Chair of Machine Learning and Reasoning (Computer Science 6)

RWTH AACHEN UNIVERSITY

Diese Arbeit wurde vorgelegt am Chair of Machine Learning and Reasoning.
The present work was submitted to the Chair of Machine Learning and Reasoning.

Sketches and IW(1) exploration in Atari Games

Sketches und IW(1) exploration in Atari Games

Bachelor of Science Thesis
Bachelorarbeit

submitted by/von

Mehta, Aaditya
445387

Supervised by Niklas Jansen, M.Sc.

1st Examiner Prof. Hector Geffner, Ph.D.

2nd Examiner Prof. Dr. rer. nat. Christopher Morris

Aachen, February 5, 2026

Acknowledgements

I would like to express my heartfelt gratitude to Niklas Jansen, my supervisor, for his invaluable assistance, support, and motivation throughout my thesis. His insights and understanding have significantly shaped my research and enhanced my grasp of the subject. Furthermore, I wish to convey my appreciation to Prof. Hector Geffner for allowing me the opportunity to explore this fascinating topic and for his valuable feedback during the development of this thesis. Special thanks to Prof. Blai Bonet for providing the Iterated Width (IW) algorithm implementation in the Arcade Learning Environment (ALE), which was crucial for this research. I would like to recognize the ALE team for their outstanding platform, which enabled the experimental aspect of this work. I am deeply thankful to my family for their constant encouragement and faith in me. Their support has been an ongoing source of inspiration. Finally, a special thanks to my dear friends—Hannah Tousiannt and Maximillian Gutwerk—for their camaraderie, thoughtful discussions, and for providing much-needed respite during this journey.

Abstract

Algorithms like Iterated Width (IW) and Rollout-IW have shown impressive results in environments such as Atari games. What differentiates these algorithms from others is the novelty-based state pruning. However, in sparse-reward contexts, such as Adventure and Montezuma's Revenge, their performance suffers. The main hurdles arise from the domain's high subgoal width and non-serializable goals. In other words, one of the key problems is interference, which makes it impossible to achieve subgoals in strict sequences. In turn, this causes exponential search complexity. To solve this, we employ sketch [10] which is a domain-knowledge encoding framework. This framework systematically breaks down the main task into smaller subproblems by defining subgoals. As a result, we use sketch rules rather than rewards in width-based planners. Thus, we employ Serialized Iterated Width with Rules (SIWR) [13], which concentrates exploration on high-level subgoals instead of atomic rewards. Our entire implementation as well as the log files, are available at `https://github.com/Quickblade22/Sketches--IW1/`.

Contents

1 Introduction **1**

2 Background **3**

2.1 Partially Observable Markov Decision Process (POMDP) Framework 3

2.2 The Arcade Learning Environment 4

2.2.1 Adventure 5

2.3 Basic Pairwise Relative Offsets in Space and Time (BPROST) Visual Features . 6

2.4 Classical Plannings 8

2.5 Formal Definition of Width and Width-based Planning 9

2.6 Sketch and Sketch Rules 11

3 Related Work **13**

4 Approach **15**

4.1 Problem Definition 15

4.2 Width-Based Planning with Policy Sketch 17

4.2.1 Adaptions of Serialized Iterated Width with Rules (SIWR) for Arcade Learning Environment (ALE) 17

4.2.2 Features for Iterated Width (IW) Searches 18

4.3 Sketch and Problem Decomposition for Adventure 19

4.3.1 Sketch Features for Adventure 19

4.3.2 Sketch Implementation for Adventure 21

4.3.3 Formal Properties of the Sketch for Adventure 22

4.4 Practical Considerations and Limitations 27

5 Implementation **29**

5.1 SIWR Implementation 29

5.1.1 Implementation Framework 29

5.1.2 Implementation Details 30

5.1.3 Serialized Iterated Width (SIW) 30

5.1.4 SIWR 32

5.1.5 Modifications 35

5.2 Sketch Implementation and Sketch features 37

5.2.1 Sketch Features Extraction 37

5.2.2 Sketch Rules Implementation 39

6 Evaluation **40**
6.1 Budget Rationale and Parameters 40
6.2 Comparative Setup 41
6.2.1 Key Findings and Interpretation 44
6.3 Problems and Solutions 45
6.4 Consequences 47

7 Conclusion **49**

A Appendix **51**

List of Acronyms **61**

List of Symbols **62**

List of Figures **63**

List of Tables **66**

List of References **67**

1 Introduction

Developing Artificial Intelligence (AI) algorithms with mastery over a wide spectrum of skills, such as control, strategizing and dealing with unexpected hindrances, is a complex challenge. To test such AI agents, general game-playing, especially fast-paced video games, serves as an appropriate testbed, as the agents have to make competent decisions and utilize various skills under stringent time constraints [15]. Thus, the ALE, an emulator for the classic Atari 2600 platform, is a cornerstone benchmark for testing such algorithms (agents) [8]. Even though modern Reinforcement Learning (RL) techniques have seen notable success in this domain [23], its peculiar sample inefficiency, requiring billions of interactions to master sparse-reward games like Montezuma's Revenge, remains a major barrier to real-world applicability [4, 6].

Hence, this limitation has redirected attention toward planning algorithms that emphasize effective online search. Among the most promising are width-based planning algorithms, such as IW and its variants [7, 18]. IW works by strategically pruning search paths that do not reveal new features, allowing it to explore high-dimensional spaces (such as raw pixels) efficiently [19]. The width-based approaches like IW, Rollout-IW, and other variants have been shown to outperform methods such as Monte Carlo Tree based on Upper Confidence bounds applied to Trees (UCT) and Deep Q-Network (DQN) [7, 17].

Especially in sparse reward settings, width-based planners have demonstrated superior performance compared to traditional RL methods [7, 19]. However, a critical limitation persists in environments with **sparse, delayed rewards**—such as Adventure and Montezuma's Revenge [1, 3]—where traditional width-based methods often fail due to two core limitations:

- **Non-serializable Goals:** Progress requires intricate subgoal sequences (exempli gratia (e.g.), retrieve key → unlock door → avoid enemies). SIW decomposes goals sequentially, but fails when subgoals interfere or require concurrent achievement [10]. In these cases, sketch serializes non-serializable goals through rules, which enforce a strict partial ordering of subgoals (e.g., "unlock door" preconditioned on "have key"), ensuring sequential achievability. For instance, the rule have(key) → unlock(gate) does not attempt to 'unlock gate' before having key and thus handles interference between simultaneous goals. This is the case in games like Montezuma's Revenge in which it is impossible to order the subgoals "retrieve key" and "avoid enemies" linearly without either interference or risk-aversion [7].
- **High Subgoal Width:** Individual subgoals may exceed the problem's effective width (e.g., $w > 2$), making polynomial-time solutions with IW intractable [18]. These limitations

manifest in SIW [18], which greedily achieves goals one-at-a-time using IW searches, when subgoals exhibit high width or interference[7, 19].

To address this, we use policy sketch–a lightweight expressive framework for encoding domain knowledge into width-based planning [10, 13]. Mimicking human problem-solving (e.g., getting out of a locked room by resolving smaller objectives such as locating keys and deactivating traps), sketch breaks down difficult problems into sub-problems of low width [13]. This method, formalized in classical planning, allows for reducing a problem to sequences of low-width subproblems that can be solved in polynomial time [10, 13]. For instance, in Adventure Figure 2.1, sketch rules might encode $\neg have(key) \rightarrow have(key)$, because one needs the key to open certain rooms of the game. In addition, sketch is ideal for games where hand-designed structures (e.g. BPROST) or learned representations (e.g. Variational Autoencoder (VAE)-based symbols) represent critical state abstractions, although these techniques require domain-specific knowledge [7, 12].

Thus, we employ SIWR by using sketch, a collection of rules over the state features, rather than a reward function in IW. This method focuses exploration on high-level subgoals rather than atomic rewards, which helps mitigate vanilla width-based methods' aimless exploration of the search space in sparse-reward settings [13]. Therefore, the problem's search space is split into smaller, manageable subproblems where aimless exploration is more likely to yield useful results [13]. Though not fully domain-agnostic, SIWR should boost performance in target games like Adventure and Montezuma's Revenge, where sparse rewards have historically thwarted vanilla width-planners. By unifying classical SIW's power with sketch's structured guidance, we enable our algorithm to perform guided search using sketch rules instead of sparse rewards. In the following we interchangeably use the terms agents and algorithms.

In this thesis we hope to advance planning in partially observable environments by implementing SIWR for the ALE domain. The core contribution is the development of a symbolic sketch, comprising carefully designed sketch features and rules for the Adventure game, which abstracts the environment to guide the planner. Therefore, in the subsequent chapters we will first review the necessary background in Chapter 2 and talk about related work in Chapter 3, followed by a detailed presentation of our SIWR approach and the Adventure-specific sketch rules and features in Chapter 4 and Chapter 5. This includes formal proofs of the sketch's properties, ensuring its theoretical validity, as well as necessary modifications needed for practical implementation. Then, we evaluate our method in Chapter 6 against baseline SIW, demonstrating its effectiveness in navigating Adventure's sparse-reward landscape. Finally, we conclude with a summary of findings and future research directions in Chapter 7.

2 Background

We now cover the necessary background on POMDP, ALE, classical planning, width-based planning, and sketch.

2.1 POMDP Framework

Partially Observable Markov Decision Process (POMDP) is a mathematical framework for modeling decision-making problems where the environment is partially observable (and stochastic) [2, 16, 25, 26]. Formally, it is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, R, \gamma, \Omega, \mathcal{Z})$ [2, 16, 21, 25].

- $\mathcal{S}$ is a finite set of states, the environment can be in.
- $\mathcal{A}$ is a finite set of actions available at any given state.
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$: The state transition probability, where $\mathcal{P}(s, a, s')$ is the transition probability from state s to state s' after taking action a. For the case of having deterministic environment, this transition probability function reduces to $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. Here, $T(s, a) = s'$ indicates that taking action a in state s leads to state s'.
- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is where $R(s, a)$ gives the immediate reward received after taking action a in state s.
- $\gamma \in [0, 1)$: A discount factor specifying the relative importance of future rewards. When the weight is close to one, future rewards are as important as immediate ones.
- Ω is a finite set of observations that the agent can observe from the environment.
- $\mathcal{Z} : \mathcal{A} \times \mathcal{S} \times \Omega \to [0, 1]$ is the observation probability function. $\mathcal{Z}(a, s', o)$ represents the probability of receiving observation o when taking action a and transitioning to state s'. In deterministic environments, this observation function simplifies to a mapping $O : \mathcal{A} \times \mathcal{S} \to \Omega$, where $O(a, s') = o$ represents that performing action a and reaching state s' will result in observation o with probability 1.

Another feature of POMDP is if the functions, $\mathcal{P}$, $\mathcal{Z}$ and R, are time dependent or not. The main difference between these both variants is whether the functions are stationary or evolve over time steps [2, 16, 25]. For instance, if the order in which one visits certain states matters, then the functions will be time-dependent [2, 16, 25]. The Atari games we will be working with are stationary environments, thus the best action to take does not depend on the time step [2, 16, 25]. Thus, so far we have only defined the time-invariant notation, where these functions do not depend on the time step [2, 16, 25].

2.2 The Arcade Learning Environment

ALE[1] is a software platform developed to facilitate research in general artificial intelligence, particularly in the domains of reinforcement learning and planning [7, 19, 23]. By using the Stella emulator, it emulates a variety of the Atari 2600 games and offers a standardized interface [8].

Primarily, ALE serves as a benchmark for evaluating agents in terms of general competence across diverse tasks, ranging from reflex-based control to long-term strategic planning [8]. Furthermore, ALE has gained significant traction in both academic and industrial settings as researchers employ it to evaluate deep RL algorithms, including DQN, as well as classical and width-based planners [7, 12, 24]. The value of this paradigm is predicated on the extensive array of games that provide varying degrees of complexity, partial observability, small rewards, and planning horizons [8].

ALE offers two primary observation modes [1]:

1. **The screen mode** is defined as a raw visual output consisting of 210×160 pixel frames, with each pixel encoding one of the 128 distinct colors in grayscale. The screen image we have is just a one-dimensional array of length $210 \times 160 = 33,600$ bytes. One could instead use the Red Green Blue (RGB) representation of each pixel (id est (i.e.), three components-per-pixel representing red-green-blue) instead of the grayscale. However, in our work, we utilize the grayscale screen representation, which is why we do not elaborate more on the RGB mode anymore.
2. **The Random Access Memory (RAM) mode** is defined as a 128-byte vector that reflects the internal state of the Atari emulator. This representation is characterized by its compact nature and its ability to convey semantically meaningful information.

Furthermore, the selection of observation mode carries profound ramifications for the implementation of planning algorithms. Foremost, RAM states are characterized by their lower dimensionality and direct manipulability. Conversely, screen states necessitate the implementation of perception modules or feature extractors. These modules have been demonstrated to exhibit a higher degree of alignment with human-like visual understanding. Additionally, they provide a more equitable benchmark against RL agents that have been trained on pixel inputs. Moreover, another significant aspect of ALE is its ability to provide two modes for each of the games, which differ in the possible action space:

1. **Compact Actions:** This mode offers a limited set of actions, which are designed to be more general and abstract. For instance, it could be just the four directions alongside fire and no action [1].

[1]This framework has been released as a free, open-source framework, under the *GNU General Public License*. The latest version of the ALE can be found at `https://github.com/Farama-Foundation/Arcade-Learning-Environment`, where a general description of the provided games and general framework is found in [1].

2. **Expanded Actions:** In contrast, this action set provides a more granular set of actions, often around 40, which allows for more precise control over the agent's behavior. At the core, these actions are combinations of compact actions. Consequently, now the agent can perform multiple compact actions as one action. For instance, the agent can move up left or move up left and fire with a single action [1].

In the context of planning, ALE facilitates proactive search by simulating forward dynamics in the game (i.e., black-box state transitions resulting from an action) [1, 8]. This enables planners, such as IW, to exploit the environment in a deterministic manner without requiring the learning of a model, unlike in model-based RL approaches [7, 18].

2.2.1 Adventure

Adventure is a game for Atari 2600 gaming console [1, 3], which features a simple-cube like avatar moving across a series of 2D rooms which form a kingdom-like environment, as seen in Fig 2.1. Furthermore, the game environment consists of multiple structures like a vast maze which stretches over multiple screens and the golden and black castle which each have a closed gate. Even though the background patterns, representing walls, are different from room to room, the movable area for the cube is always a specific grey colored area (Figure 2.1). Each player has only one life to complete the game, because once a dragon eats the avatar, the game ends. As a result, the player must start the game over again in order to reset this progress [3]. The main goal of the game is to recover the stolen chalice and return it to the golden castle. While trying to achieve this goal, the player has to avoid or kill the dragons, trying to eat one, navigate through a vast maze, and utilize several key items, each with unique properties[3]:

1. **yellow key:** Opens the golden castle door.
2. **black key:** Opens the black castle door.
3. **bridge:** Used to cross the maze walls.
4. **sword:** Used to slay the dragons.

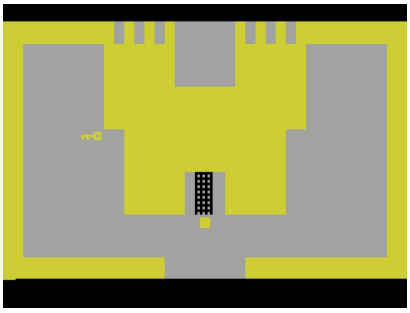


Figure 2.1: Atari 2600's *Adventure* (1979) initial game screen. One can see a yellow cube, the player, standing in front of the golden castle's gate. Also, the gate's key, the yellow key, is visible to the left of the castle.

Considering the reward function, the only positive reward is given when the main objective of the game is fulfilled [1, 3]. Thus, to receive the first positive reward, the players must complete the game, which in turn requires a series of complex sequences of actions. Therefore, a simple novel-based exploration in these games makes progress virtually impossible as there is practically no reward function to guide the search. Moreover, the core challenge lies not in reflexes but in structured exploration, inventory management, and logical deduction as the game functions as a puzzle-box, where progress is gated by the application of specific tools in specific locations. Hence, it demands a form of symbolic reasoning and memory, which pure, model-free reinforcement learning struggles to achieve.

2.3 BPROST Visual Features

Figure 2.2: Basic Features applied to *Adventure*, an Atari 2600 game, where a singular feature example of the feature set is shown in magenta.

BPROST is a rich set of handcrafted binary features derived from Atari pixel frames [7]. They consist of **Basic features**, **Basic Pairwise Relative Offsets in Space (BPROS)** and **Basic Pairwise Relative Offsets in Time (BPROT)** features, which we will describe below [7].

Basic features divide the screen into smaller pixel patches, called tiles, where it detects the presence of colors as seen in Figure 2.2.

BPROT captures the relative motion across frames (e.g. Figure 2.4). To capture temporal relationships, the offsets between tiles across two consecutive decision points' frames are compared, where each tile contains a pixel with a particular color. Decisions are made at intervals, whose value is configurable, and is typically set to 15 frames in Atari games [7]. More precisely, for each pair of colors (k_1, k_2) and offset (i, j), there exists one binary feature that is activated whenever a pixel of color k_1 can be found in some tile at time t and a pixel of color k_2 appears on the tile shifted by (i, j) at time $t + 1$.

BPROS encodes relative spatial positions between colors in two tiles (e.g., Figure 2.3). Hence, this captures spatial relationships in a single frame by comparing the offsets between tiles each containing a pixel with a particular color. Formally, for each color pair (k_1, k_2) and offset (i, j), there is a binary feature that is active if a pixel of color k_1 is present in some tile (c, r) and a

pixel of color k_2 is present in tile $(c+i, r+j)$ [7]. Here, the (c, r) is the column and row of tile in screen grid [7].

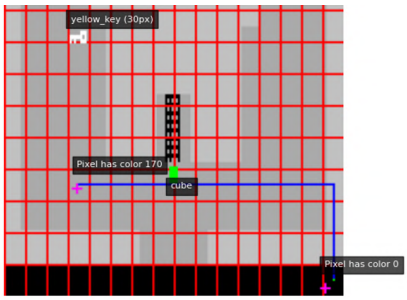
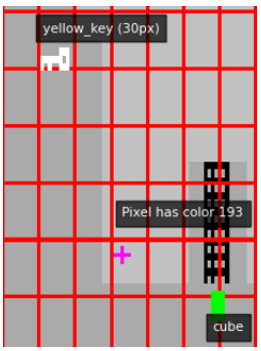
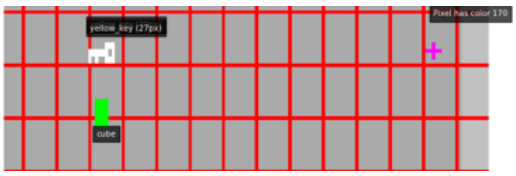


Figure 2.3: An example of BPROS applied to *Adventure*, an Atari 2600 game, where two pixels' colors from different tiles are compared with each other. One pixel's color is 0, and the other pixel's color is 170. These pixels are shown in magenta, and their relative offset is shown in blue. The red grid-like lines running horizontally and vertically represent the tiles in which the screen is divided.



(a) An example of *Adventure*'s screen at time t



(b) An example of *Adventure*'s screen at time $t+1$

Figure 2.4: An example of BPROT applied to *Adventure* where two pixels' colors from different tiles across time $(t, t+1)$ tile-offset are compared with each other. These pixels are shown in magenta.

