Solving Witness-type Triangle Puzzles Faster with an Automatically Learned Human-Explainable Predicate

by

Justin Daley Stevens

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

The Witness is a game with difficult combinatorial puzzles that are challenging for both human players and artificial intelligence based solvers. Indeed, the number of candidate solution paths to the largest puzzle considered in this thesis is on the order of 10^{15} and search-based solvers can require large amounts of time and memory to solve such puzzles. We accelerate search by automatically learning human-explainable predicates that predict whether a partial path to a Witness-type puzzle is not completable to a solution path. We prove a key property of one of the learned predicates which allows us to use it for pruning successor states in search. Our method accelerates search by an average of six times while maintaining completeness of the underlying search. We also explain how our predicate speeds up search on a specific puzzle instance by over 1,000 times. Conversely given a fixed search time budget per puzzle our predicate-accelerated search can solve more puzzle instances of larger sizes than the baseline search. We also empirically compare the performance of our learned predicate to two popular competitors, weighted A* and Levin tree search with neural networks, and show that our learned predicate outperforms both of them in terms of how much they speed up a baseline.

Preface

Parts of the work presented in this thesis are also described in a technical report (Stevens, Bulitko, and Thue 2023) and was co-authored with my supervisor, Professor Vadim Bulitko, and Professor David Thue of Carleton University. For the technical report, I wrote all of the code, ran all of the experiments, and was the lead author of the report. Vadim had the initial idea of applying inductive logic programming, provided weekly guidance, and re-wrote and provided many edits to the report. David helped discuss the report on a weekly basis, also re-wrote and provided edits to the report, and had the initial idea for the proof of Theorem 4.

To reflect the collaborative nature of the work, I use the word 'we' throughout the thesis. For each chapter of the thesis, I will describe its content adapted from the report as well as my new post-report content:

- I modified Chapter 1 from the introduction section of the report to reflect the additional results of weighted A* and Levin tree search in this thesis.
- I modified Chapter 2 from the problem formulation section of the report extensively with the discussion of the theorems, proofs, pruning, and search trees.
- I modified Chapter 3 from the related work section of the report with new discussion around procedural content generation, SAT solvers, planning, and another logical reasoning system.
- I expanded Chapter 4 from the proposed approach of the report. In particular, in Section 4.2 all of the discussion of using inductive logic

programming for the specific puzzle instance including the background knowledge and positive/negative examples are novel.

- I expanded Chapter 5 from the empirical evaluation from the report. Sections 5.1-5.4, 5.6, and 5.9 are all adapted from the report. I added the results on sorting (Section 5.5), weighted A* (Section 5.7), Levin tree search with neural networks (Section 5.8), and an explanation of local constraint checking in Section 5.9.

- I expanded Chapter 6 from the open questions and future work in the report. I added new discussion around decision trees and portfolio planning with a specific example of a puzzle instance it could be useful on.

- I expanded Chapter 7 of the conclusions from the report to discuss the new results with sorting, weighted A*, and Levin tree search with neural networks.

- I adapted Appendix D from the empirical evaluation of the report. The other appendices are all novel to this thesis.

Dedicated to the memory of my sister, Cassie Stevens. Cassie deeply loved animals and cared about environmental issues. She is dearly missed and through this tribute I hope for her memory to live on.

Acknowledgements

I deeply thank and appreciate the support of my supervisor, Professor Vadim Bulitko. Vadim's belief in my ability to do research has led me to where I am now. I appreciate his encouragement to focus on a problem that I enjoyed and his teaching me the steps of the research process. I am a much more diligent researcher, thinker, and writer because of him, and I thank him a lot.

I am eternally grateful for the constant source of unconditional love and support from my parents, Sherry and Greg, and my grandparents, Richard Daley and Sona Stevens. They taught me to love puzzles and problem solving from a young age which has influenced my entire learning trajectory. They also taught me many important life values including the power of resilience.

Thanks to Professors Levi Santana de Lelis and Csaba Szepesvári for serving on my examining committee. I would also like to thank Professor David Thue whose careful attention to detail and positive encouragements helped me out tremendously. I also thank Professors Andrew Cropper, Nathan Sturtevant, Matthew Guzdial, and Michael Cook for their advice.

I also thank many members of the AI lab and fellow graduate students for their support and advice including Faisal Abutarab, Sergio Poo Hernandez, Shuwei Wang, Paul Saunders, Matthew Gallivan, Eugene Chen, Revan MacQueen, Alex Lewandowski, Andrew Jacobsen, Rohini Das, Sacha Davis, Erfan Miahi, Maliha Sultana, Daniela Teodorescu, Kenneth Tjhia, and Aidan Bush.

I am grateful for the friendship and camaraderie of my friends in Edmonton and my friends in Reno, Nevada who reinspired my love for games.

I acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC), Alberta Innovates and Alberta Advanced Education. I also thank the Digital Research Alliance of Canada.

Contents

1 Introduction **1**

2 Problem Formulation **3**
2.1 The Witness Triangle Puzzle 3
2.2 Search for Solving Witness Puzzles 5
2.3 Baseline 8
2.4 Accelerating the Search 9

3 Related Work **12**

4 Proposed Approach **15**
4.1 Predicate Synthesis Algorithm 15
4.2 Learning Predicates from Puzzle Instances 17

5 Evaluation **21**
5.1 Puzzle Sets and Hyper Parameters for Learning a Predicate . . 21
5.2 A Learned Predicate: π° 22
5.3 Verifiability of π° 23
5.4 Search Acceleration of π° 25
5.5 Sorting with π° 27
5.6 Solving Large Puzzles with π° 27
5.6.1 A Puzzle with a Large Speedup 27
5.6.2 Solving Large Puzzle Instances 29
5.7 Weighted A* 30
5.8 Levin Tree Search with Neural Networks 31
5.9 Learned Predicate without π_{baseline} 34

6 Open Questions & Future Work **36**

7 Conclusions **38**

References **39**

Appendix A Levin Tree Search **42**

Appendix B Prolog Files Used by Popper **45**

Appendix C Example of False Positives with a Predicate **48**

Appendix D Puzzle Generators **50**

Appendix E π° Computed by Prolog **52**

List of Tables

4.1 Predicate building blocks for the background knowledge given to *Popper*. 18
4.2 The background knowledge describing the puzzle in Figure 4.1. 19
4.3 Positive and negative training examples given to *Popper*. 19

5.1 The 15,000 instances in $\mathcal{P}_{\text{test}}$ by puzzle size. Average number of expansions and wall time for the baseline search with π_{baseline} are listed as well. 22
5.2 Output of Algorithm 2: the predicate π°. On the left the predicate is shown in Prolog and on the right pseudocode in Python. 23

List of Figures

2.1 A 1×2 Witness puzzle. The green dashed line solves the puzzle since the solution path intersects the left square once and right square twice which are equal to the number of triangles in their respective squares. The red dotted line is not a solution to the puzzle. 4

2.2 A partial search tree for solving the Witness puzzle in Figure 2.1. The dashed lines show the partial paths for solving the puzzle so far. The two vertex markers $\star, \heartsuit$ mark the start and goal vertices of the grid. 8

4.1 Augmented version of the puzzle from Figure 2.1 17

5.1 Portion of a Witness puzzle with a partial path for the `one(F)` case drawn in a red dotted line and for the `two(F)` case drawn in a blue dashed line. 24

5.2 Time and expansion speedup of π° on $\mathcal{P}_{\text{test}}$. 26

5.3 A 5×5 Witness puzzle. The vertices are numbered to show the partial path that pruning with π° forces Algorithm 1 to take. The two vertex markers $\star, \heartsuit$ mark the start and goal vertices. . 28

5.4 Rounded percentage of puzzle instances solved by Algorithm 1 with pruning using π_{baseline} (left) and with π° (right). 29

5.5 Time speedup of solving the problems in $\mathcal{P}_{\text{train+filter}}$ using weighted A* with various weights. 31

5.6 Cumulative distribution of puzzle instances with expansion speedups for Algorithm 1 with pruning using π° compared to weighted A* using w° and PHS* using ANN. 33

5.7 Expansion speedup of pruning using π° compared to PHS* using ANN-50k on $\mathcal{P}_{\text{test}}$. 34

6.1 On the left is an initial partial path to complete the column of 2-triangle squares. On the right is a full solution to the puzzle. 37

A.1 Search tree with a policy for solving the puzzle from Figure 2.1. 42

C.1 An example of a partial path on which π^{ex} has a false positive. The red dashed line shows the partial path that π^{ex} incorrectly predicts as incompletable. 49

Chapter 1

Introduction

The Witness is a well known game with challenging combinatorial puzzles (Thekla, Inc. 2016). Solving its puzzles automatically with artificial intelligence (AI) can be useful both to players and to puzzle designers. Indeed both players and puzzle designers can use such an AI puzzle solver to see a solution to a particularly challenging puzzle instance. Designers can also use the AI solver in concert with a procedural puzzle generator to generate puzzles with desired solution qualities (De Kegel and Haahr 2019).

Various search techniques can be applied to Witness puzzles. For instance, in the triangle Witness-type puzzles that we use for our testbed in this thesis (Figure 2.1), a complete search can enumerate all possible paths for smaller puzzles (Sturtevant 2023). However, for puzzles of larger sizes such search becomes intractable. For instance, for the puzzle size of 8×8, the number of candidate solution paths from the bottom left to the top right corner is on the order of 10^{15} (Iwashita, Kawahara, and Minato 2012). Note that arbitrary-sized, triangle Witness-type puzzles are NP-complete (Abel et al. 2020).

To scale up search to larger instances of Witness puzzles, we automatically learn a predicate (i.e., a binary function) that predicts whether a partial path considered during search can be completed to a solution path. We use such a predicate to focus the search on more promising partial paths. We compare the effectiveness of our machine-learned predicate with a human-designed predicate that implements a simple strategy for pruning partial paths that violate local constraints. We also compare our machine-learned predicate to two

other approaches, weighted A* (Pohl 1970) and Levin tree search with neural networks (Orseau and Lelis 2021).

We automatically learn predicates from training data using an off-the-shelf Inductive Logic Programming (ILP) system, *Popper* (Cropper and Morel 2021). The learned predicates are human readable which can shed light on the puzzle structure and be useful to game designers.

This thesis makes the following contributions. First, we present a method for machine-learning predicates from triangle Witness-type puzzles. Second, we demonstrate human explainability of the learned predicate. Third, the human explainability allows us to prove a key property of the predicate which in turn enables its use for pruning while maintaining completeness. In other words, given enough time and memory, our predicate-accelerated search will solve any triangle Witness-type puzzle that is solvable. Fourth, we empirically evaluate our approach and present the substantial performance gains over the baseline on puzzle instances we generated. We show our predicate-accelerated search also speeds up search more on our testing set than both weighted A* and Levin tree search with neural networks. We finally show that running our learning process with less human knowledge causes it to learn the missing human knowledge.

Finally we make our code and data available upon request to the community in the hope of introducing the triangle Witness-type puzzles as a standard benchmark to the community of Artificial Intelligence and Procedural Content Generation via Machine Learning (Summerville et al. 2018) game researchers.

Chapter 2

Problem Formulation

We first formally define the *Witness-type triangle puzzle* considered in this thesis. We then describe how heuristic search can be used to solve such puzzle instances with predicates to accelerate search. Finally, we describe performance measures to test efficacy of these predicates.

2.1 The Witness Triangle Puzzle

In the commercial video game, *The Witness* (Thekla, Inc. 2016), triangle puzzles are scattered throughout the open world environment for players to find and solve. They also play a part of the final challenge to players. In this section, we formally describe the rules of triangle puzzles from the game. Note that the puzzles considered in this thesis are computer generated.

An $m \times n$ *Witness-type triangle puzzle* instance $p = (G, v_{\text{start}}, v_{\text{goal}}, C)$ is a single-player combinatorial puzzle using the constraint in the triangle puzzle from *The Witness*; we refer to them henceforth as *Witness puzzles*.

Witness puzzles are played on a two-dimensional rectangular grid with $m \times n$ squares. Here, $G = (V, E)$ is a graph representing the grid. Each vertex $v \in V$ is a vertex on the grid where each corner of a puzzle square is situated at a vertex. An $m \times n$ puzzle is thus represented by an $(m+1) \times (n+1)$ rectangular grid of vertices and edges. To illustrate: the 1×2 puzzle in Figure 2.1 has one row of two squares. It is represented by a 2×3 rectangular grid with two rows of three vertices each (shown as the blue circles). There are seven edges, $e_1, \ldots, e_7$, connecting the six vertices.

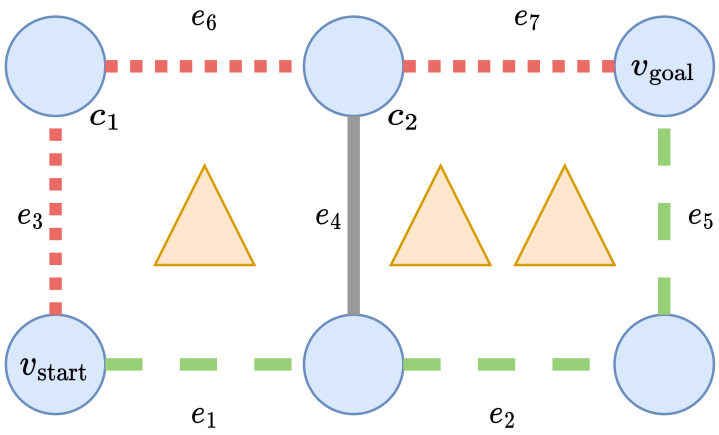

Figure 2.1: A 1×2 Witness puzzle. The green dashed line solves the puzzle since the solution path intersects the left square once and right square twice which are equal to the number of triangles in their respective squares. The red dotted line is not a solution to the puzzle.

