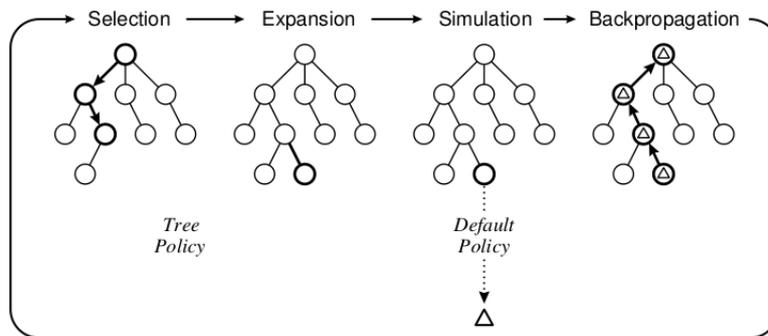


School of Electronic and Computer Engineering

Technical University of Crete

Real-time Planning and Learning in the “Settlers of Catan”
Strategy Game



Thesis

Konstantinos Panagiotis Panousis

Commitee:

Georgios Chalkiadakis, Assistant Professor-Supervisor

Antonios Deligiannakis, Associate Professor

Michail G. Lagoudakis, Associate Professor

Real-time Planning and Learning in the “Settlers of Catan” Strategy Game

Abstract

MONTE CARLO TREE SEARCH (MCTS) is a method for making optimal decisions in a given domain, taking random actions to transit from one state to another and building a tree according to the results. After the successful application of MCTS in the two-player, perfect information game of Go, researchers are trying to understand better when and why MCTS succeeds and when and why it fails. In this thesis, we apply MCTS in the Settlers of Catan Strategy game, which is a non-deterministic, partially observable, multi-player strategic board game. We implement an agent to play against existing AI and human players in the JSettlers interface. We show that the algorithm can easily adjust to multi-agent environments and we present three different enhancements -UCT, BAYESIAN UCT and VALUE OF PERFECT INFORMATION- in the tree policy of the main algorithm. This is the first time that the BAYESIAN UCT enhancement on MONTE CARLO TREE SEARCH is used in Settlers of Catan and the first time that the VALUE OF PERFECT INFORMATION is used on MCTS in general. We also implemented a simple negotiation scheme to give the agent the ability to trade with other players and created various strategies to cope with situations of the game. Our results suggest that the agent benefits from the enhancement of the tree policy and from the use of refined methods that balance the exploration -exploitation dilemma leading to good performance of the agent against one of the strongest heuristic-based implementations.

Σχεδιασμός και Μάθηση σε Πραγματικό Χρόνο για το παιχνίδι στρατηγικής Άποικοι του Κατάν

Περίληψη

Ο αλγόριθμος MONTE CARLO TREE SEARCH (MCTS) είναι μια γενική μέθοδος για την λήψη βέλτιστων αποφάσεων. Η μέθοδος αξιοποιεί τη λήψη (ουσιαστικά τυχαίων) δειγμάτων από τις πιθανές ενέργειες, και δημιουργεί ένα δέντρο αποφάσεων, μέσω του οποίου αναζητείται η βέλτιστη απόφαση.

Μετά την επιτυχημένη εφαρμογή της μεθόδου, στο παιχνίδι -δύο παικτών και τέλει πληροφορίας- Go, και τις προσδοκίες που δημιούργησε, η επαρκής κατανόηση των πλεονεκτημάτων και των αδυναμιών του αλγορίθμου είναι ένα ζητούμενο.

Στην εργασία αυτή, εφαρμόζουμε τον αλγόριθμο MCTS, στο επιτραπέζιο παιχνίδι στρατηγικής Άποικοι του Κατάν, ένα παιχνίδι πολλών παικτών, μη-ντετερμινιστικό και μερικώς παρατηρήσιμο.

Αναπτύσσουμε και αξιολογούμε τρεις διαφορετικές παραλλαγές στο κομμάτι της δημιουργίας του δέντρου του αλγορίθμου: συγκεκριμένα τη μέθοδο UCT, τη μέθοδο BAYESIAN UCT και τη μέθοδο VALUE OF PERFECT INFORMATION (VPI).

Οι αλγόριθμοι αυτοί κατ' ουσίαν επιχειρούν να ισορροπήσουν το δίλημμα μεταξύ εξερεύνησης (exploration) και εκμετάλλευσης (exploitation) στο συγκεκριμένο τομέα.

Επιπρόσθετα, δημιουργήσαμε διάφορες ευριστικές στρατηγικές για να μπορεί ο πράκτορας μας να ανταπεξέλθει σε συγκριμένες καταστάσεις που μπορούν να εμφανιστούν και οι οποίες απορρέουν από τους κανόνες του παιχνιδιού· σε αντίθεση με τους περισσότερους αυτοματοποιημένους παίχτες για τους Αποίκους του Κατάν, η υλοποίησή μας προσφέρει ένα (έστω απλό) σχέδιο διαπραγμάτευσης για να έχει ο πράκτορας μας τη δυνατότητα να ανταλλάσει πόρους με άλλους παίχτες.

Αξίζει να σημειωθεί ότι είναι η πρώτη φορά που η μέθοδος BAYESIAN UCT χρησιμοποιείται στον αλγόριθμο MCTS στο παιχνίδι Άποικοι του Κατάν και είναι επίσης η πρώτη φορά που η μέθοδος VPI χρησιμοποιείται με τον αλγόριθμο MCTS γενικότερα.

Δοκιμάζουμε και αξιολογούμε τους πράκτορες μας με βάση την αποτελεσματικότητά τους σε μεταξύ τους αναμετρήσεις, αλλά και σε αναμετρήσεις τους ενάντια σε υπαρκτές υλοποιήσεις άλλων αυτόνομων πρακτόρων, συμπεριλαμβανομένης και της ισχυρότερης υπάρχουσας ευρετικής υλοποίησης αυτόνομου πράκτορα.

Τα αποτελέσματα μας είναι ενθαρρυντικά, και υποδηλώνουν ότι ο αλγόριθμος MCTS μπορεί να επωφεληθεί από τις παραλλαγές που υλοποιήσαμε.

Ειδικά ο πράκτορας που χρησιμοποιεί τη μέθοδο VPI, εμφανίζεται να είναι αρκετά ανταγωνιστικός, και η απόδοσή του μπορεί να συγκριθεί με την απόδοση άλλων υπαρκτών αυτόνομων πρακτόρων του παιχνιδιού Άποικοι του Κατάν, παρόλο που οι υπολογιστικοί πόροι που αξιοποιεί ήταν ιδιαίτερα περιορισμένοι σε σχέση με αυτούς που αξιοποιούν οι αντίπαλοι του.

Acknowledgments

I would like to thank **my family** for supporting me throughout my studies and made all of this possible and my friends who supported and consulted me. I would like to especially thank my friend and colleague **Alexandros D. Keros** for creating a python script to process and present the results, for his implementation-related advice and for his never-ending patience and support.

I would also like to especially thank my supervisor **G. Chalkiadakis**, who gave me the opportunity to work with him, and suggested this very interesting thesis.

Without all of them the completion of this thesis in time would be impossible.

Contents

List of Figures	viii
1 Introduction	1
1.1 Motivation	1
1.1.1 Games & AI	1
1.1.2 Multi-Agent Learning (MAL)	2
1.2 Settlers of Catan Game	4
1.2.1 Board & Basic Elements	4
1.2.2 Rules	6
1.2.3 Decisions To Make	11
1.3 Contributions	14
2 Background: Monte Carlo Tree Search	16
2.1 Markov Decision Processes (MDPs)	16
2.2 Partially Observable Markov Decision Processes (POMDPs)	17
2.2.1 Belief-State MDPs	18
2.3 Game Theory	18
2.4 Monte Carlo Methods	19
2.5 Bandit Based Methods	20
2.6 The Basic MCTS Algorithm	22
2.7 Characteristics	25
2.8 Tree Policy Enhancements	26
2.8.1 General	26
2.8.2 Selected Enhancements:	28
2.8.2.1 Upper Confidence Bound for Trees (UCT):	28
2.8.2.2 Bayesian UCT:	31

2.8.2.2.1	Multinomial Estimation Problem & Dirichlet Priors	33
2.8.2.3	Value Of Perfect Information (VPI):	35
3	Related Work	38
3.1	Monte Carlo Tree Search Applications	38
3.1.1	Games Applications	38
3.1.1.1	Go	38
3.1.1.2	Connection Games	39
3.1.1.3	Other Combinatorial Games	39
3.1.1.4	Real-time Games	40
3.1.1.5	Non-deterministic games	40
3.1.2	Non-game applications	40
3.1.3	Monte Carlo Tree Search in Settlers of Catan- “SmartSettlers”	41
3.1.3.1	Rule Changes	41
3.1.3.2	Effect of starting position	41
3.1.3.3	Domain Knowledge in Monte-Carlo simulations	42
3.1.3.4	Domain Knowledge in MCTS	42
3.1.3.5	Results	42
3.2	The JSettlers Framework	44
3.2.1	Interface	44
3.2.2	Agent Implementation	49
3.2.2.1	Determining Options & Resource Estimation of Time	49
3.2.2.2	Making a Plan and Deciding What To Build .	53
3.2.2.3	Negotiation and Trading	54

4	Agent Implementation	55
4.1	Changes in the JSettlers Framework:	56
4.1.1	Code Integration:	56
4.2	Basis of the Implementation & Class Creation	59
4.3	MONTE CARLO TREE SEARCH Implementation:	63
4.3.1	Selection Step	63
4.3.1.1	UCT	64
4.3.1.2	Bayesian UCT	65
4.3.1.3	VPI	66
4.3.2	Expansion Step	67
4.3.3	Simulation	71
4.3.4	Backpropagation	75
4.3.4.1	UCT	75
4.3.4.2	Bayesian UCT	75
4.3.4.3	VPI	76
4.4	Heuristic Strategies	76
4.4.1	Opening Build Strategy	77
4.4.2	Robber Strategy	77
4.4.3	Monopoly Strategy	78
4.4.4	Road Building Strategy	79
4.4.5	Discard Strategy	79
4.5	Negotiation	80
4.5.1	Trading with Ports & Bank	80
4.5.2	Trading with Other Players	81
5	Experimental Results	83
6	Epilogue	95

6.1	Future Work	95
6.2	Conclusions	97
7	References	99
A	Code Changes in the JSettlers Framework	104
B	Class Diagrams	107
B.1	Code Creation	107
B.2	Code Integration	116

List of Figures

1	Board and Resource Production for each terrain type.	4
2	Resources Cards.	5
3	Development Cards.	6
4	Victory Points for each game element.	8
5	Turn Overview	10
6	Monte Carlo Tree Search [1].	24
7	MCTS:1000 Simulated Games against JSettlers.	43
8	MCTS:10000 Simulated Games against JSettlers.	43
9	Game Interface.	45
10	Game Interface: Board Details.	46
11	Game Interface: Game Information and Chat Area	47
12	Game Interface: Building Area.	48
13	Game Interface: Game Statistics.	48
14	Game Interface: Players Area	50
15	Game Interface: Trade.	51
17	Class Diagrams: SOCRobotBrain	58
16	Class Diagram: Important Classes.	58
18	Class Diagram: SOCRobotDM	59
19	Class Diagram: Integration.	60
20	Class Diagram: Main MCTS Classes.	62
21	Dice Outcome Probabilities.	73
22	Results: UCT agent against JSettlers.	84
23	Results: BAYESIAN UCT agent against JSettlers.	86
24	Results: VPI agent against JSettlers.	88
25	Value Of Information Gameplay.	89
26	Results: UCT vs. BAYESIAN UCT.	90

27	Results: UCT vs. VPI.	92
28	Results: BAYESIAN UCT vs. VPI.	94
29	Class Diagram: MCTS Package	107
30	Class Diagram: Distribution Package	108
31	Class Diagram: Heuristic Strategies Package	108
32	Class Diagram: TreeNode	109
33	Class Diagram: UCT	110
34	Class Diagram: Bayesian UCT	110
35	Class Diagram: VPI	111
36	Class Diagram: Expansion	112
37	Class Diagram: Simulation	113
38	Class Diagram: Checker	114
39	Class Diagram: Heuristic Strategies	115
40	Class Diagram: Negotiator	115
41	Class Diagram: Dirichlet	116
42	Class Diagram: Integration	117
43	Class Diagram: Integration Heuristic	118

1 Introduction

In this thesis, we address a multiagent, non-deterministic, partially observable environment of a *Strategic Board Game* and we implement a newly founded real-time algorithm MONTE CARLO TREE SEARCH, with many successful applications in many domains.

Strategic board games as SETTLERS OF CATAN, are of particular interest to AI, because they represent a “link” between two-player perfect information board games and video games. On the one hand they have hidden and random elements, multiple players, which make the implementation of classical techniques difficult, but on the other hand, the states are discrete and their decision-making is turn-based.

1.1 Motivation

We begin by paving the way to the main branch of this dissertation by discussing its two important pillars, namely games (and their use in AI) and multi-agent learning.

1.1.1 Games & AI

The early efforts of Shannon, Turing, Herbert Simon and of others, generated a considerable interest in researching computer performance at games.

Games not only have a cultural entertainment role but they are also used as tools, for both children and adults, for understanding the world and for developing their intelligence. We must not forget that one of the original “grand challenge” applications of AI was to build a chess program of world-championship caliber. At the early days of AI, only few realized the difficulty of creating programs that exhibit human-level intelligence. Programming

agents for games like chess is "easy" due to the nature of the game and the agent access to the entire state of the game.

However, most games are not as trivial. They are abstract environments, something that makes them easier to analyze than real-life environments, but the complexity is high enough requiring logic and intelligence .

