Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Risk Awareness in Poker Planning Agents

Robin Smith

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	M.Sc. Ebaa Alnazer
Commenced:	November 21, 2022
Completed:	Mai 21, 2023

Abstract

The research of games has proven to be an ideal test bed for development in the artificial intelligence field. By solving different games, most prominently chess and checkers, researchers were able to develop, and improve important ideas and algorithms, within a closed and well-defined domain structure, which then could be translated to the real world. The approaches and techniques for these games of perfect information, however, did not always translate perfectly to real life domains. One important factor which was not covered is the notion of risk and uncertainty, that comes with most decisions in real environments. To tackle these problems, a different class of game, where randomness and unpredictability plays a big role, must be used to study decision making. Poker is one of those games, as it builds upon the principle of incomplete information along with possibilities for deception. Poker in general, and especially the variant No Limit Texas Hold'em Poker, has proven to be a valuable yet challenging domain to solve, attracting the attention of many researchers. Multiple approaches for AI powered playing have been taken, resulting in the creation of poker playing agents which are on par with professional poker players.

In this thesis we want to explore the possibilities of a new way of approaching the complex problem of No Limit Texas Hold'em Poker, by applying a risk-aware HTN planning technique, judging and evaluating the effectiveness of this approach for this very demanding domain in particular.

Kurzfassung

Die Untersuchung von Spielen hat sich als eine ideale Testumgebung für Entwicklungen im Bereich der Künstlichen Intelligenz herausgestellt. Durch das vollständige Lösen von verschiedenen Spielen, am bekanntesten Schach und Dame, konnten Forscher wichtige Ideen und Algorithmen innerhalb einer geschlossenen und gut strukturierten Domäne entwickeln, welche sich dann auf die echte Welt übertragen ließen. Einige dieser Ansätze für Spiele, die auf perfekten informationen beruhen lassen sich allerdings nicht einwandfrei übertragen. Ein wichtiger Faktor, der dort nicht berücksichtigt wurde, ist das Konzept des Risikos und der Ungewissheit, die mit den meisten Entscheidungen einhergeht. Um diese Probleme anzugehen, muss eine andere Klasse von Spielen untersucht werden, in denen Zufälle und Unvorhersehbarkeit eine große Rolle spielen. Poker ist eines dieser Spiele, da es auf den Prinzipien der unvollständigen informationen, sowie der Täuschung beruht. Das Spiel hat sich als wertvolle, aber schwierige Umgebung herausgestellt, die die Aufmerksamkeit vieler Wissenschaftler auf sich zieht. Mehrere Ansätze für KI gesteuertes spielen wurden genutzt, was zu der Erschaffung von Computer Poker Spielern führte, die auf dem selben Niveau wie professionelle Spieler stehen. In dieser Arbeit wollen wir the Möglichkeiten einer neuen Art das komplexe Problem das No Limit Texas Hold'Em Poker darstellt erforschen, indem wir eine risiko-bewusste HTN planungs Technik anwenden, und die effektivität dieser in der herausfordernden Poker Umgebung bewerten.

Contents

List of Figures

List of Tables

Listings

List of Algorithms

1 Introduction

The interest in Artificial Intelligence has seen a big rise in recent years. Many sectors of society have been increasingly affected and are slowly transforming to incorporate AI into our day to day life. AI powered algorithms are allowing autonomous vehicles to safely navigate roads and highways, financial marked data is being analyzed by AI, leading to automated trades and healthcare patient data is used to develop personal treatment, without seeing a human doctor.
Recently, the language model "ChatGPT", developed by OpenAI has made a big impact on society and politics, causing a great amount of discussion. With the ability to create human-like text to seemingly every topic, this AI based on language processing, has many people in awe, but also raises concerns about the dangers, as well as ethical problems coming from this technology. While some people now see AI as the technology of the future, which will allow humans to focus on their strength of creativeness and thought, others fear the implications of having Artificial Intelligence take over work which previously had to be done manually.
This development of AI, having the potential of becoming a big factor in our regular life, has been made possible by many years of extensive research and innovation.
Games have been one of the driving forces in AI for a long time, dating back as far as Alan Turing, who developed one of the first ever chess algorithms, "Turochamp" in 1948. While he started development of a computer program implementing his idea, he could not finish his work, as the available hardware was not powerful enough for such a complex program [Tur53]. The development on games continued over the years. With rapid improvements to algorithms and hardware, soon the first AI programs which could play complete games were created. One of the pioneers in this area is Cristopher Strachey, who is credited with the creation of one of the earliest AI programs capable of playing a full game of checkers. Overall, the research has been very successful, with many games today being solved to such a degree, that Artificial Intelligence can beat even the best human players.
What makes games an interesting environment, often allowing the results of research to be extended to real life domains, comes down to multiple beneficial characteristics. Games follow a specific set of rules and have a well-defined goal and different game scenarios are easily reproducible, which makes comparing different approaches or implementations easy, just to name a few examples for beneficial criteria. While chess and checkers pioneered game research, Poker has also been identified to be a beneficial research topic, due to multiple properties, which other games don't, or only partially cover [BPSS98b]. The principle of imperfect information and deception coupled with strategy makes it a particularly interesting, yet very challenging testing environment. Conclusions which could be drawn from poker can prove beneficial to dealing with misinformation, incomplete information, as well as risk management, which is particularly interesting for our work.
Different approaches have been presented and implemented, aiming to create the best poker playing agent, resulting in programs which are on par even with professional poker players.
We aim to use the established testing ground of poker to research the capabilities of a Risk Aware HTN Planning approach. Just like Games, HTN Planning is a widely studied topic in AI. One of the earliest notions of a hierarchical task network, where complex tasks are decomposed into smaller sub-tasks, was proposed by McDermott in 1982 [McD82]. This work laid the foundation for

further research in this field, causing several researchers to extend the HTN planning framework and develop increasingly more sophisticated planning algorithms in the following years. One notable example of a HTN Planner developed is the SHOP system. Using a combination of heuristic search and constraint propagation, it was able to solve complex problems, and efficiently generate plans for any domain [NCLM99]. In recent years, efforts have been made to extend HTN Planning by the notion of utility, which estimates the resource consumption of each task, to find optimal plans [Aln19][GL14]. These utility based HTN Planning Algorithms are closely related to our research topic, risk aware HTN Planning. The framework for this has been established in [AGA22]. In the following we want to explore the effectiveness of this technique in the domain of poker, by creating a model of the game within the given framework, generating plans and comparing those to established strategic advice.
The remainder of this thesis is organized as follows. The first chapter we introduce various background concepts, which provide necessary information for the following study. This includes an introduction to Poker, and Poker Strategy along with different ways to analyze the current state of the game, as well as an overview of the important concepts of HTN planning. The Following chapter describes how we modeled the domain of poker into an HTN planning problem, along with the challenges and problems we faced, and how we chose to solve them. The next section we take a look at our implementation for the risk aware HTN planner. Finally, the last chapter provides an evaluation of effectiveness for our model, comparing the generated plans to the proposed poker strategy, followed by a conclusion.

2 Background

2.1 Introduction to No Limit Texas Hold’em

2.1.1 Game Rules

Among the many different variations of Poker, No Limit Texas Hold’em is the most played and popular variant. Not only is it the standard in almost all casinos, and the game most associated with poker, but it is also the ruleset the World Series of Poker, the most prestigious tournament in the Poker world uses.

Texas Hold’em is played with a standard 52-card deck, usually with up to 9 players on one table. In most tournaments, each player starts with the same amount of chips, which is the currency used to buy into rounds or bet with. Once a player loses all his chips, he is eliminated from the tournament. To win chips, multiple game rounds, also called “hands” are played. At the start of each hand, two players must make a mandatory bet of a specified amount of chips, the so called “big blind” and the “small blind”. The size of the two blinds increases over the course of the tournament, with the small blind usually equating to half of the big blind. The players having to pay the blinds, as well as the sequence each player must do their turn in depends on the position at the table, which rotates after every round is over. At a full 9 player table, the order of action, as well as their names are illustrated in Table 2.1.

Each hand can be separated into different stages, or “streets”, with each stage including a betting round, where each player must choose the action, he wants to take. The following actions are allowed:

Abbreviation	Full Name
UTG	Under-the-Gun
UTG+1	
MP	Middle Position
MP+1	
HJ	Hijack
CO	Cutoff
BTN	Button
SB	Small Blind
BB	Big Blind

Table 2.1: Positions on a full 9 handed table

a) **“raise”** - A player can choose to put chips into the pot, to “raise”. Whenever one player raises, each opponent who wants to continue playing in this round must pay at least the same amount of chips. In No Limit Texas Hold’em, this raise must be at least double the amount of the current highest bet and has no upper limit to it.

b) **“check”** - If a player does not want to put any chips into the pot, he can check to skip his turn. Checking is only possible if there were no previous raises.

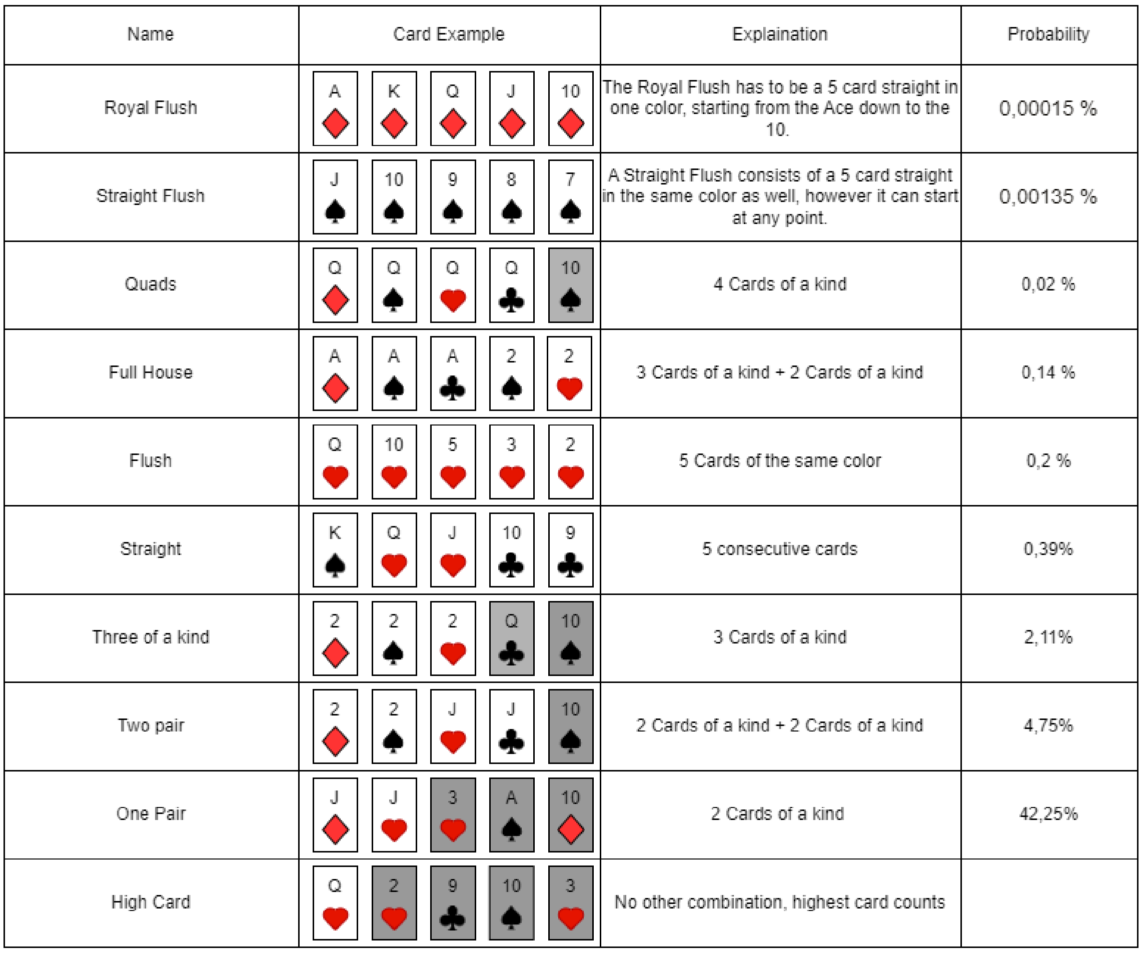
c) **“call”** - If there was a raise from a previous position, a player can choose to put in the same amount of chips as the raising player, this is called “calling” the bet.

d) **“fold”** - If a player does not wish to continue playing a hand, he can choose to “fold”. This concedes his cards and takes him out of the current hand.

Each betting round lasts either until there is only one player left, who has not folded his hand, or until every player has gotten to play one turn and has put in the same amount of chips. One exception to this is a player going “All in”, which means he bets all the chips he has available. In this case, this players turn gets skipped in every street, as he cannot do any more betting, he is however eligible to win the round in the end, and gain chips up to equal to his investment. A hand starts with the “Preflop” stage, where every player gets dealt two cards face down, the so called “hole cards”. After the betting round is over the “flop” occurs. Note that because the small blind and the big blind are considered bets, a player can only get to the flop if he put at least as many chips into the pot as the big blind. On the flop the dealer draws three cards and puts them on the table for every player to see. These open cards, as well as all following cards drawn this way are called “community cards”. Following is the “Turn” where one additional community card gets drawn, and lastly the “River”, with the last card draw, resulting in 5 community cards on board. After the final round of betting following the drawing of the fifth card has completed, and there are still two or more players who have not folded their hands left, the “Showdown” occurs. In the Showdown each Player shows his hand and it is determined who won this hand, and thus gains all the chips in the pot. To evaluate a Players hand, the five best cards, both from the players hole cards, as well as the 5 community cards in the middle of the table are considered. The ranking of poker hands with a short explanation, as well as the probability for each constellation can be seen in Figure 2.1

2.1.2 Strategy

The Game of Poker contains many random elements the player has no influence of. It is impossible to accurately predict the cards which are drawn, and nearly impossible to be certain of the cards our opponents hold. The success of a player is very dependent on how good the hands are he is dealt. Additionally, to win a lot of chips in a round, the opponents got to have a good hand themselves, to be inclined to put their money into the pot in the first place. However, in contrast to most other casino games, poker is a skill-based game where better players with superior skills are expected to win in the long run [Bje10]. Strategies have changed and evolved over the years. The increasing availability of learning resources, as well as avenues for discussions in the form of online chat rooms and forums has changed the average poker player, as well the game played.The general result of the evolution many experts seem to agree on, is that for one, players are a lot more aggressive in their playstyle than before, as well as a lot more focused on the math behind poker [Pok21]. We will base our evaluation on the strategy guidelines Dan Harrington proposes in [DB14]. He describes the very basic concepts, that every playstyle and strategy revolve around in four basic principles



Name	Card Example	Explaination	Probability
Royal Flush	A♦ K♦ Q♦ J♦ 10♦	The Royal Flush has to be a 5 card straight in one color, starting from the Ace down to the 10.	0,00015 %
Straight Flush	J♠ 10♠ 9♠ 8♠ 7♠	A Straight Flush consists of a 5 card straight in the same color as well, however it can start at any point.	0,00135 %
Quads	Q♦ Q♠ Q♥ Q♣ 10♠	4 Cards of a kind	0,02 %
Full House	A♦ A♠ A♣ 2♠ 2♥	3 Cards of a kind + 2 Cards of a kind	0,14 %
Flush	Q♥ 10♥ 5♥ 3♥ 2♥	5 Cards of the same color	0,2 %
Straight	K♠ Q♥ J♥ 10♣ 9♣	5 consecutive cards	0,39%
Three of a kind	2♦ 2♠ 2♥ Q♣ 10♠	3 Cards of a kind	2,11%
Two pair	2♦ 2♠ J♥ J♣ 10♠	2 Cards of a kind + 2 Cards of a kind	4,75%
One Pair	J♦ J♥ 3♥ A♠ 10♦	2 Cards of a kind	42,25%
High Card	Q♥ 2♥ 9♣ 10♠ 3♥	No other combination, highest card counts	

Figure 2.1: Overview of Hand Rankings

- **Strength** - "In general you want to bet your strong hands, check or call with your hands of middling strength, and fold your weak hands" [DB14].

  This very straight forward rule describes the very basic of Poker play. If you get dealt a strong hand, you want to put chips into the pot, hoping to get called by other players to win their money. Weak hands don't have any showdown value most of the time, so they should be thrown away immediately without losing any chips. For medium strength hands it is not as clear how to play. However, in general calling or checking should be the preferred actions. One might make an argument for betting with these hands as well, because they do have showdown value and might win the pot. Putting in chips with a medium strength hand however is very risky, and most of the time does not lead to additional winnings. This is because opponents with a weak hand will always fold to a bet, so there are no more chips to gain from them, while opponents with strong hands will most likely call, or even raise. This puts us in a bad situation, where we don't have much of a chance of winning at showdown, and we have already invested more chips than needed into the pot. This however, does not mean, that we can simply divide every hand in one of these categories and play them according to this rule. This would make our play very easily exploitable, and we would lose a lot against better players. There are many ways to diversify our game, while still following the strategy.