Each *square* is delineated by four grid edges and can carry a *constraint* $c \in C$ which associates that square with a positive number of *triangles* contained in that square. A *solution* to the puzzle is a path on G which (i) connects the start vertex v_{start} to the goal vertex v_{goal} using the edges in E, (ii) never visits a vertex more than once, and (iii) satisfies all constraints in C. To satisfy a square's constraint $c \in C$ (i.e., the $k > 0$ triangles in that square) the path must include exactly k of that square's four edges. We say that a path *intersects* a square if the path includes at least one of the square's edges. A square without triangles imposes no constraints.

To illustrate, the 1×2 puzzle in Figure 2.1 has two constraints. The left square contains a single triangle ($k = 1$) which means that any solution path to the puzzle must include exactly one of the square's edges. The right square carries two triangles ($k = 2$) which imposes the constraint of including two of the square's edges in any solution path.

The red dotted line connecting the start and goal vertices in Figure 2.1 is not a solution since it violates both constraints (i.e., includes two edges from the first square and only one edge from the second square). On the other hand, the dashed green line is a solution satisfying both constraints. A *partial path* is any path in G that starts out in v_{start} but does not reach v_{goal}. A partial path

is *incompletable* if it is not a prefix of a solution. In Figure 2.1, the partial path $[e_3, e_6]$ is incompletable while $[e_1, e_2]$ is *completable*. *Solution length* is the number of edges in the solution.

2.2 Search for Solving Witness Puzzles

We now describe an A* search algorithm (Hart, Nilsson, and Raphael 1968) adapted for solving Witness puzzles (Algorithm 1). Let $\mathcal{P}$ be the space of all puzzles and $\mathcal{L}$ be the space of all partial paths on the puzzles. We adapt A* by adding the use of a function $\pi : \mathcal{P} \times \mathcal{L} \to \mathbb{R}$.

The search starts at v_{start} and proceeds by expanding its search frontier (i.e., the open list) containing partial paths and housed in a priority queue. The priority queue is sorted by $(\pi, g+h, h)$ meaning that lower values of π are returned from the queue first. Ties among π values are broken in favor of lower $g+h$ (length of the partial path so far + a heuristic estimate of the remaining length). Remaining ties are broken in favor of the lower heuristic estimate h of the remaining length which is equivalent to breaking ties towards higher g, a common technique in A* search. Any residual ties are then broken in an arbitrary fixed order. Given the four-connected rectangular grid underlying our Witness puzzles, we use Manhattan distance (MD) as the heuristic h.

We will demonstrate benefits of restricting π to a two-valued function, that is, $\pi : \mathcal{P} \times \mathcal{L} \to \{\text{True}, \text{False}\}$. Such a predicate takes a puzzle instance and a partial path on that puzzle and returns True if the partial path is predicted to be incompletable to a solution and False otherwise, in which case we say π predicts the partial path is *completable*. In sorting by π, partial paths for which π returns False are placed earlier in the queue than paths labeled True. An accurate predicate will thus focus Algorithm 1 on completable partial paths.

Furthermore the binary nature of the predicate allows us to switch from sorting the open list by π to pruning partial paths by π without even putting them on the open list. Later in this section we will discuss when such a switch from sorting to pruning can be made while maintaining completeness of the underlying A* search.

Algorithm 1: A* search for Witness puzzles.

```
   input  : puzzle p = (G, v_start, v_goal, C), predicate π
   output: solution ℓ
 1 Q ← priority queue sorted by (π, g + h, h)
 2 push v_start onto Q
 3 while Q ≠ ∅ do
 4 |   ℓ ← head(Q)
 5 |   Q ← Q ∖ ℓ
 6 |   v ← end(ℓ)
 7 |   foreach v_new ∈ N(v) ∖ ℓ do
 8 |   |   ℓ_new ← [ℓ, v_new]
 9 |   |   if v_new = v_goal then
10 |   |   |   if ∀c ∈ C [ℓ_new satisfies c] then
11 |   |   |   |   return ℓ_new
12 |   |   else
13 |   |   |   push ℓ_new onto Q
14 return ∅
```

The search loop continues as long as the open list is not empty (line 3) and a solution is not found (line 11). The first sorted element of the open list, a partial path ℓ, is retrieved and removed from the queue in lines 4 and 5 and the last vertex v of the path is retrieved in line 6. We then *expand* the end of the path v by generating all neighbours v_{new} of v in the graph G that are not already on the path ℓ (line 7). We generate each new path by appending v_{new} to the end of ℓ (line 8). If the new path ℓ_{new} reaches the goal vertex, we check if it satisfies all constraints in C (line 10). If so, the search stops and the solution is returned in line 11. Otherwise, each new partial path is put onto the open list with its $\pi, g + h$, and h values (line 13). When π is constant (e.g., always True or always False) we have the standard A* algorithm.

Note that Algorithm 1's search is *complete* (i.e., it will find a solution to a puzzle if there is one) even when π has false positives, since any solution that is wrongly predicted to be incompletable by π will still end up in the queue – just with a lower priority.

We *prune* partial paths that π predicts are incompletable by never putting them on the queue. Line 13 in Algorithm 1 is replaced with the lines below:

13' **if** *Not* $\pi(p, \ell_{\text{new}})$ **then**
14' push ℓ_{new} onto Q

In line 13' if π predicts ℓ_{new} is completable for a puzzle p, then we push ℓ_{new} onto Q as before in line 14'. Otherwise if π predicts ℓ_{new} is incompletable (i.e., $\pi(p, \ell_{\text{new}})$ is True) we prune ℓ_{new}. We refer to Algorithm 1 replacing line 13 with line 13' and 14' above as *Algorithm 1 with pruning* throughout the thesis.

Theorem 1. When π has no false positives (i.e., π returns True only if a partial path is indeed incompletable) we can use π to prune partial paths that π predicts are incompletable by never putting them in the queue while preserving completeness of the search.

Proof. Assume that pruning by replacing line 13 with line 13' and line 14' makes Algorithm 1 return no solution for a puzzle that has a solution. Since the open list starts with v_{start}, which is completable since the puzzle has a solution, at some point in line 13' a partial path which is completable must have been incorrectly pruned from Q. However, we assumed that π has no false positives, which means that such pruning cannot happen. □

Define a *search tree* as a tree with the initial state of the search (the partial path containing only the starting vertex v_{start}) as the root partial path. The children of each partial path are obtained by adding a new vertex to the end of the path (line 8 of Algorithm 1). An example of the first two layers of a search tree for solving the puzzle in Figure 2.1 is shown in Figure 2.2. The root of the tree, A, only contains the path of the starting vertex (the $\star$). The two children of A are the partial paths shown with dashed lines moving right from the start (B) and moving up from the start (C). The two children of B are the partial paths moving right (D) and moving up (E). The partial path C only has one child, since it can be extended only to the partial path F.

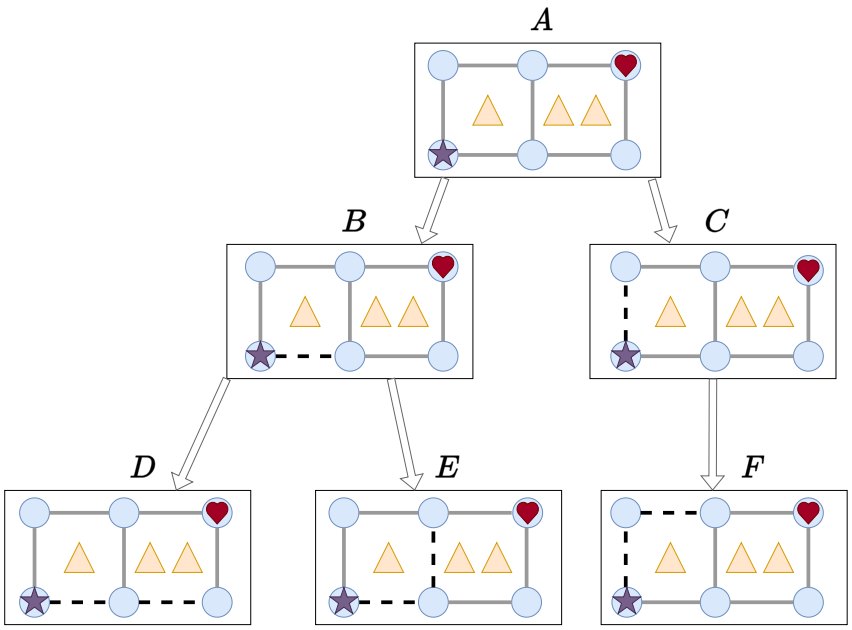

Figure 2.2: A partial search tree for solving the Witness puzzle in Figure 2.1. The dashed lines show the partial paths for solving the puzzle so far. The two vertex markers $\star, \heartsuit$ mark the start and goal vertices of the grid.

2.3 Baseline

Any partial path that includes more of a square's edges than there are triangles in that square cannot possibly be completed to a full solution. Suppose that we had a predicate, which we refer to as *local constraint checking* throughout the thesis, that predicted path incompletability based on this observation.

Theorem 2. The local constraint checking predicate has no false positives, that is, if it predicts a path is incompletable, then it is actually incompletable.

Proof. Suppose for the sake of contradiction that local constraint checking predicts a partial path ℓ is incompletable that is actually completable. Notice adding any edges to such a path would still result in a path that violated the triangle constraint that made local constraint checking predict True. Therefore it is impossible for ℓ to be completable to a solution path. □

Corollary 1. Since the local constraint checking predicate has no false positives, by Theorems 1 and 2, we can use local constraint checking for pruning of partial paths while preserving completeness of the search.

To illustrate, consider the search tree in Figure 2.2 for solving the puzzle in Figure 2.1. The predicate would label partial paths E and F incompletable since they include two edges of the square that has one triangle in it. Since this predicate has no false positives, these partial paths (and all of their completions) can be pruned from the search tree. Consequently, of the partial paths of length two (the bottom row in the figure), only D would be added to the priority queue. This partial path is a solution prefix.

We refer to the local constraint checking predicate throughout the thesis as π_{baseline}. We use Algorithm 1 with pruning using the local constraint checking predicate (implemented by hand) as our *baseline*.

2.4 Accelerating the Search

The problem we tackle in this thesis is to automatically find a predicate π that speeds up Algorithm 1 with pruning on a set of Witness puzzles $\mathcal{P} = \{p_1, \ldots, p_n\}$. We use two metrics to quantify the speed-up: improvements in the *solution time* and the *number of nodes expanded*.

For a given puzzle $p \in \mathcal{P}$, let the time it takes our baseline with pruning to solve the puzzle be $t(\pi_{\text{baseline}}, p, \text{prune})$ and the number of nodes expanded by the baseline be given by $\mathcal{E}(\pi_{\text{baseline}}, p, \text{prune})$. Similarly let the time it takes Algorithm 1 with pruning using a learned predicate π be $t(\pi, p, \text{prune})$ and the number of nodes expanded be $\mathcal{E}(\pi, p, \text{prune})$.

The relative time reduction from using the predicate π on the set of puzzle instances $\mathcal{P}$ is defined as the *time speedup*:

$$\text{speedup}_t(\pi, \mathcal{P}, \text{prune}) = \frac{\sum_{p \in \mathcal{P}} t(\pi_{\text{baseline}}, p, \text{prune})}{\sum_{p \in \mathcal{P}} t(\pi, p, \text{prune})}. \tag{2.1}$$

Similarly the *expansion speedup* is:

$$\text{speedup}_{\mathcal{E}}(\pi, \mathcal{P}, \text{prune}) = \frac{\sum_{p \in \mathcal{P}} \mathcal{E}(\pi_{\text{baseline}}, p, \text{prune})}{\sum_{p \in \mathcal{P}} \mathcal{E}(\pi, p, \text{prune})}. \tag{2.2}$$

Note that the set $\mathcal{P}$ can consist of a single problem instance which then defines a per-instance speedup.

We can also define the speedup measure when using Algorithm 1 for sorting instead of pruning. Let the time it takes Algorithm 1 using a learned predicate

π be $t(\pi, p, \text{sort})$ and the number of nodes expanded be $\mathcal{E}(\pi, p, \text{sort})$. We can then define the time and expansion speedup with sorting to be

$$\text{speedup}_t(\pi, \mathcal{P}, \text{sort}) = \frac{\sum_{p \in \mathcal{P}} t(\pi_{\text{baseline}}, p, \text{sort})}{\sum_{p \in \mathcal{P}} t(\pi, p, \text{sort})} \tag{2.3}$$

$$\text{speedup}_{\mathcal{E}}(\pi, \mathcal{P}, \text{sort}) = \frac{\sum_{p \in \mathcal{P}} \mathcal{E}(\pi_{\text{baseline}}, p, \text{sort})}{\sum_{p \in \mathcal{P}} \mathcal{E}(\pi, p, \text{sort})}. \tag{2.4}$$

Theorem 3. Given a predicate π that has no false positives, for every set of puzzles $\mathcal{P}$ the expansion speedup metrics for sorting and pruning are the same, that is $\text{speedup}_{\mathcal{E}}(\pi, \mathcal{P}, \text{prune}) = \text{speedup}_{\mathcal{E}}(\pi, \mathcal{P}, \text{sort})$.

Proof. We begin by proving that on a single puzzle instance p that Algorithm 1 with pruning or sorting expands the same number of nodes. Note that Algorithm 1 with pruning is the same as with sorting except partial paths that are predicted to be incompletable in line 13' are not added to the open list. However, partial paths that are predicted to be incompletable will not be expanded in Algorithm 1 with sorting. This is because they are given a lower priority in the open list than the solution path which is predicted to be completable by π since π has no false positives. Thus Algorithm 1 will keep expanding completable partial paths until it finds a solution before it ever expands a partial path predicted to be incompletable by π. Since both approaches sort completable partial paths in the same way we have $\mathcal{E}(\pi, p, \text{prune}) = \mathcal{E}(\pi, p, \text{sort})$.

Since we proved this on every puzzle instance, we therefore see that $\sum_{p \in \mathcal{P}} \mathcal{E}(\pi, p, \text{prune}) = \sum_{p \in \mathcal{P}} \mathcal{E}(\pi, p, \text{sort})$ for every predicate π with no false positives. In particular since we proved π_{baseline} has no false positives in Theorem 2 we see that the numerator and denominator of Equation 2.2 and Equation 2.4 are the same, so we conclude $\text{speedup}_{\mathcal{E}}(\pi, \mathcal{P}, \text{prune}) = \text{speedup}_{\mathcal{E}}(\pi, \mathcal{P}, \text{sort})$.

