Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

AI Planning for Poker Player

Mike Ashi

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Marco Aiello

Supervisor: Ebaa Alnazer, M.Sc.

Commenced: June 15, 2022

Completed: December 15, 2022

Abstract

Throughout the history of computer science, games have always been a major driving factor in the development of the artificial intelligence field. The pioneers of computer science have identified the importance of games like chess, checkers, and other similar games and have spent years working on solving them. These games offered a great research framework with well-defined rules, distinct objectives, and a means of measuring performance. However, it was also recognized that perfect information games do not reflect the decision-making process in real life, where agents must embrace uncertainty and often deal with misinformation and deception. The game of poker successfully models these aspects of decision-making and can be used to study the behavior of intelligent agents in real-life domains. Poker has recently regained the attention of researchers, leading to significant breakthroughs elevating AI agents to the level of professional players. Several approaches have been used to solve poker, such as knowledge-based agents, enhanced simulation systems, theoretic equilibrium solutions, and exploitative counter-strategies. Nevertheless, to the best of our knowledge, HTN planning techniques have never been used before. In this thesis, we will explore the use of risk-aware HTN planning for playing no limit Texas Hold'em, the most strategically demanding variation of poker, which will allow us to investigate the potential of this approach.

Kurzfassung

In der Geschichte der Computerwissenschaft waren Spiele immer ein wesentlicher Faktor bei der Weiterentwicklung des Bereichs der künstlichen Intelligenz. Die Pioniere der Informatik haben die besondere Wichtigkeit von Spielen wie Schach, Dame und anderen ähnlichen Spielen erkannt und jahrelang daran geforscht, sie zu verstehen. Diese Spiele boten einen hervorragenden Forschungsrahmen mit klar definierten Regeln, eindeutigen Zielen und einem geeigneten Messinstrument für die Leistung. Es wurde jedoch auch erkannt, dass perfekte Informationsspiele nicht den Entscheidungsprozess im wirklichen Leben widerspiegeln, wo Agenten Ungewissheit in Kauf nehmen und oft mit Fehlinformationen und Täuschung umgehen müssen. Das Pokerspiel modelliert erfolgreich diese Aspekte der Entscheidungsfindung und kann verwendet werden, um das Verhalten intelligenter Agenten in realen Domänen zu untersuchen. Poker hat in letzter Zeit die Aufmerksamkeit der Forscher zurückgewonnen, was zu bedeutenden Durchbrüchen geführt hat, die KI-Agenten auf das Niveau von Profispielern gehoben haben. Zur Lösung des Pokerspiels wurden verschiedene Ansätze verwendet, z. B. wissensbasierte Agenten, verbesserte Simulationssysteme, theoretische Gleichgewichtslösungen und exploitative Gegenstrategien. Unseres Wissens nach wurden jedoch noch nie HTN-Planungstechniken eingesetzt. In dieser Arbeit wird der Einsatz von risikobewusster HTN-Planung für no limit Texas Hold'em, der strategisch anspruchsvollsten Pokervariante, untersucht, um das Potenzial dieses Ansatzes zu erforschen.

Contents

1 Introduction **17**

2 Background **19**
2.1 Poker . . . 19
2.2 Automated Planning . . . 23
2.3 Hierarchical Task Networks . . . 26
2.4 Utility and Risk Awareness . . . 30

3 PokerShark **39**
3.1 The Planner . . . 39
3.2 Opponent Modeling . . . 44
3.3 Preflop Strategy . . . 46
3.4 Postflop Strategy . . . 51
3.5 Domain Model . . . 57
3.6 World State . . . 58
3.7 Goal State . . . 59

4 Evaluation **61**
4.1 Dynamic Risk Attitude . . . 61
4.2 Static Risk Attitude . . . 65
4.3 Self Play Experiments . . . 66

5 Related work **67**

6 Conclusion and Future Work **69**

Bibliography **71**

A Blocks World Domain **75**

B PokerShark Preconditions **77**

C Different Risk Attitudes Experiments **79**
C.1 Static Risk Attitude - Risk Seeking . . . 79
C.2 Static Risk Attitude - Risk Averse . . . 81
C.3 Static Risk Attitude - Risk Neutral . . . 83

List of Figures

2.1 A simple HTN task network for commuting to work. With T_p and T_c representing primitive and compound tasks, respectively, and M_1 and M_2 representing methods. 31
2.2 A simple example of hypothetical game. 32
2.3 Risk-neutral agents' decision-making 34
2.4 Risk-averse agents' decision-making 34
2.5 Risk-seeking agents' decision-making 35
2.6 Different risk-based utility functions 35

3.1 Multiple translation and read/write operations required. 40
3.2 FluidHTN with shared state. 42
3.3 PokerShark architecture. 42
3.4 Selects can choose one path, Sequences need to decompose all subtasks. 43
3.5 Adjustment matrix example. 45
3.6 Adjustment matrix progression. 46
3.7 Example of converting a guideline into a recommendation. 47
3.8 Tight opponent range 53
3.9 Utility Functions, Where α is the risk sensitivity 55
3.10 Decision Tree without adjusment 56
3.11 Adjustment Factors 57
3.12 Overview of PokerShark domain model. 58

4.1 Results of playing 100 games against the Bold player. 62
4.2 Results of playing 100 games against the Fish player. 63
4.3 Results of playing 100 games against the Honest player. 63
4.4 Results of playing 100 games against the Random player. 64
4.5 Results of around 1000 hands against human players. 64

C.1 Results of playing 100 games against the Bold player - With Risk Seeking attitude. 79
C.2 Results of playing 100 games against the Fish player - With Risk Seeking attitude. 79
C.3 Results of playing 100 games against the Honest player - With Risk Seeking attitude. 80
C.4 Results of playing 100 games against the Random player - With Risk Seeking attitude. 80
C.5 Results of playing 100 games against the Bold player - With Risk Averse attitude. 81
C.6 Results of playing 100 games against the Fish player - With Risk Averse attitude. 81
C.7 Results of playing 100 games against the Honest player - With Risk Averse attitude. 82
C.8 Results of playing 100 games against the Random player - With Risk Averse attitude. 82
C.9 Results of playing 100 games against the Bold player - With Risk Neutral attitude. 83
C.10 Results of playing 100 games against the Fish player - With Risk Neutral attitude. 83
C.11 Results of playing 100 games against the Honest player - With Risk Neutral attitude. 84
C.12 Results of playing 100 games against the Random player - With Risk Neutral attitude. 84

List of Tables

2.1 Hand Ranking in No-limit Texas Hold'em 22
2.2 The stack task from the block-world domain in different formats. 29

3.1 Planner comparison. 41
3.2 Example of the hand groups based on potential strength defined by [Skl03] 47
3.3 Simulation results hash table. 48
3.4 Number of unique and distinct hands 52
3.5 Expected Utility Constructs 55

4.1 Results of playing 100 games using the different static risk attitudes. 65
4.2 Average WPH for each risk attitude. 66
4.3 Results of self play with different risk attitudes; WPH of player at start of row . . 66

List of Listings

2.1 PDDL domain definition example adopted from [HLM+19]. 25
2.2 PDDL problem definition example 25
2.3 The stack task from the block-world domain in GTPyhop. 30
2.4 Example of using risk attitude without utility 36
3.1 RandomSelector definition. 43
3.2 Snippet of PokerShark domain builder. 43
3.3 Fold if the pocket is 23. 50
3.4 Decision object that folds 90% of the time 50
A.1 Blocks world domain definiton. 75
A.2 Blocks world problem definiton. 75

List of Algorithms

3.1 PokerShark Risk Attitude 48
3.2 Hand Strength estimation against one opponent 52
3.3 Find the task with the highest utility 56

Acronyms

CFR Counterfactual Regret Minimization. 18

HDDL Hierarchical Domain Definition Language. 29

HTN Hierarchical Task Network. 18

PDDL planning domain definition languages. 24

SHOP Simple Hierarchical Ordered Planner. 39

1 Introduction

The fascination with the idea of artificial intelligence and sentient robots dates back at least as far as ancient Greek mythology, which tells the story of Hephaestus, the Greek god of smithing, and his creation of Talos, a giant constructed of bronze to act as a guardian for the island of Crete. This fascination could be observed throughout history, where the concept of thinking machines has captured the imagination of mathematicians, philosophers, and scientists. These brilliant minds pushed, throughout the years, the boundaries of what was possible, bringing artificial intelligence into the modern day, where it can drive a vehicle, write code, compose symphonies, and create astounding works of art.

Games and puzzles played an extremely significant part in this advancement, as the pioneers of the field recognized them as a great platform on which to conduct research. Alan Turing, for example, spent many years working on a chess program called Turochamp [Tur53]. To date, Turochamp stands as the oldest known computer game to have begun development. However, its algorithm was too complex for the primitive computers of the day, and the program was never completed. Turing also considered other games, such as go and poker, but his research on these games was never published. Claude Shannon, known as the father of information theory, also worked on chess and published the first paper on the topic in 1949. Shannon opened his paper by stating how solving chess will pave the way to solving more complicated problems and enable the development of machines capable of translating different languages, making strategic decisions in military operations, and orchestrating melodies [Sha50]. Shannon was able to estimate the computational complexity of chess and understood that the game could not be solved using brute-force search approaches. In 1951 Christopher Strachey started working on a program that could play the game of checkers. By the summer of 1952, Strachey ran the earliest successful AI program capable of playing a complete game of checkers at a reasonable speed.

These games provided a fertile environment for the growth of the field of artificial intelligence, as they have characteristics that make them suitable for acquiring experiences that can be transferred to more valuable domains of the real world. However, not all games were treated in the same way, as games such as chess and checkers received a lot of attention due to the ease of achieving higher performance through brute-force search [BPSS98]. Games such as poker, on the other hand, which are more complex and require more sophisticated approaches, had only attracted limited attention [Bil06] until recently, when it was identified as a beneficial domain for AI research [BPSS98]. Traditional strategies like heuristic search and evaluation methods are inadequate to overcome the sheer complexity of the game [BDSS02]. Moreover, the properties of the game, such as imperfect information, stochastic outcomes, risk management, deception, and unreliable information, make it a very difficult computational problem to solve. Solving poker will provide important insights into dealing with deliberate misinformation and making intelligent decisions despite incomplete information [Bil06]. Several approaches have been used to solve poker, such as knowledge-based agents, enhanced simulation systems, theoretic equilibrium solutions, and exploitative counter-strategies [RW11].

More recently, the Counterfactual Regret Minimization (CFR) algorithm was utilized by bots such as DeepStack [MSB+17] and Pluribus [BS19] to achieve significant breakthroughs, elevating AI agents to the level of professional poker players.

Different strategies and methods have been utilized in these approaches, but to the best of our knowledge, HTN planners have never been used. Hierarchical Task Network (HTN) are structures utilized by AI planning systems that offer a high level of expressiveness [EHN94] and are widely implemented in real-world applications. It was previously [AGA22] demonstrated that HTN planning can become risk-aware. Poker is a game centered around making decisions under uncertainty, and risk is an essential element in the game. HTN planning techniques provide a wide range of features, such as the capability of encoding expert knowledge, risk awareness and utility, and the ability to introduce abstraction levels, making it easy to break the problem into smaller segments. These features make HTN planning a potentially great framework for developing a risk-aware agent that can play poker competently. In this thesis, we will explore the use of HTN planning for playing no limit Texas Hold'em as it is considered to be the most strategically demanding variation of poker, which will allow us to investigate the potential of this approach.

The remainder of this thesis is organized as follows. In the first chapter, we introduce the background information on the different topics related to this study, including poker, automated planning, and risk awareness. In the following chapter, we discuss the specifics of our implementation, including a review of HTN planners and the approaches we utilized to model the opponents. After that, we break down the playing strategies of the bot. Finally, in the last chapter, we provide an evaluation of the approach and a conclusion.

2 Background

This thesis touches on a wide range of subjects, many of which have been the focus of several publications and bodies of research. Given the scope of this work, it will not be possible for us to go into considerable depth about any of the topics. However, to establish common ground, we have devoted this chapter to provide a high-level overview of the most prominent ones.

2.1 Poker

Poker is an umbrella term used to describe a family of card games with a similar structure but different rules and objectives. The origins of poker are shrouded in mystery; however, there is a consensus [Roy21] that poker in its modern form evolved from the French game Poque in the early 18th century [Wil12]. The game was popularized by the World Series of Poker[1], which was first held in 1970. At the beginning of the 21st century, poker's popularity soared mostly due to the introduction of television coverage, which transformed poker into a spectator sport. Online poker also played a significant role in making the game more popular because the game was suddenly readily accessible. As a result, poker is now considered a staple of casinos and card rooms across the globe.

Numerous factors make poker a fascinating game. It is a game that is easy to learn but hard to master. The most complicated poker variations are difficult due to the strategies and playing dynamics involved, not the rules. Good players must master a broad range of skills, including reading other players, calculating odds and probabilities, and utilizing mathematics to make profitable decisions. The element of luck and the randomness of outcomes make the game full of surprises and excitement. A single card may completely alter the outcome of a hand. In contrast to other gambling games, such as blackjack, baccarat, and roulette, that are designed to give the house an edge, poker is a game where players compete against one another. Although luck plays a big role in poker, its effects are only short-term, and better players will always beat weaker players over the long run.

2.1.1 The Game

A game of poker consists of a series of rounds (also called hands or deals). Depending on the variant of the game, a round's structure may vary significantly, but typically a round consists of multiple stages. Generally, each stage begins with introducing new cards; after that, players have the opportunity to place bets. Players bet when they believe that their hand is superior to the hands of their opponents. The bettor must contribute their bet amount to the pot when a bet is placed. If they wish to continue playing, the highest bet has to be matched (*called*) or *raised* by other players.

[1] https://en.wikipedia.org/wiki/World_Series_of_Poker

When a player is unwilling to match the highest bet, they can *fold*, but that will cost them all the money they have already contributed to the pot. If all players except one have folded, the remaining player wins the pot automatically. Otherwise, the round eventually reaches the *showdown*, where players reveal their hands, and the best hand wins the pot. How player hands are constructed and ranked is specific to each game variation.

2.1.2 Game Dynamics

Many elements of poker make it a suitable environment for a wide range of strategies and tactics. In this section, we will explore some of these elements.

Forced bets

Winning the pot is the main incentive for players to accept the risks involved in the game. A larger pot will motivate players to take higher risks and make them more likely to bet on weaker hands. Without this incentive, the game would be too slow and boring, with players only betting when they hold stronger hands. Poker utilizes forced bets to ensure players always have the incentive to act. There are different types of forced bets, but we will discuss only two: the *ante* and the *blinds*. *Ante* is a fixed amount that each player must contribute to the pot before the beginning of each round. In contrast, *Blinds* are mandatory bets that only certain players must place. The player to the dealer's left must place the *small blind*, and the player to his left must post the *big blind*, typically double the amount of the small blind. The size of the blinds can increase as the game progresses to account for the fact that the number of participating players will decrease as the game approaches its final rounds. The position of the dealer is rotated around the table to keep the game fair.

Raise, Call, or Fold

Usually, players fold when they deem it too risky or expensive to continue, so they try to cut their losses by folding, as folding will only get more expensive in the later stages of the round. When players are not confident with their cards but believe there is a potential for their hand to get better later in the round, they call the last bet by contributing a matching amount to the pot. If there were no previous bets, then the player can *check*, which is a free call. In games where blinds are collected, no checks can be placed in the first round stage because blinds are considered bets. A raise can happen when players believe their hand is strong enough to win the pot. After a raise, all players who have not folded yet have to decide whether they want to fold and lose all the money they have already contributed to the pot or continue by calling or *re-raising*, making the pot bigger. If a player is already in the pot and does not have enough money to match the last bet, they can go *all-in*, which means they will contribute all the money they have left to the pot.

These three actions offer players endless possibilities for employing tactics and strategies. In poker, information is invaluable since it enables players to make better decisions. Players always leak information about their hands to their opponents with every action they take, which creates a dynamic where players must balance the risk of revealing too much information while trying to maximize their potential win. It also allows players to *bluff*, which is a strategy that involves deceiving opponents by making them believe that a player has a better hand than they actually do.

When a player has a fairly strong hand but does not wish to be challenged by other players, they may make a high raise to scare the opponents. Sometimes a player will check even though they have a very strong hand because they do not want to discourage other players from entering the pot. When holding a strong hand, a combination of calls and small raises could also entice opponents into making high raises. Even folds can be used to deceive opponents. For example, a player may often fold, even with stronger hands, to make opponents believe that he only participates when he has a strong hand, making his raises more credible and scary to challenge.

Some game variations limit the minimum and maximum raising amounts or the number of raises that can be made in a round. These restrictions prevent players from raising too much and help maintain the game at a reasonable pace.

Position

Player position refers to the order in which players are prompted to act. A player *has position* over opponents who act before him and is *out of position* to opponents who act after him. In most poker variations, the player's position is a crucial factor. Being in a late position allows players to observe how the opponents have acted. Additionally, as players fold, the players in position will have fewer players challenging their hands. Players will always fold a given set of cards in an early position, but they will have no problem calling or raising the same cards from a late position. Early positions also have advantages; for example, being out of position enables a player to raise the stakes for the players after him. Each successive round rotates positions around the table to keep the game fair.

Optimality vs. Exploitability

The characteristics that make poker an interesting game also increase the game's complexity. To combat this complexity, good players utilize two strategies in their play. The first strategy is based on Nash equilibrium, which is a concept in game theory that describes a situation where a player cannot increase their potential gain by changing their strategy. Such a strategy, called optimal strategy, will always consider the worst case and choose a course of action that minimizes loss. A player following an optimal strategy cannot find a better strategy when playing against another optimal player. The term "optimal" is quite misleading here, as an optimal strategy does not maximize potential gain but minimizes potential loss. An optimal strategy does not seek to defeat other players but to defend against their best possible course of action. This means an optimal strategy is not necessarily the best strategy, especially in the context of poker, where players greatly deviate from the optimal strategy. Another major downside is being quite predictable. The static nature of optimal strategies allows opponents who are not constrained by optimality to explore weaknesses without fear of being punished.

While optimal strategy is indifferent to the opponent's actions, exploitative strategy focuses on identifying and capitalizing on the opponent's tendencies. Exploitative strategy is based on the premise that if the opponent's strategy is known or can be predicted, one should choose the action that maximizes gain.