One core technique for this would be the “bluff”. When bluffing, we pretend to have a strong hand, by betting and raising, even though we have a weak hand. Adding this to our playbook, the opponent now must guess if we really have a strong hand, or if we are just bluffing when we are putting money into the pot.

- **Aggression** - “In general, aggression (betting and raising) is better than passivity (checking and calling)” [DB14]

  To understand this rule, we must look at the ways for us to win a hand. The obvious way of winning is to have the best hand at showdown. However, good poker players only take their hand to showdown about a quarter of the time and win their hands most often by forcing all their opponents to fold [**bragonier2010statistical**]. By playing passively and only ever checking and calling we completely disable this win condition. Additionally aggressive play makes us take control of the game, which often allows us to avoid difficult decisions, and force our opponents into undesirable positions.

- **Betting** “In general, a good bet should do one of three things:

  1. Force a better hand to fold.
  2. Force a weaker hand to call.
  3. Force drawing hands to put more money in the pot to see another card.

  ”[DB14]

  While we previously stated that betting and aggression is usually better than playing passively, this rule helps us to understand that we should not always bet very high, if at all. The first variation of a bet is what we call a “bluff”. Here we want to bet, even though we know our opponents have better hands than us. Because of our weak hand, the chance we are winning at showdown is very low, so our only option is either folding our hand or forcing every opponent out. It might seem like the best strategy in this case would be to bet the maximum amount possible, to put maximum amount of pressure on the opponents, this however comes with an increased risk, and smart opponents might be able to see through this deception, calling our bet and thus making us lose our whole stack. In the second variation, we know we have the best hand at the table. At this point it is important to get the maximum value out of this round. If we immediately start betting very high, our opponents will know that they can’t continue playing their hand for profit, and will fold early, giving us no chance of further increasing our winnings. Instead, we want to bet just the right amount to give the other players the right price to continue playing, and maybe even raising the pot themselves. There might even be situations where we don’t want to bet at all, hoping our opponents take the betting lead, giving us a very strong position to continue playing. The third type of bet applies to situations, in which we currently hold the best hands, but there is a high probability of our opponents gaining the upper hand with more cards on the board. This is a situation where we either want to force the opponents out of the round immediately or have them take the risk of paying chips in this round, even though they know they are behind. By betting these hands high, we can either ensure we are winning the pot at this spot already, win it in a later round if the opponents don’t hit their draw with a higher profit for us, or lose the round with higher costs.

- **Deception** “In general,you never want to do anything all of the time.” [DB14]
  This rule is pretty straight forward, and very important when playing against good opponents. Whenever we do an action, we give our opponents more information about our current hand, and our general playstyle. Highly skilled players take every action we took in this, as well as previous rounds into account, when making their decisions. This is why we should change up our actions, to stay less predictable. A good method for this, used by many professional players is to give each possible action a weighted random distribution, with higher probability, the better the action seems at the time, followed by randomly rolling which action to take.

2.2 Evaluation Metrics

Whether a human is playing a game of poker, or an Planning Algorithm is generating a plan, we need a way to evaluate the current situation of the game, in order to make an informed decision on the best possible action to take. For this we will define multiple metrics, used to evaluate the current situation on the board. The following ideas and algorithms, were adopted from [BDSS02]. We can split these in two main categories, **Hand Assessment Metrics** and **Pot Assessment Metrics**

2.2.1 Hand Assessment

Immediate Hand Strength

Without any additional information about opponents, one can calculate the Immediate Hand Strength(IHS) of the two hole cards he is dealt along with the community cards on board, which corresponds to the probability that the hand is best among all active opponents. For this we need to enumerate all the possible holdings, an opponent can have and counting the number of times our hand is better, worse or equally as strong as the opponents. This gives us the formula for our immediate hand rank as follows:

$$IHS = \frac{wins + \frac{ties}{2}}{wins + ties + losses}$$

Which could be computed by using the Algorithm 2.1

If there are no community cards on board yet, i.e. we are in the Preflop stage of the game, there are $\binom{52}{2} = 1326$ possible hand combinations, many of these can however be put into the same strength category, as their specific suit does not matter, as with only two cards we can either have cards of the same suit, or cards of different suits. This leaves us with 169 hand combinations to consider. This is significantly less than on the other streets, e.g the Flop, where we hold 2 cards, and there are 3 cards on the board, leaving 47 in the deck. This means our opponents can hold $\binom{47}{2} = 1081$ possible hands. If there are multiple opponents left, we have to raise our resulting percentage to the power of the number of opponents [DBSS00]. This already shows us, that the more opponents are left in a hand, the more our chance of holding the best hand decreases, which will be an important consideration later. The immediate hand strength metric gives us a quick and easy way to roughly estimate the strength of our hand in combination with the current community cards. However, it has a few flaws, as it does not consider important aspects of the game. For one, it assumes an equal distribution of hands for our opponents. However, unless our opponents choose the hands they want

Algorithm 2.1 Immediate Hand Strength Algorithm

```
**CalculateImmediateHandStrength(holeCards,boardCards)**{
    wins = 0;
    ties = 0;
    losses = 0;
    handRank = calculateHandRank(holeCards,boardCards)
    for each (opponentCards){
        opponentRank = calculateHandRank(opponentCards,boardCards);
        If (handRank > opponentRank){
            wins += 1; }
        else if (handRank == opponentRank){
            ties += 1; }
        else {
            losses += 1; }
    }
    handStrength = (wins+losses/2) / (wins+losses+ties);
    return(handStrength);
}
```

to play, and the actions they want to take in regard to betting and calling randomly, this assumption does not hold. The card combinations which are very weak, like low unsuited cards, are much more likely to be folded early in a round, and the combinations that are very good probably won't be folded most of the time, thus increasing the chance our opponents are holding good combinations which our hand might not win against. Additionally, when considering multiple opponents, we are assuming each hand is independent of each other, which strictly speaking is not true either. More accurate computations are available, however as we only need a rough estimation of strength, we accept this margin of error in favor of quick and easy computation.

Hand Potential

Another problem, the immediate hand rank faces is that it does not consider the ability of hands to improve with additional cards drawn to improve of hands. Hands like 9♡ 8♡ with a board of 10 ♡ J ♡ 2 ♢ on the flop has a very low IHS, as there are $\binom{47}{2}$ = 1081 possible hands an opponent can hold. Of those, each pair, two pair, three of a kind or any card higher than 9 is better than ours, resulting in roughly 813 better hands, 15 ties and 253 worse hands, giving us an IHS of 0,248. However, this hand has many cards it can draw to improve a lot. Any ♡ on the next 2 cards would give us a flush, any Q or 7 would give us a straight, and the Q♡ or 7♡ would even give us a straight flush. Thus, our hand has a lot of potential to improve, hence a high probability to improve to the best hand among all opponents. We call this the positive potential (Ppot). Similarly, we can define and compute the negative potential (Npot). Both values are calculated by once again enumerating all possible hands our opponents can hold, and in addition all the possible board cards which can be drawn, counting up the number of times our hand was behind, but ended up ahead (PPot) and the number of times we were ahead, but ended up losing (NPot), see algorithm 2.2.

Computing this Potential is rather expensive, especially if we are in the Flop stage of the game, with two more unknown community cards to be considered.

Algorithm 2.2 Hand Potential Algorithm

```
CalculateImmediateHandPotential(holeCards,boardCards){
    HandPotential[3][3];
    HPTotal[3];     // /* index 0 represents wins, 1 represents ties and 2 represents losses */
    handRank = calculateHandRank(holeCards,boardCards);
    for each (opponentCards){
        opponentRank = calculateHandRank(opponentCards,boardCards);
        If (handRank > opponentRank){
            index = 0; }
        else if (handRank == opponentRank){
            index = 1; }
        else {
            index = 2; }
        HPTotal[index] += 1;
        for each (turnCard){
            for each (riverCard){
                board = boardCars + turnCard + riverCard;
                ourRankFinal = Rank(ourCards,board);
                oppRankFinal = Rank(oppCards,board);
                If (ourRankFinal > oppRankFinal){
                    HP[index][0] += 1; }
                else if (handRank == opponentRank){
                    HP[index][1] += 1; } }
                else {
                    HP[index][2] += 1; }
                }
            }
        }
    PPot = (HP[2][0]+ HP[2][1] / 2 + HP[1][0]/2) / (HPTotal[2] + HPTotal[1] / 2);
    NPot = (HP[0][2]+ HP[1][2] / 2 + HP[0][1]/2) / (HPTotal[0] + HPTotal[1] / 2);
    return(PPot,NPot);
}
```

Effective Hand Strength

Putting both considerations together, we can define and calculate our Effective Hand Strength(EHS), which gives us the odds that we have the best Hand at Showdown by taking into account our current rank, along with the ability to improve. This leads us to the formula:

$$EHS = IHS^n * (1 - NPot) + (1 - IHS^n) * PPot$$

where n is the number of opponents left in the round. Some implementations set the NPot of each hand to 0 to disregard the potential of decline, as we want to bet when we are ahead at the moment, to prevent opponents from being able to gain the advantage on us. For an accurate computation of winning chance however, we include the NPot into our calculations.

2.2.2 Pot Assessment

While the different Metrics established above give us good tools to evaluate the strength of our hand, they are not always sufficient to make decisions. One of the most common scenarios each player must face is whenever an opponent raises the pot, forcing us to pay the same amount of chips in order to continue playing the hand. If we don't pay the necessary amount, we are out of the round, which prevents us from gaining any chips. If we call, or raise our self, we have to invest more of our stack, for a chance of winning the pot. A big part of accessing these situations is understanding this risk versus reward relation, which can be quantified by calculating the pot odds for each bet.

Expressed Pot odds

A very simple, and easy to calculate indicator, whether it is worth to stay in a hand are the so called *expressed pot odds*. By dividing the cost of calling by the possible return on investment, we get a percentage with which we need to be able to win the hand, in order for calling to be profitable in the long term. The resulting formula looks as follows:

$$EPO = \frac{c}{pc}$$

where c is the amount to call, and p is the amount of chips already in the pot. By combining the assessment of our hand, for example through the use of the Effective Hand strength, together with the Expressed Pot Odds, a player can calculate the mathematically correct decision for each opponent bet. Assume we are holding a hand which proposes an EHS of 0.25 i.e our hand wins 25% of all games from this position. One of our opponents makes a bet of 100 chips, to a 400 chip pot, increasing it to 500 chips. Calculating the pot odds would give us EPO = 0.2. That means to justify calling the bet, we have to have a hand, which wins at least 20% of the time. As we are above this threshold, we should decide to call the bet if we base our decisions purely on mathematics. Of course, most of the time other factors might influence our decisions, such as previous opponent behaviour or our attitude towards risk.

Implied Pot Odds

Expressed Pot Odds often don't tell us the whole story of the hand, because the calculations only look at the current situation to calculate the odds, we need to make calling profitable. As stated before, some hands have a high chance of improving, while others may be best at one street, but have high potential to fall off with every new draw coming in. Additionally, if an opponent bets now, the chance he is going to put in even more chips over the next streets is greatly increased,

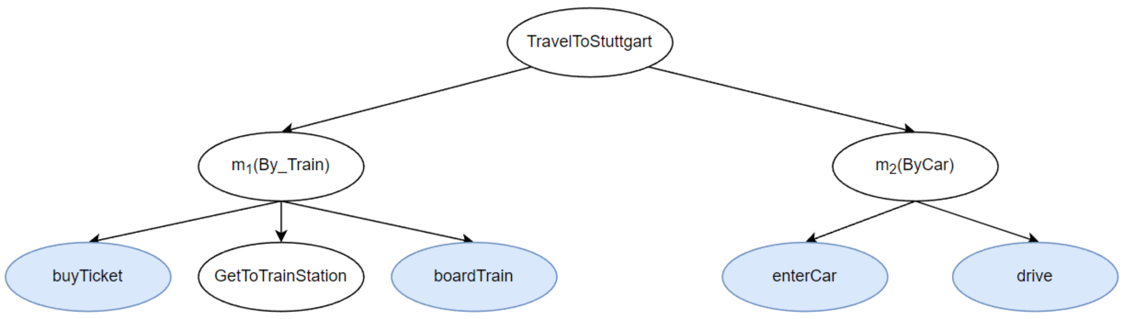


Figure 2.2: Basic example of a HTN task network

which results in an even bigger payout for us, if we make our hand. All these Aspects can be put together in the concept of *implied pot odds*, which are much harder to calculate. Because we are already considering the positive and negative potential of our hands in the Effective Hand Strength Calculations, we are content with using the expressed pot odds as a metric for decision.

2.3 HTN Planning

Hierarchical Task Network (HTN) planning is an AI planning technique, which has been widely used in various domains. The core principle of HTN planning is to represent the planning problem as a hierarchical task network. Complex tasks, which can't be achieved by a single operation get decomposed into smaller sub tasks, until a point is reached, where only tasks that are easily executable in one step remain. This hierarchical representation makes it easier to model complex domains and allows for a high flexibility, as we can alter the definitions for tasks, without drastic effects to the overall structure [GNN+17]. However, creating a fitting HTN planning model for a given domain, well-conceived and well-structured expert knowledge is needed. In contrast to classical planning techniques, rich information and guidance must be encoded into the model, steering the planner to a solution, rather than leaving the creation of the solution entirely up to the planning algorithms [GA15]. An example of a very small HTN task network can be seen in 2.2, here we have two options to decompose our initial compound task TravelToStuttgart, either with m_1(By_Train) or m_2(By_Car). Each of the method decomposes the initial task into multiple smaller tasks, which are primitive i.e they can be achieved directly, marked as blue or have to be further decomposed. In the remainder of this section, we want to provide the formal definitions for the different HTN planning constructs. We first provide an overview of the classical HTN Planning, followed by the extension towards Risk Aware HTN Planning, as specified in [AGA22].

2.3.1 Classical HTN Planning

A HTN planning problem is a 3-tuple $P = \langle s_0, tn_0, D \rangle$, where s_0 is the initial state, tn_0 is a task network, called initial task network, and D is a planning domain consisting of a set of operators and methods [AGA22]. Here, a *task network* is a pair $\langle T_n , \prec \rangle$ with T_n being a set of tasks, and $\prec$

being a partial order over these tasks. Tasks are called *primitive* if they can be accomplished by an operator directly. The operator o = ⟨ pt(0), pre(o), eff(o) ⟩ is a set consisting of a name *pt(o)*, which is identical to the task that can be executed using it, the preconditions *pre(o)* as well as its effects *eff(o)*. Tasks are called *compound* if they have to be decomposed into smaller sub-tasks before they can be solved. For this, we use a method m = ⟨ ct(m), pre(m),tn(m) ⟩ where the parameters describe the name, precondition and the method's task network. If all preconditions of a method are fulfilled, this method is *applicable* and can be used to decompose a task. In order to get to a solution, also called a plan π = ⟨ o_1,o_2,...,o_n⟩ for the problem, we successively decompose tasks, until we can use applicable operators o_1 ... o_n to solve the resulting primitive tasks.

2.3.2 HTN - Planning with Cost-Variable Operators

The Classical Planning Framework assumes that each action causes a definite result to the agent, and the world. However in real world applications, including the game of poker, *uncertainty* and *risk* are involved in most actions, affecting their cost, effects or both at the same time. In order to model these, we need extend the Classical HTN Planning Framework by introducing the concept of *cost-variable operators*. They are defined as a tuple o = ⟨ pt(o),pre(o),eff(o),c(o) ⟩ , where pt(o) and pre(o) are defined as before, while eff(o) and c(o) are tuples of the effect and the cost the operator can have. Formally we define eff(o) as eff(o) = ⟨ p_1(o),eff_1(o),p_2(o),eff_2(o),...,p_n(o),eff_n(o)⟩ where p is the probability, and eff the effect caused. Similarly c(o) is defined as ⟨ p_1(o) c_1(o),p_2(o) c_2(o),...,p_n(o) c_n(o) ⟩ with p and c the probability and the cost respectively. Additionally the following Axioms have to be full filled to create a valid probability space:

- $\forall$ n > 0, $\forall$ i $\in$ [1,n], 0 < p_i < 1
- $\sum_{i=1}^{n}$ p_i(o) = 1
- c_i(o) < 0

2.4 Risk-Aware HTN Planning

By adding Cost-Variable Operators to our definition of the Classical HTN Planning framework, we introduced the concept of risk to our planning problem. Whenever our risk aware decision maker is faced with multiple options, each with a corresponding risk, concepts of decision theory can be used in order to evaluate each option, and choose the action which corresponds to our risk attitude. In general, there can be three types of planning decisions we have to consider

- Choosing a method when decomposing a compound task
- Choosing a binding for variables
- Choosing the order in which compound tasks are decomposed

Each of these choices eventually influences the outcome of the planning process, and thus changes the plan generated. Traditionally, many of these choices were made non-deterministic, resulting in a plan which accomplishes the given task, but there is no guarantee for a high quality of the plan.