□

Treating the time speedup measure as an objective function, we set out to find an approximate solution to the optimization problem:

$$\pi^* = \underset{\pi \in \Pi}{\text{argmax}}\, \text{speedup}_t(\pi, \mathcal{P}, \text{sort}) \tag{2.5}$$

where Π is a space of predicates defined on puzzle instances and paths. Thus, given a puzzle instance set $\mathcal{P}$ we wish to find the predicate π^* which speeds up Algorithm 1 with sorting the most relative to the baseline.

A desirable solution to the optimization problem would be found automatically by a computer and will be portable: a predicate synthesized for one puzzle should speed up search on other puzzles. We also prefer the predicate to be compact and human readable insomuch as their operation can be analyzed and explained by a human. Finally, we prefer the predicate to be verifiable in that it has no false positives so it can be used for pruning and maintain completeness of Algorithm 1 with pruning.

Chapter 3

Related Work

Browne (2013) used deductive search to solve *Slitherlink* puzzles (Nikoli Co., Ltd 1989) whose constraints are similar to triangle Witness puzzles. They modelled each puzzle as a constraint satisfaction problem and used hand-coded rules to manually reduce the domains of each variable. In our work, we aim to automatically find such rules that can speed up search. Other work that uses SAT solvers for solving puzzles (Bright et al. 2020) is parallel to our work. These solvers by themselves have a large number of possible solutions to search over, therefore the predicates we learn can speed up such solvers.

Butler, Torlak, and Popović (2017) used program synthesis to find strategies for solving *Nonogram* puzzles. They evaluate the output of their system by comparing it to a set of documented strategies, while we test the effectiveness of our learned predicate in speeding up A* search for solving puzzles in *The Witness*. Krishnan and Martens (2022) synthesized chess tactics using ILP. However, their approach required data from human chess games. On the other hand, the data we use for training *Popper* is generated by a computer. Furthermore, they do not test the effectiveness of their chess tactics in actual games, while we test the effectiveness of our predicates on *Witness* puzzles.

Krajňanskỳ et al. (2014) learned predicates for pruning actions in classical planning. This approach and similar work in classical planning that pruned states from heuristic search use the relaxed plan of a problem (Hoffmann and Nebel 2001; Richter and Westphal 2010). Such a relaxed plan is computed by dropping constraints from a problem. However, an effective constraint re-

laxation strategy is not obvious for triangle Witness puzzles since removing a single constraint can substantially change the solution path. For instance, Sturtevant, Decroocq, et al. (2020) demonstrated the consequences of making small changes to a puzzle in the game *Snakebird*. It remains an open question to see if other relaxation strategies may lead to helpful actions for solving *Witness* puzzles (Hoffmann and Nebel 2001). Furthermore, other logical reasoning systems such as Theorist (Poole, Goebel, and Aleliunas 1987) benefit from finding potential hypothesis with a dataset that grows in size. In our work, we keep the dataset fixed for learning predicates and only learn a predicate on a single puzzle instance at a time.

Chen, Sturtevant, and White (2023) automatically found the difficulty of puzzles using human-designed predicates for solving different types of *Witness* puzzles. Our work is parallel to theirs in that automatically finding these predicates can improve the puzzle difficulty estimator. Also the predicates they use are also human-designed while ours are machine-learned.

Sturtevant and Ota (2018) generated Witness puzzles using a different constraint in the game using exhaustive procedural content generation which can deterministically generates all possible puzzles of a given size. The generators we present in our work are stochastic on the other hand. Bulitko and Botea (2021) evolve Romanian crossword maps to find instances on which their AI solver can achieve high scores. In our work, we do not generate puzzles with the goal of achieving a high performance with any solver.

Research using MAP-Elites (Mouret and Clune 2015), an algorithm for generating diverse content, also attempts to generate content with specific properties such as levels that require a certain amount of dexterity (Khalifa et al. 2018). However, we solve puzzles while they generate levels. Cook et al. (2013) generated mechanics for solving puzzles in a platformer game using an evolutionary search. Their method is specific to platformer games which does not have a natural extension to the logical puzzles in *The Witness*.

Bulitko, Wang, et al. (2022) used genetic algorithms and simulated annealing to synthesize algebraic heuristic functions that speed up search on video game maps. Our work is complementary in that they synthesized heuristic

functions while we hold the heuristic function fixed and attempt to synthesize predicates. Chen and Sturtevant (2021) used priority functions to avoid reopening nodes in heuristic search. These priority functions were manually designed and based on g and h, which makes it unclear how to encode a binary property of a given path using them. We do make a comparison between our approach and weighted A*, which uses priority functions of the form $g + w \cdot h$.

Botea and Bulitko (2022) used multiple expansion tiers, placeholder nodes and constraint propagation to speed up solving Romanian crosswords puzzles. The tiers were defined by the two types of words specific to Romanian crosswords and the number of points they contribute to the solution. Their tiering mechanism is similar to our sorting of the open list with a predicate but we have an additional ability to completely prune partial paths instead of delaying their expansions. Furthermore our predicates are machine-learned.

Orseau and Lelis (2021) extended the Levin tree search algorithm to learn a neural network for both the policy and heuristic, which they call policy-guided heuristic search, to solve a different non-triangular type of Witness puzzle. We adapt their method to our type of Witness puzzles and empirically compare its performance to our approach. We provide background on Levin tree search as well as policy-guided heuristic search in Appendix A.

Chapter 4

Proposed Approach

In this chapter we begin by showing our predicate synthesis algorithm for finding an incompletability predicate, π. In order to achieve the preferences outlined in Section 2.4, we use inductive logic programming. In particular, inductive logic programming enables the predicates to be found automatically and to be human readable. We later on show theoretically that one of the learned predicates provably lacks false positives and can thus be used for pruning while preserving search completeness (Theorem 1).

4.1 Predicate Synthesis Algorithm

Our overall approach is shown in the pseudocode in Algorithm 2.*

Lines 2 through 5 of Algorithm 2 comprise our machine learning algorithm for the predicates with the puzzle sets $\mathcal{P}_{\text{train}}$, $\mathcal{P}_{\text{filter}_1}$, $\mathcal{P}_{\text{filter}_2}$, and $\mathcal{P}_{\text{filter}_3}$ being the data used for machine learning. The predicate π° is the sole output of the learning process.

Each instance p from the first set, $\mathcal{P}_{\text{train}}$, is used to generate positive and negative training examples for *Popper* (Section 4.2). For each $p \in \mathcal{P}_{\text{train}}$, *Popper* learns an incompletability predicate π^p in line 2.

We have no guarantees on how the predicated learned on single instances will perform on other instances so we filter the predicates. We therefore partition the filtering set into three disjoint subsets, $\mathcal{P}_{\text{filter}_1}$, $\mathcal{P}_{\text{filter}_2}$, and $\mathcal{P}_{\text{filter}_3}$ such

*Henceforth in this section we generalize argmax in $(x_1, \ldots, x_k) \leftarrow \text{argmax}_{x \in X} f(x)$ to return the k elements from X that have the k maximum values of f.

Algorithm 2: Predicate synthesis algorithm for incompletability predicates.

input : training problem set $\mathcal{P}_{\text{train}}$, filter problem sets $\mathcal{P}_{\text{filter}_1}$, $\mathcal{P}_{\text{filter}_2}$, and $\mathcal{P}_{\text{filter}_3}$, filter numbers k_1 and k_2
output: predicate π°
assert : $|\mathcal{P}_{\text{filter}_1}| < |\mathcal{P}_{\text{filter}_2}| < |\mathcal{P}_{\text{filter}_3}|$ and $|\mathcal{P}_{\text{train}}| > k_1 > k_2$

1 **foreach** $p \in \mathcal{P}_{train}$ **do**
2 $\quad \pi^p \leftarrow \mathit{Popper}(p)$
3 $(\pi_1, \ldots, \pi_{k_1}) \leftarrow \underset{\{\pi^p \mid p \in \mathcal{P}_{\text{train}}\}}{\text{argmax}} \text{speedup}_t(\pi^p, \mathcal{P}_{\text{filter}_1}, \text{sort})$
4 $(\pi_1, \ldots, \pi_{k_2}) \leftarrow \underset{\{\pi_i \mid i \in \{1,\ldots,k_1\}\}}{\text{argmax}} \text{speedup}_t(\pi_i, \mathcal{P}_{\text{filter}_2}, \text{sort})$
5 $\pi^\circ \leftarrow \underset{\{\pi_i \mid i \in \{1,\ldots,k_2\}\}}{\text{argmax}} \text{speedup}_t(\pi_i, \mathcal{P}_{\text{filter}_3}, \text{sort})$
6 **return** π°

that $|\mathcal{P}_{\text{filter}_1}| < |\mathcal{P}_{\text{filter}_2}| < |\mathcal{P}_{\text{filter}_3}|$ to implement our version of the triage used by Bulitko, Hernandez, and Lelis (2021).

We first evaluate each of the generated incompletability predicates on $\mathcal{P}_{\text{filter}_1}$. Note we use the sorting speedup metric since we do not know yet whether or not the predicates have any false positives. We then choose the k_1 predicates with the highest time speedup in line 3. We then evaluate these k_1 predicates on a second, larger set of puzzle instances, $\mathcal{P}_{\text{filter}_2}$. We retain the k_2 predicates with the highest time speedup in line 4.

Finally, we evaluate the remaining k_2 predicates on the largest filter set of puzzles, $\mathcal{P}_{\text{filter}_3}$ and the single highest-speedup predicate π° is selected in line 5. The predicate π° is the sole output of the learning process and constitutes our machine-learned approximation to π^* (Equation 2.5).

To measure performance of the learned predicate π° we use a novel testing set of puzzle instances, $\mathcal{P}_{\text{test}}$, disjoint from the training and filtering sets and thus not seen by the learning algorithm. On this set we measure $\text{speedup}_t(\pi^\circ, \mathcal{P}_{\text{test}}, \text{sort})$ and $\text{speedup}_{\mathcal{E}}(\pi^\circ, \mathcal{P}_{\text{test}}, \text{sort})$. If we can successfully show that the learned predicate has no false positives then we can also measure $\text{speedup}_t(\pi^\circ, \mathcal{P}_{\text{test}}, \text{prune})$ and $\text{speedup}_{\mathcal{E}}(\pi^\circ, \mathcal{P}_{\text{test}}, \text{prune})$.

4.2 Learning Predicates from Puzzle Instances

We begin by detailing our approach of how *Popper* learns a predicate in line 2 of Algorithm 2. To simplify the learning task we will learn a predicate π' defined on a *single* constraint from the set of constraints contained in the puzzle p and a partial path. Then to compute π in Algorithm 1 we will call π' repeatedly on each constraint from in the puzzle. If π' returns True for any constraint we immediately set π to True. Otherwise we set the output of π to False (Appendix B).

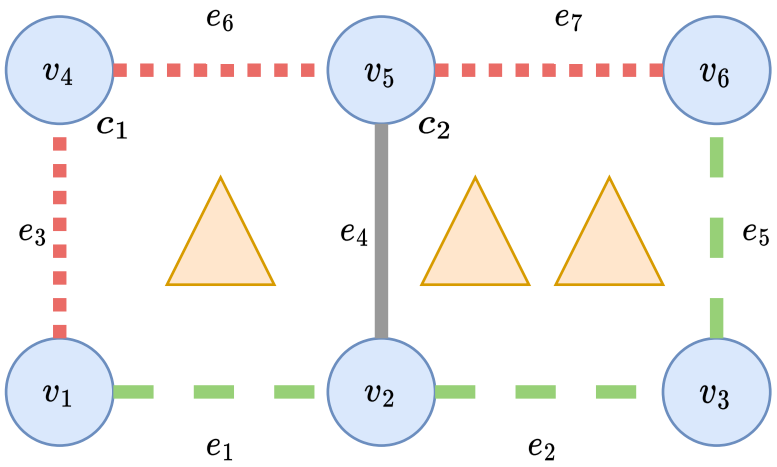

Figure 4.1: Augmented version of the puzzle from Figure 2.1

To learn an incompletability predicate, we use inductive logic programming. A problem in inductive logic programming (Cropper, Dumančić, et al. 2022; Muggleton 1991) is specified by background knowledge, which includes basic function definitions and knowledge about the problem setting, as well as positive and negative examples. The goal of inductive logic programming is to produce a predicate such that all positive examples are satisfied (entailed) by the predicate, and all of the negative examples are not satisfied by the predicate. The inductive logic programming algorithm we use is *Popper* (Cropper and Morel 2021) which learns predicates in the Prolog programming language. We illustrate with the puzzle from Figure 2.1 which we augment with additional notation in Figure 4.1.

Predicate	Returns True if
`edge(A,B,C)`	A is an edge connecting vertices B and C
`square(A,B,C)`	B is the number of triangles and C is the list of edges around square A
`path(A,B)`	B is the list of edges in path A
`pathHead(A,B)`	B is the vertex at the head of path A
`count(A,B,C)`	the number of occurrences of list A in list B is C, that is $\|A \cap B\| = C$
`len(A,B)`	the length of list A is B
`gte(A,B)`	$A \geq B$
`greaterThan(A,B)`	$A > B$
`incident(A, B)`	last vertex of path A has an edge incident to it in square B
`notIncident(A,B)`	last vertex of path A does not have an edge incident to it in square B
`one(A), two(A), three(A)`	A is 1, 2 or 3 respectively.

Table 4.1: Predicate building blocks for the background knowledge given to *Popper*.

We supply *Popper* with *background knowledge*: predicates to use as building blocks while learning the incompletability predicate (Table 4.1). For instance, `count(A,B,C)` specifies the number of elements in common between lists A and B is C. Thus `count([1,2], [2,3], 1)` is True, while `count([1, 2, 3], [2, 3, 4], 3)` is False since the lists have two elements in common.