Although BPROST is high-dimensional, IW's novelty pruning is well-suited to sparse inputs [7]. Thus, BPROST is used as a feature basis for planning algorithms like IW and Rollout-IW in Atari domains, enabling direct comparison to RL agents trained on raw pixels [7]. This enables pixel-based planning, but also faces limitations:

1. **Non-Markovianity:** Temporal features (BPROT) rely on previous states, violating Markov assumptions, and require storing previously seen screen data [7]. Additionally, if one were to use BPROST how an enemy moves across the screen, one would need to compare positions across multiple frames, violating Markov assumptions. Here, one would compare the BPROS features to identify the enemy's position and BPROT features to track its movement across frames [7].
2. **Background Subtraction:** Dynamically identifies static pixels by sampling 100 random actions. This is done by doing couple random actions from the initial state and

marking pixels that remain unchanged as static [7]. The main idea behind this is that "[a] background pixel is one that preserves its color in all reachable screens [...]" [7]. Thus, this reduces noise but fails in dynamically changing backgrounds [7]. However, in Adventure where each room has a different background pattern and color, this makes static identification challenging, which leads to two cases where this might be a problem. Firstly, one might not identify certain pixels as static, as they have distinct colors in different rooms, but are always part of the background. Secondly, certain pixels could be marked as static, due to the rooms one saw during the random sampling phase, but they might not be static in other rooms with different background patterns. Thus, when backgrounds change dynamically, the system may misidentify static elements, leading to incorrect assumptions about the environment.

2.4 Classical Plannings

Classical planning provides a formal framework for modeling deterministic, fully observable sequential decision-making problems [14]. It serves as the foundation for many width-based planning algorithms, including IW and its variants [18]. A classical planning problem can be described in two related forms: an explicit state model and a compact Stanford Research Institute Problem Solver (STRIPS) representation [14].

The explicit state model is represented as a tuple $S = (\mathcal{S}, s_0, \mathcal{S}_G, \mathcal{A}, f, c)$, with $\mathcal{S}$ being a finite discrete set of states representing all possible configurations of the environment [14, 18]. Furthermore, $s_0 \in \mathcal{S}$ is the known initial state and $\mathcal{S}_G \subseteq \mathcal{S}$ is the non-empty set of goal states [14, 18]. Moreover, $\mathcal{A}$ is a finite set of actions available to the agent, where $\mathcal{A}(s) \subseteq \mathcal{A}$ is the set of actions applicable in state s [14]. The deterministic state transition function $f : S \times A \to S$ is where $f(s, a) = s'$ indicates that taking action a in state s leads to state s' [14, 18]. Finally, the cost function $c : S \times A \to \mathbb{R}_{\geq 0}$ assigns a non-negative cost to each action in each state [14, 18].

A plan is an action sequence $\pi = [a_1, a_2, \dots, a_n]$ that generates a state sequence $[s_0, s_1, \dots, s_n]$ where $\forall i \in [1, \dots n] : s_i = f(s_{i-1}, a_i) \wedge s_n \in S_G$ [14, 18]. This plan transforms the initial state s_0 into a goal state s_n by applying the action sequence [14]. The plan's cost is the sum of the individual action costs: $\text{cost}(\pi) = \sum_{i=1}^{n} c(s_{i-1}, a_i)$ [14]. Thus, an optimal plan achieves the goal with the minimum total cost [14, 18]. A simplification is to assume unit costs for all actions, $\forall s \in \mathcal{S}, a \in \mathcal{A} : c(s, a) = 1$ [18]. Consequently, the cost of a plan is simply its length, i.e. the number of actions in the plan [18]. We will be using this simplification in our work.

In practice, the explicit state model is often too large to enumerate. Instead, planning problems are compactly represented using languages like STRIPS [7, 14, 19]. A classical planning problem is typically expressed in a STRIPS-like representation as a tuple $P = (F, I, O, G)$ [7, 19], where F is a finite set of fluents, boolean variables, that describe the state of the world. The initial state $I \subseteq F$ is the set of fluents true at the beginning of the planning problem and the goal $G \subseteq F$ is

the set of fluents that must be true to consider the problem solved. Furthermore, O is a finite set of actions, where each action $a \in \mathcal{A}$ is defined by a tuple $(pre(a), add(a), del(a))$. Here, $pre(a) \subseteq F$ is the set of preconditions that must be satisfied for action a to be applicable. The set $add(a) \subseteq F$ contains the fluents that become true after executing action a, while $del(a) \subseteq F$ contains the fluents that become false after executing action a [7, 19]. Finally, STRIPS uses boolean variables, but classical planning languages also support multivalued variables with finite domains, which we will not elaborate on as we do not utilize them [14].

A STRIPS problem P encodes the state model $S(P)$ implicitly as follows [14, 18]. A state $s \in \mathcal{S}$ is a subset of F, representing the fluents that are true at a given time [14]. The initial state is $s_0 = I$, and the goal states are $S_G = \{s \in \mathcal{S} \mid G \subseteq s\}$[14]. An action $a \in \mathcal{A}$ is applicable in state s if $pre(a) \subseteq s$[14]. Applying a results in a new state $s' = (s \setminus del(a)) \cup add(a)$ and the action cost are uniformly 1 [14]. A plan is a sequence of actions $\pi = [a_1, a_2, \dots, a_n]$ that transforms the initial state I into a state satisfying the goal G [18]. The resulting number of states in $S(P)$ is $2^{|F|}$, which is often exponentially large in the number of fluents [14, 18].

2.5 Formal Definition of Width and Width-based Planning

In the context of width analysis, a tuple t is a conjunction of at most k fluents (atoms). The width $w(P)$ of a planning problem P is formally defined as follows [18]:
Definition 1 (Admissible Subgoal Sequence and Width)**.** A sequence of subgoals $t_0, t_1, \dots, t_n$ (each a conjunction of atoms) is *admissible* for a Problem P if:

1. t_0 holds in the initial state s_0,
2. Optimal plans for t_n are optimal for P,
3. Any optimal plan for t_i can be extended to an optimal plan for t_{i+1} by adding one action.

The *width* $w(P)$ is the smallest integer k such that there exists an admissible sequence with $|t_i| \leq k$ for all i. Here $|t_i|$ is the number of atoms in subgoal t_i that are true.

Width-based planning algorithms exploit the *width* property of planning problems for efficient search. Moreover, SIW is a methodology that extends IW to conjunctive goals by solving subgoals sequentially [18]. IW is a state-pruning search algorithm, which was initially introduced for classical planning's domains [18]. IW employs a breadth-first search on these states and prunes states using the idea of "state" novelty [18]. Formally, for state s with novelty$(s) = k \in \mathbb{N}$, k is the size of the smallest feature set true in s for the first time in the IW search.

Thus, the objective of this method is to restrict the search to states that present novel combinations of features. This brings us to IW(k), which explores the state space using breadth-first search, pruning states whose novelty exceeds k [10, 18]. It's search complexity is in $O(|F|^k)$, yielding exponential savings over brute-force breadth-first search [7, 18]. In our ALE setting, the role of the feature set F in classical planning is fulfilled by the BPROST feature set.

These features serve as the atomic boolean predicates for novelty computation in IW searches. Concerning, the cost of each action, we assume unit costs for all actions in ALE [18].

The fundamental principle underpinning this approach is that most solvable problems have a fixed width. A substantial body of classical planning benchmarks demonstrates an effective width w, thereby suggesting that IW(w) can solve them in polynomial time [10, 18]. Classical domains (Blocks World, Logistics) exhibit subproblems with width $\leq$ 2, explaining IW's effectiveness [10, 18]. For example:

- CLEAR(X) in Blocks World: width 1, but ON(X,Y): width 2
- Delivery tasks in Logistics: width 1-2 depending on goal structure.

Before running SIW, one must define an upper limit $n \in \mathbf{N}$ for the width of the subgoals. Then firstly, SIW starts with the initial state s_0 and calls IW(w) for $w \in [1, \dots, n]$ until it finds a state s where at least one subgoal is fulfilled. Now, SIW sets $s_0 = s$ as the new initial state and again calls IW(w) for $w \in [1, \dots, n]$ until it finds a state where more subgoals are fulfilled. This is the recurrent phase of SIW, and it continues until all subgoals are achieved. SIW has been demonstrated to be effective in scenarios involving serializable goal structures. However, its performance is hindered when subgoals interfere with one another or exhibit a high individual width [18]. Therefore, SIW is incomplete.

To address these limitations in real-time or large state spaces like ALE, Rollout-IW was introduced [7]. Consequently, the system replaces breadth-first search with randomized rollouts from the current state, retaining states only when novel features are encountered. Additionally, Rollout-IW enables decisions in bounded real-time (typically 5-15 seconds per action) by limiting search depth and rollouts [7]. Empirical studies show this finds novel states with 87% success in sparse-reward domains within 5 seconds [7]. On that account, this enables the execution of decisions in near real-time, thereby rendering Rollout-IW particularly well-suited for pixel-based ALE planning [7].

On the one hand the benefits of IW family are that it resolves low-width problems in polynomial time via an effective exploration of state spaces [7, 13, 18, 19]. Additionally, this family of algorithms are model-based planners that do not require learning transition models, as they operate directly with state transitions. Moreover, as width-based planners, they do not require transition models or goal specifications, but just a goal and initial state [19]. On the other hand, performance degrades on problems with large or poorly-structured feature spaces [7, 13, 18, 19]. Thus, these algorithms are sensitive to feature design and struggle with non-serializable or high-width subgoals [7, 13, 18, 19]. Therefore, IW(w) has exponential complexity in w, as it expands $\mathbf{O}\big((m \times D)^w \times b\big)$ nodes where m is the number of problem variables, D is their domain size and b is the branching factor [19]. Finally, SIW is incomplete as it lacks robustness to guarantee a solution[19]. The main reasons are its greedy search, which may lead to dead ends, or subproblems with a width exceeding n. In particular, once SIW sets a state, which

fulfilled a new subgoal, as it root state, it cannot change this even if the new state might be a dead end [19].

2.6 Sketch and Sketch Rules

To address limitations in SIW and exploit subgoal structure in planning domains, sketch was proposed [10]. A sketch is a high-level plan abstraction defined as a set of sketch rules:

1. **Syntax:** $C \longrightarrow E$, where C is a context and E is an effect (subgoal condition).
2. **Semantics:** If C holds, then the agent should find a path to a state where E is fulfilled.

Each rule $C \rightarrow E$ encodes domain-specific knowledge, where C is a precondition and E is a high-level effect. The agent prioritizes achieving E when C holds, reducing exploration to structured subgoals rather than atomic rewards [10, 13]. Sketch R_Φ is a collection of sketch rules over features Φ. Moreover, the sketch is *well-formed* if Φ distinguish goals in problem class Q and R_Φ induces an irreflexive partial order on feature evaluations, ensuring finite subgoal chaining[10, 13]. The last condition prevents cyclic dependencies between subgoals and is often called **terminating**.

Another important aspect of **sketch** is the **Subgoal States** $G_r(s)$, where $r \in R_\Phi$ is a sketch rule and s is a state [10, 13]. Intuitively, it means when C of the sketch rule r holds in s, $G_r(s)$ is the set of states reachable from s where E of r is true. So it provides intermediate targets toward the ultimate goal [10]. Formally, for sketch rule $r : C \rightarrow E$ and state s [10]:

$$\textbf{Subgoal states } G_r(s) := \begin{cases} \varnothing & \text{if } C \text{ false in s} \\ \{s' \mid (s, s') \text{ satisfies } r \Leftrightarrow C \text{ true in } s \text{ and } E \text{ true in } s'\} & \text{otherwise} \end{cases}$$

Moreover, the sketch rules do not specify how to achieve E, but provide a subgoal ordering. This makes them expressive yet flexible, ideal for planning in domains with interleaved or non-serializable goals. Also, it is important to define what a **sketch width** is, as this differs from the width of a planning problem [10].

Definition 2 (Sketch Width [10])**.** Let $P \in Q$, where Q be a class of problems, where R_Φ be a well-formed sketch for Q. Then, sketch width of R_Φ at a reachable state s in P, $w_R(P[s])$, is the width of the subproblem $P[s]$. $P[s]$ is like P but with the initial state set to s and the goal set to $G_r(s)$ for any rule $r \in R_\Phi$. With this sketch width can be defined as: $w_R(Q) = \max_{w_R(P[s])}\{P[s] | \forall P \in Q : \text{reachable states } s \text{ in } P\}$.

SIWR integrates sketch into IW-based planning [10, 13]. This algorithm is the one we will be using, but instead of classical planning domains, it will have ALE's games as domains. Hence SIWR operates as follows:

1. Define a policy sketch R_Φ for the problem class Q.
2. Starts at initial state s_0.

3. Use IW to find a state s satisfying either the problem goal or being in the subgoal states $G_r(s_0)$ for $r \in R_\Phi$.
4. If s is not a goal state for the problem, set s as s_0 and repeat.

It retains the polynomial-time guarantees of IW when the sketch width is low, while achieving vastly better performance on domains where plain SIW fails [10, 13]. Additionally, sketch in comparison to SIW are complete as they provide a ”roadmap” for the solution of the problem [7, 13]. In a more formal sense [10]:

Theorem 1 (SIWR Complexity)**.** *Let R_Φ be a policy sketch for problem class Q with sketch width $w_R(Q) \leq k \in \mathbb{N}$. Then, $\forall P \in Q : SIWR(P) \in O\big(b \cdot N^{|\Phi|+2k-1}\big)$ time $\wedge$ $SIWR(P) \in O\big(b \cdot N^k + N^{|\Phi|+k}\big)$ space. Here, b is the branching factor, N is the number of atoms in P, and $|\Phi|$ is the number of features in the sketch.*

3 Related Work

Our work sits at the intersection of planning-based methods and structured decomposition for complex environments. Below, we review four lines of research that are most relevant for either our domain or our approach:

Deep Reinforcement Learning from Pixels

Convolutional networks were shown to be able to learn competitive policies directly from raw Atari pixels using only the game score as a reward signal in the groundbreaking DQN [22]. Although there are many extensions of this end-to-end, reward-driven paradigm (such as A3C, Rainbow, and PPO), they are all fundamentally dependent on a meticulously designed reward function. Although these strategies work well in games (with non-sparse rewards), applying them to real-world situations necessitates intensive reward function engineering, which is frequently unfeasible. By substituting a subgoal-fulfilling reward function for the reward signal, our approach circumvents this reliance. These subgoals are solutions to the subproblems, which are broken down from the primary tasks using sketches.

Width-Based Planning for Atari

In ALE, width-based search algorithms like IW and its variations have demonstrated exceptional sample efficiency [5]. By eliminating non-novel states, these techniques allow for efficient planning with a lot fewer interactions with the environment than deep reinforcement learning. To enhance the feature set utilized by the width-based search, recent work such as Olive ([5]) further incorporates online representation learning. They operate at the low-level state features, which lessens the dependency on reward signals. Hence, these techniques' core idea is discovering new states through novelty in the search space of the problem. On the other hand, our method uses SIWR to tackle each subproblem specified by the sketch rules, building on the same width-based planning assumptions. Thus, we still share the same core strengths and weaknesses of width-based planners but try to alleviate some of their limitations in sparse-reward domains through sketch guidance.

Sketch-Based Planning for Task and Motion

Task and Motion Planning (TAMP) problems are where robot manipulates objects to achieve a goal [11]. Often it requires sequencing symbolic actions while satisfying geometric constraints in continuous space. SIWR method solved TAMP problems, where symbolic and geometric reasoning are interwoven through hand-crafted designs [11]. A key limitation of existing

TAMP approaches is global backtracking. When a plan fails due to geometric infeasibility, the search must often restart from scratch, discarding previous computation and causing combinatorial explosion. To overcome this, the authors, using sketch, break TAMP problems into subproblems, each solvable greedily in linear time under a width-1 assumption. By isolating geometric failures to specific subproblems, sketch localizes backtracking and enables adaptive, per-subproblem sampling rather than global pre-sampling. Consequently, their algorithm solved all benchmark tasks, outperformed existing planners, and reduced motion-planning calls by roughly half through action validation. Similarly, we use the same sketch structure and SIWR structure in the ALE domain, where sketch decomposes games into smaller stages that can be solved greedily. By decomposing the game into smaller subproblems, we can effectively guide the width-based search to focus on relevant parts of the state space.

Object-Centric Abstraction for Reinforcement Learning

To enhance generalization, object-centric approaches have been developed for ALE. They also decrease susceptibility to misleading visual correlations. For example, OCCAM [9] forces the agent to concentrate on task-relevant entities by filtering out irrelevant pixels using an object-centric attention mask. Without complete symbolic extraction, these methods introduce structural biases. Even if they share our objective of shifting away from raw-pixel input, they stay within the reward-driven RL paradigm. In contrast, our sketch-based method offers an alternative route to structured, sample-efficient problem solution. Namely, through completely substituting the reward signal with a symbolic decomposition.

Positioning of Our Work

The reviewed literature demonstrates a variety of methods, ranging from fully reward-driven deep reinforcement learning to planning-based strategies that reduce reward dependence. Sketch-based decomposition effectively improves breadth-first search in robotic domains [11]. Additionally, width-based planners [5] perform exceptionally well considering sample efficiency and performance. Despite these advantages, they operate at the level of low-level features, which makes them vulnerable to sparse rewards and high subgoal widths. Our work connects these ideas by utilizing **sketch** to deconstruct ALE's games into smaller problems and resolving the subproblems using a **SIW**. Consequently, the need for a specific reward function is eliminated, which is a significant limitation of traditional RL approaches. Nonetheless, it maintains the sample efficiency of width-based planning. Thus, our method combines the effective search of width-based techniques with sketch specifically designed for the Atari environment. These sketch rules provide the width-based search a structured, human-like issue breakdown of the problem.

4 Approach

Our methodology integrates sketch with width-based planning to decompose high-width Atari tasks into solvable subproblems through some adaptations. Thus, our approach builds on the framework of width-based planning methods, specifically leveraging the SIWR algorithm [10] for the game of Adventure. SIWR is designed to efficiently solve planning problems in environments with sparse rewards and has seen success in classical planning domains and TAMP [10, 11, 18]. In other words, we shall employ SIW algorithm [7, 18] and extended it with sketch [10, 13]. We will be using the compact action space for the game to simplify the action representation and reduce the search space. Since our algorithm comes from a classical planning setting, it is also vital to remark that we will follow the simplification for the action-cost function. Thus, each action will have the same cost, a plan's cost will be its length, and an optimal plan is one with the shortest action sequence.

4.1 Problem Definition

Following other papers in the domain of the ALE ([16, 20, 21]), the Adventure game, using the ALE framework [8], can be formalized as a time-invariant deterministic POMDP = $(\mathcal{S}, \mathcal{A}, T, R, \gamma, \Omega, O)$. In Adventure with our setting, at any point the agent only observes pixel image rather than the full underlying state, as we operate on the screen pixels. Simply put, the agent only knows what is happening in the current room rather than in the entire game, which makes this environment partially observable. Moreover, the reward function, the transition or observation function are time independent, as the game dynamics and reward structure do not change over time. Therefore, the POMDP is time-invariant.

An important property of ALE is being deterministic; given a sate s and an action a, the next resulting state s' is uniquely determined [16, 20]. Therefore, we will be working with a transition function T. Similar to other papers in this domain [4, 7, 12, 21], we assume that the agent has access to the ALE simulator, which is used to generate next states. This simulator is deterministic, and the agent has a fixed budget of simulator calls per iteration to call the simulator to plan its next action. It can be used to simulate the effect of taking an action in any state without actually executing it in the real game environment. Another assumption is that the observed states can be cached and can be used to reset the simulator to any previously observed state [4].

In the context of the Adventure game, $\mathcal{S}$ can be defined as the RAM states of the ALE's emulator being the finite set of states the environment can be in [1]. In the ALE, the theoretical number

of possible RAM states is 256^{128} since the Atari 2600 has 128 bytes of RAM, and each byte can take on 256 different values [16, 20]. On the contrast, in theory $\Omega = 256^{210\times160}$ as these are 210×160 grayscale pixel images, with 256 possible pixel values [1]. Although every pixel on the screen is denoted by a single byte value between 0 and 255, the Atari 2600 system employs a color palette containing 128 unique colors, which confines the actual color representation to these 128 options. In theory, the screen's state space is astronomically larger than the RAM's because each pixel can be any color. In reality, the screen is a deterministic function of the RAM, so the set of actually possible screens is a tiny subset constrained by and smaller than the set of possible RAM states. It is actually many to one mapping from RAM states to screen images, as multiple RAM states can produce the same screen image [16, 20]. Considering, we work with the compact action space, the action set $\mathcal{A} = \{0, 1, 2, 3, 4, 5\}$ consists of 6 discrete actions available to the agent at every decision point. These are No-op (0), drop the current item equipped (1), up (2), right (3), left (4), down (5). We chose $\gamma = 1$ to promote long-term planning toward the objective because Adventure has few rewards. Practically speaking, this will not have much of an impact because rewards are only given at the conclusion of the game.