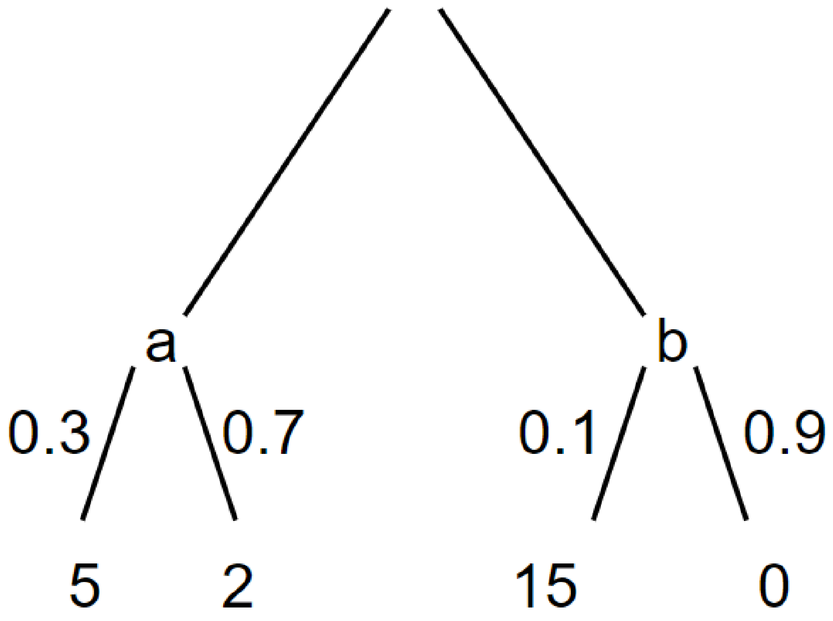


Figure 2.3: A basic example of two choices with different outcomes

2.4.1 Risk Attitudes

In HTN planning with risk, we aim to maximize the quality of the generated plan, by calculating the best option for each planning decision. However, determining, the "best" action to take, when there is risk and uncertainty involved is not straight forward. Whenever we think about decisions involving risk, i.e we face multiple options, with each of them having the possibility for different outcomes, with different probabilities and associated costs, we must think about how much risk we are willing to take, to gain the biggest rewards. Consider the scenario in 2.3. Here, we are faced with the choice between two actions. We can either take action a, which gives us a 30% chance of winning 5 dollars and 70% to win 2, or we could take option b, where we get a low chance of winning 15 dollars, but most of the time we gain nothing. We could calculate the expected value for both options as :

$E(a) = 0.3 * 5 + 0.7 * 2 = 2.9 > E(b) = 0.1 * 15 + 0.9 * 0 = 1,5$

and thus, decide option a to be the better possibility. However, some humans or AI agents might think that the chance of winning 15 dollar is worth it to accept the possibility of leaving empty handed. In the end, the decision we take here is subjective and might be influenced by preferences, the current situation of the agent or their stance on taking risks. This is why in order to evaluate every possible action during the planning process, we must have the specific *risk attitude* of our agent in mind. Some agents might be *risk-neutral*, meaning they only focus on the expected gains or expected losses of actions. However, in most cases, especially in high-stake domains like poker, decision makers will most likely have a different stance on risk. Some might be *risk-averse*, meaning they avoid risky choices, and always pick the safer options, which might not get them the greatest rewards, but won't have high costs associated. On the other end of the spectrum, we have *risk-seeking* agents, which will tolerate the possibility of big losses, if they have a chance of gaining great rewards from their actions. The specific risk attitude an agent has can be split into two basic cases, *static* or *dynamic*. Static attitudes entail no change during planning, i.e. they are not affected by any factors and are pre-determined. Dynamic attitudes on the other hand can have changes, depending on different circumstances. One prominent example would be that humans tend to be ever increasingly risk-seeking in monetary contexts, if their wealth increases. For good Poker players, this effect is often reversed. If a player has a big amount of chips, which can last him many blinds, without any additional winnings, he can play more passively, picking only the best positions and hands to play in, i.e. he can play risk averse. If a player only has a stack of the size of a few blinds however, he has to seek risks, and bet or call in positions which are not perfect, just because he won't be able to stay in the game long enough, to wait for good hands.

2.4.2 Utility Functions

To evaluate each action, while accounting for the specific risk attitude of the acting agent, we use *utility functions*. These are strictly monotonically, non-decreasing functions, which transform the outcome of an action i.e the cost or reward, into an utility value. For neutral decision makers, the utility function is linear, in contrast to risk sensitive agents, which could for instance have a function with exponential growth. In general, for static risk attitudes, if a function is concave, it expresses the decline in utility with increasing cost, which translates to risk averseness. Convex functions gain increasing utility, the higher the costs are, which makes them fitting for risk seeking agents. By propagating these expected utilities to the different methods, we allow informed decision making in every step of the plan.

3 Poker Domain Model

We now defined the framework, in which we want to model our domain of Texas Hold'em Poker. This will enable us to create HTN Planning Problems, which a Risk Aware Planning Algorithm can solve. In this chapter, we want to go over the process of translating poker into an HTN Planning Problem, along with the challenges and decisions we faced.
We mapped the constructs in the given framework to three main building blocks which have to be defined. Cost Variable Operators are our primitive tasks, which can be executed by an agent in one step. We must define their possible costs, as well as effects on the world state, together with a probability distribution over the different actions. Additionally, we need compound tasks, which have one or more methods to decompose into a set of additional primitive and compound tasks. For illustration, we chose to represent the domain using graphs, where compound tasks are represented as blue, methods as white and primitive tasks as green nodes. Black Arrows show the connections in the task networks, while orange arrows signal costs and effects of operations.
Our First problem arose, when we compared the domain of poker to other domains which HTN Planning has been previously used for, like Web service composition [GA15], or container transportation [Aln19]. While there are possibilities of unexpected outside events, which the planner did not account for in almost every domain, these are usually rare occurrences, and are therefore not considered in the planning model. Most Planning systems deal with these situations by including a Monitoring Component, which observes the execution of a generated plan, and generating a fault state whenever a discrepancy between the expected results and the monitored state occurs. Handling the fault can then be done by either trying to repair the plan, or re-planning from the current state of the world. Both options come with additional computation effort, and in some cases even the loss of constraints given by the hierarchical structure [HBBB20]. In the domain of games with multiple human players or AI Agents, and especially No Limit Texas Hold'em making accurate predictions about the outcome of certain actions, or the decisions our opponents make is nearly impossible. Because of this, it is almost to be expected that in each of our planning steps, our predictions are inaccurate, and the plan must be repaired or the planning process has to be restarted. This high computational effort, made us consider two different approaches to modeling the game of poker. The first approach, which we call *one-turn planning*, tries to keep assumptions about actions outside of our control to a minimum, by only ever planning for our next move. In a game of poker, our agent would wait for its turn, before collecting all the necessary information about the current state of the world, like the current street the game is on, or the current amount of chips in the pot, and would then use the planning algorithm to decide the best move to take in this situation. This approach is very resistant to external influence, as each action we take cannot be interrupted by our opponents. When we are considering risk in this model, we only need to reason about the possible costs of our operators, and not their effects, because these will be factored into the starting world state of our next planning iteration. While this approach is much easier to model and eliminates the problem of external events, it comes with some drawbacks regarding the optimality of plans. Because we are only ever thinking about, and calculating the expected utility of our next action, much of the long term considerations a poker player might consider, get lost in this approach. If we are taking

an action in the game, we always get an immediate result, as well as immediate costs, positive or negative, for this decision. Often times however, the real value in taking the action, and the real payoff only shows later in the game. By only looking at the immediate rewards, we lose a lot of potential to set us up for bigger rewards later.

In contrast to one-turn planning we developed *whole-round planning*. Whole-round planning follows the traditional approach of creating plans for the whole problem, instead of splitting it up into smaller sub-problems. This allows our planning algorithm to consider long term potential, and make decisions which might pay out later in the game's process. For an effective consideration of our options, in addition to the consideration of risks in the cost distribution of each action, our model has to include assumptions about all the possible effects our actions cause, including the reaction of the opponents. This is a very hard task to accomplish, especially if there is a high number of opponents playing. Whenever an opponent does not act as we expected, a new planning process has to be started from the current state, which causes multiple re-planning steps in a single round of poker.

In conclusion, to solve our problem of unexpected events we developed two different approaches to modeling the poker domain. The one-turn planning approach is easily implemented, and easy to compute plans for, however not accounting for long term effects might cause problems regarding the optimally of plans. The whole-round planning approach is harder to model, and the problem might have to be planned several times, however the generated plans follow a more optimal strategy. We will compare and evaluate the two different approaches in Chapter 5.

As we have seen, guessing our opponent's response to our actions is very important in both modelling approaches. This, however, is not an easy task, because we only have limited information about our opponent. In our models, we chose to use the Pot Odds as our main source of predicting our opponent's reaction to our bets. Whenever we raise the pot, we calculate the current pot odds, giving us a required equity for calling to be profitable. We then count all the hands which meet this equity requirement, and assume that every opponent would call our bet if he were holding one of these hands. By dividing the number of hands which will call by the total number of possible hands, we get an estimation of one opponent calling. This number can be raised to the power of the number of opponents, in cases where multiple opponents remain. While this calculation of odds makes many assumptions, which are not entirely accurate, like assuming an equal distribution over every hand, or disregarding the ability of our opponent to re-raise, it is an easy to compute estimation, which gives us a rough idea of what we can expect from our raise.

We use a similar technique for estimations regarding our showdown performance. By calculating the rank of our hand, in conjunction with the five community cards on board, we gain the number of hands which are weaker, tied and stronger than ours. We now assume equal distribution over every possible hand and calculate the odds that none of our opponents hold a better hand than we do[1].

Betting is one of the most important actions we can take during a poker game. Bets are used to reduce the number of opponents we must face, increase the amount of chips in the pot, so we can win bigger sums and when bluffing, obscure the information our opponents might have about our hand. In addition to the decision when we want to bet, we also have to consider of how

[1]For the implementation of the ranking of hands, we adopted and modified code from https://github.com/VermeirJellen/Poker_HandEvaluation

many chips our bet should consist. Bet sizes in Poker are always relative, either to the current pot or to the amount of chips which form the current blinds. For example, a bet of 100 chips might be big, if the pot is only 10 chips, but it would be a very small bet, if the pot is already sitting at 10000 chips. Because of this, we decided to split our raising options into 4 different categories - low,medium,high and all-in, with the exact amount of chips for this category relative to the current pot, except for the all-in raise, which depends on our current stack. We decided to define a low raise as half the pot, medium raise as exactly the amount of chips in the pot, effectively doubling the pot size and big raises as a raise double the size of the pot. This allows each of our raises to stay relevant throughout each hand but comes with the drawback of a potential for very big pots, causing us to potentially consume our whole stack quickly, which however is not unusual to see in no limit poker, even in professional play.

As a general design philosophy, because we are more interested in the potential of HTN planning in poker, rather than creating the best possible poker AI, we opted to include as little strategic guidance as possible in our model. This means that we did not include betting suggestions for specific hand strengths or any guidelines as to which actions perform the best in any given situation.

3.1 One-turn planning

We have now talked about the basic decisions we took which play a role in both approaches. Now we want to go over the one-turn planning approach in detail. A full graphical representation of the domain can be found in Appendix A. For the planning process, our world state needs to consist of the following

- **current-street** this value encodes the different stages the game could be in into numbers. If current-street = 1, we are in the Preflop Stage, current-street = 2 means we are at the Flop etc.
- **position** this encodes our position at the table. While the Player to act first is the UTG player, sitting to the left of the big-blind, we decided to encode the small-blind player as the position number 1, going around the table in usual direction from there. This means that number 1 and 2 have the special positions of small-blind and big-blind respectively, which forces them to make the mandatory bets at the start of the Preflop stage.
- **number-of-opponents** this value represents the number of opponents which are playing in this round. This value is important for probability calculations. Valid input is any number greater than 0, however usually full poker tables consist of at most 9 players.
- **stack** this state variable keeps track of the amount of chips our agent has available. While the gain and loss of our resources are determined and kept track of with the costs of operators, our stack value is important to determine whether or not preconditions for betting actions are fulfilled.
- **pot** the pot is one of the most important values in a game of poker. We use the pot to determine winning amounts, as well as the probability of our opponents playing in a round. The bigger the pot is, the more profitable it is to play.

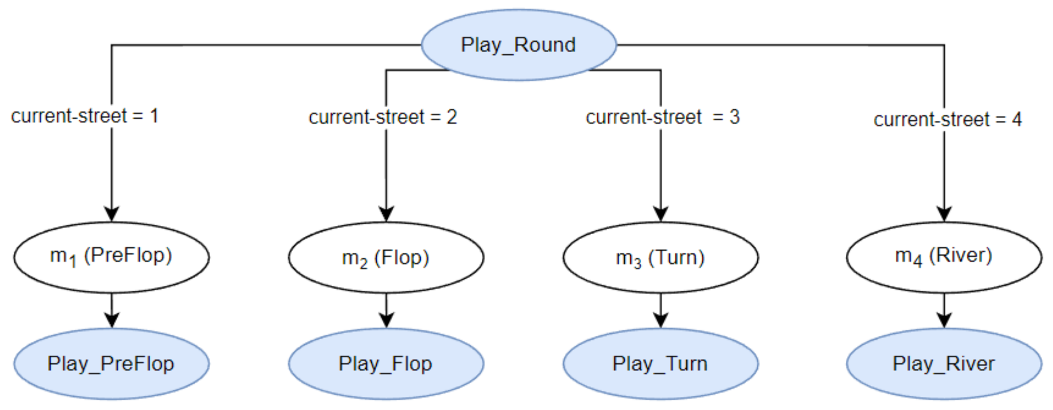


Figure 3.1: The fist part of the one-turn domain model, which controls the planning process based on the current-street

- **amount-to-call** The amount-to-call describes the current amount of chips we would have to put in the pot to keep playing. In order to stay in a hand, this value has to be 0 at the end of our turn.
- **big-blind and small-blind** Describes the current rate for the big and small blind.

We now have defined all the values our planner needs to create plans. In the beginning of each planning process, we must check in which stage of the game we are, as available actions are different for the individual stages. For this we define the methods “Play_PreFlop,Play_Flop,Play_Turn,Play_River”, which decompose our initial Task Network consisting of the Play_Round compound task. For each of these methods, our planner will check the current-street state variable, to determine which one is able to be applied, see Figure 3.1. Our Task network ends up with one compound task, which is different depending on the current-street of the game. Each of the resulting compound tasks can be decomposed by one method, where it is split into Sub-tasks. These Sub-tasks are different for each stage of the game; however they always include the Determine_Action compound task. This is where our planner must decide, there are four actions we can take at this point, corresponding to 4 methods to decompose the task. We can either raise, call, check or fold. Each of the options has specific preconditions, which have to be fulfilled, to do the action. Checking requires the amount-to-call to be zero, while calling needs it to be greater than zero. The rules of poker don’t include any requirements for the folding action, meaning you can fold your hand, even if there has not been a raise and you could go to the next street without any investment. In our model however, we decided to limit folding to be applicable, only if the amount to call is greater than zero. This is done, to prevent our planner considering this option as a alternative to checking, as both actions don’t have any cost attached to them, and thus have equal utility values. While the call, check and fold methods decompose the compound task into primitive tasks, which can be included in our plan, the raise method first generates another compound task to determine the amount we want to raise for. The whole Determine_Action model can be seen in Figure 3.2.