The predicates in Table 4.2 can be used to describe a puzzle instance, for instance Figure 4.1 is described in Table 4.2. In Table 4.2 for example `square(c1, 1, [e1, e3, e4, e6])` states the fact that the square `c1` in Figure 4.1 contains one triangle and is defined by the edges listed: `e1`, `e3`, `e4`, `e6`.

To generate examples for *Popper* given a puzzle instance p we enumerate all paths (without vertex repetitions) between the start and the goal vertices and check to see which are solutions to the puzzle. Any partial path that is not the prefix of a solution is incompletable and is thus recorded as a positive training

```
edge(e1, v1, v2).   edge(e3, v1, v4).   edge(e3, v4, v1).   edge(e6, v4, v5).
edge(e1, v2, v1).   edge(e2, v2, v3).   edge(e4, v2, v5).   edge(e6, v5, v4).
edge(e4, v5, v2).   edge(e7, v5, v6).   edge(e2, v3, v2).   edge(e5, v3, v6).
edge(e7, v6, v5).   edge(e5, v6, v3).
square(c1, 1, [e1, e3, e4, e6]).   square(c2, 2, [e2, e4, e5, e7]).
path(p2,[e1]).      path(p3,[e3]).      path(p4,[e1, e2]).   path(p5,[e1, e4]).
path(p6,[e3, e6]).  path(p7,[e1, e4, e6]).
path(p8,[e3, e6, e4]).                  path(p9,[e3, e6, e4, e2]).
pathHead(p2,v2).    pathHead(p3,v4).    pathHead(p4,v3).    pathHead(p5,v5).
pathHead(p6,v5).    pathHead(p7,v4).    pathHead(p8,v2).    pathHead(p9,v3).
```

Table 4.2: The background knowledge describing the puzzle in Figure 4.1.

example. Any other partial path is recorded as a negative training example. For example, the only solution to the puzzle in Figure 2.1 is $[e_1, e_2, e_5]$. We show the positive and negative training examples in Table 4.3. Notice the only prefixes of the solution path are $p_2 = [e_1]$ and $p_4 = [e_1, e_2]$ which are the two negative training examples in Table 4.3.

```
neg(incompletable(p2, [c1 , c2 ])).   pos(incompletable(p3, [c1 , c2 ])).
neg(incompletable(p4, [c1 , c2 ])).   pos(incompletable(p5, [c1 , c2 ])).
pos(incompletable(p6, [c1 , c2 ])).   pos(incompletable(p7, [c1 , c2 ])).
pos(incompletable(p8, [c1 , c2 ])).   pos(incompletable(p9, [c1 , c2 ])).
```

Table 4.3: Positive and negative training examples given to *Popper*.

The background knowledge, a bias file controlling *Popper*'s operation (Appendix B), and the training examples for a single puzzle instance p are given as inputs to *Popper*. *Popper* then attempts to learn a predicate π^p that predicts incompletability for a partial path and a single constraint (Appendix B).

Popper uses answer set programming to generate predicates (Cropper and Morel 2021). *Popper* can then evaluate a candidate predicate using the positive and negative examples. If a candidate predicate satisfies a negative example, *Popper* prunes generalizations (disjunctions of the candidate predicate) from the space of candidates. If a predicate fails to satisfy some positive examples, *Popper* prunes specializations (conjunctions of the candidate predicate) from the space. This allows *Popper* to prune multiple candidate predicates from the remaining set of candidate predicates based on a single predicate.

Pruning disjunctions and/or conjunctions of the candicate predicate adds

extra constraints to the generation phase, used by answer set programming, to further constrain the next program generated. This loop continues until a solution predicate is found. If *Popper* cannot find a predicate that satisfies all positive examples and does not satisfy any negative examples, it returns a predicate that satisfies the most positive examples while not satisfying any negative examples. If no such predicate is found, *Popper* reports finding no solution.

Popper generates predicates of increasing size, where the size of a predicate is the number of head and body predicates contained in it. The smallest size of a predicate is 2, which consists of one head and body predicate. One of the first such predicates considered is

```
h1 = f(A,B) :- incident(A,B).
```

This predicate will return True for all of the examples in Table 4.3 since every partial path is incident to at least one of the two squares `c1` and `c2`. Thus `h1` satisfies the two negative examples for partial paths `p2` and `p4`. Therefore, `Popper` prunes generalizations of `h1`. A generalization of `h1` is a disjunction with other predicates, therefore predicates such as

```
f(A,B) :- incident(A,B); incident(B,A).
```

are pruned. Note the semicolon in Prolog indicates a logical or. Pruning the generalization of `h1` give additional constraints that *Popper* must take into consideration during the next generation step.

For the example puzzle in Figure 4.1 this process of generating and evaluating predicates is continued by *Popper* until π^{ex} is learned after considering 1,625 predicates in total. Note the commas are equivalent to logical ands.

π^{ex}`(A,B):- square(B,D,E),path(A,C),count(E,C,F),greaterThan(F,D).`

π^{ex}`(A,B):- notIncident(A,B),path(A,D),one(C),len(D,C).`

As we later show in Appendix C, the first *clause* (the first line of π^{ex} ending with the period) is equivalent to local constraint checking, while the second clause of this predicate may have false positives on some puzzle instances.

Chapter 5

Evaluation

In this chapter we describe the puzzle sets and hyperparameters used in our empirical study. We then present and discuss the results.

5.1 Puzzle Sets and Hyper Parameters for Learning a Predicate

As there is no existing repository of triangle puzzle instances we generated them using our generators in Appendix D with the hope the generators will be used by the community to build additional triangle puzzle datasets.

To generate a puzzle set we sampled the length and width of the puzzle uniformly at random between 2 and a maximum size. Note there are fewer possible puzzles of smaller sizes (such as 2×2) so the overall distribution of puzzle sizes is not uniform. Puzzle sets $\mathcal{P}_{\text{train}}, \mathcal{P}_{\text{filter}_1}, \mathcal{P}_{\text{filter}_2}, \mathcal{P}_{\text{filter}_3}$, and $\mathcal{P}_{\text{test}}$ were generated with Algorithm 3 (Appendix D).

We generated $\mathcal{P}_{\text{train}}$ as a set of 500 puzzles of sizes between 2×2 to 4×4. The number of puzzles and the puzzle sizes were chosen to generate enough training data so that *Popper* can learn each predicate within a few minutes.

We picked the maximum number of variables used when *Popper* generates predicates to be 7. Setting the number of variables higher creates the potential to learn better predicates but the learning time increases substantially. The filter sets contained puzzles of sizes between 2×2 and 5×5 with $|\mathcal{P}_{\text{filter}_1}| = 100, |\mathcal{P}_{\text{filter}_2}| = 500$, and $|\mathcal{P}_{\text{filter}_3}| = 2{,}500$. The predicates were filtered to the top $k_1 = 25$ predicates, then the top $k_2 = 5$ predicates, and finally determine π°.

size	squares	instances	expansions	solution time (sec)
2×2	4	135	11.2 ± 6.4	0.0005 ± 0.0002
2×3	6	1,321	30.5 ± 24.4	0.001 ± 0.0006
2×4	8	1,788	71.7 ± 66.5	0.002 ± 0.002
3×3	9	1,012	109.4 ± 114.0	0.003 ± 0.003
2×5	10	1,977	172.8 ± 182.2	0.005 ± 0.005
3×4	12	2,112	400.9 ± 526.9	0.01 ± 0.02
3×5	15	2,313	$1{,}533.1 \pm 2{,}418.8$	0.05 ± 0.08
4×4	16	1,137	$2{,}397.2 \pm 3{,}741.3$	0.08 ± 0.13
4×5	20	2,123	$1.4 \times 10^4 \pm 2.5 \times 10^4$	0.5 ± 1.1
5×5	25	1,082	$1.4 \times 10^5 \pm 2.9 \times 10^5$	6.6 ± 14.8

Table 5.1: The 15,000 instances in $\mathcal{P}_{\text{test}}$ by puzzle size. Average number of expansions and wall time for the baseline search with π_{baseline} are listed as well.

The testing set $\mathcal{P}_{\text{test}}$ contained 15,000 puzzles of sizes between 2×2 and 5×5. The test size included more puzzles to get more reliable results for $\text{speedup}_t(\pi^\circ, \mathcal{P}_{\text{test}}, \text{sort})$ and $\text{speedup}_{\mathcal{E}}(\pi^\circ, \mathcal{P}_{\text{test}}, \text{sort})$. The distribution of testing puzzle sizes is listed in Table 5.1. The large standard deviation is due to variance between puzzles in terms of their difficulty for the baseline algorithm.

5.2 A Learned Predicate: π°

Learned predicates progress through triage based on how much time speedup they offer in line 3 to line 5 in Algorithm 2. In using Algorithm 1 to evaluate a learned predicate, a partial path was predicted to be incompletable if either local constraint checking (i.e., π_{baseline} returned True (Section 2.3)) or the learned predicate predicted the partial path was incompletable.

To find the initial set of learned predicates, running *Popper* on 500 puzzle instances in $\mathcal{P}_{\text{train}}$ produced 489 predicates with 11 runs not finding a solution in the approximate one-hour time limit. All experimental runs were done on the Digital Research Alliance of Canada's Cedar cluster using the Intel Xeon processors. On each of the successful runs, *Popper* took an average of about eight and a half minutes, with a range of $[0.15, 60.1]$ minutes. This large range reflects the difference between learning predicates for smaller puzzles

	Predicate $\pi^\circ(A, B)$	**Simplified Python**
1	`square(B,D,C), path(A,E), count(E,C,F), notIncident(A,B), three(D), one(F).`	`if path.head is not incident to square and puzzle.triangles[square]=3 and path.edges[square]=1: incompletable=True`
2	Same except `two(F)`.	Same except `path.edges[square]=2`

Table 5.2: Output of Algorithm 2: the predicate π°. On the left the predicate is shown in Prolog and on the right pseudocode in Python.

which have fewer examples to learn from (less than 30) and larger puzzles that have many examples to learn from (over 40,000).

Our learning algorithm for predicates, Algorithm 2, returned the predicate shown in Table 5.2 as π°; its value computed as a logical disjunction of the two clauses. These two clauses offer useful additions to what π_{baseline} checks. They capture the idea that if a partial path leaves a three-triangle constraint square's edges without including enough of them, the path can never return there because doing so would require visiting one of the square's vertices twice. This is implemented by returning True if the path head is not incident to the square (`notIncident(A,B)`), the number of triangles is 3 (`three(D)`), and the number of current intersections is 1 or 2 (`one(F)` or `two(F)`).

Note that while the table lists our learned predicate π° in Prolog it can be ported to another language. In our experiments we re-implemented π° in Python in which our A* was implemented. This allowed us to call π° from Algorithm 1 without invoking Prolog, improving search speed substantially (Appendix E).

5.3 Verifiability of π°

The fact that the learned predicate is a binary function with only two values, True and False, invites us to use it in Algorithm 1 with pruning instead of merely putting incompletable partial paths at the end of the open list (the original line 13 of Algorithm 1). Pruning speeds up search relative to sorting because it never puts partial paths it predicts are incompletable on the open list

but it also carries the risk of pruning a prefix to the only solution by mistake. Doing so may render Algorithm 1 incomplete on that puzzle instance.

A partial path is pruned when π returns True on it. Thus if the predicate has no false positives (i.e., never returns True on a prefix of a solution) then such π preserves completeness of Algorithm 1 by Theorem 1.

Theorem 4. The learned predicate π° in Table 5.2 has no false positives (i.e, it never predicts that a solution prefix is incompletable).

Proof. Since π° consist of a disjunction of the two clauses in Table 5.2, it suffices to prove that each individual clause yields no false positives.

In both clauses by `square(B,D,C)` and `three(D)` we know that `B` is a square containing a three-triangle constraint with the list of edges given by `C`. Let the vertices incident to this square be v_1, v_2, v_3, v_4. By `path(A,E)` we know that `A` is a path with the list of edges `E`. Furthermore by `notIncident(A,B)` we know that the head of path `A` (a vertex in the graph) is not incident to any edge in square `B`. Therefore the head of the path is not v_1, v_2, v_3 or v_4.

In the first clause given `count(E, C, F)` and `one(F)` we know that path `A` (with edges in `E`) and square `B` (with edges in `C`) share only a single edge in common. Without loss of generality, let vertices v_1 and v_2 be the two vertices along this shared edge (the edge (v_1, v_2) in the red dotted line in Figure 5.1).

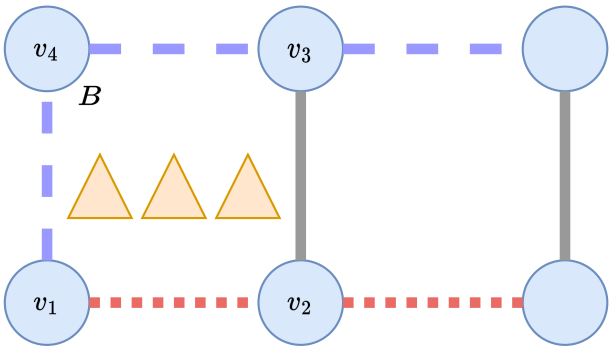

Figure 5.1: Portion of a Witness puzzle with a partial path for the `one(F)` case drawn in a red dotted line and for the `two(F)` case drawn in a blue dashed line.

Assume that path `A` is completable. Hence square `B`'s triangle constraint must be satisfied which means that path `A` needs to be extended to include two

more of square B's edges. However, every pair of the three remaining edges in square B contain an edge that either has v_1 or v_2 as one of its vertices. Thus this extension of A would need to visit the same vertex twice, which violates rule (ii) of Witness puzzles from Section 2.1. Thus path A is incompletable which contradicts our assumption.

The proof for the second clause is similar, however, we are given `count(E, C, F)` and `two(F)` instead, so we know that path A and square B share two edges in common in this case. These two edges must include at least three vertices in square B. Without loss of generality, assume these vertices are v_1, v_4, and v_3 (the edges (v_1, v_4) and (v_4, v_3) of the blue dashed line in Figure 5.1).