Regarding the transition function T, this is unknown to the agent in general, as it cannot observe the RAM state directly or through the simulator. Even though, the agent could do this, in our work we solely rely on pixel observations for planning. Thus, $T(s, a) = s'$ if action a in state s leads to a new unique state s'. This resulting state is unique and will always be the same when executing the same action in the same state. On the other hand, the observation function $O(s', a)$ maps the RAM state s' to a pixel image $o \in \Omega$, which is the visual representation of a part of the game at new state s' after taking action a from current state. This is because that the agent only sees the screen pixels of its current room rather than the full game at once. Furthermore, we observe here the partially observable nature of the environment, as the pixel image may not fully capture all aspects of the underlying RAM state. To illustrate this, let us consider the RAM state s_1 of Adventure game, where the agent has killed one of the dragons and is in another room. The corresponding RAM state s_1 reflects the fact that a dragon is killed, but the pixel observation $o_1 = O(s_1)$ may not visually indicate this, as the dragon is not present in the current room's pixel image. The agent can only obtain an observation through the ALE simulator, but does not know the function itself.

Similarly, the agent does not know the reward function R beforehand, but the current state reward can be obtained from the ALE simulator after executing an action in a state. As discussed in Chapter 2, the rewards in Adventure are sparse and only given when the game is completed. Therefore, $\mathcal{R}(s, a)$ is only bigger than 0, when the resulting state $s' = T(s, a)$ is the game completed state. In ALE, the agent interacts with the environment in episodes, where each episode starts from the initial state and ends when the game is completed, the agent dies or a maximum number of steps is reached. Thus, the POMDP is characterized by the maximum total rewards an agent can obtain in an episode [20, 21]. Adventure's maximum

total reward is the reward for completing the game, as it is the only non-zero reward in the game.

4.2 Width-Based Planning with Policy Sketch

We will extend the SIWR algorithm [7, 18] to ALE's domains by implementing sketch rules. Therefore, we develop domain specific sketch for IW to be able to use **SIWR** in these domains [8, 10, 18]:

1. Define a sketch R_Φ for the problem class Q.
2. **Initialization**: At the root node (current state s_0), identify all applicable sketch rules $C_i \rightarrow E_i$ where C_i holds. Let $R_i \subseteq R_\Phi$ be sketch rules whose conditions hold at this node.
3. **SIWR**: Perform IW from s, the current root node, pruning states with novelty $> k$ (typically $k \leq 2$). During expansion:
   - **Node evaluation**: For each generated state s, check if s satisfies any E_j (the effect of a sketch rule in R_i). In other words, check if s is in the subgoal states $G_r(s_0)$ for $r \in R_i$.
   - **Termination condition**: If s achieves E_j, return the action sequence to reach s as a *subgoal solution*. If IW fails to reach any subgoal state within the budget, the agent executes 1-5 random actions (uniformly sampled from valid actions) to induce state perturbation before restarting the sketch-guided search.
4. **Subgoal Chaining**: If s is non-terminal:
   - Set s as the new root node and $R_{i+1} \subseteq R_\Phi$ as the applicable sketch rules at this node.
   - Repeat Step 3.
5. **Termination**: If no applicable sketch rules remain or the goal state is reached

Furthermore, the sketch is designed to ensure a **width-bounded admissible chain** $t_0, t_1, \ldots, t_n$ (Section 2.5)[18]. Thus, the search is guided by the sketch rules, focusing on achieving sketch rules earlier rather than exploring all possible states.

4.2.1 Adaptions of SIWR for ALE

Additionally, there are certain adaptations to SIWR to ensure pixel-based planning for which we oriented our approach on the Rollout-IW [7]:

- **Frame Skipping:** Decisions are made every N frames (actions repeated for N-1 frames) to reduce computation. Selected actions persist for N frames, reducing decision frequency while maintaining kinematic continuity.
- **Caching Trajectories:** For the next IW search, we will preserve node's information from previous searches like their rewards, screen pixels, state and sketch feature values. Reusing them for the new search reduces simulator calls and computation time to

recompute the values of already seen states. However, the novelty table is reset for each new search to ensure correct novelty computations.

Firstly, SIWR delivers significant efficiency gains by decomposing complex tasks into low width subproblems. Each subgoal, constrained by the sketch rules to a width $w \leq 1$, can be solved in $O(|F|)$ time-a polynomial improvement over $O(|S|)$ for brute-force search, where $|F|$ is the features count and $|S|$ is the state-space size [18]. Also, theoretically the theorem (Definition 2) guarantees that SIWR solves any problem P in the domain Q in $O\big(b \cdot N^{|\Phi|+2k-1}\big)$ time for k-width sketches if the sketch R_Φ is well-formed. Furthermore, b is the branching factor, and N is the number of atoms in P. This ensures scalable performance for bounded-width sketches. Moreover, sketch generally generalize across problem instances (e.g., all levels of Adventure) eliminating the need for instance-specific heuristics [10, 13]. However, this only holds if the sketch is well-formed and the subproblems bounded sketch-width for the problem class Q (Definition 2). By integrating sketch with a sample efficient width-based algorithm SIW, we enable sample-efficient exploration.

4.2.2 Features for IW Searches

To enable width-based planning in pixel-based domains like ALE, we utilize screen features extracted from pixel observations. These features serve as the basis for novelty computations in IW searches and are crucial for effective planning. We employ the **BPROST** feature set (Chapter 2), which consists of binary features derived from pixel observations [7].

BPROST divides the ALE's screen (210×160 pixels) into a grid of 16×14 tiles. Thus, the screen is divided into 224 tiles, where each tile is 10×15 pixels big. Our work is in grayscale mode, with 128 colors, resulting in $224 \times 128 = 28672$ basic features, as there is a basic feature for each color-tile combination. In our case, the offsets i and j range from -13 to 13 and -15 to 15 respectively, due to how the screen is divided into tiles, and the pixel colors $k_1, k_2 \in [0, \dots, 127]$. So, BPROT entails $128 \times 128 \times 27 \times 31 = 13,713,408$ features [7]. In order to capture the spatial relationships across the entire screen, BPROS first considers the spatial relationships between two pixels with different colors. Therefore, BPROS encompasses $128 \times 127 \times 27 \times 31 = 6,803,136$ features to capture spatial relationships between two pixels with different colors. To encapture the relationships between pixels of the same color and non-zero offsets, BPROS has additional $\frac{128 \times (27 \times 31 - 1)}{2} = 53,504$ features. Finally, for the spatial relationships between pixels of the same color and zero offsets, BPROS includes another 128 features. Thus, BPROS in total has $6,803,136 + 53,504 + 128 = 13,713,408$ features. Together, BPROST offers over 26 million sparse binary features per frame.

Additionally, we developed a new custom feature set tailored for the Adventure game, on top of the BPROST features, including 2 handcrafted features specifically designed to capture critical game elements and states relevant to Adventure. This new feature set is called **Adventure-**

Features and consists of a total of 314 new binary features alongside BPROST features. These are as follows:

- **Room Detection Features**: Identifies the current room the player is in based on the dominant colors present in the background. In total, Adventure has 14 rooms; thus, these features introduce 14 binary features.
- **Item Distance Features**: Binary features indicating the agent's distance to key items (e.g., sword, keys, chalice), which are crucial for progressing through the game. For this, we first calculate **cube position** on the screen by searching for a 4×8 block of matching colors (the cube has a size of 4×8). Then, we try to locate the key item through the screen search for pixel clusters that match the item's color and size. Finally, we compute the Manhattan distance between the cube and the item, discretize this distance into 75 bins of 5 width each, and set binary features indicating whether the distance falls within each bin. If the item is not visible, we discretize the distance to the 0th bin, meaning the item is not visible. So, we gain additional $75 \times 4 = 300$ binary features.

Thus, the Adventure-Features are just a more refined version of BPROST for the Adventure game and share the same properties as BPROST features.

4.3 Sketch and Problem Decomposition for Adventure

To address the sparse reward challenge in Adventure, we designed a sketch R consisting of 24 sketch rules involving 4 features, which decomposes the game into a sequence of width-1 subproblems. Despite the sparse rewards, we can see the subgoals defined by the sketch rules as intermediate rewards. Thus, we can define a modified reward function $R'(s, a)$ that incorporates subgoal achievements as intermediate rewards. In this function $\mathcal{R}'(s, a) = 1$ if the resulting state s' achieves at least one subgoal, and 0 otherwise. It is important to note that this modified reward function only accounts for subgoals whose preconditions are satisfied at s (In the IW(1) searches, this will be the root state). As a result, the number of sketch rules whose preconditions are met during an episode is equal to the maximum total reward under R'.

4.3.1 Sketch Features for Adventure

The sketch R relies on handcrafted features that translate pixel observations into symbolic predicates. We will be using a total of 4 features, which we will denote as F_S, for the sketch in Adventure, which are defined below. These features detect game objects and states using color matching, spatial clustering, and state tracking. They include:

1. **Cube Detection ($at(x, y)$)**: Here, one determines the cube's (agent's) location on the screen by searching for a cube-sized cluster of pixels, which match the room color in the possible regions of the screen. This is possible, as the cube has the same color as the

room it is in. The x, y coordinates represent the top-left corner of the cube on the screen. This is sufficient as the cube has a fixed size of 4×8 pixels, thus by knowing the top-left corner, we can determine the full cube position on the screen.

2. **Item Possession (***have*(*item*)**)**: Detect when the agent carries critical items using state variables and pixel analysis. For this, it first needs to detect the cube's position and then, it checks if the cube is touching an item or carrying one. In order to do this, one checks if there is a cluster of the same-colored pixels, which are not grey, and matches an item's size and color. If they are adjacent to the cube's pixels, then the agent is touching them. In other cases, when they are in cube's vicinity but not adjacent, then the agent is carrying them. State variables are used to keep track of whether the agent is carrying an item or not, as this information is not always directly observable from the pixel observation. For instance, Adventure being an old game, sometimes the item might vanish behind a wall, making it impossible to detect it through pixel analysis alone. Thus, state variables help us in these instances to say whether the agent is carrying an item or not. This is possible, as these variables are updated based on the pixel observations and inherited through the search tree nodes. Therefore, if we do not see the cube drop an item, we can assume it is still carrying it. Other than that, we use state variables to identify clusters which might contain cube and an item, because they are adjacent and have same color. This is further explained in the implementation chapter (Chapter 5).
3. **Location Features (***in*(*item_room*)**)**: Determine in which room the cube is in via dominant color pattern and item presence. Thus, it confirms which room it is in by checking for a pixel-cluster, whose size and color match the item, in the room. Also, there is a detection function not only for each of the items (sword, yellow key, black key and chalice), but also for the two dragons. Additionally, if none of the items or dragons are detected, then the room is identified based on its dominant background color. For instance, in the maze rooms, where no items or dragons are present, the room is identified by its unique blue background pattern.
4. **Combat Features** *killed*(*dragon*): Track dragon states using cluster size and color analysis. For this, we utilize the above methods to detect each item on the screen as well as detecting the cube's location. Then by collecting non-grey pixels in the room, we form clusters based on same color and adjacency. These clusters are then filtered out if their location, size and color correspond to items or the cube. So, all the remaining clusters are either part of the room-borders or dragons, because everything else has been filtered out. Finally checking the size of the clusters and doing a pattern matching, it can determine if a dragon is killed or not. We can do this via pattern matching, as when a dragon is alive, it has a specific pattern of pixels, and when it is killed, its pixel pattern changes. Even though, both dragons have the same pattern when alive or killed, they can be distinguished via their color.

Besides these sketch features, we also utilize some variables to keep indirect track of certain states in the game, like for instance if the yellow dragon is killed or not. We need this as we are utilizing a classical planning algorithm in a partially observable environment Adventure. This is important, as at any given point in the game, the agent can only see the current room it is in, and not the other rooms. Thus, relying on pixel observations would not be sufficient to track such states, as the dragon might not be visible at the current room. Therefore, we use boolean variables to keep track of such states throughout the game. These variables represent our knowledge about the game state that is not directly observable from the current pixel observation. We integrate them at node level in the SIWR search tree, inheriting them from parent nodes and updating them based on their screen's sketch evaluation. In general, we believe that one cannot create a well-formed sketch for Adventure without such variables, as the environment is partially observable. For instance, to know if one still needs the sword to kill the yellow dragon, one needs to know if it is already killed or not. However, if the agent already is another room after killing the yellow dragon, it cannot see it anymore to determine if it is killed or not.

4.3.2 Sketch Implementation for Adventure

Using the features defined above, the goal of the sketch R is to guide the agent through the sequence of actions needed to complete a huge portion of the game, which is to acquire the chalice. This final state can be expressed as `have(chalice)`, which would be the final sketch rule's effect. Thus, the R is not designed to complete the entire game, but rather to acquire the chalice, which is a significant portion of the game. This sketch R consists of 24 sketch rules described below under the notation of the work that first introduced sketch [10]. Important to note is that any non-mentioned feature must maintain its value, whereas any feature with a question mark (`have(sword)?`) may change arbitrarily. Our sketch on a symbolic level can be seen in the table (Chapter A). One can observe that the sketch rules are grouped into three main categories:

- **Item Acquisition Sketch Rules**: These sketch rules (1,3,9,24) focus on acquiring essential items like the yellow key, sword, black key, and chalice.
- **Combat Sketch Rules**: These sketch rules guide the agent through the process of killing the yellow and green dragons to ensure future safe navigation. These include the sketch rules 5 and 7.
- **Navigation Sketch Rules**: These sketch rules help the agent navigate through the complex layout of Adventure to reach key locations. This category can be subdivided into: (1) Navigating to items or dragon rooms (2,4,6,8,10,23) and (2) Traversing the blue labyrinth (11-22).

Together, these sketch rules offer the agent a well-organized strategy that divides the difficult job of obtaining the chalice into feasible subgoals. Since we still need to return to the golden castle

after acquiring the chalice to finish the game, this merely symbolizes a significant percentage of the entire game. In accordance with the three categories, the sketch rules follow logical progression and guarantee that the agent obtains necessary items, gets rid of hazards, and successfully navigates the surroundings. The journey is divided into several successive sketch rules due to the intricacy of the blue labyrinth. Each sketch rule concept creates a logical plan for finishing the game by building on the framework established by previous sketch rules. Due to its complexity, the blue labyrinth's navigation is broken into multiple sequential sketch rules. The final goal is to access the chalice room, which requires that all earlier conditions be fulfilled.

4.3.3 Formal Properties of the Sketch for Adventure

SIWR's effectiveness to work on the Adventure game depends on the problem decomposition provided by the sketch R built using features F_S, and the subproblem search through IW(1) using BPROST features. Therefore, to guarantee that SIWR can solve any instance of Adventure in polynomial time (Theorem 1), we need to ensure two properties: we need to verify that the sketch R is well-formed and has sketch-width 1 (Definition 2). Formally, we need to show that R is well-formed sketch for the problem class Q of all Adventure instances and that the subproblems induced by R have width 1 when using BPROST features for the IW(1) searches. For this, we shall prove R's well-formedness and provide an explanation why the sketch width of R is 1. With Q we mean here the class of all Adventure instances, which differentiate in their original seed for the ALE environment.

Well-formedness of R

To ensure R is well-formed for Q, we need to show that F_S distinguishes goal states in all Adventure instances and R is terminating.
Termination:

Proof. We need to show that the sketch rules in R can only be utilized a finite number of times to create subgoal states, ensuring a finite subgoal chaining. Hence, this would guarantee that from a state s, the algorithm can only move to $s' \in G_R(s)$ finitely. For this, we first observe that at the start of the game, none of the features in F_S are satisfied except for the feature $in(yellow\_key\_room)$. In the following, we will show that each sketch rule of R can only be utilized once in a sequential manner, leading to a finite sequence of subgoals.

Firstly, notice that each sketch rule in R flips at least one feature of its precondition from true to false or from false to true. For instance, the first rule turns $have(yellow\_key)$ to true from false. Therefore, a sketch rule cannot immediately follow itself. Now to show that each sketch rule cannot be utilized multiple times after another rule or multiple other rules. For this, we

will illustrate that the required feature combinations for each rule condition are only satisfied once in a sequential manner.

We notice that the first sketch rule is only applicable once at the start of the game, as it requires being in the yellow key room, not having the yellow key or being in the sword room, which all are true at the start of the game. After applying this sketch rule, any sequence of other sketch rules cannot make this condition true again, making the first sketch rule can never be applicable again. Continuing this reasoning, we can see that the second sketch rule in R also is the only one that can follow the first sketch rule, as all the other rules require having the sword or being in other rooms. When the second sketch rule is used, the feature `in(sword_room)` becomes true. Due to the fact, that after the second sketch rule, `in(yellow_key_room)` is false, and after the third sketch rule `have(yellow_key)` is false and no other sketch rule or sequences sets either of the features back to true, the second sketch rule can never be applicable again. Similarly, the only sketch rule that can follow the second sketch rule is the third one, because other sketch rules require having killed dragons, having other items, or being in other rooms. After using the third sketch rule, the feature `have(sword)` becomes true and `have(yellow_key)` becomes false, and just like before, no other sketch rule makes the feature `have(yellow_key)` true again, making the third sketch rule never applicable again. So far, the features that have become true are `have(sword)` and `in(sword_room)`. Furthermore, the only available sketch rules are 4-24, as all other sketch rules have been disabled by the previous reasoning.

The only sketch rule that can follow the third sketch rule is the fourth one, as all others sketch rules require having killed dragons or being in other rooms. After applying the fourth sketch rule, the agent moves to `in(yellow_dragon_room)`. It is the only successor, because as all other available sketch rules require having killed either one of the dragons After applying the fifth rule, `killed(yellow_dragon)` becomes true, permanently disabling the fifth and fourth rules, because no other sketch sets this back to false. Following the same reasoning, every rule's condition are false, requiring having killed the green dragon or being another room, except the sixth rule's condition, whose effect moves the agent to `in(green_dragon_room)`. The only logical next sketch rule is the seventh one, because all others available sketch rules require having killed the green dragon, having the black key or being in other rooms. Applying it sets `killed(green_dragon)` to true, making the seventh and sixth rules inapplicable forever, because no sketch rules or sequences will lead to `killed(yellow_dragon)` being false. This far, the features that have become true are `have(sword)`, `killed(yellow_dragon)` and `killed(green_dragon)`. Also, the sketch rules that can follow are 8-24 as all others have been excluded by the previous reasoning.

Furthermore, only the eighth sketch rule can follow the seventh one, as all other enabled sketch rules require having the black key or being in other rooms. If the eighth sketch rule is applied, the agent moves to `in(black_key_room)`, turning the feature `in(black_key_room)`

to true and *in*(*green_dragon_room*) to false, the only sketch rule that can follow the eighth one is the ninth one, because all other sketch rules require having the black key or being in other rooms. After the eight sketch rule has been made use of, it is disabled because `in(green_dragon_room)` will not be true again. Additionally, ninth rule applies in the black key room, setting `have(black_key)` to true. The rules eighth and nine become inapplicable thereafter because its preconditions `have(sword)` will remain false.

Moving forward, the maze navigation sketch rules (rules 11-21) guide the agent through maze. Each rule moves the agent to a new location (e.g., `in(blue_maze_start)`, `in(blue_maze_mid)`, etc.). They have a general structure where the precondition requires being in a specific room and location, and the effect moves the agent to a new location either in that room or new room. So, they have a form of `in(room_X)` and `at(`A_1`)` → `in(room_Y)` and `at(`A_1`)` and ¬ `at(`A_1`)`, where room X and Y can be the same room. Here, A_1 and A_2 are specific areas in the respective rooms rather than specific coordinates, ensuring that agent still has some freedom of movement for each sketch rule. This is possible, as we know with `at(x,y)` feature where the cube is located on the screen. Thus, we can define areas on the screen and compare cube's location to these areas bounding boxes. Since room locations are never revisited in the sketch chain, each rule is used at most once. After completing the maze, the true features are `have(black_key)`, `in(blue_maze_end)`,
`killed(green_dragon)`, `killed(yellow_dragon)`. In addition, only rules that can now follow are 22-24, as all other rules have been disabled by the previous reasoning.