So games, as the real world, require the ability to make decisions even if an optimal decision can't be calculated. The complexity and variety of games gave them the role as an important test-bed for AI research and led to very interesting ideas concerning time management [2].

1.1.2 Multi-Agent Learning (MAL)

Techniques that work for two-player perfect information games do not carry over to games of imperfect information or with random elements. Agents must now deal with incomplete knowledge, multiple competing agents, risk management, opponent modeling and deception [3].

So as one would expect, when an agent is situated in an experienced, rich and complex environment, intelligence can emerge [4]. Shoham, Powers and Granager [5] started a discussion regarding the definition, goals and evaluation criteria of multiagent learning. Even though as they suggest, it would be useful to take a step back and identify possible research agendas, classify existing research and face open challenges, they present a broad AI research in limited terms. Peter Stone in *Multiagent learning is not the answer, It is the question*[6] provides a great response from an AI perspective. He argues that even though there is a great MAL research that is within *game theory* field, there are limitations of the tools and language provided by it. Every multiagent encounter can be classified as a normal form or extensive form game. However, the formulation of the encounter as such, does little progress

to solve that encounter and not everything is about convergence to an equilibrium. The author provides examples of his personal work in multiagent systems in which the complexity is so high, that we are not even close in calculating an equilibrium or an optimal solution with the current methods and presents many multiagent problems -as *distributed network routing, collaborative multi-robot localization, in-city driving and more*- that didn't and shouldn't be approached using game-theoretic approaches.

The concluding statement -in my opinion- **defines** the approach that must be taken in multiagent learning:

“Multiagent learning is a good tie between game theory and AI, but from an AI perspective, multiagent learning should be considered more broadly than game theory can address.”

1.2 Settlers of Catan Game

This chapter is dedicated to analyzing the “Settlers of Catan” game, a multi-player strategic board game, first published in 1995 and one of the first board games to become popular even outside Europe. Settlers of Catan is a multiplayer strategy game by Klaus Teuber, where the players take the role of settlers inhabiting an island. The goal is to settle the island and expand your territory to become the largest and most glorious in Catan. Due to its popularity, several expansion packs were created [7] but in this thesis we choose the original *Settlers of Catan* (also called the *Base Game*) with a maximum of four players.

1.2.1 Board & Basic Elements

The game board is composed of 19 terrain hexes surrounded by ocean and which are randomly laid out at the beginning of each game.

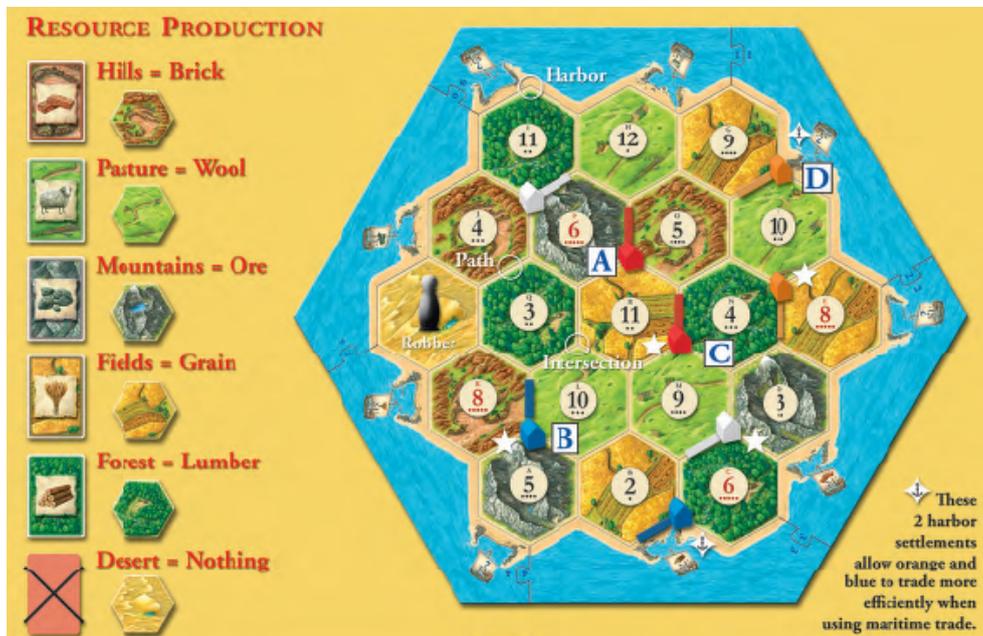


Figure 1: Board and Resource Production for each terrain type.

There are six different types of hexes and each one produces a resource, except for the desert hex:

- *Plains*, which produce *Wheat*,
- *Meadows*, which produce *Wool*,
- *Mountains*, which produce *Ore*,
- *Hills*, which produce *Clay*,
- *Forest*, which produces *Wood* and
- one *Desert*, which produces *nothing*.

Each player gets *pieces* that represent their cities, settlements and roads, and when they build, they place the appropriate piece on the board.

Cards are used to represent the resources of each player, which they use when they build.



Figure 2: Resources Cards.

There is also another set of cards, which are called the *Development Cards*, and which are bought in the same way as other elements (The resources needed to buy a game piece are described below in the Rules section).

They essentially give to players other ways to expand their territory.

There are several types of Development cards:

- The *Road Building* cards, allow a player to build two roads without spending any resources,
- the *Monopoly* cards, allow a player to claim Monopoly on one resource of his choice,
- the *Victory Points* cards, are worth Victory Points, but most cards are
- the *Soldier/Knight* cards, allow a player to move *The Robber*, a piece that:
 - prevents the hex in which is on, to produce resources.
 - the player that moves it, can steal one resource from a player who has a settlement or a city adjacent to the new hex of the robber.

If a player is the first that has the 3 Soldier/Knight Cards face-up, he takes the “*Longest Army*” Special Card, which is worth 2 Victory Points. If some player plays more Soldier cards, he takes the “*Longest Army*” Special Card, along with the 2 Victory Points.



Figure 3: Development Cards.

1.2.2 Rules

The game begins with the initial placement phase. Each player begins with two settlements and two roads, but each player places one settlement and

one road at a time.

The settlements are placed between the intersection of three hexes, they worth 1 Victory Point and they must be placed at least two intersections apart from any other settlement. You can only build a settlement if you have a road leading to an unoccupied intersection that satisfies the above criteria.

Roads are placed on the edge between two hexes, so each road basically connects two corners. At the initial placement phase, each road put must connect to the settlement that it was placed with it. Roads worth 0 Victory Points, but the first player to build a continuous path with at least 5 road pieces, receives the "*Longest Road*" Special Card, which awards 2 Victory Points. If another player, builds a path with more pieces, he takes the Special Card and the 2 Victory Points that come with it.

So at the beginning, after the placement of the two first settlements of each player, they all have 2 Victory Points. To gain more Victory Points, a player must build new roads and settlements, upgrade existing settlements into cities or he must acquire Victory Point cards. Each city is worth 2 Victory Points. To build roads and settlements, to upgrade to Cities and to buy Development Cards, the player must acquire resources.



Figure 4: Victory Points for each game element.

How does he acquire resources?

As we can see from the image of the board (Figure 1 on page 4), during the setup phase of the board, a number is assigned to each hex.

In the beginning of each turn, the current player rolls the dices, and the hexes with the outcome number on them produce resources. The player only collects resources if a settlement or a city is adjacent to a terrain hex that produces the resource.

The resources required to build or upgrade are:

- Roads require (1) Clay Resource and (1) Wood Resource,
- Settlements require (1) Wood Resource,(1) Clay,(1) Wheat Resource and (1) Sheep Resource,
- Cities require (2) Wheat Resources and (3) Ore Resources,
- Development Cards require (1) Ore Resource,(1) Sheep Resource and (1) Wheat Resource.

Because rarely one has settlements adjacent to all the hexes needed for building or upgrading pieces or for buying Development Cards, there is the option to trade with other players! Essentially you can offer some of your resources in exchange for other resources. There is also the option of trading with the bank, where you can give four(4) of one type of resource to get one(1) of one other type that you want. There are some ports on the board, that if a player has a settlement bordering with one, he can trade with another rate (3:1 or 2:1, according to type and characteristics of the port).

If a “7” is rolled, there is no resource production and *The Robber* is activated.

In this case:

- Any player that has seven(7) or more Resource Cards, must discard half of them and return them to the supply stacks. If the player has an odd number of Resource Cards he can round down. (e.g if player has 11 resources, he can discard 5).
- The player, who rolled the “7” **must** move the robber from its current spot to another hex. The selected hex stops production, until the robber is moved from it.
- Also, the player can steal one Resource Card from an owner of an adjacent settlement or city to the destination hex. If there are more than one players with adjacent pieces, he can chose to steal from whoever he wants.

We can see that, there is a difference between activating *The Robber* by rolling “7” and activating *The Robber* by playing a *Soldier/Knight* card. In the latter case, the players aren’t forced to discard resources.

The first player to acquire 10 Victory Points (on his turn) is declared the winner. The complete set of rules is provided in [8].

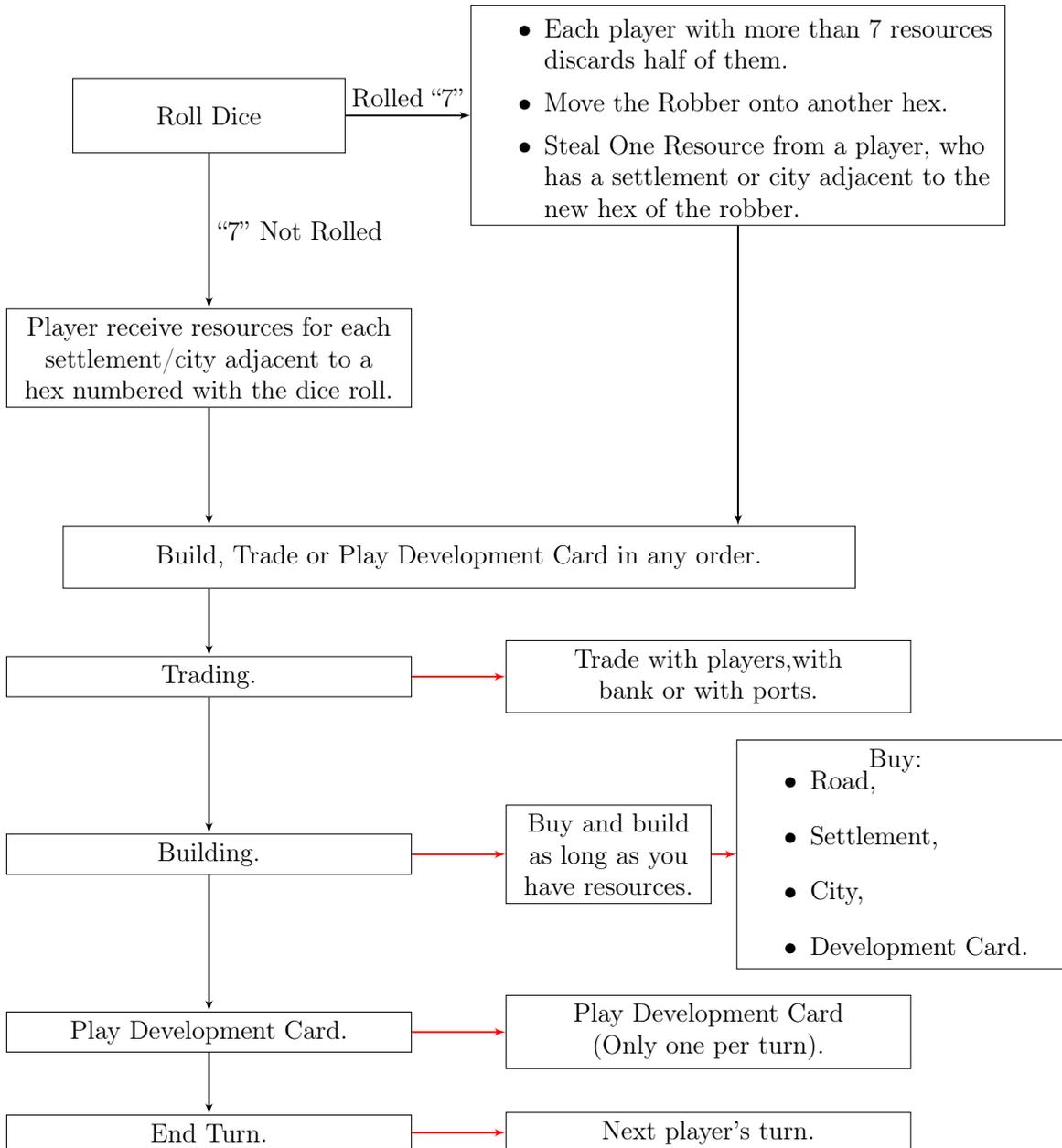


Figure 5: Turn Overview

1.2.3 Decisions To Make

There are many choices for the players to make throughout the game, as to decide where to build next or what kind of strategy will they use to be the first to acquire 10 Victory Points. Due to the nature of the game, there are many factors that contribute to the creation of a player's strategy.

The first factor is for the player to formulate a **long-term plan**. That is, to create a rough plan of how he will emerge victorious victory and be the first to achieve the 10 Victory Points. Then, in each turn, he must consider how a particular piece can help him succeed with his plan. But the environment of the game is unpredictable and even a well designed plan can't be dependable.

The second factor is the formulation of a **short-term plan**. In this plan, the player has to decide about many short-term goals, such as updating a settlement to a city to gain more resources, build a settlement in order to have access to a port or even a strategy to gain the *Largest Army* or the *Largest Road* Special Cards.

