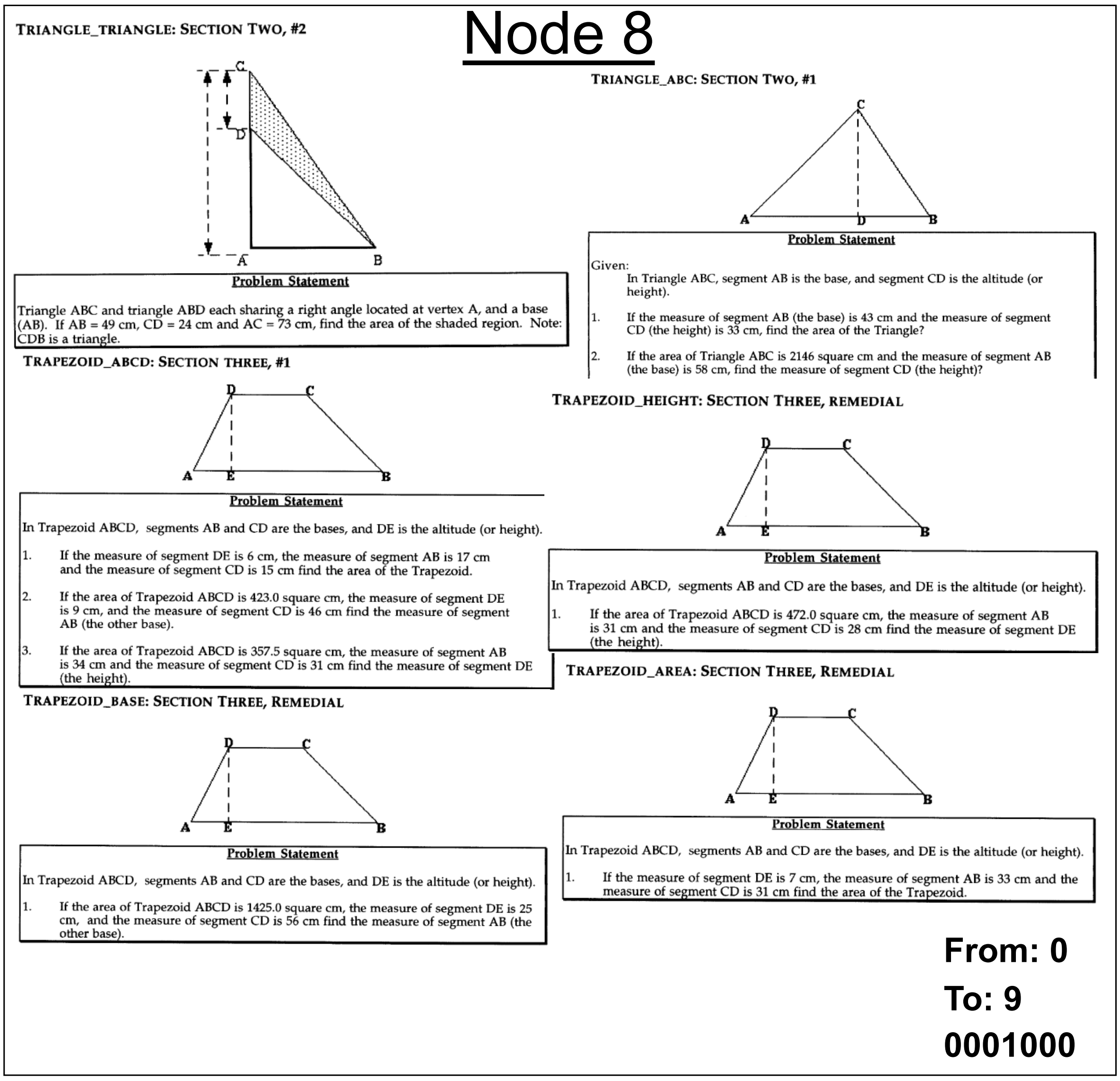
3. The height of a wall is 25.0' and a 8' x 20.0' rectangular door is positioned on the wall such as there is 10' of wall remaining on the left side and 2.5' of the wall remaining on the right side.

   Find the area of the wall to be painted. Do not paint the door.

From:
To: 8
0000000

Figure 33: Node 0 of the final knowledge graph. This node contains none of the skills in the underlying data set.

B.2 Node 8 (`0001000`)



Node 8

TRIANGLE_TRIANGLE: SECTION TWO, #2

Problem Statement

Triangle ABC and triangle ABD each sharing a right angle located at vertex A, and a base (AB). If AB = 49 cm, CD = 24 cm and AC = 73 cm, find the area of the shaded region. Note: CDB is a triangle.

TRIANGLE_ABC: SECTION TWO, #1

Problem Statement

Given:
In Triangle ABC, segment AB is the base, and segment CD is the altitude (or height).

1. If the measure of segment AB (the base) is 43 cm and the measure of segment CD (the height) is 33 cm, find the area of the Triangle?

2. If the area of Triangle ABC is 2146 square cm and the measure of segment AB (the base) is 58 cm, find the measure of segment CD (the height)?

TRAPEZOID_ABCD: SECTION THREE, #1

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the measure of segment DE is 6 cm, the measure of segment AB is 17 cm and the measure of segment CD is 15 cm find the area of the Trapezoid.

2. If the area of Trapezoid ABCD is 423.0 square cm, the measure of segment DE is 9 cm, and the measure of segment CD is 46 cm find the measure of segment AB (the other base).

3. If the area of Trapezoid ABCD is 357.5 square cm, the measure of segment AB is 34 cm and the measure of segment CD is 31 cm find the measure of segment DE (the height).

TRAPEZOID_HEIGHT: SECTION THREE, REMEDIAL

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the area of Trapezoid ABCD is 472.0 square cm, the measure of segment AB is 31 cm and the measure of segment CD is 28 cm find the measure of segment DE (the height).

TRAPEZOID_BASE: SECTION THREE, REMEDIAL

Problem Statement

In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the area of Trapezoid ABCD is 1425.0 square cm, the measure of segment DE is 25 cm, and the measure of segment CD is 56 cm find the measure of segment AB (the other base).

TRAPEZOID_AREA: SECTION THREE, REMEDIAL

Problem Statement

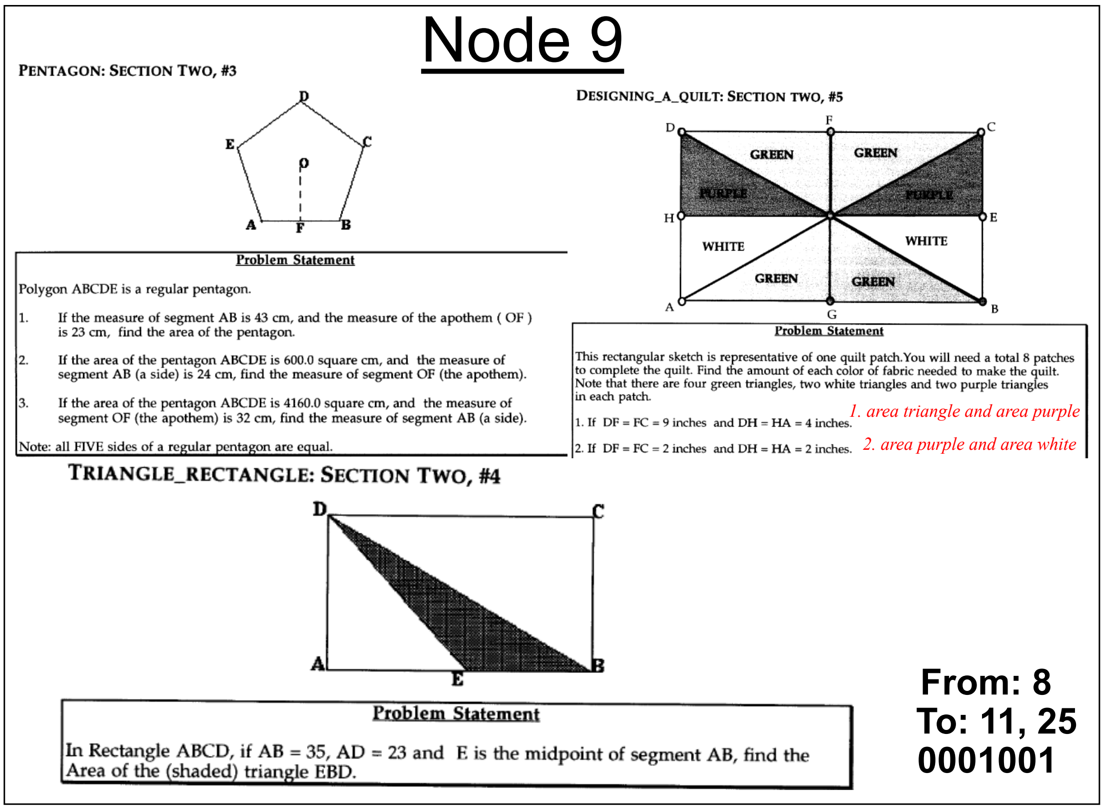
In Trapezoid ABCD, segments AB and CD are the bases, and DE is the altitude (or height).

1. If the measure of segment DE is 7 cm, the measure of segment AB is 33 cm and the measure of segment CD is 31 cm find the area of the Trapezoid.

From: 0
To: 9
0001000

Figure 34: Node 8 of the final knowledge graph. This node contains 1 skill.

B.3 Node 9 (`0001001`)



Node 9

PENTAGON: SECTION TWO, #3

Problem Statement

Polygon ABCDE is a regular pentagon.