Let us consider the game of rock-paper-scissors to showcase the fundamental difference between the two strategies. An optimal player would choose his actions randomly, disregarding the opponent's past actions. Even if the opponent had just played rock eight consecutive times, an optimal player

would not prefer paper over the other two alternatives. A maximizer player, on the other hand, will recognize the opponent's tendency to play rock and choose paper as the next action. It is important to note that deviating from the optimal strategy will always introduce risk and create weaknesses that can be exploited. For example, the opponent may have been setting a trap with his previous moves. Therefore the maximizing player is taking a risk by choosing paper. Predicting the opponent's next move is not easy, but analyzing the opponent's playing style can give a good indication of their preferences and risk tolerance.

Good poker players utilize the two strategies in tandem, switching between them seamlessly.

Texas Hold'em

In this work, we are mainly interested in "No-limit Texas Hold'em", one of the most popular poker variations. Texas Hold'em is regarded as one of the more advanced poker variations from a strategy point of view, evidenced by the experts' divergent opinions on how to play a specific hand [Skl03]. Professional poker players favor Texas Hold'em because it is the variant in which they can win the highest amount of money with the lowest level of risk since luck has less of an impact on the outcome of the game compared to other poker variants [Mal04].

Each round begins with the *Preflop*, during which the small and big blinds are collected. Next, each participant is dealt two *Pocket* cards face down. In the next stage, the *Flop*, three face-up *community cards* are dealt to the *board*. All players can use community cards to build a 5-card hand. The terms "community cards" and "board" can be used interchangeably. In the following stages, *Turn* and *River*, two additional face-up cards are added to the board one at a time. After betting on the River, the *Showdown* begins, and the player with the best hand wins the pot. In the case of a tie, the pot is split. Table 2.1 shows the hand ranking in No-limit Texas Hold'em.

Rank	**Description**	**Example**
Royal Flush	Five high cards in sequence, all of the same suit	A♦,K♦,Q♦,J♦,T♦
Straight Flush	Five cards in sequence, all of the same suit	7♦,6♦,5♦,4♦,3♦
Four of a Kind	Four cards of the same rank	8♠,8♥,8♦,8♣,2♠
Full House	Three of a kind and one paire	3♥,3♦,3♣,2♥,2♦
Flush	Five cards of the same suit	A♦,T♦,6♦,4♦,2♦
Straight	Five cards in sequence, not of the same suit	9♥,8♣,7♣,6♦,5♥
Three of a Kind	Three cards of the same rank	8♣,8♠,8♦,4♦,2♣
Two Pair	Two one pairs	T♠,T♣,8♣,8♣,4♥
One Pair	Two cards of the same rank	T♠,T♣,9♦,6♠,2♥
High Card	None of the above	T♠,7♣,5♦,3♣,2♠

Table 2.1: Hand Ranking in No-limit Texas Hold'em

2.2 Automated Planning

The term **Planning** is commonly used to describe the act of scheduling. However, planning can mean much more than just scheduling; for example, the Cambridge Dictionary defines it as the act of deciding how to do something[2], which is a better definition to use in the context of intelligent agents that use planning to decide how to achieve their goals. The notion of intelligent agents entails the ability to observe and interact with the environment in a deliberate manner to achieve some objective. However, to act deliberately, the agent has to have a clear objective and be able to predict the effect of its actions in a given state of the environment. Furthermore, some objectives are inherently complex, and to accomplish them, the agent has to decide which actions to perform and in which order to execute them. This reasoning process of choosing actions and organizing them is called planning [GNT06]. Automated planning can be described as the study of computational models and methods of creating, analyzing, managing, and executing plans [HLM+19].

2.2.1 Classical Planning

As a means of introduction, we will explore classical planning, which refers to the most basic form of automated planning. In order to avoid the complexity of the real world, classical planning deals with an abstract model of the environment, imposing a number of restrictive assumptions:

- **Finite, observable, and static:** The environment has only a finite number of states and actions. The current state of the environment is known to the agent, and it can not be changed without an action initiated by the agent.
- **Deterministic, no uncertainty:** The agent can predict with certainty the changes in the current state if a given action is performed.
- **Implicit time:** No explicit model of time. There is only a linear sequence of instantaneous states.

We can now introduce the formal definition of classical planning. The formal structures we use are adapted from formal definitions described in [GNT06][GGA14][Aln19]:

Definition 2.2.1 (Predicate)
A predicate p is defined as follows: $p = \langle symbol(p), terms(p) \rangle$, where:
- $symbol(p)$ is the predicate symbol.
- $terms(p)$ is a set of terms such that $terms(p) = \langle \tau_1, \tau_2, \dots, \tau_n \rangle$.
- A term is either:
  - A constant $co \in CO$, where CO is a finite set of constants.
  - A variable $v \in \mathcal{V}$, where $\mathcal{V}$ is an infinite set of variables.

Predicates can be used to describe the objects of the environment and their relations.

Definition 2.2.2 (Ground Predicate)
A ground predicate is a predicate where all the terms are constants.

[2] https://dictionary.cambridge.org/dictionary/english/planning

Definition 2.2.3 (State)
A state s is defined as a set of ground predicates: $s = \{p_1, p_2, \dots, p_n\}$ while describing a state, we adopt the *closed world assumption*[3].

Definition 2.2.4 (Action)
An Action a is defined as follows: $a = \langle pre(a), eff(a) \rangle$, where:

- $pre(a)$ is the precondition that has to hold to perform the action.
- $eff(a)$ is the effect of the action on the state.

Definition 2.2.5 (Applicable Action)
Given a state s and an action a, we say that a is applicable in s if and only if:

$$\forall p_i \in pre(a) : p_i \in s \text{ and } \forall \bar{p}_i \notin s$$

Applying action a in state s will result in a new state $\acute{s}$ in which some predicates are deleted (i.e., are given the boolean value false) while others are added. The set of deleted and added predicates are denoted by $eff^-(a), eff^+(a)$, respectively. We denote the application of an action a to a state s as:

$$\gamma(s, a) := (s \cup eff(a)^+) \setminus eff(a)^- = \acute{s}$$

For a sequence of actions $\mathbf{A} = \langle a_1, \dots, a_n \rangle$ we recursively define the application of $\mathbf{A}$ to s as :

- $\hat{\gamma}(s, \langle\rangle) := s$
- $\hat{\gamma}(s, \langle a_1, \dots, a_n \rangle) := \hat{\gamma}(\gamma(s, a_1), \langle a_2, \dots, a_n \rangle)$

given that γ is well defined for each a_i.

Planning domain

For an intelligent agent to be able to conduct a reasoning process, it should have a model of the environment. This model does not have to reflect each element of the environment, but it should be a good and accurate approximation.

Definition 2.2.6 (Planning Domain)
A planning domain is a tuple $\Sigma = (\mathcal{P}, \mathcal{A})$ where:

- $\mathcal{P}$ is a finite set of predicates.
- $\mathcal{A}$ is a finite set of actions.
- The finite set $\mathcal{S}$, which contains all possible states, can be inferred from $\mathcal{P}$ and $\mathcal{A}$.

In practice, planning domain definition languages (PDDL) are used to describe the domain [AHK+98]. There are many different versions and variations of PDDL, each with a certain set of capabilities and limitations. Listing 2.1 shows a simple example from [HLM+19], where a description of a simple switch is provided with two actions:

[3] The state description only includes predicates that evaluate to true

Listing 2.1 PDDL domain definition example adopted from [HLM+19].

```
(define (domain switch)
    (:requirements :strips)
    (:predicates (switch_is_on)(switch_is_off))
    (:action switch_on
        :precondition (switch_is_off)
        :effect (and (switch_is_on) (not (switch_is_off))))
    (:action switch_off
        :precondition (switch_is_on)
        :effect (and (switch_is_off) (not (switch_is_on)))))
```

Planning problem

A planning domain is not a description of the current state of the environment but rather a model of all the possible states of the environment. Nevertheless, this is insufficient for the agent. The agent must be aware of the current state of the environment and have a clear definition of what to achieve.

Definition 2.2.7 (Planning Problem)
A planning problem is a triple $\pi = (\Sigma, s_I, s_G)$, where :

- Σ is a problem domain.
- s_I is the initial state.
- s_G is the goal state.

Just like planning domains, planning problems are described in practice using PDDL. Listing 2.2 shows a problem definition for the previously defined domain:

Listing 2.2 PDDL problem definition example

```
(define (problem turn_it_off)
    (:domain switch)
    (:init (switch_is_on))
    (:goal(switch_is_off)))
```

Appendix A contains a more complex example of a planning domain and problem.

Definition 2.2.8 (Valid Plan)
Given a planning domain Σ and a planning problem π, a plan $\mathbf{P} = \langle a_1, a_2, \ldots, a_n \rangle$ is a valid plan for π if:

$$\hat{\gamma}(s_I, \mathbf{P}) = s_n \in s_G$$

Planners

A planner takes a planning problem and formulates a sequence of actions to transform the current state of the environment into a state that satisfies the objective. Various planners employ different problem-solving strategies. For example, some planners use theorem proving [KS92] to generate a plan, whereas others use state-space search. Planning is computationally hard [ENS95], and sometimes it is not enough to find a plan that achieves the goal, but the planner has to consider also a set of metrics that make plan *a* better/worse than plan *b*, which causes the planning process to be more and more complicated.

2.3 Hierarchical Task Networks

Classical planning has many shortcomings; for example, it views plans as a linear sequence of actions, but it was clear from the early stages of research [SE75] that planners can solve some problems much better when plans are not constrained by the limitations of linearity. If a planner is capable of reasoning about nonlinear plans, it can delay committing to a specific order of actions until sufficient information is gathered. This can alleviate the need for an exhaustive search of all possible plan orderings. Another big problem with classical planners is the fact that they are easily overwhelmed by complex domains [Wil89]. The lack of abstraction levels makes it hard to consider meta-level reasoning. While it is feasible to integrate expert knowledge into the search algorithms, it is not easy, and steering the trajectory of execution is also not a simple task.

HTN planning tries to overcome some of those shortcomings by introducing more abstraction layers and using expert knowledge to guide the planning process. HTN is based on the premise that goal states as objectives are usually unnatural, and composing abstract actions from smaller concrete sub-actions is much more intuitive. Furthermore, having the domain knowledge organized hierarchically will allow the planner to create plans using action reduction, meaning the most important conditions will be considered first, and the details will be considered later in the planning process [Yan90].

The abstraction that HTN introduces is implemented using two types of constructs [HBBB21]: *primitive* and *compound tasks*. These constructs are combined to form partially ordered sets called *task networks*. The primitive tasks are comparable to actions in classical planning; they can be executed if the current state of the environment adheres to some condition and they have some effect on the environment. The compound tasks represent more complex actions that cannot be executed in one step and must be decomposed using some *decomposition method*. The decomposition methods are part of the planning domain, and each method can decompose a specific compound task into a task network. Multiple methods can decompose one compound task; in contrast, there is a one-to-one mapping between operators and primitive tasks.

2.3.1 Formal Definitions

Throughout the years, numerous attempts have been made to establish a standard formalism for HTN planning [BAH19]. Some of these approaches were actually unique, while others were essentially comparable. The purpose of this section is not to provide a comprehensive formal framework but rather to provide a formal background that will assist the reader in comprehending HTN's underlying structure. The following formal structures are derived from [GGA14] [Aln19].

Definition 2.3.1 (Primitive task)
A primitive task $t_p \in T_p$ is a pair $t_p =< symbol(t_p), terms(t_p) >$, where:

- T_p is a finite set of primitive tasks.
- $symbol(t_p)$ is a primitive task symbol.
- $terms(t_p)$ is a set of terms, such that $terms(t_p) = \langle \tau_1, \tau_2, \ldots, \tau_n \rangle$.

Definition 2.3.2 (Operator)
An operator $o \in O$ is a triple $o =< p(o), pre(o), eff(o) >$, where:

- O is a finite set of operators.
- $p(o)$ is a primitive task.
- $pre(o)$ and $eff(o)$ are precondition and effect.

It is important to remember that each primitive task must have one and only one operator, but multiple primitive tasks can utilize a single operator. An operator is quite comparable to an action from classical planning. We say an operator o is applicable in a state s if and only if:

$$\forall p_i \in pre(o) : p_i \in s \text{ and } \forall \bar{p_i} \notin s$$

Definition 2.3.3 (Compound task)
A compound task $t_c \in T_c$ is a pair $t_c =< symbol(t_c), terms(t_c) >$, where:

- T_c is a finite set of compound tasks.
- $symbol(t_c)$ is a compound task symbol.
- $terms(t_c)$ is a set of terms, such that $terms(t_c) = \langle \tau_1, \tau_2, \ldots, \tau_n \rangle$.

In practice, it is often beneficial to extend this definition to include a set of preconditions $pre(t_c)$ that must hold in the current world state for the planner to even consider this compound task for decomposition. We have found this very useful for our implementation, especially for compound tasks that can be decomposed by a large number of methods. Although it does not provide any benefits semantically nor significant performance improvements, it allowed us to quickly adjust the preconditions of many methods from one place.

Definition 2.3.4 (Task network)
A task network tn is a pair $< T, <>$, where:

- T is a finite set of tasks.
- $<$ is a partial order on T.

Definition 2.3.5 (Method)
A method $m \in M$ is a triple $m =< c(m), pre(m), tn(m) >$, where:

- M is a set of methods.
- $c(m)$ is a compound task.
- $pre(m)$ is a precondition.

- $tn(m)$ is a task network.

A method m is applicable in a state s if and only if:

$$\forall p_i \in pre(m) : p_i \in s \text{ and } \forall \bar{p}_i \notin s$$

Definition 2.3.6 (Decomposition)
Given a task network $tn =< T, <>$, and an applicable method m for a compound task $t = c(m)$, we say that m decomposes tn into $\acute{tn}$ with:

- $\acute{tn} = ((T \setminus \{t\}) \cup T_m, \prec \cup \prec_m \cup \prec_D)$ where
- $\prec_D = \{(t_1, t_2) \in T \times T_m | (t_1, t) \in <\} \cup \{(t_1, t_2) \in T_m \times T | (t, t_2) \in <\}$

Definition 2.3.7 (HTN planning domain)
A HTN planning domain is a pair $\Sigma = (O, M)$ where:

- O is a set of operators.
- M is a set of methods.

Definition 2.3.8 (HTN planning problem)
A HTN planning problem is a triple $\pi = (\Sigma, s_0, tn_0)$, where:

- Σ is the planning domain.
- s_0 is the initial state.
- tn_0 is the initial task network.

Planners

There are numerous strategies implemented by different HTN planners to solve an HTN planning problem. HTN planners can be classified based on the search space in which the planner operates. This work is not intended to provide a full overview of HTN planners, as has been done in [BAH19] and [GGA14]. Therefore, we will only provide an oversimplified overview of how HTN planners function. HTN planners are fundamentally different from classical planners in that they do not pursue some goal state but rather try to simplify a hierarchical network of tasks into a linear sequence of “actions”. Some HTN planners perform a decomposition-based search, in which applicable methods repeatedly decompose the task network until no more compound tasks are left. The result would be a task network that consists of only primitive tasks. Others perform a progression-based search, where the planner decomposes compound tasks and executes primitive tasks when applicable until the task network is empty. It is important to know that decomposing a compound task changes the structure of the task network but not the state. On the other hand, executing a primitive task changes the state of the environment but not the structure of the task network. When performing a progression-based search, the planner must keep track of the executed primitive tasks; otherwise, it is necessary to retrace the path the planner took from the initial task network to the empty one.

Description Languages

Using a description language, HTN planning domains and problems can be described just as they are in classical planning. Different planners use different description languages. Nevertheless, there has been an effort to create a standard description language by extending PDDL to support HTN constructs. Recently a new description language based on PDDL called Hierarchical Domain Definition Language (HDDL) has been adopted as the standard input language for the track on hierarchical planning at the International Planning Competition. Regardless, planners are typically reluctant to adopt new formats; currently, most planners only support a custom input format. Table 2.2 shows the stack task from the block-world domain (described in detail in appendix A) in different description languages.

<table>
<tr><th>Planner</th><th>Description</th><th>Planner</th><th>Description</th></tr>
<tr><td>HDDL</td><td>

```
(:action stack
  :parameters  (?a ?b)
  :task (stack ?a ?b)
  :precondition (
                and(clear ?b)
                (holding ?a))
  :effect (and (arm-empty)
               (clear ?a) (on ?a ?b)
               (not (clear ?b))
               (not (holding ?a))))
```

</td><td>SHOP</td><td>

```
(:operator (!stack ?a ?b)
  (
    (holding ?a)
    (clear ?b)
  )
  (
    (on ?a ?b)
    (clear ?a)
  )
)
```

</td></tr>
<tr><td>CHIMP</td><td>

```
(:operator
        (Head stack(?a,?b))
        (Pre p0 holding(?a))
        (Pre p1 clear(?b))
        (Del p0)
        (Del p1)
        (Add e1 on(?a,?b))
        (Add e1 clear(?a))
)
```

</td><td>TF</td><td>

```
actschema stack
  pattern << put $*a on $*b>>
  conditions
  holds <<holding $*a >> at self
  holds <<clear $*b >> at self
  effects
  + << on $*a $*b >>
  + << clear $*a >>
  - <<holding $*a >>
  - <<clear $*b >>
end
```

</td></tr>
</table>

Table 2.2: The stack task from the block-world domain in different formats.

There are also planners that use a programming language to describe the domain and problem rather than a specialized description language. This approach has the advantage that the planner can be directly integrated into the system without needing a translation layer between the system and the planner. RAE[4], GTPyhop[5] and FluidHTN[6] are all great examples of this approach. Listing 2.3 shows the stack task from the block-world domain defined as a python function for the GTPyhop planner.