Note that it is very difficult to decide between all the possible short term goals due the adversarial environment, the lack of knowledge of the other players goals and resources and the random results of the dice in each round which can lead to very different states than those that a player might expect. There are, however, ways to make more sophisticated guesses about some elements of the game and the next moves of the opponents by keeping track of their resources or even trading with them to see what kind of resources they wish to obtain. Also, there is a lot of "table talk" during the game, that can be used to reveal the true intentions of a player.

The third factor is to decide **what to build next**. As mentioned above, there is a long term plan and many short term plans for the player to satisfy. On a player's turn, as we can see in Figure 5, there are many actions to be considered. A player can choose to build according to his available resources, an action that can contribute to an immediate -but maybe not so important- expansion of his territory, or he can hold out and wait for more resources to build something better. If the player chooses to hold out, there is the risk of losing resources due to the activation of *The Robber* and also the risk that he underestimated the estimated time to acquire the necessary resources to build a piece. Knowing when to wait for resources and when to build something is a very important skill in the Settlers of Catan.

The fourth is to *evaluate the opponent's positions*. This evaluation helps one to understand how close is another player to winning and what will their next move be in order to win. In this way, the player can consider an offer from a different perspective and to avoid giving resources that will help another player to win. Many times it is more important to slow down the leader's plans by interfering with his colonies, than expanding our own colony. Such moves include building a settlement to cut-off his *Largest Road*, buy Development Cards to attempt to "steal" the *Largest Army* card away from him and creating "barriers" to prevent him from expanding his territory to certain areas of the board. Again many elements are hidden, such as the Development Cards and the Resource Cards of the opponents and also the outcome of the dice in future turns is unknown. But we can see their pieces on the board and we can calculate an approximation of their victory points, as well as an estimate of the resources that they will probably receive in latter turns. Many times it is impossible for a player

to slow down the leader due to the distances of their colonies but he can convince another player to cooperate in order to succeed.

The last factor is to **decide what to trade and with whom to trade it with**. Many different combinations of Resources are required in order to build something. A player doesn't always have settlements or cities capable of providing every resource needed for all these combinations. He can make trades with ports or with the bank when it's possible, but a better deal can sometimes be made with another player. Even though a player is competing with the others, each player has a different plan in order to win and of course different needs for Resources, according to his pieces on the board. Trading also requires a strategy. You can either see what resources you have and what resources are missing in order to build a specific piece and trade a combination of the remaining cards with another player or you can have more than one possible options to build and if someone refuses your first offer, try to make a deal for the second one and so on. In both cases, the player must keep in mind not to trade resources that can be of great benefit to the opponent.

1.3 Contributions

Szita et al.[4] (their implementation is briefly discussed in 3.1.3), enhanced the selection step of the main algorithm using *Search Seeding*, where the initialization of statistics in the nodes of the tree involves some heuristic knowledge. Essentially, they seeded the tree with “virtual wins”, but this is a method that requires hand-tuning to set the appropriate number of virtual wins. In comparison, we enhance the selection step of the algorithm using different approaches to the exploration-exploitation dilemma and specifically we use the standard UCT approach, a Bayesian approach to UCT based on Tesauro et al.[9] and the model-based concept of *Value of Information* by Dearden et al.[10].

This is the first time that Bayesian UCT is used on Monte Carlo Tree Search in Settlers of Catan strategy game and the first time that VPI is used on the Monte Carlo Tree Search method in general. The results of these implementation are presented at Chapter 5.

Unlike Szita et al., we didn’t remove the elements of imperfect information and so the agent does not know at a particular state what Development Card will come up if he intends to buy and what cards are stolen. Their agent also does not accept or initiate trades with other players, a decision that -as they admit- creates a handicap on their player.

Even though they argue that these modifications do not alter the gameplay, we do not think that’s the case. So in our implementation the agent considers and initiates trades with other players and imperfect information elements are still present. Apart from the MONTE CARLO TREE SEARCH and the NEGOTIATION needed for the agent, there are various situations in the game, in which the agent must make decisions, such as when a “7” and the player must *Discard* resources and/or must move *The Robber*. For these

situations, some of the existing strategies were used and some new strategies were created to suit the needs of our agent. These strategies are presented in Chapter 4.4.

The rest of this thesis is structured as follows:

Chapter 2 presents the MONTE CARLO TREE SEARCH history, algorithm and characteristics and Chapter 3 presents applications of MCTS and a Java implementation of SETTLERS OF CATAN. In Chapter 4 and in Chapter 5 we describe the implementation of the agent and the results against existing implementations respectively. Chapter 6 summarizes the results of the project and discusses future work, in Appendix A we present some specific changes made in the code of *JSettlers* framework and in Appendix B we present the class diagrams of our implementation and class diagrams for code integration.

2 Background: Monte Carlo Tree Search

Monte Carlo Tree Search is a newly founded method for making optimal decisions in a given domain, by building a tree according to the results of taking random samples in the decision space. In the last years there has been a great interest in the study and development of Monte Carlo Tree Search (MCTS) methods. This interest is due to the many successful applications of the method in many games, and especially in Go[9][1].

The algorithm is based on **two principles**:

“The true value of an action can be approximated using random simulations; and that these values may be used efficiently to adjust the policy towards a best-first strategy” [1].

The evaluation function of Monte Carlo Tree Search does not depend on the “man-made” evaluation function of the traditional search algorithms, but on the observed values of the simulations.

In this section, we cover the theory that led to the development of Monte Carlo Tree Search techniques. Specifically, we will examine Markov Decision Processes (MDPs), Game Theory, Monte Carlo and Bandit-based methods. Then, in Chapter 2.6, we present the algorithm itself, its characteristics in Chapter 2.7 and in Chapter 2.8 we talk about the modifications on the *Tree Policy* of the algorithm.

2.1 Markov Decision Processes (MDPs)

Probability Theory and *Utility Theory* are combined under *Decision Theory*, in order to provide a complete framework for making decisions under uncertainty.

The study of Markov Decision Processes is central in Decision Theory.

A particular Markov Decision Process is defined by a four-tuple (S, A, p_T, p_R) , where:

- S is the states space,
- A is the actions space,
- $p_T(s \xrightarrow{a} t)$ is the transition model that captures the model of reaching state t after we execute a at state s and
- $p_R(s \xrightarrow{a} r)$ is a reward model than captures the probability of receiving reward r after executing a at state s .

A policy is a mapping from states to actions and the aim is to find the policy π that yields the highest expected reward.

2.2 Partially Observable Markov Decision Processes (POMDPs)

When the state isn't fully observable we must choose the Partially Observable MDPs (POMDPs) to model our problem. In this case, Bayesian methods can be used to compute at each time step the probability of the environment's being in each state of the underlying MDP [11]. This is a more complex formulation and we must add the following tuple:

- $O(s, o)$: An observation model that specifies the probability of perceiving observation o in state s .

In all cases the optimal policy π is deterministic.

2.2.1 Belief-State MDPs

A belief state is a probability distribution over states and we can model a POMDP as a Belief-State MDP. Each MDP state of a belief-state MDP, is a probability distribution over the the states of the original POMDP. The transitions in this case are a result of actions and observations and the reward for each state is modeled as the expected reward of the original POMDP.

2.3 Game Theory

Game Theory expands *Decision Theory* in environments, where many agents interact [5]. In these environments the actions of the players are chosen either simultaneously or sequentially and the players may or may not know the preferences of the opponents.

A (stochastic) game in extensive form can be described by the following components[1]:

- S : the set of states, where s_0 is the initial state.
- S_T : the set of terminal states.
- $n \in \mathbb{N}$: The number of players.
- A : the set of actions.
- $f : S \times A \rightarrow S$: the state transition function.
- $R : S \rightarrow \mathbb{R}^k$: the utility function.
- $\rho : S \rightarrow (0, 1, \dots, n)$: Player about to act in each state.

Each game has a root state s_0 and continues until a terminal state is reached. Each player i makes a move that leads to state s_{t+1} , according to the state transition function f . The reward received by each player is defined in the *Utility Function* R , and this reward's values are often defined as 0 for non-terminal states and $+1, 0, -1$ for terminal states if the result is a win, a draw or a loss respectively. These values are called the *game-theoretic* values of a terminal state.

2.4 Monte Carlo Methods

Monte Carlo methods are essentially decision methods employing experiments on random numbers. Their real use as a research tool comes from work on the atomic bomb during World War II [12]. Since then, they have been used in a wide array of domains such as chemistry, biology, medicine but also in game research.

In the context of a game defined as above, the value of each action a from state s , is estimated using simulations and taking the mean of the outcomes, which start with action a .

So, the *Monte-Carlo Q-value* of an action a in state s , $Q(s, a)$ is:

$$Q(s, a) = \frac{1}{N(s, a)} \cdot \sum_{i=1}^{N(s)} \mathbb{I}_i(s, a) z_i$$

where z_i is the outcome of the i th simulation, $\mathbb{I}_i(s, a) z_i$ is an indicator function returning 1 if action a was selected in state s during the i th simulation, and 0 otherwise; and $\sum_{i=1}^{N(s)} \mathbb{I}_i(s, a) z_i$ is the total number of simulations in which a was selected from state s [13].

When actions from a given state s are uniformly sampled, we call this ap-

proach as *flat Monte-Carlo*. It is easy to create cases where *flat Monte-Carlo* fails, due to the fact that it does not allow opponent modeling [14].

Improvements on the reliability of the game theoretic estimates can be made, based on past experience. During action selection it makes sense to favor action selection to actions that have a higher observed reward [1].

2.5 Bandit Based Methods

The *Bandit Problems* is a well known class of problems, where one is faced repeatedly with a choice among n different options in order to maximise the cumulative reward over some time period, by taking the optimal action. The underlying reward distributions for each action is unknown (although you can estimate the value based on past observations), making the choice of an action a difficult task. If you maintain estimates of the values for each action, at each time, there will always be an action that has the highest estimated reward. This action is called the *greedy action*. If you select a greedy action, it is said that you are *exploiting* your knowledge on the values of the actions. If you prefer not to select a greedy action, then you are *exploring* because in this way you can improve the values of non-greedy actions [11]. This leads to the *Exploration-Exploitation dilemma*, where one needs to balance the exploitation of an action, currently believed to be optimal, with the exploration of other actions that may prove to be superior in the long run.

The policy should attempt to minimize the *regret* or *learning loss* after n plays, which is defined as the difference between the maximum expected reward when the probability measure of each arm is known and

the maximum reward obtained by a particular policy [15]:

$$R_N = \mu^* n - \sum_{j=1}^K \mu_j \cdot \mathbb{E}[T_j(n)]$$

Non-zero probabilities must be attached to all arms, in order to not miss the optimal arm, by selecting a sub-optimal arm with temporarily promising rewards. So we must introduce an *Upper Confidence Bound(UCB)* in order to ensure this.

The simplest of the UCB policies, was proposed by Auer et al.[16] and it's called *UCB1*. The policy dictates that:

$$UCB1 = \bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}}$$

where \bar{X}_j is the average reward from arm j , n_j is the number of times arm j was played and n is the overall number of plays so far.

The first term of the equation encourages the *exploitation* of arm j and the second the *exploration* of less visited choices. This policy has an expected algorithmic growth of regret, that is uniform over n without assuming any prior knowledge regarding the reward distributions [1].

2.6 The Basic MCTS Algorithm

The basic algorithm involves iteratively building a search tree until some predefined computational budget is reached, at which point the search is halted and the best performing root action is returned [1].

To find the most “urgent” node of the built tree, a *tree policy* is used in each iteration. This policy tries to balance the exploration-exploitation dilemma. When the node is found, a simulation is run and we update the search tree (add child nodes according to the selected action and update the statistics of the nodes) according to the results. Moves during the simulation are usually selected based on a *default policy*.

It’s important to note that due to the fact that the values of intermediate states are not needed, we can greatly reduce the domain knowledge required. That is, MCTS does not require the use of any utility-estimating heuristics. We only need the value of the terminal state (or of a non-terminal state, if the computational budget available is reached).

The general Monte Carlo Tree Search approach is presented below:

Algorithm 1 General MCTS approach.

```
function MCTSSEARCH( $s_0$ )
  create root node  $v_0$  with state  $s_0$ 
  while within computational budget do
     $v_l \leftarrow$  TREEPOLICY( $v_0$ )
     $\Delta \leftarrow$  DEFAULTPOLICY( $s(v_l)$ )
    BACKUP( $v_l, \Delta$ )
  return  $a(\text{BESTCHILD}(v_0))$ 
```

Until the computational budget is reached, the search iterates through (executes repeatedly and sequentially) the following steps[17]:

1. **Selection:**

In this step, we start at the root node and we apply a *tree policy* to descend the tree until we reach the *most urgent expandable node*. A node is *expandable*, if it represents a *non-terminal state* and has *children* that are still *unvisited*. The *most urgent* node is determined by the *tree policy*, which determines the *Best Child*, a node that maximises the specific equations of the policy. This step is used with the expansion step in order to create the game tree and it is used by itself in order to determine the best action to perform after the computational budget is reached. Many enhancements are proposed in [1], some of which we discuss later in the thesis.

2. **Expansion:**

According to the available actions, we create new child nodes (one or more) and we add them to the tree.

In the literature, there are no enhancements for this step. The *expansion policy* that is used for a given problem is mainly an implementation choice (“typically between single node expansion and full node expansion”), and depends on the particular domain and the specified constraints[1].

3. **Simulation:**

A simulation is run from the new node(s) according to the *default policy* to produce an outcome. The default simulation policy for MCTS is to select randomly amongst the available actions. This has the advantage that it is simple, requires no domain knowledge, and repeated trials will most likely cover different areas of the search space; but the games played are not likely to be realistic, compared to games played

by rational players.

4. Backpropagation:

Based on the simulation results, we then *backup*, the value of the terminal node to the *visited nodes*, in order to update their statistics.

These steps presented above are summarized in Figure 6.