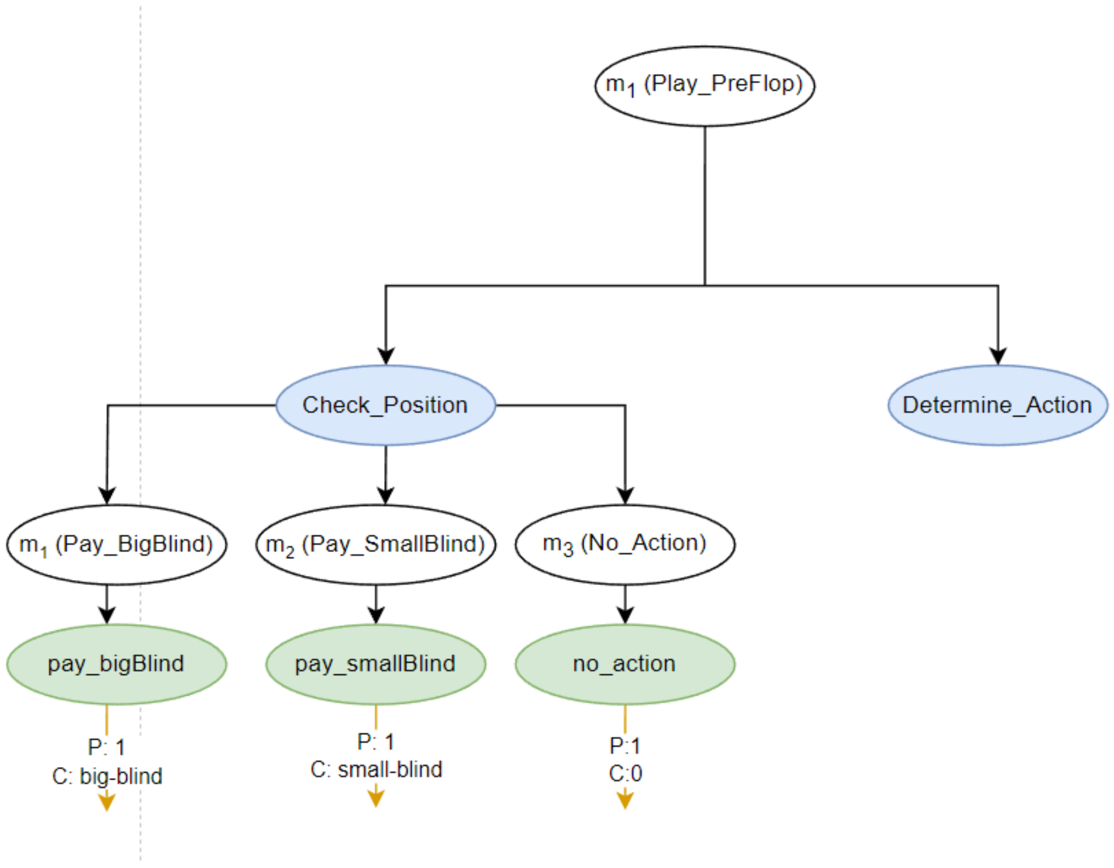


Figure 3.2: The Determine_Action compound Task

Apart from the Determine_Action Sub-task, the method for the Preflop stage is special, because it decomposes its compound task into an additional Sub-task, called "Check_Position". This Branch is for deciding whether our agent has to pay either the big, or the small blind, adding it to the pot of the round, as well as to the cost for the plan. The full decomposition of the Preflop Task can be seen in Figure 3.3.

3.2 Whole-round-planning

While the one turn approach can be used to calculate the next best action, according to our risk attitude, it lacks the ability to incorporate the effects our actions have on the future. The whole-round planning approach fixes these problems, however it is susceptible to unpredicted changes to the world state, which forces the planner to either repair, or recalculate the plan. In addition to the constants and variables we defined for the world state of the first approach, whole-round-planning needs a few extra state atoms for control flow reasons. These are

- **playing** This boolean value is set to true, while we are still playing in the round. If we fold, or have no chips left, we set playing to false, to indicate that we can't take any other actions.

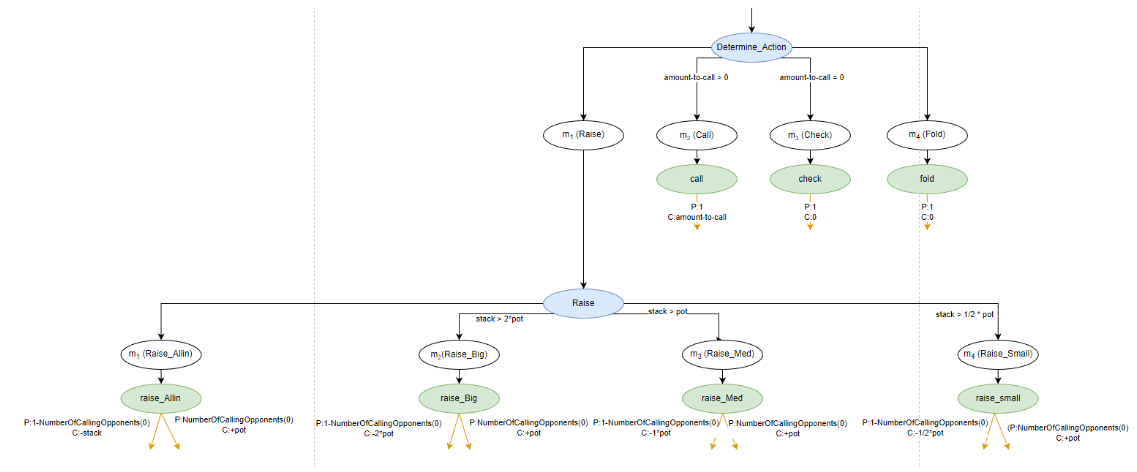


Figure 3.3: The Pre_Flop compound task

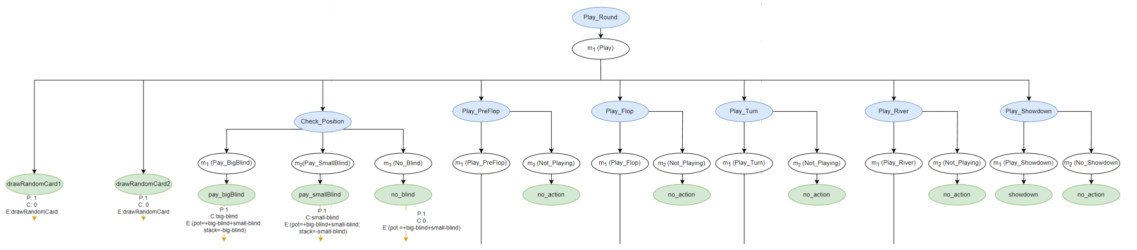


Figure 3.4: First Part of the whole round domain

- **round-over** This boolean value is set to true, whenever there is no opponent left playing in the round. This indicates to the planner, that we don't need to plan for any future streets.
- **cards one and two, and community cards one to five** Our hole cards, as well as the five community cards, get encoded as a number, where the first digit between 1 and 4 represents the cards suit, and the following digits determine the rank of the card.

In contrast to the one-turn-approach, this model has only one method to decompose the initial play_round task, because we have to plan for each of the four phases of the game. This single Methods decomposes into the compound tasks Play_PreFlop, Play_Flop,Play_Turn, Play_River, Play_Showdown as well as Check_Position and two operators which manage the drawing of our two hole cards. As before, Check_Position is used to select whether we have to pay the big, the small or no blind, and adjusts the pot, the amount we need to call, as well as our stack accordingly. Each of the other compound tasks are similar in their decomposition options. They can either be decomposed to the no_action primitive task, if we are not playing in the current round anymore, or to the Determine_Action compound tasks, which we already used in the previous model. Along with two more compound tasks used for control flow, the Guess_Opponent_Raise Task and the Check_End compound task. Guess_Opponent_Raise is used to try to estimate whether or not one of our opponents raises the pot, while the Check_End compound tasks is used to either continue the betting round, if an opponent raised i.e we have to call a bet to continue, or ends the street, allowing us to move on to the next one.
In contrast to our first model, each of the primitive tasks does not only affect our costs, but it also has

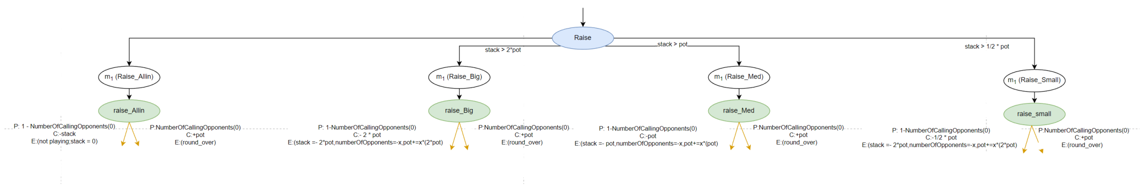

Raise
stack > 2*pot
stack > pot
stack > 1/2 * pot
m1 (Raise_Allin)
raise_Allin
P: 1 - NumberOfCallingOpponents(0)
C:-stack
E:(not playing;stack = 0)
P:NumberOfCallingOpponents(0)
C:+pot
E:(round_over)
m1 (Raise_Big)
raise_Big
P: 1-NumberOfCallingOpponents(0)
C:- 2 * pot
E:(stack =- 2*pot,numberOfOpponents=-x,pot+=x*(2*pot)
P:NumberOfCallingOpponents(0)
C:+pot
E:(round_over)
m1 (Raise_Med)
raise_Med
P: 1-NumberOfCallingOpponents(0)
C:-pot
E:(stack =- pot,numberOfOpponents=-x,pot+=x*(pot)
P:NumberOfCallingOpponents(0)
C:+pot
E:(round_over)
m1 (Raise_Small)
raise_small
P: 1-NumberOfCallingOpponents(0)
C:-1/2 * pot
E:(stack =- 2*pot,numberOfOpponents=-x,pot+=x*(2*pot)
P:NumberOfCallingOpponents(0)
C:+pot
E:(round_over)

Figure 3.5: Different raising options, along with their cost and effects

an effect on the overall world state. This also includes the effect our actions have on our opponents. Figure 3.5 for example, shows our different raising actions, which can have different outcomes, depending on the number of opponents who call our bet. If no opponent calls, we have won this round, which sets a flag, indicating that the round is over and no additional streets have to be played, and we get the current amount of chips as a positive cost. If one or more opponents call a bet, the round continues and we adjust the number of opponents, the pot and our stack accordingly. The addition of effects for each primitive task, as well as the extra control flow adjustments this model has to take care of, causes the model to be a lot more complex. By reusing many of the tasks in different locations, we did, however, manage to create a maintainable model, allowing adjustments to be made rather easily.

4 Implementation

In the previous chapters, we created a model of our poker domain, in the risk aware HTN planning framework, we also presented algorithms which we can use in order to find optimal plans according to our risk attitude. In this chapter we show our implementation of a HTN planner, that is capable of creating solutions to these problems. We start by introducing the HTN planner we extended to achieve risk awareness, followed by explaining the structure of the input files. Lastly we want to demonstrate the algorithms our program uses to generate risk aware plans.

4.1 Overview

There are various HTN Planners available, implemented in many different languages. For this work, we chose to use the Java Simple Hierarchical Order Planner (JSHOP2) as a basis, to extend with our risk aware functionality, thus creating Risk Aware JSHOP2 (RAJSHOP2) [Ilg06]. We chose this particular Planner, due to its high popularity and efficiency among other state based alternatives, as well as its high expressive power, which enables it to be used even for complex domains. What makes JSHOP2 stand out from other domain independent planners, is that rather than just interpreting a domain and problem description, it compiles the input into a domain specific planner. This process comes with various advantages, including a much higher performance [IN03]. For this translation from our input files to programming-level objects that we can later use for the planning process we need a Lexer as well as a Parser tailored to the structure of our input. In

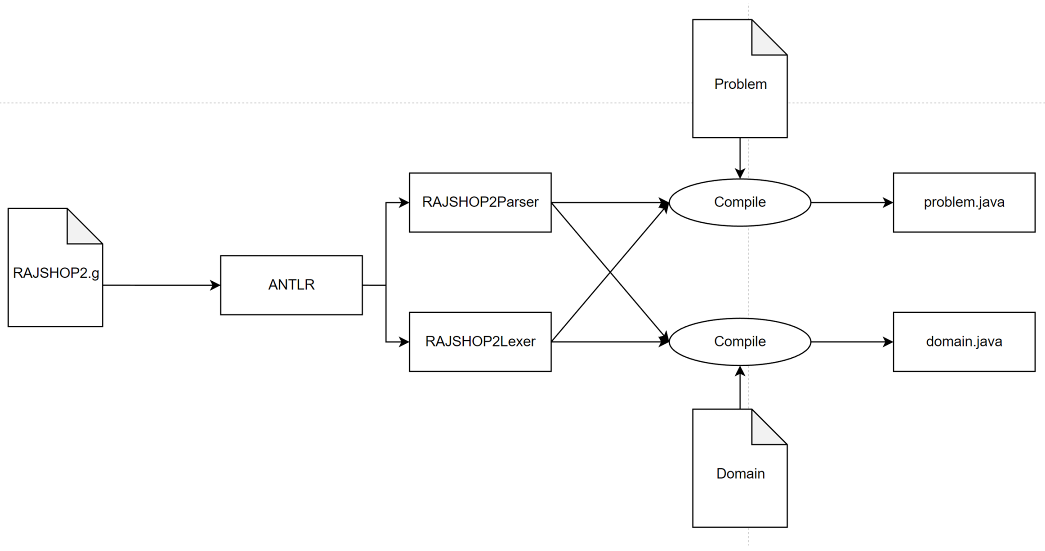


Figure 4.1: Compilation Process (Adopted from [Aln19])

JSHOP2 these are generated with the help of the ANTLR (ANother Tool for Language Recognition) tool [PQ95]. For the generation of the Lexer and Parser, ANTLR takes a grammar file as an input, which defines the structure of the input language. The whole compilation process is illustrated in 4.1,

4.2 Input Structure

In order for the Lexer and Parser to translate the domain and problem input files, into programming objects, they have to follow a specific syntax. This syntax is very similar to the one used in JSHOP2, with a few changes to accommodate for the risk awareness. We want to give a quick overview here, as well as provide examples taken from our poker domain input files. We will be going through the different structures bottom up, because higher level structures often contain lower-level structures as building blocks. The complete domain description can be found in Appendix B.

4.2.1 Symbols

In RAJSHOP2 there are five different kinds of symbols : *variable symbols, constant symbols, primitive task symbols, compound task symbols* and *function symbols*. Each one of those, must only consist of letters, digits, hyphens and underlines. Question marks and exclamation points can form valid symbols as well, however they are reserved for distinguishing between the different variations as follows :

- **variable symbols** can be any symbol , which begins with a question mark (such as ?pot or ?position)
- **primitive task symbols** can be any symbol, which begins with an exclamation point (!raise_Big,!all_In,!fold)
- **constant symbols, predicate symbols, compound task symbols** can be any symbol beginning with a letter or an underline (play_Preflop, maxPlayers)
- **function symbols** can be any valid Java identifier

Additionally, there are a few RAJSHOP2 keywords, which are reserved and cannot be used as symbols.

4.2.2 Terms

A term can be one of the following :

- A variable symbol
- A constant symbol
- A number.
- A **list term**
- A **call term**

A **list term** must have the following form :
(t_1 t_2 ... t_n [. L])
where each t_i is a term. As the name and the syntax suggest, these terms are part of the list. The last element is optional, and only serves to ensure backward compatibility with SHOP and SHOP2.

call terms are of the form:
(call f t_1 t_2 ... t_n)
Here, each t_i is once again a term, while f can either be a built-in RAJSHOP2 function (such as + , - , *), or an external function the user wrote. The semantics of call terms are that the function f gets applied to the terms t_1 ... t_n. Because terms can be variables, the result of the call term replaces the call term during the planning process, all appearing variables are bound to values.

4.2.3 Logical Expressions

A **logical atom** is of the form :
(p t_1 t_2 ... t_n)
with p being a predicate symbol, and each t_i a term.

A **logical expression** can either be a single logical atom, or one of the following combination of multiple logical atoms :

- conjunction : (and L_1 L_2 ... L_n)
- disjunction : (or L_1 L_2...L_n
- negation : (not L)
- call expression: as before, external functions can be called, in this case they have to evaluate to true or false

Logical expressions are further used in two concepts, the first one being **logical preconditions**, which will be used later. The second one are **axioms**, which are of the form (:- a [$name_1$] L_1[$name_2$]L_2 ... [$name_n$] L_n) Here, a is a predicate, and the List of L_i's with their respective optional name is a list of logical preconditions. Axioms can be used to define conditions, which depends on multiple different factors. Its meaning is, that a is true if L_1 is true , or if L_1 is false but L_2 is true,..., or if L_1 to L_{n-1} are false but L_n is true.

4.2.4 Operators and Methods

Operators and Methods in RAJSHOP2 are the equivalent of our primitive and compound tasks. They are defined as follows, with examples in Listing 4.1 and Listing 4.2:
(: *operator h* P D A c) where:

- h is the head of the atom, which consists of a primitive task symbol and a list of terms
- P is a logical Precondition
- D is a delete risk list, with each delete risk consisting of term, representing a probability, followed by a list of elements to delete from the current state

- A is an add list, which is defined similary as D
- c is the operator's cost, unlike JSHOP2 , RAJSHOP2 does not define the cost as Term, but rather as an pair of (probability cost), with both elements being Terms themselves.

```
(:operator (!pay_big_blind)
  ((playing) (stack ?s)(big-blind ?bb)(small-blind ?sb) (pot ?p))
  ((1 ((pot ?p)(stack ?s))))
  (( 1 ((pot (call +  ?p (call + ?sb ?bb)))(stack (call - ?s ?bb)))))
  (( 1 (call * ?bb -1))))
```

Listing 4.1: operator example

(:*method h* [name] L T) where:

- h is the head of the method, which consists of a compound task symbol as well as a list of terms
- L is a logical Precondition
- T is the tail of the method, which consists of a list of tasks

```
(:method (determine_action)
  CallBet
  ((amount-to-call ?a) (call > ?a 0))
  ((!call_bet)))
```

Listing 4.2: method example

4.3 Plan creation process

Because our implementation is an extension of the JSHOP2 Planner, it shares the same general Plan creation process. In order to account for risk and uncertainty however, we had to modify the algorithm, to account for risk, as well as the utilities of operators. While JSHOP2 chooses the method it uses to decompose by the order they are specified in the domain description, which results in valid but not optimal plans, we are interested in maximizing the expected utility of our result. This corresponds to an optimal plan, following a given risk attitude. To calculate the utility of each operator, we use a static Risk Attitude, with the following exponential utility function, as denoted in [AGA22] :

$$U_c(c_i(\mathrm{o})) = \frac{ae^{a\alpha c_i(o)}-1}{\alpha}$$

Where:

1. $c_i(o)$ describes the i-th possible cost of the operator o
2. a is an attitude-determinant coefficient

3. α is a curving coefficient

If the attitude-determinant coefficient a is positive, our utility function describes a risk seeking attitude, while a negative value describes a risk averse attitude. The curving coefficient can be used to control the degree of the risk sensitivity of this agent. Figure 4.2 and Figure 4.3 show different utility functions, with varying size of curving coefficient. As our operators are associated with risk and uncertainty, their costs cannot be determined deterministically, and thus, the utility for the operator can change. For this reason, we use the **expected utility (EU)** of an operator for evaluation, calculated as :

$$EU(o) = \sum_{i=1}^{n} U_{\mathrm{c}}(c_{\mathrm{i}}(o)) \times p_{\mathrm{i}}(o)$$

For methods, which decompose compound task we define the utility as the sum of the utilities of each sub-task this method includes. This way, we can compute the optimal methods to use for decomposition, by searching for the best path through our search tree in a bottom-up manner.