Assume that path A is completable so an extension of A needs to include one more of square B's edges. However, any additional edge will include at least one of the vertices v_1 or v_3, so this extension of A would need to visit the same vertex twice which violates rule (ii) of Witness puzzles from Section 2.1. Hence path A is incompletable which contradicts our assumption.

Since we have shown each clause has no false positives their disjunction, π°, has no false positives. □

5.4 Search Acceleration of π°

On the test set $\mathcal{P}_{\text{test}}$ we compared two Python implementations of Algorithm 1 with pruning using π° and π_{baseline}. The predicate π° had the following speedups:

$$\text{speedup}_t(\pi^\circ, \mathcal{P}_{\text{test}}, \text{prune}) = 6.38, \tag{5.1}$$

$$\text{speedup}_{\mathcal{E}}(\pi^\circ, \mathcal{P}_{\text{test}}, \text{prune}) = 6.27. \tag{5.2}$$

The range of time speedups was $[0.59, 1414]$ and of expansion speedups was $[1, 1447]$. Thus Algorithm 1 with pruning using π° never expanded more nodes than the baseline on any problem in $\mathcal{P}_{\text{test}}$. When computing the per puzzle instance time speedup for each $p \in \mathcal{P}_{\text{test}}$, the predicate π° had a time speedup of less than 1 (i.e., $\text{speedup}_t(\pi^\circ, p, \text{prune}) < 1$) on roughly a third (33.5%) of instances. However, π° also has the potential to significantly speed

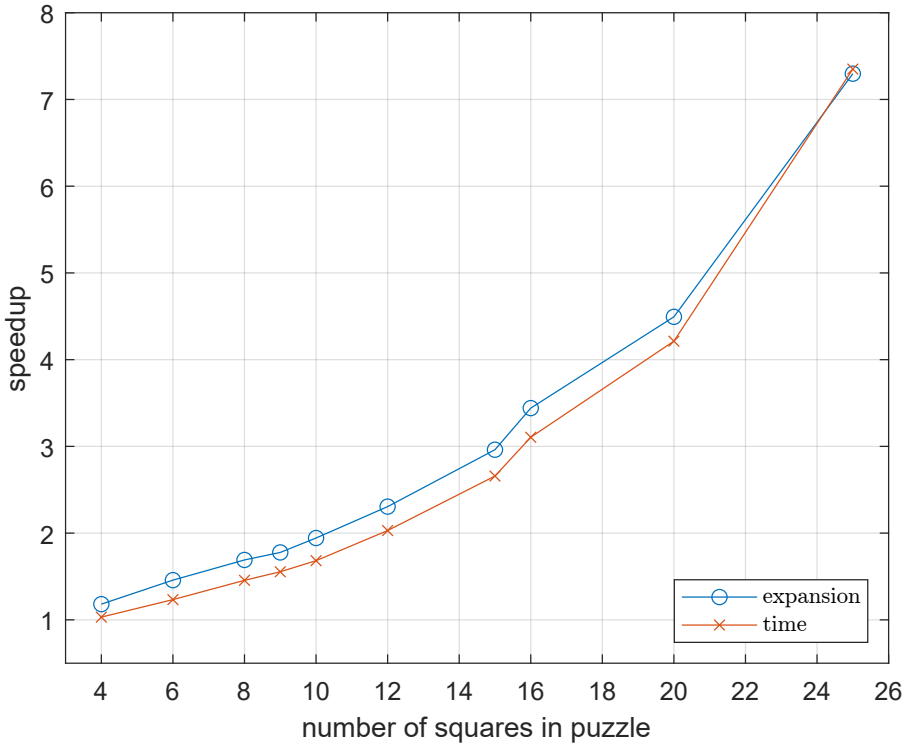

Figure 5.2: Time and expansion speedup of π° on $\mathcal{P}_{\text{test}}$.

up search, and on one instance had a time speedup of 1,414 and expansion speedup of 1,447. We analyze this instance in Section 5.6.1.

We partitioned the 15,000 instances from $\mathcal{P}_{\text{test}}$ into ten buckets based on size (Table 5.1). Figure 5.2 shows time and expansion speedups for the instances in each bucket, plotted as a function of the number of squares in the bucket's puzzles. The speedup afforded by using the predicate appears to increase with the puzzle size. This is possibly because larger puzzles on average have more constraints when generated with Algorithm 3 (an average of 5.2 on 5×5 puzzles compared to 1.9 on 2×2 puzzles) in Appendix D. Thus since predicate π° only applies to three-triangle constraints, there are on average more three triangle constraints that π° applies to (an average of 1.5 on 5×5 puzzles compared to 0.6 on 2×2 puzzles), potentially resulting in a higher speedup. This is illustrated with the puzzle in Figure 5.3 which has five three-triangle constraints. We show a distribution of the expansion speedups in Figure 5.6.

5.5 Sorting with π°

We were able to prove that π° had no false positives and thus could be used for Algorithm 1 with pruning. It is of interest to consider the speedups afforded by π° when used for sorting instead of pruning:

$$\text{speedup}_t(\pi^\circ, \mathcal{P}_{\text{test}}, \text{sort}) = 6.66, \tag{5.3}$$

$$\text{speedup}_{\mathcal{E}}(\pi^\circ, \mathcal{P}_{\text{test}}, \text{sort}) = 6.27. \tag{5.4}$$

Observe that the expansion speedup is the same for sorting and pruning as we proved in Theorem 3, however, the time speedup is now higher as follows. Using π_{baseline} for sorting in Algorithm 1 takes about 11% longer to solve all the puzzles compared to when used for pruning. On the other hand, using π° for sorting takes only about 6% longer to solve all the puzzles compared to when used for pruning.

5.6 Solving Large Puzzles with π°

In this section we explore how the learned predicate π° can speed up search on large puzzles. We begin by showing a puzzle instance with a large speedup in the first section and then show how Algorithm 1 with pruning using our learned predicate π° can perform on solving puzzles larger than 5×5.

5.6.1 A Puzzle with a Large Speedup

In this section we show the puzzle instance from our testing set P_{test} where π° sped up Algorithm 1 the most. On this instance, Algorithm 1 with pruning using π° had a time speedup of 1,414 and expansion speedup of 1,447. The 5×5 puzzle is shown in Figure 5.3. We show that predicate π° forces Algorithm 1 with pruning to have the first eight vertices visited in a solution to Figure 5.3 to be $[1, 2, 3, 4, 5, 6, 7, 8]$. This demonstrates how Algorithm 1 with pruning using π° can sometimes eliminate many partial paths from its search tree.

From the start vertex ($\star$) we can either move up to vertex 3 or right to vertex 1. Either path has one edge in common with the three-triangle

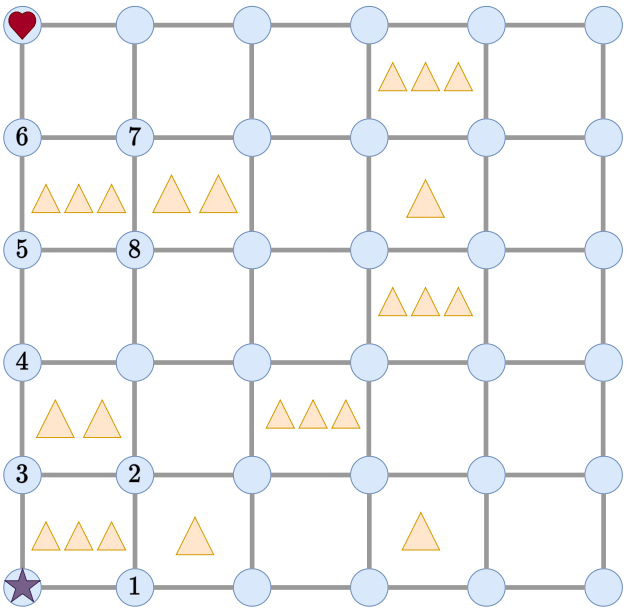

Figure 5.3: A 5×5 Witness puzzle. The vertices are numbered to show the partial path that pruning with π° forces Algorithm 1 to take. The two vertex markers $\star, \heartsuit$ mark the start and goal vertices.

square, so predicate π° predict that leaving this square before satisfying the square's triangle constraint will result in an incompletable path. There are two possible paths to satisfy the three-triangle constraint: $[3, 2, 1]$ or $[1, 2, 3]$. In the first case, we are then forced to move right from vertex 1, which would then intersect a one-triangle square twice. The first line of π° detects that this is incompletable, forcing the sequence of moves to be $[1, 2, 3]$.

From vertex 3 we can only move up to vertex 4. From vertex 4 we cannot move right since then the path will intersect a two-triangle square three times, which π° detects is incompletable. Thus we must move up to vertex 5 and our partial path so far is $[1, 2, 3, 4, 5]$.

From vertex 5 we must move either up to vertex 6 or right to vertex 8. In either case we have one edge in common with the three-triangle square. Predicate π° predicts that leaving this square without satisfying the square's triangle constraint will make the path incompletable. We have two ways to satisfy the three-triangle constraint: either $[6, 7, 8]$ or $[8, 7, 6]$. In the latter case, our partial path would be $[1, 2, 3, 4, 5, 8, 7, 6]$, and our only move without repeating vertices is up to the goal. However, there are still squares with trian-

gle constraints that are not satisfied so this path is invalid. Thus no successor states of this partial path will be added to the open list due to the if statements in line 10 of Algorithm 1 returning False. Hence the former case is the only possibility and the starting sequence of eight moves is $[1, 2, 3, 4, 5, 6, 7, 8]$.

5.6.2 Solving Large Puzzle Instances

We investigated how large a puzzle could be solved with Algorithm 1 with pruning using π° given a fixed time and memory limit. We generated a set of 500 puzzle instances of each of the following sizes: 5×5, 5×6, 5×7, 5×8, 6×6, 6×7, 6×8, 7×7, 7×8, and 8×8 using Algorithm 4 in Appendix D.*

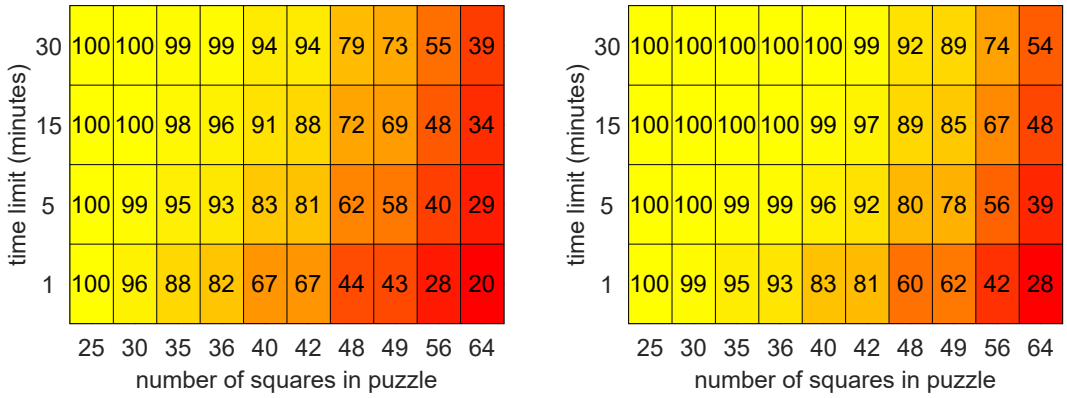

Figure 5.4: Rounded percentage of puzzle instances solved by Algorithm 1 with pruning using π_{baseline} (left) and with π° (right).

We then ran Algorithm 1 with pruning using π° and π_{baseline} for solving these large puzzles. We recorded the number of puzzle instances solved using each of the predicates under a time limit of $\{1, 5, 15, 30\}$ minutes and memory limit of 128 Gbytes (Figure 5.4).

The puzzle instances were difficult for the baseline search: only the 25 and 30 square puzzles could all be solved in under 30 minutes. Algorithm 1 with pruning using π° solved all puzzles up to 36 squares as well as over half of the 64 square puzzles. Furthermore, Algorithm 1 with pruning using π° given a time limit of 5 minutes solves approximately 84% of puzzles which is more than the 83% using π_{baseline} solves given a time limit of 30 minutes.

*We chose it over Algorithm 3 as it can generate larger puzzle instances faster, not needing to verify existence of a solution.

We show an example of a puzzle that is not solvable by Algorithm 1 with pruning using π° in under 30 minutes in Chapter 6.

5.7 Weighted A*

A variation of A* search is weighted A* (Pohl 1970). In it, nodes are inserted into the priority queue by $(\pi, g + w \cdot h, h)$, where w is a weight. Notice that when $w = 1$ we have Algorithm 1. When the value of the weight is $w = 0$, we have breadth-first search since nodes are expanded by the smallest value of g first. Also when the value of the weight is sufficiently high, nodes are expanded by lowest value of the heuristic first. To implement weighted A* we can replace line 1 of Algorithm 1 with the line 1' below:

1' $Q \leftarrow$ priority queue sorted by $(\pi, g + w \cdot h, h)$

We evaluate weighted A* with pruning using π_{baseline} to see how its performance compares to that of Algorithm 1 with pruning using the learned predicate π°. Define $\mathcal{P}_{\text{train+filter}} = \mathcal{P}_{\text{train}} \cup \mathcal{P}_{\text{filter}_1} \cup \mathcal{P}_{\text{filter}_2} \cup \mathcal{P}_{\text{filter}_3}$ as all of the problems in the training and filter sets. For a given weight w, let the time it takes weighted A* with pruning using π_{baseline} to solve puzzle p be $t(\pi_{\text{baseline}}, w, p, \text{prune})$ and the number of expansions be $\mathcal{E}(\pi_{\text{baseline}}, w, p, \text{prune})$.