[4] https://github.com/patras91/rae_release
[5] https://github.com/dananau/GTPyhop
[6] https://github.com/ptrefall/fluid-hierarchical-task-network

Listing 2.3 The stack task from the block-world domain in GTPyhop.

```
def stack(s,b1,b2):
    if s.pos[b1] == 'hand' and s.clear[b2] == True:
        s.pos[b1] = b2
        s.clear[b1] = True
        s.holding['hand'] = False
        s.clear[b2] = False
        return s
```

2.4 Utility and Risk Awareness

Classical HTN planners have several advantages over classical planners, but they lack any concept of *utility*. Classical HTN planners are satisfied when a solution is found, but in the real world, this is rarely sufficient. In the real world, we are typically interested in finding an optimal solution in some sense. For instance, we may only be interested in solutions that are safe, fast, or inexpensive. This section will begin with an overview of the concept of utility, followed by a discussion of how utility can be used to guide the planning process. In subsection 2.4.3, we introduce expected utility, and the section concludes with a discussion of risk attitudes.

2.4.1 Utility

The core concept of utility revolves around modeling the outcomes of an action or situation based on a set of criteria or metrics, which allows us to rank actions or outcomes and choose the most preferable option. Mathematically, utility can be represented via a set of binary relations over a set of actions or situations $\mathbf{\Delta}$ [Fis68].

Definition 2.4.1 (Preference-indifference relation $\preceq$)
The preference-indifference relation $\preceq$ is a binary relation over a set of actions/situations $\mathbf{\Delta}$ that satisfies the following:

$$\forall \delta_1, \delta_2 \in \Delta : \delta_1 \preceq \delta_2 \vee \delta_1 \npreceq \delta_2$$

$\delta_1 \preceq \delta_2$ means that δ_1 is not preferred to δ_2.

Definition 2.4.2 (Strict preference relation $\prec$)
The strict preference relation $\prec$ is a binary relation over a set of actions/situations $\mathbf{\Delta}$ that satisfies the following:

$$\forall \delta_1, \delta_2 \in \Delta : \delta_1 \prec \delta_2 \iff \delta_1 \preceq \delta_2 \wedge \delta_2 \npreceq \delta_1$$

$\delta_1 \prec \delta_2$ means that δ_2 is preferred to δ_1.

Definition 2.4.3 (Indifference relation $\sim$)
The indifference relation $\sim$ is a binary relation over a set of actions/situations $\mathbf{\Delta}$ that satisfies the following:

$$\forall \delta_1, \delta_2 \in \Delta : \delta_1 \sim \delta_2 \iff \delta_1 \preceq \delta_2 \wedge \delta_2 \preceq \delta_1$$

$\delta_1 \sim \delta_2$ means that δ_1 is indifferent to δ_2.

The previous binary relations form an invaluable framework that enables us to rank solutions, given that we have a model that can map any action/situation to a numeric value (utility). It is essential to recognize that utility and value do not constitute the same concept. For example, in the context of commuting to work, one can use the train or a taxi. The train would cost two dollars, whereas the taxi would cost twenty. These values do not take into consideration factors such as comfort, speed, or emissions. Concrete values such as cost and time are objective, whereas utility is quite subjective. An agent with an emphasis on comfort would give the train a lower utility score. In comparison, an agent with an emphasis on reducing emissions would give the train a higher utility score.

2.4.2 Utility-based HTN planning

One way of utilizing utility to guide an HTN planner is to assign each operator a utility score, but this is insufficient. HTN structures are intrinsically hierarchical, and the utility of a task is not necessarily identical to the utility of its subtasks. A significantly better approach is to assign a utility score to each operator, method, and compound task. The utility of a task is then the sum of its own utility and the utility of the subtask with the highest utility. In addition, it is essential to bear in mind that utility is extremely dynamic. Therefore, the planner must be able to recalculate the utility of each task as the planning process progresses or whenever the environment's state changes.

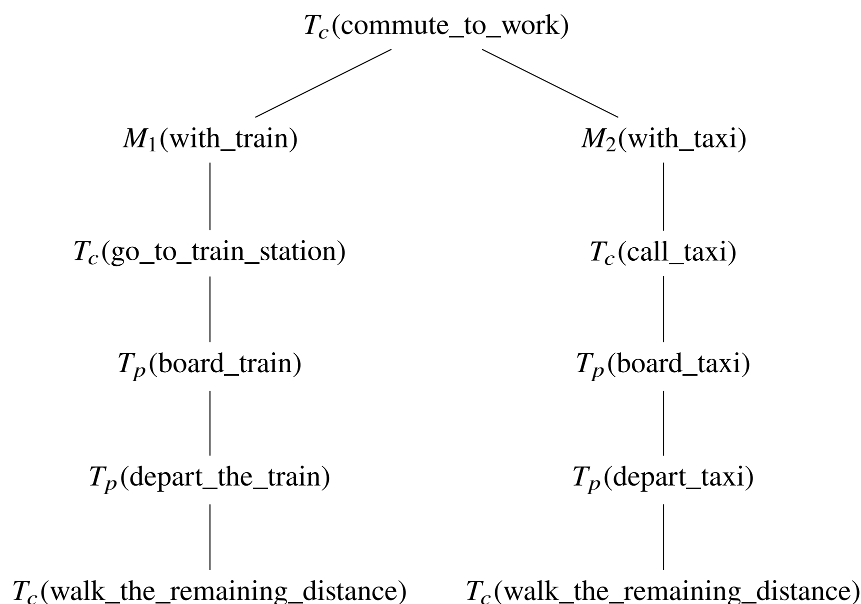

Figure 2.1: A simple HTN task network for commuting to work. With T_p and T_c representing primitive and compound tasks, respectively, and M_1 and M_2 representing methods.

Figure 2.1 shows a simple HTN task network for commuting to work. This example will help us to illustrate the necessity of having a utility score for each HTN construct. If we exclusively assign utility values to the operators, there will be no way to express whether an agent prefers or dislikes using the train, for example. The compound task "walk_the_remaining_distance" can be reached

via two distinct paths, and it would be illogical to assign it the same utility in both scenarios. The compound task “commute_to_work” itself could have an inherently low utility value if the agent prefers to work from home.

We will refrain from providing a concrete algorithm for integrating utility into HTN planning since it is highly dependent on the search strategies implemented by the planner and because it has already been done in [Aln19] [GL14] [AGA22]. However, we will go over our own implementation in detail later on.

2.4.3 Expected utility

Basic utility is an effective tool for comparing solutions and ensuring that the agent always chooses the solution with the most preferable outcome. However, basic utility assumes that the world is deterministic and that each action has only one outcome, which significantly impacts its applicability in the real world, where uncertainty is a fundamental challenge. To illustrate the limitations of basic utility, consider the scenario shown in Figure 2.2 in which an agent is offered the choice between two actions, δ_1 and δ_2. There are two possible outcomes for action δ_1: the agent will receive three dollars 40% of the time, and nothing the remaining 60% of the time. Action δ_2 also has two possible outcomes: 10% of the time, the agent will receive seven dollars, and 90% of the time, it will receive nothing.

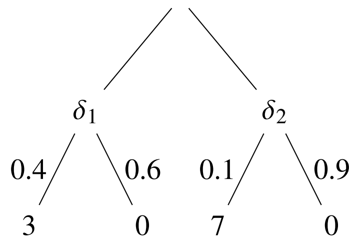

Figure 2.2: A simple example of hypothetical game.

Introducing probability renders basic utility absolute. Looking at the problem, a rational agent will virtually always choose action δ_1, because it has, on average, a better outcome, but how can we capture this notion mathematically?

$$E(\delta_2) = 0.1 \times 7 + 0.9 \times 0 < E(\delta_1) = 0.4 \times 3 + 0.6 \times 0$$

At first glance, the expected value might seem like a suitable tool to represent expected utility. However, it was clear from the early stages of research that rational agents do not always act based on expected value. As previously discussed, value is an objective concept, whereas utility is far more subjective and can be influenced by an agent’s preferences, needs, and risk tolerance. Expected value fails to capture the subjective nature of utility; thus, it is not a suitable tool for representing expected utility.

To further illustrate this point, we need to introduce the *St. Petersburg paradox*[7], which demonstrates how an agent whose only criteria for decision-making is expected value would propose a course of action that no rational agent would ever consider:

> A hypothetical casino in St. Petersburg offers the following game to a single player: After paying a fixed entrance fee, a fair coin is tossed until the first head appears, which ends the game. When the game ends, the player wins 2^n, where n is the round where the first head appeared.

Now the question is, *how much would a rational agent be willing to pay as an entrance fee to play this game?*

An agent using the expected value would be willing to pay an entrance fee up to the game's expected value which is infinite:

$$E = \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 4 + \frac{1}{8} \cdot 8 + \frac{1}{16} \cdot 16 + \cdots = \infty$$

which contradicts the fact that no rational agent would be willing to pay an arbitrarily large entrance fee to play this game. A better approach would be to define a utility function $\mathfrak{u}$ that reflects the agent's preferences. We can use $\mathfrak{u}$ to define the expected utility of an action δ over the probability distribution $\mathbb{P}$ of all its possible outcomes Ω.

Definition 2.4.4 (Expected Utility)
The expected utility of an action δ is the weighted sum of all possible outcomes of δ:

$$\mathbb{E}(\delta) = \sum_{\omega} \mathbb{P}(\omega) \cdot \mathfrak{u}(\omega)$$

Using expected utility, we can extend our prefrence relations:

$$\forall \delta_1, \delta_2 \in \Delta :$$

$$\begin{aligned} \delta_1 \preceq \delta_2 &\Longleftrightarrow \mathbb{E}(\delta_1) \leq \mathbb{E}(\delta_2) \\ \delta_1 \prec \delta_2 &\Longleftrightarrow \mathbb{E}(\delta_1) < \mathbb{E}(\delta_2) \\ \delta_1 \sim \delta_2 &\Longleftrightarrow \mathbb{E}(\delta_1) = \mathbb{E}(\delta_2) \end{aligned}$$

2.4.4 Risk attitude

As mentioned earlier, the utility function is a highly subjective model of the world that reflects the agent's preferences, most notably its risk tolerance. In this section, we provide a short overview of the standard risk attitude models and how the risk attitude can guide the behavior of an agent.

Sometimes rational agents are inclined to change behavior when the stacks are too high. For example, let us consider the following game, where a player has a 60% chance of winning 100 dollars and a 40% chance of losing 1 dollar. Conversely, some players would simply refuse to play

[7] https://en.wikipedia.org/wiki/St._Petersburg_paradox

the game because losing one dollar is too much of a risk. Some players would be willing to play the game, but they might also refuse to play if the stakes were higher. In other words, the risk attitude can be defined as the change in utility based on the ratio of potential wins to potential losses.

It is difficult to quantitatively model risk, and different fields have attempted to do so using different theories; since we cannot cover all of them here, we will explore the topic from the agent's viewpoint.

Maximizing gains or minimizing losses would be the primary goal of a naive agent. Such an agent would have an indifferent attitude toward risk (*Risk neutral*). This type of agent will always choose a course of action that maximizes/minimizes its gains/losses, regardless of the risk involved.

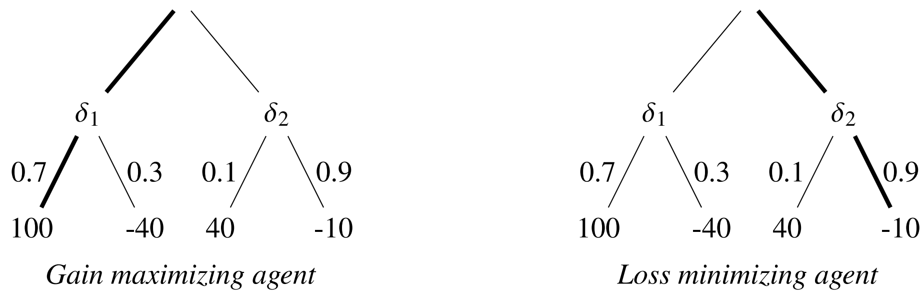

Figure 2.3: Risk-neutral agents' decision-making

Figure 2.3 shows two agents that are risk neutral. The first agent is gain-maximizing, and the second agent is loss-minimizing. The gain-maximizing agent will always choose δ_1 because it has a higher potential gain of 100. Whereas the loss-minimizing agent will always choose δ_2 because it has a lower potential loss, even though the probability of losing 10 is much higher than the probability of losing 40, the agent will still choose δ_2. Risk-neutral agents will always have a linear utility function. This means that the utility of an outcome is proportional to the outcome itself. More rational agents tend to be more sensitive to risk, especially when dealing with high-stakes situations. In general, rational agents show a tendency to avoid high risks when the payoff is insufficient to compensate for them, and they usually prefer an action course with a lower payoff and a higher probability of success. In addition, rational agents sometimes accept to bear small losses in order to avoid situations where there is a low probability of suffering a very large loss. This type of agent is called *Risk-Averse*.

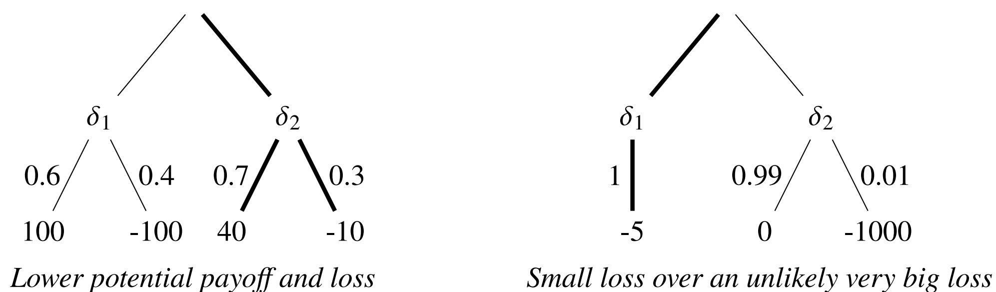

Figure 2.4: Risk-averse agents' decision-making

Two distinct situations are shown in Figure 2.4 to illustrate how an agent with a risk-averse attitude will always choose the action that involves the lowest risk. The utility function of a risk-averse agent is concave, with low risk outcomes having a bigger utility score.

Some agents are inherently more risk loving (*Risk-Seeking*). They will be more willing to take risks in order to achieve a higher payoff, even if it involves a higher risk. The utility function of a risk-seeking agent is convex, with high profitability outcomes having a bigger utility score. Figure 2.5 shows how a risk-seeking agent will behave in the same previously shown situations.

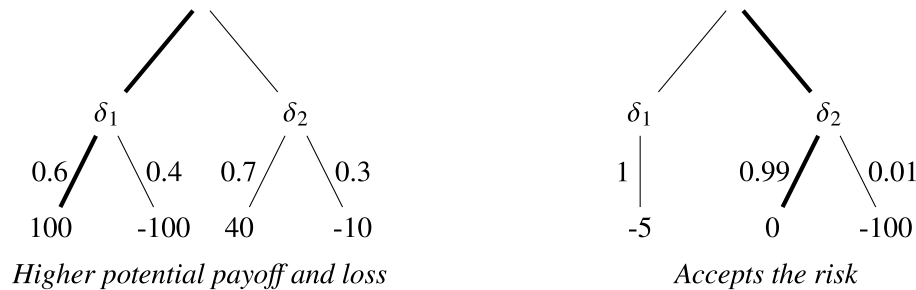

Figure 2.5: Risk-seeking agents' decision-making

Typically, an agent's preferences and risk tolerance will be dynamic, changing based on a broad range of factors such as current wealth, type of risk, and previous experiences. These factors make it hard to model the agent's preferences in one mathematical construct; however, [AGA22] proposes some risk-based utility functions. Figure 2.6 shows how the utility function of an agent might look based on their risk attitude.

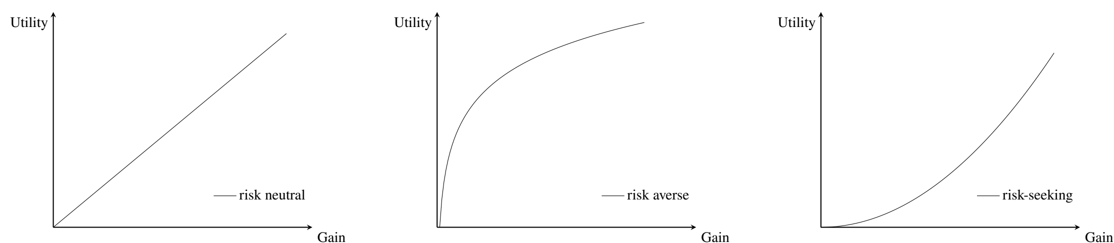

Figure 2.6: Different risk-based utility functions

At the end of this section, we would like to point out that although it has been shown [KT13] that the expected utility theory fails as a descriptive theory due to its inability to predict human behavior in some situations successfully, it is a very suitable normative theory that can be used to guide the decision-making process of rational agents, especially when combined with a good dynamic utility function.

2.4.5 Risk attitude and HTN planning

Our model of utility-based HTN planning can be easily extended to enable agents to make decisions based on their risk attitude by having a utility function that takes into account the risk involved in the decision. One benefit that we have is that we are not bound to one mathematical construct;

we can have multiple utility functions corresponding to different risk attitudes, and we can switch between them, choosing the function that better suits the agent's current situation. This allows our agents to have a more flexible and dynamic attitude toward risk. We can also utilize the rule-based nature of HTN to create a set of rules that can override the current risk attitude of an agent in some situations where we believe the agent would be better off with a more or less risk-sensitive attitude. It is important to note that these rules should not replace the risk attitude model of the agent but rather complement it to handle exception cases.

To further illustrate this idea, let us consider the following scenario where we have a stock-trading agent: The agent has a hand-crafted utility function with an overall risk-averse attitude that is based on a plethora of financial and economic factors. Each decision the agent will consider involves a certain amount of risk that will be weighted using its utility function. However, it would be beneficial for the agent to have a set of rules in place that can override its risk tolerance in real time to react to some edge cases. Such rules should not only be used to change the agent's attitude toward risk; they can also be used to directly interfere and change the evaluation of a given action to be more or less risky. Returning to our previous example, shorting a stock, which is betting that the value of a given stock will decrease, is generally regarded as a highly risky investment strategy. However, if a reputable investor is shorting stock X with a large sum of money, the action of shorting that stock should no longer be seen as risky.

Lastly, we would like to note that it is beneficial to utilize the concept of risk attitude in HTN agents even when the concept of utility is not modeled since it enables us to group actions and guide the decision-making process using one dynamic switch.