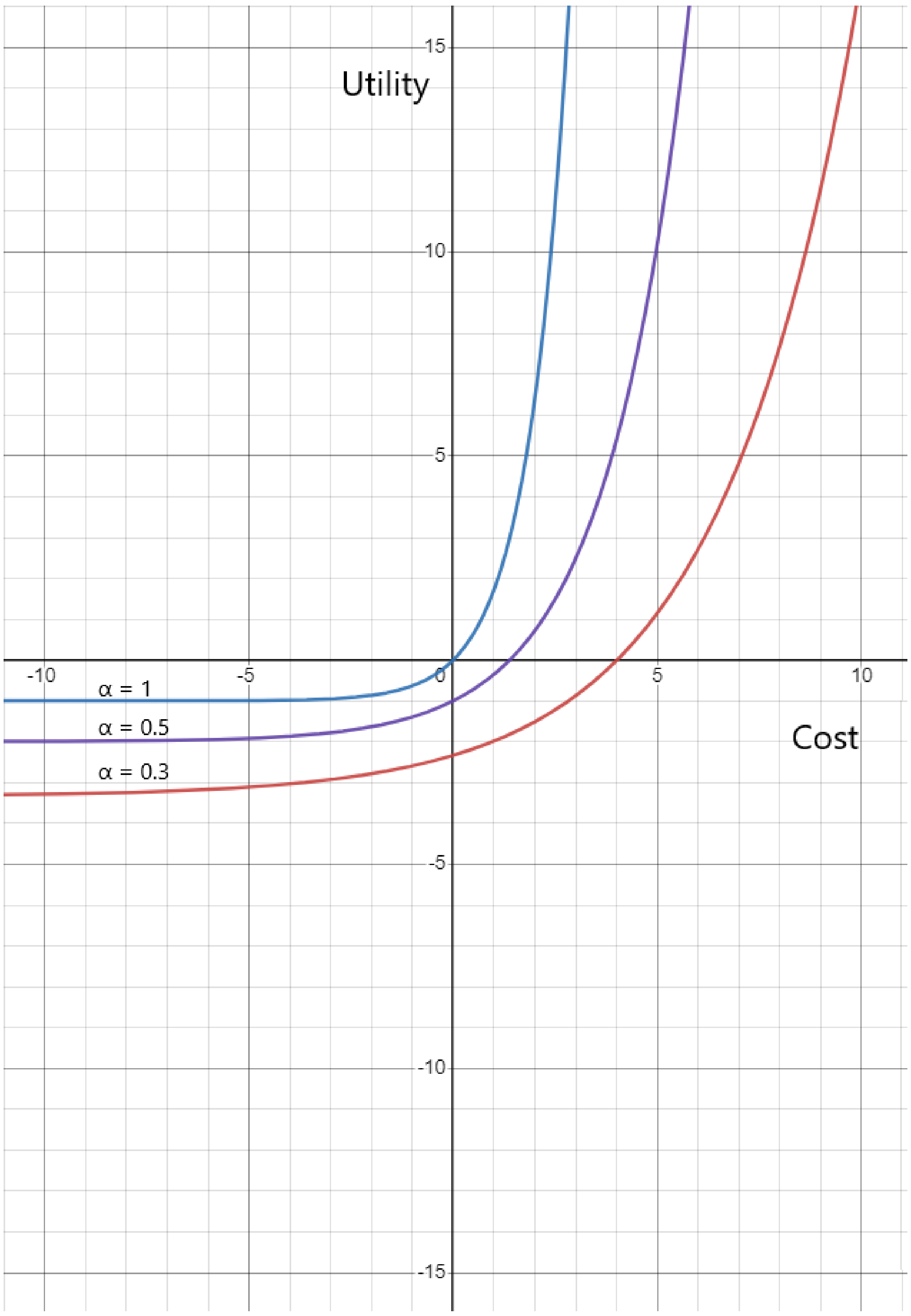


Figure 4.2: Risk seeking utility functions

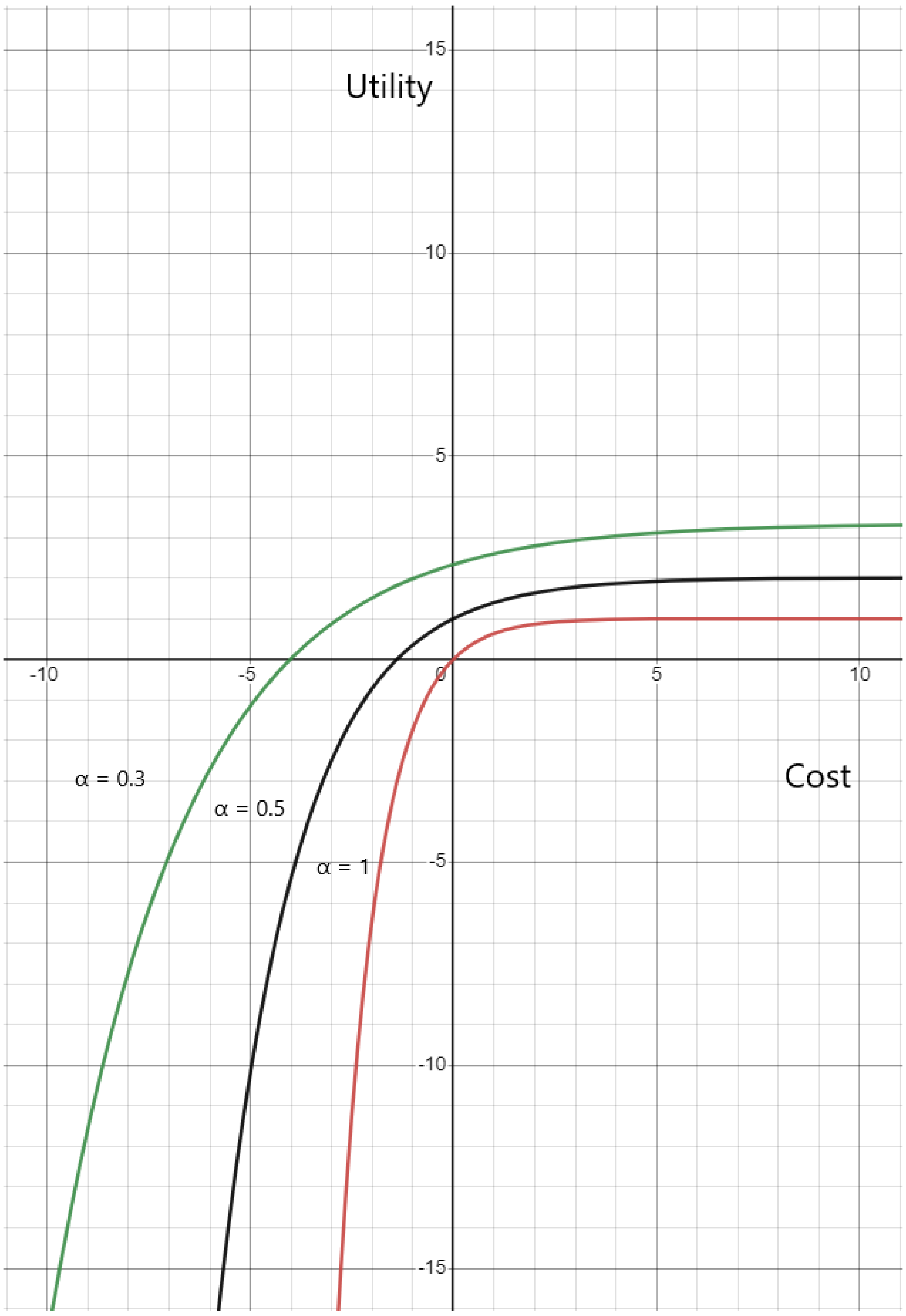


Figure 4.3: Risk averse utility functions

5 Evaluation

In this chapter, we want to present our results, and evaluate the plans RAJSHOP2 generated, with our two domain models as input. We want to showcase how different factors, such as risk attitude, pot size, or number of opponents influenced the generated plans, their cost as well as the expected utility, while judging whether the suggested course of action corresponds to the strategic principals we established.

5.1 Evaluating one round planning

We first want to look at the one-round planning approach. One of the most standard decisions, a poker agent must face is whether to call a given bet, fold or even re-raise. Figures 5.1 and 5.2 show the expected utilities of each possible action with increasing size of the pot respectively. We chose to run our calculations on 4 opponents, with an amount of chips to call of 10, as this simulates a standard situation in one of the middle streets. While we offer multiple different bet sizes in our model, we chose to only display the utility for the most rewarding variant for clarity. Because calling and folding do not propose any immediate rewards, and our one turn planning model does not account for long term gains, both of those actions do not change in their utility with an increase

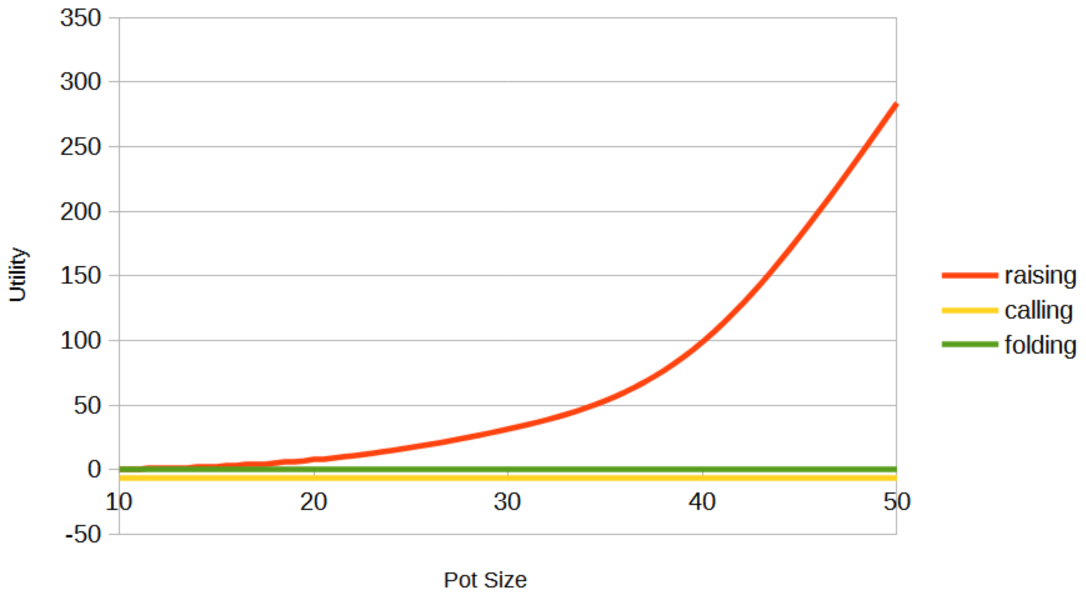


Figure 5.1: Utilities of available actions for a risk seeking agent

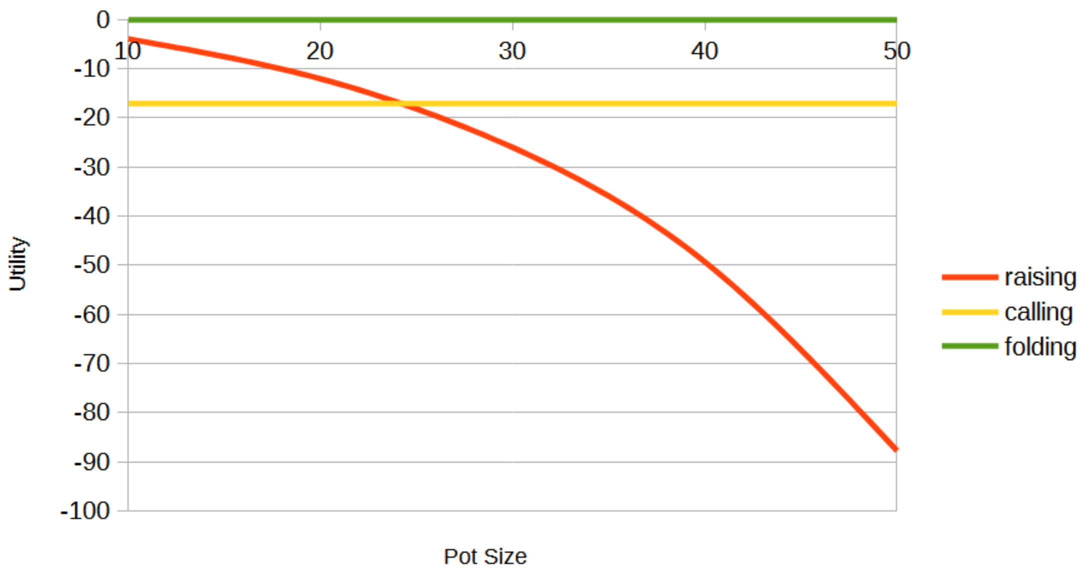


Figure 5.2: Utilities of available actions for a risk averse agent

in the size of the pot. For the risk seeking Agent, we can see that even for low possible pot winnings, raising is the action which is slightly preferred. Due to our exponential Utility function, this preference gets even bigger, the higher the amount of chips in the pot. In contrast, we have the risk averse attitude, who wants to refrain from betting chips as much as possible. As expected, to avoid any costs, an agent with this risk tolerance would fold to the bet every time. It is interesting to see, that with higher possible rewards, the act of raising, which would give us a chance to win the pot, decreases in utility, nevertheless. This effect is caused by the nature of our model, as we decide our raise amounts in relation to the pot size, which causes potential higher rewards to have the chance of bigger costs as well.

We have seen that in this domain model, raising is the most interesting action we can take, being influenced by multiple factors.

Figure 5.3 shows the calculated utilities for different available raise sizes for a risk seeking agent, with an increasing number of opponents. We can see, that the small raise is always the preferred raise size for this approach. While the small raise is an action with positive utility for our agent even for a full table of seven opponents, making it preferable over neutral actions like checking or calling, the bigger raises only gradually improve over this threshold with lower opponents present.