1. If the measure of segment AB is 43 cm, and the measure of the apothem (OF) is 23 cm, find the area of the pentagon.
2. If the area of the pentagon ABCDE is 600.0 square cm, and the measure of segment AB (a side) is 24 cm, find the measure of segment OF (the apothem).
3. If the area of the pentagon ABCDE is 4160.0 square cm, and the measure of segment OF (the apothem) is 32 cm, find the measure of segment AB (a side).

Note: all FIVE sides of a regular pentagon are equal.

DESIGNING_A_QUILT: SECTION TWO, #5

Problem Statement

This rectangular sketch is representative of one quilt patch.You will need a total 8 patches to complete the quilt. Find the amount of each color of fabric needed to make the quilt. Note that there are four green triangles, two white triangles and two purple triangles in each patch.

1. If DF = FC = 9 inches and DH = HA = 4 inches. *1. area triangle and area purple*
2. If DF = FC = 2 inches and DH = HA = 2 inches. *2. area purple and area white*

TRIANGLE_RECTANGLE: SECTION TWO, #4

Problem Statement

In Rectangle ABCD, if AB = 35, AD = 23 and E is the midpoint of segment AB, find the Area of the (shaded) triangle EBD.

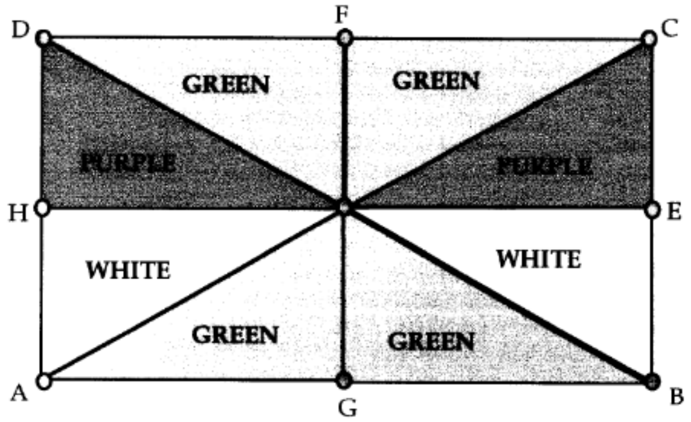
From: 8
To: 11, 25
0001001

Figure 35: Node 9 of the final knowledge graph. The red text specifies which steps are included in the node. This node contains 2 skills.

B.4 Node 11 (`0001011`)

Node 11

DESIGNING_A_QUILT: SECTION TWO, #5



Problem Statement

This rectangular sketch is representative of one quilt patch.You will need a total 8 patches to complete the quilt. Find the amount of each color of fabric needed to make the quilt. Note that there are four green triangles, two white triangles and two purple triangles in each patch.

1. If DF = FC = 9 inches and DH = HA = 4 inches.

2. If DF = FC = 2 inches and DH = HA = 2 inches.

3. If DF = FC = 27 inches and DH = HA = 12 inches.

1. white area

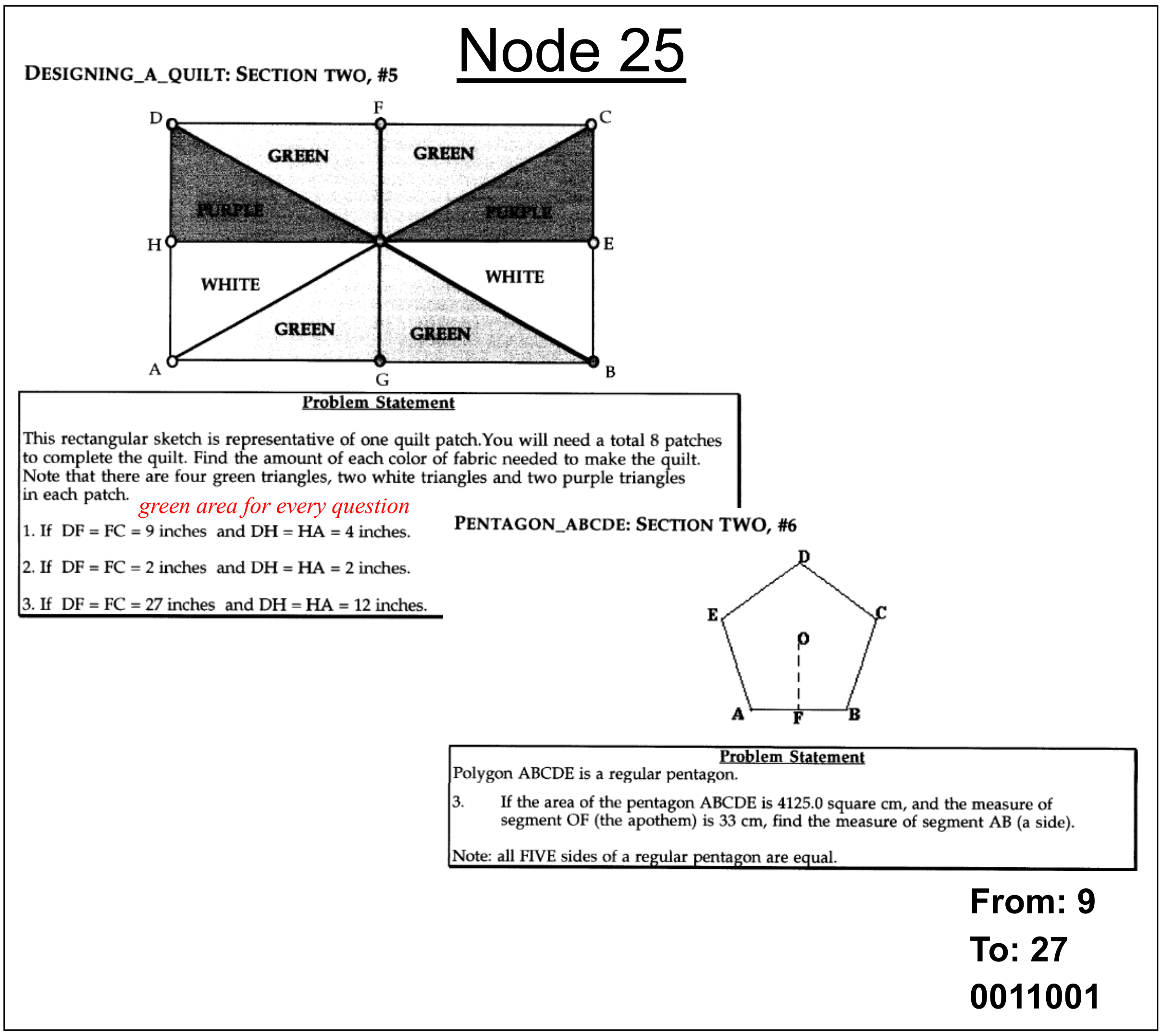
2. area triangle

3. area triangle and area white

From: 9
To: 27
0001011

Figure 36: Node 11 of the final knowledge graph. The red text specifies which steps are included in the node. This node contains 3 skills.

B.5 Node 25 (`0011001`)



Node 25

DESIGNING_A_QUILT: SECTION TWO, #5

D
F
C
GREEN
GREEN
PURPLE
PURPLE
H
E
WHITE
WHITE
GREEN
GREEN
A
G
B

Problem Statement

This rectangular sketch is representative of one quilt patch.You will need a total 8 patches to complete the quilt. Find the amount of each color of fabric needed to make the quilt. Note that there are four green triangles, two white triangles and two purple triangles in each patch.

green area for every question

1. If DF = FC = 9 inches and DH = HA = 4 inches.
2. If DF = FC = 2 inches and DH = HA = 2 inches.
3. If DF = FC = 27 inches and DH = HA = 12 inches.

PENTAGON_ABCDE: SECTION TWO, #6

Problem Statement

Polygon ABCDE is a regular pentagon.

3. If the area of the pentagon ABCDE is 4125.0 square cm, and the measure of segment OF (the apothem) is 33 cm, find the measure of segment AB (a side).

Note: all FIVE sides of a regular pentagon are equal.

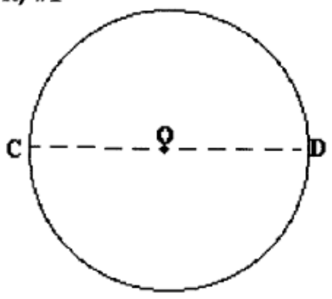
From: 9
To: 27
0011001

Figure 37: Node 25 of the final knowledge graph. The red text specifies which steps are included in the node. This node contains 3 skills.

B.6 Node 27 (`0011011`)

Node 27

CIRCLE_O: SECTION FOUR, #1

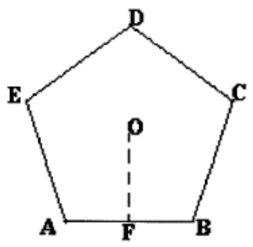


Problem Statement

1. In Circle O, diameter CD has a measure of 28 cm, find the radius, the area and the circumference of the circle.

2. In Circle O, radius OD has a measure of 28 cm, find the diameter, the area and the circumference of the circle.

PENTAGON_ABCDE: SECTION TWO, #6



Problem Statement

Polygon ABCDE is a regular pentagon.

1. If the measure of segment AB is 65 cm, and the measure of the apothem (OF) is 37 cm, find the area of the pentagon.

2. If the area of the pentagon ABCDE is 900.0 square cm, and the measure of segment AB (a side) is 24 cm, find the measure of segment OF (the apothem).