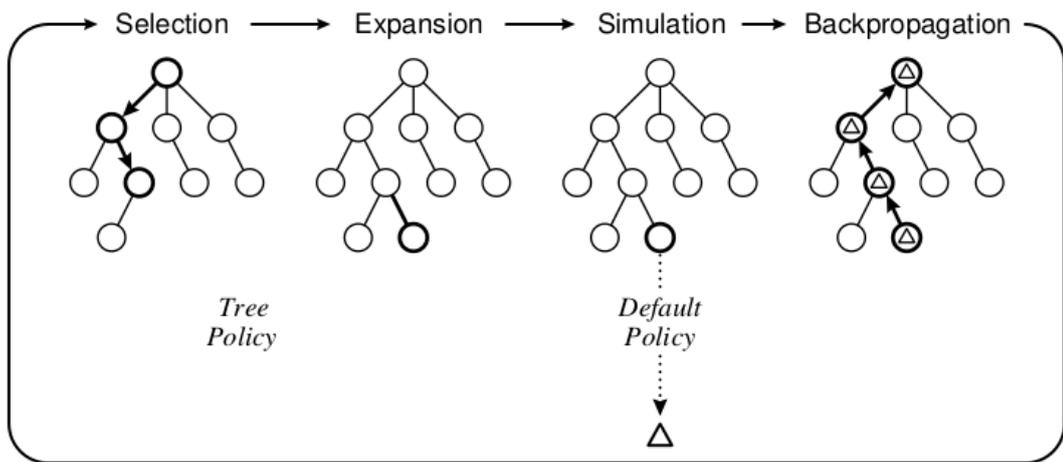


Figure 6: Monte Carlo Tree Search [1].

We can group these steps into two policies:

- *Tree Policy:* The selection and expansion step together. These two steps are used to iteratively build the game tree. We select or create new leaf nodes from the existing tree nodes.
- *Default Policy:* Simulate from a given non-terminal state to a terminal state in order to produce value estimates.

As far as the backpropagation step is concerned, it does not use a policy itself, but its role is to update the node statistics for the future tree policy decisions.

2.7 Characteristics

Monte Carlo Tree Search Algorithm is a very popular choice in a variety of domains, due to its characteristics[1]:

1. **Aheuristic:**

Monte Carlo Tree Search does not need domain-knowledge, which essentially makes it applicable to a variety of domains (if they can be modeled as a tree). Nonetheless, we can see significant improvement in performance if we add domain-specific knowledge in order to bias selection. All top-level MCTS Go programs use game-specific information. There are certain trade-offs to consider for one to use biased move selection using domain specific knowledge. Although, it may reduce the variance in the results, it also decreases the number of simulations possible. On the other hand, one great advantage of uniform random move selection is its speed, which can allow many simulations.

2. **Anytime:**

The result of each simulation is immediately backpropagated to the upper nodes and all statistics are up-to-date in every iteration of the algorithm. So in any moment in time, an action from the root can be returned, but if we allow more iterations the performance will improve.

3. **Asymmetric:**

The tree selection allows favoring more promising nodes, leading to an asymmetric tree over time. So, the building of the tree is skewed towards more promising areas, and the tree shape that emerges can even be used to gain a better understanding of the game.

2.8 Tree Policy Enhancements

There are many modifications one can make to the basic MCTS algorithm in order to improve its performance. Here we describe the main ones found in [1].

2.8.1 General

These modifications can basically be divided into two categories:

- *Domain Independent:*

In this category, no domain specific knowledge is used, so all modifications belonging in this category can be applied to any domain without prior domain-knowledge. These modifications, offer small improvements or are better suited to a particular type of domain.

- *Domain Dependent:*

As the name suggests, these modifications are specific to particular domains. They might be used to exploit some unique aspect or use prior knowledge of a domain.

There are many enhancements proposed such as the *UCB1-Tuned*, which tunes the bounds of UCB1 (which we described above), the *Bayesian UCT*, which introduces a Bayesian framework to estimate node values with a limited number of simulations (and which we selected as one of the enhancements used in this thesis), the *Search Seeding*, which “warms up” the search tree by initializing the statistics of each node according to some heuristic knowledge, the *Progressive Bias*, which describes a technique for adding domain specific heuristic knowledge to MCTS and many more (see [1] for an extensive list of the enhancements).

It is important to note that MCTS works well in some domains but not in others. There are some conditions that may cause problems when adding enhancements.

1. **Consistency:**

As the computational needs of the MCTS algorithm increase with its modification and with the application of various enhancements, it is possible to observe unwanted and faulty behavior, such as wrong deduction about the agent's current position.

2. **Parameterisation of Game Trees:**

Long et al.[18] define three basic properties that directly influence the performance of Perfect Information Monte Carlo search in Game Trees that could be used to predict the success of MCTS to new games.

These properties are:

- the *Leaf Correlation*, which gives the probability that all siblings, terminal nodes have the same payoff value.
- the *Bias*, which determines the probability that the game will favor a particular player over the other, and
- the *Disambiguation Factor*, which determines how quickly the number of nodes in a player's set shrinks with regard to the depth of the tree.

3. **Comparing Enhancements:**

Another issue to be addressed is the measurement of the performance of different enhancements. As we mentioned, different enhancements may increase computational cost and reduce the number of performed

simulations. Some metrics for comparing approaches must be used, like the win rate against specific opponents, the number of iterations per second, or even the amount of memory used by the algorithm.

2.8.2 Selected Enhancements:

As we saw, there are many proposed enhancements for the tree policy of the Monte Carlo Tree Search algorithm. We chose 3 such MCTS variants, in order to test their performance in *Settlers of Catan* game domain.

The rest of this section is dedicated to analyzing the proposed methods and their characteristics. We begin with the most popular algorithm in the MCTS family, the *Upper Confidence Bound for Trees* algorithm and we continue with *Bayesian UCT* and *Value of Perfect Information* algorithms.

2.8.2.1 Upper Confidence Bound for Trees (UCT):

The UCT algorithm is the most popular algorithm in the MCTS family. The success of Monte Carlo Tree Search, especially in Go, is mainly a result of this policy. Koscis and Szepešvári [19][20], suggested the use of UCB1 (see 2.5) as a tree policy. If we treat the choice of a child as a multi-armed bandit problem, it can be said that the value of a child node is the expected reward approximated by the MC simulations and so these rewards correspond to random variables with unknown distributions.

UCB1 has some promising properties: it is very simple and efficient and guaranteed to be within a constant factor of the best possible bound on the growth of regret. It is thus, a promising candidate to address the exploration-exploitation dilemma in MCTS: every time a node (action) is to be selected within the existing tree, the choice may be modeled as an

independent *multi-armed* bandit problem ¹.

A child node j is selected to maximise:

$$UCT = \bar{X}_j + 2C_p \sqrt{\frac{2 \ln n}{n_j}} \quad (1)$$

, where \bar{X}_j is the average reward of the node j , n the number of times the current (parent) node has been visited, n_j the number of times child j has been visited and $C_p > 0$ is a constant.

There is a balance in the exploration-exploitation dilemma, and this balance lies on the first and second terms of *UCB* equation respectively. The more we visit one node, the more the exploration term is decreased, reducing its contribution to the equation. On the other hand, if we visit another child of the parent node, the exploration term increases for the remaining unvisited children. This exploration term, ensures that the nodes have a non-zero probability of selection. The constant C_p in the exploration term can be adjusted to lower or increase the amount of exploration. The rest of the algorithm proceeds as described in Algorithm 1 and Algorithm 2 shows the UCT algorithm in pseudocode.

Kocsis and Szepesvári [19][20], show that the bound of regret of UCB1 still holds, when we have non-stationary reward distributions. They also showed that the failure probability at the root of the tree converges to zero at a polynomial rate as the number of simulated games grows to infinity. This directly implies that given enough time and memory, UCT allows MCTS to converge to the minimax tree ,and thus to optimality[1].

¹In the *n-armed bandit problem*, you repeatedly have to choose from n actions. After each selection, one receives a reward from a “stationary probability distribution” that differs according to the action selected. The goal is to maximise the expected total reward over each *play*(action selection)[11].

Algorithm 2 The UCT Algorithm.

```
function UCTSEARCH( $s_0$ )
  create root node  $v_0$  with state  $s_0$ 
  while within computational budget do
     $v_l \leftarrow$  TREEPOLICY( $v_0$ )
     $\Delta \leftarrow$  DEFAULTPOLICY( $s(v_l)$ )
    BACKUP( $v_l, \Delta$ )
  return  $a(\text{BESTCHILD}(v_0))$ 

function TREEPOLICY( $v$ )
  while  $v$  is non-terminal do
    if  $v$  not fully expanded then
      return EXPAND( $v$ )
    else
       $v \leftarrow$  BESTCHILD( $v, C_p$ )
  return  $v$ 

function EXPAND( $v$ )
  choose  $a \in$  untried actions from  $A(s(v))$ 
  add new child  $v'$  to  $v$ 
    with  $s(v') = f(s(v), a)$ 
    and  $a(v') = a$ 
  return  $v'$ 

function BESTCHILD( $v, c$ )
  return  $\arg \max_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v' )}}$ 

function DEFAULTPOLICY( $s$ )
  while  $s$  is non-terminal do
    choose  $a \in A(s)$  uniformly at random
     $s \leftarrow f(s, a)$ 
  return reward for state  $s$ 

function BACKUP( $v, \Delta$ )
  while  $v$  is not null do
     $N(v) \leftarrow N(v) + 1$ 
     $Q(v) \leftarrow Q(v) + \Delta(v, p)$ 
     $v \leftarrow$  parent of  $v$ 
```

2.8.2.2 Bayesian UCT:

The bandit-based method used for node selection in the tree policy is central to the MCTS method being used. A wealth of different upper confidence bounds have been proposed, often improving bounds or performance in particular circumstances such as dynamic environments.

Tesauro et al.[9] propose that the Bayesian framework potentially allows much more accurate (Bayes-optimal) estimation of node values and node uncertainties from limited numbers of simulation trials. The basic premise of their work is that, in practical applications of MCTS to MDPs or games, the developers will know the characteristics of the reward distribution, and algorithms that can make use of such a reward model, could outperform the distribution-free methods. In this approach, stochastic trial results at leaf nodes are combined with prior information to yield posterior distributions. If the leaf nodes priors and inference models are correct, we can enable Bayes-optimal estimation of interior node values. Even though the Bayesian Inference requires more computational budget, it can allow more robust convergence under a wide range of sampling policies.

They propose two modified versions of UCB1² (the sampling formula in UCT) to descend the tree and choose where to sample next.

In some detail, an upper confidence bound B_i for each arm i is computed and then the arm with the maximum bound is selected:

$$\text{Maximise } B_i = ri' + \sqrt{\frac{2\ln N}{n_i}} \quad (2)$$

n_i : Number of trials for each arm

r_i : Average rewards obtained in these trials, rewards are scaled to $[0,1]$

²Note that UCB1 solves the exploration-exploitation dilemma in the multi-armed bandit problem

$N = \sum n_i$: Total trials of all arms

The first proposed equation, replaces the average reward of child node i by μ_i , the mean of P_i :

$$\text{Bayes-UCT1: maximise } B_i = \mu_i + \sqrt{\frac{2\ln N}{n_i}} \quad (3)$$

The second equation is motivated by the central limit theorem and also replaces the $\frac{1}{\sqrt{n_i}}$ factor in the exploration term by σ_i , the standard deviation of P_i :

$$\text{Maximise } B_i = \mu_i + \sqrt{\frac{2\ln N}{n_i}} \sigma_i \quad (4)$$

μ_i : Mean of an extremum (minimax) distribution P_i

σ_i : Square root of variance of P_i

n_i : Number of visits to node i

P_i : Each node i in the tree maintains a probability distribution over its true expected reward value.

When trials at leaf nodes are performed, the results are combined with priors in the standard way to compute posterior distributions.

The first equation is a strict improvement of UCT if the independence assumption and leaf nodes priors are correct. The results indicate that the second equation outperforms the first and both outperform the standard

UCT approach. So in our implementation we chose to use the second equation.

2.8.2.2.1 Multinomial Estimation Problem & Dirichlet Priors

Now, let X be a random variable that can take K possible values. Given training set D , which contains outcomes of N independent draws x^1, \dots, x^N of X from an unknown multinomial distribution P^* . Finding a good approximation for P^* constitutes the *multinomial estimation* problem. This problem can also be stated as predicting the outcome x^{N+1} given x^1, \dots, x^N .

The Bayesian estimate, given a prior distribution over the possible multinomial distributions is:

$$P(x^{N+1} | x^1, \dots, x^N, \xi) = \int P(x^{N+1} | \theta, \xi) P(\theta | x^1, \dots, x^N, \xi) d\theta \quad (5)$$

where

$$P(\theta | x^1, \dots, x^N, \xi) \propto P(\theta | \xi) \prod_i \theta_i^{N_i} \quad (6)$$

and $\theta = \langle \theta_1, \dots, \theta_K \rangle$ are the possible values over the probabilities $P^*(1), \dots, P^*(K)$ and ξ is a variable containing assumptions over the domain. We chose the Dirichlet distribution as a prior distribution for each node. Dirichlet distribution are a parametric family that is a conjugate prior to the multinomial/categorical distribution[10].