Listing 2.4 Example of using risk attitude without utility

```
(define (domain Little_Red_Riding_Hood)
    ...
    (:task get-to :parameters (?l - location ?a - riskAttitude))
    ...
    (:method stay-on-path
        :parameters (?l - location ?a - riskAttitude)
        :task (get-to ?l)
        :precondition(isRiskAverse ?a)
        ...
    )
    (:method wander-into-the-woods
        :parameters (?l - location ?a - riskAttitude)
        :task (get-to ?l ?a - riskAttitude)
        :precondition(isRiskSeeking ?a)
        ...
    )
)
```

Listing 2.4 shows how we can use risk attitude without utility. In this example, we have one compound task **get-to**, that can be achieved using one of two methods **stay-on-path** or **wander-into-the-woods**. The agent will choose between these methods based on its risk attitude. Integrating risk based decision-making in this way is quite simple and it can greatly impact the behavior of an agent in a dynamic environment.

Although it is a very static way to deal with risk, it is very useful, especially in domains where a detailed study of the domain by domain experts has already produced rich and stable guidelines that can be translated to HTN constructs. For example, we have utilized this in our implementation in the preflop stage, which domain experts have exhaustively studied and constructed detailed guidelines that can be grouped based on the risk attitude of the agent.

To summarise the ideas, we have covered in this section: The concept of utility can be incorporated into HTN planning to evaluate the potential outcomes of different actions and to choose the action that leads to the desired outcome. Furthermore, expected utility can be employed in probabilistic domains to serve the same purpose. In both cases, the concept of risk attitude can be implemented to adjust the utility of different actions based on the agent's risk tolerance. Finally, we have proposed grouping rules based on the risk attitude associated with their actions to control the behavior of the agent even if utility is not modeled for each component of the planning domain.

3 PokerShark

PokerShark is a Texas Hold'em bot built using HTN planning techniques. The bot utilizes the knowledge of domain experts and concepts of utility and risk awareness in its decision-making process. In the next sections, we will discuss the concept used by the bot and explore the specific implementation choices we have made.

3.1 The Planner

The first big decision we faced during the development of PokerShark was choosing which planner we wanted to use. We had several requirements that we wanted present in the planner. First, the planner must be performant. We are not looking to find the most efficient planner because we anticipate that the network size will be limited to a few hundred tasks. We prefer the planner to be battle-tested. Many planners were conceived and developed in academic settings but have yet to be tested in the real world. We wanted to avoid these planners. Poker is a complex domain, and a game of poker has many elements and variables, making it more challenging to describe the game's state. That is why we need the planner to have an expressive description language that is easy to use and not too difficult to learn. Furthermore, defining the bot strategy will be tedious and require a lot of fine-tuning. Hence the planner needs to provide easy debugging tools.

Typically HTN planners do not support utility or expected utility, which is essential to our work. We have to assume that we will have to extend the functionality of the planner, so we have to consider the extensibility of the planner, which introduces an array of factors, such as the programming language, the quality of the code, the architecture, the frameworks..., and a lot of other factors. Most likely, we will also have to modify the description language, which makes the parser of the planner a very important factor to consider. Lastly, we have to think about the integration between the planner and the bot. We prefer a seamless integration where the bot can notify the planner about the game's state and receive the planner's decision. We will explore next some of the planners that we have considered:

JSHOP

JSHOP is part of the Simple Hierarchical Ordered Planner (SHOP) project[1] developed by the University of Maryland. The planner is a Java-based implementation of SHOP2. JSHOP was first released in 2005, making it one of the older planners we have considered. It was part of numerous studies and was used in many applications, which makes us more confident in its performance and reliability. SHOP planners have a custom description format developed and extended throughout the

[1] http://www.cs.umd.edu/projects/shop/description.html

years. JSHOP uses ANTLR3[2] to parse domain and problem definitions. Rather than interpreting the definition files, JSHOP compiles them into Java code that the planner can call, which makes JSHOP harder to integrate.

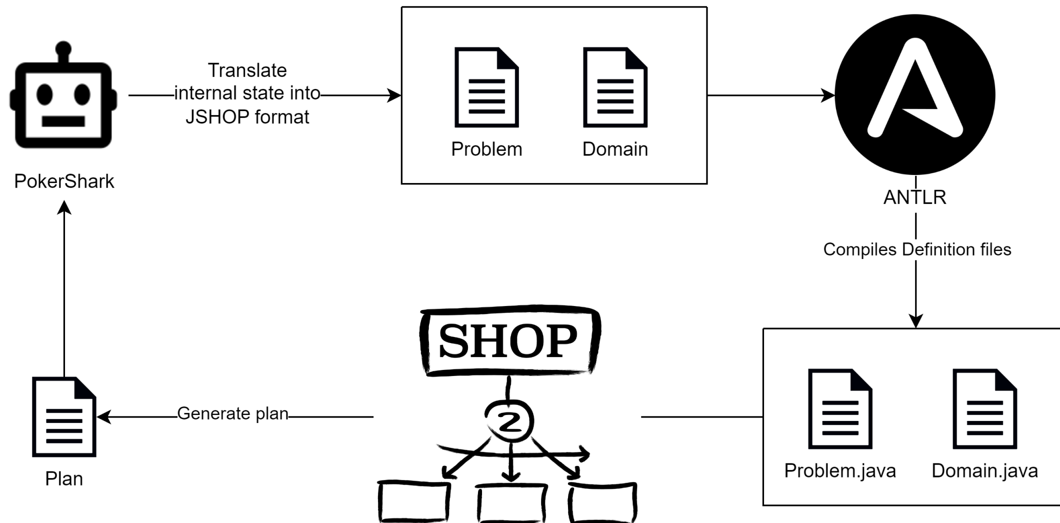

Figure 3.1: Multiple translation and read/write operations required.

JSHOP does not support utility nur expected utility. Although the code is well-documented and can be considered clean, it will be challenging to extend due to the architecture. Since the description format also needs to be extended, ANTLR rules and grammar files must be modified. ANTLR3 is notoriously tedious, and we rather not deal with it. The planner has a GUI to run plans, which can be helpful for debugging, but it is not ideal. Nevertheless, JSHOP is a great planner with some advantages and disadvantages that must be considered for the final decision.

SHOP3

SHOP3[3] is the successor of SHOP2, which means it benefits from all the expertise that went into SHOP2. Unfortunately, it is Lisp-based, which is personally a big downside. In addition, the planner does not support expected utility and must be extended. On the other hand, the planner supports PDDL and provides excellent debugging features.

PANDA

The PANDA framework[4] is one of the most promising candidates. The planner is developed and maintained by the Institute of Artificial Intelligence at Ulm University. The PANDA framework consists of several c++ modules that provide various features, such as plan verification and repair.

[2] https://www.antlr3.org/
[3] https://shop-planner.github.io/
[4] https://panda-planner-dev.github.io/

However, the planner uses a custom parser that supports HDDL and the SHOP format. The high complexity of the parser and the parsing pipeline, including the number of modules and transitionary formats involved, can require a significant amount of time and effort to manage. Undoubtedly, PANDA is a good planner but has yet to be battle-tested, and it does not have great potential for extension.

InductorHtn

InductorHtn[5] was not developed in an academic setting. It was originally intended to be part of an ios-based game engine. The planner draws a lot of inspiration from SHOP2. It uses Prolog as a description format, but it also provides the ability to use native C++ constructs to define the planning domain and problem. This is a significant advantage as no compilation is required to run the planner, which greatly facilitates the integration process. InductorHtn also provides good debugging features. However, the planner's code lacks documentation and does not look like a great candidate for modification.

FluidHTN

FluidHTN[6] is a feature-rich planner based around the Builder pattern's principles. The planner offers numerous features, including partial planning, domain splicing, and early rejection. FluidHTN supports only native domain definition. However, it has a powerful and expressive domain builder, which greatly simplifies domain and problem description. Moreover, it has great debugging tools that are superior to almost all other planners.

FluidHTN is the first candidate that was built with extensibility in mind. As a result, all the planner's components and elements are easy to modify, extend or replace. In addition, the planner has surprisingly a big active community, especially game developers, which suggests that the planner is performant and battle-tested enough for us to consider a good candidate. Finally, having native domain definition support makes the integration much easier.

Planner	Maturity	Expressivnise	Extensibility	Debugging	Integration	Overall
JSHOP	5	4	3	2	2	★★★☆☆
SHOP3	5	4	1	3	1	★★★☆☆
PANDA	2	4	1	2	0	★★☆☆☆
InductorHtn	4	5	3	3	2	★★★☆☆
FluidHTN	5	5	5	4	5	★★★★★

Table 3.1: Planner comparison.

Table 3.1 shows our final evaluation of the planners. We have used a 5-point scale to evaluate the planners. Our evaluation is based on the following criteria: documentation, expressiveness, extensibility, debugging, and integration and is heavily influenced by our preferences. Overall, FluidHTN is the most complete planner for our use case.

[5] https://github.com/EricZinda/InductorHtn

[6] https://github.com/ptrefall/fluid-hierarchical-task-network

FluidHTN Showcase

FluidHTN is clearly influenced by game development but is also designed to be adaptable for any use case. The planner has a unique structure. Thus we would like to dedicate this section to showcasing its underlying concepts. FluidHTN is built for an agent to be dynamic. Instead of calling the planner and providing the problem definition, the planner can be integrated directly with the world state enabling a more dynamic behavior where the planner can observe changes in the environment, changing the plan during run-time. By default, FluidHTN does not require the use of explicit goal states. Instead, it uses the results of task decomposition to determine whether planning is complete, should continue, or has failed. This allows the planner to be more flexible and adaptable, as it can handle a wider range of planning scenarios without the need for pre-defined goals.

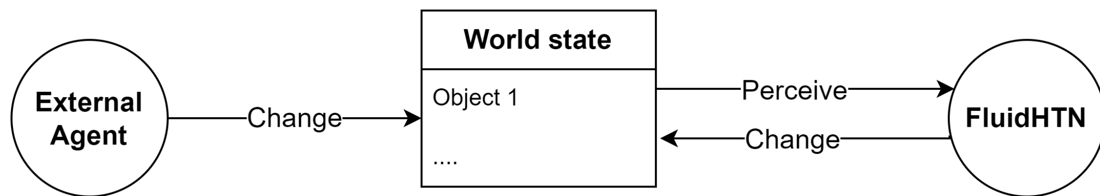

Figure 3.2: FluidHTN with shared state.

Using the planner in the configuration shown in figure 3.2, where FluidHTN uses a pre-defined domain, has numerous advantages, but it forces the planning domain to be static. We would like to have more control over the planning domain. Therefore we are not going to use the planner in this configuration. Instead, we will use the planner as a standalone planner that is not aware of the agent's state. The planner will be used to generate plans that will be executed by the agent. The agent is responsible for a number of tasks, including receiving game state information from the game server, updating its internal world state model, generating the necessary domain and context definitions, running the planner, and executing the resulting plans. This configuration is shown in Figure 3.3.

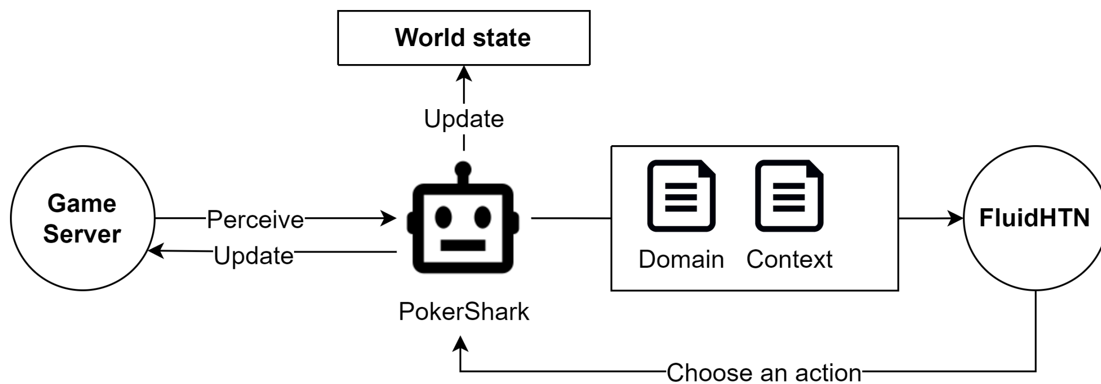

Figure 3.3: PokerShark architecture.

FluidHTN adopts the concepts of compound and primitive tasks of HTN. Compound tasks can be implemented using one of two structures. The first structure is called *Sequence*. In order to decompose a Sequence, all of its sub-tasks must decompose successfully in contrast to the second structure, the *Select*, which only requires the decomposition of one sub-task to be successfully decomposed. It should be noted that preconditions can be applied to tasks regardless of type.

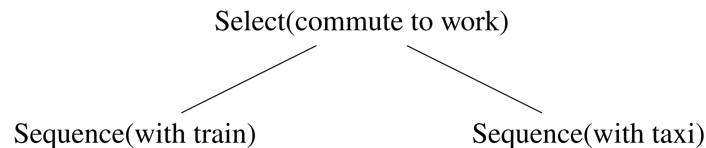

Figure 3.4: Selects can choose one path, Sequences need to decompose all subtasks.

The architecture of the planner makes it easy to introduce new representations of any component in the planning process, such as tasks, conditions, and effects. For example, Listing 3.1 shows RandomSelector, which randomly selects and decomposes one of its subtasks.

Listing 3.1 RandomSelector definition.

```
public class RandomSelector : Selector
{
    protected DecompositionStatus OnDecompose(IContext c, int i, out Queue<ITask> result)
    {
        var taskIndex = new Random().Next(i, Subtasks.Count);
        var task = Subtasks[taskIndex];
        return OnDecomposeTask(c, task, taskIndex, null, out result);
    }
}
```

Primitive tasks are defined as actions where each action could have a set of preconditions and a set of effects. Lastly, FluidHTN enable us to produce highly readable domain definitions, making them easier to modify or maintain. Listing 3.2 shows a snippet of PokerShark domain that showcases how FluidHTN definition format is easier to understand compared to other description languages such as PDDL.

Listing 3.2 Snippet of PokerShark domain builder.

```
Select("Raise Recommendation");
    IfRaiseRecommendation();
    Select("Too Risky Raise");
        IfRaiseRecommendationTooRisky();
        Action("Fold if call to risky");
            IfCallTooRisky();
            Do(Fold);
            ...
```

3.2 Opponent Modeling

As we have already established, information is a key factor in the game of Texas Hold'em. Depending on the position, we might get some or no information about the strength of the opponent's hand. In the early stages of the round, especially the preflop, we often need to make a decision solely based on the strength of our pocket. However, the strength of a hand is affected by many factors. Unlike chess, where we can assign a concrete evaluation of a given position no matter who the players are, the strength of a pocket or hand, in general, can only be estimated in correlation to the current situation and opponents. For example, a player in an early position with a pocket of A♦Q♦ is contemplating whether he should raise, call or fold. The cards are suited, which makes the pocket stronger. Raising would be the correct action if the player is convinced his raise will not scare the opponents and make them fold. On the other hand, calling would be better if the opponents were tight and easily scared by preflop raises. Even folding would make sense if a tight passive opponent has just made a raise[7]. The terms tight and loose refer to a player's calling/raising range, with tight players only calling/raising with strong hands. Passive and aggressive refer to a player's action preferences, with passive players calling more often than raising and aggressive players raising more often than calling.

Understanding the opponent's playing style is essential for predicting future actions. PokerShark captures this notion using opponent models. When a game starts, a model is created for each opponent. This model gets refined with each action the opponent makes. We have to note here that deducing the preferences of an opponent based solely on the observed behavior is very hard, particularly if the available action history is limited. PokerShark does not keep track of the opponent's action history after the game is finished, primarily because that can be considered unfair, and the effects of doing so are out of the scope of this work.

3.2.1 Opponent Statistics

In order to model the opponent's playing style, we need to keep track of some statistics. The following statistics are used in PokerShark and are adopted from [JS09]:

VPIP - Voluntarily Put Money In Pot

Shows how often a player has voluntarily entered the pot in the preflop, either by calling or raising. VPIP is one of the most fundamental statistics used in player profiling, especially when combined with other statistics. VPIP can be directly correlated with how loose or tight a player is. Higher values mean the player tends to call or raise even with weaker hands. Lower values mean the player is more selective and only plays strong hands.

[7] a full analysis of AQ can be found in [Skl03]

PFR - Preflop Raise

PFR complements VPIP to create an accurate estimation of the player preflop strategy. Using PFR alone can be quite misleading. For example, loss-aggressive and tight-aggressive players will have higher PFR. The first is more likely to raise on weaker hands, whereas the other is much more deliberate with his raises. The VPIP includes both player calls and raises. Consequently, it will always have a bigger (or equal) value. If the PFR makes up the majority of VPIP, the player can be considered aggressive. If the VPIP is much higher than the PFR, the player is more likely to be passive.

WTSD - Went To Showdown

WTSD shows how often a player has made it to the showdown, which can indicate how likely the player is to fold before the showdown. Aggressive players tend to refuse to fold, resulting in a higher value. In contrast, passive players that get shy when challenged have lower WTSD. Combining WTSD with other statistics can give us a good idea of the opponent's playing style post-flop.

WSD - Won Money at Showdown

A percentage that shows how accurate the opponent's bets are. A lower WSD means the player usually bets on weaker hands or bluffs too much. However, this can be misleading since not all wins/losses are equal. Sometimes players can hold weaker hands if the stakes are low, which can happen a lot in games where the majority of players are tight-passive. In this case, the WSD will be low, but the player is not necessarily a bad player.

WWSF - Won When Saw Flop

This statistic complements the last two. It shows how often the player won post-flop. A high WWSF means the player's bets tend to scare the opponents, which means the player is good at bluffing or makes good bets. A low WWSF indicates the player is not bluffing much, which can signify a tight-aggressive player.

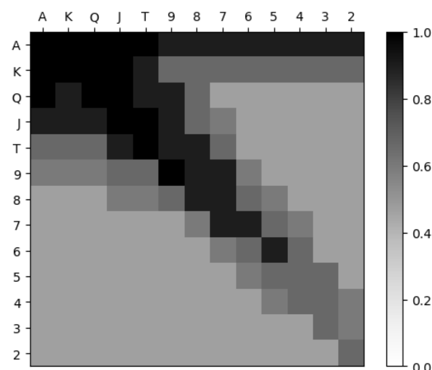

Figure 3.5: Adjustment matrix example.