Moreover, only the 22 sketch rule can follow the 21 sketch rule, as all other sketch rules require being in other rooms. When the 22 sketch rule is applied, the agent moves to `in(black_room)`, turning the feature `in(black_room)` to true and `in(blue_maze_end)` to false. This rule also becomes inapplicable thereafter because its preconditions `in(blue_maze_end)` will remain false. Likewise, only the 23 sketch rule can follow the 22 sketch rule, as all other sketch rules require being in other rooms. After applying it, the agent moves to `in(chalice_room)`, turning the feature `in(chalice_room)` to true and `in(black_room)` to false. This rule also becomes inapplicable thereafter because its preconditions `in(black_room)` will remain false. Finally, only the 24 sketch rule can follow the 23 sketch rule, as all other sketch rules are unfit to be used anymore. After applying it, `have(chalice)` becomes true, and `have(black_key)` becomes false. Subsequently, no sketch rule resets `have(chalice)` to false, ensuring that the 24 sketch rule is used at most once. To summarize, each sketch rule in R can be applied at most once in a sequence, ensuring finite subgoal chaining. □

Distinguishing Goal States:

Proof. The goal in Adventure is to reach the yellow_throne_room with the chalice. Feature *have*(*chalice*) directly indicates chalice possession. Additionally, *in*(*yellow_key_room*) confirms the agent's location in the final room. This is due to fact, that earlier in game we dropped

the yellow key when picking up the sword in the final room. Thus, the features in F_S effectively distinguish goal states across all Adventure instances. □

Even more, if one were to extend R to complete the entire game, one would need to add more sketch rules to return to the golden castle after acquiring the chalice. Still in these cases, the new sketch rules would not lead to any of the previous sketch rules being applicable again, ensuring termination. For instance, for sketch rule 22, one might return to the black room after acquiring the chalice, but this would not make sketch rule 22 applicable again, as one does not possess the black key. Similarly, for sketch rule 23, one needs the black key to apply it again, but after acquiring the chalice, the agent does not hold the black key anymore, meaning its condition will stay false.

With this we have shown that the sketch R is well-formed for the problem class Q of all Adventure instances.

Sketch Width of R

In the following, we will give an intuitive explanation why the sketch width of R should be 1 when using BPROST features for the IW(1) searches under the following assumptions. Thus, we demonstrate that each subproblem defined by the sketch rules could be solved with width 1 using the BPROST features in Adventure. For this, we first state the assumptions needed for the width-1 explanation, which are:

Assumption 1. Feature Sufficiency: The BPROST feature set (Basic, BPROS, and BPROT features) is strong enough to distinguish goal states from non-goal states, meaning it classifies the goal states during the IW(1) search as novel. This ensures that during the IW search, the goal states are not pruned, and thus can be reached. The sketch features F_S are only used to define subgoals and not for the novelty computations in the IW(1) searches.

Assumption 2. Visual Feature Stability: BPROST features reliably detect game objects and their relationships across frames, with temporal features (BPROT) correctly capturing object motion. Here, we assume that the visual features are stable enough to allow the IW(1) search to make progress toward subgoals without being misled by transient visual changes. We need this assumption, so that the features used for novelty computations remain consistent during the search process, allowing the SIWR to find an optimal path to the subgoal.

Assumption 3. Navigability: Every reachable game position can be accessed through a sequence of basic movement actions (up, down, left, right), and items can be picked up when the agent is adjacent to them. This guarantees that there are no unreachable areas in the game that would prevent the agent from achieving subgoals defined by the sketch. Also moving between 2 locations in the same room is width 1.

Assumption 4. Sketch Precondition Accuracy: The handcrafted sketch features (cube detection, item possession, location and combat features) accurately reflect the true game state.

These assumptions, while strong, are needed to explain the power and scope of the proposed method. Importantly, the pixel-based planning with BPROST features does not require declarative action models, only the ability to simulate state transitions and extract visual features. The sketch for Adventure consists of 24 rules, which can be grouped into three categories for width analysis. For each category, we show that the subproblem of reaching a state $s' \in G_r(s)$ (where r is an applicable sketch rule) should be width 1. For the following, $m \in \mathbf{N}_{>\mathbf{0}}$:

Item Acquisition: Sketch rules 1, 3, 9, 24 involve acquiring items (yellow key, sword, black key, chalice). When these rules are active, the subproblem involves navigating to the yellow key (black key, sword) and picking it up. An optimal plan for this subproblem would be: (1) a sequence of movement actions that bring the agent adjacent to the key, chalice, or sword and (2) a pick-up action when adjacent (In adventure, the agent automatically picks up the item when touching it and therefore also drops any currently held item). The BPROST features made true in an optimal plan form an admissible chain would be:

- t_0: BPROST features indicating the agent's starting position (true in s).
- $t_1, \ldots, t_{k-1}$: Sequence of BPROST features indicating progressive movement toward the key (e.g., "agent tile at (x, y)" features becoming true one at a time).
- t_k: BPROST feature indicating adjacent to the key (agent and key tiles adjacent).
- t_{k+1}: BPROST feature indicating key possession (key tile moves with agent tile).

Each t_i is a conjunction of at most one new BPROST feature (e.g., agent appearing in a new tile), and optimal plans for t_i can be extended to achieve t_{i+1} by adding one action (movement or pick-up). Thus, the subproblem should have width 1 and could be solved by IW(1).

Room Navigation: Sketch rules 2, 4, 6, 8, 10-23 involve navigating to specific rooms (sword room, dragon room, black key room, maze rooms, chalice room). Under these rules, the subproblem involves moving from the current room to the target room and under the navigability assumption, there exists a path of movement actions leading to the target room. The BPROST features made true in an optimal plan form an admissible chain would be:

- t_0: BPROST features for current room (true in s) as each room has distinct color patterns.
- $t_1, \ldots, t_{m-1}$: BPROST features for intermediate positions in corridors/doors.
- t_m: BPROST features indicating entry into the particular room (distinct color pattern of item, enemy or maze in the room).

At most one new BPROST feature (e.g., agent entering a new tile with distinct room color) for each $t_i \forall i \in [m]$. The chain is admissible because moving to the next tile requires only one action, and BPROS, the spatial features, capture the relative positions. Hence, the subproblem should have width 1.

Maze Navigation (Sketch Rules 10-22): The blue maze navigation uses precise positional constraints (e.g., "agent at coordinates $(135 - 160, 146 - 178)$ in room 9"). These constraints

are still single conditions that can be achieved by a sequence of movement actions. Each movement makes a new BPROST feature true, maintaining width 1.

Dragon Combat: In Sketch Rules 5 and 7, the green and yellow dragons must be killed. The subproblem is to place the agent (with sword) next to the dragon and execute it while these rules are in effect. Moving next to the dragon and then executing the attack action—which in Adventure happens automatically when the sword hits the dragon—is the best course of action. An admissible chain of actions formed by the BPROST features would be:

- t_0: BPROST features for current position in dragon room.
- $t_1, \ldots, t_{p-1}$: BPROST features for movement toward dragon.
- t_p: BPROST features indicating adjacency to dragon.
- t_{p+1}: BPROST features indicating dragon disappearance (killed).

Like the first two categories, each t_i necessitates the achievement of a maximum of one new BPROST feature (e.g., agent in a new tile closer to the dragon). When nearby, the assault action modifies the dragon's pattern, which is reflected in a change in BPROST characteristics. This should be width 1 according to the definition of width (Definition 1).

Therefore, each subproblem could be width 1 under the assumptions mentioned above. Thus, Definition 2 gives us the sketch width $w_R(Q) = 1$ for the class of Adventure problems Q. With sketch width being $k = 1$, Theorem 1 gives the following complexity bounds for SIWR on Adventure problems: $O\big(b \cdot N^{|\Phi|+1}\big)$ time and $O\big(b \cdot N + N^{|\Phi|+1}\big)$ space. Concerning the variables, b is the branching factor (number of available actions, typically 6 in compact action mode). N is the number of BPROST atoms (features) $\approx 2.87 \times 10^7$, and Φ is the set of sketch features used in R ($21 = |F_S|$). In our case, the sketch features are `have()` for each of the 4 items (yellow key, black key, sword, chalice), `in()` for each of the 13 possible rooms, and `killed()` for each of the 2 dragons. Also, the `at()`, which indicates the cube's position, can be seen as a numerical feature derived from the room features. Regarding the state variables, we do not include them in Φ because they are calculated using the sketch features. Even though, they provide information about the ancestor states, we could achieve the same by using the sketch features on those ancestor states directly. Thus, they serve more as a convenience for implementation rather than being essential for sketch width analysis. Although N is large, the exponential term $N^{|\Phi|+1}$ becomes N^{22}, which is polynomial in the number of features. More importantly, the width-1 property ensures that IW(1) can solve each subproblem in linear time with respect to the number of features explored, making the overall planning tractable.

4.4 Practical Considerations and Limitations

In practice, the assumptions above may not be held perfectly. In our experiments, we observed that the Assumption 1 of feature sufficiency does not are always held, because BPROST features

may not capture all changes happening on the screen. Evident in the maze section, which has only a single path which leads to the black castle and back. While following this path, IW(1) cannot expand certain nodes necessary to follow the path due to exactly how IW(1) prunes states, which is further discussed in Chapter 5 and Chapter 6.

Similarly, Assumption 2 of visual feature stability may not always hold due to numerous factors like item flickering. This becomes also evident in our empirical results, where we see that for some sketch rules IW(1) finds a chain of actions that lead to the goal state, but is not an admissible one, as there exists another shorter, more optimal sequence of actions leading to the same goal state. We will dive deeper into this in the evaluation section Chapter 6.

Other limitations are related to Assumption 3 and Assumption 4, which may not always hold due to the frame skipping mechanism in ALE and potential inaccuracies in sketch feature detection. However, these particular limitations were rarely seen in our empirical evaluation, as the agent was able to navigate to all required locations and pick up items successfully. Nonetheless, we could not guarantee that these assumptions always hold in every situation, because we were not able to test every sketch rule or all possible game states. Despite these practical limitations, theoretical analysis provides a strong foundation for understanding why the sketch-based approach is effective in Adventure. Our empirical results still validate that these theoretical properties are evident in the practical performance.

5 Implementation

In this chapter, we provide an overview of the implementation details of our approach to using SIWR for Adventure. We will first describe the overall algorithm implementation as well as the modifications made during development, followed by the sketch features and sketch integration. Our starting point are the release version of ALE, along with the openly available implementation of SIW for the ALE domain [1], done by Bonet et al [7].

5.1 SIWR Implementation

The original implementation of SIW by Bonet et al [7] provides three variants: a standard SIW, a Rollout-IW, and a randomized algorithm. We will focus on the standard SIW implementation, as it is the basis for the SIWR algorithm.

5.1.1 Implementation Framework

System Architecture

The overall system architecture follows a modular design, with separate components for the ALE interface and the SIW algorithm. The key components are:

1. **Main Control (main)**: This module initializes the ALE environment, encompassing the emulator and simulator, using the specified parameters. Furthermore, it initializes the SIW algorithm instance with the designated parameters, including the maximum node count, time constraints, and feature extraction modalities. It also computes the background images, vital for the feature extraction in screen-based modes, and manages episodic execution of the SIW.
2. **Planning Interface (planner)**: Defines the abstract planner interface with implementations for random, fixed-sequence, and IW-based planners.
3. **Simulation Planner (sim_planner)**: Facilitates the common functionalities for various algorithms, including state management, feature extraction, and novelty table operations.
4. **Tree Structure (node)**: Encapsulates the tree node representation, including parent-child relationships, and mechanisms for node enlargement, value backup, and branch selection.
5. **Feature Extraction (screen)**: Holds the logic for extracting features from the game state, including RAM-based features and pixel-based features that use background subtraction.

[1] The source code is available at `https://github.com/bonetblai/rollout-iw/tree/master`, which we forked and modified to implement SIWR.

6. **SIW Algorithm** : Contains the fundamental logic for the SIW algorithm, which uses value backup and novelty pruning to accomplish the breadth-first search.

The modular design still allows for a number of configurable parameters; here grouped the most relevant ones into the following categories alongside their default values. They will be explained in the following sections where relevant.

Episodic Execution Loop

The episodic execution loop is managed within the main control module. For each episode, the following steps are performed:

Initialization: The ALE environment is reset to its initial state, as well as other global parameters like the accumulated reward, accumulated frames, number of simulator calls, expanded nodes' number and accumulated number of actions executed. In order to stabilize the background subtraction and the game state, an initial prefix of noop action sequence is executed. The length of this sequence depends on the parameter *initial_noops*, when set to 0, a singular random action is performed.

Planning-Execution Loop: At each decision point within the episode, the SIW:

- The SIW planner is invoked to build and search the planning tree within the given resource constraints (time, nodes, simulations) with the current node as root. It returns the best action sequence found, based on the reward function from the simulator. Then this action sequence is executed in the ALE environment. One can modify how much of this action sequence is executed before the next decision point, but by default the entire sequence is executed.
- The current tree is advanced along the actions taken. The rationale behind this is to improve efficiency across decision points, and the code offers three caching modes. We will be utilizing **full caching**, where the entire tree, as well as the state information is preserved. Due to fact that we are not using the other two modes, we will not explain them in detail here.
- After this, the loop continues as we have arrived at a new decision point.

Termination Check: The loop continues until a terminal state is reached, the game ends, or a predefined maximum frame limit is reached (default is 18000 frames).

5.1.2 Implementation Details

5.1.3 SIW

The SIW algorithm's core functionality happens withing the function `get_branch()`, which constructs or extends a lookahead tree from the current node state. It also maintains the **novelty table**, which maps atoms (initialized with max integer value) to the lowest tree depth.

At this depth, these atoms were true for the first time. Additionally, the function also utilizes a **priority queue**, which is a min-heap ordered primarily by depth, where ties are broken via the path rewards.

Before we dive into the main steps of the IW search algorithm, it is vital to explain `frame_rep_` attribute. This attribute represents whether the node's features match the parent's features or not. Depending on this, either the newly created node has `frame_rep_` $= 0$, when the node's features differ from the parent's features, or is initialized with the parent's `frame_rep_` value incremented by 15 (frame_skip), when the features are the same. The nodes with an increased `frame_rep_` are also handled differently during the node expansion phase. If the `frame_rep_` exceeds the *max_rep* parameter, the node is not expanded further to prevent infinite loops. This mechanism effectively allows the planner to skip frames where the agent's state might not change significantly. Another reason is in some games, one cannot proceed further until another character in the game takes an action, e.g. going back to a previous location. They need to return to their previous location so we can avoid getting hit by them while collecting items or to kill them. In such cases, not all actions are expanded as this would create exponential branching without actual state change. The main steps of the algorithm where the IW search is performed are as follows:

1. **Starting phase**: One initializes a new empty priority queue, adds the root node's children to it, and initializes a new novelty table.
2. **IW Exploration loop**: While the queue is not empty and the resource budgets (time, nodes, simulations) are not exhausted, the following steps are performed:
   (a) **Node Selection**: The node with the minimum depth is popped from the priority queue. This is the top node in the min-heap.
   (b) **Node Information Update**: If the node's information is incomplete (i.e., it has not been simulated yet), a simulation is performed to obtain its reward and feature set.
   (c) **Terminal and Frame Repetition Check**: If the node is terminal or its `frame_rep_` exceeds the *max_rep*, the node is skipped for expansion, pruning this node.
   (d) **Novelty Pruning**: If the node's `frame_rep_` $= 0$ (signifying a new state), the novelty of the node is evaluated using the novelty table. To accomplish this, one goes through all the features (atoms) of the node and looks if any feature turns true for the first time at this node. Thus, in the novelty table this feature's depth, shallowest depth where it has been true, will be greater than the current node's depth. If so, the first found feature's index is called novelty atom, and the node is considered novel. Also, the novelty table is updated to reflect that this feature has now been seen at this depth. In other cases, the node is pruned. Therefore, if the cousin of the current node (another node at a similar depth) has been explored first

and activates the same atom for the first time in the search that the current node also activates, the current node is eliminated.

(e) **Node Expansion**: So far, the node has passed the above novelty check or has a `frame_rep_` higher than 0. In both cases, since the nodes are not terminal and do not exceed the *max_rep*, they are expanded. Here we differentiate between two cases:

1) If the node's `frame_rep_` is 0, the node is expanded regularly, creating a child node for each action.

2) If the node's `frame_rep_` is greater than 0, only the child with the same action as the current node is expanded. This child's `frame_rep_` is incremented by the frame skip value (15 by default). Thus, the `frame_rep_` are always multiples of frame skip (default 15) or 0.

(f) **Queue Update**: The newly created child nodes from the expansion are added to the priority queue for further exploration.

3. **Branch Selection**: After the loop concludes, either due to resource exhaustion or an empty queue, the final step is to select the best action sequence from the root node. In our implementation, we follow a different approach than the original SIW implementation. Originally, the action sequence leading to the child node with the highest Q-value is selected, whereas in our case, we select the shortest action sequence leading to a node that fulfills the goal condition. We will explain this in more detail, but the rationale behind this change is that in our case, we use sketch rules that define specific goal conditions. Therefore, it is more relevant to find the shortest path to a goal state rather than the path with the highest Q-value.

5.1.4 SIWR

To adapt the existing SIW implementation into SIWR, we made several modifications to incorporate sketch-based guidance into the planning process.

Sketch Rule Definition

```
1  struct SketchRule {
2      // Precondition: Evaluates whether the rule is applicable
3      std::function<bool(const SimPlanner&,
4      const std::vector<pixel_t>& current_screen,
5      const std::vector<pixel_t>& previous_screen)> precondition;
6          // Goal: Evaluates whether the rule's subgoal is achieved
7      std::function<bool(const SimPlanner&,
8      const std::vector<pixel_t>& current_screen,
9      const std::vector<pixel_t>& previous_screen,
10     const std::vector<pixel_t>& grandfather_screen)> goal;
11         std::string description; // Human-readable rule description};
```

Every sketch rule is implemented as a structured pair of functions that work on pixel states of the screen. The precondition function checks if the sketch rule can be applied in the current state by analyzing the current and previous screen pixel data. The goal function determines if the subgoal defined by the sketch rule has been achieved, using the current, father's, and grandfather's screen pixel data. Originally, we used the previous screen to identify the cube in some rooms like the black key room, where the items flicker, or the maze area, where in some cases the cube is partially obscured by walls. Now, we do not require the previous screen for any particular room, as we use template matching to identify the cube in such cases. Thus, we could have removed the previous screen parameter from both functions and even the grandfather screen parameter from the goal function, but we kept them for future sketch rules that might require temporal context. Even though, typically most of our sketch features only rely on the current screen, some sketch features like detecting chalice possession require temporal context. In particular , the chalice changes its color every frame, sometimes mimicking the color of the ground around cube. Hence to determine if the agent is carrying the chalice in such cases, one needs to check if the chalice was present in the previous screen around the cube. For Adventure, we established 24 sketch rules that collectively break down the primary task of defeating the game (detailed in Chapter 4).

Node Structure Enhancements

To facilitate sketch-based planning, we augmented the existing node structure used in SIW with additional attributes.

```
class Node {
    // Sketch tracking
    std::vector<bool> pre; // Which sketch preconditions hold
    std::vector<bool> post; // Which sketch goals are achieved
    bool sketch_completed; // If any sketch rule is completed
    // State representation
    // Grayscale screen pixels (210×160)
    std::vector<pixel_t> screen_pixels_;
    Node* grandfather; // Grandparent for temporal context

    // Game-specific state tracking
    bool node_ykeyt, node_bkeyt, node_yswrt, node_chalicet;
    bool node_ydragon, node_gdragon;
    int node_Last_room_color;

    // Additional fields for sketch integration ...};
```

The grandfather pointer allows for sketch rules that necessitate multi-step temporal reasoning. For instance, identifying if an object, who constantly shifts its colors is present in the room. For this one could compare the present screen pixels with both the parent and grandparent pixels However, this attribute is only used for debugging purposes in our current implementation, as none of our sketch rules require it. Furthermore, to avoid unnecessary sketch feature computations, we have an attribute `sketch_completed` that tells if the node's sketch features have been computed.