A Dirichlet prior consists of two parameters:

1. $K \geq 2$, the number of rival events and
2. $\alpha_1, \alpha_2, \dots, \alpha_K$, the concentration parameters, where $\alpha_i > 0$

The Dirichlet distribution is a generalization of Beta Distribution and is a distribution over Multinomials. It has a probability density function:

$$p(P = \{p_i\} | \alpha_i) = \frac{\prod_i (\Gamma(\alpha_i))}{\Gamma(\sum_i \alpha_i)} \prod_i p_i^{\alpha_i - 1} \quad (7)$$

The initial prediction for each value of the random variable X, given a Dirichlet prior, is [10]:

$$P(X = i | \xi) = \int \theta_i P(\theta | \xi) d\theta = \frac{\alpha_i}{\sum_j \alpha_j} \quad (8)$$

If the prior is a Dirichlet prior with concentration parameters $\alpha_1, \dots, \alpha_K$ and N_i is the number of occurrences of the symbol i in the training data, then the posterior is also a Dirichlet with concentration parameters $\alpha_1 + N_1, \dots, \alpha_K + N_K$ and thus the prediction for X^{N+1} is:

$$P(X^{N+1} = i | x^1, \dots, x^N, \xi) = \frac{\alpha_i + N_i}{\sum_j (\alpha_j + N_j)} \quad (9)$$

2.8.2.3 Value Of Perfect Information (VPI):

As mentioned earlier, a central problem in complex environments is to strike a balance between the *exploration* and the *exploitation* of actions. We can estimate the benefit of exploration by using the notion of *Value Of Information*, which is the expected improvement that might arise from information acquired by exploration. The estimation of this quantity requires the assessment of the agent’s uncertainty about value estimates for states.

The aim is to find a policy that maximises the expected reward of an agent. Dearden et al. in [21], examine a model free Bayesian RL and their approach builds on the notion of Q-Value Distributions, and in [10] they present a Bayesian approach to model-based reinforcement learning.

They present two new approaches to exploration:

- **Q-Value Sampling:**

This approach is based on Wyatt[22], who proposed a method for solving bandit problems. The agent’s knowledge of the available rewards is explicitly represented as probability distributions. An action is then stochastically selected based on the current probability of its optimality. This probability depends not only on the current expected reward, but also on the current level of uncertainty about the actual reward. In their work they extend this idea to multi-state RL problems. They present a Bayesian method for “representing, updating and propagating probability distributions over rewards”.

- **Myopic-VPI:**

Myopic Value of Perfect Information provides a direct way of evaluating the exploration-exploitation trade-off, by approximating the utility

of an information-gathering action in terms of the expected improvement in the decision quality that result from the acquired information.

Their results show that the state space is explored more effectively (than conventional model-free learning algorithms) and that their performance advantage appears in bigger problems.

In this thesis, we chose *Myopic-VPI* action selection, due to the fact that in [21],[23],[24] and [25], it was uniformly the best approach on many domains. Below we describe the approach in more detail.

The method considers the improvement in the agent's policy through exploration of actions. The idea is to balance the expected gains from exploration -in the form of improved policies- against the expected cost of doing a potentially suboptimal action and what can be gained by learning the true value $q_{(s,a)}^*$ of $q_{(s,a)}$. The only interesting scenarios is when the new knowledge changes the agent's policy. This can happen in two cases:

1. when an action that was -until now- considered sub-optimal, is revealed as the best choice and
2. when an action that was -until now- considered best, is actually inferior to other actions.

For the first case, suppose a_1 is the best action; then $E[q_{(s,a_1)}] \geq E[q_{(s,a')}]$ for all other actions a' . If the new knowledge indicates that a is a better action, then $q_{(s,a)}^* > E[q_{(s,a_1)}]$. So, we expect gain of $q_{(s,a)}^* - E[q_{(s,a_1)}]$.

For the second case, suppose a_1 is the action with the highest expected value and a_2 the action with the second highest. If we have an indication from the new knowledge that $q_{(s,a_1)} < E[q_{(s,a_1)}]$, the agent should perform a_2 with expected gain $q_{(s,a)}^* - E[q_{(s,a_1)}]$.

So, we define the gain from learning the value $q_{(s,a)}^*$ of $q_{(s,a)}$:

$$Gain_{s,a}(q_{s,a}^*) = \begin{cases} E[q_{s,a_2}] - q_{s,a}^*, & \text{if } a = a_1 \text{ and } q_{s,a}^* < E[q_{s,a_2}] \\ q_{s,a}^* - E[q_{s,a_1}], & \text{if } a \neq a_1 \text{ and } q_{s,a}^* > E[q_{s,a_1}] \\ 0, & \text{otherwise} \end{cases} \quad (10)$$

where $Q_{(s,a)}^*$ is the expected reward if we execute action a at state s and we continue with optimal selection of actions. We need to compute the expected gain given our prior beliefs, because the agent does not know in advance the values that will be revealed:

$$VPI_{(s,a)} = \int_{-\infty}^{\infty} Gain_{s,a}(x) Pr(q_{s,a} = x) dx \quad (11)$$

The value of perfect information gives an upper bound on the myopic value of information for exploring action a . The expected cost incurred for this exploration is given by the difference between the value of a and the value of the current best action. This suggests we choose the action that maximises:

$$VPI_{(s,a)} - (\max_{a'} E[q_{s,a'}] - E[q_{s,a}]) \quad (12)$$

We assume parameter independence to represent our distribution over q -values and for each prior, we have a Dirichlet prior. So the posterior at each state can be easily updated, since a Dirichlet prior with hyperparameters $\alpha_1, \dots, \alpha_n$, that intuitively correspond to counts of specific reward occurrences.

3 Related Work

MONTE CARLO TREE SEARCH has many successful applications in many -game and non-game- domains and some of these applications are presented in this chapter along with related work concerning the “Settlers of Catan” game, for which many computer implementations exist. We focus our analysis in the existing MCTS application in “Settlers of Catan” and in the *JSettlers* framework, which is one of the two most powerful implementations concerning “Settlers of Catan”.

3.1 Monte Carlo Tree Search Applications

Most early AI game research had mainly focused on Chess, which was used as a tool for testing new algorithms. After the success of DEEPBLUE, that focus shifted away on to the game of *Go*. Due to the fact that GO is a domain in which computers don’t reach the level of top human players, it has become a new benchmark for AI in games. So, it’s natural that many applications of MCTS are in Go; However, these methods have many other potential uses as well. We now summarize the main applications of MCTS methods found in the literature.

3.1.1 Games Applications

We begin with the applications of MCTS in various games and the results of these implementations.

3.1.1.1 Go

Go is a traditional board game for two players. The players alternately place black and white pieces on the vacant intersections of a 19x19 board. The

game is terminated when both players *pass*, and the player who controls the most territory wins. Strong AI methods for go are not the best choice; programs using α - β search reached a strong beginner level in 1997 and stayed to that point until the first implementations of MCTS methods. Nowadays, all Go programs use MCTS, such as MOGO[26], CRAZY STONE[27], LEELA and FUEGO[28].

3.1.1.2 Connection Games

Connections games, are games in which players try to complete a specified connection between pieces by connecting two or more goal regions, forming loops etc. All the strongest known agents in this area use MCTS. Such games are Hex[29], Havannah[30], Lines of Actions[31] and more.

3.1.1.3 Other Combinatorial Games

Zero-sum games of perfect information, deterministic, with discrete-finite set of moves and usually two players, are called combinatorial games. OTHELLO is such an example and like Go is a game with delayed rewards. The board is quite dynamic and even in the last few moves, it's not clear who the winner is. These characteristics, make Othello a good benchmark for MCTS methods but it is important to note that even before MCTS programs were stronger than the experts human players. Othello has many implementations, but there is room for many improvements, making Othello an open challenge for MCTS methods. Other games in this category are SHOGI[32], AMAZONS[20], BLOKUS DUO[33] and many more.

3.1.1.4 Real-time Games

Monte Carlo Tree Search methods were implemented in many real-time games, from TRON to STARCRAFT. The challenge is to achieve the same level of intelligence and realistic behavior that already exists by the methods of scripting and triggering.

3.1.1.5 Non-deterministic games

These games usually have hidden information or random elements. Randomness may arise through a dice roll or a shuffling of the deck and hidden information through cards or other elements not visible to the player. Hidden information and randomness, make the game trees harder to search, increasing the branching factor and depth.

A common approach is to sample through the perfect information game instances (that arise when we are assuming that the hidden and random outcomes are known) in order to deal with the increasing branching factor[1]. In this category we have card games as MAGIC: THE GATHERING, POKER, board games as BACKGAMMON, PHANTOM CHESS and the game in which we implement MCTS methods in this thesis, the SETTLERS OF CATAN.

3.1.2 Non-game applications

Apart from games applications, MCTS methods have been applied to other domains such as *Combinatorial Optimisation, Constraint Satisfaction, Sample-based Planning and Scheduling tasks*[1]. In the *Combinatorial Optimisation* domain, there are applications in *Security, Traveling Salesman Problem, function approximation and more*. In the *Constraint Satisfaction* domain, in *Constraint Problems* and in *Mathematical Expressions*. In the *Scheduling Problems* Monte Carlo tree based techniques are used in *Bench-*

marks, *Printer Scheduling* and in *Sample-Based Planning in Feature Selection and Large State Spaces*. For a complete guide and analysis of the applications of MCTS methods in these domains, see [1].

3.1.3 Monte Carlo Tree Search in Settlers of Catan- “SmartSettlers”

Szita et al.[4], research the possible and effective use of MCTS methods to implement an agent for games like SETTLERS OF CATAN. They implemented a *standalone* Java software module based on *JSettlers*, created by Robert S. Thomas (see 3.2), designed for fast gameplay, move generation and evaluation.

3.1.3.1 Rule Changes

For an easier implementation of the game, elements of imperfect information were removed. They also chose to not let their game-playing agent initiate or accept trades from other players (but the agent may trade with the bank/-ports). This creates a handicap for their agent and in our implementation we chose to provide the agent with the ability to trade with other players.

3.1.3.2 Effect of starting position

Szita et al.[4] investigated the effects of the starting position of the agent. Their tests showed that an effect to the game outcome from the seating order exists and it’s statistically significant, but may differ on different strategies. In order to eliminate these effects the seating order was randomized.

3.1.3.3 Domain Knowledge in Monte-Carlo simulations

They suggest that if all legal actions are selected with equal probability, the resulting strategy is weak and the quality of the simulation is low. On the other hand, if the action selection is very deterministic, exploration is limited, and again simulation suffers. They tried to balance the exploration-exploitation dilemma in the simulation with the introduction of heuristic knowledge, so the probability to choose an action is proportional to its defined weight. However, with the use of modified probabilities instead of uniform sampling, the performance of the agent dropped significantly.

3.1.3.4 Domain Knowledge in MCTS

They injected MCTS with only limited amount of domain knowledge, by using the *Search Seeding* enhancement, essentially giving “virtual wins” to preferred actions. This means that when a settlement-building action is added to the tree, its counter for number of visits and number of wins is initialized to specific a number (they chose 20 for settlement-building and 10 for city-building). It is also important that these values are not backpropagated through the tree, because that would cause a distortion in the selection step. With those addition the agent playing strength increased considerably.

3.1.3.5 Results

The *SmartSettlers* agent was tested against JSettlers and against humans. In the first experiment consisting of 100 games against JSettlers agents, they concluded that MCTS with 1000 simulated games is roughly as strong as the JSettlers agent, winning 27% of games; and with 10000 simulated games, winning 47% of the time, and reaching good scores even when it loses.

Against human players, Szita “played a few dozen games against a combina-

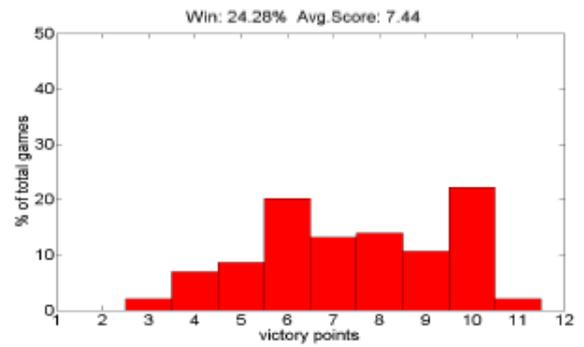
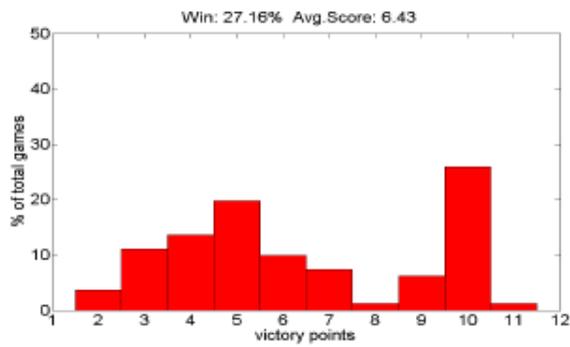


Figure 7: MCTS:1000 Simulated Games against JSettlers.

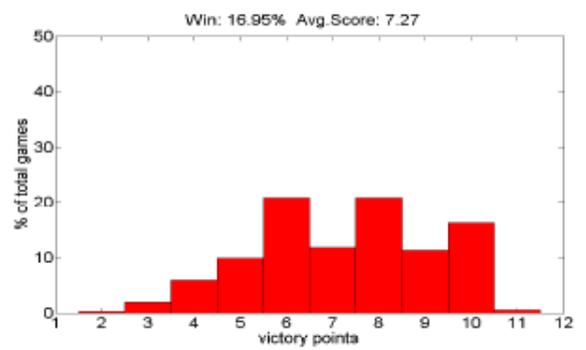
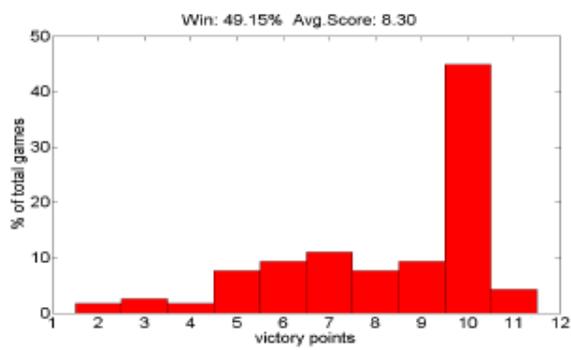


Figure 8: MCTS:10000 Simulated Games against JSettlers.

tion of two JSettlers and one SmartSettlers agent” and the results indicated that an expert human player can confidently beat the SmartSettlers agent.