For the risk averse agent, as seen in Figure 5.4, as expected each of the bet sizes lays below the neutral point, with the big raise being almost at a maximum negative utility even for low player numbers. This effect is even greater for the all-in raise under the given circumstances, which we left out of the diagram for visibility reasons. If we compare these results to the strategy guideline we established, we can conclude that the risk seeking agent, while following an aggressive playstyle which is in accordance with strategy suggestions, the fact that raising small always seems to be considered as the preferred action does not correspond to good poker player. For one, we usually want to diversify our actions to stay unpredictable, but we also usually want to adjust our turn to the current state of the game, which our suggestion does not do. In contrast, the risk averse agent does change up the suggested action based on our game state, it does however never take the lead in the

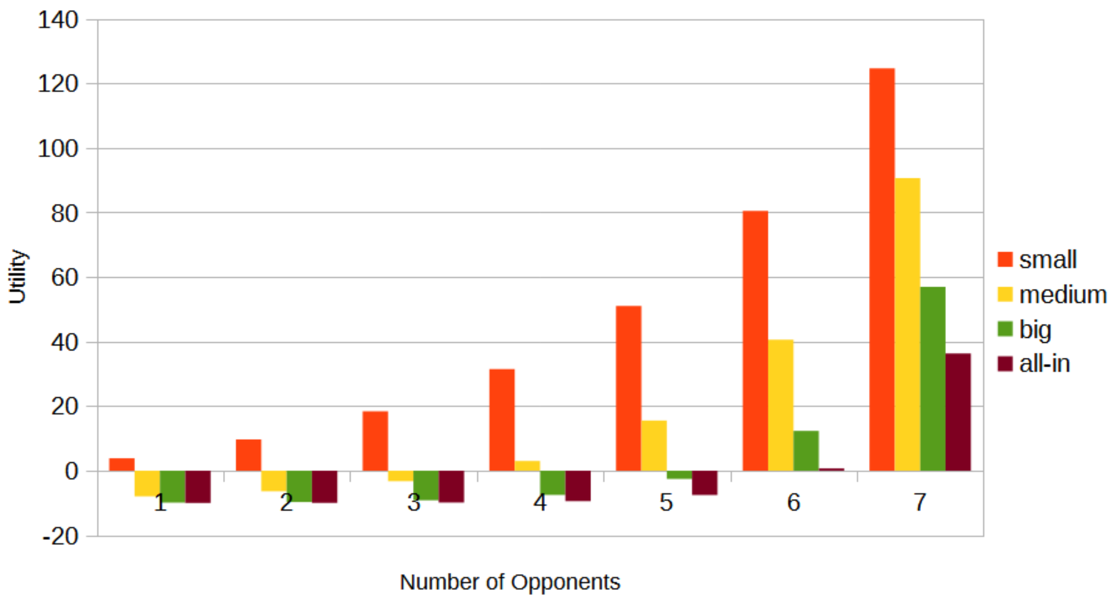


Figure 5.3: Utilities of different raise sizes for an risk seeking agent

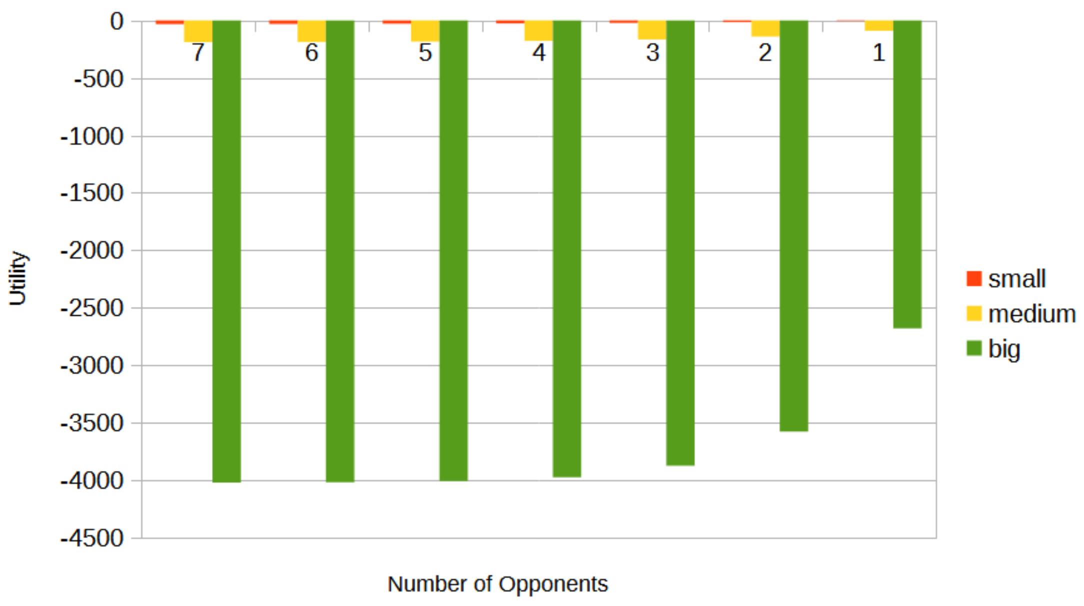


Figure 5.4: Utilities of different raise sizes for an risk averse agent

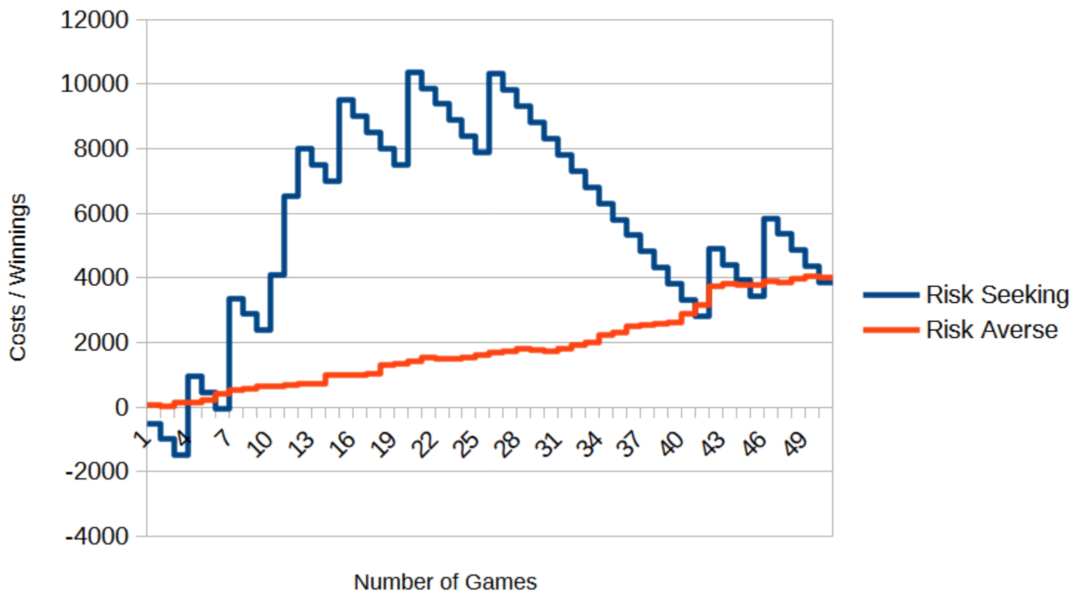


Figure 5.5: Cumulative Winnings over 50 games

game and play aggressively, which makes it unlikely to win any round.
These problems can be led back to our domain model. Because aggressive actions do not offer great immediate rewards, they are mostly ignored. In a real game of poker however, the effects of raises have a big impact on any of the following streets, which is not considered in this approach.

5.2 Evaluating whole-round approach

In order to evaluate the whole-round approach, a different methodology had to be used. Because the model makes assumptions about effects of actions, trying to predict as accurately as possible how the game is going to develop, we gain a simulation of a possible game. As most of the effects are tied to uncertainty, some of which are entirely up to chance, such as the drawing of cards, we can not guarantee the game to follow this exact course. Additionally, this means, that not only does the cost for each plan we generate change for each planning iteration, but the game can also turn out in completely different ways, given the same input. This is why, for our analysis we decided to simulate 50 games for each experiment setting. First, we tested a rather standard round of poker, on a full table of 8 players, with a medium stack size of 200 chips, and low blinds of 2 and 1 chips for the big and small blind respectively.
Figure 5.5 shows the cumulative winnings our simulation predicted over the course of the 50 games for a risk averse as well as risk seeking attitude. We can immediately see the big difference between the two approaches. While the Risk seeking attitude regularly gains a large amount of winnings, losses are also frequent and severe. In comparison, the risk averse attitude generates an almost steadily increasing number of chips won, with very rare, small losses in between. This behaviour does correspond to our general understanding of risk. Taking risks has the upside of great rewards,

Attitude	Fold	Check	Call	Raise-Small	Raise-Med	Raise-Big	Raise-all-in
Risk Seeking	0	37	0	4	4	19	50
Risk Averse	6	64	0	93	18	1	0

Table 5.1: Count of Actions over 50 games

however the chance of losing is heavily increased as well. The attitude and playstyle of the two agents becomes even more apparent if we look at the frequencies for the different actions taken, as displayed in table 5.1.

As you would expect, a risk seeking attitude entails to almost never fold, and to raise on most opportunity's instead. The high number of all-In raises shows us why we have so much fluctuation between the amount of chips we win and lose. The risk seeking agent wins chips in our model by bloating up the pot with big raises in the beginning of a round, scaring a few of our opponents away, and then going all in in the end, causing us to either win a big pot, or lose our whole stack. In Contrast, the risk averse agent prefers small raises, and checking to every other action. Additionally its utility function allows to fold very weak hands, if there are no rewards to be expected without taking a big risk. Looking back at the strategy guidelines we laid out earlier, one could argue that the risk seeking attitude is playing more ideal. It plays an aggressive style of poker, trying to win the pot even before showdown. The problem however is that every hand gets played as if it is a very strong hand, which works well against inexperienced or very tight players, which quickly fold to big bets. Good opponents will realize this behavior quickly and develop strategies to exploit this weakness. Our less risk tolerant agent in comparison, does not bring any aggression to the table, playing every hand very passively. As we have seen before, this takes away our win condition of getting all other players to fold their hand. While we have seen that this can lead to an increase in chips, by winning small pots, without losing big sums, it must be mentioned that our domain models opponents as rather passive, having them raise the pot rather infrequently.

5.3 Comparison

Due to the different nature of the results, comparing our two approaches is not a straightforward task. While the first approach is well suited for an independent look at each action separately, it does not give us very deep insight into the consequences of our actions in regard to the game as a whole. Because most reasonable strategies, including the one we want to follow, do however set a focus on long term effects and rewards, this approach does not correspond to a good strategic playstyle.
The whole-round model, on the other hand, allows us to simulate complete games, judging the value of actions in a broader scope. This, however does not mean that the game will actually play out the way we predicted it to, and in fact it is rather unlikely that it will. Small errors, like wrongly predicted cards, might be easily repairable by an automated system, for inconsistencies which have a bigger effect on the plan. However, re-planning is likely to be the best option. A combination of

both techniques could be feasible, by starting with the whole-round planning and switching to the one-turn option, whenever an opponents actions differs to much from our generalized opponent model. In the end, both models can give us vital information about the planning process, as well as the effectiveness of risk aware HTN planning in the domain of poker. The whole-round approach however seems to deliver more interesting results, as well as a more precise plans.

6 Related Work

Both our main topics , AI in Poker and Risk in AI planning has been researched in several works. We want to quickly summarize previous approaches to using AI to solve poker, followed by different approaches to the problem of opponent modelling. Then we want to give an overview of studies which also considered risk in Planning.

How AI can solve Poker has been studied in various different research projects over the years. It was quickly recognized, that full scale, No Limit Texas Hold'em is a very complex game, which requires complex algorithms to play effectively. In order to reduce the problem size different simplifications to the game have been applied. “Libratus” is an AI which focuses on Heads Up No Limit Hold'em [BSM17]. Here, Heads Up means, that only two players are in the game. Libratus consists of three basic modules

- Pre planning of approximnate nash equilibrium strategy, which uses further abstractions to simplify the problem, and reduce the decison points. Action abstractions are used to group different bet sizes into categories, while card abstractions group similar hands together.
- Subgame Solving during play In order to ensure that the subgame solution is not worse than the pre planned solution, in this module only action abstraction is used.
- Improvements based on opponent adjustment

Kuhn goes even further in simplification, in addition to restricting the analysis to a two player game, he also used only a three card deck, one card hands and only allowed for one betting round with a maximum of two betting decisions [Kuh50].
While these works can deliver valuable information, we are interested in this domain for its high complexity, which is why we chose to research one of the most challenging variants in No Limit Texas Hold'em One of the first Artificial Intelligent systems for full scale poker was Loki, followed by its successor Poki. [SBPS99] One of the main areas of research in regards to Poker is the field of Opponent Modelling. Multiple different approaches to this have been presented over the years. In [Van10] an Algorithm which is based on a decision tree is presented, while the Poker Program Loki uses a weighting table, which gets adjusted after every opponents move. [BPSS98a] Artificial intelligence has also been considered as a possible, and effective way of predicting opponents moves. [FKP12] presents an Machine Learning approach, using Neural Networks to gain a rate of about 18% of correct classifications. A more extensive review of different approaches can be found in [Bou14] The great amount of effort which has been put into developing good opponent modelling approaches, shows how important this area is for an efficient poker agent. For our work, because creating an AI which can perfectly predict each opponent individually was not the goal, as we are more interested in the abilities of our planning approach, we opted to use much simpler, generic opponent modelling based on statistical values.

Risk and uncertainty in HTN planning has been considered in previous work. In [MCM12] actions with static utilities for robotic domains are considered. Here, the utility is implemented as a binary value, where the outcomes utility is set to 0 if it corresponds to a failure. While utility functions are used in [LZZ12] and [MMS+18] to evaluate costs and effects of actions, minimizing cost and maximizing utility respectively. However, both approaches do not incorporate risk and risk attitudes into their planning process.

7 Conclusion and Outlook

The domain of Poker is a very complex and challenging area of research. The dependency on the actions of multiple players, along with misinformation and deception is one of the biggest factors as to why this game is still not perfectly solved to this day. It is this same factor which makes it a challenge to model this domain into any AI Planning construct. In this work, we explored the possibilities, as well as the obstacles of applying a risk aware HTN Planning technique to this domain. We successfully created a model of the game, which a planner can use to generate courses of actions according to specific risk attitudes, which can lead an AI through the hands that are played. We developed two different approaches, both with their own advantages and disadvantages. In the end, the approach which creates plans for the whole game, instead of just one turn turned out to be both more practical, as well as strategically diverse. This model did manage to create plans, that implemented a playstyle which comes close to strategy suggestions by experts. As we wanted to see the potential of HTN planning, we tried to refrain from putting too many strategic guidelines into the model itself and rely mainly on the risk attitude for playstyle. However, despite our best efforts, the model created does have shortcomings in various aspects. The biggest issue we see, is the general opponent modelling we used, which does not differentiate opponents by their playstyles, but rather assumes each player is basing their own decisions purely on mathematics. An additional assumption our model makes which would not resemble real games completely, is that we do not include any interaction between multiple opponents without the involvement of our agent. This loses a big dynamic of the game, and does limit strategic options, as well as reducing planning variety. With that said, we feel like we could demonstrate the potential for risk aware HTN Planning. By using a decent amount of domain expert knowledge in combination with already established concepts and algorithms, we were able to create a model which can create reasonable plans for a specific domain. By improving the model in some areas, most importantly by incorporating more sophisticated opponent modelling techniques, possibly powered by AI themselves, we feel like HTN Planning with risk can be a powerful option to create fully automated poker playing agents. Additionally we can see a great potential of using the Planning framework itself, as well as the presented planner implementation for other domains, which feature a great amount of uncertainty as well.

7.1 Appendix A. Complete Domain Model Images

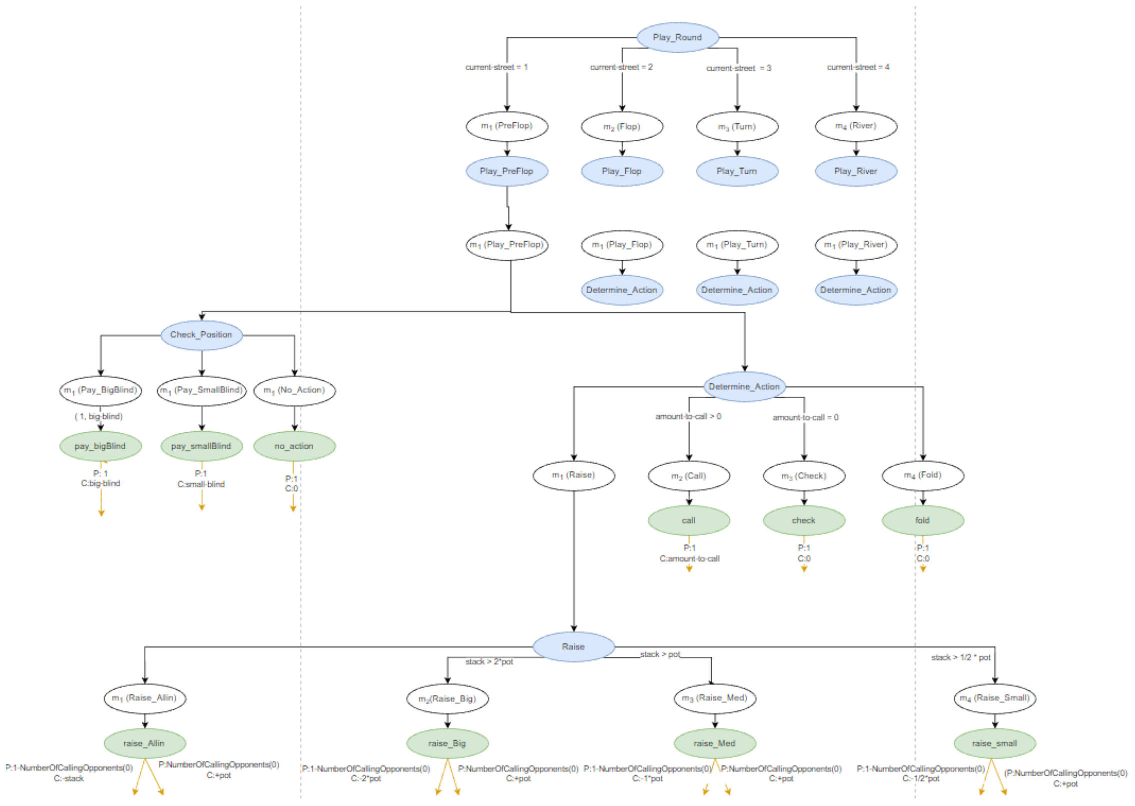


Figure 7.1: The Complete one turn domain Model, duplicate compound tasks are not extended for visibility