From: 11, 25

To: 91

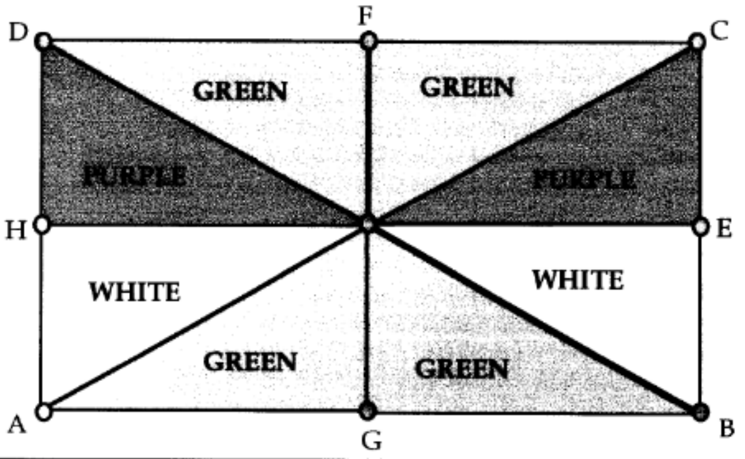
0011011

Figure 38: Node 27 of the final knowledge graph. This node contains 4 skills.

B.7 Node 108 (`1101100`)

Node 108

DESIGNING_A_QUILT: SECTION TWO, #5



Problem Statement

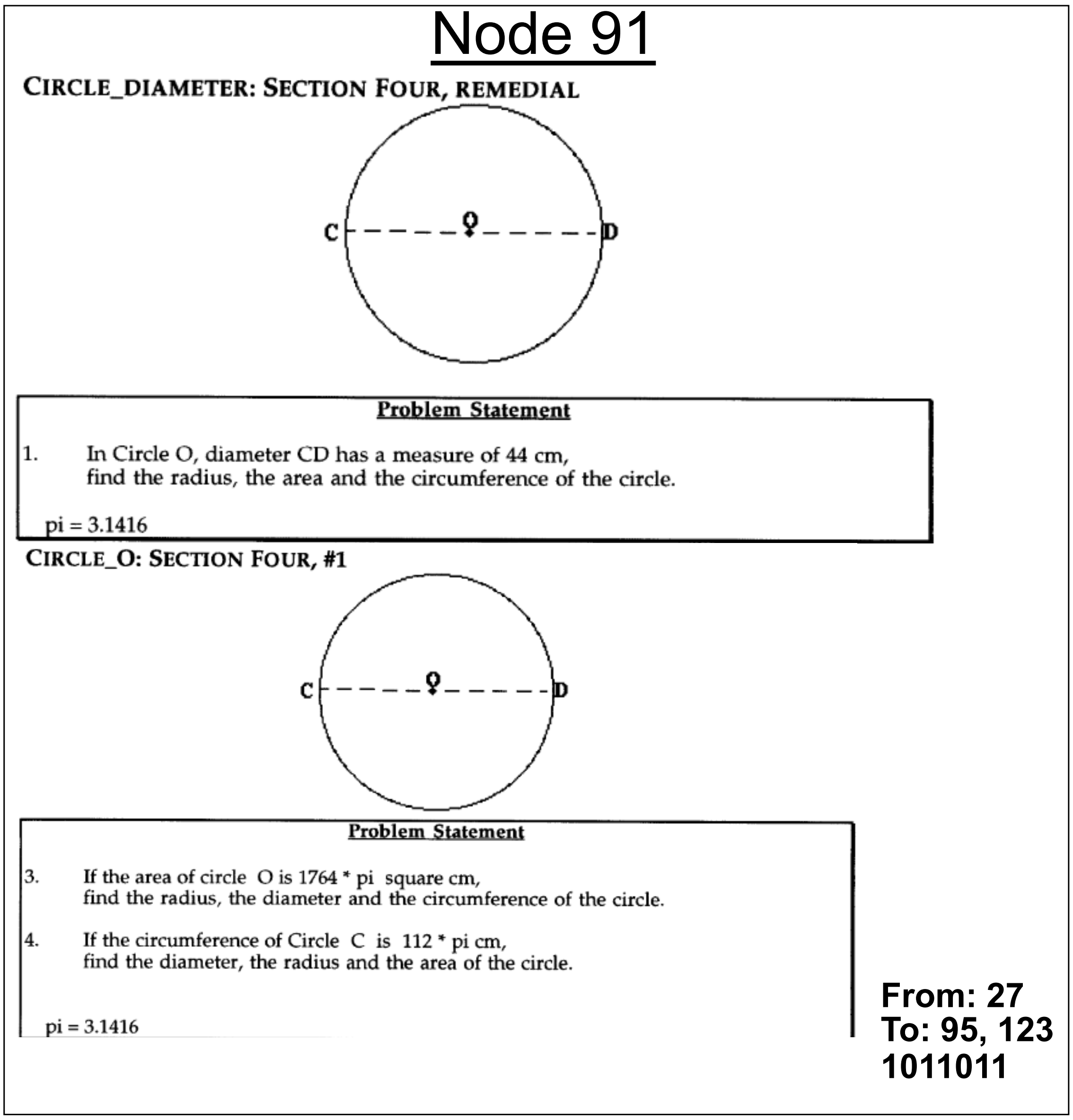
This rectangular sketch is representative of one quilt patch.You will need a total 8 patches to complete the quilt. Find the amount of each color of fabric needed to make the quilt. Note that there are four green triangles, two white triangles and two purple triangles in each patch. *purple area only*

3. If DF = FC = 27 inches and DH = HA = 12 inches.

From: 8
To: 127
1101100

Figure 39: Node 108 of the final knowledge graph. The red text specifies which steps are included in the node. This node contains 4 skills.

B.8 Node 91 (`1011011`)


Node 91

CIRCLE_DIAMETER: SECTION FOUR, REMEDIAL

Problem Statement

1. In Circle O, diameter CD has a measure of 44 cm, find the radius, the area and the circumference of the circle.

pi = 3.1416

CIRCLE_O: SECTION FOUR, #1

Problem Statement

3. If the area of circle O is 1764 * pi square cm, find the radius, the diameter and the circumference of the circle.

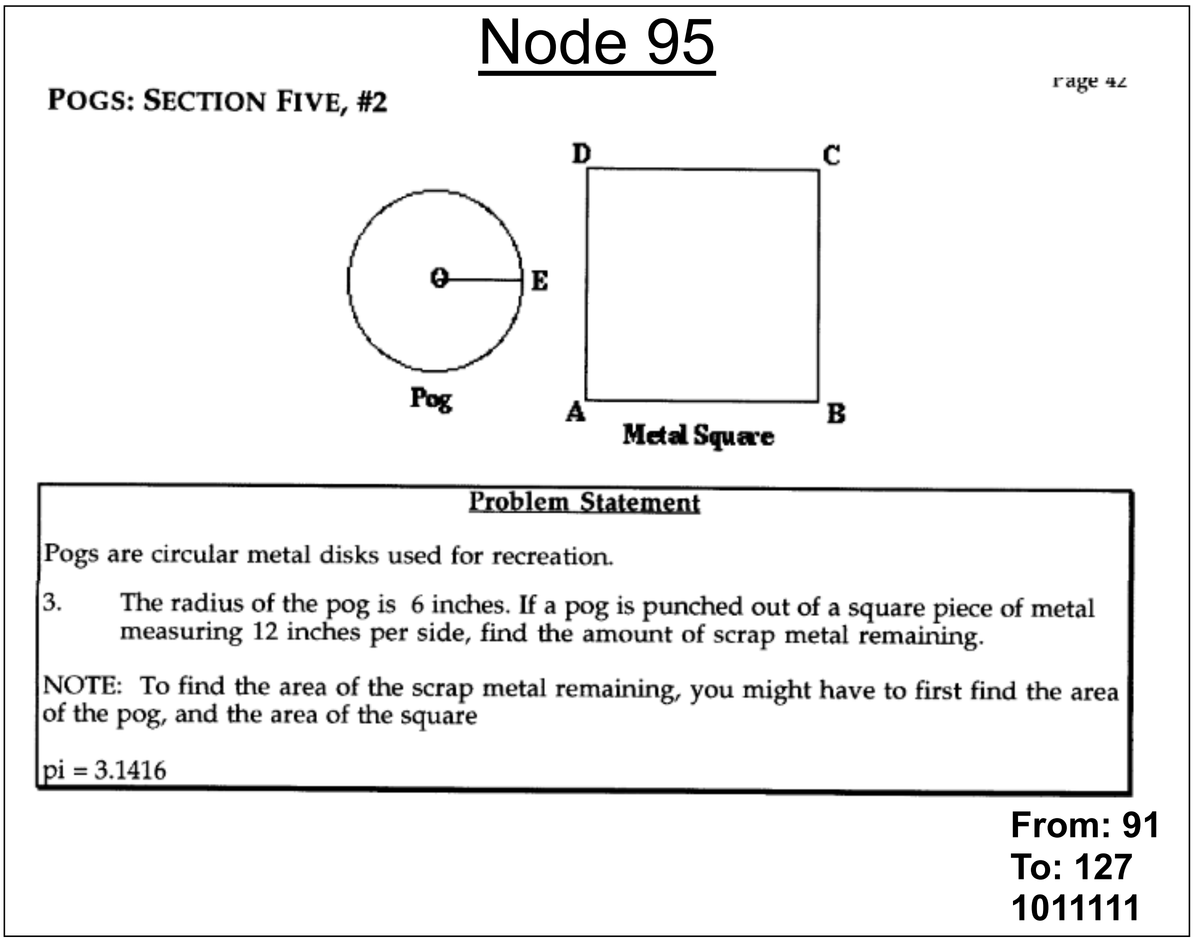
4. If the circumference of Circle C is 112 * pi cm, find the diameter, the radius and the area of the circle.

pi = 3.1416

From: 27
To: 95, 123
1011011

Figure 40: Node 91 of the final knowledge graph. This node contains 5 skills.