3.2.2 Opponent Ranges

PokerShark uses the previous ratios to infer the opponent's playing style. Additionally, an adjustment matrix is created for each opponent. The adjustment matrix has an entry for each possible pocket. Each entry represents the probability of the opponent holding that pocket. This matrix is updated following each action the opponent performs based on its playing style.

The adjustment matrix begins with all possible pockets being considered equally likely to be chosen by the opponent. After each decision made by the opponent, a set of adjustment factors is applied to the matrix based on the observed playing style.

These factors reflect the preferences of players with similar states and help the matrix better predict the opponent's choices. As the game progresses, the accuracy of the matrix's predictions for the opponent's calling and raising ranges will improve. Figure 3.6 shows how the adjustment matrix is refined over time. After several rounds, PokerShark will have a basic estimate of the opponent's preferences and playing ranges.

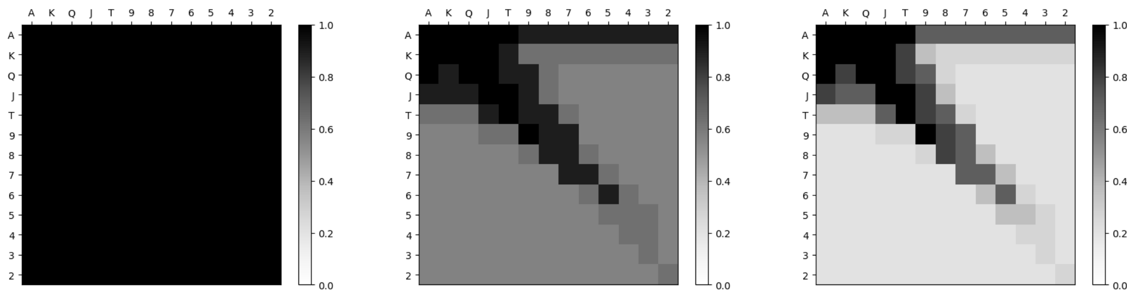

Figure 3.6: Adjustment matrix progression.

The opponent's decision-making process is complex and involves many factors, such as position, the opponent's perception of other players, pot size, stack size, and many other factors. Therefore, we can not pretend that having an estimate of the preferred playing ranges is sufficient. However, it is important to understand how PokerShark interprets and utilizes this model to make better decisions, which we will explore in the next sections.

3.3 Preflop Strategy

Due to the dynamic of Texas Hold'em, how you play the preflop will greatly influence whether you win or lose money over the long run. Good players can predict how their hand will fare against their opponents and assess the potential of the hand improving over the following stages. The amount and the quality of information available to the player can vary depending on the situation. However, the two cards composing the pocket can be a great starting point for rational decision-making.

The number of starting hands is limited $\binom{52}{2} = 1326$ possible combinations. The majority of these hands are equal or share some similarities. Thus, they can be categorized into 169 unique types. Because the number of combinations is relatively small, poker experts could set clear guidelines on how the preflop should be played, considering different factors such as position, number of opponents, playing style, and number of bets.

PokerShark incorporates the guidelines from [Skl03] to construct a recommendation system, enabling the planner to leverage expert knowledge while choosing an action in the preflop. The guidelines suggest grouping the 169 different types into nine groups based on hand strength. The groups are ordered from the strongest to the weakest. Table 3.2 shows a small example of the defined groups and their corresponding hand types.

Group 1:	AA, KK, QQ, JJ, AKs
Group 2:	TT, AQs, AJs, KQs, AK
Group 3:	99, JTs, QJs, KJs, ATs, AQ
Group 4:	...

Table 3.2: Example of the hand groups based on potential strength defined by [Skl03]

The guidelines are grouped based on the position of the acting player: Early position, Middle position, Late position, and for the Blinds. For each position, detailed rules are given based on the state of the bot, the number of bets, and the state of the game. We have tried to the best of our ability to translate the guidelines into rules. Of course, some ambiguity is unavoidable, but we believe we have successfully captured the most important ideas and tactics.

In an unraised pot, you can play all hands in Groups 1-5 when the game is typical or tough. In a loose, passive game it is all right to play the Group 6 hands as well.

```
IfFirstToPot();
Select("Loose Passive Game");
    IfLoosePassiveGame();
    Action("Always Raise on Groups 4,5,6");
        IfPocketFromGroup(4, 5, 6);
        Do(RecommendAlwaysRaise3Or4BB);
    End();
End();
```

Figure 3.7: Example of converting a guideline into a recommendation.

We decided to introduce an intermediate step between the guidelines and the decision for several reasons. First, following the guidelines strictly would result in the bot folding around 70% of the time, which is correct when facing strong players. However, folding so often, particularly this early in the round, means the bot forfeits many potential wins by being too tight. Another reason is that the guidelines do not consider the calling/raising amounts. Thus a reasoning process needs to occur before making the final decision.

Hand groups are great for categorizing hands, but they are too general and do not consider one important factor: the number of opponents. The number of opponents is a crucial factor that directly affects hand strength. For example, 99 and JTs are both in group 3. However, 99 has much bigger win odds when facing one opponent, while JTs has bigger odds when facing seven or more opponents.

Pocket odds are difficult to estimate in the preflop because of the size of the search tree, especially when the number of opponents increases. However, Monte Carlo analysis can be used to run simulations to approximate the results.

For every opponent count from one to nine, a Monte Carlo simulation was used to approximate the win odds for every possible starting hand. The results were cached in a hash table. PokerShark uses the recommendation of the guidelines and the odds from the hash table to make a decision based on the current risk attitude of the bot at that point in time.

Pocket	1	2	3	4	5	6	7	8	9
AA	85.3	73.4	63.9	55.9	49.2	43.6	38.8	34.7	31.1
AKs	67.0	50.7	41.4	35.4	31.1	27.7	25.0	22.7	20.7
AK	65.4	48.2	38.6	32.4	27.9	24.4	21.6	19.2	17.2
AQs	66.1	49.4	39.9	33.7	29.4	26.0	23.3	21.1	19.3
AQ	64.5	46.8	36.9	30.4	25.9	22.5	19.7	17.5	15.5
AJs	65.4	48.2	38.5	32.2	27.8	24.5	22.0	19.9	18.1
...	...	...	...	...	...	...	...	...	...

Table 3.3: Simulation results hash table.

3.3.1 Risk Attitude

PokerShark has a dynamic risk attitude directly derived from the ratio of its current stack compared to its initial stack. The bot can have one of three attitudes toward risk: Risk-Averse, Risk-Neutral, and Risk-Seeking. The risk attitude is used to guide the planner in the preflop stage and choose a utility function in the later stages. We preferred this implementation over a dynamic utility function because this implementation gives us the benefit of clear thresholds that we can easily modify without crafting complex mathematical functions. However, we have built the bot in a flexible and modular way, so replacing this behavior with a sound mathematical function should be doable without much effort.

Algorithm 3.1 PokerShark Risk Attitude

```
procedure GetAttitude(CurrentStack, InitialStack)
    Attitude ← RiskNeutral
    if CurrentStack > InitialStack * 1.5 then
        Attitude ← RiskSeeking
    else if CurrentStack < InitialStack * 0.5 then
        Attitude ← RiskAverse
    end if
    return Attitude
end procedure
```

3.3.2 From Recommendation to Decision

The guidelines that we have implemented as HTN tasks will determine a recommendation. PokerShark tries to improve this recommendation based on the current situation. The recommendation can suggest to fold, call, or raise. PokerShark deals with each recommendation differently. Converting the recommendation into a decision required us to define different thresholds. We have tried to

choose the thresholds and the different decisions that we had to make reasonably, relying on intuition and our limited knowledge of the game, considering the different risk attitudes that the bot might have.

We also want to clarify again how PokerShark utilizes the monte-carlo simulation mentioned in the previous sections. The planner has access to the hash table, which contains the results of the simulation shown in table 3.3. We implemented a precondition that checks if the current hand of the bot is greater/less than a given probability, also called equity.

Fold Recommendation

As we have already discussed, strictly following the guidelines will result in the bot folding 70% of the time. Therefore, we decided to follow a looser strategy in the preflop.

The authors of the guidelines did overlook some of the obvious situations. However, we believe they did not mention them because the player should always act rationally and deviate from these guidelines when the situation calls for it. For example, one situation where folding is obviously wrong is when it does not cost us anything to call. By folding, we lose a free chance of potentially getting to see the flop.

The fold recommendation is handled differently based on the bot's risk attitude. If the bot's current attitude is risk-averse, we convert the fold recommendation into a call if the pocket has more than 60% win odds, with one exception if the call amount is too big and the pocket has less than 80% odds, then we fold. Otherwise, we follow the recommendation. If the bot is risk-neutral, we follow the previous process, but instead of calling, we do a min-raise if we are the first to the pot. In the case that the bot is more risk-seeking, we only fold if the pocket odds are less than 50% or if the calling amount is too big and the pocket has less than 70% win odds. If nobody has entered the pot yet, we raise one or two big blinds. Else we do a min-raise.

Call Recommendation

A call recommendation usually indicates that the bot's pocket and position are good. However, the guidelines do not consider the calling amount. Therefore, we have defined thresholds for each risk attitude. If the threshold is crossed, PokerShark converts the call recommendation into a fold decision. Otherwise, we either follow the call recommendation or do a min-raise based on the bot's attitude.

Raise Recommendation

We do a similar process to interpret a raise recommendation. The bot usually follows the recommendation. If the raise is too risky, we consider calling. If calling is also risky, we then fold. A big raise in the preflop can scare the opponents and make them exit the hand without betting. That is why the bot occasionally utilizes the tactic of check-raising, where the raise is replaced by a call, and after the opponents enter the pot, a raise can be made.

3.3.3 From Decision to Action

Human players are quick to notice tendencies in the opponent's behavior. A rule-based system means the bot will have clear patterns that rational players can pick up and exploit. We have decided to introduce stochasticity into the decisions making process of PokerShark by implementing the decision as a vector composed of three components, each corresponding with the probability of fold, call, and raise. PokerShark will randomly choose an action based on the weights in the decision vector. This produces the random behavior that we are looking for while providing the ability to definitely choose an action by setting its weight to one and the other two vector components to zero. A similar mechanism is used for the betting amount making the bot unpredictable and harder to read.

We have to note here that each primitive task in the domain definition that is responsible for producing a decision will also set the different weights in the decision vector. For example, let us assume that somewhere in the domain definition of PokerShark, we have a sequence that causes the bot to fold if it holds a hand made up of 2 and 3. Listing 3.3 shows such a sequence.

Listing 3.3 Fold if the pocket is 23.

```
Select("Fold on 23");
  IfPocketIsOneOf(new Pocket(Rank.Two, Rank.Three));
    Do(OftenFold);
  End();
End();
```

The OftenFold action will create a decision object. Like the one shown in Listing 3.4.

Listing 3.4 Decision object that folds 90% of the time

```
function Decision OftenFold(){
    return new Decision() { Fold = 0.9, Call = 0.1, Raise = 0 } ;
}
```

When converting the decision into an action, PokerShark will respect the weights defined in the decision vector and randomly choose one of the actions using the defined probabilities. For example, the OftenFold function will produce a decision that will get converted to a fold action 90% of the time, which is the intended behavior of the sequence. However, there is a 10% chance that PokerShark will call. This stochastic behavior is intended to deceive opponents and make predicting the behavior of PokerShark harder to read.

3.3.4 Fish Net

Players that do not have a great deal of experience in the game of poker are referred to as fish. In contrast, more experienced players are called sharks, hence the name PokerShark. Fish players tend to take more risks by calling or raising big amounts. Facing fish players in the preflop, especially if they are loose-aggressive, could be challenging. Therefore, we have created a mechanism that detects fish players by analyzing their behavior and identifying patterns that are characteristic of fish players. The Fish Net also tries to exploit their playing style. Of course, this means PokerShark will deviate from the optimal strategy and take big risks. For example, PokerShark will call big amounts or go all-in more often when facing a fish.

3.4 Postflop Strategy

Reaching the flop means we have access to more information that enables us to conceptualize how the hand will progress. Usually, as the round progress, the number of active opponents decreases, making it easier to estimate the strength of our hand. In the next section, we will explore how PokerShark estimates the strength of its pocket.

3.4.1 Hand Strength

We need to define a metric that enables us to assess how our pocket would perform against the hands of the opponents. Numerous methods were proposed in the literature, such as Artificial Neural Networks [BEW13], Monte Carlo analysis, and enumeration [BDSS02].

Before discussing hand strength, we need to solve the problem of comparing two hands. There are three community cards on the flop, and each player has two pocket cards, which means a 5-card hand can already be constructed. So if the opponents show their cards, how can we tell which hand is the strongest?

Absolute Hand Rank

Comparing hands is not trivial, but it can be done efficiently using the absolute hand rank, which is a numerical value assigned to each possible hand that allows us to compare hand strength. There are many implementations of the absolute hand rank, but most implementations are based on Cactus Kev's Algorithm[8]. Although we do not want to go extensively into this topic, we will provide a high-level outline. There are 2.598.960 unique possible hand combinations, but most hands are not distinct. For example, A♥A♦A♣A♠K♥ and A♥A♦A♣A♠K♦ are unique hands but have the same rank. If we only consider distinct hands, we have only 7462 different combinations, which is manageable.

[8] https://suffe.cool/poker/evaluator.html

Hand Value	Unique	Distinct
Straight Flush	40	10
Four of a Kind	624	156
Full Houses	3744	156
Flush	5108	1277
Straight	10200	10
Three of a Kind	54912	858
Two Pair	123552	858
One Pair	1098240	2860
High Card	1302540	1277
TOTAL	**2598960**	**7462**

Table 3.4: Number of unique and distinct hands

These combinations are ordered into a hash table with an index representing the hand's value. Finally, an encoding function is used to create a hand mask that can be used to look up the hand's value using the hash table. This algorithm enables us to compare hands efficiently, which is a very important tool that we will utilize in estimating hand strength. We have to note here that the following methods and algorithms are adopted from [BDSS02],[Pap98], and [DBSS00].

3.4.2 Facing one Opponent

when facing one opponent post-flop enumerating all his possible pockets is not computationally hard. The deck of cards has 52 cards, we have two cards, and on the flop, we get to see three of the board cards, leaving the opponent with $\binom{47}{2} = 1081$ possible combinations that a modern CPU can enumerate in a matter of milliseconds. The number of possible combinations only goes down as the round progress.

Algorithm 3.2 Hand Strength estimation against one opponent

procedure HANDSTRENGTH($Pocket, Board$)
  $PocketRank \leftarrow GetAbsoluteHandRank(Pocket|Board)$
  for $OpponentHand \in GetPossibleHands(Pocket, Board)$ **do**
    $OpponentRank \leftarrow GetAbsoluteHandRank(OpponentHand)$
    if $OpponentRank > PocketRank$ **then**
      $Wins \leftarrow Wins + 1$
    end if
    if $OpponentRank = PocketRank$ **then**
      $Wins \leftarrow Wins + 0.5$
    end if
    $Count \leftarrow Count + 1$
  end for
  return $Wins/Count$
end procedure

Algorithm 3.2 is used to deliver the win odds of our hand by iterating throw all possible opponent hands. We consider a draw as half as good as a win. Although the algorithm might seem very reasonable at first glance, it does not consider opponent preferences, which is a significant weakness because in the post-flop, almost all of the weak hands have already been folded. Many of the wins we are counting toward hand strength have no value in practice. Let us consider the following board: Q♣Q♠J♦7♥3♦ and the pocket A♣K♦. The previous algorithm will assume all opponent's hands are equally likely and will give a 0,51 win odds. If we are facing a tight opponent with a range similar to the one shown in figure 3.8, the actual win odds are closer to 0,12 which is a big difference.

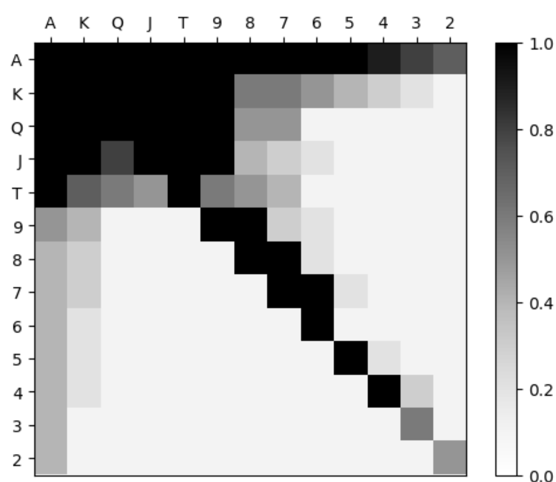

Figure 3.8: Tight opponent range

Here is where adjustment matrices and opponent models prove valuable. We can adjust the algorithm to weigh each win/draw, by simply adding the probability of the opponent holding that pocket in the case of a win and half of that in the case of a draw, resulting in a much more accurate estimation.

3.4.3 Hand Potential

One very important factor that we need to take into consideration when assessing hand strength is how future board cards are going to affect our pocket. For example, our pocket is 8♥7♥ and a board of A♥T♥3♠ the direct hand strength of our hand is low 0,18 (assuming the opponent has no preferred range). However, any heart card will give us a flush. With a seven or eight, we can get one or two pairs. Nine and six or Jack and nine will make a straight. The hand has a significant chance to improve with two board cards remaining.

Creating a mathematical notion for hand potential has to include the probability PP, the probability of the hand improving from a loss or draw to a win, and PN, the probability of the hand worsening going from a win or draw to a loss.

There are three board cards on the flop, and as calculated earlier, the opponent has 1081 possible pockets. The two remaining board cards can be chosen from 45 cards, meaning $\binom{45}{2}$ = 990 possible combinations. In total, we have to enumerate 1081 * 990 = 1.070.190 different cases, which is not challenging for a modern CPU.

On the turn, there are significantly fewer cases to consider, $\binom{46}{2}$= 1035 possible opponent's pockets, and one of 44 cards as a potential board card leaving us with 1035 * 44 = 45.540 cases to consider.