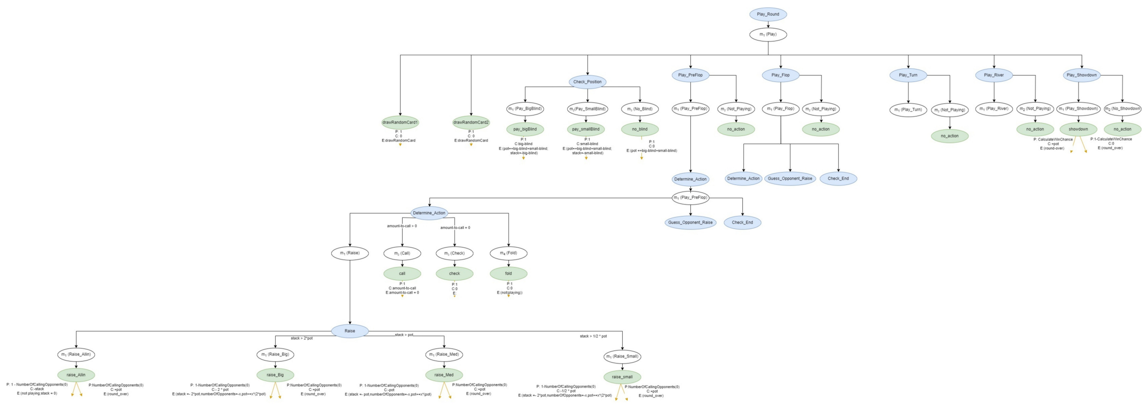


Figure 7.2: The Complete whole-round domain Model, duplicate compound tasks are not extended for visibility

7.2 Appendix B. Complete Domain Description

7.2.1 One round approach

```
(defdomain Texas_Hold_EM_Round (
(:operator (!fold)
  ((playing))
  (( 1 ((playing))))
  (( 1 ((done-action))))
  ((1 0)))
(:operator (!check)
  ((playing)(amount-to-call ?a)(call = ?a 0))
  ((1 ((done-action))))
  ((1 ((done-action))))
  ((1 0))
  )
(:operator (!pay_big_blind )
  ((stack ?s) (pot ?p) (big-blind ?bb)(small-blind ?sb))
  ((1 ((pot ?p)(stack ?s))))
  ((1 ((pot (call +  ?p (call + ?sb ?bb)))(stack (call - ?s ?bb)))))
  ((1 (call * ?bb -1))))

(:operator (!pay_small_blind)
  ((playing) (pot ?p)(stack ?s)(small-blind ?sb)(big-blind ?bb) (  ?stack (call - ?s ?sb)))
  (( 1 ((pot ?p)(stack ?s))))
  (( 1 ((pot (call +  ?p (call + ?bb ?sb)))(stack (call - ?s ?sb)))))
  (( 1 (call * ?sb -1))))

(:operator (!call_bet )
  ((playing)(pot ?p)(stack ?s)(amount-to-call ?a)(call > ?stack ?a))
  (( 1 ((pot ?p)(stack ?s))))
  ((1 ((done-action)(amount-to-call 0)(pot (call + ?p ?a))(stack (call - ?s ?a)))))
  (( 1 (call * ?a -1))))

(:operator (!raise_all_in )
  ((stack ?s)(call > ?s 0)(pot ?p) (numberOfPlayers ?nop))
  ((1 ((pot ?p)(stack ?s))))
  ((1((done-action)(amount-to-call 0) (pot (call + ?p ?s))(stack 0))))
  (((call - 1 (call NumberOfCallingOpponents ?p ?s  ?nop 0)) (call * ?s -1))((call
NumberOfCallingOpponents ?p ?s ?nop 0) ?p)))

(:operator (!raise_big )
  ((numberOfPlayers ?nop)(stack ?s)(pot ?p)(call > ?stack (call * ?p 2)))
  (( 1 ((pot ?p)(stack ?s))))
  ((1 ((done-action)(amount-to-call 0) (pot (call + ?p (call * 2 ?p)))(stack (call - ?s (call
* ?p 2))))))
  (((call - 1 (call NumberOfCallingOpponents ?p (call * ?p 2) ?nop 0)) (call * ?p -2))((call
NumberOfCallingOpponents ?p (call * ?p 2) ?nop 0) ?p)))

(:operator (!raise_med )
  ((numberOfPlayers ?nop)(stack ?s)(pot ?p)(call > ?stack ?p))
  (( 1 ((pot ?p)(stack ?s))))
  ((1 ((done-action)(amount-to-call 0) (pot (call + ?p ?p))(stack (call - ?s ?p)))))
  (((call - 1 (call NumberOfCallingOpponents ?p ?p ?nop 0)) (call * -1 ?p))((call
NumberOfCallingOpponents ?p ?p ?nop 0) ?p)))
(:operator (!raise_small )
  ((numberOfPlayers ?nop)(stack ?s)(pot ?p)(call > ?stack (call * ?p 0.5)))
  (( 1 ((pot ?p)(stack ?s))))
  ((1 ((done-action)(amount-to-call 0) (pot (call + ?p (call * 0.5 ?p)))(stack (call - ?s (
call * ?p 0.5))))))
  (((call - 1 (call NumberOfCallingOpponents ?p (call * ?p 0.5) ?nop 0)) (call * ?p -0.5))((
call NumberOfCallingOpponents ?p (call * ?p 0.5) ?nop 0) ?p)))
(:operator (!no_action)
  ()
  ()
  ()
  ((1 0)))
(:operator (!no_blind)
  ((playing)(big-blind ?bb)(small-blind ?sb)(amount-to-call ?a) (pot ?p))
  ((1 ((pot ?p)(amount-to-call ?a))))
  (( 1 ((pot (call +  ?p (call + ?sb ?bb)))(amount-to-call (call + ?sb ?bb)))))
  (( 1 0 )))


;; ------------------------------------- methods
(:method (check_position)
  bigBlind
    ( (position ?p) (call = ?p 1))
    ((!pay_big_blind))
  smallBlind
    ( (position ?p) (call = ?p 2))
    ((!pay_small_blind))
  other
    ( (position ?p) (call > ?p 2))
    ((!no_blind)))
(:method (play_round)
  PreFlop
    ((current-street ?cs)(call = ?cs 1))
    ((play_pre_flop))
  Flop
    ((current-street ?cs)(call = ?cs 2))
    ((play_flop))
  Turn
    ((current-street ?cs)(call = ?cs 3))
    ((play_turn))
  River
    ((current-street ?cs)(call = ?cs 4))
    ((play_river))
  )
(:method (determine_action )
  Fold
  ((amount-to-call ?a)(call > ?a 0))
  ((!fold)))

(:method (determine_action)
  CallBet
  ((amount-to-call ?a)(call > ?a 0))
  ((!call_bet)))

(:method (determine_action )
  Check
  ((amount-to-call ?a)(call = ?a 0))
  ((!check))
  )
(:method (determine_action)
  Raise
  ()
  ((determine_raise_amount)))
(:method (determine_raise_amount)
  RaiseBig
  ()
  ((!raise_big)))
(:method (determine_raise_amount)
  RaiseMedium
  ()
  ((!raise_med)))
(:method (determine_raise_amount)
  RaiseSmall
  ()
  ((!raise_small)))
(:method (determine_raise_amount)
  RaiseAllIn
  ()
  ((!raise_all_in))
  )
(:method (play_pre_flop)
  PlayPreFlop
  ()
  ((check_position)(determine_action)))
(:method (play_flop )
  PlayFlop
  ()
  ((determine_action)))
(:method (play_turn)
  PlayTurn
  ()
  ((determine_action)))
(:method (play_river)
  PlayRiver
  ()
  ((determine_action)))
)
)
```