B.9 Node 95 (1011111)



Node 95

rage 42

POGS: SECTION FIVE, #2

D
C
O
E
Pog
A
B
Metal Square

Problem Statement

Pogs are circular metal disks used for recreation.

3. The radius of the pog is 6 inches. If a pog is punched out of a square piece of metal measuring 12 inches per side, find the amount of scrap metal remaining.

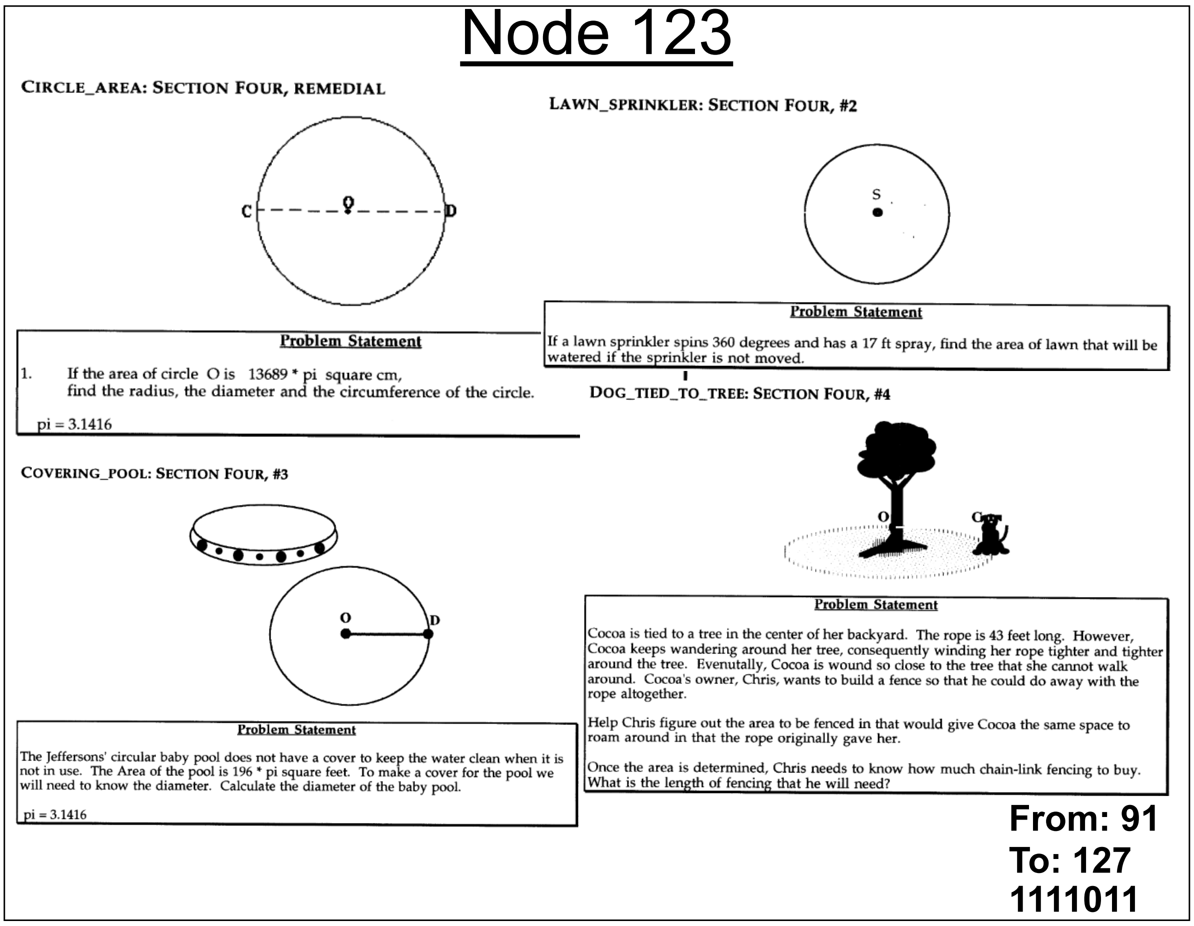
NOTE: To find the area of the scrap metal remaining, you might have to first find the area of the pog, and the area of the square

pi = 3.1416

From: 91
To: 127
1011111

Figure 41: Node 95 of the final knowledge graph. This node contains 6 skills.

B.10 Node 123 (`1111011`)



Node 123

CIRCLE_AREA: SECTION FOUR, REMEDIAL

Problem Statement

1. If the area of circle O is 13689 * pi square cm,
   find the radius, the diameter and the circumference of the circle.

pi = 3.1416

LAWN_SPRINKLER: SECTION FOUR, #2

Problem Statement

If a lawn sprinkler spins 360 degrees and has a 17 ft spray, find the area of lawn that will be watered if the sprinkler is not moved.

COVERING_POOL: SECTION FOUR, #3

Problem Statement

The Jeffersons' circular baby pool does not have a cover to keep the water clean when it is not in use. The Area of the pool is 196 * pi square feet. To make a cover for the pool we will need to know the diameter. Calculate the diameter of the baby pool.

pi = 3.1416

DOG_TIED_TO_TREE: SECTION FOUR, #4

Problem Statement

Cocoa is tied to a tree in the center of her backyard. The rope is 43 feet long. However, Cocoa keeps wandering around her tree, consequently winding her rope tighter and tighter around the tree. Evenutally, Cocoa is wound so close to the tree that she cannot walk around. Cocoa's owner, Chris, wants to build a fence so that he could do away with the rope altogether.

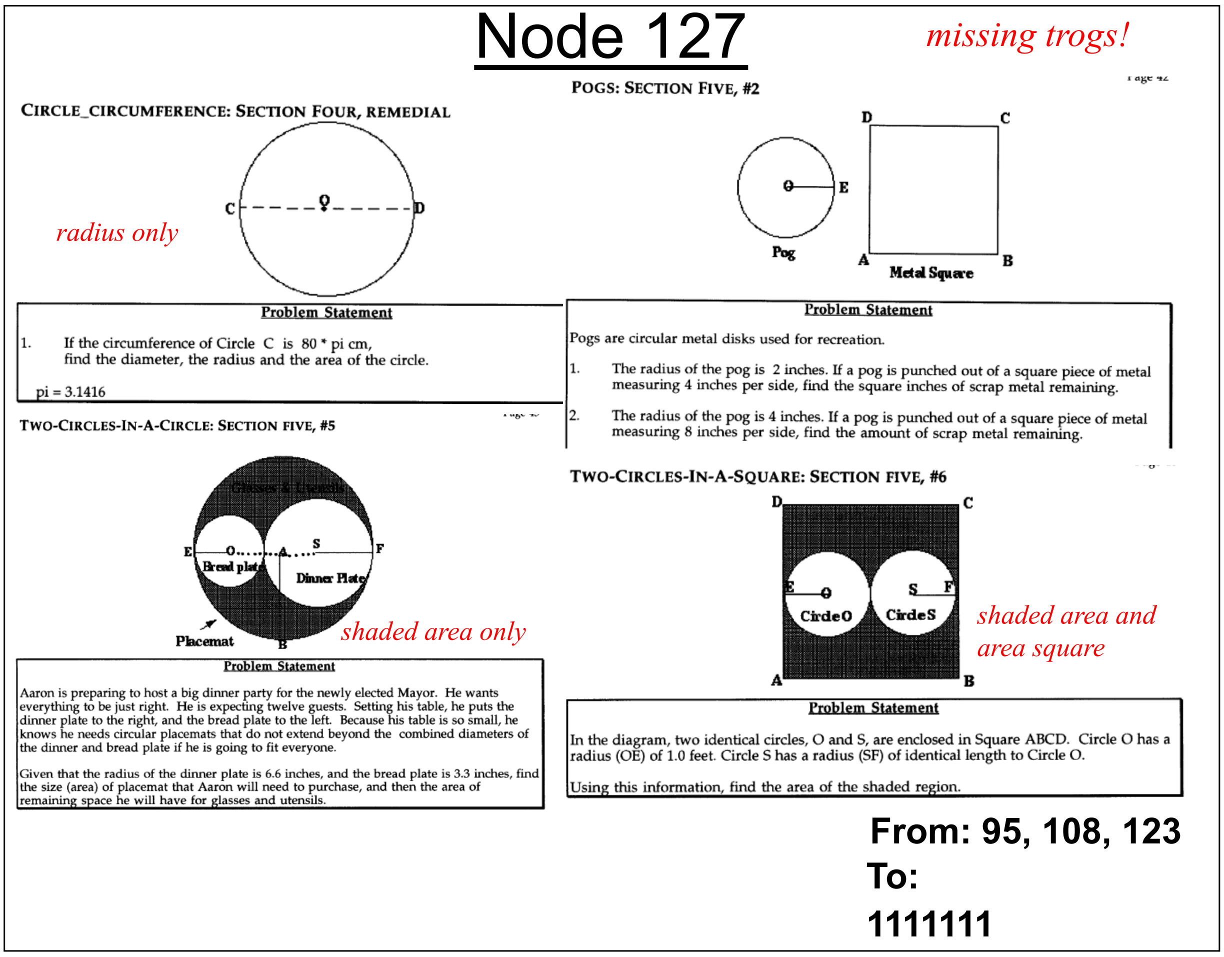
Help Chris figure out the area to be fenced in that would give Cocoa the same space to roam around in that the rope originally gave her.

Once the area is determined, Chris needs to know how much chain-link fencing to buy. What is the length of fencing that he will need?

From: 91
To: 127
1111011

Figure 42: Node 123 of the final knowledge graph. This node contains 6 skills.

B.11 Node 127 (1111111)



Node 127

missing trogs!