3.2 The JSettlers Framework

There are several computer implementations of SETTLERS OF CATAN (about ten), which mainly include hand-designed, rule-based AI. Even though, the strength of each implementation varies, all can easily be defeated by an expert human player. The two strongest ones [4] are:

- the Castle Hill's Studios' version, which features strong AI players that use extensive trading and
- Robert S. Thomas' JSettlers, an open-source Java version of the game, which also has **heuristic-based** AI players and is a basis of many SETTLERS OF CATAN servers online.

Even though the original JSettlers framework was implemented in 2003, it is updated and maintained up to this day, with new functions and many corrections to the original implementation.

Pfeiffer[34] and Szita et al.[4] use JSettlers environment to implement a learning agent. The first uses hand-coded high level heuristics with low-level model trees constructed by RL and the latter are using MCTS, described in 3.1.3. Our implementation also uses the JSettlers environment so it is important to analyze some of its main elements.

We begin with a brief presentation of the interface and then we proceed in the description of the agent implementation.

3.2.1 Interface

After a user joins the game, a separate window is displayed, the Game Interface as seen -after a few turns- on Figure 9. We can see that the window is divided to many different regions.

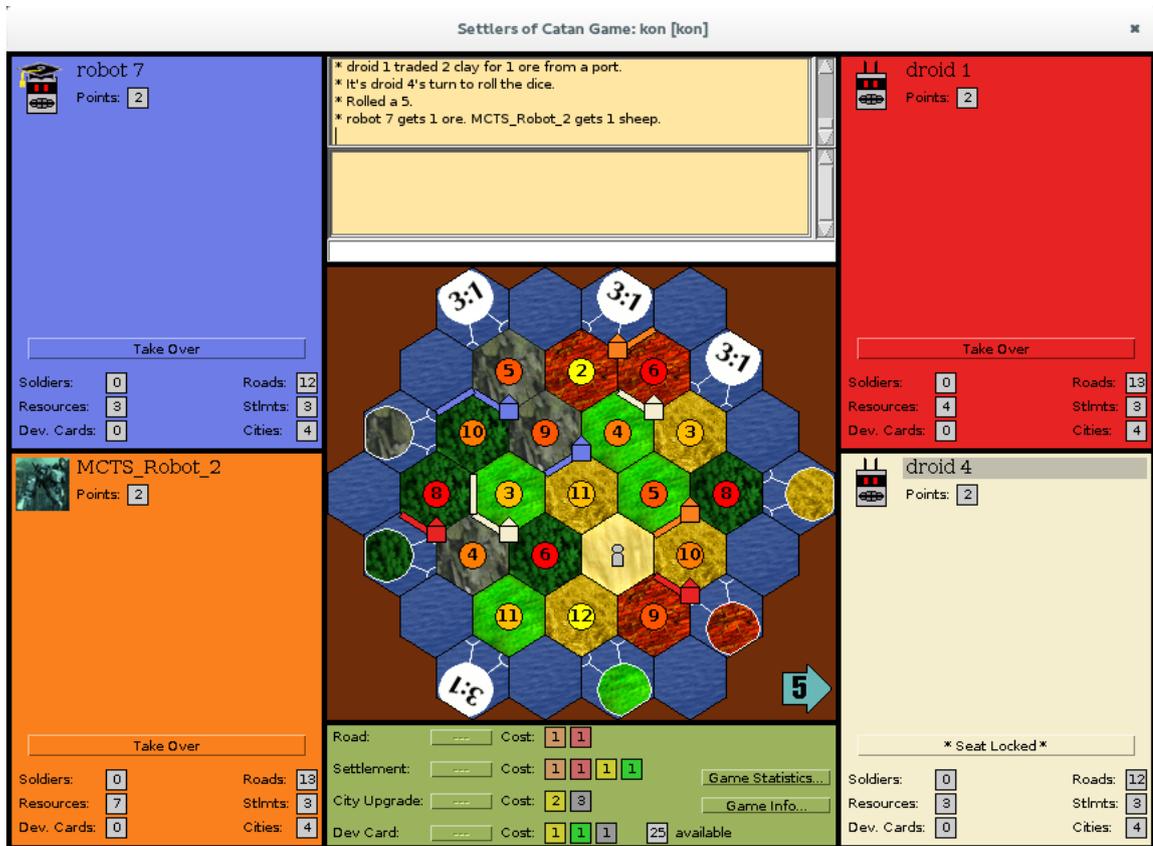


Figure 9: Game Interface.

In the center of the window lies the game board. Before the game begins, all hexes are water hexes. After the game starts, the board is created as described in the rules, the land hexes are shuffled, they are placed inside a hexagon surrounded by ports and then numbers are placed on them, depicting that a hex will produce resources if the dice rolled is the number on it and there is a player's piece adjacent to it.

In Figure 10 the different pieces of the game are highlighted in different colors:

-  Board Hexes,
-  Player's pieces: Cities, Settlements and Roads,
-  Resource Specific Port,
-  General Port,
-  The Robber,
-  Dice Rolled in this turn.



Figure 10: Game Interface: Board Details.

Above the game board, there is a region containing the chat and information area. These are presented in Figure 11 and each area is highlighted as:

 Information Area, which is in the upper section and displays messages from the server about the current game. These messages include, the outcome of dice roll, the resources gained by each player etc.

 Chat Area in the lower section, which displays messages from other people in the game area. The user can send a message by typing in the text field below the display area and pressing *Enter*.

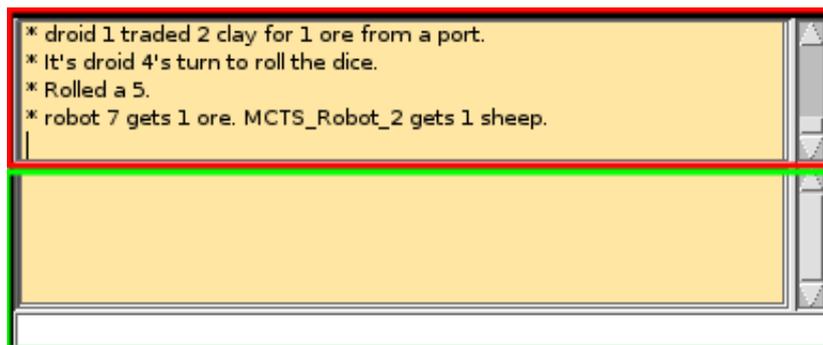


Figure 11: Game Interface: Game Information and Chat Area .

Below the game board, there is a region, which contains information about building actions. It shows what resources are needed in order to build a piece or to buy a development card. It is also the interface to perform these actions. The colored boxes with the numbers in them, represent the set of resources needed in order to build something. The color of the box represents the type of resource and the number in the box represents the quantity needed. The boxes with the dashes indicate that the player does not have the necessary resources to build a piece. When the resources are gathered, the dashes are replaced with a *Buy* indication.

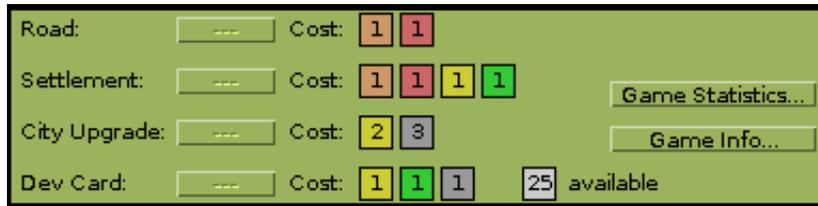


Figure 12: Game Interface: Building Area.

There is also a small box for the number of available development cards. The *Game Statistics* and *Game Info* buttons are used to show the statistics of the game (see Figure 13) and the game options respectively.

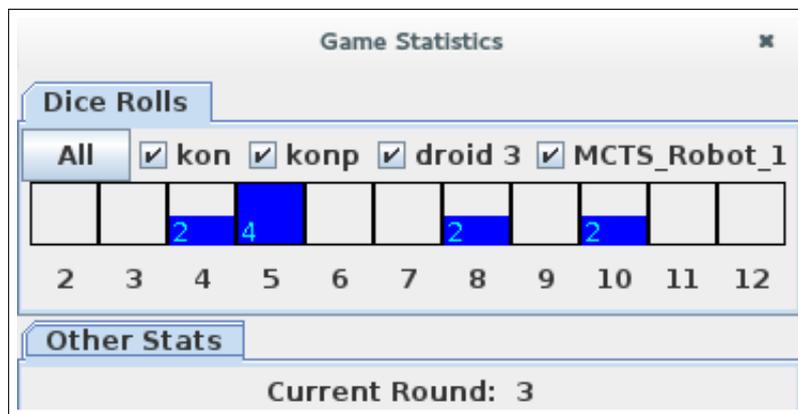


Figure 13: Game Interface: Game Statistics.

There are two kinds of players' areas.

- If one is just observing the game, the information presented is limited to the publicly available information as the player's army size, the total number of his resources, the number of development cards, the number of pieces available for placing and the public Victory Points (the total Victory Points minus the Victory Points obtained by cards). Figure 14a is such an area.
- If one is playing instead, the player's area presents more information and more options, like in figure 14b. We can see that this information

includes a detailed description of the available resources, using the same coloring as in the building region, and numbers to indicate the quantity. The development cards are shown as a list next to the resources and a player can select one from the list and click the *Play Card* button to play the card. Above this section, lies the trading area, where a player can define the resource sets that he is willing to give in exchange for a resource set, he wants to obtain. There are three buttons for *Clearing* the sets, *Trading with ports or banks* and for Offering resources to other players. When a player initiates a trade a “balloon” appears to the offered players (see Figure 15) with the options of *Accepting or Rejecting the trade* and *Making a counter offer*. Next to that section are the player’s available pieces and in the lowest section there are three buttons to *Roll the dice*, *Quit the game* and *Restart the game*. Lastly, in the upper section, we find the player’s icon, Victory Points and awards (such as *Largest Army* and *Largest Road*).

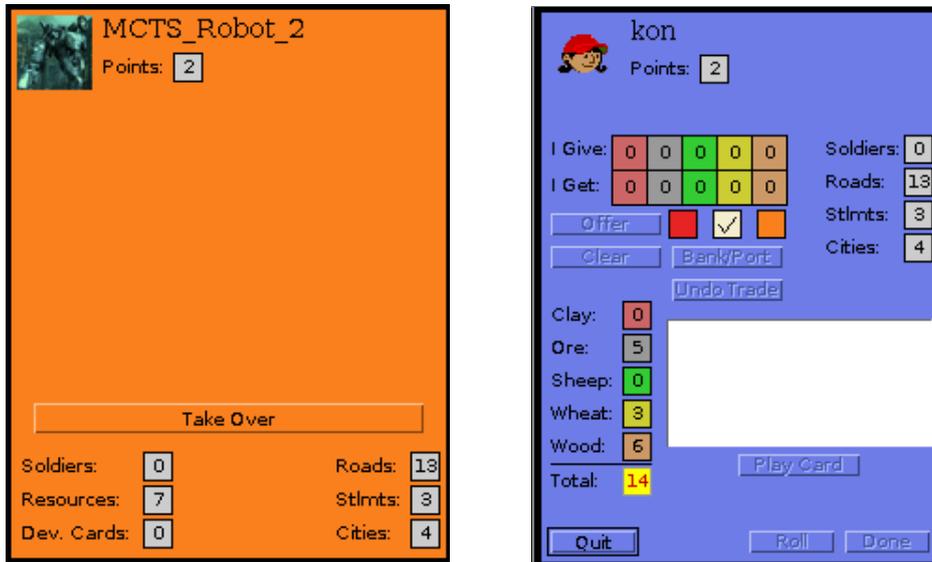
To sum up, in this section we described the basic elements of the interface needed for understanding the flow of the game. In the next sections we briefly present the agent implementation of Robert S. Thomas, which is considered to be one of the strongest implementations.

3.2.2 Agent Implementation

Robert Shaun Thomas in [35], dedicates three chapters in the agent implementation analysis.

3.2.2.1 Determining Options & Resource Estimation of Time

In Settlers of Catan, players spend most of the time thinking what to build next. Thomas outlines some of the basic strategies that a player can use to



(a) Other Player's Area.

(b) Player's Area.

Figure 14: Game Interface: Players Area

decide where to build his initial pieces. These strategies are related by the fact that they seek to maximise the player's "building speed" (speed at which a player can build).

Before deciding where to build, it is critical to know what your options are. The legal places to build are simple and are determined by the rules described in 1.2.2. The *SOCPlayer* object was created, in order to maintain a record of the legal places for each player. For settlements and roads, there are two lists in that object: one for containing the *legal places* (i.e it contains spots on the board where if the building requirements are met, a player can build) and another list for the *potential places* (which contains spots where the building requirements are met).