PP and PN are calculated as follows (equations from [Pap98]):

$$PP = \frac{P_{loss,win} + \frac{P_{loss,tie}}{2} + \frac{P_{tie,win}}{2}}{P_{loss} + P_{tie}}$$

$$PN = \frac{P_{win,loss} + \frac{P_{win,tie}}{2} + \frac{P_{tie,loss}}{2}}{P_{win} + P_{tie}}$$

The enumeration can also be weighted using the opponent model to get more accurate estimations. PokerShark uses The *HandPotential* Algorithm, which calculates PP and PN. The Algorithm is explained in detail in [Pap98], to give a brief overview, the algorithm enumerates all possible combinations of the opponent's pockets and the remaining board cards. For each combination, it adds the probability of the opponent holding that pocket to the corresponding win/draw/loss case. The algorithm then calculates PP and PN as explained above.

3.4.4 Facing n Opponents

Introducing another opponent to the game will dramatically increase the number of combinations and cases we need to enumerate. Adding more opponents will increase the number of the needed enumrations to the point that it is not feasible anymore. To solve this problem, we approximate the odds by calculating the product of the odds facing each opponent separately.

$$W = W_{op1} * W_{op2} * \cdots * W_{opn}$$

This approach does not consider the dynamic of the opponents interacting with each other. However, it gives us a simple solution to a complex problem while utilizing the different preferences each opponent has shown. A similar approach is used for the hand potential.

3.4.5 Monte Carlo Simulation

The accuracy of the two algorithms, HandStrength and HandPotential, directly depends on the accuracy of the model. To adjust for any inaccuracy in the modeled preferences of the opponents, we use the result of a Monte Carlo simulation. The simulation uses the same number of opponents that the bot is facing, but it does not take into consideration any preferences. The results of the simulation are then used to adjust our estimation as explained in the following section.

3.4.6 Hand Strength Estimation

We combine the previously examined algorithms and methods to get an estimate of the strength of the hand. Since inaccurately profiling the opponent has a significant impact on the algorithm's predictions, we must place a greater emphasis on the simulation result:

$$S = W_n * 0.35 + MCS * 0.65$$

Finally, we add the normalized sum of the hand potential:

$$HS_n = S + (1 - S)PP - (S * PN)$$

3.4.7 Expected Utility Implementation

Although FluidHTN has a big community, we could not find any prebuilt solutions for expected utility. Fortunately, FluidHTN is very flexible and easy to extend. We have extended Primitive tasks to include a list of possible utilities with their probabilities and added a new type of Compound tasks called ExpectedUtilitySelector, which decomposes the task with the highest expected utility from the applicable tasks.

Construct	**Description**
VariableUtility	Pair of utility an probability.
VariableUtilityTask	Extention of PrimitiveTask to include a list of VariableUtility.
ExpectedUtilitySelector	Extention of Selector that can have only VariableUtilityTasks as subtasks.

Table 3.5: Expected Utility Constructs

VariableUtilityTask can calculate its expected utility based on the current risk attitude of the bot. Each one of the attitudes corresponds to a utility function. Figure 3.9 shows the mathematical functions used to adjust the expected return of each outcome.

$$U(c) = -\frac{e^{-\alpha c} - 1}{\alpha}$$

(a) Risk Averse

$$U(c) = c$$

(b) Risk Neutral

$$U(c) = \frac{e^{\alpha c} - 1}{\alpha}$$

(c) Risk Seeking

Figure 3.9: Utility Functions, Where α is the risk sensitivity

ExpectedUtilitySelector uses Algorithm 3.3 to find the task with the highest utility among the applicable tasks.

Algorithm 3.3 Find the task with the highest utility

```
procedure FindBestTask(Context, SubTasks)
    for SubTask ∈ SubTasks do
        if SubTask.IsApplicable(Context) then
            Utility ← SubTask.GetUtility(Context)
            if Utility > BestUtility then
                BestUtility ← Utility
                BestTask ← SubTask
            end if
        end if
    end for
    return BestTask
end procedure
```

3.4.8 Decision making

Although expected utility gives us a great deal of control to guide the decision-making process of the bot, we need other mechanisms to achieve certain behaviors. For example, raising actions involve inherently more risk than folding or calling. Thus the utility of raise actions should be treated differently than the utility of folding or calling. Strictly following expected utility would result in a bot that rarely bluffs, which is a very important tactic in Texas Hold'em. Additionally, there is a wide range of tactics that we would like the bot to utilize. One example of such tactics is Slow-Playing, where even when holding a strong hand, the player does not place a big raise but rather performs a series of calls or small raises, setting up a trap for the opponents.

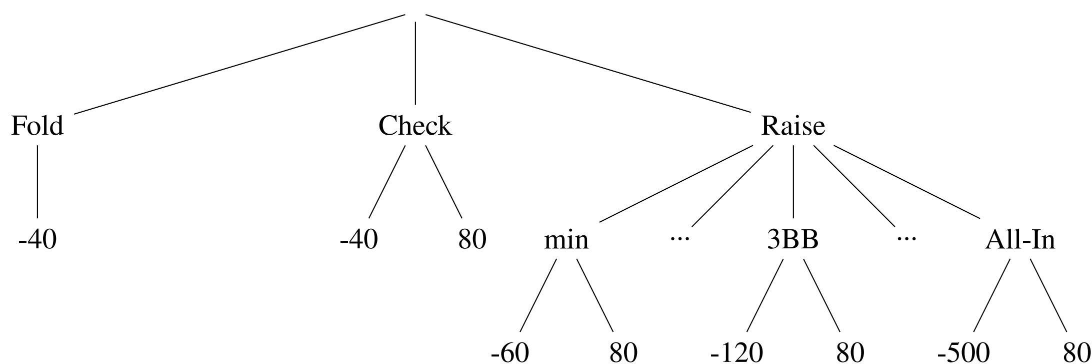

Figure 3.10: Decision Tree without adjusment

We solved this problem by introducing an adjustment matrix used to increase or decrease the utility of raises based on the raise amount and hand strength. Figure 3.11 shows how the raise amount and hand strength affect the adjusment factors. These factors are choosen to give a higher utility to big raises when the hand is strong and a lower utility for big raises when the hand is weak. Some rules have also been added to deter the bot from calling when its hand is weak. In addition, some limits on the raise amount have also been implemented in some cases. For example, the bot will not even consider raising more than half of its stack when holding a very weak hand. Finally, we utilized the

probabilistic nature of the decision object to prevent premature raises and to shape the bot's behavior to prefer slow playing, that was done by giving lower weights to the raise actions at the early stages of the game and by giving higher weights to the raise actions at the later stages of the game.

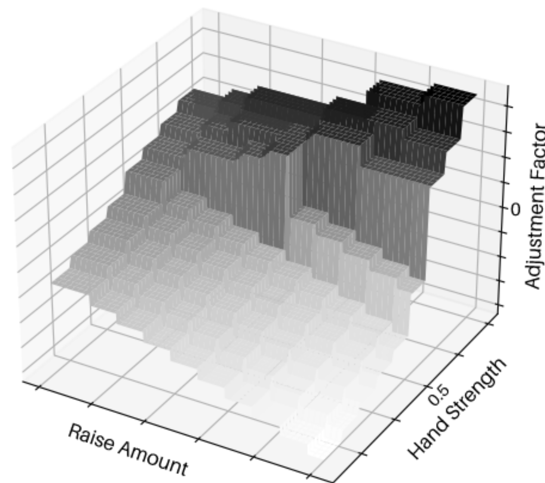

Figure 3.11: Adjustment Factors

3.5 Domain Model

To build an agent that can play poker, the agent has to be able to choose one of the three legal actions: *Fold*, *Call*, or *Raise*. For the raise action, the bot needs to choose also a raise amount, which has to be greater than the highest current bet. While it is possible to treat all game stages the same, we found it reasonable to split the domain into two main compound tasks. The first is responsible for the preflop stage, and the second for the post-flop stages.

The *Preflop Sequence* represents the task responsible for the preflop stage. The sequence includes three compound tasks. The first two tasks will produce a recommendation and convert it to a decision, as described in section 3.3. The last task represents the *Fish Net* discussed in section 3.3.4. The tasks responsible for producing a recommendation are grouped into four compound tasks based on the current position of PokerShark on the table, which is how the original guidelines were grouped in the first place. Each one of these tasks includes a number of compound and primitive tasks representing each guideline. The planner will choose which task to decompose based on a set of preconditions defined for each task. Appendix B includes a listing of all preconditions used by PokerShark. The *Preflop Sequence* does not use utility or expected utility in the decision-making process. However, the current risk attitude of the agent plays an important role in converting the recommendation into a decision, as described in section 3.3.2.

The *Postflop Sequence* is dynamically generated each time PokerShark is asked to make a decision. The sequence includes one compound task of the custom type *ExpectedUtilitySelector*, meaning this part of the domain is risk aware. For each possible action: Fold, Call and Raise with different bet sizes ranging from the current highest bet to the amount of the current stack, a primitive task of type *VariableUtilityTask* is created. Each one of these tasks gets a utility score assigned based on

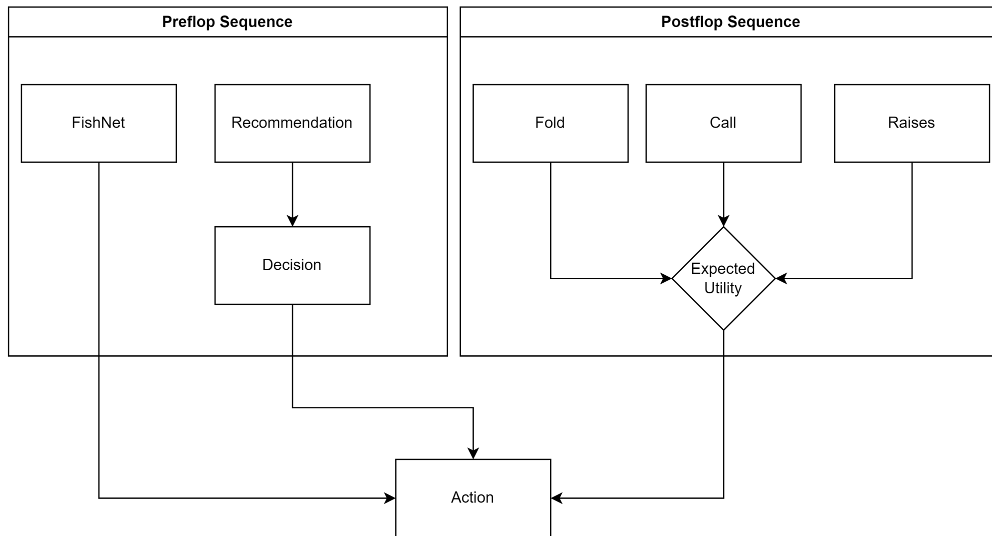

Figure 3.12: Overview of PokerShark domain model.

the current calling amount, money paid, and hand strength. This score is adjusted using the utility function corresponding with the current risk attitude of the agent. The utility score of raise actions is also adjusted based on the amount of the raise and the strength of the hand to ensure that the agent prefers a higher raise amount when it has a stronger hand.

The *FindBestTask* from section 3.4.7 is then used to choose the task with the highest utility. The result of decomposing the best task will be a decision vector. To ensure that the agent will often adopt a slow-playing strategy, the weights in the decision vectors of all raise actions are skewed to prefer calls over raises in the early stages of the round and the other way around in the later stages.

3.6 World State

The world state of PokerShark is a hash map that contains different objects, such as game and decision objects, in addition to different primitive variables and flags, such as the raise amount and the check-raise flag. It also contains a list of the current legal actions.

The game object is a complex object that contains all the information related to the game, such as the game rules, the number of players, the initial stack, the size of the ante and the blinds, and other related information. It also includes all rounds and action history, including player-related information such as name, stack size, position on the table, and the player model. Nevertheless, the game object and PokerShark have no access to the opponent's current hand; the game server handles this information, and the agent has no access to it. With each new event, the game server will send a notification to PokerShark, which will update the local game object with the new game state. This allows PokerShark also to update the opponent model after each action performed by the opponent.

The other objects in the world state are used to store information during the planning process. For example, the check-raise flag is used while performing the check-raise tactic, which continues throughout the duration of two stages.

3.7 Goal State

The way PokerShark is set up allows us to run the planner without the need to define a goal state. Instead, we run the planner inside a loop until a flag is set in the world state, which informs us that the planner has decided on an action. The planner requires theoretically only one iteration to reach a decision in the post-flop stage and two iterations in the preflop stage, one to find a recommendation and another to convert it into a decision. Running the planner inside a loop can be problematic, as it can lead to an infinite loop if the agent is in a situation where it cannot make a decision. To solve this problem, we have implemented a timeout mechanism that will stop the planner after a certain number of iterations. If the planner has not found a solution by then, the agent will Fold.

4 Evaluation

Evaluating the performance of a poker player is a challenging task. Unfortunately, the majority of strong AI poker players are not publicly available. Furthermore, conducting a big-scale experiment against human players is also not feasible, as the big online poker-playing sites prohibit the use of bots. To get an idea of the performance of PokerShark, we conducted a number of experiments where PokerShark played against dummy-AI player and human players. To study the effect of the risk attitude on the agent's behavior, we repeated the experiments where PokerShark played against dummy-AI players using different risk attitudes.

In this section, the graphs are used to demonstrate how PokerShark's winnings changed throughout the games. For each opponent, the graphs display three lines: the red line represents the number of chips won before the showdown (i.e., when one of the players has folded), the blue line shows the winnings only at the showdown, and the green line is the cumulative total of PokerShark's winnings. The y-axis of each graph shows the number of chips, and the x-axis represents the round number.

Another important metric we use to evaluate the performance of PokerShark is WPH, which represents the average amount a player wins in each game round measured in small blinds. It is calculated by dividing the total winnings by the number of rounds played times the small blind amount. For example, if the agent always folds, it would have a WPH of -1,5 small blinds per round.

4.1 Dynamic Risk Attitude

For the experiments in this section, we used the dynamic attitude discussed in section 3.3.1 and the corresponding utility functions discussed in section 3.4.7. To summarize the ideas discussed in the previously mentioned sections, PokerShark adopts a risk-neutral attitude most of the time. The agent changes its attitude toward risk as the number of chips in its stack changes. When the stack of chips starts getting smaller, the agent changes its risk attitude to be more risk-averse. If the stack becomes significantly bigger than the starting amount, then the agent adopts a more risk-seeking attitude.

Playing against Dummy-AI players

Dummy-AI players are AI players that use a simple static strategy to play poker. We used four such players: Bold, Fish, Honest and Random. The bold player always raises the maximum allowed amount regardless of his pocket. Similarly, the fish player always calls, whereas the honest player chooses the best action based on the estimated strength of his hand; finally, the random player

chooses a random action. At first glance, one might overlook the challenges imposed by these simple strategies. However, playing against these bots will provide a good chance to see how PokerShark will identify and exploit the weaknesses of each bot.

Bold Player

The bold player imposes an interesting problem because the bot will start each round by betting the entire stack; calling such a bet is very risky from the preflop. Looking at the graph shown in figure 4.1, we can see that most losses occur before the showdown because PokerShark is folding many weak hands. Although it is a very difficult challenge, PokerShark lost only 0.71 small blinds per hand. This is a very good result, considering that the bold player is a very loose and aggressive player.

Number of games	100	
Games won	38	38.0%
Games drew	0	00.0%
Games lost	62	62.0%
Number of rounds	3237	
Rounds won	110	30.4%
Rounds drew	0	00.0%
Rounds lost	3127	96.6%
WPH	-0.71	

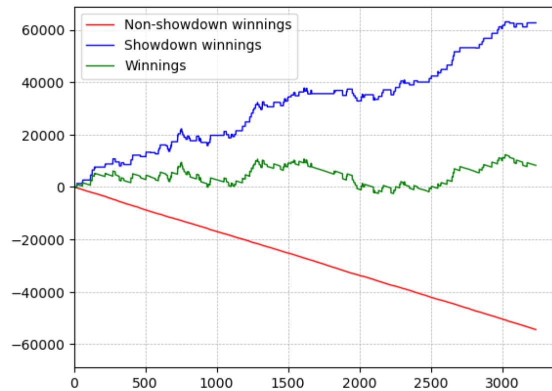

Figure 4.1: Results of playing 100 games against the Bold player.

Fish Player

Playing against the fish player is a good opportunity to showcase PokerShark's ability to exploit players' tendencies. For example, we can see from the Non-showdown winnings in figure 4.2 that PokerShark did not fold as much, meaning it has adjusted its folding range. We can also see that although PokerShark is calling/raising a lot more, it manages the risk by using small raises.

Number of games	100	
Games won	68	68.0%
Games drew	0	00.0%
Games lost	32	32.0%
Number of rounds	3385	
Rounds won	1511	44.6%
Rounds drew	21	00.6%
Rounds lost	1853	54.7%
WPH	1.10	

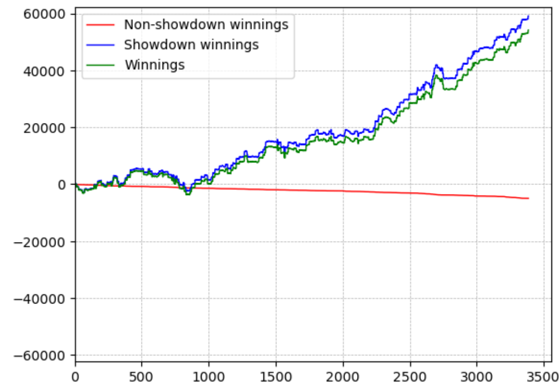

Figure 4.2: Results of playing 100 games against the Fish player.

Honest Player

Playing against a player that does not bluff is quite difficult because it folds when his hand is weak, and he only raises with a strong hand. Therefore, knowing when to challenge such a player is crucial to winning the game. Looking at the figure 4.3, we can see the games were more dynamic, with PokerShark successfully winning big bets but also lost many small bets with varying tolerance to call the raises of the honest player. Overall, We think PokerShark performed well against this player.