CIRCLE_CIRCUMFERENCE: SECTION FOUR, REMEDIAL

radius only

Problem Statement

1. If the circumference of Circle C is 80 * pi cm, find the diameter, the radius and the area of the circle.

pi = 3.1416

TWO-CIRCLES-IN-A-CIRCLE: SECTION FIVE, #5

shaded area only

Problem Statement

Aaron is preparing to host a big dinner party for the newly elected Mayor. He wants everything to be just right. He is expecting twelve guests. Setting his table, he puts the dinner plate to the right, and the bread plate to the left. Because his table is so small, he knows he needs circular placemats that do not extend beyond the combined diameters of the dinner and bread plate if he is going to fit everyone.

Given that the radius of the dinner plate is 6.6 inches, and the bread plate is 3.3 inches, find the size (area) of placemat that Aaron will need to purchase, and then the area of remaining space he will have for glasses and utensils.

POGS: SECTION FIVE, #2

Problem Statement

Pogs are circular metal disks used for recreation.

1. The radius of the pog is 2 inches. If a pog is punched out of a square piece of metal measuring 4 inches per side, find the square inches of scrap metal remaining.
2. The radius of the pog is 4 inches. If a pog is punched out of a square piece of metal measuring 8 inches per side, find the amount of scrap metal remaining.

TWO-CIRCLES-IN-A-SQUARE: SECTION FIVE, #6

shaded area and area square

Problem Statement

In the diagram, two identical circles, O and S, are enclosed in Square ABCD. Circle O has a radius (OE) of 1.0 feet. Circle S has a radius (SF) of identical length to Circle O.

Using this information, find the area of the shaded region.

From: 95, 108, 123
To:
1111111

Figure 43: Node 127 of the final knowledge graph. The red text specifies which steps are (or are not) included in the node. This node contains all skills in the data set.

C Model Code

C.1 Model 0: Version 1

```
// Model: Painting the wall (find the shaded area)
// Written by: I.D.M. Akrum
// Date: 07/03/2022

define task shaded-area {
  initial-goals: (read-task)
  default-activation: 1.0 // All chunks defined in this model receive a
     fixed baselevel activation of 1.0
  ol: t
  rt: -2.0
  lf: 0.2
  default-operator-self-assoc: 0.0
  egs: 0.05
  retrieval-reinforces: t
}

// read the task from the screen and make it your current goal
define goal read-task {
   operator read {
      V1 = screen
      V4 <> nil
   ==>
      V4 -> G1
      >>V3 // shift focus to the first shape
   }
}

// Write down a previously-found answer and reset the goal and WM
define goal do-write {
   operator write-answer {
      G1 = write
   ==>
      write -> AC1
      *task -> AC2
      *answer -> AC3
      nil -> G1
      nil -> WM0
   }
}

// Find a shaded area, i.e. the area of a shape minus the other areas
   within that shape
define goal shaded-area {
   // Find the area of the top shape
   operator top-area {
      G1 = shaded-area
      V1 = rectangle
      V3 <> nil
      V6 <> nil
   ==>
      V6 -> WM1
      V3 -> WM2
      >>V3
   }

   // If an area is missing, find that first
   operator missing-area {
      G1 = shaded-area
      V1 = rectangle
      V6 = nil
   ==>
      area -> G1
      nil -> WM0
   }

   // Subtract intermediate areas from the top area
   operator subtract-area {
      G1 = shaded-area
      WM1 <> nil
      WM2 = V0
      V1 = rectangle
      V6 <> nil
   ==>
      V6 -> WM2
      subtract-fact -> *fact-type
   }

   // Move on to the next shape within the top shape
   operator next-shape {
      G1 = shaded-area
      WM1 <> nil
      WM2 = nil
      V2 <> nil
   ==>
      V2 -> WM2
      >>V2
   }

   // This runs if the next V2-level item is not a shape but e.g. a base
      or height
   operator not-a-shape {
      G1 = shaded-area
      WM1 <> nil
      WM2 = V0
      V1 <> rectangle
   ==>
```

```
        nil -> WM2
    }

    // Keep WM1 as the total of the shaded-area calculations
    operator update-shaded-area {
        G1 = shaded-area
        WM3 <> nil
    ==>
        WM3 -> WM1
        nil -> WM2
        nil -> WM3
    }

    // Finish the shaded area calculation and signal the writing action
    operator end-shaded-area {
        G1 = shaded-area
        WM1 <> nil
        WM2 = nil   // ensures this only happens after an update-shaded-area call
        V2 = nil
    ==>
        V<<   // return to the top shape
        G1 -> *task
        WM1 -> *answer
        write -> G1
    }
}

// Calculate the area of a rectangle
define goal area {
    // If we have the base and height, multiply them through the math skill
    operator base-times-height {
        G1 = area
        V4 <> nil
        V5 <> nil
        WM1 = nil
        WM2 = nil
    ==>
        V4 -> WM1
        V5 -> WM2
        multiply-fact -> *fact-type
    }

    // If we have the answer in the imaginal buffer, signal the writing action
    operator end-area {
        G1 = area
        WM3 <> nil
    ==>
```

```
      G1 -> *task
      WM3 -> *answer
      write -> G1
   }

   // If we miss the base, we switch to the base skill to determine the
      base
   operator missing-base {
      G1 = area
      V4 = nil
      V3 <> nil
   ==>
      base -> G1
      nil -> WM0
      >>V3
   }
}

// Find the base of a rectangle, where the base is given by known lines
   and/or other shapes
define goal base {
   // Initialise the base as the first known line we come across
   operator first-base {
      G1 = base
      WM1 = nil
      V1 <> height // we are only interested in lines that make up the
         base of the shape
      V4 <> nil
   ==>
      V4 -> WM1
   }

   // Move on to the next V2-level item if it exists
   operator next-base {
      G1 = base
      WM1 <> nil
      WM2 = nil
      V2 <> nil
   ==>
      V2 -> WM2    // temporary placeholder to show another number will
         come
      >>V2
   }

   // Add the base of a line to that in the WM1 slot
   operator add-base {
      G1 = base
      WM1 <> nil
      WM2 = V0
      V1 <> height
      V4 <> nil
   ==>
      V4 -> WM2
      add-fact -> *fact-type
   }

   // Keep WM1 as the total of the base calculations
   operator update-base {
      G1 = base
      WM3 <> nil
   ==>
      WM3 -> WM1
      nil -> WM2
      nil -> WM3
   }

   // If there are no more lines/shapes, write down the total base
   operator end-base {
      G1 = base
      WM1 <> nil
      WM2 = nil   // ensures this only happens after an update-base call
      V2 = nil
   ==>
      V<<   // return to the top shape
      G1 -> *task
      WM1 -> *answer
      write -> G1
   }
}

// Do some math with slots in the imaginal buffer
// Doing math is simply "remembering" the answer
define goal math {
   // start when there's two numbers in the imaginal buffer but no answer
   operator start-math {
      WM1 <> nil
      WM2 <> nil
      WM3 = nil
      RT1 = nil   // Don't start math if we already retrieved something
   ==>
      *fact-type -> RT1 // fact-type specifies which math operation we're
          doing e.g. addition or multiplication
      WM1 -> RT2
      WM2 -> RT3
   }

   // End math if we retrieved the answer. Place the answer in WM3 of
      imaginal buffer
   operator end-math {
      RT4 <> nil
      WM3 = nil
   ==>
      RT4 -> WM3
   }
}

define action write {
   latency: 0.3 // min-fitts-movement 0.1 + prep-time 0.05 + init time 0.
      05 + punch 2*0.05 (2 burst-movements)
   noise: 0.1
   distribution: uniform
   output: Writing
}

// Contains all the math we need to know to solve the problems
define facts {
   (mf1 multiply-fact 7 17.5 122.5)
   (mf2 multiply-fact 20 22.5 450)
   (af1 add-fact 3 7 10)
   (af2 add-fact 10 10 20)
   (mf3 multiply-fact 8 20 160)
   (mf4 multiply-fact 20 25 500)
   (af3 add-fact 4 8 12)
   (af4 add-fact 12 8 20)
   (af5 add-fact 2.5 8 10.5)
   (af6 add-fact 10.5 10 20.5)
   (mf5 multiply-fact 20.5 25 512.5)
   (sf1 subtract-fact 450 122.5 327.5)
   (sf2 subtract-fact 500 160 340)
   (sf3 subtract-fact 512.5 160 352.5)
}

// Define the visual with a graph representation. V2 holds same-level
   items. V3 holds lower-level items
define visual {
   // Screen for question 1
   (screen1 screen nil rect1 shaded-area)
   (rect1 rectangle nil base1 nil 22.5 nil)
   (base1 base rect2 nil 3)
   (rect2 rectangle base2 nil 7 17.5 nil)
   (base2 base nil nil 10)
   // Screen for question 2
   (screen2 screen nil rect3 shaded-area)
   (rect3 rectangle nil base3 nil 25 nil)
   (base3 base rect4 nil 4)
   (rect4 rectangle base4 nil 8 20 nil)
   (base4 base nil nil 8)
   // Screen for question 3
   (screen3 screen nil rect5 shaded-area)
   (rect5 rectangle nil base5 nil 25 nil)
   (base5 base rect6 nil 2.5)
   (rect6 rectangle base6 nil 8 20 nil)
   (base6 base nil nil 10)
}

define script {
   screens = ["screen1", "screen2", "screen3"]
   current_screen = 1

   // Run until we've seen all screens (i.e. done all questions)
   while(current_screen != 4) {
      trial-start()
      print("Question␣", current_screen, ":␣")
      print("Find␣the␣area␣of␣the␣wall␣to␣be␣painted.␣Do␣not␣paint␣the␣door.")
      screen(screens[current_screen - 1])
      done = 1

      // Run until the shaded-area has been written
      while (done) {
         run-until-action("write")
         ac = last-action()
         if(ac[1] == "base") {
            set-buffer-slot("input", "slot4", ac[2]) // Update V4 (the base) of the rectangle currently in visual
         }

         if(ac[1] == "area") {
            set-buffer-slot("input", "slot6", ac[2])  // Update V6 (the area) of the rectangle currently in visual
         }

         if(ac[1] == "shaded-area") {
            done = 0 // Signal script we are done with this question
         } else {
            screen(screens[current_screen - 1]) // Make the top-level of the visicon (the screen) the visual focus
         }
      }
      issue-reward()
      purge-bindings()
      trial-end()
      current_screen = current_screen + 1
   }
}
```