Moreover, we added boolean flags to track game-specific states, such as whether certain keys have been obtained or specific events have transpired. These flags are crucial for evaluating sketch preconditions and goals that depend on the game's internal state beyond pixel data. For example, in Adventure after killing the yellow dragon, the flag `node_ydragon` is set to true, which is then used in sketch rules that require the yellow dragon to be defeated. These attributes are needed as the killed dragon stays in one room, hence when one is another room, the pixel-based features alone cannot determine if the dragon is killed or not. Also, Adventure being an old game, when one touches the item (e.g. key), it floats around the agent and can even disappear from the screen, when the agent is too close to a wall. Therefore, just relying on pixel-based features is not sufficient to determine if the key is being carried by the agent or not. These attributes are set when the node is created during the node expansion phase, based on the parent node's attributes. While calculating their sketch features, these attributes can also be updated based on the sketch features. To conclude, the pre- and post-boolean vectors track which sketch preconditions are satisfied and which sketch goals have been achieved at each node.

Sketch Integrated BFS exploration

The core of the SIWR implementation lies in modifying the SIW search process to use sketch rule evaluations to guide the search.

Root Node Sketch Feature Computation: Before starting the IW search, we compute the sketch preconditions and goals for the root node based on its screen pixels and game state, the additional game-specific node attributes. In other terms, one determines which sketch preconditions are satisfied and which sketch goals are already accomplished at the root. We designate as active the sketch rules whose preconditions are fulfilled at the root for the following search. This information is stored in the root node's `pre` and `post` attributes.

Per-Node Evaluation: During the IW search, for each node that is expanded, we first check if the sketch features have already been computed using the `sketch_completed` attribute. If not, we compute its sketch preconditions and goals using its screen pixels and game state and set the attribute to true. Similarly to the root node, we update the node's `pre` and `post`

attributes to reflect which sketch preconditions are satisfied and which sketch goals have been achieved at that node. As mentioned earlier, when during the search, a node's children are being expanded, we set the game-specific attributes based on the node and the action taken.

Sketch Rule Completion Check: During the IW search, we check if any sketch rule has been completed at the current node, whose preconditions were satisfied at the root. If a sketch rule is completed, we immediately return the action sequence leading to this node as the solution. Thus, we break out of the search early. Otherwise, we fall back to standard action selection as in SIW, which would be either a random action sequence or the longest zero-reward action sequence. This ensures perturbation before redoing the IW search.

Considering the memory overhead, one adds $n = 33,600$ pixels overhead for each node to store the screen pixels. Nonetheless, the pruning of non-novel nodes helps mitigate the overhead. Furthermore, the integration of sketch adds roughly $\mathbf{O}(k)$ time overhead, where k is the number of sketch rules during the IW search. The reason being that for each node expanded during the IW search, we need to evaluate k sketch rules. For this, we need the screen pixels and game-specific attributes of the node, which are stored at each node. Despite this additional overhead, the sketch-guided approach can significantly reduce the number of nodes expanded during the search, as it can lead to early termination upon sketch completion. This indirectly reduces branching in irrelevant directions. Additionally, to mitigate the computational cost during the search, screen pixels and sketch evaluations are cached at each node.

5.1.5 Modifications

During the development of the SIWR implementation, we made two modifications to the original SIW algorithm to better accommodate sketch-based planning in the Adventure game (Section 4.4). This was done to ensure that the sketch rules could effectively produce an action sequence to navigate through the maze. Hence, these modifications only apply when the cube is in the maze section of Adventure and can be seen more as fixes rather than general improvements to the SIW algorithm.

Node Expansion Adjustments: Instead of just expanding the child node with the same action as the parent when the `frame_rep_` is greater than 0 but less than or equal to `max_rep`, we now expand all children of such nodes if any sketch precondition is active. This change ensures that all potential paths relevant to a sketch feature completion are explored, even when the node represents repeated frames. Therefore, this change can also be seen as a solution to a runtime issue faced during the development of SIWR for Adventure, as at the core we are expanding more nodes, but not exponentially more. Previously, in such cases we used to expand only one child, with the same action and frame repetition of parent node plus frame skip. With the parent node having `frame_rep_` > 0 (meaning it was `frame_rep_` $= 15$), the

child node will have `frame_rep_` = 30. the same process repeated: only the child with the same action was produced. This continued until a child's `frame_rep_` exceeded `max_rep`, at which point it was pruned. Thus, in default case, where `max_rep` is 30, we expanded up to one node in such a case. Now, instead of just expanding one child, we expand all children of such nodes, which results in up to 6 children being expanded in such cases (Adventure has 6 compact actions). The subsequent expansion of these children depends on their resulting state.

- **Similar Feature State (Typical Case):** If a child has the same features as its parent, its `frame_rep_` will be 30. Since this is equal to or less than `max_rep`, all children of this node will also be expanded, and their frame repetition will again be controlled.
- **New Feature State:** If a child reaches a novel feature state, its `frame_rep_` is reset to 0. It is treated as a novel node and will be expanded regularly.
- **Exceeding Max Repetition:** If a child's `frame_rep_` (e.g., 45) becomes greater than `max_rep`, it is pruned.

Therefore, for typical cases with default settings, there would be linear increase in the number of nodes. Nonetheless, if any of the children reach a novel feature state, they will have `frame_rep_` = 0 and will be expanded regularly. Consequently, the increase in the number of nodes will be exponential in such cases, but when this fix is used the needed action sequences are short. Thus, the depth of the tree is limited artificially, preventing excessive node expansion and in such cases the sketch feature completion is more likely, justifying the increased exploration. This fix is vital for the maze navigation sketch rules, which we elaborate on in Chapter 6. In short, without this fix certain nodes that are relevant for sketch completion were not expanded, leading to failure in completing the sketch rules related to maze navigation.

Relaxed Pruning for Sketch Relevance: The relaxation of when to prune a node, due to novelty was altered from being greater than node_depth to node(atom) $\geq$ node_depth. This change prevents premature pruning of nodes that might complete sketch rules but appear non-novel due to node ordering in the breadth-first IW search. However, to avoid exponential branching, this relaxation is only applied if the node's `frame_rep_` $\in [0, \texttt{max\_rep})$. Hence, this change ensures that nodes relevant to sketch completion are not overlooked while still controlling the search space growth size. This modification does increase the number of nodes expanded exponentially, but we only use it for sketch rules in blue maze, where the resulting action sequences are short. Despite this, we still try to limit the node expansion by only relaxing the pruning condition for nodes that are within the `max_rep` limit. Another way of solving this issue could have been to change the node ordering in the priority queue, but that would have introduced bias in the search. For instance, currently nodes always expand in a fixed order of actions (e.g. UP, DOWN, LEFT, RIGHT, FIRE etc.) and are also ordered like this in the priority queue. So, if we were to change the ordering of node expansion, this could lead to some actions being preferred over others, which is not desirable.

Another change we made was a skip function to speed up the debugging process. This function takes a set sequence of actions and carries it out in the ALE environment, bypassing the early phase of the game. It is particularly useful for quickly reaching specific game states, such as Adventure's maze area, without having to start the game over each time. However, this function is intended to streamline the development and testing process and has no effect on the main SIWR algorithm. This is only possible because the ALE environment is deterministic, which guarantees that carrying out the same action sequence consistently results in the same game state as stated in the Chapter 4. Furthermore, it provides the skip sequence as a single executable branch and extends each node, just like the IW search would. Once the branch has been executed within the main loop, the tree progresses to the final node of the skip sequence, and the regular SIWR planning cycle continues from that point.

5.2 Sketch Implementation and Sketch features

Following the implementation of SIWR, we now describe functions, which are the sketch features, their extraction, and the sketch rules implementation.

5.2.1 Sketch Features Extraction

Similar to the description of the features in Chapter 4, Section 4.3.1, the sketch features are extracted based on pixels of the current screen. In the following, we group the sketch features based on their functionality.

Room Identification Functions: Contains functions like `regions_for_cube`, which is used to identify each room's regions where the cube might be. Additionally, `identify_room_from_database` stores room templates of particular rooms, where the cube might be partially obscured by walls. To find the correct regions or use the right template, we check for room-specific color and pattern which help us identify the room. This is the backbone of other sketch features, as they provide context about the current room, which is essential for accurate item and cube detection.

Cube Detection Functions: The location of the cube on the screen is detected by these functions. The principal cube detection function (`highlight_cube`) combines database matching and region-based search to produce the top-left pixel coordinates of the cube. The region-based search is sufficient to find the cube throughout the game, but database matching is required in the maze area, where half of the cube may be obscured by walls. While the database matching (`find_cube_using_database`) evaluates the screen in comparison to stored room templates to help detect partially obscured cubes, the region-based search (`find_cube_without_reference`) looks for color patterns unique to cubes in specific screen

sections. Furthermore, there are helper methods to extract the cube's boundary (`get_cube_boundary`) and center (`get_cube_center`).

Item Detection Functions: By examining their unique color patterns and measurements, they assist in locating the position of important objects (yellow key, black key, sword, and chalice) on the screen. The method `detect_items_entire_screen` often uses color and cluster size evaluation to determine whether items are present over the whole screen. In contrast, functions like `detect_items_around_cube` just search for objects that are close to the cube. In order to do this, every non-background pixel from the entire screen or the area around the cube is collected and arranged based on its color and location. Lastly, these clusters are filtered by identifying items by matching known item properties with cluster attributes (color, size limits). Specifically for the chalice, we used a pattern matching method (`check_pattern_chalice`) in addition to the cluster size analysis since the chalice alters its color every frame, rendering color-based detection inconsistent.

Item Possession Functions: The core function here is `detect_ykey_touching_cube`, which determines if the agent is carrying or touching a particular item. For this, the game-specific node attributes (e.g. `node_ykeyt` for yellow key) are used to track if the item was picked beforehand or not. Depending on this flag, one first checks if the item is still in the cube's vicinity using the item detection functions. If not, then the item must be dropped and check if the item is present on the screen utilizing the item detection functions. If the item is not found on the screen, we check if the agent is touching the item, because to clustering of similar colors, the item might not be detected as a separate entity. This is done by analyzing pixels around the cube's boundary area for item-specific color patterns. If so, we conclude that the agent will be carrying the item and set the appropriate game-specific node attribute to true and others to false. Moreover, for each item we have specialized functions like `ykey`, for the yellow key, which calls upon the core function with item-specific parameters.

Dragon State Functions: Primarily, the functions `ydragon_killed` and `gdragon_killed` determine if the yellow or green dragon has been defeated. They are also one that update the game-specific node attributes (e.g. `node_ydragon` for yellow dragon) when a dragon is killed. For this, we use clustering techniques, similar to the **item detection functions**. These clusters representing items are then first filtered out, then using pattern matching, we identify a cluster representing the dragon. For each dragon's state, which in total are 3, we have a pattern. Thus, the functions depict the dragon's state on the screen. Additionally, functions like `dectect_dragons_in_room` help us identify dragons' presence in the room via the same techniques.

5.2.2 Sketch Rules Implementation

We now describe how we implemented the sketch rules for Adventure (Chapter A). Instead of going through all 24 sketch rules, we will explain which functions represent the sketch features.

at(): This feature is implemented using cube detection functions. For the maze navigation sketch rules, we directly utilize cube detection functions and precise coordinate checks to break down the maze navigation into subgoals. They include reaching specific areas in the maze, which can be described using coordinate boundaries. This is done by checking if the cube's coordinates, obtained from `highlight_cube`, fall within predefined ranges corresponding to target areas in the maze.

in(): This sketch feature is implemented using the item and dragon detection functions. Essentially, the function `ykeyr` uses `detect_items_entire_screen` to check for the presence of the yellow key on the screen. If the yellow key is detected, the sketch feature is considered satisfied. Similar functions are created for the black key room (`bkeyr`), sword room (`yswordr`) and chalice room (`chalicer`). For the dragon rooms, the functions `ydragonr` and `gdragonr` translate directly to the sketch feature. Important to mention here is the mutual exclusion for room detection rather than explicit room tracking. So, if one is in the yellow key room, one cannot be in any other room at the same time.

have(): For this sketch feature, we utilize the item possession and detection functions. For instance, the function `ykey` employs `detect_ykey_touching_cube` to determine if the agent is carrying the yellow key. Likewise, functions such as `bkey`, `ysword`, and `chalice` are implemented for the black key, sword, and chalice, respectively.

killed(): This sketch feature is implemented using the dragon state functions. For each dragon, we have a dedicated function `dragon_killed` to check the particular dragons' state on the current screen. These functions utilize the core function `detect_dragon`, which identifies the dragon's presence and state on the screen using clustering, pattern matching techniques alongside state variables. The state variables help track if the dragon was killed in previous nodes, which is essential as we are in a partially observable setting.

6 Evaluation

We evaluated SIWR using the sketch rules defined in Section 4.3.2 against Reward-based SIW in Adventure under identical computational budgets. The evaluation design explicitly addresses sparse-reward challenges through budget dimensions [10, 18]:

6.1 Budget Rationale and Parameters

Following Theorem 1, for k-width problems SIWR's complexity scales as $O\big(b \cdot N^{|\Phi|+2k-1}\big)$ time and $O\big(b \cdot N^k + N^{|\Phi|+k}\big)$ space. In Adventure, the sketch width k is 1 (as illustrated in Chapter 4), the branching factor b is 6 (actions: up, down, left, right, fire (drop), nothing), and $|\Phi|$, the number of features in the sketch, is 20 in our case (Section 4.3.1). Our sketch features are exactly `have()` for each of the 4 items (yellow key, black key, sword, chalice), `in_room()` for each of the 13 rooms, and `killed()` for each of the 2 dragons. Also, the `at()`, which indicates the cube's position, can be seen as a numerical feature derived from the room features. Regarding the state variables, we did not include them in $|\Phi|$ as they are only used to track the dragons' states (killed or alive) or which item is currently held by the cube. On a semantic level, this information is already captured by the sketch features `have()` and `killed()`. Thus, $|\Phi| = 4 + 13 + 2 + 1 = 20$ sketch features. Additionally, as each state in the search trees is described using the BPROST feature set, which has over 20 million atoms, we can for now define $N = |F_{BPROST}|$. Hence, the complexity simplifies to $O\big(6 \cdot N^{20+2\cdot1-1}\big) = O\big(N^{21}\big)$ time and $O\big(6 \cdot N^1 + N^{20+1}\big) = O\big(6 \cdot N^1 + N^{21}\big) \leq O\big(N^{21}\big)$ space. Using these complexity bounds, we set the following budget parameters for our evaluation and explain the most critical ones in detail.

Simulation Budget: This controls the limit of the simulator calls that can be made during the tree search. Specifically, the simulator calls are made every time a new node is added to the tree, as the simulator is used to generate the screen image and reward for that state. Thus, for each node expansion, in our case, 6 simulator calls are made (one for each action). Therefore, the simulation budget directly limits the number of nodes that can be expanded in the search tree per IW iteration. However, since we also utilize caching of previously seen states to avoid redundant simulator calls, the actual number of simulator calls may be lower than the theoretical maximum. We used a simulation budget of 5,000,000 calls ($< 7 \cdot N^{21} \approx 7 \cdot 20^{21} \cdot 10^{11}$) per IW iteration, allowing for thorough exploration of the search space while remaining computationally feasible. This huge budget was chosen to ensure that the planner could fully leverage the sketch structure without being prematurely constrained. This

limit has never been reached remotely in our experiments, indicating that the planner was able to find solutions well within this budget.

Time Budget: The time budget constrains real-world computation per decision to ensure practical applicability, independent of simulator complexity. Hence, it has influence over the total time allocated to each planning call, all operations like simulator calls, node expansion, feature computations, novelty check, etc. The maximum time budget per iteration directly limits how long the planner can search for a solution in each IW iteration. Most of this budget is consumed by simulator calls for node expansions, each call involves rendering the game state, and computing features. The rest is used for novelty, frame repetition checks, and other overheads. We set this to 10,000 seconds ($< 7 \cdot N^{21} \approx 7 \cdot 20^{21} \cdot 10^{11}$) per iteration as a generous upper bound to ensure the planner could complete its search without artificial time constraints during evaluation.

Other Critical Parameters: The BPROST feature set was used for state representation. The novelty width was set to 1, and we used a maximum repetition threshold of 30 frames to prevent infinite loops. Additionally, the nodes threshold was increased to 500,000 to accommodate deeper search trees required for Adventure's complex state space. All other parameters were kept at their default values as specified in the original SIW implementation.

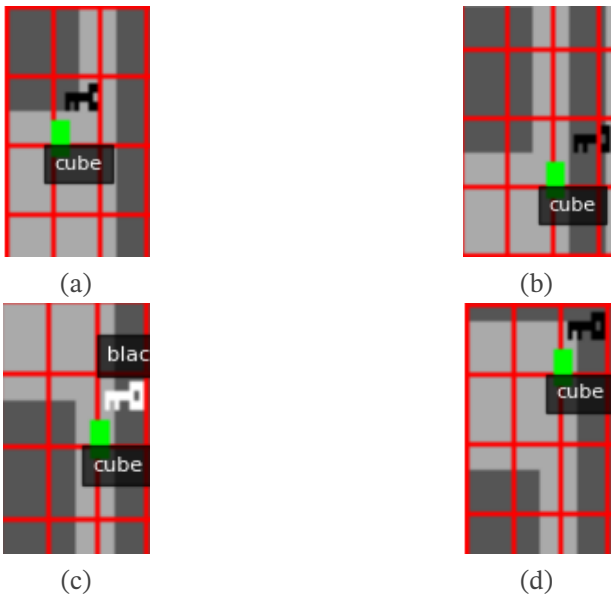
6.2 Comparative Setup

Both the Reward-based SIW and SIWR were evaluated in the Adventure game environment. Both planners received identical seeds (initial state) and the same budget combinations (simulation and time) to ensure a fair comparison. Moreover, as both planners are deterministic except while choosing random actions, they will always follow the same action sequence given the same initial state and budget. Therefore, we tested each combination for time and simulation budget once using compact actions to reduce the action search space.

Reward IW: the agent exhibited undirected room exploration, achieving only 1-2 subgoals across different parameter settings due to absent reward functions. Its log files (A.2) further illustrate the points and the Figure A.1 highlights the last location the cube reached. Namely, the agents' inability to make systematic progress, with frequent room revisits and no clear path towards key objectives.

SIWR with Original Implementation: Our initial implementation successfully navigated through the first part of the maze but encountered a critical limitation at a specific transition point. The agent could reach what we identified as "the lower half of room 6", but failed to progress further. This is illustrated in Figure A.3, where the agent struggled to progress beyond the "lower half of room 6". Additionally, we also provide the log file (A.4) for the actual run. Particularly, this exact breaking point is also highlighted in the illustration Figure 6.1a. For the

illustration, we use the skip functionality with the same action sequence as the branch taken by the planner to reach this point.



(a) (b) (c) (d)

Figure 6.1: Multiple images showing different stages of the Adventure game where the modified SIWR agent overcomes previous limitations. Picture A shows the last position reached by the standard SIWR agent and full image is shown in Figure A.7. Picture B illustrates the position after executing action 3 (Move Right) from Picture A. The whole image is shown in Figure A.8. Picture C depicts the position after executing action sequence 3,2 (Move Right, Move Up) from Picture A. The whole image is shown in Figure A.9. Picture D displays the position after executing action sequence 3,2,2 (Move Right, Move Up, Move Up) from Picture A, where the agent successfully navigates further using the modified SIWR. The whole image is shown in Figure A.10.