Number of games	100	
Games won	17	17.0%
Games drew	0	00.0%
Games lost	83	83.0%
Number of rounds	9340	
Rounds won	3670	39.3%
Rounds drew	3	00.6%
Rounds lost	5667	60.7%
WPH	-0.16	

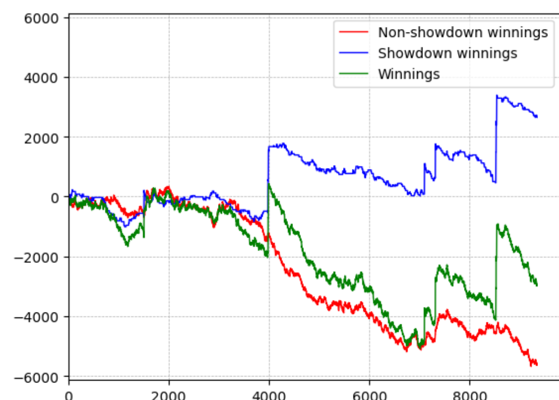

Figure 4.3: Results of playing 100 games against the Honest player.

Random Player

Facing a random opponent is a great way to see how PokerShark handles contradictory information while modeling its opponent. The graph in figure 4.4 shows how PokerShark used bigger betting amounts while also folding more often.

Number of games	100	
Games won	48	48.0%
Games drew	0	00.0%
Games lost	52	52.0%
Number of rounds	4540	
Rounds won	1276	28.1%
Rounds drew	0	00.6%
Rounds lost	3264	71.9%
WPH	-0.03	

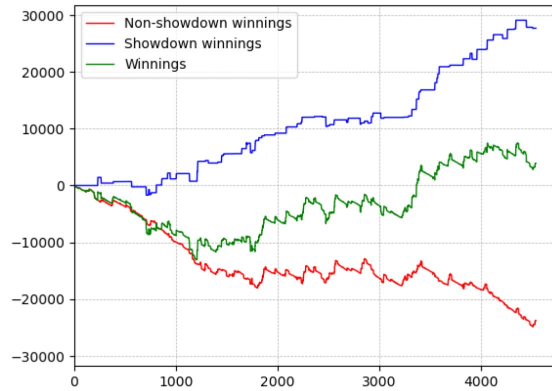

Figure 4.4: Results of playing 100 games against the Random player.

4.1.1 Playing against human players

Unfortunately, we could not organize a big experiment to test PokerShark's performance against human players. However, we were able to organize a few games with novice poker players. In total, PokerShark played a round thousand hands against human players.

Number of games	18	
Games won	10	55.6%
Games drew	0	00.0%
Games lost	8	44.4%
Number of rounds	955	
Rounds won	359	37.6%
Rounds drew	1	00.1%
Rounds lost	595	62.3%
WPH	0.19	

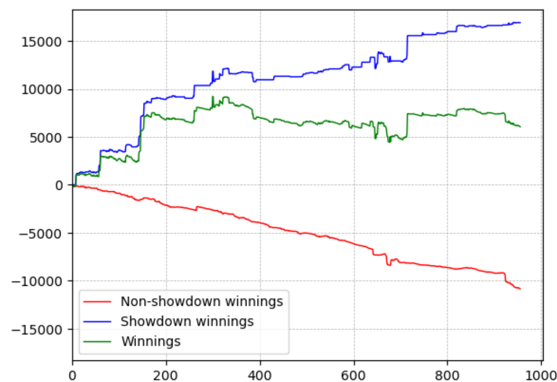

Figure 4.5: Results of around 1000 hands against human players.

Observing the graph shown in figure 4.5, it is clear that PokerShark forfeited a significant portion of its profits by folding. That is due to the majority of human players being able to quickly identify and exploit PokerShark's tendency to fold when calling is too risky. The big fluctuations in winnings indicate that PokerShark did not always fold but was also able to call the opponent's bluff and score big wins.

4.2 Static Risk Attitude

To investigate the impact of risk attitude on an agent's behavior, we reconducted the previous experiments in which PokerShark played against dummy AI players but this time using static risk attitudes.

The first set of experiments was run with the risk attitude of PokerShark being set to risk-seeking. Having a more risk-seeking attitude can change the behavior of a poker player in several ways. For one, a player who is more willing to take risks will be more likely to bet or raise with weaker hands since they are more willing to gamble in order to potentially win a larger pot. This can make them more unpredictable and difficult for their opponents to read. Additionally, risk-seeking players may also be more likely to bluff since they are more willing to risk betting with a weak hand to try and scare their opponents. Overall, a risk-seeking attitude should make PokerShark more aggressive and less predictable in its betting and playing style.

The second set of experiments was conducted with PokerShark having a risk-averse attitude. Having a risk-averse attitude can change the behavior of the agent, causing it to be more conservative in its betting and playing style. This should make PokerShark less likely to take risks, such as going all-in on a hand or making large bets, in order to avoid the potential loss of chips.

The last set of experiments was conducted using a risk-neutral attitude, which should make PokerShark play in a similar style to the Honest player. Only calling or raising when it thinks it has the strongest hand. The detailed analysis of the games conducted is provided in appendix C. We have also summarized the findings in table 4.1.

		Win	**Drew**	**Lost**	**WPH**
Bold Player	**Dynamic risk**	38.0%	0.0%	62.0%	-0.71
	Risk seeking	44.0%	0.0%	56.0%	-0.47
	Risk averse	49.0%	0.0%	51.0%	-0.04
	Risk neutral	40.0%	0.0%	60.0%	-0.57
Fish Player	**Dynamic risk**	68.0%	0.0%	32.0%	1.10
	Risk seeking	77.0%	0.0%	23.0%	1.70
	Risk averse	74.0%	0.0%	26.0%	1.20
	Risk neutral	76.0%	1.0%	23.0%	1.58
Honest Player	**Dynamic risk**	17.0%	0.0%	83.0%	-0.16
	Risk seeking	92.0%	1.0%	7.0%	0.40
	Risk averse	13.0%	0.0%	87.0%	-0.26
	Risk neutral	16.0%	0.0%	84.0%	-0.16
Random Player	**Dynamic risk**	48.0%	0.0%	52.0%	-0.03
	Risk seeking	55.0%	0.0%	45.0%	0.39
	Risk averse	27.0%	0.0%	73.0%	-0.67
	Risk neutral	16.0%	0.0%	84.0%	0.36

Table 4.1: Results of playing 100 games using the different static risk attitudes.

The different risk attitudes yielded varying results against different players. Overall, the dynamic attitude performed the worst on average, with risk-seeking outperforming all other attitudes. This could be due to the nature of the game, or perhaps the experiments we conducted favored this attitude. Table 4.2 shows how each attitude performed on average.

Attitude	**WPH**
Dynamic risk	0.05
Risk seeking	0.51
Risk averse	0.06
Risk neutral	0.30

Table 4.2: Average WPH for each risk attitude.

4.3 Self Play Experiments

Due to the findings of our earlier experiments, we have chosen to conduct additional self-play experiments in order to further investigate the effect of risk attitude on the performance of the agent. To evaluate the impact of different risk attitudes on performance more thoroughly, we set up a series of head-to-head games between multiple copies of PokerShark, each with a different risk attitude. We wanted to see how the different risk attitudes would affect the outcome of the games, so we had each copy play 100 games against each of the other copies. This was important to verify the findings of our previous experiments.

	Dynamic	**Risk-Seeking**	**Risk-Averse**	**Risk-Neutral**	**Average**
Dynamic	-	-0.33	0.01	0.02	**-0.10**
Risk-Seeking	0.33	-	0.37	0.36	**0.35**
Risk-Averse	-0.01	-0.37	-	-0.01	**-0.13**
Risk-Neutral	-0.02	-0.36	0.01	-	**-0.12**

Table 4.3: Results of self play with different risk attitudes; WPH of player at start of row

Table 4.3 shows the results of WPH for the player at the start of the row against the player at the start of the column. The results clearly show that the risk-seeking attitude outperforms all other attitudes. The dynamic risk attitude performed better than all other attitudes, which contradicts our previous experiments. However, it aligns with our initial intuition.

A larger experiment with human players would be required to make confident conclusions about the impact of risk attitude on performance. However, for now, we are led to believe that the domain of poker tends to favor risk-seeking players.

5 Related work

HTN planning is a powerful tool for representing and solving complex problems in a hierarchical and modular fashion that has been applied to a wide range of real-world domains. In the context of poker, however, and to the best of our knowledge, HTN planning techniques have never been used before. One of the key tools that were crucial to the development of PokerShark was the extension of the HTN planning framework to incorporate risk awareness and guide the planner using the agent's risk attitude. This approach was studied in detail in previous research [Aln19] [GL14] [AGA22] and provided a strong foundation for the development of PokerShark's risk-aware decision-making process.

Over the years, researchers and engineers have attempted to develop AI systems that can compete with human players in the game of poker. One of the first successful AI poker players was Loki and its successor Poki, which used a formula-based betting strategy and were the focus of several research papers [BDSS02] [BPSS98] [Pap98] that laid the foundation for later work, including the development of PokerShark. There were also other approaches that used different strategies, such as simulations, game theory strategies, and neural networks, with varying degrees of success. An extensive review of the different approaches is provided in [RW11]. One of the studies that are somewhat comparable to ours is provided in [Fol03], where a rule-based expert system (SoarBot) is analyzed, which has a similar structure to PokerShark, particularly in the way it relies on expert knowledge. However, SoarBot does not consider the risk of different actions and has no notion of utility or risk attitudes.

The topic of opponent modeling is closely related to the research presented in this thesis. Many approaches have been developed to model and predict opponents' behavior in poker, including the use of machine learning and neural networks. A comprehensive review of these approaches can be found in [Bou14]. Examples of successful approaches using machine learning and neural networks are discussed in [FKP12] and [YXYZ20].

Estimating hand strength is an important topic for developing a strong AI player. In [XSH+21], an algorithm for poker hand classification is provided that uses linear regression to calculate the approximate effective hand strength. A similar approach was also presented in [BEW13].

6 Conclusion and Future Work

Developing a strong AI poker player involves many challenges, including decision-making under uncertainty, risk management, game theory, and understanding the behavior and psychology of the opponents. PokerShark succeeded in overcoming some of these challenges resulting in a mediocre poker player with great potential for improvement. We have successfully modeled the domain using HTN planning constructs and integrated risk awareness into the planning process. Along the way, we have implemented guidelines provided by the domain experts and tried to create an agent that uses different metrics to estimate the cost of its actions. Despite our efforts to find a good balance while integrating risk awareness into the agent decision-making process, we found that our reliance on arbitrary thresholds based on intuition and limited knowledge of the game often led to suboptimal solutions. This was evident in the experiments we conducted, which showed that our dynamic risk attitude performed worse on average than other static attitudes. We have also resorted to using approximation and interpolation to estimate values that are computationally too hard to calculate. The opponent modeling mechanism that PokerShark uses relies on a small number of states, and it only delivers information about the ranges that the opponents like to play. However, it does not take into account important factors such as player position, pot size, and deception. Furthermore, the overall strategy of PokerShark is centered around hand strength, which might not be the best strategy for playing the game. Game theory is still unable to deliver an optimal strategy for poker, but we believe the strategy we have implemented can be vastly improved by adopting tactics that place a greater emphasis on deception and opponent psychology.

Despite the aforementioned shortcomings, the aim of this study was not to create the best poker player but rather to explore the potential of HTN planning in a complex domain such as poker. While PokerShark may not be the most skilled poker player, we believe it demonstrates the effectiveness of using HTN planning with risk awareness as a foundation for building powerful AI agents capable of playing poker efficiently.

There is much potential for PokerShark to improve by conducting more research and experiments. This would allow the current playing strategies to be refined and new tactics to be introduced. Additionally, incorporating other AI techniques would open the door to new exciting possibilities. Furthermore, incorporating improved opponent modeling techniques could allow PokerShark to better predict and adapt to the behavior of its opponents. This could give it a significant advantage in games against other players or AI opponents. Overall, PokerShark has a great deal of room to grow and improve through further research and development.

Bibliography

[AGA22] E. Alnazer, I. Georgievski, M. Aiello. “Risk Awareness in HTN Planning”. In: *arXiv preprint arXiv:2204.10669* (2022) (cit. on pp. 18, 32, 35, 67).

[AHK+98] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. SRI, A. Barrett, D. Christianson, et al. “PDDL| The Planning Domain Definition Language”. In: *Technical Report, Tech. Rep.* (1998) (cit. on p. 24).

[Aln19] E. Alnazer. “HTN Planning with Utilities”. 2019 (cit. on pp. 23, 27, 32, 67).

[BAH19] P. Bercher, R. Alford, D. Höller. “A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations”. In: *IJCAI* (Aug. 1, 2019), pp. 6267–6275. DOI: 10.24963/ijcai.2019/875 (cit. on pp. 27, 28).

[BDSS02] D. Billings, A. Davidson, J. Schaeffer, D. Szafron. “The challenge of poker”. In: *Artificial Intelligence* (2002). DOI: 10.1016/s0004-3702(01)00130-8 (cit. on pp. 17, 51, 52, 67).

[BEW13] J. Bensson, A. Eckert, M. Wu. “Predicting Texas Holdem Hand Strength”. In: (2013) (cit. on pp. 51, 67).

[Bil06] D. Billings. “Algorithms and assessment in computer poker”. In: (2006) (cit. on p. 17).

[Bou14] N. Boudewijn. “Opponent Modeling in Texas Hold’em”. B.S. thesis. 2014 (cit. on p. 67).

[BPSS98] D. Billings, D. Papp, J. Schaeffer, D. Szafron. “Poker as a Testbed for AI Research”. In: *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer. 1998, pp. 228–238 (cit. on pp. 17, 67).

[BS19] N. Brown, T. Sandholm. “Superhuman AI for multiplayer poker”. In: *Science* 365.6456 (2019), pp. 885–890 (cit. on p. 18).

[DBSS00] A. Davidson, D. Billings, J. Schaeffer, D. Szafron. “Improved Opponent Modeling in Poker”. In: (2000) (cit. on p. 52).

[EHN94] K. Erol, J. Hendler, D. S. Nau. “HTN planning: Complexity and expressivity”. In: *AAAI*. Vol. 94. 1994, pp. 1123–1128 (cit. on p. 18).

[ENS95] K. Erol, D. S. Nau, V. S. Subrahmanian. “Complexity, Decidability and Undecidability Results for Domain-Independent Planning”. In: *Artificial Intelligence* 76.1 (July 1, 1995), pp. 75–88. DOI: 10.1016/0004-3702(94)00080-k (cit. on p. 26).

[Fis68] P. C. Fishburn. “Utility theory”. In: *Management science* 14.5 (1968), pp. 335–378 (cit. on p. 30).

[FKP12] G. Fedczyszyn, L. Koszalka, I. Pozniak-Koszalka. “Opponent modeling in Texas Hold’em poker”. In: *International Conference on Computational Collective Intelligence*. Springer. 2012, pp. 182–191 (cit. on p. 67).

[Fol03] R. I. Follek. “SoarBot: A rule-based system for playing poker”. PhD thesis. Pace University, 2003 (cit. on p. 67).

[GGA14] I. Georgievski, I. Georgievski, M. Aiello. “An Overview of Hierarchical Task Network Planning”. In: *arXiv: Artificial Intelligence* (Mar. 28, 2014) (cit. on pp. 23, 27, 28).

[GL14] I. Georgievski, A. Lazovik. “Utility-based HTN planning”. In: *ECAI 2014*. IOS Press, 2014, pp. 1013–1014 (cit. on pp. 32, 67).

[GNT06] M. Ghallab, D. S. Nau, P. Traverso. “Automated Planning, Theory And Practice”. In: (Jan. 1, 2006) (cit. on p. 23).

[HBBB21] D. Höller, G. Behnke, P. Bercher, S. Biundo. “The PANDA Framework for Hierarchical Planning”. In: *Künstliche Intelligenz* (2021), pp. 1–6. DOI: 10.1007/s13218-020-00699-y (cit. on p. 26).

[HLM+19] P. Haslum, N. Lipovetzky, D. Magazzeni, Christian Muise, C. Muise. “An Introduction to the Planning Domain Definition Language”. In: *An Introduction to the Planning Domain Definition Language* (Apr. 2, 2019). DOI: 10.2200/s00900ed2v01y201902aim042 (cit. on pp. 23–25).

[JS09] U. Johansson, C. Sönströd. “Fish or Shark: Data Mining Online Poker”. In: *5th International Conference on Data Mining-DMIN 09, Las Vegas, USA*. CSREA. 2009 (cit. on p. 44).

[KS92] H. Kautz, B. Selman. “Planning as Satisfiability”. In: *ECAI* (Aug. 30, 1992), pp. 359–363 (cit. on p. 26).

[KT13] D. Kahneman, A. Tversky. “Prospect theory: An analysis of decision under risk”. In: *Handbook of the fundamentals of financial decision making: Part I*. World Scientific, 2013, pp. 99–127 (cit. on p. 35).

[Mal04] M. Malmuth. *Gambling theory and other topics*. Two plus two, 2004 (cit. on p. 22).

[MSB+17] M. Moravcik, M. Schmid, N. Burch, V. Lisy, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, M. Bowling. “Deepstack: Expert-level artificial intelligence in heads-up no-limit poker”. In: *Science* 356.6337 (2017), pp. 508–513 (cit. on p. 18).

[Pap98] D. Papp. “Dealing with imperfect information in poker”. In: (1998) (cit. on pp. 52, 54, 67).

[Roy21] W. Roya. “History of Poker”. In: *Card night: Classic games, classic decks, and the history behind them*. Running Press, 2021, p. 203 (cit. on p. 19).

[RW11] J. Rubin, I. Watson. “Computer poker: A review”. In: *Artificial intelligence* 175.5-6 (2011), pp. 958–987 (cit. on pp. 17, 67).

[SE75] E. D. Sacerdoti, Earl D. Sacerdoti. “The Nonlinear Nature of Plans”. In: (Sept. 3, 1975), pp. 206–214 (cit. on p. 26).

[Sha50] C. E. Shannon. “A chess-playing machine”. In: *Scientific American* 182.2 (1950), pp. 48–51 (cit. on p. 17).

[Skl03] D. Sklansky. *Hold ’em poker: For advanced players, 21st Century edition*. Two Plus Two Pub., 2003 (cit. on pp. 22, 44, 47).

[Tur53] A. M. Turing. “Digital computers applied to games”. In: *Faster than thought* (1953) (cit. on p. 17).

[Wil12] G. Williamson. *Frontier Gambling*. GR Williamson, 2012 (cit. on p. 19).