We extend the definition of time and expansion speedup in Equation 2.1 and Equation 2.2 to allow the first argument to be a weight instead of a predicate. We can then define the time and expansion speedup for weighted A* with pruning to be

$$\begin{aligned} \text{speedup}_t(w, \mathcal{P}, \text{prune}) &= \frac{\sum_{p \in \mathcal{P}} t(\pi_{\text{baseline}}, w, p, \text{prune})}{\sum_{p \in \mathcal{P}} t(\pi_{\text{baseline}}, p, \text{prune})} \\ \text{speedup}_{\mathcal{E}}(w, \mathcal{P}, \text{prune}) &= \frac{\sum_{p \in \mathcal{P}} \mathcal{E}(\pi_{\text{baseline}}, w, p, \text{prune})}{\sum_{p \in \mathcal{P}} \mathcal{E}(\pi_{\text{baseline}}, p, \text{prune})}. \end{aligned}$$

We compute the weight which takes the least time to solve the puzzles in the training and filtering set by sampling 100 weights in an interval of 0.1 between 0 and 9.9:

$$w^\circ = \underset{w \in \{0, 0.1, \ldots, 9.9\}}{\text{argmax}} \text{speedup}_t(w, \mathcal{P}_{\text{train+filter}}, \text{prune}). \tag{5.5}$$

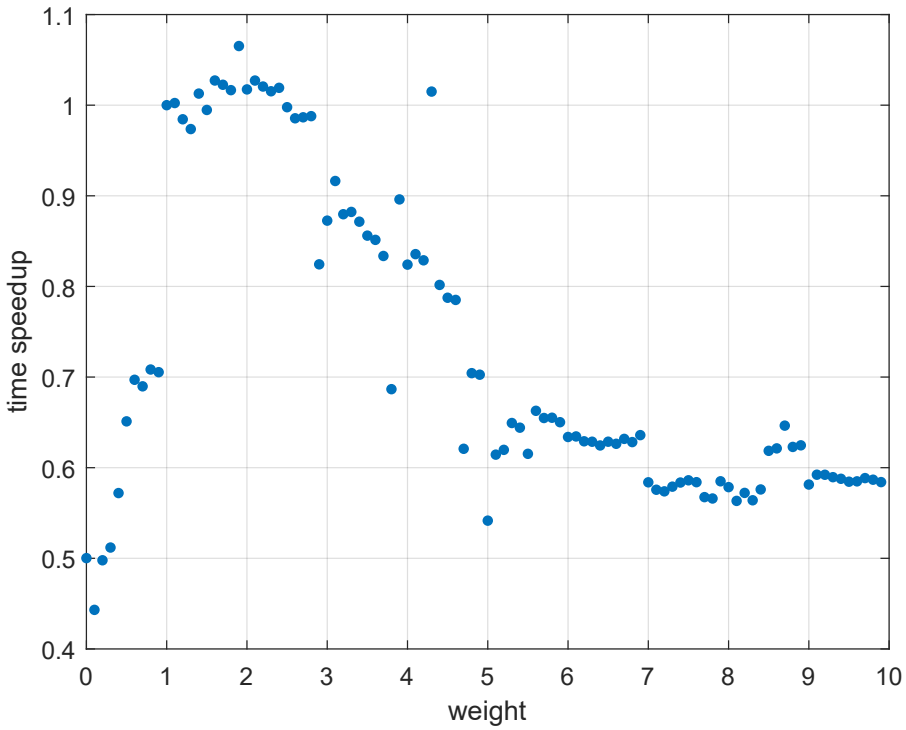

Figure 5.5: Time speedup of solving the problems in $\mathcal{P}_{\text{train+filter}}$ using weighted A* with various weights.

Figure 5.5 plots the speedup as a function of weight. The best weight found is $w^\circ = 1.9$ which gives us the following performance on the test set:

$$\text{speedup}_t(w^\circ, \mathcal{P}_{\text{test}}, \text{prune}) = 1.11, \tag{5.6}$$

$$\text{speedup}_{\mathcal{E}}(w^\circ, \mathcal{P}_{\text{test}}, \text{prune}) = 1.04. \tag{5.7}$$

These speedups are sufficiently below the speedups afforded by π°.

5.8 Levin Tree Search with Neural Networks

We extended the work of Orseau and Lelis (2021) to add local constraint checking to their policy-guided heuristic search algorithm, PHS*, described in Appendix A. We trained a neural network, ANN, for PHS* on the 500 problems in $\mathcal{P}_{\text{train}}$ for 2.52 hours on a local computer with an Intel i5-12600K and NVIDIA GeForce RTX 3060 until all of the problems in the training set were solved by bootstrapping (Arfaee, Zilles, and Holte 2011). Bootstrapping works by solving puzzles with an expansion budget and if on the last iteration

no new puzzles were solved, the budget for expansions is doubled. We then solved the 15,000 problems in $\mathcal{P}_{\text{test}}$ on the Digital Research Alliance of Canada's Cedar Cluster with Tesla V100s using the neural network ANN.

Let the time it takes PHS* with ANN using pruning with π_{baseline} to solve puzzle p be $t(\pi_{\text{baseline}}, \text{ANN}, p, \text{prune})$ and the number of expansions be $\mathcal{E}(\pi_{\text{baseline}}, \text{ANN}, p, \text{prune})$. We extend the definition of time and expansion speedup in Equation 2.1 and Equation 2.2 to allow the first argument to be a neural network instead. We can then define the time and expansion speedup for policy-guided heuristic search to be:

$$\begin{aligned}
\text{speedup}_t(\text{ANN}, \mathcal{P}, \text{prune}) &= \frac{\sum_{p \in \mathcal{P}} t(\pi_{\text{baseline}}, \text{ANN}, p, \text{prune})}{\sum_{p \in \mathcal{P}} t(\pi_{\text{baseline}}, p, \text{prune})} \\
\text{speedup}_{\mathcal{E}}(\text{ANN}, \mathcal{P}, \text{prune}) &= \frac{\sum_{p \in \mathcal{P}} \mathcal{E}(\pi_{\text{baseline}}, \text{ANN}, p, \text{prune})}{\sum_{p \in \mathcal{P}} \mathcal{E}(\pi_{\text{baseline}}, p, \text{prune})}.
\end{aligned}$$

Using the ANN learned from training above, we obtain the following time and expansion speedups:

$$\text{speedup}_t(\text{ANN}, \mathcal{P}_{\text{test}}, \text{prune}) = 0.02, \tag{5.8}$$

$$\text{speedup}_{\mathcal{E}}(\text{ANN}, \mathcal{P}_{\text{test}}, \text{prune}) = 0.77. \tag{5.9}$$

The range of expansion speedups for the neural network was $[0.003, 511]$ and time speedups was $[3 \times 10^{-5}, 7.8]$. Observe that the time speedup is about 39 times lower than the expansion speedup. One theory for why this occurred is due to the neural network forward-pass time for each expanded node in the search tree.

Overall, the performance of the neural network is worse than the previous two methods. One theory for why this is the case is that the number of puzzles instance from $\mathcal{P}_{\text{train}}$ with only 500 puzzles was too small. This may lead to the neural network overfitting to these training instances. Secondly, the puzzle sizes between the training and testing set are also different. In particular, the largest puzzle set in the training set is 4×4, while the testing set has puzzle sizes of up to size 5×5.

These lead the neural network to have the potential to misguide the search. When computing the per puzzle instance expansion speedup the neural network had an expansion speedup of less than 1 on 65% of instances. We show

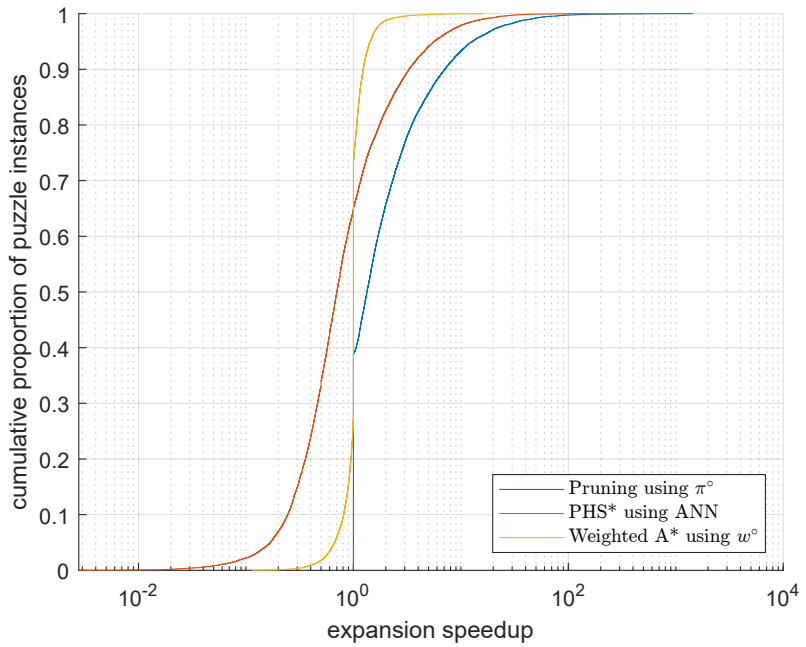

Figure 5.6: Cumulative distribution of puzzle instances with expansion speedups for Algorithm 1 with pruning using π° compared to weighted A* using w° and PHS* using ANN.

a cumulative distribution of the number of puzzle instances with various expansion speedups for the three methods in Figure 5.6.

To overcome the problem of having too little data for the neural network to train, we then generated a new dataset of 50,000 puzzle instances of sizes 2×2 to 4×4 that are disjoint from the testing set. This neural network took 138.5 hours to train to solve all these puzzle instances. We call the neural network that was trained on these ANN-50k.

The time and expansion speedup for this network are shown below:

$$\text{speedup}_t(\text{ANN-50k}, \mathcal{P}_{\text{test}}, \text{prune}) = 0.02, \tag{5.10}$$

$$\text{speedup}_\mathcal{E}(\text{ANN-50k}, \mathcal{P}_{\text{test}}, \text{prune}) = 1.04. \tag{5.11}$$

Note that the time speedup is similar to what it was before with ANN (Equation 5.8), while the expansion speedup is similar to the expansion speedup of weighted A* using w° (Equation 5.7). The plot of bucketing the expansion speedups by puzzle size is shown in Figure 5.7. Observe that PHS* using ANN-50k outperformed pruning using π° on the puzzles of size 3×4 and 4×4.

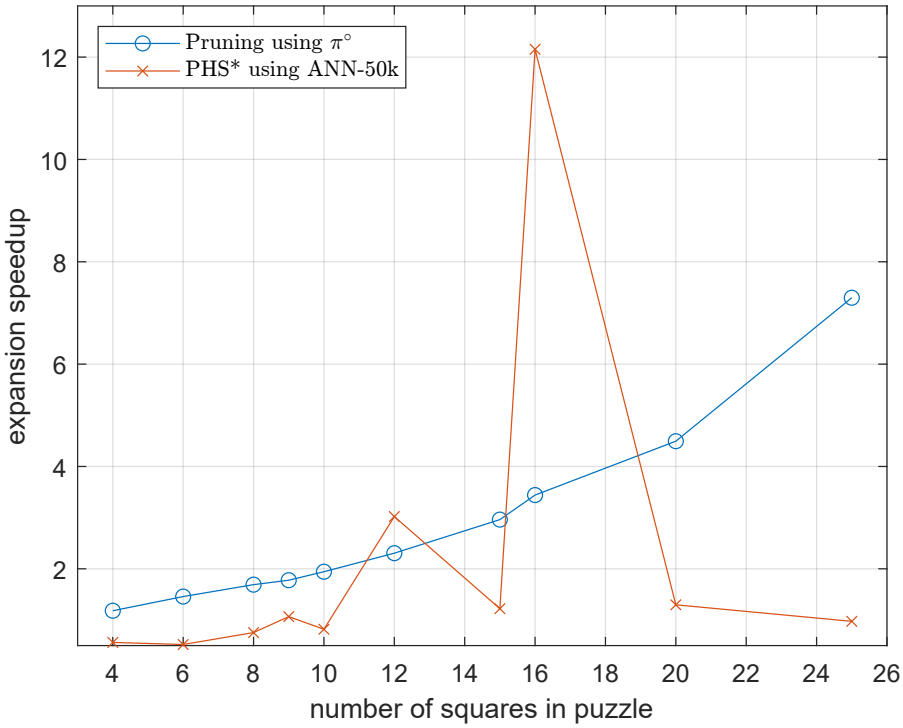

Figure 5.7: Expansion speedup of pruning using π° compared to PHS* using ANN-50k on $\mathcal{P}_{\text{test}}$.

These two puzzle sizes were the largest in the training set. This shows the potential PHS* has to learn a policy to effectively solve puzzles, however, it remains an open question of how to extend the work of Orseau and Lelis (2021) to train a neural network on puzzles of size 5×5.

5.9 Learned Predicate without π_{baseline}

We experimented with using Algorithm 1 without π_{baseline}. While in Section 5.2 we used π_{baseline} by default in computing the speedups on the filter set, we now run Algorithm 1 with only the learned predicates. We re-ran Algorithm 2 by taking the 489 predicates learned by *Popper* and evaluated them on the filter sets without π_{baseline}. The final predicate returned by Algorithm 2, π' contained three clauses. Two of the three clauses were functionally equivalent to the learned predicate π° in Table 5.2. Interestingly, the third clause

```
square(B,D,E),path(A,C),count(E,C,F),greaterThan(F,D).
```

is *equivalent* to π_{baseline} as it performs local constraint checking. Since `path(A,C)` we know that `A` is a path with the list of edges given by `C`. Furthermore by `square(B,D,E)` we know that `B` is a square containing `D` triangles with the list of edges defining the square given by `E`. By `count(E,C,F)` we know that the number of edges in common between the path and square is equal to `F`. Finally, by `greaterThan(F,D)` we know that the number of edges in common is larger than the number of triangles in the square. This is the same as local constraint checking, π_{baseline}.

Thus the learned predicate π' is equivalent to the learned predicate π° disjuncted with π_{baseline}. This means that without any prior knowledge of completability our approach automatically learned a predicate that captures the human-designed π_{baseline}.

Chapter 6

Open Questions & Future Work

We have demonstrated how an off-the-shelf ILP system with a triage-based filtering method (Algorithm 2) is able to learn a predicate that substantially speeds up a baseline search algorithm in solving Witness puzzles. The results lead us to the following open questions.