Now that the agent knows his options, it is crucial to decide where to build. The placement of the initial settlements is very important because not only it determines what resources will be produced but it also determines where he can build in the future. So a player must decide which subset of the five

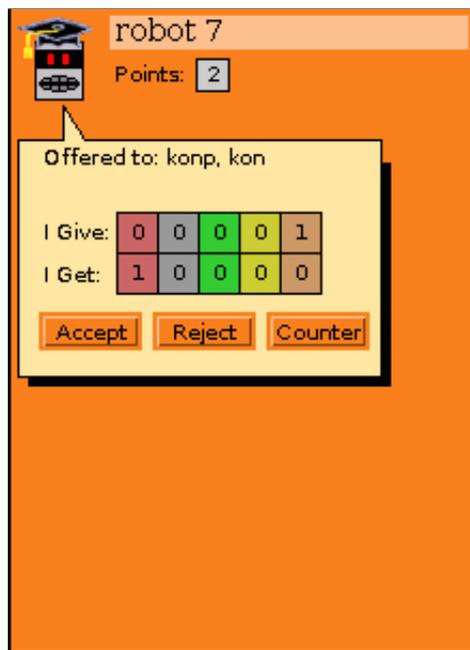


Figure 15: Game Interface: Trade.

resources he likes. This is linked to three potential “high level” strategies:

- road-building, which necessitates wood and clay production, which are used for roads and settlements builds,
- city-building, which requires ore and wheat to build cities or buy development cards and
- monopolizing, in which a player tries to monopolize a resource and have access to a matching 2:1 port.

To choose one of these strategies we must estimate how fast we can build something. There are two approaches to calculate the “building speed”, one taking into account trading with ports and the bank, and one not. None of the approaches takes the results of potential negotiations with other players into account.

1. The first approach can be divided into two parts:
 - (a) The first is the simplest way, in which we ignore trading, and consider how often we will receive resources based only on the types and numbers of the adjacent hexes of our pieces, but also by estimating how often these numbers are rolled. So in this approach, a frequency table is created to estimate how many rolls are needed to acquire a specific set of resources. As the author notes, there are some “intended” inaccuracies in this algorithm because it employs estimates to make different kinds of decisions. What we really have is an estimate, that is accurate relative to other estimates we make. So if we have an estimation speed that says that it takes much longer to build a city than a road, this should be true in reality too.
 - (b) The second part takes into consideration the possibility of trading with the ports, or with the bank. But this addition of the trading phase, can lead to an estimate that is inconsistent in some situations, because the algorithm does not look ahead concerning the resource production (e.g. when we build a city).
2. The alternate approach was based on the assumption that a more accurate estimation can lead to an improvement in the gameplay. Although, it was proved to be very slow to be used in practice, the author presents it as a motivation for future work. It is basically a modification to the algorithm described above, in which there was the restriction that a player can only receive at most one type of particular resource per roll. So we need to keep track of how many resources of a particular type player receives with a given frequency and construct a frequency table.

Precision is not always the most important consideration. In the case of the two presented approaches an algorithm that provided a rough estimation but was fast, was shown by Thomas to be the best solution.

3.2.2.2 Making a Plan and Deciding What To Build

Thomas[35] then proceeds to describe the methods used for the selection of the initial settlements and initial roads. For the initial settlements he considers all pairs of legal settlement spots, and takes the estimates of how long it would take to build each possible type (roads, settlements, cities and development cards) starting with no resources, and chooses the ones with the lowest building speed. If we wanted to find pairs for the other strategies of *road-building* or *city-building*, we must take weighted sums according to what buildings we want to build. For the placement of the initial roads we must take into account where other players are likely to place initial placements and where it will lead us in order to build good settlements later on. So what Thomas does is that he guesses where players are going to build, pretends to build settlements on these spots and out of the remaining legal spots picks the best and find an edge leading to that spot from the first settlement. This is where we place the initial road. He then uses a rough plan as a guide for creating a utility-based measure for making decisions in a dynamic environment with imperfect information (the details of the implementation can be found in [35] in *Chapter 4: Agent Implementation Part II: Making a Plan and Deciding What To Build* on page 86).

The results showed that the computer players win between 47% and 63% of the time competing against a single human opponent.

3.2.2.3 Negotiation and Trading

Before going into a negotiation one should know what he'll do if the an agreement is not reached. This is called the *Best Alternative to a Negotiated Agreement (BATNA)*[35][36]. If we do not have the resources that we want, we must determine our BATNA, by looking what resources we need and what resources can be used for trading. After determining our BATNA, we must make some offers. We start by reasonable offers of giving one resource that is easy to get for one resource that it is hard to get. By reasonable, we mean that the offer must be better than our BATNA and that we consider it as an offer that someone will accept.

To estimate if the offer is better than our BATNA, we must estimate the building speed with what we have now and in case the offer is accepted. If the latter case's speed is less than the first, then the offer is better. To determine if another player will accept our offer, we must keep track of resources he received recently and the offers that he rejected. We might lose track of the resources when he needs to discard or if he is robbed. After considering all the offers where we give an unneeded resource for one needed, we take into consideration giving a needed resource. Finally we consider giving two resources for one needed resource starting again with a combination of unneeded resources and then needed. Counter offers are made in the same way but we must make an offer that makes sense given the made offer.

Results of this attempt showed that the performance of the agents dropped and that the computer players were agreeing to offers that are not in their favor. So, we can see that BATNA, applied by Thomas is not yielding good results and in our future work, we intend to use *Game Theory*, *Machine Learning* and *argumentation* methods in order to improve the negotiation abilities of our agent in order to improve his overall performance.

4 Agent Implementation

In this chapter we describe the integration of our code in the *JSettlers Framework*, our implementation of MCTS for *action selection* in the *Settlers of Catan*, the creation of various heuristic strategies **apart** from the MCTS implementation and the *negotiation scheme* created for trading with other players.

As described in Chapter 3.2, Robert Shaun Thomas implements a “Settlers of Catan” agent, based on three sequential strategies. The first is to *determine the options and compute a resource estimation of time*, the second one is *to make a plan and decide what to build* and the third is *the negotiation and trading* of the agent. For the implementation of these strategies, many classes were created, in order to keep track of the state of the game and to decide the agent’s next move. The main package, containing code for the agent implementation of the *JSettlers* framework, is the SOC.ROBOT package.

In Chapter 4.1.1 we describe the main classes of this package used for agent planning in the *JSettlers* and how is our code connected to the existing implementation. In Chapter 4.2, we present the base Class Diagram of our implementation and some implementation choices. Chapter 4.3 is dedicated in the MONTE CARLO TREE SEARCH implementation, where each step of the algorithm is presented in detail and in Chapter 4.4 we describe the created strategies for dealing with various situations of the game apart from the MCTS algorithm. Lastly, in Chapter 4.5, we describe a simple negotiation scheme that was created in order to provide the agent the ability to trade with other players.

4.1 Changes in the JSettlers Framework:

In order to be able to create an agent to play in the *JSettlers* framework, we had to study the code structure in detail and make many changes. Specific code changes in various classes of the existing framework are described in Appendix A and below in Chapter 4.1.1 we describe the integration of our code to the existing structure with our created CLASSES.

4.1.1 Code Integration:

To integrate our code into the *JSettlers Framework*, we had to study the structure of the code. After extensive research we determined the classes needed and we present them in this section.

The *robot* package contains code concerning the agent designed by Thomas. In figure 16, some of the core classes of the *robot* package and their connection are presented.

The SOCROBOTBRAIN(Figure 17) class contains the AI for playing Settlers of Catan. This class contains many handler functions for treating messages and performing the selected actions. Before planning what to build, the agent checks if he has a *Development Card* -apart from the *Road Building Card*- and if so, he must determine if he'll play it and what choices will he make. So, when for example we have a *Monopoly* card and the agent uses the MCTS STRATEGY, the appropriate function of our implementation is called, in this case the *decideOnMonopoly()* function from our MONOPOLYSTRATEGY class in the HEURISTICSTRATEGIES package in order to determine if the agent will play the card. The same logic applies to the DISCARDSTRATEGY call, when a "7" is rolled. In this class exists the *planBuilding()* function, which calls the SOCROBOTDM's function *planStuff()*, in which we included calls to our classes to determine the agent's

next action. When the action is determined, SOCROBOTBRAIN checks if we have all the available resources for building the target pieces. In case we don't, the *makeOffer()* and *tradeToTarget2()* functions are called to trade with players and with ports and bank respectively. For trading with the ports of bank using our functions of the CHECKER and NEGOTIATOR classes, we created two functions in BRAIN class, called *makeOffer2()* and *tradeWithGame()*.

Decision-making code of the *JSettlers* framework, is implemented in the classes OPENINGBUILDSTRATEGY, ROBBERSTRATEGY, MONOPOLYSTRATEGY, SOCROBOTNEGOTIATOR and mainly in the SOCROBOTDM class. There is where the TREENODE.MCTS() function is called for initiating the MCTS implementation in order to decide what to build. In addition, after the decision, the agent checks if he has the *Road Building* card, and if so our heuristic strategy ROADBUILDINGSTRATEGY is called to determine the 2 roads to build. It is important to note that we did not modify the MONOPOLYSTRATEGY and the SOCROBOTNEGOTIATOR classes, but instead we created our own. The OPENINGBUILDSTRATEGY and the ROBBERSTRATEGY were kept as they were and they are used by our agent as described in Chapter 4.4.

The *SOCPlayerTracker* is used to track strategic planning information as possible building spots for itself and for other players. It wasn't necessary to modify this class but we mention it for further reference. Lastly, the *SOCPlayerClient* is a robot client for playing Settlers of Catan and we did not modify this class either.

All of our created classes and the code integration are described in detail in Appendix B, but below we present the basic class diagrams of our class creation and integration.

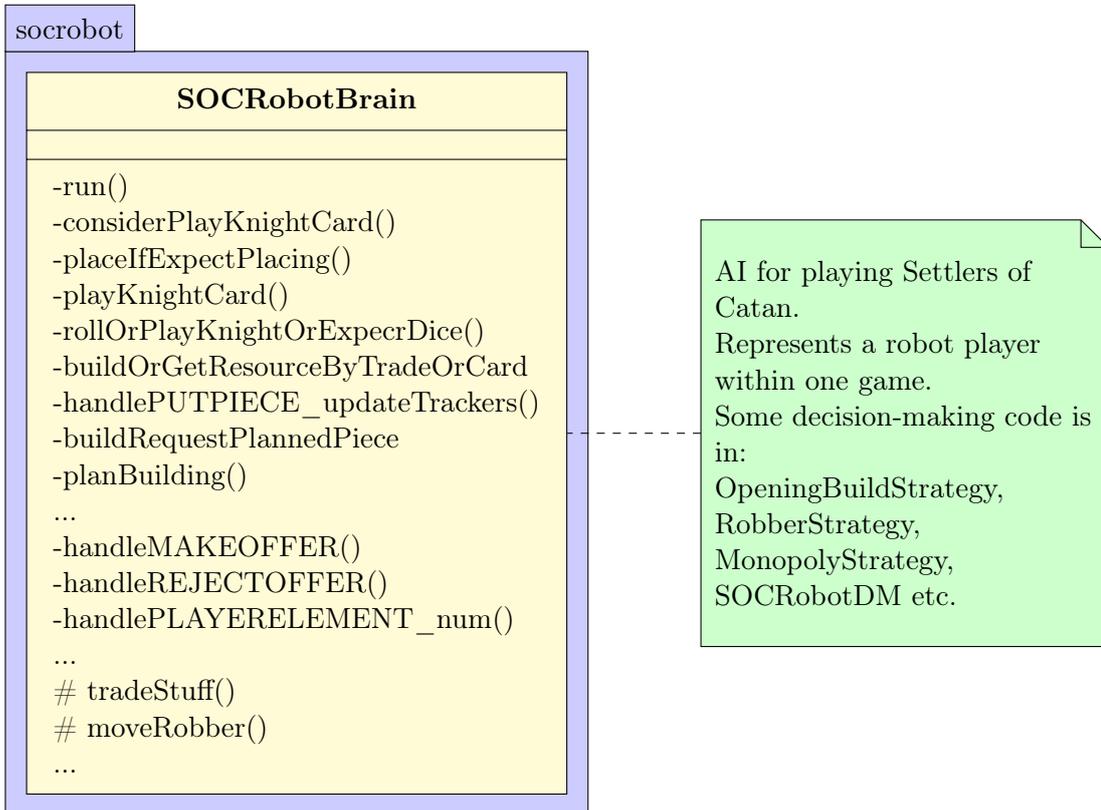


Figure 17: Class Diagrams: SOCRobotBrain

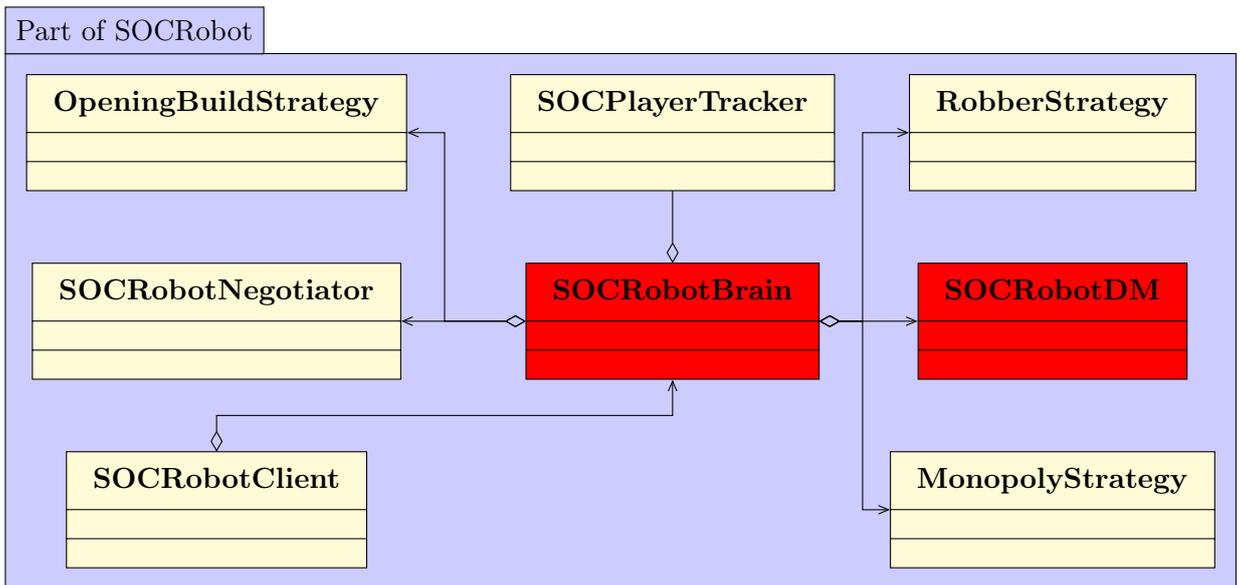


Figure 16: Class Diagram: Important Classes.