[Wil89] D. E. Wilkins. “Practical Planning: Extending the Classical AI Planning Paradigm”. In: (Jan. 14, 1989) (cit. on p. 26).

[XSH+21] Z. Xiaochuan, D. Song, Z. Hailu, L. He, W. Fan. “A method of computing winning probability for Texas Hold’em poker”. In: *2021 33rd Chinese Control and Decision Conference (CCDC)*. IEEE. 2021, pp. 1820–1824 (cit. on p. 67).

[Yan90] Q. Yang. “Formalizing Planning Knowledge for Hierarchical Planning”. In: 6.1 (Jan. 3, 1990), pp. 12–24. DOI: 10.1111/j.1467-8640.1990.tb00126.x (cit. on p. 26).

[YXYZ20] X. Yan, L. Xia, J. Yang, Q. Zhao. “Opponent Modeling in Poker Games”. In: *2020 IEEE 9th Data Driven Control and Learning Systems Conference (DDCLS)*. IEEE. 2020, pp. 1090–1097 (cit. on p. 67).

All links were last followed on November 25, 2022.

A Blocks World Domain

Blocks World is a toy problem used in automated planning research[1]. The domain consists of a table containing several blocks. Each block can be moved individually, and only the block on top of a stack can be moved. Listing A.1 and A.2 shows the domain and problem defination of Blocks World in PDDL from the second international planning contest[2].

Listing A.1 Blocks world domain definiton.

```
(define (domain BLOCKS)
    (:requirements :strips :typing) (:types block)
    (:predicates (on ?x - block ?y - block) (ontable ?x - block)(clear ?x - block)
             (handempty)(holding ?x - block))
    (:action pick-up
           :parameters (?x - block)
           :precondition (and (clear ?x) (ontable ?x) (handempty))
           :effect (and (not (ontable ?x))(not (clear ?x)) (not (handempty))(holding ?x)))
    (:action put-down
           :parameters (?x - block) :precondition (holding ?x)
           :effect (and (not (holding ?x))(clear ?x)(handempty)(ontable ?x)))
    (:action stack
           :parameters (?x - block ?y - block)
           :precondition (and (holding ?x) (clear ?y))
           :effect (and (not (holding ?x))
             (not (clear ?y))(clear ?x)(handempty)(on ?x ?y)))
    (:action unstack
           :parameters (?x - block ?y - block)
           :precondition (and (on ?x ?y) (clear ?x) (handempty))
           :effect(and (holding ?x)(clear ?y)(not (clear ?x))
             (not (handempty))(not (on ?x ?y)))))
```

Listing A.2 Blocks world problem definiton.

```
(define (problem BLOCKS-4-0)
    (:domain BLOCKS) (:objects D B A C - block)
    (:INIT (CLEAR C) (CLEAR A) (CLEAR B) (CLEAR D) (ONTABLE C) (ONTABLE A)(ONTABLE B)
        (ONTABLE D) (HANDEMPTY))
    (:goal (AND (ON D C) (ON C B) (ON B A))))
```

[1] https://en.wikipedia.org/wiki/Blocks_world
[2] https://ipc00.icaps-conference.org/

B PokerShark Preconditions

IsAggressive	Is the game dominated by aggrissive players
IsLoose	Is the game dominated by loose players
IsPassive	Is the game dominated by passive players
IsTight	Is the game dominated by tight players
OneOpponent	Is there only one active player left
PocketEquityGreaterThan	Does the hand of the bot have more equity than the given number
PocketEquityLessThan	Does the hand of the bot have less equity than the given number
PocketFromGroup	Does the hand of the bot belongs to the given group
PocketIsOneOf	Does the hand of the bot one of the hands given
InBigBlind	Does the bot sit in the big blind seat on the table
InBlind	Does the bot sit in a blind seat on the table
InEarly	Does the bot have an early position
InLate	Does the bot have a late position
InMiddle	Does the bot have a middle position
InSmallBlind	Does the bot sit in the small blind seat on the table
FirstToPot	Nobody has called or raised yet.
FirstToRaise	Nobady has raised yet.
OnlyCalls	Somebody has already called, but no raises were made yet.
SecondToRaise	One player has already raised.
ThreeBBCall	It costs three times the bigblind to call
TwoOrMoreRaises	There is at least more than one raise
ZeroCostCall	It costs nothing to call
CallRecommendation	The planner has produced a call recommendation
FoldRecommendation	The planner has produced a fold recommendation
RaiseRecommendation	The planner has produced a raise recommendation
NoRecommendation	The planner has not produced any recommendation yet
NotTooRiskyMinRaise	It is not risky to raise the minimum legal amount
RiskAverseAttitude	The bot has a risk-averse attitude
RiskNeutralAttitude	The bot has a risk-neutral attitude
RiskSeekingAttitude	The bot has a risk-seeking attitude
TooRiskyCall	It is to risky to call
TooRiskyRaiseRecommendation	It is too risky to raise the recommended amount
InPreflop	The game is in the preflop stage
NoDecision	The planner has not produced a decision yet
Occasionally	Randomly true less than 30% of the time
RaiseOrCallDecision	Did the planner produce a raise or a call decision
TooFishy	Is the opponent too loose
CallingFish	Is the opponent aggrisve loose
LooseRaiser	The raising player is too loose

C Different Risk Attitudes Experiments

C.1 Static Risk Attitude - Risk Seeking

Bold Player

Number of games	100	
Games won	44	44.0%
Games drew	0	00.0%
Games lost	56	56.0%
Number of rounds	2492	
Rounds won	133	5.3%
Rounds drew	3	0.1%
Rounds lost	2356	94.5%
WPH	-0.47	

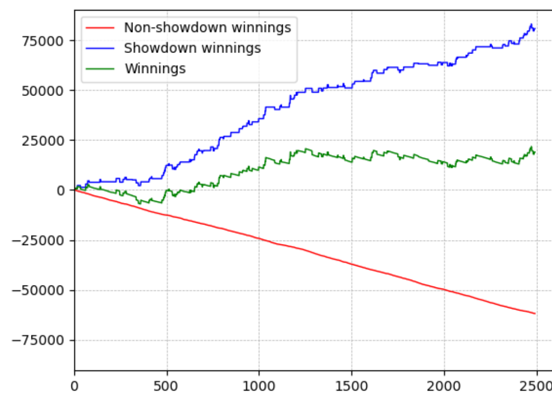

Figure C.1: Results of playing 100 games against the Bold player - With Risk Seeking attitude.

Fish Player

Number of games	100	
Games won	77	77.0%
Games drew	0	00.0%
Games lost	23	23.0%
Number of rounds	3164	
Rounds won	1522	48.1%
Rounds drew	21	0.7%
Rounds lost	1621	51.2%
WPH	1.7	

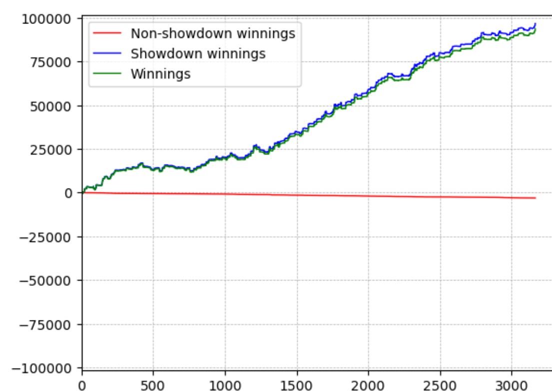

Figure C.2: Results of playing 100 games against the Fish player - With Risk Seeking attitude.

Honest Player

Number of games	100	
Games won	92	92.0%
Games drew	1	1.0%
Games lost	7	7.0%
Number of rounds	9631	
Rounds won	5483	56.9%
Rounds drew	9	0.1%
Rounds lost	4139	43.0%
WPH	0.4	

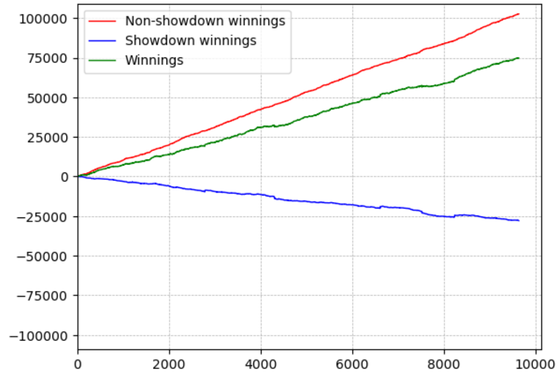

Figure C.3: Results of playing 100 games against the Honest player - With Risk Seeking attitude.

Random Player

Number of games	100	
Games won	55	55.0%
Games drew	0	0.0%
Games lost	45	45.0%
Number of rounds	2463	
Rounds won	928	37.7%
Rounds drew	1	0.0%
Rounds lost	1534	62.3%
WPH	0.39	

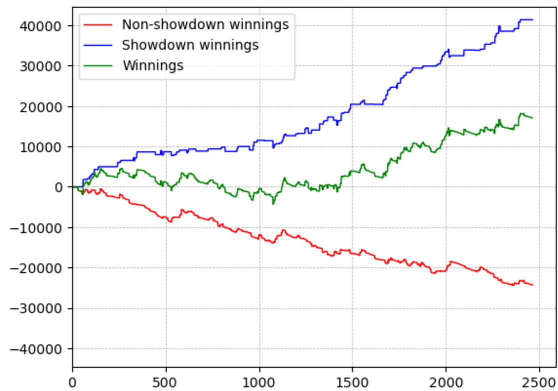

Figure C.4: Results of playing 100 games against the Random player - With Risk Seeking attitude.

C.2 Static Risk Attitude - Risk Averse

Bold Player

Number of games	100	
Games won	49	49.0%
Games drew	0	00.0%
Games lost	51	51.0%
Number of rounds	3382	
Rounds won	139	4.1%
Rounds drew	0	0.0%
Rounds lost	3243	95.9%
WPH	-0.04	

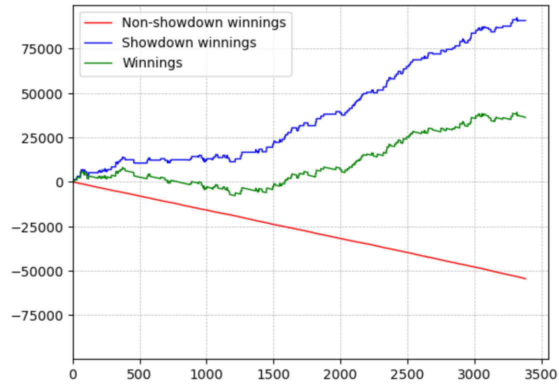

Figure C.5: Results of playing 100 games against the Bold player - With Risk Averse attitude.

Fish Player

Number of games	100	
Games won	74	74.0%
Games drew	0	00.0%
Games lost	26	26.0%
Number of rounds	3764	
Rounds won	1706	45.3%
Rounds drew	18	0.5%
Rounds lost	2040	54.2%
WPH	1.2	

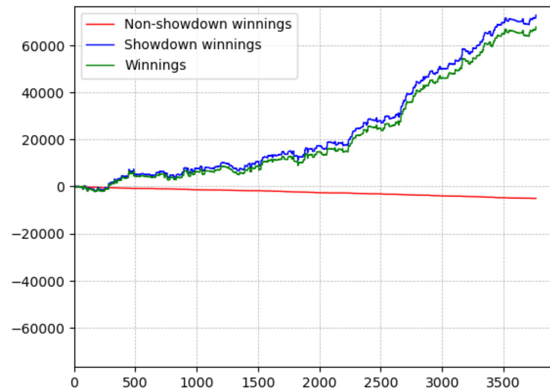

Figure C.6: Results of playing 100 games against the Fish player - With Risk Averse attitude.

Honest Player

Number of games	100	
Games won	13	13.0%
Games drew	0	00.0%
Games lost	87	87.0%
Number of rounds	9537	
Rounds won	3741	39.2%
Rounds drew	3	0.0%
Rounds lost	5793	60.7%
WPH	-0.26	

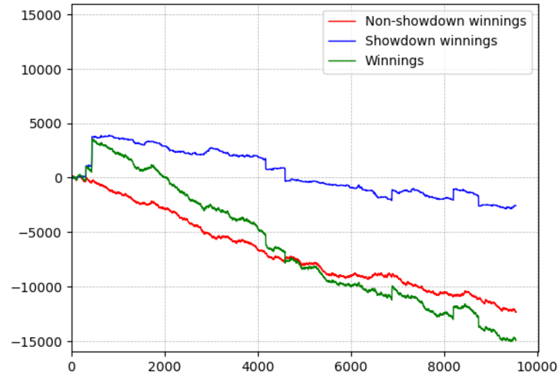

Figure C.7: Results of playing 100 games against the Honest player - With Risk Averse attitude.

Random Player

Number of games	100	
Games won	27	27.0%
Games drew	0	00.0%
Games lost	73	73.0%
Number of rounds	6385	
Rounds won	1652	25.9%
Rounds drew	0	0.0%
Rounds lost	4733	74.1%
WPH	-0.67	

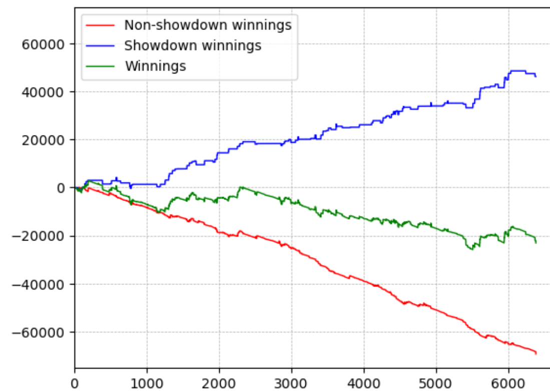

Figure C.8: Results of playing 100 games against the Random player - With Risk Averse attitude.

C.3 Static Risk Attitude - Risk Neutral

Bold Player

Number of games	100	
Games won	40	40.0%
Games drew	0	00.0%
Games lost	60	60.0%
Number of rounds	3162	
Rounds won	119	3.7%
Rounds drew	2	0.1%
Rounds lost	3041	96.2%
WPH	-0.57	

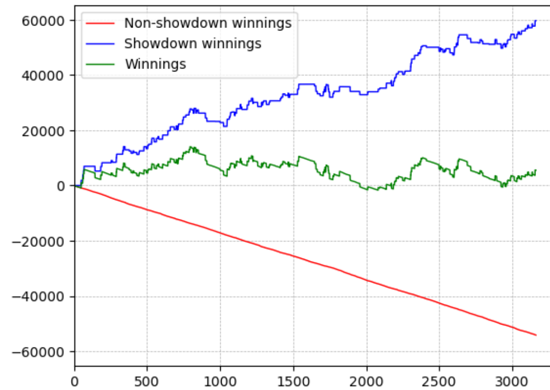

Figure C.9: Results of playing 100 games against the Bold player - With Risk Neutral attitude.

Fish Player

Number of games	100	
Games won	76	76.0%
Games drew	1	1.0%
Games lost	23	23.0%
Number of rounds	3372	
Rounds won	1578	46.8%
Rounds drew	18	0.5%
Rounds lost	1776	52.7%
WPH	1.58	

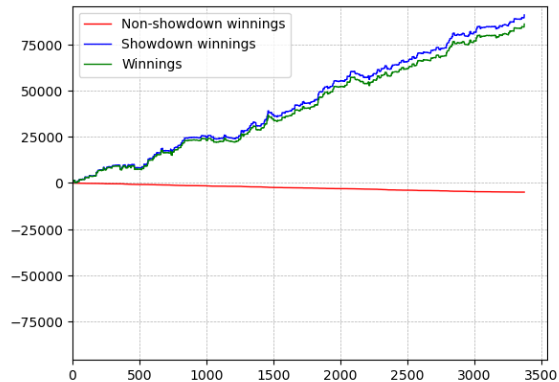

Figure C.10: Results of playing 100 games against the Fish player - With Risk Neutral attitude.

Honest Player

Number of games	100	
Games won	16	16.0%
Games drew	0	0.0%
Games lost	84	84.0%
Number of rounds	9579	
Rounds won	3866	40.4%
Rounds drew	4	0.0%
Rounds lost	5709	59.6%
WPH	-0.16	

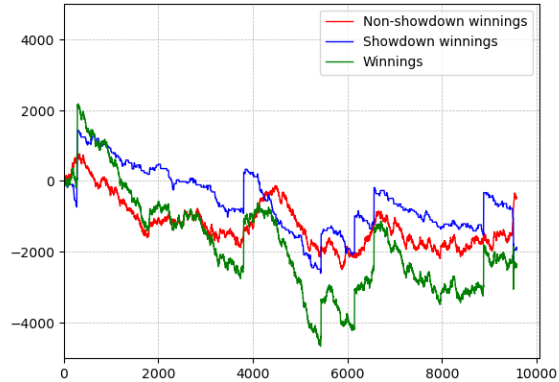

Figure C.11: Results of playing 100 games against the Honest player - With Risk Neutral attitude.

Random Player

Results against: Random number of games: 100 number of games won: 58 58.0number of games drew: 0 0.0number of games lost: 42 42.0number of rounds: 4648 number of rounds won: 1328 28.6number of rounds drew: 0 0.0number of rounds lost: 3320 71.4starting stack: 100000 end stack: 116767.0 win/loss: 16767.0

Number of games	100	
Games won	58	16.0%
Games drew	0	0.0%
Games lost	42	84.0%
Number of rounds	4648	
Rounds won	1328	40.4%
Rounds drew	0	0.0%
Rounds lost	3320	59.6%
WPH	0.36	

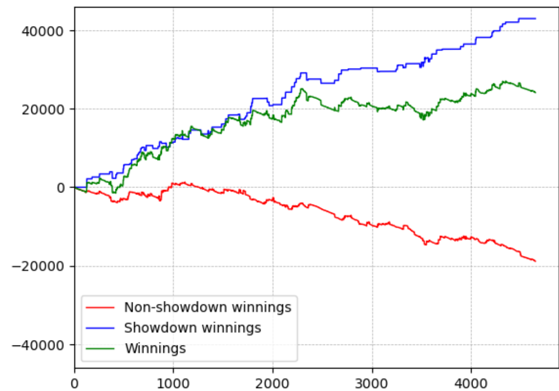

Figure C.12: Results of playing 100 games against the Random player - With Risk Neutral attitude.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.