C.2 Model 0: Version 2

```
// Model: Node 0 - v4 (final)
// Written by: I.D.M. Akrum
// Date: 11/03/2022

define task node0 {
  initial-goals: (read-task)
  default-activation: 1.0
  ol: t
  rt: -2.0
  lf: 0.2
  default-operator-self-assoc: 0.0
  egs: 0.05
  retrieval-reinforces: t
   //bindings-in-dm: t // if bindings are stored to dm, their activation
      drops too low and the model doesn't work because operator
      conditions don't match
}

// read the task from the screen and make it your current goal
define goal read-task {
   // A simple goal assumes the top shape is the focus
   operator read-simple-goal {
      G1 = read-task
      V1 = screen
      V4 <> nil
      V5 = nil
   ==>
      V4 -> G1
      >>V3
   }

   // A complex goal has a shape specified as focus (e.g. area rect5 asks
       to find the area of rect5 in the visicon)
   operator read-complex-goal {
      G1 = read-task
      V1 = screen
      V4 <> side
      V5 <> nil
   ==>
      search-visual -> G1  // Before we get to the top goal, we first
         need to find the specified shape
      V4 -> G2 // The top goal is stored in G2
      V5 -> WM1   // The specified shape gets placed in WM1 so that we
         can iterate until it's found
      >>V3 // shift focus to the first shape
   }

   // If the goal is to find a side, V5 will specify which side we are
      interested in
   operator read-side {
      G1 = read-task
      V1 = screen
      V4 = side
      V5 <> nil
   ==>
      V4 -> G1
      V5 -> *side // save the specified side as a constant
      >>V3 // shift focus to the first shape
   }
}

// This skill is used for searching the visual until a specified shape is
    found
// It relies on the one-level-down and next-item operators from the
   iterate-over skill
define goal search-visual {
   // If the shape is reached, clear the working memory so that
      associated goal skill can fire
   operator reached-shape {
      G1 = search-visual
      WM1 = WM2
   ==>
      nil -> WM1
      nil -> WM2
      G2 -> G1 // Move on to the top goal
      nil -> G2
   }

   // If the shape isn't reached, clear only WM2 so that the iterate-over
       skill can continue firing
   operator not-the-shape {
      G1 = search-visual
      WM1 <> WM2
   ==>
      nil -> WM2
   }
}

// Write down a previously-found answer and reset the goal and WM
define goal write {
   operator write-answer {
      G1 = write
   ==>
      write -> AC1
      *task -> AC2
      *answer -> AC3
      nil -> G1
      nil -> WM0
   }
}

// This skill iterates over items in the visicon
define goal iterate-over {
   operator first-item {
      G1 <> nil
      WM1 = nil
      *item <> none
   ==>
      *item -> WM1
   }

   operator second-item {
      G1 <> nil
      WM1 <> nil
      WM2 = V0
      *item <> none
   ==>
      *item -> WM2
   }

   // Move down to a level by moving to the V3 slot in the visual chunk
   operator one-level-down {
      G1 <> nil
      V3 <> nil
      V2 = nil // Only go down a level if there's no items at the same
         level
      WM1 <> nil
      WM2 = nil
   ==>
      >>V3
      V0 -> WM2   // Place V0 in WM2 to show we are looking at a new
         visual chunk that needs to be evaluated
      none -> *item
   }

   // Move to the next item at the V2 level
   operator next-item {
      G1 <> nil
      V2 <> nil
      WM1 <> nil
      WM2 = nil
   ==>
      >>V2
      V0 -> WM2   // Place V0 in WM2 to show we are looking at a new
         visual chunk that needs to be evaluated
      none -> *item
   }

   // Keep WM1 as the total of an iterative calculation
   operator update-wm {
      G1 <> nil
      WM3 <> nil
   ==>
      WM3 -> WM1
      nil -> WM2
      nil -> WM3
   }

   // Finish iteration if there's no more V2 or V3 level items
   operator end-iterate {
      G1 <> nil
      WM1 <> nil
      WM2 = nil   // Only finish iteration after update-wm
      V2 = nil
      V3 = nil
      *done-iterate = no
   ==>
      yes -> *done-iterate
      none -> *item
   }
}

// This skill has the knowledge of how to do multiplication
define goal multiply {
   operator multiply-action {
      WM1 <> nil
      WM2 <> V0
      WM3 = nil
      *action = multiply
      *fact-type = none
   ==>
      multiply-fact -> *fact-type
   }
}

// This skill has the knowledge of how to do subtraction
define goal subtract {
   operator substract-action {
      WM1 <> nil
      WM2 <> V0
      WM3 = nil
      *action = subtract
      *fact-type = none
   ==>
      subtract-fact -> *fact-type
   }
}
```

```

// This skill has the knowledge of how to do addition
define goal add {
   operator add-action {
      WM1 <> nil
      WM2 <> V0
      WM3 = nil
      *action = add
      *fact-type = none
   ==>
      add-fact -> *fact-type
   }
}

// Find a shaded area, i.e. the area of a shape minus the other areas
   within that shape
define goal shaded-area {
   // When we start this goal, set the action to subtract
   operator init-shaded-area {
      G1 = shaded-area
      V1 = shape
      *action = none
      *done-iterate = na
   ==>
      subtract -> *action
      no -> *done-iterate  // We begin iterating and aren't finished yet
   }

   // Find the area of a shape and put it in item
   operator area-of-shape {
      G1 = shaded-area
      V1 = shape
      V7 <> nil
      *item = none
      *done-iterate = no
   ==>
      V7 -> *item
   }

   // If an area is missing, find that first
   operator missing-area {
      G1 = shaded-area
      V1 = shape
      V7 = nil
      *done-iterate = no
   ==>
      area -> G1
      nil -> WM0
      none -> *action   // Reset the action since we're changing goals
      na -> *done-iterate  // Set iterate to NA since we halt the iterate
          process
   }

   // This skill fires when a visual chunk isn't a shape
   operator not-a-shape {
      G1 = shaded-area
      V1 <> shape
      WM2 = V0
      *item = none
      *done-iterate = no
   ==>
      nil -> WM2  // Clear the WM2 slot to continue iteration
   }

   // Finish the shaded area calculation and signal the writing action
   operator end-shaded-area {
      G1 = shaded-area
      WM1 <> nil
      *done-iterate = yes
   ==>
      V<<   // Return from the iteration to the top shape
      G1 -> *task
      WM1 -> *answer
      write -> G1
      none -> *action   // We're done with this goal, so reset the action
      na -> *done-iterate  // We're done with the goal, so we're no
         longer in an interation process
   }
}

// Calculate the area of a rectangle
define goal area {
   // Get the base of the focus shape
   operator base-of-shape {
      G1 = area
      V1 = shape
      V5 <> nil
      WM1 = nil
   ==>
      V5 -> WM1
   }

   // Get the height of the focus shape
   operator height-of-shape {
      G1 = area
      V1 = shape
      V6 <> nil
      WM2 = nil
   ==>
```

```
        V6 -> WM2
    }

    // If we have the base and height, signal we want to multiply them through the action constant
    operator base-times-height {
        G1 = area
        WM1 <> nil
        WM2 <> nil
        WM3 = nil
        *action = none
    ==>
        multiply -> *action
    }

    // If we have the answer in the imaginal buffer, signal the writing action
    operator end-area {
        G1 = area
        WM3 <> nil
    ==>
        G1 -> *task
        WM3 -> *answer
        write -> G1
        none -> *action    // Reset the action since we are finished with the goal
    }

    // If we miss the base, we switch to the add-segments skill to determine the base
    operator missing-base {
        G1 = area
        V1 = shape
        V5 = nil
        V3 <> nil
        WM1 = nil
    ==>
        segments -> G1
        base -> *side  // Set the focus to base
        add -> *action
        nil -> WM0
        >>V3  // Move down to the segments that make up this shape
    }

    // If we miss the height, we switch to the add-segments skill to determine the height
    operator missing-height {
        G1 = area
        V1 = shape
        V6 = nil
```

```
      V3 <> nil
      WM2 = nil
   ==>
      segments -> G1
      height -> *side   // Set the focus to height
      add -> *action
      nil -> WM0
      >>V3  // Move down to the segments that make up this shape
   }
}

// Skill for finding a side. Since we're missing a divide skill, the only
    way to find a side is through segments
define goal side {
   // Find a side by adding segments
   operator side-by-segments {
      G1 = side
      V7 = nil
      V3 <> nil
   ==>
      segments -> G1
      add -> *action
      nil -> WM0
      >>V3
   }
}

// Find the specified side of a shape where the side is given by known
   segments (i.e. lines or other shapes)
// Functions virtually the same as shaded-area, but looks at different
   slots and has a different action
define goal segments {
   operator init-segments {
      G1 = segments
      *action <> none
      *done-iterate = na
   ==>
      no -> *done-iterate
   }

   // If we're trying to find a base, read the width of a segment/shape
   operator read-width {
      G1 = segments
      *side = base
      *item = none
      *done-iterate = no
      V5 <> nil
   ==>
      V5 -> *item
   }
```

```

    // If we're trying to find a height, read the height of a segment/
       shape
    operator read-height {
       G1 = segments
       *side = height
       *item = none
       *done-iterate = no
       V6 <> nil
    ==>
       V6 -> *item
    }

    // Clear the WM2 if our current segment isn't the side we're
       interested in (e.g. a height when we're looking at bases)
    operator wrong-segment {
       G1 = segments
       *item = none
       *done-iterate = no
       V1 = segment
       V4 <> *side
       WM2 = V0
    ==>
       nil -> WM2
    }

    // If there are no more lines/shapes, write down the total of the
       added segments
    operator end-segments {
       G1 = segments
       WM1 <> nil
       *done-iterate = yes
    ==>
       V<<   // Return from the iteration to the top shape
       *side -> *task
       WM1 -> *answer
       write -> G1
       none -> *action
       na -> *done-iterate
    }
}

// Do some math with slots in the imaginal buffer
// Doing math is simply "remembering" the answer
define goal math {
    // start when there's two numbers in the imaginal buffer but no answer
    operator start-math {
       WM1 <> nil
       WM2 <> nil
       WM3 = nil
```

```
      RT1 = nil   // Don't start math if we already retrieved something
      *fact-type <> none // Only start math if we know the fact-type
   ==>
      *fact-type -> RT1 // fact-type specifies which math operation we're
          doing e.g. addition or multiplication
      WM1 -> RT2
      WM2 -> RT3
   }

   // End math if we retrieved the answer. Place the answer in WM3 of
      imaginal buffer
   operator end-math {
      RT4 <> nil
      WM3 = nil
   ==>
      RT4 -> WM3
      none -> *fact-type   // Clear the fact-type since we finished this
         math-action
   }
}

// The write action approximates punching keys on the key board
// Since there is no default hand position, it assumes the ACT-R min-
   fitts-movement variable (default 0.1) as movement time
define action write {
   latency: 0.4 // min-fitts-movement 0.1 + prep-time 0.05 + init time 0.
      05 + punch 4*0.05 (4 burst-movements equals 2 keys)
   noise: 0.1  // Allows us to reach minimum of 1 key (2 burst-movements
      of 0.05) and maximum of 3 keys (6 burst-movements of 0.05)
   distribution: uniform
   output: Writing
}

// Contains all the math we need to know to solve the problems
define facts {
   (mf1 multiply-fact 7 17.5 122.5)
   (mf2 multiply-fact 20 22.5 450)
   (af1 add-fact 3 7 10)
   (af2 add-fact 10 10 20)
   (mf3 multiply-fact 8 20 160)
   (mf4 multiply-fact 20 25 500)
   (af3 add-fact 4 8 12)
   (af4 add-fact 12 8 20)
   (af5 add-fact 2.5 8 10.5)
   (af6 add-fact 10.5 10 20.5)
   (mf5 multiply-fact 20.5 25 512.5)
   (sf1 subtract-fact 450 122.5 327.5)
   (sf2 subtract-fact 500 160 340)
   (sf3 subtract-fact 512.5 160 352.5)
}
```

```

// Define the visual with a graph representation. V2 holds same-level
   items. V3 holds lower-level items
// V4 is a further specification of the V1 type. V5 is width, V6 is
   height. V7 is area for shapes
define visual {
   // Screen for question 1
   (screen1 screen nil rect1 shaded-area)
   (rect1 shape nil base1 rectangle nil 22.5 nil)
   (base1 segment rect2 nil base 3 nil)
   (rect2 shape base2 nil rectangle 7 17.5 nil)
   (base2 segment nil nil base 10 nil)
   // Screen for question 2
   (screen2 screen nil rect3 shaded-area)
   (rect3 shape nil base3 rectangle nil 25 nil)
   (base3 segment rect4 nil base 4 nil)
   (rect4 shape base4 nil rectangle 8 20 nil)
   (base4 segment nil nil base 8 nil)
   // Screen for question 3
   (screen3 screen nil rect5 shaded-area)
   (rect5 shape nil base5 rectangle nil 25 nil)
   (base5 segment rect6 nil base 2.5 nil)
   (rect6 shape base6 nil rectangle 8 20 nil)
   (base6 segment nil nil base 10 nil)
   // Screen for question 4 (extra to check that the search-visual goal
      works)
   (screen4 screen nil rect3 area rect4)
   // Screen for question 5 (extra to check that the side skill works)
   (screen5 screen nil rect7 side base)
   (rect7 shape nil base5 rectangle nil 25 nil)
}

define script {
   screens = ["screen1", "screen2", "screen3"]
   current_screen = 1

   // Run until we’ve seen all screens (i.e. done all questions)
   while(current_screen != 4) {
      trial-start()

      // Initialise the bindings
      add-binding("item","none")
      add-binding("action","none")
      add-binding("fact-type","none")
      add-binding("done-iterate", "na")

      // Print information on screen for modeller
      print("Question␣", current_screen, ":␣")
      print("Find␣the␣area␣of␣the␣wall␣to␣be␣painted.␣Do␣not␣paint␣the␣
         door.")
      screen(screens[current_screen - 1])
      done = 1

      // Run until the shaded-area has been written
      while (done) {
         run-until-action("write")
         ac = last-action()
         if(ac[1] == "base") {
            set-buffer-slot("input", "slot5", ac[2]) // Update V5 (the base) of the rectangle currently in visual
         }

         if(ac[1] == "area") {
            set-buffer-slot("input", "slot7", ac[2])  // Update V7 (the area) of the rectangle currently in visual
         }

         screen(screens[current_screen - 1]) // Make the top-level of the visicon (the screen) the visual focus
         goal = get-buffer-slot("input", "slot4")
         sub-goal = get-buffer-slot("input", "slot5")

         // Check if task is complete, if not, read the task and continue with new information
         if((ac[1] == goal) || (ac[1] == sub-goal)) {
            done = 0 // Signal script we are done with this question
         } else {
            set-buffer-slot("goal", "slot1", "read-task")   // Reset the goal to reading the task
         }
      }
      issue-reward()
      purge-bindings()
      trial-end()
      current_screen = current_screen + 1
   }
}
```