Three activities made up the ideal action sequence from that moment, according to additional analysis: (3,2,2). This path could not be explored because of the planner's pruning and expansion process, even though the action sequence was small. Analysis revealed the following issues:

1. **Action 3 (Move Right):** According to the BPROST state atoms, this operation produced a new state that was similar to its parent, as shown in Figure 6.1b. As a result, this node was initialized with a `frame_skip` > 0, which prevented the `frame_rep` $== 0$ condition from being met during node expansion. Since only the same action-based child is expanded when `frame_rep` > 0, the only child node created from this new state was the child of the same action (3) once more. However, the next action in the desired sequence was action (2) (Move Up), which was never expanded due to this condition. Thus, relaxing this condition allowed the planner to explore this necessary action sequence.

2. **Action 2 (Move Up):** The next action (2), with a `node_depth` of 2, was trimmed because its novelty atom value was 2. Recall, the novelty atom indicates the first novel feature of BPROST index. In essence, this shows if any new feature became true in this state for the first time during the current IW iteration. Thus, if the novelty atom value is less than or equal to the `node_depth`, then the node made no new feature true, which had not already been made true by its ancestors or siblings. This node was pruned using the original pruning criterion (novelty atom $\leq$ node_depth), which stopped further investigation. We speculate that this occured because the agent, whose color was blue, was positioned exactly between four tiles, which already contained walls of the same color as shown in Figure 6.1c. From the parent state, where it was also between four tiles, moving in any direction would still leave it between four tiles. When SIWR expanded another action (action x) from the parent state, it saw that the cube is again between four new tiles. Thus, when it looked at the child state resulting from action 2 (Move Up), it saw no difference in the feature representation of the resulting state and the state after doing action x. Therefore, the SIWR failed to find this path even with just node expansion, regarding the `frame_rep`, modification. Another reason for this pruning could be that all the tiles, which the agent was between, had parts of maze walls which are the same color as the cube. Therefore, there could not be any new feature that became true in this state, which was not already true in its ancestor states.

Modified SIWR: To address this pruning of Action 2, we modified the pruning criterion from ”novelty atom $\leq$ node depth” to ”novelty atom $<$ node depth”. The reasoning being that this would counteract the ordering of children based on action indices, which can lead to some nodes being pruned, due to their sibling nodes being expanded first. Hence, this relaxation allowed the planner to explore previously pruned but necessary action sequences.

With this modification or rather fixes, when we set the agent at the breaking point, it completed the first half of the maze autonomously. Specifically, it successfully executed the action sequence (3,2,2) to transition from the lower half of room 6 to the upper part of room 6. The final state is shown in Figure 6.1d, after which also the normal SIWR could proceed further. It managed to solve the next sketch rule (16) successfully, but in the following sketch rule (17), it encountered similar difficulties. With the modifications, the agent continued to navigate through further parts of the maze until it reached the halfway point of the maze (sketch rule 19). For the rest of the maze, we would need to break down the sketch rules (19-22) further to cover more subgoals, which we did not implement in this work. However, both fixes increase the number of nodes expanded, either linearly (first fix), exponentially in rare cases, or exponentially (second fix). Despite this, these modifications significantly improved the planner’s ability to navigate through complex state transitions. The execution log (Figure A.6) demonstrates this improved performance and the Figure A.5 shows the agent successfully navigated through the first half of the maze using the modified SIWR. Similar to the previous

case, we used skip functionality to illustrate the action sequence taken by the planner to reach this point. Thus the modified SIWR was used for the final evaluation against Reward-based SIW. In the following, we illustrate iteration-wise action sequences taken by the modified SIWR to reach each subgoal state. Each iteration corresponds to one IW search to reach a subgoal state defined by the sketch rules.

1. **Iteration 1:** Found path to yellow key.
2. **Iteration 2:** Navigated to sword room.
3. **Iteration 3:** Picked up sword.
4. **Iteration 4+5:** Moved to yellow dragon room and killed it.
5. **Iteration 6+7:** Moved to green dragon room and killed it.
6. **Iteration 8:** Collected black key.
7. **Iteration 9+10:** Returned to yellow dragon room and proceeded to the maze entrance.
8. **Subsequent Iterations:** Successfully navigated complex room transitions (rooms 6, 8, 9, 10) while holding the black key.

6.2.1 Key Findings and Interpretation

Key Findings: In contrast to reward-SIW's restricted progress, modified SIWR and standard SIWR achieved consecutive subgoal fulfillment, proving the efficacy of width-1 decomposition in conjunction with relaxed pruning. Budget analysis revealed that time, not simulator calls, was the main constraint. While atom processing and state management took up very little of the time, sketch feature extraction accounted for 85–99% of the time. With simulator utilization never surpassing 0.25% of its budget and time usage peaking at 74%, budgets were noticeably overprovisioned, suggesting that both could be reduced without compromising performance. Throughout the run, the planner oversaw challenging room changes and kept an eye on the item's condition. Further log analysis confirmed that in both SIWR versions, at most one sketch rule was active at any given iteration. Since none of the iterations required revisiting previous sketch rules, each sketch rule was only utilized once. Additionally, SIW has significantly more accumulated frames over time than both modified and original SIWR.

Interpretation: Instead of focusing on sparse incentives, SIWR achieves success by offering organized subgoals that direct search toward feature milestones. The budget breakdown shows that feature extraction overhead is high and has to be reduced. This implies that the biggest speedup would result from optimizing feature extraction through parallelization or incremental processing. Budgets can also be securely reduced (e.g., time to 8,000 seconds and simulator calls to 50,000) to free up resources for additional trials or in-depth investigation while preserving the quality of the solution. In terms of the most intensive sketch guidelines, there is a general tendency that the more actions are needed to achieve a subgoal, the longer the iteration takes.

Additionally, the planner's ability to track cube positions and item states (such as keys, dragons killed, and room colors) suggests effective state abstraction for this domain using screen pixels alone. Even though, we use state variables to track dragons' states and held items, these are set using sketch features only. Thus, they are also indirectly tracked using screen pixels alone and help the planner in this partially observable setting. Furthermore, as stated in Chapter 4, the planner's capacity to connect subgoals without dead ends or inconsistencies confirms the **well-formedness** of the sketch R2. The theoretical proof of the **finite-subgoal property** is practically validated because each drawing rule was used just once. The **sketch width** of 1 is also empirically justified because the traditional SIWR consistently solved most subproblems using IW(1) without requiring higher-width searches. The subproblems, which were not solved by standard SIWR, were only solvable after the pruning criterion was relaxed and not higher width searches. Lastly, the improved trimming criterion has to be applied when examining crucial but seemingly not novel action sequences. This allowed for the effective navigation of important transitions that the previous trimming rule would have made impossible.

Observations Regarding Black Key Collection: The log reveals an interesting phenomenon during black key collection. The agent first killed the green dragon and found an optimal path to the black key. However, we specifically wanted the key to be positioned above the cube so that we could subsequently unlock the black gate. When we modified the drawing rule to require "key above cube," the planner found an action sequence, though it was not necessarily the optimal one. We think this is due to item flickering in this setting when objects disappear for a frame before reappearing in the same location in the next frame. The feature detector may occasionally fail to identify the key's location due to this visual irregularity, which could result in less-than-ideal path selection.

6.3 Problems and Solutions

The evaluation revealed several challenges and corresponding solutions were required which we discuss below.

Insufficient Feature Sensitivity in BPROST: The biggest issue was that the BPROST feature set was insufficient for Adventure's intricate spatial reasoning requirements, despite being strong for many Atari games. Even when the agent (cube) moved to new locations on the screen, the feature representation often failed to identify them as new states. Our research indicates that even after traveling, the agent stayed in the same feature patch because the patch sizes (10×15 pixels) were too big compared to the agent's size, shape and movement increments. When we attempted to use smaller patch sizes (5×10) or the enhanced Adventure features, the planner failed early on (e.g., could not discover paths to the green dragon room), suggesting that the feature representation became too noisy or insufficiently discriminative.

As a compensating measure, we must relax our pruning technique from "atom $\leq$ depth" to "atom $<$ depth" due to this basic constraint. Because of the constraints of feature representation, this challenge shows that one of the fundamental assumptions made for the sketch width of R (Section 4.3.3) to be 1 may not hold in practice. In particular, we expected that the BPROST features would be sufficient to detect all necessary goal states as new throughout the IW search. However, due to the problems mentioned above, this assumption is not accurate in practice.

Similarly, the assumption about the **Visual Feature Stability** made in Section 4.3.3 is not always held in practice due to these limitations. The standard SIWR was able to navigate the first few rooms of the maze before becoming stranded at the aforementioned transition point, therefore this assumption is still largely accurate.
Solution: In order to allow the extension of states whose novelty was equal to the current depth, we proposed a flexible novelty criterion. This workaround acknowledged the limitations of the feature representation while maintaining the algorithm's ability to navigate the surroundings. Future studies should investigate domain-specific feature engineering or more complex feature representations for spatial reasoning problems.

Frame Repetition Limitations: The limited node expansion when `frame_rep` > 0 prevented expansion of nodes in critical states.
Solution: This condition was changed so that nodes would expand properly as long as `frame_rep` $<$ `max_rep`, even if `frame_rep` > 0. This made it possible to explore crucial action sequences by expanding nodes that had previously been cut because of frame repetition. Additionally, it did not lead to excessive looping because the novelty trimming and overall repetition barrier remained in place. Also, the actual cost of expanding these nodes was little because they would usually lead to the same state and be quickly cut. In the few cases where they did produce new states, it was essential for navigating the environment.

Item Flickering: Temporary item identification errors brought on by visual anomalies in the game, including item flashing, led to suboptimal path choices. Due to this, the current Sketch features F_S were not completely immune to these visual aberrations. However, while this flickering makes it challenging to determine the optimal course of action to accomplish a subgoal, this does not violate the **Sketch Precondition Accuracy** assumption mentioned in Chapter 4. Nonetheless, the preconditions in the sketch rules faithfully capture the game's circumstances.
Solution: One could implement multi-frame verification for item detection, but we opted out of this due to computational overhead. Therefore, this remains an area for future improvement, potentially through more robust visual feature extraction techniques.

Sketch Feature Sensitivity: In more intricate layouts, such as the maze, where the node might be obscured, detection was restricted by hand-coded room sections. Furthermore, the

clustering techniques used for item detection did not always accurately identify objects in settings where the cube was trying to pick them up. This is since the cube and the item are often classified together due to their similar hue. Although individuals could easily perceive that the cube was in contact with the object, the basic color clustering algorithm originally failed to do so. The cube detection would also fail in these circumstances since the initial cube detection function anticipated the cube would be alone in its cluster.
Solution: To increase detection robustness, we supplemented the room region tests with template matching. However, since the template matching necessitates maintaining extra templates for the problematic rooms, this is still an area that has to be improved. Perhaps learnt feature detectors for room recognition might be investigated in future research.

We also modified the cube detection algorithm to consider clusters that contain the cube color and have background colors surrounding them-this way, even if the cube and item are clustered together, the cube can still be detected. Regarding item detection, once we could reliably detect the cube, we modified the item detection algorithm to check for items in the immediate vicinity of the cube's detected position. This way, even if the cube and item are clustered together, we can still detect the item based on its proximity to the cube. This improved the robustness of item detection during pick-up scenarios, and we could successfully say when the cube will pick up an item.

Sketch Dependence: Incorrect rule ordering can cause chaining failures. However, our implementation shows that with proper rule design and the relaxed pruning criterion, effective subgoal chaining is achievable even in complex environments like Adventure.

Sketch size: At first, we had a singular sketch rule for navigating the maze, which proved insufficient due to the maze's complexity and BPROST feature limitations. Thus, we decomposed the maze navigation into multiple sketch rules corresponding to individual rooms and transitions. However, this decomposition increased the sketch size $|\Phi|$, impacting computational complexity as per Theorem 1 and still was insufficient for full maze navigation. One still needs to consider that a major contributor to this was the BPROST feature limitations discussed earlier.

6.4 Consequences

The evaluation demonstrates that SIWR with appropriate modifications can successfully navigate complex, sparse-reward environments like Adventure by leveraging policy sketch rules for structured exploration and relaxed pruning criterion to ensure necessary state exploration. Furthermore, the main challenge identified was the limitations of the BPROST feature representation for spatial reasoning tasks, suggesting avenues for future research in feature engineering and sketch design.

Also, the modifications made illustrated that some of the theoretical guarantees of the SIWR algorithm (mentioned in Section 4.3.3) do not always hold in practice. The main reason being the item flickering, or assumptions about the visual feature sufficiency being invalid at certain portions of the game. Consequently, the planner did not always find an optimal action sequence to reach certain subgoals. In other cases, the planner could not even reach certain subgoals at all without the modifications made. Even after breaking down the sketch rules further for maze navigation, the planner could not reach the end of the maze due to these limitations. Nonetheless, most of the theoretical properties still hold in practice, as they hold for most part of the game. Therefore, the standard SIWR could still navigate through the initial rooms and parts of the maze before getting stuck.

On the one hand, most of the other approaches in this domain heavily rely on the reward signal, which means when these algorithms are applied to domain like Adventure, they completely fail to make any progress at all. Thus, when these approaches are applied in real life scenarios, they would need a sophisticated reward shaping mechanism to be able to make any progress. This is where SIWR shows its strength, as it completely bypasses the need for a reward signal by relying on the sketch structure alone to guide exploration. Hence, it reflects a more human-like approach to problem solving, where we break down complex tasks into smaller, manageable subgoals based on our understanding of the domain. On the other hand, SIWR's strength also lies in its state abstraction mechanism through the feature representation. So, if the feature representation is not sufficient to represent the necessary states for solving the problem at hand, then SIWR would also fail to make any progress. To conclude, SIWR shows promise as a planning approach in complex domains when guided by well-formed subgoal structures.

7 Conclusion

The basic problem of planning in high-complexity, sparse-reward environments—such as those seen in Atari 2600 games like Adventure—was the focus of this thesis. Conventional width-based planners, such as SIW, perform well in many deterministic domains but fall short in certain situations due to large subgoal width and non-serializable objectives, which result in exponential search complexity. A major hurdle for these planners is a sparse reward signal, which makes it difficult to explore the state space effectively as there is nothing to guide the search toward meaningful progress. Likewise, in Adventure, with the only reward being granted upon completing the final objective, the agent lacks intermediate feedback to direct its exploration.

Our approach, SIWR, integrated policy sketch into the width-based planning framework. Sketch breaks down the monolithic task into a series of low width subproblems by encoding domain knowledge as a collection of high-level rules (e.g., have(sword) → kill(green_dragon)). This ensures polynomial-time solutions when the sketch width is constrained, shifting exploration from sparse atomic rewards to structured subgoal achievement. In particular, the planner's attention is diverted from the scarce environmental reward to a new goal, which is achieving structured subgoals. Therefore, the necessary search space is drastically reduced from the entire state space to only those states that are relevant for fulfilling the sketch subgoals.

This reformulation provides the crucial path for research when traditional reward signals are absent. In contrast to reward-driven exploration performed by standard IW, this enables systematic pursuit of significant progress throughout sparse-reward domains like Adventure. Ensuring that IW(1) can solve any induced subproblem in polynomial time, which lays the groundwork for the SIWR's effectiveness, requires that the prerequisite conditions for the sketch and the state features are satisfied. Specifically, the constrained sketch width and well-formedness of the sketch are the two main characteristics that determine the theoretical benefit of SIWR. By formally proving that this Adventure-specific sketch satisfies both requirements, the SIWR could, in theory be as effective as possible.

There were significant advantages to using SIWR for Adventure. Important subgoals that reward-based IW was unable to complete, such as obtaining objects, defeating monsters, and negotiating difficult maze rooms, were accomplished by the planner with the use of a 24-rule sketch. In theory, we created a sketch width of 1 for Adventure and proved that our sketch was well-formed, ensuring the efficacy of the SIWR algorithm. However, the major challenges found during practical evaluation were mostly caused by the limitations of the BPROST visual

feature set not the algorithm itself. Compared to the agent's size and shape, the tile size used in BPROST is relatively large, leading to insensitivity to small but crucial visual changes in the game state. Hence, frequently important state changes, such as the agent moving a single action forward, were not recognized as novel by IW(1). In turn, this caused excessive pruning of necessary states and ultimately stalling further sketch fulfillment. Resulting from this, a single sketch rule for maze navigation could not be created. We had to split the maze into several smaller, more precise subgoals as a result. This increased the number of sketch rules and, hence, the sequential IW searches required, which had an impact on the overall planning time. We believe that employing a more complex and sensitive feature representation for the IW search, a shorter sketch could guide the agent through the entire maze without the necessary modifications. Game-specific issues like item flashing and visual occlusions in maze chambers further affected the reliability of the created sketch rules. These two major modifications to the original algorithm are: (1) relaxing the novelty pruning condition and (2) allowing full node expansion even with repeated frames. These changes were crucial to overcoming the visual feature limitations and enabling successful sketch completion. Particularly in maze navigation, this ensured expansion of necessary nodes that were otherwise skipped due to frame repetition or novelty, enabling successful completion of some of the maze-related subgoals.

As a result, the first major portion of Adventure can be completed by the modified SIWR agent. This portion involved obtaining important items, eliminating dragons, and navigating the first parts of the maze. Overall, this was a significant improvement over the baseline SIW algorithm's aimless search. This observation suggests several interesting avenues for additional research. Namely, one should examine domain-specific design features. If they could better capture spatial relationships and object states, it would remove the necessary modifications, as visual feature sensitivity is the primary downside. It would even be useful to investigate more advanced, learned feature extractors. These advantages include a decrease in the necessary sketch size, improved robustness, and overall planning efficiency. Also, developing hierarchical sketch frameworks or automating sketch learning removes the need for tedious manual design. Hence, this steers to more adaptability across game scenarios and even different domains. Finally, the generality of the SIWR framework can be further validated by applying and improving it to other difficult sparse-reward games such as Montezuma's Revenge and Pitfall. This could reveal new research difficulties or illustrate advantages of sketch-based planning in pixel-based domains.

In conclusion, this thesis shows that policy sketch is an excellent way to introduce structured domain knowledge into width-based planners. By substituting subgoal attainment for the absent reward signal, policy sketch successfully overcomes the exploration barrier in sparse-reward scenarios. While the practical performance is currently dependent on the quality of underlying state representations via the screen features, the SIWR paradigm successfully bridges classical planning rigor with the perceptual challenges of pixel-based domains.

A Appendix

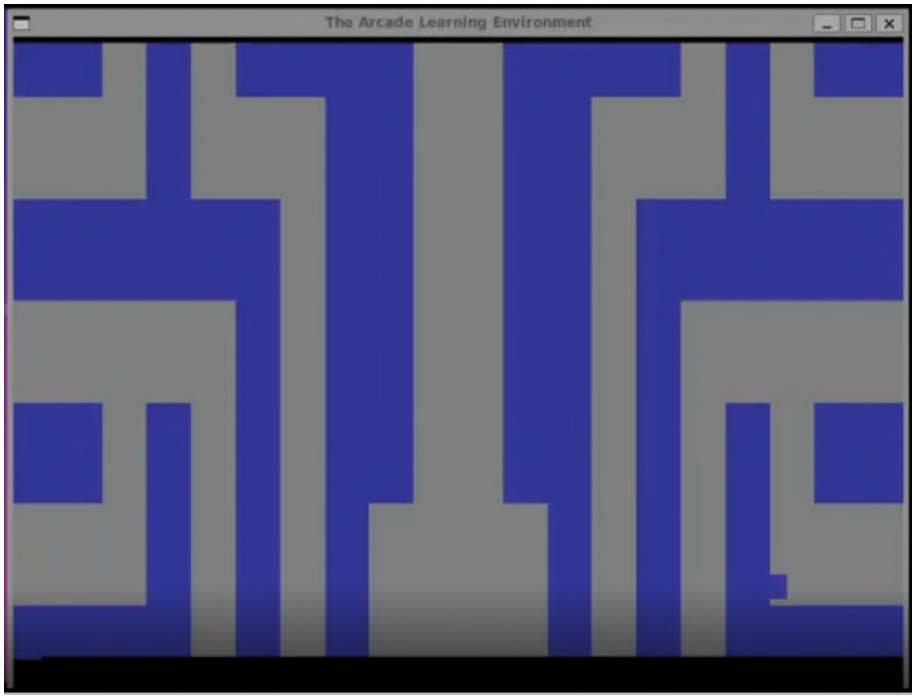


Figure A.1: Reward-IW agent exhibited undirected exploration in Adventure. Note the absence of subgoal sequences and frequent room revisits. Above is the section till where the reward-IW could proceed successfully, but on its way there it neither killed any dragons nor collected any key items.