How much does the background knowledge provided to *Popper* impact the final predicates learned? In particular, one can attempt to find the minimum background knowledge needed to learn the predicates in Table 5.2. Conversely, future work can study how to extend the grammar to learn a predicate that can speed up search even more.

Future work can also compare our approach to other types of program synthesis such as bottom-up search or genetic algorithms. It can also explore automatically converting predicates from Prolog to Python to gain performance benefits without the effort of a manual re-implementation.

Since the final predicate learned can be converted into if and else statements (Table 5.2), future work can also explore how decision tree learners can learn the predicate (Alur, Radhakrishna, and Udupa 2017; Orfanos and Lelis 2023; Trivedi et al. 2021). Future work can see which features are needed to construct such a decision tree, similar to Table 4.1.

Future work can also investigate *portfolio search* where Algorithm 1 with pruning using another learned predicate is run in parallel with Algorithm 1 and π°. As Algorithm 1 with pruning using π° is complete (Theorem 4) the other learned predicate can be used for pruning even if it has false positives. Thus

on some puzzle instances the aggressively pruning member of the portfolio can yield a solution more quickly than Algorithm 1 with pruning using the more conservative π°.

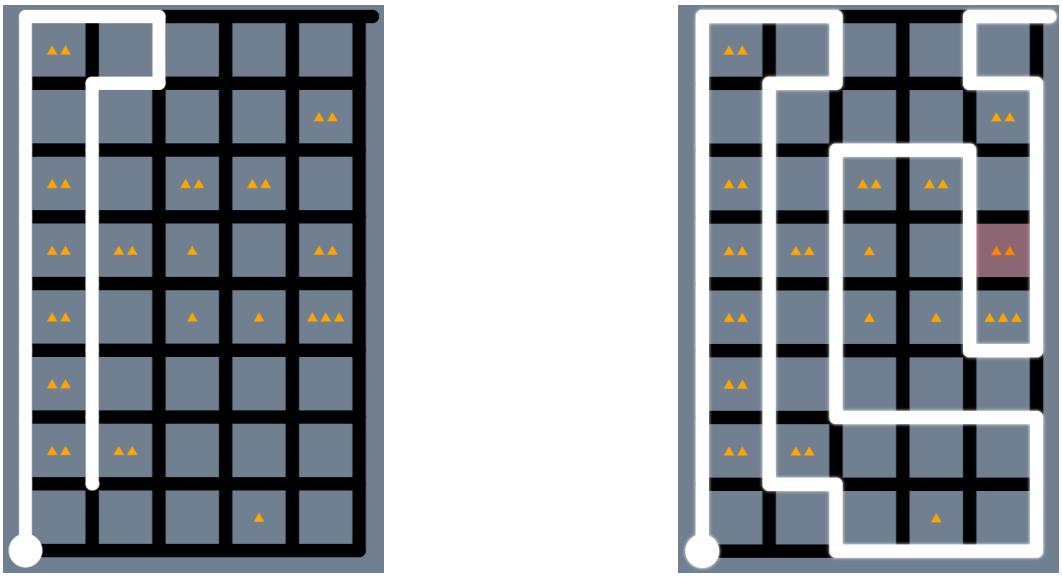

Figure 6.1: On the left is an initial partial path to complete the column of 2-triangle squares. On the right is a full solution to the puzzle.

For example consider the puzzle in Figure 6.1. This puzzle was in the dataset from Section 5.6.2 and was not solvable by Algorithm 1 with pruning using π° in under 30 minutes and 128 GBytes of memory (Figure 5.4). However this puzzle was solvable by a human using the strategy of moving up along the first column and then down along the second column to satisfy the 2-triangle squares. This strategy is similar to an opening move book in chess and to the puzzle in Figure 5.3 where the first few moves of solving the puzzle were forced. Using such an aggressively pruning predicate where only the partial path from the left part of Figure 6.1 is considered would lead to significantly speeding up search on this problem. Notice furthermore that there is only one 3-triangle square in Figure 6.1 which is likely why Algorithm 1 with pruning using π° was not able to solve it in 30 minutes.

Chapter 7

Conclusions

Searching for solutions to Witness puzzles can be combinatorially difficult. We presented an automated approach to speeding up a variation of A* via machine-learned, human-explainable predicates that predict the incompletability of a partial path. Their human explainability can offer insights to both Witness players and puzzle designers (e.g., the solution in Figure 5.3). Our approach to learning such predicates may afford the opportunity to prove that the predicates have no false positives which in turn allows them to be used to prune successor nodes in Algorithm 1 and preserve completeness. When used for pruning or sorting our predicate accelerated the baseline search by between six to seven times on our test puzzle instances. We also showed how Algorithm 1 with pruning can solve more 64 square puzzles than the baseline. We finally showed that our method sped up search more than weighted A* and Levin tree search with neural networks (Orseau and Lelis 2021).

In closing, we hope that our work presented in this thesis will encourage other game AI researchers to add Witness puzzles to their portfolio of game AI testbeds. To that end we will be sharing our code and data sets to the community upon request.

References

Abel, Zachary et al. (2020). "Who witnesses *The Witness*? Finding witnesses in *The Witness* is hard and sometimes impossible." In: *Theoretical Computer Science* 839, pp. 41–102.

Alur, Rajeev, Arjun Radhakrishna, and Abhishek Udupa (2017). "Scaling enumerative program synthesis via divide and conquer." In: *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*. Springer, pp. 319–336.

Arfaee, Shahab Jabbari, Sandra Zilles, and Robert C Holte (2011). "Learning heuristic functions for large state spaces." In: *Artificial Intelligence* 175.16-17, pp. 2075–2098.

Blow, Jonathan (2011). *Truth in Game Design*. Presented at Game Developers Conference Europe. URL: `http://gdcvault.com/play/1014982/Truth-in-Game`.

Botea, Adi and Vadim Bulitko (2022). "Tiered State Expansion in Optimization Crosswords." In: *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 79–86.

Bright, Curtis, Jürgen Gerhard, Ilias Kotsireas, and Vijay Ganesh (2020). "Effective problem solving using SAT solvers." In: *Maple in Mathematics Education and Research*. Springer, pp. 205–219.

Browne, Cameron (2013). "Deductive search for logic puzzles." In: *Proceedings of the Conference on Computational Inteligence in Games*, pp. 1–8.

Bulitko, Vadim and Adi Botea (2021). "Evolving romanian crossword puzzles with deep learning and heuristic search." In: *Proceedings of the Conference on Games*, pp. 1–5.

Bulitko, Vadim, Sergio Poo Hernandez, and Levi H. S. Lelis (2021). "Fast Synthesis of Algebraic Heuristic Functions for Video-game Pathfinding." In: *Proceedings of the Conference on Games*, pp. 1–5.

Bulitko, Vadim, Shuwei Wang, Justin Stevens, and Levi H. S. Lelis (2022). "Portability and Explainability of Synthesized Formula-based Heuristics." In: *Proceedings of the International Symposium on Combinatorial Search*, pp. 29–37.

Butler, Eric, Emina Torlak, and Zoran Popović (2017). "Synthesizing interpretable strategies for solving puzzle games." In: *Proceedings of the International Conference on the Foundations of Digital Games*, pp. 1–10.

Chen, Eugene, Nathan Sturtevant, and Adam White (2023). "Computing the Entropy of a Puzzle: Modeling Difficulty, Enjoyment and Learning." In: *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*, In Press.

Chen, Jingwei and Nathan R Sturtevant (2021). "Avoiding Re-expansions in Suboptimal Best-First Search." In: *Proceedings of the International Symposium on Combinatorial Search*, pp. 162–163.

Cook, Michael, Simon Colton, Azalea Raad, and Jeremy Gow (2013). "Mechanic miner: Reflection-driven game mechanic discovery and level design." In: *Proceedings of the Applications of Evolutionary Computation*. Springer, pp. 284–293.

Cropper, Andrew, Sebastijan Dumančić, Richard Evans, and Stephen H Muggleton (2022). "Inductive logic programming at 30." In: *Machine Learning* 111, pp. 147–172.

Cropper, Andrew and Rolf Morel (2021). "Learning programs by learning from failures." In: *Machine Learning* 110, pp. 801–856.

De Kegel, Barbara and Mads Haahr (2019). "Procedural puzzle generation: A survey." In: *IEEE Transactions on Games* 12.1, pp. 21–40.

Hart, Peter E, Nils J Nilsson, and Bertram Raphael (1968). "A formal basis for the heuristic determination of minimum cost paths." In: *IEEE transactions on Systems Science and Cybernetics* 4.2, pp. 100–107.

Hoffmann, Jörg and Bernhard Nebel (2001). "The FF planning system: Fast plan generation through heuristic search." In: *Journal of Artificial Intelligence Research* 14, pp. 253–302.

Iwashita, Hiroaki, Jun Kawahara, and Shin-ichi Minato (2012). "ZDD-based computation of the number of paths in a graph." In: *Hokkaido University Division of Computer Science TCS Technical Report*.

Khalifa, Ahmed, Scott Lee, Andy Nealen, and Julian Togelius (2018). "Talakat: Bullet hell generation through constrained map-elites." In: *Proceedings of The Genetic and Evolutionary Computation Conference*, pp. 1047–1054.

Krajňanskỳ, Michal, Jörg Hoffmann, Olivier Buffet, and Alan Fern (2014). "Learning pruning rules for heuristic search planning." In: *Proceedings of the European Conference on Artificial Intelligence*, pp. 483–488.

Krishnan, Abhijeet and Chris Martens (2022). "Synthesizing Chess Tactics from Player Games." In: *Proceedings of the Workshop on Explainable Agency in Artificial Intelligence Workshop, AAAI Conference on Artificial Intelligence*.

Levin, Leonid Anatolevich (1973). "Universal sequential search problems." In: *Problemy peredachi informatsii* 9.3, pp. 115–116.

Mouret, Jean-Baptiste and Jeff Clune (2015). "Illuminating search spaces by mapping elites." In: *arXiv preprint arXiv:1504.04909*.

Muggleton, Stephen (1991). "Inductive logic programming." In: *New generation computing* 8, pp. 295–318.

Nikoli Co., Ltd (1989). Slitherlink.

Orfanos, Spyros and Levi HS Lelis (2023). "Synthesizing Programmatic Policies with Actor-Critic Algorithms and ReLU Networks." In: *arXiv preprint arXiv:2308.02729*.

Orseau, Laurent and Levi H. S. Lelis (2021). "Policy-Guided Heuristic Search with Guarantees." In: *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 12382–12390.

Pohl, Ira (1970). "First results on the effect of error in heuristic search." In: *Machine Intelligence* 5, pp. 219–236.

Poole, David, Randy Goebel, and Romas Aleliunas (1987). *Theorist: A logical reasoning system for defaults and diagnosis*. Springer.

Richter, Silvia and Matthias Westphal (2010). "The LAMA planner: Guiding cost-based anytime planning with landmarks." In: *Journal of Artificial Intelligence Research* 39, pp. 127–177.

Stevens, Justin, Vadim Bulitko, and David Thue (2023). "Solving Witness-type Triangle Puzzles Faster with an Automatically Learned Human-Explainable Predicate." In: *arXiv preprint arXiv:2308.02666*.

Sturtevant, Nathan (2023). *HOG2*. `https://github.com/nathansttt/hog2/tree/PDB-refactor`.

Sturtevant, Nathan, Nicolas Decroocq, Aaron Tripodi, and Matthew Guzdial (2020). "The unexpected consequence of incremental design changes." In: *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 130–136.

Sturtevant, Nathan and Matheus Ota (2018). "Exhaustive and semi-exhaustive procedural content generation." In: *Proceedings of the Conference on Artificial Intelligence and Interactive Digital Entertainment*, pp. 109–115.

Summerville, Adam et al. (2018). "Procedural content generation via machine learning (PCGML)." In: *IEEE Transactions on Games* 10.3, pp. 257–270.

Thekla, Inc. (2016). The Witness.

Trivedi, Dweep, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim (2021). "Learning to synthesize programs as interpretable and generalizable policies." In: *Advances in Neural Information Processing Systems* 34, pp. 25146–25163.

Appendix A

Levin Tree Search

We illustrate how Levin tree search works (Levin 1973) on an example. Consider the search tree in Figure 2.2 augmented with a policy that gives a conditional probability of a child node given the parent node in Figure A.1.

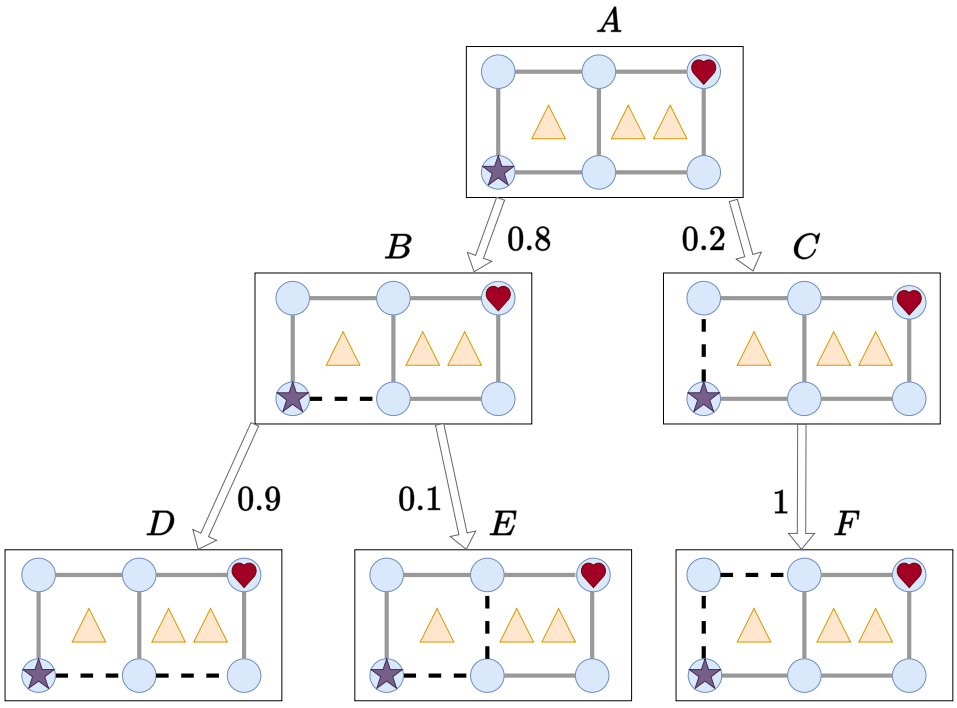

Figure A.1: Search tree with a policy for solving the puzzle from Figure 2.1.