C.3 Model 8

```
// Model: Node 8
// Written by: I.D.M. Akrum
// Date: 14/03/2022

define task node8 {
  initial-goals: (read-task)
  default-activation: 1.0
  ol: t
  rt: -2.0
  lf: 0.2
  default-operator-self-assoc: 0.0
  egs: 0.05
  retrieval-reinforces: t
   //bindings-in-dm: t // if bindings are stored to dm, their activation
      drops too low and the model doesn't work because operator
      conditions don't match
}

// This skill has the knowledge of how to do division
define goal divide {
   operator divide-action {
      WM1 <> nil
      WM2 <> V0
      WM3 = nil
      *action = divide
      *fact-type = none
   ==>
      divide-fact -> *fact-type
   }
}

// Calculate the area of the shapes: rectangle, triangle, and trapezoid
// Given that this skill "overloads" the one from model 0, its operators
   have a higher activation to promote them over the operators from model
    0
define goal area {
   // Since trapezoids don't have a single base, get the total base by
      adding its base1 and base2
   operator add-trapezoid-bases(activation=10.0) {
      G1 = area
      V4 = trapezoid
      WM1 = nil
   ==>
      G1 -> G2 // Save the original goal so it can be returned to
      segments -> G1
      base -> *side  // Set the focus to base
      add -> *action
      nil -> WM0
      >>V3  // Move down to the segments that make up the trapezoid base
   }

   // This skill triggers after the trapezoid bases have been added
   operator base-of-trapezoid(activation=10.0) {
      G1 = area
      WM1 <> nil
      *action = add
      V<<   // Returning from segments, look at the trapezoid again
         rather than the last base
      V4 = trapezoid
   ==>
      none -> *action
   }

   // If we're looking a triangle, divide the previously found area by 2
   operator area-triangle(activation=10.0) {
      G1 = area
      WM3 <> nil
      V4 = triangle
      *action = multiply
   ==>
      WM3 -> WM1
      *two -> WM2
      nil -> WM3
      divide -> *action
   }

   // Do the same for a trapezoid as we do for a triangle
   operator area-trapezoid(activation=10.0) {
      G1 = area
      WM3 <> nil
      V4 = trapezoid
      *action = multiply
   ==>
      WM3 -> WM1
      *two -> WM2
      nil -> WM3
      divide -> *action
   }

   // If we have the answer in WM3 and we're dealing with a rectangle,
      write the answer
   operator end-area-rect(activation=10.0) {
      G1 = area
      WM3 <> nil
      V4 = rectangle
   ==>
      G1 -> *task
      WM3 -> *answer
      write -> G1
      none -> *action   // Reset the action since we are finished with
         the goal
   }

   // If we have the answer in WM3 and we did the area-triangle operator,
       write the answer
   operator end-area-triangle(activation=10.0) {
      G1 = area
      WM3 <> nil
      V4 = triangle
      *action = divide
   ==>
      G1 -> *task
      WM3 -> *answer
      write -> G1
      none -> *action   // Reset the action since we are finished with
         the goal
   }

   // If we have the answer in WM3 and we did the area-trapezoid operator
      , write the answer
   operator end-area-trapezoid(activation=10.0) {
      G1 = area
      WM3 <> nil
      V4 = trapezoid
      *action = divide
   ==>
      G1 -> *task
      WM3 -> *answer
      write -> G1
      none -> *action   // Reset the action since we are finished with
         the goal
   }

   // If we miss the base, we switch to the segments skill to determine
      the base
   operator missing-base(activation=10.0) {
      G1 = area
      V1 = shape
      V4 <> trapezoid   // This doesn't hold for trapezoids since they
         don't have a single base
      V5 = nil
      V3 <> nil
      WM1 = nil
   ==>
      segments -> G1
      base -> *side  // Set the focus to base
      add -> *action
      nil -> WM0
      >>V3  // Move down to the segments that make up this shape
   }
}

// Skill for finding a side.
define goal side {
   // If we're trying to determine the base, read the height
   operator other-side-base {
      G1 = side
      *side = base
      WM2 = nil
      V6 <> nil
   ==>
      V6 -> WM2
   }

   // If we're trying to determine the height, read the base
   operator other-side-height {
      G1 = side
      *side = height
      WM2 = nil
      V5 <> nil
   ==>
      V5 -> WM2
   }

   // Read the two bases of the trapezoid
   operator read-bases-of-trapezoid {
      G1 = side
      *side = height
      V4 = trapezoid
      WM2 = nil
   ==>
      >>V3
      G1 -> G2
      segments -> G1
      base -> *side  // temporarily change the side to the base for the
         addition
      add -> *action
   }

   // After we know the total base for the trapezoid, store it in WM2 as
      per usual
   operator other-side-height-trapezoid {
      G1 = side
      *side = base
      *action = add
      V<< // We are still looking at the last base of the trapezoid, so
         return visual focus to trapezoid
      V4 = trapezoid
```

```
        V7 <> nil
        WM1 <> nil
        WM1 <> V7   // Should trigger if we have a value in WM1, but it's
           not the area
    ==>
        WM1 -> WM2
        nil -> WM1
        height -> *side
        none -> *action
    }

    // Put the area in WM1 and divide WM1 by WM2
    operator area-by-other-side {
        G1 = side
        WM1 = nil
        WM2 <> nil
        V7 <> nil
    ==>
        V7 -> WM1
        divide -> *action
    }

    // If we're dealing with a triangle, we have to multiply the answer
        from the division by 2
    operator side-triangle {
        G1 = side
        WM3 <> nil
        V4 = triangle
        *action = divide  // This triggers after we've divided the area by
           the other side
    ==>
        WM3 -> WM1
        *two -> WM2
        nil -> WM3
        multiply -> *action
    }

    // As with the area, we also multiply by 2 for trapezoids
    operator side-trapezoid {
        G1 = side
        WM3 <> nil
        V4 = trapezoid
        *action = divide
    ==>
        WM3 -> WM1
        *two -> WM2
        nil -> WM3
        multiply -> *action
    }