Rule	Precondition (C)	Effect (E)
1	in(yellow_key_room) ∧ ¬have(yellow_key) ∧ ¬in(sword_room)	have(yellow_key)
2	have(yellow_key) ∧ ¬in(sword_room)	in(sword_room) ∧ ¬in(yellow_key_room)
3	in(sword_room) ∧ have(yellow_key) ∧ ¬have(sword)	have(sword) ∧ ¬have(yellow_key)
4	have(sword) ∧ ¬in(yellow_dragon_room) ∧ ¬killed(yellow_dragon)	in(yellow_dragon_room) ∧ ¬in(sword_room)
5	in(yellow_dragon_room) ∧ have(sword) ∧ ¬killed(yellow_dragon)	killed(yellow_dragon)
6	killed(yellow_dragon) ∧ have(sword) ∧ ¬in(green_dragon_room) ∧ ¬killed(green_dragon)	in(green_dragon_room) ∧ ¬in(yellow_dragon_room)
7	in(green_dragon_room) ∧ killed(yellow_dragon) ∧ have(sword) ∧ ¬killed(green_dragon)	killed(green_dragon)
8	killed(green_dragon) ∧ have(sword) ∧ ¬in(black_key_room) ∧ ¬have(black_key)	in(black_key_room) ∧ ¬in(green_dragon_room)
9	in(black_key_room) ∧ killed(green_dragon) ∧ have(sword) ∧ ¬have(black_key)	have(black_key) ∧ ¬have(sword)
10	in(black_key_room) ∧ killed(green_dragon) ∧ have(black_key)	in(yellow_dragon_room) ∧ ¬in(black_key_room)
11	have(black_key) ∧ killed(green_dragon) ∧ in(yellow_dragon_room)	in(blue_maze_start) ∧ ¬in(yellow_dragon_room)
12-21	have(black_key) ∧ in(blue_maze_start_X_position_a)	in(blue_maze_room_Y_position_b) ∧ ¬in(blue_maze_room_X_position_a)
22	have(black_key) ∧ in(blue_maze_end)	in(black_room) ∧ ¬in(blue_maze_end)
23	have(black_key) ∧ in(black_room) ∧ ¬in(chalice_room)	in(chalice_room) ∧ ¬in(black_room)
24	have(black_key) ∧ in(chalice_room)	have(chalice) ∧ ¬have(black_key)

Table A.1: Sketch Rules for **Adventure** game, where all have width $w = 1$. Here we have left out rules 12 to 21, which all involve particular position-based navigation that uses exact coordinate constraints to direct travel across the layout of maze rooms 6, 7, 8, 9, and 10. We replaced this series of rules with a single exemplary sketch rule that illustrates the intended navigation within the maze. Hereby, room X and room Y placeholders represent specific rooms within the blue maze (6,7,8,9,10), while position a and position b denote exact valid areas within those rooms. Also, X and Y do not have to be different rooms; the rule can also represent movement within the same room from one position to another. However, in implementation, all the rules from 12 to 21 are still present. So far, we have achieved that the agent can complete up to rule 18 successfully.

Figure A.2: We did not include the real log file due to its size here. We uploaded it online and linked it here. One can see the undirected exploration of the reward-SIW agent exhibiting undirected exploration in Adventure. This link where the file is available is: `https://drive.google.com/file/d/1msL7_IB3QzNtXJ5iYQCveKz-z6F_qH-2/view?usp=sharing` and can also be accessed by clicking on the image.

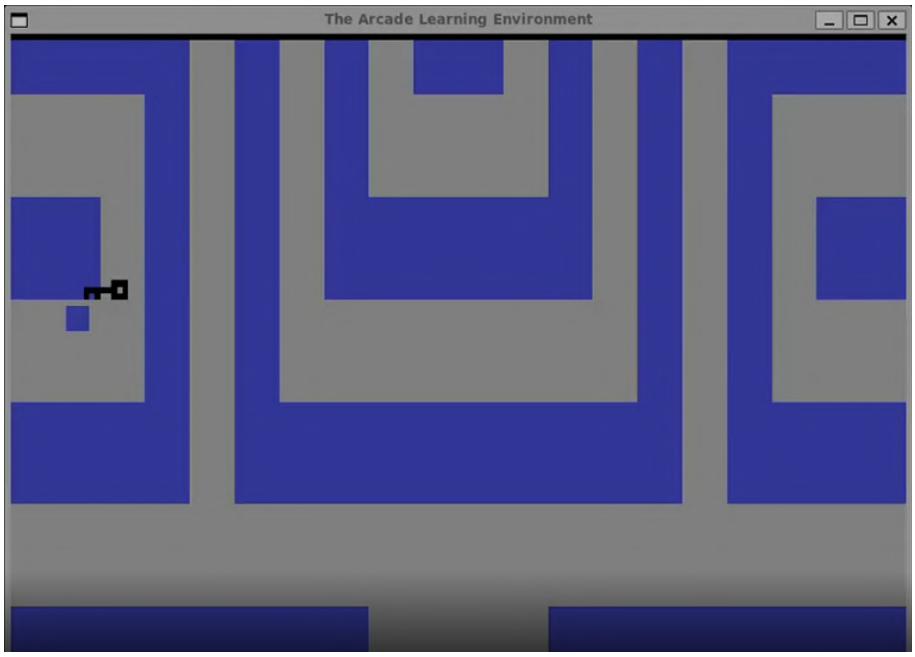


Figure A.3: Standard SIWR agent executed sequential subgoals in Adventure, which are defined by the sketch rules. Note the systematic item collection, dragon defeats, and maze navigation. The last section where the standard SIWR fails is illustrated above.

Figure A.4: We did not include the real log file due to its size here. We uploaded it online and linked it here. One can see the subsequent exploration of the standard SIWR agent using the sketch rules. For the blue maze section, we had included an additional sketch rule, which was to navigate through its entirety to reach the black castle. We never removed this rule, even after observing that the standard SIWR failed to solve this part due to feature representation issues. Despite this, this sketch rule is still visible in the log file but is never fulfilled. This link where the file is available is: `https://drive.google.com/file/d/1XB-MoobwjcbFCDSsJgoO2JdF_i-Mxg2b/view?usp=sharing` and can also be accessed by clicking on the image.

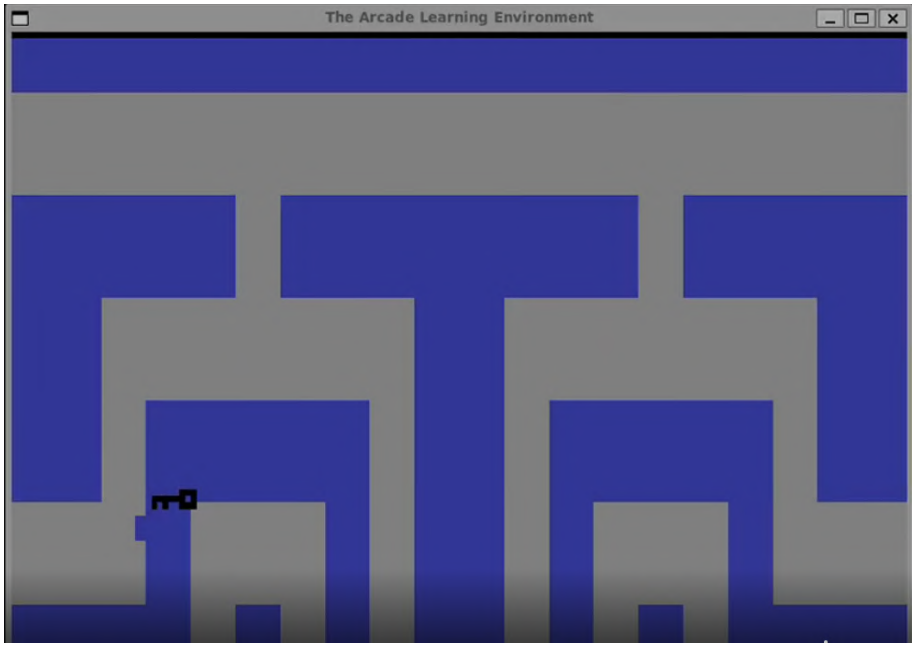


Figure A.5: Modified SIWR agent successfully completed the first major segment of Adventure after addressing feature representation challenges. The agent retrieves key items, defeats dragons, and navigates initial maze sections, highlighting the effectiveness of the proposed algorithmic adjustments. Above is the section till where the modified SIWR could proceed successfully. Further progress would require breaking down the remaining sketch rules into smaller subgoals, which we did not implement in this work.

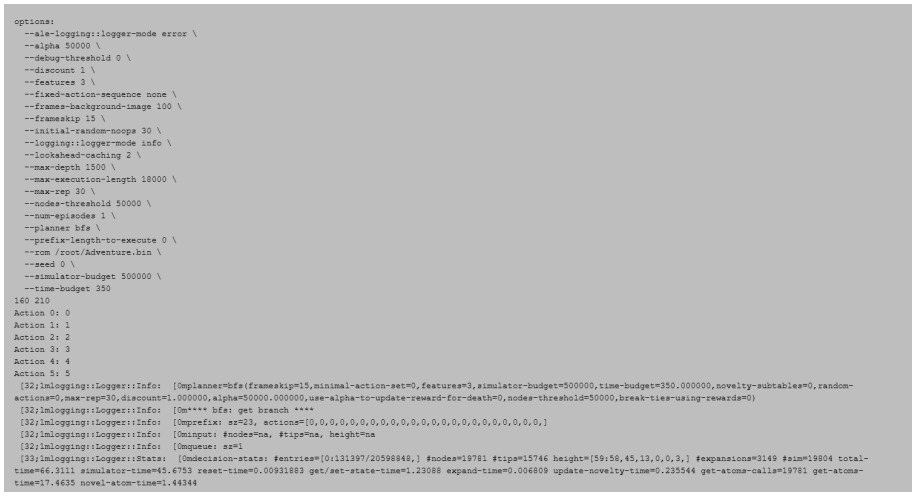


Figure A.6: We did not include the real log file due to its size here. We uploaded it online and linked it here. One can see the subsequent exploration of the modified SIWR agent using the sketch rules. For the blue maze section, we had included an additional sketch rule, which was to navigate through its entirety to reach the black castle. We never removed this rule, even after observing that the modified SIWR failed to solve this part due to feature representation issues. Despite this, this sketch rule is still visible in the log file but is never fulfilled. Moreover, one can also see the use of the skip function after the first iteration, which is why we skip so many sketch rules in the beginning.

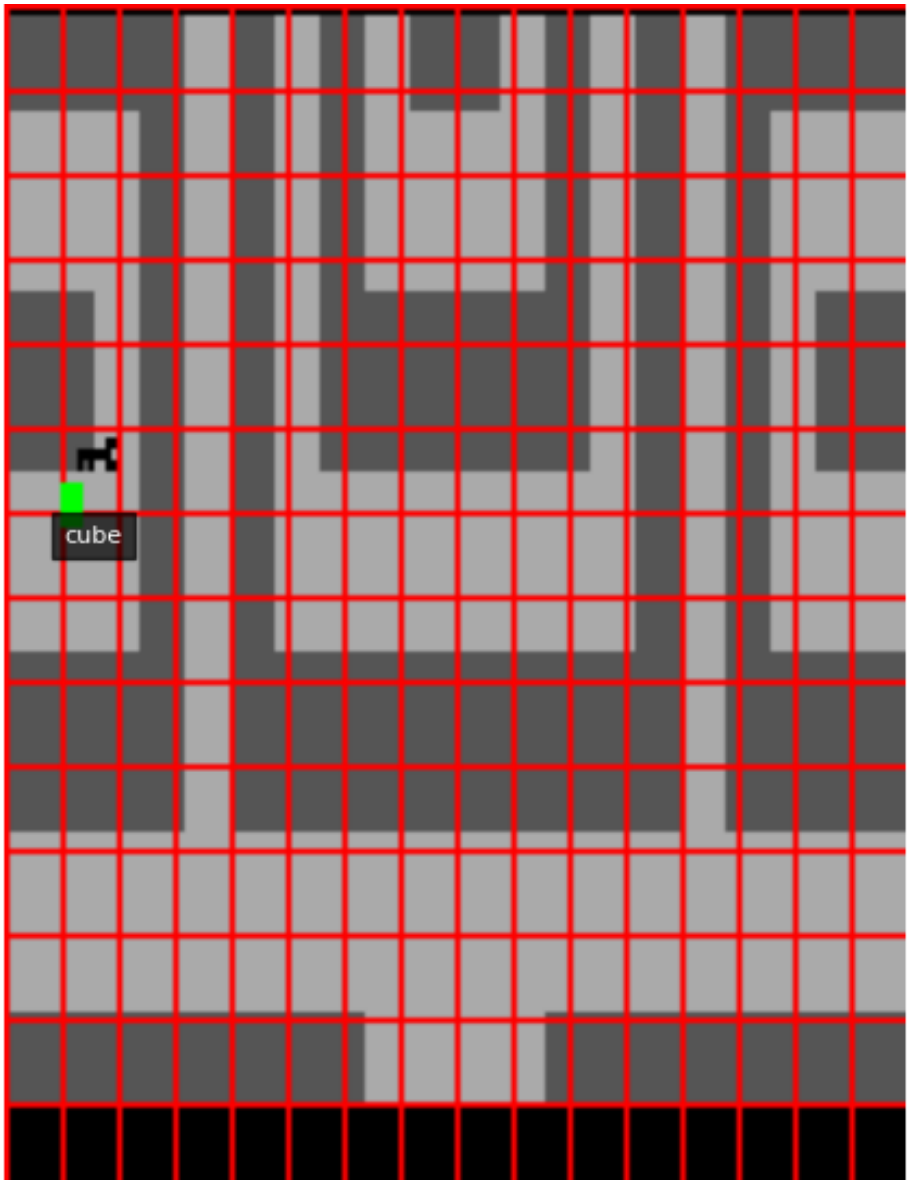


Figure A.7: This figure illustrates the location in the Adventure game, till where the normal SIWR agent could autonomously navigate using the provided sketch rules. The agent successfully collects key items, defeats dragons, and navigates through initial maze sections up to this point. Beyond this location, the agent encounters challenges in the blue maze section due to feature representation issues, preventing further autonomous progress. However, the fixed SIWR agent, with the proposed modifications, can continue from this point onward. Even though, the key is not highlighted in this figure, the algorithm is aware that the cube is carrying the black key, as it has already picked it up in the previous steps.

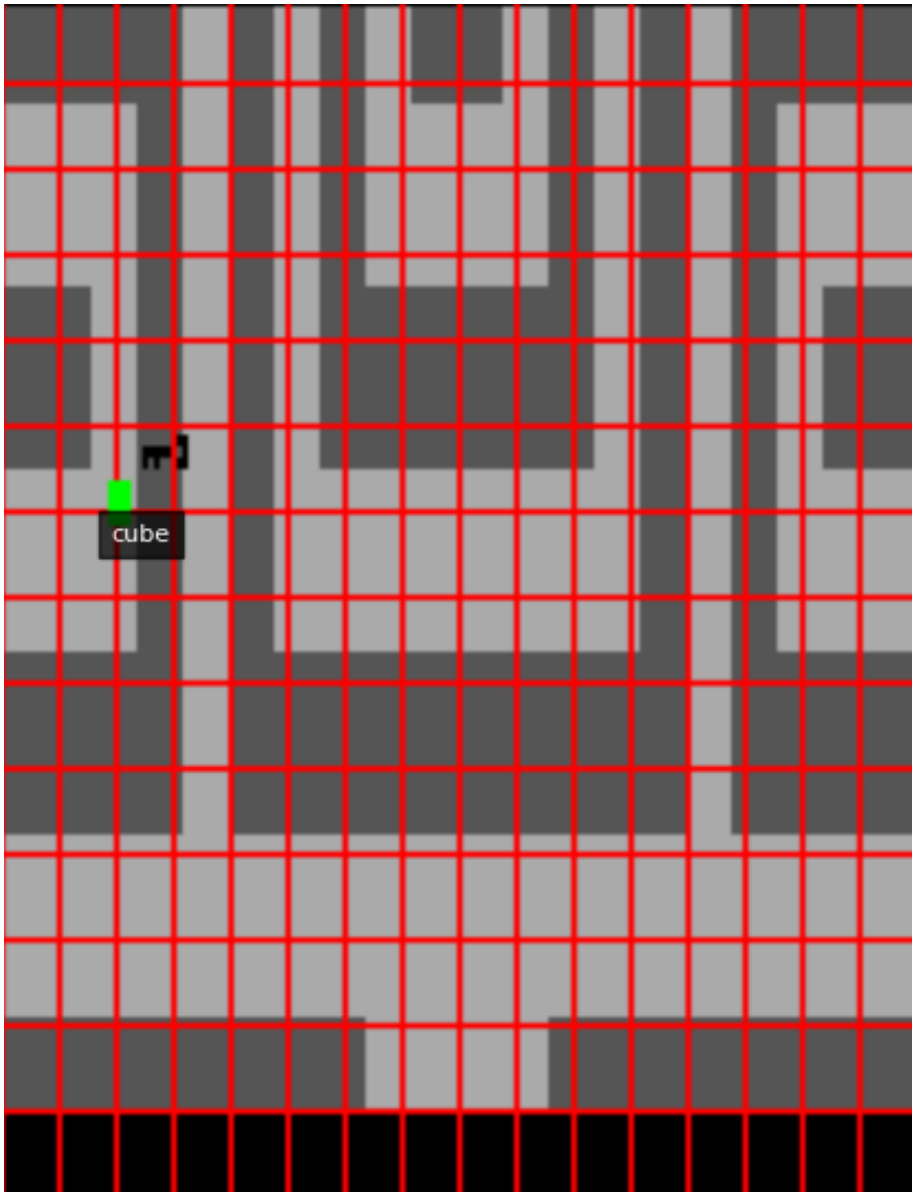


Figure A.8: From the original location, where the normal SIWR agent stopped, we set the agent manually to this breaking point in the blue maze. One arrives at this position after executing Action 3 (Move Right) from the previous position. From here, using the modified SIWR, the agent could successfully navigate through the remaining maze sections that we tested. One can see that the agent is exactly in the middle of 4 tiles, so if the agent moved in any direction it would still be in the middle of 4 tiles. Despite them being different, BPROST could treat them all identically. Thus, when SIWR looked at moving in any direction, it saw no difference in the feature representation of the resulting state (between the resulting states), which maybe a reason why the normal SIWR failed to proceed from here. Even though, the key is not highlighted in this figure, the algorithm is aware that the cube is carrying the black key, as it has already picked it up in the previous steps.