Let n_0 be the root node of the search tree. Define the depth of a node n as $d(n') = d(n) + 1$, where n' is a child node of n, with $d(n_0) = 0$. That is, the depth of a node is the number of edges away from the root of the tree the node is. For instance, in the tree in Figure A.1, $d(A) = 0, d(B) = d(C) = 1$,

and $d(D) = d(E) = d(F) = 2$. Furthermore, define the probability of getting to a given node n recursively by the formula $\pi(n') = \pi(n|n')\pi(n)$, and $\pi(n_0) = 1$, where n' is a child node of n (Orseau and Lelis 2021). That is, the probability of a given node is the product of all the probabilities to get from the root to that node. Notice that we use the definition π as $\pi : \mathcal{P} \times \mathcal{L} \to \mathbb{R}$ from Section 2.2 since each node in the search tree has a partial path in it. For example, $\pi(A) = 1, \pi(B) = 0.8, \pi(C) = 0.2, \pi(D) = 0.8 \cdot 0.9 = 0.72, \pi(E) = 0.8 \cdot 0.1 = 0.08$, and $\pi(F) = 0.2 \cdot 1 = 0.2$.

The ratio $d(n)/\pi(n)$ is called the *Levin value* of node n. Levin tree search is a best-first search algorithm that sorts the open list by their Levin values. That is, nodes with smaller values of d/π are expanded from the open list first.

To illustrate, for the search tree in Figure A.1, we begin by adding the node A to the open list with its value of $d(A)/\pi(A) = 0/1 = 0$. We then remove A from the open list and add its children to the open list. The children of A are B and C with Levin values of

$$d(B)/\pi(B) = 1/0.8 = 1.25 \tag{A.1}$$

$$d(C)/\pi(C) = 1/0.2 = 5. \tag{A.2}$$

Of these, B has a lower value, therefore B's children, D, and E are added to the open list with Levin values of

$$d(D)/\pi(D) = 2/(0.8 \cdot 0.9) = 2.\overline{7}$$

$$d(E)/\pi(E) = 2/(0.8 \cdot 0.1) = 25.$$

The open list now contains $(C, 5), (D, 2.\overline{7})$, and $(E, 25)$. Of these, D has the lowest value, therefore we expand it first. The only child of D is the solution path shown in green in Figure 2.1, therefore, the search algorithm stops here. Notice that similar to local constraint checking (Section 2.3), Levin tree search also does not expand nodes E or F.

Orseau and Lelis (2021) added an additional factor to Levin tree search to learn a heuristic function from training data on solving other puzzles to guide the search. They propose policy-guided heuristic search (PHS) which adds a node n to the open list with its value computed by $\frac{d(n)+h(n)}{\pi(n)}$. In their work

they use a neural network with two outputs of both a probability function (policy) π and a heuristic, h. The actions in a heuristic search problem are moving up, down, left, and right, so the policy predicts the probability of taking these actions. Observe that the numerator considers the entire cost of the path (with the current length of the node plus the estimated cost to go), while the denominator computes the probability of a node using the same recursive formula as in Levin tree search.

Note the probability of node n is computed by multiplying $d(n)$ numbers together. Hence the geometric mean of these probabilities is $\pi(n)^{1/d(n)}$. Since the length of the entire solution path is estimated to be $d(n) + h(n)$, multiplying the geometric mean by itself $d(n) + h(n)$ times gives a probability of $\pi(n)^{\frac{d(n)+h(n)}{d(n)}} = \pi(n)^{1+\frac{h(n)}{d(n)}}$. Thus in an improved version of policy-guided heuristic search, PHS*, nodes are added to the open list with the value of $\frac{d(n)+h(n)}{\pi(n)^{1+\frac{h(n)}{d(n)}}}$. Notice in this formula Orseau and Lelis (2021) train a neural network for both the probabilities, $\pi(n)$, and for the heuristic, $h(n)$.

Appendix B

Prolog Files Used by Popper

In this section, I show two of the Prolog files used by *Popper* in order to learn incompletability predicates. In particular, I begin by showing the the recursive formula for how the incompletability predicate learned on a single constraint is used to evaluate all of the constraints as described in Section 4.2. I then show the bias file used by *Popper* in learning predicates.

The recursive incompletability predicate is shown below:

```
not(incompletable([], _)).
incompletable([C|T], P):- breaksConstraint(C, P); incompletable(T, P).
```

The `incompletability` predicate takes in two arguments, a list of constraints and partial path `P`. If the list of constraints is empty, then the base case says that no partial path was detected to be incompletable, and therefore returns False. The recursive case is shown in the second line. For each constraint `C`, the partial path `P` is checked to see if it breaks the single constraint through the `breaksConstraint` predicate which is learned. If so, then the entire predicate returns True showing that the partial path `P` breaks at least one of the constraints. If the partial path is not detected to break the constraint of the predicate, then the recursive call checks to see if it breaks any constraints in the rest of the list, `T`.

I now show the bias file used by *Popper* in learning the `breaksConstraint` predicate. Notice that `breaksConstraint` is the desired head predicate (meaning it is the target predicate we are trying to learn) with arity 2, meaning it takes in two arguments.

The predicate `count` is a body predicate (meaning it is used to learn the target predicate). The types of the first two arguments of `count` are lists while the third argument is an integer. Recall from before that for example, `count([1, 2], [2, 3], 1)` is True. The direction of the arguments is `in, in, out` meaning the first two arguments are input (ground) while the third argument is output (unground).

Below all of the body predicates are shown with each respective arity, type, and direction. Notice there is also the addition of the constants, which are defined as the integers 1, 2, and 3 (for the triangle constraints).

```
head_pred(breaksConstraint, 2).
body_pred(path, 2).
body_pred(count, 3).
body_pred(square, 3).
body_pred(greaterThan, 2).
body_pred(gte, 2).
body_pred(len, 2).
body_pred(adjacent, 2).
body_pred(notAdjacent, 2).
max_vars(7).
type(breaksConstraint, (element, element)).
type(path,(element, list)).
type(count,(list, list, int)).
type(square, (element, int, list)).
type(greaterThan, (int, int)).
type(gte, (int, int)).
type(len, (list, int)).
type(adjacent, (element, element)).
type(notAdjacent, (element, element)).
constant(one, int).
constant(two, int).
constant(three, int).
direction(breaksConstraint, (in, in)).
```

```
direction(path,(in, out)).
direction(count,(in, in, out)).
direction(square, (in, out, out)).
direction(greaterThan, (in, in)).
direction(gte, (in, in)).
direction(len, (in, out)).
direction(adjacent, (in, in)).
direction(notAdjacent, (in, in)).
body_pred(P,1):- constant(P,_).
type(P,(T,)):- constant(P,T).
direction(P,(out,)):-constant(P,_).
```

Appendix C

Example of False Positives with a Predicate

As we mentioned in Section 4.2, *Popper* can learn a predicate with false positives. In this Appendix, we analyze the predicate that was learned on our running example puzzle in Figure 2.1:

π^{ex}`(A,B):- square(B,D,E),path(A,C),count(E,C,F),greaterThan(F,D).`

π^{ex}`(A,B):- notIncident(A,B),path(A,D),one(C),len(D,C).`

The first clause of the predicate as mentioned in Section 5.9 is equivalent to local constraint checking. Of the incompletable partial paths (the positive examples) in Table 4.3, local constraint checking makes the `incompletable` predicate return True for paths `p5`, `p6`, `p7`, `p8`, and `p9`. In the second clause, by `path(A,D)` we know that `A` is a partial path across a list of edges `D`. By `len(D,C)` and `one(C)` we know that `D` has one edge in it. Furthermore by `notIncident(A,B)` we see that the head of path `A` (a vertex in the graph) is not incident to any edge in square `B`. Therefore the second clause returns True for any partial paths of length 1 that are not incident to square `B`.

Note that the second clause of π^{ex} returns True for path `p3` and `c2` since `p3` is not incident to square `c2` and also the length of path `p3` is equal to one. Furthermore, this predicate does not cover any of the negative examples, since path `p2` is adjacent to both squares, while path `p4` has length of two. Thus this predicate satisfies all the positive examples while satisfying none of the negative examples for the puzzle in Figure 2.1 and so `Popper` returns π^{ex} as the solution by its learning algorithm described in Section 4.2.

However, the second clause of predicate π^{ex} may have false positives. For example, consider the puzzle in Figure C.1. The second clause of π^{ex} predicts that the partial path e_3 is incompletable since the head of the path is not incident to square c_2 and the length of the path is 1. However, the only solution to this puzzle is $[e_3, e_6, e_4, e_2, e_5]$. Therefore if predicate π^{ex} was used for pruning, it would prune the only valid solution of this puzzle. Therefore Algorithm 1 with pruning using π^{ex} is not complete since π^{ex} has false positives.

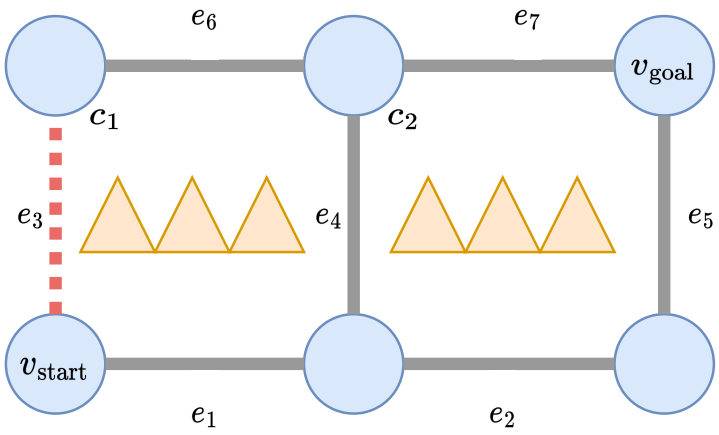

Figure C.1: An example of a partial path on which π^{ex} has a false positive. The red dashed line shows the partial path that π^{ex} incorrectly predicts as incompletable.

Appendix D

Puzzle Generators

In this Appendix, I show the algorithms used to generate the puzzles used for our evaluation (Chapter 5).

Algorithm 3: Generate a puzzle instance by placing triangles randomly.

input : puzzle dimensions m, n
output: puzzle p

```
1  repeat
2      p ← empty puzzle(m × n)
3      goal ← random location on peripheral of puzzle
4      k ← U({1, ..., ⌊mn/2⌋})
5      S ← k randomly picked squares from the grid
6      foreach s ∈ S do
7          q ← U({1, 2, 3})
8          put q triangles into square s
9      ℓ ← Algorithm 1's solution to p
10 until ℓ ≠ ∅
11 return p
```

The first generator places triangles randomly on the grid (Algorithm 3). In line 2 we create an empty $m \times n$ Witness puzzle. We randomly initialize the goal location to be any vertex that is peripheral to the graph (meaning it has edge degree less than 4) such that $v_{\text{start}} \neq v_{\text{goal}}$ in line 3. The starting location, v_{start}, is always fixed to be the bottom left corner.

In lines 4 and 5 we pick the number of squares containing triangles uniformly randomly between 1 and half of all squares (in the spirit of the challenge room maze in the actual game (Thekla, Inc. 2016)). We then randomly select

such squares in the grid. For each of these squares we uniformly randomly assign between 1 and 3 triangles to it in lines 7 and 8. After generating the puzzle we use Algorithm 1 with pruning using π_{baseline} to determine if the puzzle is solvable (line 9). If it is not, we restart the generation process (i.e., go to line 2).

The second puzzle generator (Algorithm 4) starts with a path from the start vertex to the goal vertex and then populates the empty grid with triangles so that the path becomes a solution to the puzzle (in the spirit of work by Blow (2011)).

Algorithm 4: Generate a puzzle instance from a path.

input : puzzle dimensions m, n
output: puzzle p

```
1 p ← empty puzzle (m × n)
2 ℓ ← random path from start to goal
3 S′ ← all squares in p with at least one edge in ℓ
4 k ← U({1, ..., |S′|})
5 S ← k randomly picked squares from S′
6 foreach s ∈ S do
7 |   q ← count s's edges in ℓ
8 |   put q triangles in square s
9 return p
```

Line 2 generates a path from the start to the goal vertices that does not contain repeating vertices. We then find all squares that have at least one edge in ℓ (line 3) and uniformly randomly sample a number of these squares (lines 4-5). For each of these squares, we add to the puzzle the number of edges it has in ℓ so that ℓ is a solution to the puzzle.

Appendix E

π° Computed by Prolog

Instead of converting the predicate to Python, we now show the results of keeping the local constraint checking (written in Python) followed by using the learned predicate computed by Prolog.

The time speedup and expansion speedup of with π° computed by Prolog (denoted $\pi^\circ_{\text{prolog}}$) are shown below:

$$\text{speedup}_t(\pi^\circ_{\text{prolog}}, \mathcal{P}_{\text{test}}, \text{sort}) = 1.15, \tag{E.1}$$

$$\text{speedup}_{\mathcal{E}}(\pi^\circ_{\text{prolog}}, \mathcal{P}_{\text{test}}, \text{sort}) = 6.27. \tag{E.2}$$

Notice that the expansion speedup is the same as before since $\pi^\circ_{\text{prolog}}$ is functionally equivalent to π° computed in Python and by Theorem 3 we know the expansions are the same for pruning and sorting. On the other hand, the time speedup is lower now since calling Prolog from Python is slower than running the Python-converted predicate.