```

```
   // For a trapezoid, one base will be given. We must subtract the total
       base we found by the base already given
   operator other-base-trapezoid {
      G1 = side
      *side = base
      *action = multiply
      V4 = trapezoid
      WM3 <> nil
   ==>
      G1 -> G2
      segments -> G1
      subtract -> *action
      WM3 -> WM1
      nil -> WM2
      nil -> WM3
   }

   // The side of rectangle is found after the area has been divided by
      the other side
   operator end-side-rect {
      G1 = side
      WM3 <> nil
      V4 = rectangle
      *action = divide
   ==>
      *side -> *task
      WM3 -> *answer
      write -> G1
      none -> *action
   }

   // The side of a triangle is found after we've multiplied the found
      side by two
   operator end-side-triangle {
      G1 = side
      WM3 <> nil
      V4 = triangle
      *action = multiply
      WM2 = *two
   ==>
      *side -> *task
      WM3 -> *answer
      write -> G1
      none -> *action
   }

   // The other base of a trapezoid has been found after the segment
      skill is finished
   // In this case, WM1 will hold the final answer and the action will be
       none
   operator end-base-trapezoid {
      G1 = side
      *side = base
      WM1 <> nil
      *action = subtract
   ==>
      *side -> *task
      WM1 -> *answer
      write -> G1
      none -> *action
   }

   // The height of a trapezoid has been found after we've multiplied the
        found side by two (as triangle)
   operator end-height-trapezoid {
      G1 = side
      WM3 <> nil
      V4 = trapezoid
      *side = height
      *action = multiply
   ==>
      *side -> *task
      WM3 -> *answer
      write -> G1
      none -> *action
   }
}

// Find the specified side of a shape where the side is given by known
   segments (i.e. lines or other shapes)
// Functions virtually the same as shaded-area, but looks at different
   slots and has a different action
define goal segments {
   // If we're interested in bases, but the width of a base isn't given,
      skip it
   operator skip-base {
      G1 = segments
      *side = base
      V4 = base
      V5 = nil
      WM2 = V0
   ==>
      nil -> WM2
   }

   // If we're interested in heights, but the height of a height segment
      isn't given, skip it
   operator skip-height {
      G1 = segments
      *side = height
```

```
        V4 = height
        V6 = nil
        WM2 = V0
    ==>
        nil -> WM2
    }

    // If G2 isn't nil, return to the G2 skill rather than writing down
        the WM1 answer
    operator end-segments-alt(activation=10.0) {
        G1 = segments
        G2 <> nil
        WM1 <> nil
        *done-iterate = yes
    ==>
        G2 -> G1
        nil -> G2
        na -> *done-iterate
    }

    // If there are no more lines/shapes, write down the total of the
        added segments
    operator end-segments(activation=10.0) {
        G1 = segments
        G2 = nil
        WM1 <> nil
        *done-iterate = yes
    ==>
        V<<   // Return from the iteration to the top shape
        *side -> *task
        WM1 -> *answer
        write -> G1
        none -> *action
        na -> *done-iterate
    }
}

// Contains all the math we need to know to solve the problems
define facts {
    (mf6 multiply-fact 43 33 1419)
    (df1 divide-fact 1419 2 709.5)
    (df2 divide-fact 2146 58 37)
    (mf7 multiply-fact 37 2 74)
    (af7 add-fact 31 33 64)
    (mf8 multiply-fact 64 7 448)
    (df3 divide-fact 448 2 224)
    (df4 divide-fact 1425 25 57)
    (mf9 multiply-fact 57 2 114)
    (sf4 subtract-fact 114 56 58)
    (af8 add-fact 28 31 59)
```

```
   (df5 divide-fact 472 59 8)
   (mf10 multiply-fact 8 2 16)
}

// Define the visual with a graph representation. V2 holds same-level
   items. V3 holds lower-level items
// V4 is a further specification of the V1 type. V5 is width, V6 is
   height. V7 is area for shapes
define visual {
   // Screen for question 1 (question 3 from model 0)
   (screen1 screen nil rect1 shaded-area)
   (rect1 shape nil base1 rectangle nil 25 nil)
   (base1 segment rect2 nil base 2.5 nil)
   (rect2 shape base3 nil rectangle 8 20 nil)
   (base3 segment nil nil base 10 nil)
   // Screen for question 2 (triangle_area in node 1)
   (screen2 screen nil triangle1 area)
   (triangle1 shape nil nil triangle 43 33 nil)
   // Screen for question 3 (triangle_height in node 1)
   (screen3 screen nil triangle2 side height)
   (triangle2 shape nil nil triangle 58 nil 2146)
   // Screen for question 4 (trapezoid_area in node 1)
   (screen4 screen nil trapezoid1 area)
   (trapezoid1 shape nil base4 trapezoid nil 7 nil)
   (base4 segment base5 nil base 31 nil)
   (base5 segment nil nil base 33 nil)
   // Screen for question 5 (trapezoid_base in node 1)
   (screen5 screen nil trapezoid2 side base)
   (trapezoid2 shape nil base6 trapezoid nil 25 1425)
   (base6 segment base7 nil base 56 nil)
   (base7 segment nil nil base nil nil)
   // Screen for question 6 (trapezoid_height in node 1)
   (screen6 screen nil trapezoid3 side height)
   (trapezoid3 shape nil base8 trapezoid nil nil 472)
   (base8 segment base9 nil base 28 nil)
   (base9 segment nil nil base 31 nil)
}

define script {
   screens = ["screen1", "screen2", "screen3", "screen6", "screen4", "
      screen5"]
   current_screen = 1

   // Run until we've seen all screens (i.e. done all questions)
   while(current_screen != 7) {
      trial-start()

      // Initialise the bindings
      add-binding("item","none")
      add-binding("action","none")
      add-binding("fact-type","none")
      add-binding("done-iterate", "na")
      add-binding("two", 2)

      screen(screens[current_screen - 1])
      done = 1

      goal = get-buffer-slot("input", "slot4")
      sub-goal = get-buffer-slot("input", "slot5")
      shape = get-buffer-slot("input", "slot3")

      // Determine the shape of interest
      if((goal != "side") && (sub-goal != "nil")) {
         shape = sub-goal
      }

      // Print information on the screen for modeller
      print("Question", current_screen, ":␣")

      if(goal != "side") {
         print("Find␣the", goal, "of", shape)
      } else {
         print("Find␣the", sub-goal, "of", shape)
      }

      // Run until the goal has been reached and the answer written down
      while (done) {
         run-until-action("write")
         ac = last-action()

         if(ac[1] == "base") {
            set-buffer-slot("input", "slot5", ac[2]) // Update V5 (the base) of the current visual chunk
         }

         if(ac[1] == "height") {
            set-buffer-slot("input", "slot6", ac[2]) // Update V6 (the height) of the current visual chunk
         }

         if(ac[1] == "area") {
            set-buffer-slot("input", "slot7", ac[2])  // Update V7 (the area) of the shape currently in visual
         }

         screen(screens[current_screen - 1]) // Make the top-level of the visicon (the screen) the visual focus

         // Check if task is complete, if not, read the task and continue with new information
         if((ac[1] == goal) || (ac[1] == sub-goal)) {
            done = 0 // Signal script we are done with this question
         } else {
            set-buffer-slot("goal", "slot1", "read-task")   // Reset the goal to reading the task
         }
      }
      issue-reward()
      purge-bindings()
      trial-end()
      current_screen = current_screen + 1
   }
}
```