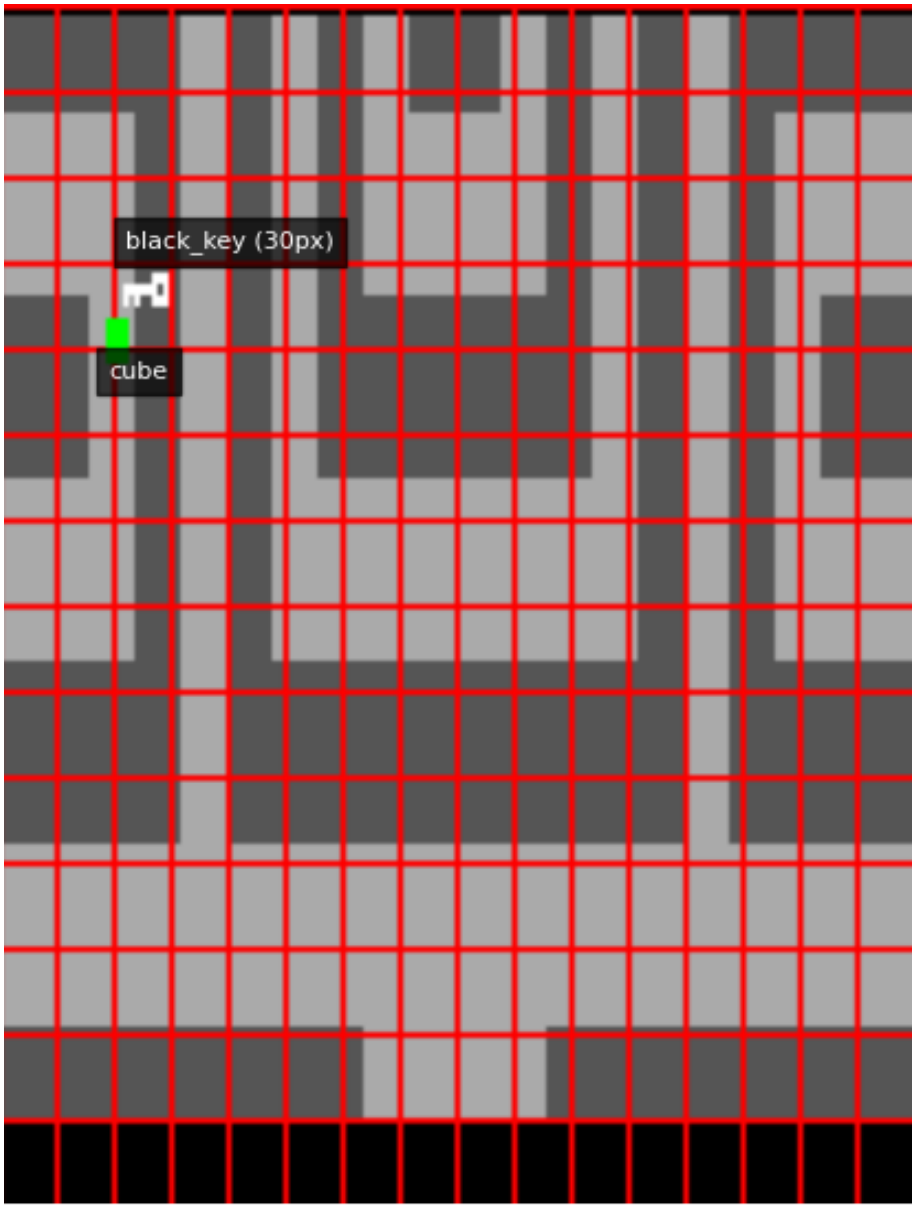


Figure A.9: From the original location, where the normal SIWR agent stopped, we set the agent manually to this breaking point in the blue maze. This position is arrived at after executing action sequence 3,2 (Move Right, Move Up) from the previous position, where normal SIWR stopped. One can see that the agent again is in middle of 4 tiles, which is why it is not novel for BPROST features. From here, using the modified SIWR, the agent could successfully navigate through the remaining maze sections that we tested.

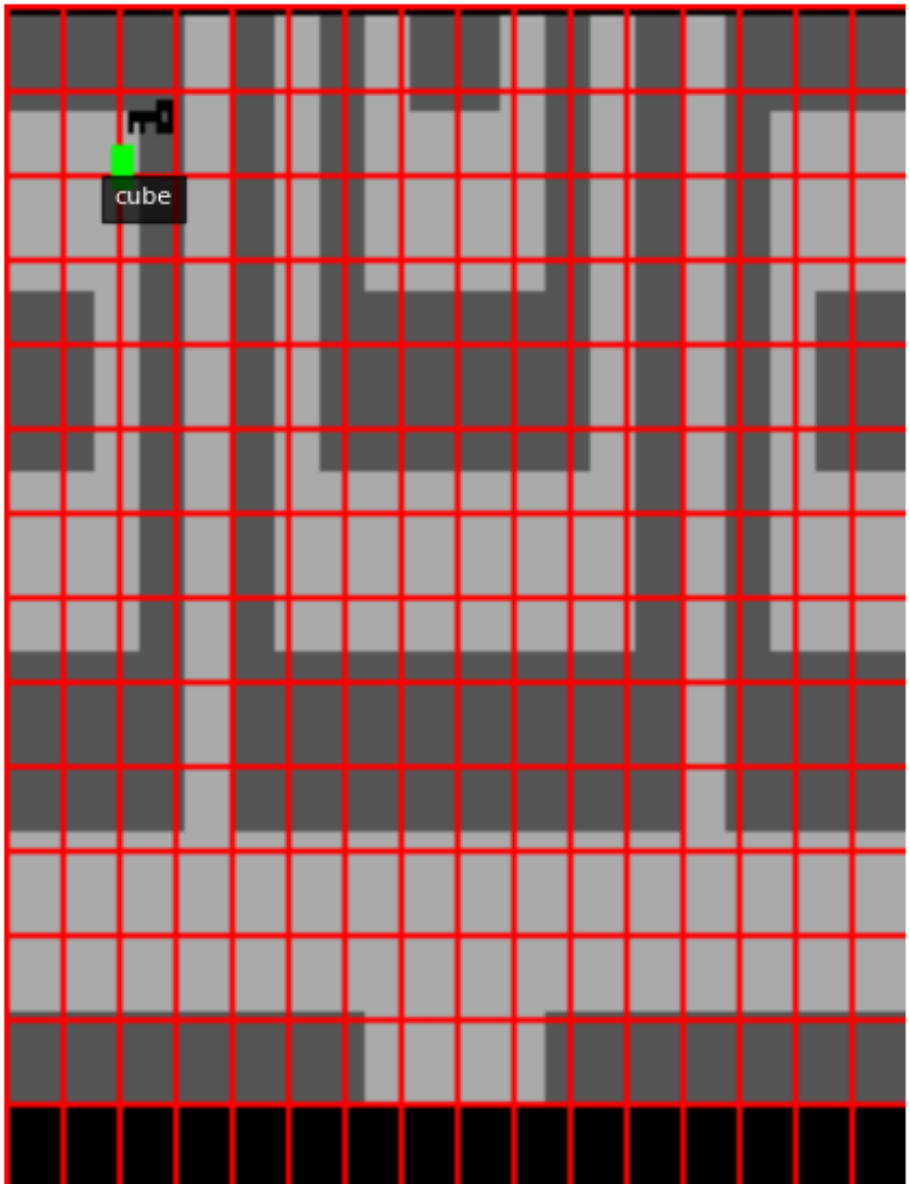


Figure A.10: This is the location in the blue maze that the agent reaches after executing action sequence 3,2,2 (Move Right, Move Up, Move Up) from the previous breaking point. From here, using the modified SIWR, the agent could successfully navigate through the remaining maze sections that we tested. Even the normal SIWR found the following path from this point onward, as the feature representation issues were resolved by then. Nonetheless, the SIWR could not reach this point autonomously without the modifications. Even though, the key is not highlighted in this figure, the algorithm is aware that the cube is carrying the black key, as it has already picked it up in the previous steps.

List of Acronyms

AI	Artificial Intelligence
ALE	Arcade Learning Environment
BPROS	Basic Pairwise Relative Offsets in Space
BPROST	Basic Pairwise Relative Offsets in Space and Time
BPROT	Basic Pairwise Relative Offsets in Time
DQN	Deep Q-Network
e.g.	exempli gratia
i.e.	id est
IW	Iterated Width
POMDP	Partially Observable Markov Decision Process
RAM	Random Access Memory
RGB	Red Green Blue
RL	Reinforcement Learning
SIW	Serialized Iterated Width
SIWR	Serialized Iterated Width with Rules
STRIPS	Stanford Research Institute Problem Solver
TAMP	Task and Motion Planning
UCT	Upper Confidence bounds applied to Trees
VAE	Variational Autoencoder

List of Symbols

General

(x, y)	Notation for a tuple
$\approx$	approximately equal
$\mathbb{R}$	set of real numbers
$\mathbb{N}$	set of natural numbers
$\mathbb{N}_{>0}$	set of natural numbers without zero
D	Domain of a function
$O(g(x))$	$\forall x \in D : \exists m \in \mathbb{R}_{\leq 0} : f(x) \leq O(g(x))$ function, where D domain of f and g
$\|F\|$	Absolute value of a boolean features set F is the number of features true
$max_{x \in D} f(x)$	represents $x' \in D$ such that $\forall x \in D : f(x) \leq f(x')$
$[x, .., y]$	Shorthand notation for a subset of $\mathbb{N}$, where $y > x$. With this we refer to the set that contains all the natural numbers from x to y inclusive.
$[x, y]$	Shorthand notation for a subset of $\mathbb{R}$, where $y > x$. With this we refer to all the rationale numbers from x to y inclusive.
$x \rightarrow y$	Implication arrow, which we often for sketches. Here x indirectly refers to the orignal state, where we are, and y to state we need to reach with a series of action. It is indirect as we refer to the conditions valid in these states rather than specifically stating the states.

Partially Observable Markov Decision Process

t	time step index
$\mathcal{S}$	set of all states
$\mathcal{A}$	set of all actions
$\mathcal{P}$	transition function probability
$\mathcal{R}$	reward function with domain $\mathcal{S} \times \mathcal{A}$ and value in $\mathbb{R}$
T	transition function if system deterministic
γ	discount factor
Ω	set of all observations
$\mathcal{Z}$	observation function probability
$O(a, s')$	observation function, which maps an action a and state s' to observation o and $s' = T(a, s)$
$p(s'\|s, a)$	probability of state s' given state s and action performed a

List of Figures

2.1 Atari 2600's *Adventure* (1979) initial game screen. One can see a yellow cube, the player, standing in front of the golden castle's gate. Also, the gate's key, the yellow key, is visible to the left of the castle. 5
2.2 Basic Features applied to *Adventure*, an Atari 2600 game, where a singular feature example of the feature set is shown in magenta. 6
2.3 An example of BPROS applied to *Adventure*, an Atari 2600 game, where two pixels' colors from different tiles are compared with each other. One pixel's color is 0, and the other pixel's color is 170. These pixels are shown in magenta, and their relative offset is shown in blue. The red grid-like lines running horizontally and vertically represent the tiles in which the screen is divided. 7
2.4 An example of BPROT applied to *Adventure* where two pixels' colors from different tiles across time $(t, t + 1)$ tile-offset are compared with each other. These pixels are shown in magenta. 7

6.1 Multiple images showing different stages of the Adventure game where the modified SIWR agent overcomes previous limitations. Picture A shows the last position reached by the standard SIWR agent and full image is shown in Figure A.7. Picture B illustrates the position after executing action 3 (Move Right) from Picture A. The whole image is shown in Figure A.8. Picture C depicts the position after executing action sequence 3,2 (Move Right, Move Up) from Picture A. The whole image is shown in Figure A.9. Picture D displays the position after executing action sequence 3,2,2 (Move Right, Move Up, Move Up) from Picture A, where the agent successfully navigates further using the modified SIWR. The whole image is shown in Figure A.10. 42

A.1 Reward-IW agent exhibited undirected exploration in Adventure. Note the absence of subgoal sequences and frequent room revisits. Above is the section till where the reward-IW could proceed successfully, but on its way there it neither killed any dragons nor collected any key items. 51

A.2 We did not include the real log file due to its size here. We uploaded it online and linked it here. One can see the undirected exploration of the reward-SIW agent exhibiting undirected exploration in Adventure. This link where the file is available is: https://drive.google.com/file/d/1msL7_IB3QzNtXJ5iYQCveKz-z6F_qH-2/view?usp=sharing and can also be accessed by clicking on the image. 53
A.3 Standard SIWR agent executed sequential subgoals in Adventure, which are defined by the sketch rules. Note the systematic item collection, dragon defeats, and maze navigation. The last section where the standard SIWR fails is illustrated above. 53
A.4 We did not include the real log file due to its size here. We uploaded it online and linked it here. One can see the subsequent exploration of the standard SIWR agent using the sketch rules. For the blue maze section, we had included an additional sketch rule, which was to navigate through its entirety to reach the black castle. We never removed this rule, even after observing that the standard SIWR failed to solve this part due to feature representation issues. Despite this, this sketch rule is still visible in the log file but is never fulfilled. This link where the file is available is: https://drive.google.com/file/d/1XB-MoobwjcbFCDSsJgoO2JdF_i-Mxg2b/view?usp=sharing and can also be accessed by clicking on the image. 54
A.5 Modified SIWR agent successfully completed the first major segment of Adventure after addressing feature representation challenges. The agent retrieves key items, defeats dragons, and navigates initial maze sections, highlighting the effectiveness of the proposed algorithmic adjustments. Above is the section till where the modified SIWR could proceed successfully. Further progress would require breaking down the remaining sketch rules into smaller subgoals, which we did not implement in this work. 55
A.6 We did not include the real log file due to its size here. We uploaded it online and linked it here. One can see the subsequent exploration of the modified SIWR agent using the sketch rules. For the blue maze section, we had included an additional sketch rule, which was to navigate through its entirety to reach the black castle. We never removed this rule, even after observing that the modified SIWR failed to solve this part due to feature representation issues. Despite this, this sketch rule is still visible in the log file but is never fulfilled. Moreover, one can also see the use of the skip function after the first iteration, which is why we skip so many sketch rules in the beginning. 56

A.7 This figure illustrates the location in the Adventure game, till where the normal SIWR agent could autonomously navigate using the provided sketch rules. The agent successfully collects key items, defeats dragons, and navigates through initial maze sections up to this point. Beyond this location, the agent encounters challenges in the blue maze section due to feature representation issues, preventing further autonomous progress. However, the fixed SIWR agent, with the proposed modifications, can continue from this point onward. Even though, the key is not highlighted in this figure, the algorithm is aware that the cube is carrying the black key, as it has already picked it up in the previous steps. . . . 57

A.8 From the original location, where the normal SIWR agent stopped, we set the agent manually to this breaking point in the blue maze. One arrives at this position after executing Action 3 (Move Right) from the previous position. From here, using the modified SIWR, the agent could successfully navigate through the remaining maze sections that we tested. One can see that the agent is exactly in the middle of 4 tiles, so if the agent moved in any direction it would still be in the middle of 4 tiles. Despite them being different, BPROST could treat them all identically. Thus, when SIWR looked at moving in any direction, it saw no difference in the feature representation of the resulting state (between the resulting states), which maybe a reason why the normal SIWR failed to proceed from here. Even though, the key is not highlighted in this figure, the algorithm is aware that the cube is carrying the black key, as it has already picked it up in the previous steps. 58

A.9 From the original location, where the normal SIWR agent stopped, we set the agent manually to this breaking point in the blue maze. This position is arrived at after executing action sequence 3,2 (Move Right, Move Up) from the previous position, where normal SIWR stopped. One can see that the agent again is in middle of 4 tiles, which is why it is not novel for BPROST features. From here, using the modified SIWR, the agent could successfully navigate through the remaining maze sections that we tested. 59

A.10 This is the location in the blue maze that the agent reaches after executing action sequence 3,2,2 (Move Right, Move Up, Move Up) from the previous breaking point. From here, using the modified SIWR, the agent could successfully navigate through the remaining maze sections that we tested. Even the normal SIWR found the following path from this point onward, as the feature representation issues were resolved by then. Nonetheless, the SIWR could not reach this point autonomously without the modifications. Even though, the key is not highlighted in this figure, the algorithm is aware that the cube is carrying the black key, as it has already picked it up in the previous steps. 60

List of Tables

A.1 Sketch Rules for **Adventure** game, where all have width $w = 1$. Here we have left out rules 12 to 21, which all involve particular position-based navigation that uses exact coordinate constraints to direct travel across the layout of maze rooms 6, 7, 8, 9, and 10. We replaced this series of rules with a single exemplary sketch rule that illustrates the intended navigation within the maze. Hereby, room X and room Y placeholders represent specific rooms within the blue maze (6,7,8,9,10), while position a and position b denote exact valid areas within those rooms. Also, X and Y do not have to be different rooms; the rule can also represent movement within the same room from one position to another. However, in implementation, all the rules from 12 to 21 are still present. So far, we have achieved that the agent can complete up to rule 18 successfully. 52

List of References

[1] *ALE Documentation*. [Online]. Available: https://ale.farama.org/.

[2] K. J. Åström, "Optimal control of markov processes with incomplete state information i," *Journal of mathematical analysis and applications*, vol. 10, pp. 174–205, 1965.

[3] *AtariAge - Atari 2600 Manuals (HTML) - Adventure (Atari)*. [Online]. Available: https://atariage.com/manual_html_page.php?SoftwareLabelID=1.

[4] B. Ayton and M. Asai, *Is policy learning overrated?: Width-based planning and active learning for atari*, 2022. arXiv: 2109.15310 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2109.15310.

[5] B. Ayton and M. Asai, *Is policy learning overrated?: Width-based planning and active learning for atari*, 2022. arXiv: 2109.15310 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2109.15310.

[6] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, Z. D. Guo, and C. Blundell, "Agent57: Outperforming the atari human benchmark," in *International conference on machine learning*, PMLR, 2020, pp. 507–517.

[7] W. Bandres, B. Bonet, and H. Geffner, "Planning with pixels in (almost) real time," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[8] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of artificial intelligence research*, vol. 47, pp. 253–279, 2013.

[9] J. Blüml, C. Derstroff, B. Gregori, E. Dillies, Q. Delfosse, and K. Kersting, *Deep reinforcement learning via object-centric attention*, 2025. arXiv: 2504.03024 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2504.03024.

[10] B. Bonet and H. Geffner, "General policies, representations, and planning width," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 11 764–11 773.

[11] M. Dalmau-Moreno, N. García, V. Gómez, and H. Geffner, *Combined task and motion planning via sketch decompositions (extended version with supplementary material)*, 2024. arXiv: 2403.16277 [cs.RO]. [Online]. Available: https://arxiv.org/abs/2403.16277.

[12] A. Dittadi, F. K. Drachmann, and T. Bolander, "Planning from pixels in atari with learned symbolic representations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, 2021, pp. 4941–4949.

[13] D. Drexler, J. Seipp, and H. Geffner, "Expressing and exploiting the common subgoal structure of classical planning domains using sketches: Extended version," *arXiv preprint arXiv:2105.04250*, 2021.

[14] H. Geffner and B. Bonet, *A concise introduction to models and methods for automated planning*. Morgan & Claypool Publishers, 2013.

[15] I.-A. Hosu and T. Rebedea, *Playing atari games with deep reinforcement learning and human checkpoint replay*, 2016. arXiv: 1607.05077 [cs.AI]. [Online]. Available: https://arxiv.org/abs/1607.05077.

[16] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial intelligence*, vol. 101, no. 1-2, pp. 99–134, 1998.

[17] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*, Springer, 2006, pp. 282–293.

[18] N. Lipovetzky and H. Geffner, "Width and serialization of classical planning problems," in *ECAI 2012*, IOS Press, 2012, pp. 540–545.

[19] N. Lipovetzky, M. Ramirez, and H. Geffner, "Classical planning with simulators: Results on the atari video games," in *Proc. IJCAI*, 2015.

[20] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling, "Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents," *Journal of Artificial Intelligence Research*, vol. 61, pp. 523–562, 2018.

[21] J. Martin-Saquet, A. Dorise, E. Villain, S. Sanchez, and D. Panzoli, "Sequential decision-making in atari 2600 games: Comparing temporal features," in *2024 Artificial Intelligence Revolutions (AIR)*, IEEE, 2024, pp. 117–123.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1312.5602.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[24] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[25] R. D. Smallwood and E. J. Sondik, "The optimal control of partially observable markov processes over a finite horizon," *Operations research*, vol. 21, no. 5, pp. 1071–1088, 1973.

[26] C. T. Tan and H.-l. Cheng, “Implant: An integrated mdp and pomdp learning agent for adaptive games,” in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 5, 2009, pp. 94–99.