Listing 7.1: one turn domain description

7.2.2 Whole round approach

```
(defdomain Texas_Hold_EM_Round (
(:operator (!fold)
  ((playing))
  (( 1 ((playing))))
  (( 1 ((done-action)(folded)(not_playing))))
  ((1 0)))
(:operator (!check)
  ((playing)(amount-To-Call ?a)(call = ?a 0))
  ((1 ((done-action))))
  ((1 ((done-action))))
  ((1 0))
  )
(:operator (!pay_big_blind)
  ((playing) (stack ?s)(big-blind ?bb)(small-blind ?sb) (pot ?p))
  ((1 ((pot ?p)(stack ?s))))
  (( 1 ((pot (call +  ?p (call + ?sb ?bb)))(stack (call - ?s ?bb)))))
  (( 1 (call * ?bb -1))))

(:operator (!pay_small_blind )
  ((playing) (stack ?s)(small-blind ?sb)(big-blind ?bb) (pot ?p) )
  ((1 ((pot ?p)(stack ?s))))
  (( 1 ((pot (call +  ?p (call + ?bb ?sb)))(stack (call - ?s ?sb)))))
  (( 1 (call * ?sb -1))))



(:operator (!call_bet )
  ((playing)(pot ?p)(amount-to-call ?a)(call > ?a 0) (stack ?s))
  ((1 ((pot ?p)(stack ?s))))
  ((1 ((done-action) (amount-to-call 0) (stack (call - ?s ?a)) (pot (call + ?p ?a)))))
  (( 1 (call * ?a -1))))

(:operator (!raise_all_in )
  ((stack ?s)(pot ?p)(current-street ?cs)(call > ?p 0)(numberOfPlayers ?nop)(amount-to-call ?a
))
    ((1 ((pot ?p)(stack ?s)(numberOfPlayers ?nop)(amount-to-call ?a))))
    (( (call NumberOfCallingOpponents ?p ?s ?nop 0) ((done-action)(amount-to-call 0)(pot 0)(
numberOfPlayers 0) (round-over)(stack (call + ?s ?p))))
    ( (call NumberOfCallingOpponents ?p ?s ?nop 1) ((done-action)(not_playing)(amount-to-call
0)(numberOfPlayers 1)(pot (call + ?p (call * 2 ?s)))(stack (call - ?s ?s))))
    ( (call NumberOfCallingOpponents ?p ?s ?nop 2) ((done-action)(not_playing)(amount-to-call
0)(numberOfPlayers 2)(pot (call + ?p (call * 3 ?s)))(stack (call - ?s ?s))))
    ( (call NumberOfCallingOpponents ?p ?s ?nop 3) ((done-action)(not_playing)(amount-to-call
0)(numberOfPlayers 3)(pot (call + ?p (call * 4 ?s)))(stack (call - ?s ?s))))
    ( (call NumberOfCallingOpponents ?p ?s ?nop 4) ((done-action)(not_playing)(amount-to-call
0)(numberOfPlayers 4)(pot (call + ?p (call * 5 ?s)))(stack (call - ?s ?s))))
    ( (call NumberOfCallingOpponents ?p ?s ?nop 5) ((done-action)(not_playing)(amount-to-call
0)(numberOfPlayers 5)(pot (call + ?p (call * 6 ?s)))(stack (call - ?s ?s))))
    ( (call NumberOfCallingOpponents ?p ?s ?nop 6) ((done-action)(not_playing)(amount-to-call
0)(numberOfPlayers 6)(pot (call + ?p (call * 7 ?s)))(stack (call - ?s ?s))))
    ( (call NumberOfCallingOpponents ?p ?s ?nop 7) ((done-action)(not_playing)(amount-to-call
0)(numberOfPlayers 7)(pot (call + ?p (call * 8 ?s)))(stack (call - ?s ?s))))
```

```
    )
  (((call - 1 (call NumberOfCallingOpponents ?p ?s ?nop 0)) (call * ?s -1))((call
NumberOfCallingOpponents ?p ?s ?nop 0) ?p))
)

(:operator (!raise_big )
  ((stack ?s)(pot ?p) (current-street ?cs)(amount-to-call ?a)(call > ?p 0)  (call > ?s (call *
 2 ?p))(numberOfPlayers ?nop))
  ((1 ((pot ?p)(stack ?s)(numberOfPlayers ?nop)(amount-to-call ?a))))
  (( (call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop 0) ((done-action)(amount-to-call 0)
(pot 0)(numberOfPlayers 0) (round-over)(stack (call + ?s ?p))))
    ( (call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop 1) ((done-action)(amount-to-call 0)
(numberOfPlayers 1)(pot (call + ?p (call * 2 (call * 2 ?p))))(stack (call - ?s (call * 2 ?p)))
))
    ( (call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop 2) ((done-action)(amount-to-call 0)
(numberOfPlayers 2)(pot (call + ?p (call * 3 (call * 2 ?p))))(stack (call - ?s (call * 2 ?p)))
))
    ( (call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop 3) ((done-action)(amount-to-call 0)
(numberOfPlayers 3)(pot (call + ?p (call * 4 (call * 2 ?p))))(stack (call - ?s (call * 2 ?p)))
))
    ( (call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop 4) ((done-action)(amount-to-call 0)
(numberOfPlayers 4)(pot (call + ?p (call * 5 (call * 2 ?p))))(stack (call - ?s (call * 2 ?p)))
))
    ( (call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop 5) ((done-action)(amount-to-call 0)
(numberOfPlayers 5)(pot (call + ?p (call * 6 (call * 2 ?p))))(stack (call - ?s (call * 2 ?p)))
))
    ( (call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop 6) ((done-action)(amount-to-call 0)
(numberOfPlayers 6)(pot (call + ?p (call * 7 (call * 2 ?p))))(stack (call - ?s (call * 2 ?p)))
))
    ( (call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop 7) ((done-action)(amount-to-call 0)
(numberOfPlayers 7)(pot (call + ?p (call * 8 (call * 2 ?p))))(stack (call - ?s (call * 2 ?p)))
))
    )
  (((call - 1 (call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop 0)) (call * (call * 2 ?p)
-1))((call NumberOfCallingOpponents ?p (call * 2 ?p) ?nop ?cs) ?p)))



(:operator (!raise_med )
  ((stack ?s)(pot ?p)(amount-to-call ?a)(call > ?s ?p)(call > ?p 0)(numberOfPlayers ?nop))
  ((1 ((pot ?p)(stack ?s)(numberOfPlayers ?nop)(amount-to-call ?a))))
  (( (call NumberOfCallingOpponents ?p ?p ?nop 0) ((done-action)(amount-to-call 0)(pot 0)(
numberOfPlayers 0) (round-over)(stack (call + ?s ?p))))
    ( (call NumberOfCallingOpponents ?p ?p ?nop 1) ((done-action)(amount-to-call 0)(
numberOfPlayers 1)(pot (call + ?p (call * 2 ?p)))(stack (call - ?s ?p))))
    ( (call NumberOfCallingOpponents ?p ?p ?nop 2) ((done-action)(amount-to-call 0)(
numberOfPlayers 2)(pot (call + ?p (call * 3 ?p)))(stack (call - ?s ?p))))
    ( (call NumberOfCallingOpponents ?p ?p ?nop 3) ((done-action)(amount-to-call 0)(
numberOfPlayers 3)(pot (call + ?p (call * 4 ?p)))(stack (call - ?s ?p))))
    ( (call NumberOfCallingOpponents ?p ?p ?nop 4) ((done-action)(amount-to-call 0)(
numberOfPlayers 4)(pot (call + ?p (call * 5 ?p)))(stack (call - ?s ?p))))
    ( (call NumberOfCallingOpponents ?p ?p ?nop 5) ((done-action)(amount-to-call 0)(
numberOfPlayers 5)(pot (call + ?p (call * 6 ?p)))(stack (call - ?s ?p))))
```

```
    ( (call NumberOfCallingOpponents ?p ?p ?nop 6) ((done-action)(amount-to-call 0)(
numberOfPlayers 6)(pot (call + ?p (call * 7 ?p)))(stack (call - ?s ?p))))
    ( (call NumberOfCallingOpponents ?p ?p ?nop 7) ((done-action)(amount-to-call 0)(
numberOfPlayers 7)(pot (call + ?p (call * 8 ?p)))(stack (call - ?s ?p))))
    )
  (((call - 1 (call NumberOfCallingOpponents ?p ?p ?nop 0)) (call * ?p -1))((call
NumberOfCallingOpponents ?p ?p ?nop 0) ?p)))

(:operator (!raise_small)
  ((stack ?s)(pot ?p)(amount-to-call ?a)(call > ?p 0)(call > ?s (call * 0.5 ?p))(
numberOfPlayers ?nop))
  ((1 ((pot ?p)(stack ?s)(numberOfPlayers ?nop)(amount-to-call ?a))))
  (( (call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 0) ((done-action)(numberOfPlayers
0)(pot 0) (round-over)(stack (call + ?s ?p))))
    ( (call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 1) ((done-action)(amount-to-call
0)(numberOfPlayers 1)(pot (call + ?p (call * 2 (call * 0.5 ?p))))(stack (call - ?s (call * 0.5
 ?p)))))
    ( (call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 2) ((done-action)(amount-to-call
0)(numberOfPlayers 2)(pot (call + ?p (call * 3 (call * 0.5 ?p))))(stack (call - ?s (call * 0.5
 ?p)))))
    ( (call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 3) ((done-action)(amount-to-call
0)(numberOfPlayers 3)(pot (call + ?p (call * 4 (call * 0.5 ?p))))(stack (call - ?s (call * 0.5
 ?p)))))
    ( (call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 4) ((done-action)(amount-to-call
0)(numberOfPlayers 4)(pot (call + ?p (call * 5 (call * 0.5 ?p))))(stack (call - ?s (call * 0.5
 ?p)))))
    ( (call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 5) ((done-action)(amount-to-call
0)(numberOfPlayers 5)(pot (call + ?p (call * 6 (call * 0.5 ?p))))(stack (call - ?s (call * 0.5
 ?p)))))
    ( (call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 6) ((done-action)(amount-to-call
0)(numberOfPlayers 6)(pot (call + ?p (call * 7 (call * 0.5 ?p))))(stack (call - ?s (call * 0.5
 ?p)))))
    ( (call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 7) ((done-action)(amount-to-call
0)(numberOfPlayers 7)(pot (call + ?p (call * 8 (call * 0.5 ?p))))(stack (call - ?s (call * 0.5
 ?p)))))
    )
  (((call - 1 (call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 0)) (call * (call * 0.5 ?
p) -1))((call NumberOfCallingOpponents ?p (call * 0.5 ?p) ?nop 0) ?p)))

(:operator (!no_action)
  ((pot ?p))
  ((1 ((pot ?p))))
  ((1((pot ?p))))
  ((1 0)))
(:operator (!no_blind)
  ((playing)(big-blind ?bb)(small-blind ?sb)(amount-To-Call ?a) (pot ?p))
  ((1 ((pot ?p)(amount-To-Call ?a))))
  (( 1 ((pot (call +  ?p (call + ?sb ?bb)))(amount-To-Call (call + ?sb ?bb)))))
  (( 1 0 )))
(:operator (!guess_opponent_raise)
  ((numberOfPlayers ?nop) (amount-To-Call ?a) (pot ?p)(call > ?p 0)(call > ?nop 0)(enemy-
aggression ?ea))
  ((1 ((amount-To-Call ?a))))
  (( ( call - 1 (call ^ 0.95 ?nop))((amount-To-Call ?p)))
  ((call ^ 0.95 ?nop)((amount-To-Call 0))))
  (( 1 0 )))
(:operator(!showdown)
  ((numberOfPlayers ?nop) (stack ?s) (pot ?p) (card1 ?c1)(card2 ?c2)(communitycard1 ?cc1)(
communitycard2 ?cc2)(communitycard3 ?cc3)(communitycard4 ?cc4)(communitycard5 ?cc5))
  ((1 ((round-over))))
  (( 1 ((round-over))))
  (((call HandRank ?c1 ?c2 ?cc1 ?cc2 ?cc3 ?cc4 ?cc5 ?nop)?p)((call - 1 (call HandRank  ?c1 ?c2
 ?cc1 ?cc2 ?cc3 ?cc4 ?cc5 ?nop)) 0 )))
(:operator(!drawRandomCard1)
((card1 ?c1)(card2 ?c2)(communitycard1 ?cc1)(communitycard2 ?cc2)(communitycard3 ?cc3)(
communitycard4 ?cc4)(communitycard5 ?cc5))
((1 ((card1 ?c1 ))))
((1 ((card1 (call DrawCard ?c1 ?c2 ?cc1 ?cc2 ?cc3 ?cc4 ?cc5)))))
((1 0))
)
(:operator(!drawRandomCard2)
((card1 ?c1)(card2 ?c2)(communitycard1 ?cc1)(communitycard2 ?cc2)(communitycard3 ?cc3)(
communitycard4 ?cc4)(communitycard5 ?cc5))
((1 ((card2 ?c2 ))))
((1 ((card2 (call DrawCard ?c1 ?c2 ?cc1 ?cc2 ?cc3 ?cc4 ?cc5)))))
((1 0))
)
(:operator(!drawRandomCommunityCard1)
((card1 ?c1)(card2 ?c2)(communitycard1 ?cc1)(communitycard2 ?cc2)(communitycard3 ?cc3)(
communitycard4 ?cc4)(communitycard5 ?cc5)(call = ?cc1 0))
((1 ((communitycard1 ?cc1 ))))
((1 ((communitycard1 (call DrawCard ?c1 ?c2 ?cc1 ?cc2 ?cc3 ?cc4 ?cc5)))))
((1 0))
)
(:operator(!drawRandomCommunityCard2)
((card1 ?c1)(card2 ?c2)(communitycard1 ?cc1)(communitycard2 ?cc2)(communitycard3 ?cc3)(
communitycard4 ?cc4)(communitycard5 ?cc5)(call = ?cc2 0))
((1 ((communitycard2 ?cc2 ))))
((1 ((communitycard2 (call DrawCard ?c1 ?c2 ?cc1 ?cc2 ?cc3 ?cc4 ?cc5)))))
((1 0))
)
(:operator(!drawRandomCommunityCard3)
((card1 ?c1)(card2 ?c2)(communitycard1 ?cc1)(communitycard2 ?cc2)(communitycard3 ?cc3)(
communitycard4 ?cc4)(communitycard5 ?cc5)(call = ?cc3 0))
((1 ((communitycard3 ?cc3))))
((1 ((communitycard3 (call DrawCard ?c1 ?c2 ?cc1 ?cc2 ?cc3 ?cc4 ?cc5)))))
((1 0))
)
(:operator(!drawRandomCommunityCard4)
((card1 ?c1)(card2 ?c2)(communitycard1 ?cc1)(communitycard2 ?cc2)(communitycard3 ?cc3)(
communitycard4 ?cc4)(communitycard5 ?cc5)(call = ?cc4 0))
((1 ((communitycard4 ?cc4 ))))
((1 ((communitycard4 (call DrawCard ?c1 ?c2 ?cc1 ?cc2 ?cc3 ?cc4 ?cc5)))))
((1 0))
)
(:operator(!drawRandomCommunityCard5)
((card1 ?c1)(card2 ?c2)(communitycard1 ?cc1)(communitycard2 ?cc2)(communitycard3 ?cc3)(
communitycard4 ?cc4)(communitycard5 ?cc5)(call = ?cc5 0))
((1 ((communitycard5 ?cc5 ))))
((1 ((communitycard5 (call DrawCard ?c1 ?c2 ?cc1 ?cc2 ?cc3 ?cc4 ?cc5)))))
((1 0))
)


;; ------------------------------------- methods
(:method (check_position)
  bigBlind
    ((position ?p)(call = ?p 1))
    ((!pay_big_blind))
  smallBlind
    ((position ?p)(call = ?p 2))
    ((!pay_small_blind))
  other
    ((position ?p)(call > ?p 2))
    ((!no_blind)))
(:method (play_round)
  ()
  ((!drawRandomCard1)(!drawRandomCard2)(check_position)(play_pre_flop)(play_flop)(play_turn)(
play_river)(play_Showdown)))
(:method (determine_action)
  Raise
  ()
  ((determine_raise_amount )))
(:method (determine_action)
  CallBet
  ((amount-to-call ?a) (call > ?a 0))
  ((!call_bet)))
(:method (determine_action)
  Fold
  ()
  ((!fold)))
(:method (determine_action)
  Check
  ((amount-to-call ?a) (call = ?a 0))
  ((!check ))
  )
(:method (determine_raise_amount)
  RaiseBig
  ()
  ((!raise_big)))
(:method (determine_raise_amount)
  RaiseMedium
  ()
  ((!raise_med)))
(:method (determine_raise_amount)
  RaiseSmall
  ()
  ((!raise_small)))
(:method (determine_raise_amount)
  RaiseAllIn
  ()
  ((!raise_all_in))
  )
(:method (play_pre_flop)
  PlayPreFlop
  ((playing)(not (round-over))(stack ?s)(call > ?s 0))
  ((determine_action)(guess_opponent_raise)(check_end_PreFlop)))
(:method (play_pre_flop)
  NotPlaying
  ((or(not_playing)(round-over)))
  ((!no_action)))
(:method (play_flop)
  PlayFlop
  ((playing)(not (round-over))(stack ?s)(call > ?s 0))
  ((!drawRandomCommunityCard1)(!drawRandomCommunityCard2)(!drawRandomCommunityCard3)(
determine_action)(guess_opponent_raise)(check_end_Flop)))
(:method (play_flop)
  NotPlaying
  ((or(not_playing)(round-over)))
  ((!drawRandomCommunityCard1)(!drawRandomCommunityCard2)(!drawRandomCommunityCard3)(!
no_action)))
(:method (play_turn)
  PlayTurn
  ((playing)(not (round-over))(stack ?s)(call > ?s 0))
  ((!drawRandomCommunityCard4)(determine_action)(guess_opponent_raise)(check_end_Turn)))
(:method (play_turn)
  NotPlaying
  ((or(not_playing)(round-over)))
  ((!drawRandomCommunityCard4)(!no_action)))
(:method (play_river)
  PlayRiver
  ((playing)(not (round-over))(stack ?s)(call > ?s 0))
  ((!drawRandomCommunityCard5)(determine_action)(guess_opponent_raise)(check_end_River)))
(:method (play_river)
  NotPlaying
  ((or(not_playing)(round-over)))
  ((!drawRandomCommunityCard5)(!no_action)))
(:method (check_end_PreFlop)
  End
  ((amount-To-Call ?a)(call = ?a 0))
  ((!no_action))
  Not_End
  ((amount-To-Call ?a)(call > ?a 0))
  ((determine_action)(check_end_PreFlop))
)
(:method (check_end_Flop)
  End
  ((amount-To-Call ?a)(call = ?a 0))
  ((!no_action))
  Not_End
  ((amount-To-Call ?a)(call > ?a 0))
  ((determine_action)(check_end_Flop))
  )
(:method (check_end_Turn)
  End
  ((amount-To-Call ?a)(call = ?a 0))
  ((!no_action))
  Not_End
  ((amount-To-Call ?a)(call > ?a 0))
  ((determine_action)(check_end_Turn))
  )
(:method (check_end_River)
  End
  ((amount-To-Call ?a)(call = ?a 0))
  ((!no_action))
  Not_End
  ((amount-To-Call ?a)(call > ?a 0))
  ((determine_action)(check_end_River))
)
(:method (guess_opponent_raise)
  NoOpponentsLeft
  ((numberOfPlayers ?nop)(or ((call = ?nop 0)(round-over))))
  ((!no_action))
  OpponentsLeft
  ((numberOfPlayers ?nop)(call > ?nop 0)(playing)(not(folded)))
  ((!guess_opponent_raise)))

(:method (play_Showdown)
  NoShowdown
  ((numberOfPlayers ?nop)(or (call = ?nop 0)(folded)(round-over)))
  ((!no_action))
  Showdown
  ((numberOfPlayers ?nop)(call > ?nop 0)(playing))
  ((!showdown)))
)
)
```

Listing 7.2: whole round domain description

Bibliography

[AGA22] E. Alnazer, I. Georgievski, M. Aiello. “Risk Awareness in HTN Planning”. In: *arXiv preprint arXiv:2204.10669* (2022) (cit. on pp. 16, 25, 40).

[Aln19] E. Alnazer. “HTN Planning with Utilities”. MA thesis. 2019 (cit. on pp. 16, 29, 37).

[BDSS02] D. Billings, A. Davidson, J. Schaeffer, D. Szafron. “The challenge of poker”. In: *Artificial Intelligence* 134.1-2 (2002), pp. 201–240 (cit. on p. 21).

[Bje10] O. Bjerg. “Problem gambling in poker: Money, rationality and control in a skill-based social game”. In: *International Gambling Studies* 10.3 (2010), pp. 239–254 (cit. on p. 18).

[Bou14] N. Boudewijn. “Opponent Modeling in Texas Hold’em”. B.S. thesis. 2014 (cit. on p. 51).

[BPSS98a] D. Billings, D. Papp, J. Schaeffer, D. Szafron. “Opponent Modeling in Poker.” In: Jan. 1998, pp. 493–499 (cit. on p. 51).

[BPSS98b] D. Billings, D. Papp, J. Schaeffer, D. Szafron. “Poker as a Testbed for Machine Intelligence Research”. In: 1998 (cit. on p. 15).

[BSM17] N. Brown, T. Sandholm, S. Machine. “Libratus: The Superhuman AI for No-Limit Poker.” In: *IJCAI*. 2017, pp. 5226–5228 (cit. on p. 51).

[DB14] H. Dan, R. Bill. *Harrington On Modern Tournament Poker*. Two Plus Two Publishing LLC, 2014 (cit. on pp. 18–21).

[DBSS00] A. Davidson, D. Billings, J. Schaeffer, D. Szafron. “Improved opponent modeling in poker”. In: *International Conference on Artificial Intelligence, ICAI’00*. 2000, pp. 1467–1473 (cit. on p. 21).

[FKP12] G. Fedczyszyn, L. Koszalka, I. Pozniak-Koszalka. “Opponent modeling in Texas Hold’em poker”. In: *Computational Collective Intelligence. Technologies and Applications: 4th International Conference, ICCCI 2012, Ho Chi Minh City, Vietnam, November 28-30, 2012, Proceedings, Part II 4*. Springer. 2012, pp. 182–191 (cit. on p. 51).

[GA15] I. Georgievski, M. Aiello. “HTN planning: Overview, comparison, and beyond”. In: *Artificial Intelligence* 222 (2015), pp. 124–156 (cit. on pp. 25, 29).

[GL14] I. Georgievski, A. Lazovik. “Utility-based HTN planning”. In: *ECAI 2014*. IOS Press, 2014, pp. 1013–1014 (cit. on p. 16).

[GNN+17] I. Georgievski, T. A. Nguyen, F. Nizamic, B. Setz, A. Lazovik, M. Aiello. “Planning meets activity recognition: Service coordination for intelligent buildings”. In: *Pervasive and Mobile Computing* 38 (2017), pp. 110–139. issn: 1574-1192 (cit. on p. 25).

[HBBB20] D. Höller, P. Bercher, G. Behnke, S. Biundo. “HTN plan repair via model transformation”. In: *KI 2020: Advances in Artificial Intelligence: 43rd German Conference on AI, Bamberg, Germany, September 21–25, 2020, Proceedings 43*. Springer. 2020, pp. 88–101 (cit. on p. 29).

[Ilg06] O. Ilghami. “Documentation for JSHOP2”. In: *Department of Computer Science, University of Maryland, Tech. Rep* (2006), pp. 41–42 (cit. on p. 37).

[IN03] O. Ilghami, D. S. Nau. *A general approach to synthesize problem-specific planners*. Tech. rep. MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE, 2003 (cit. on p. 37).

[Kuh50] H. W. Kuhn. “A simplified two-person poker”. In: *Contributions to the Theory of Games* 1 (1950), pp. 97–103 (cit. on p. 51).

[LZZ12] J. Luo, C. Zhu, W. Zhang. “Messy Genetic Algorithm for the Optimum Solution Search of the HTN Planning”. In: *Foundations of Intelligent Systems: Proceedings of the Sixth International Conference on Intelligent Systems and Knowledge Engineering, Shanghai, China, Dec 2011 (ISKE2011)*. Springer. 2012, pp. 93–98 (cit. on p. 52).

[McD82] D. McDermott. “A temporal logic for reasoning about processes and plans”. In: *Cognitive Science* 6.2 (1982), pp. 101–155. ISSN: 0364-0213 (cit. on p. 15).

[MCM12] S. Magnenat, J.-C. Chappelier, F. Mondada. “Integration of online learning into HTN planning for robotic tasks”. In: *2012 AAAI Spring Symposium Series*. 2012 (cit. on p. 51).

[MMS+18] F. Meneguzzi, M. C. Magnaguagno, M. P. Singh, P. R. Telang, N. Yorke-Smith. “Goco: Planning expressive commitment protocols”. In: *Autonomous Agents and Multi-Agent Systems* 32.4 (2018), pp. 459–502 (cit. on p. 52).

[NCLM99] D. Nau, Y. Cao, A. Lotem, H. Munoz-Avila. “SHOP: Simple hierarchical ordered planner”. In: *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*. 1999, pp. 968–973 (cit. on p. 16).

[Pok21] PokerListings. *5 Drastic Ways Poker Strategy Has Changed Since 2010*. https://www.pokerlistings.com/5-drastic-ways-poker-strategy-has-changed-since-2010. Accessed: 19-12-2022. 2021 (cit. on p. 18).

[PQ95] T. J. Parr, R. W. Quong. “ANTLR: A predicated-LL (k) parser generator”. In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810 (cit. on p. 38).

[SBPS99] J. Schaeffer, D. Billings, L. Peña, D. Szafron. “Learning to play strong poker”. In: *The International Conference on Machine Learning Workshop on Game Playing*. Vol. 4. 1999 (cit. on p. 51).

[Tur53] A. M. Turing. “Digital computers applied to games”. In: *Faster than thought* (1953) (cit. on p. 15).

[Van10] A. Van der Kleij. “Monte Carlo tree search and opponent modeling through player clustering in no-limit Texas hold’em poker”. In: *University of Groningen, The Netherlands* (2010) (cit. on p. 51).

All links were last followed on May 18, 2023.

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.

place, date, signature