The highlighted classes are the ones that we modified in order to integrate our code.

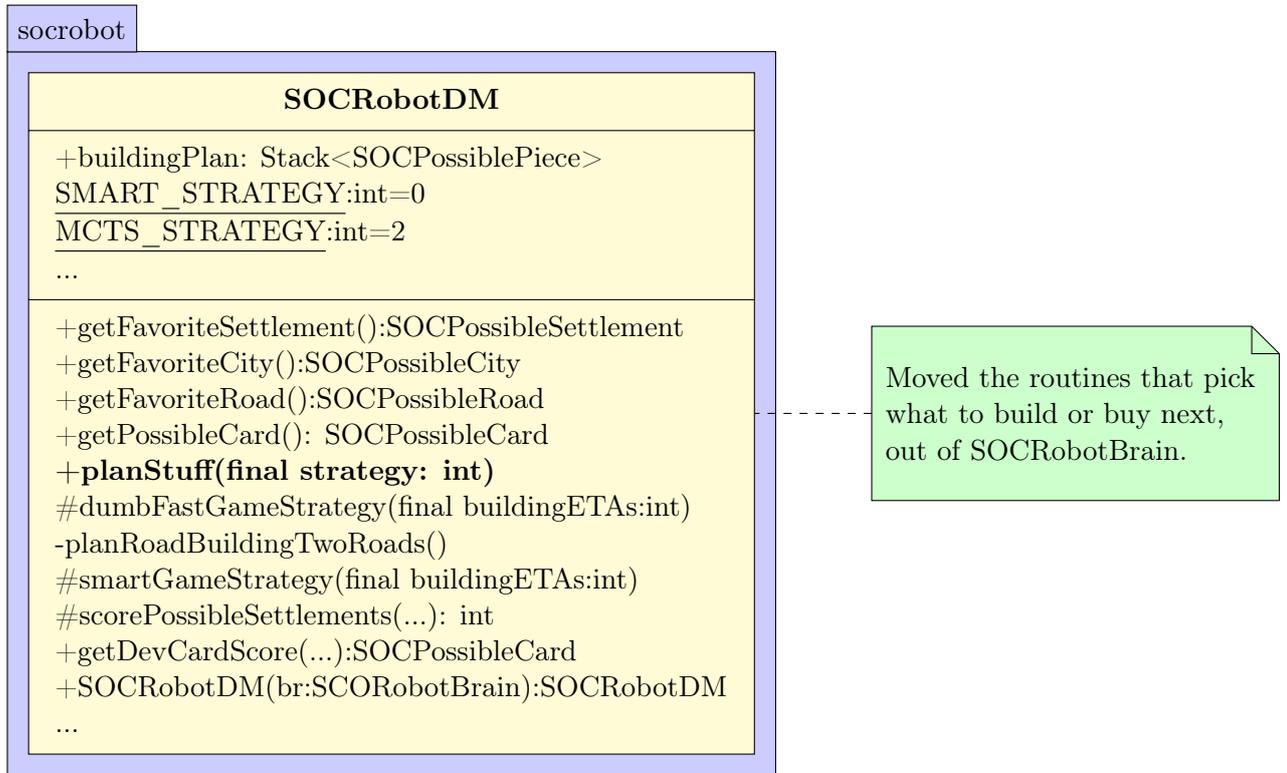


Figure 18: Class Diagram: SOCRobotDM

Initially, the routines that existed in SOCROBOTBRAIN, that pick what to buy or build next, were moved to SOCROBOTDM(Figure 19). We chose to modify this class as described above in order to implement the MONTE CARLO TREE SEARCH. The structure not only of the class but also of the entire package reinforced this decision.

4.2 Basis of the Implementation & Class Creation

We based the implementation of MONTE CARLO TREE SEARCH in the algorithms 1 and 2, which describe *General MCTS approach* and the UCT implementation in pseudocode, in the detailed background of the algorithm provided by Browne et al.[1] and in the minimal one-page MCTS

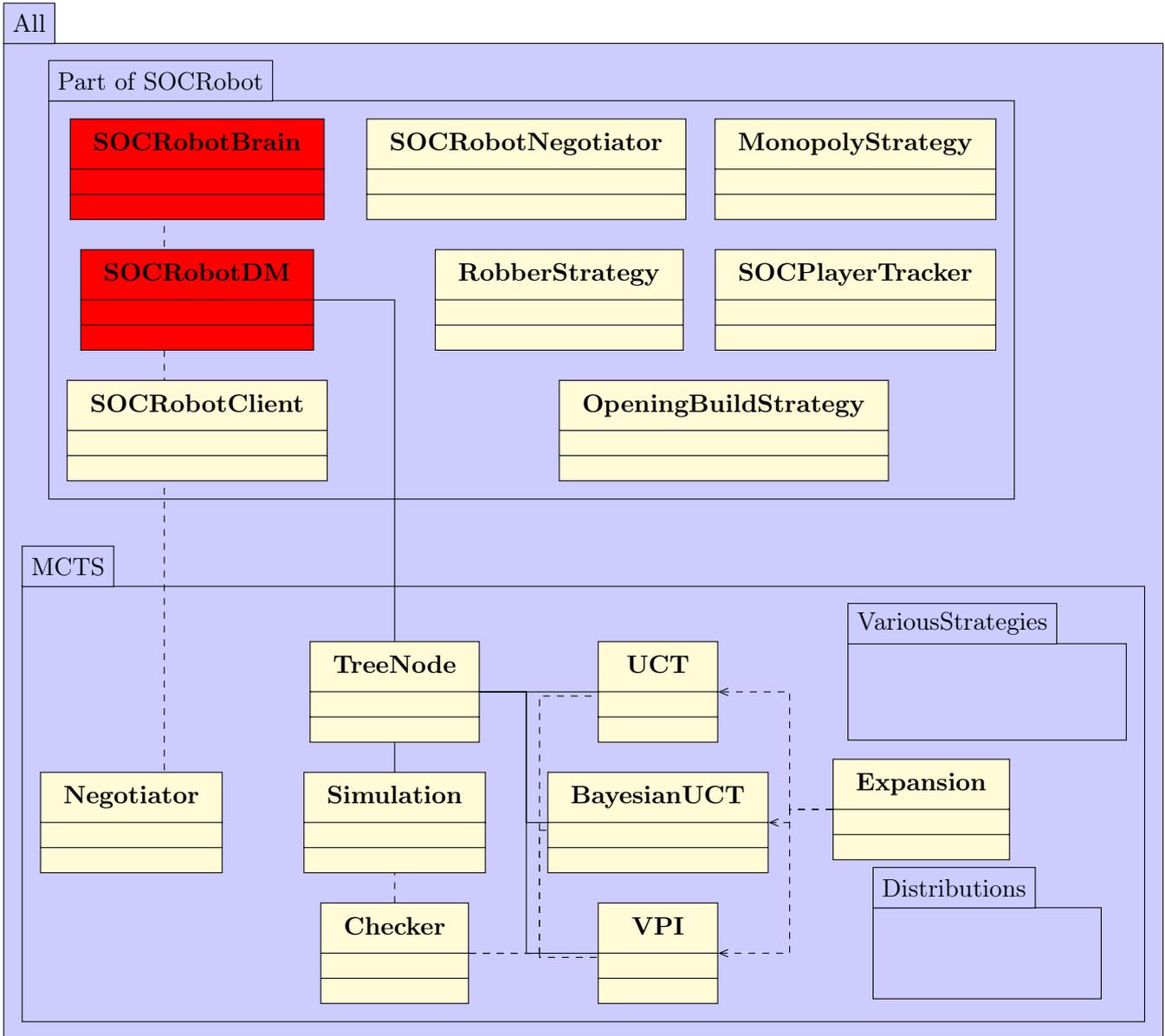


Figure 19: Class Diagram: Integration.

The highlighted classes are the ones that we modified and are connected to our classes.

implementation by Simon Lucas[37], which despite its simplicity, presents the operation of the algorithm. In that spirit, we created several classes and modified many classes of the *JSettlers* framework and made many changes to the code, which were described above in Chapter 4.1.1 (For code-specific changes see Appendix A and for UML class diagrams see Appendix B).

The main class is the `TREENODE` class, which represents a node in the `MONTE CARLO TREE`. In this class, there are variables needed for representing a state of the game as the player's *Settlements*, *Cities*, *Roads*, *Development Cards* and *Victory Points*, but also variables needed for the calculation of the *UCT*, *Bayesian UCT* and *Value of Perfect Information* values described in Chapter 2.8.2. These include the number of visits of the node, the total number of visits for all nodes, the average rewards that were returned for the simulation and a *Dirichlet* variable for representing the probability distribution over the expected rewards.

A `TREENODE` object thus corresponds to a state s of the game (at least from the point of view of the player, given the information that he has at hand), and potentially has children `TREENODES`, each corresponding to a state s' resulting from executing action a at state s .

The selection and expansion step of the algorithm (see 2.6) are implemented by the classes `UCT`, `BAYESIANUCT` and `VPI`, where the appropriate method is called through the `TREENODE` class.

In Figure 20, we present the class diagram of our implementation, which contains the aforementioned classes and their elements and in the next section we provide the details of the implementation and the choices we made in every step.

Monte Carlo Tree Search

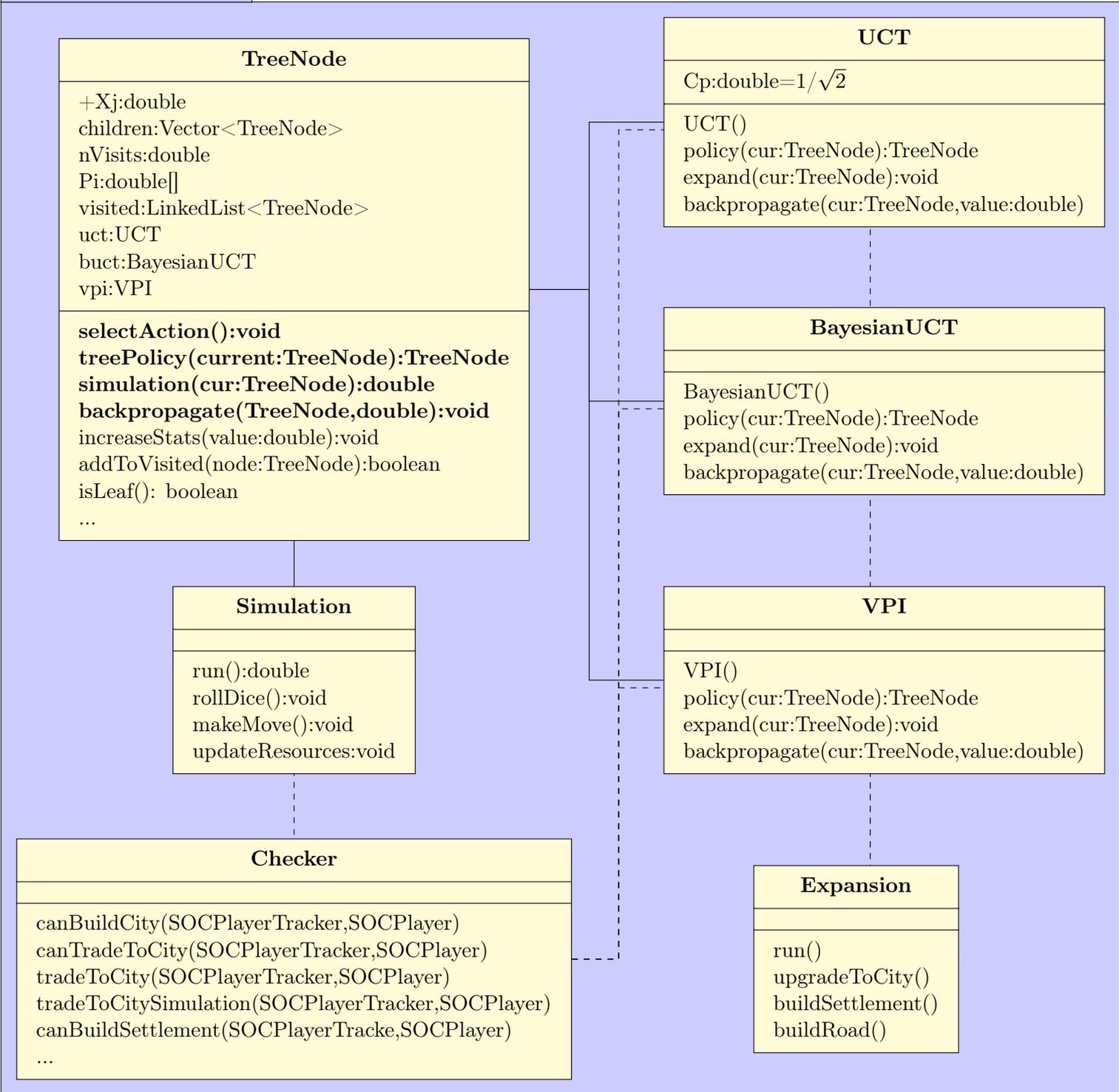


Figure 20: Class Diagram: Main MCTS Classes.

4.3 MONTE CARLO TREE SEARCH Implementation:

The MCTS algorithm is used for action selection, when the agent has to decide his next move. This move is determined by the set of available actions, which includes: buying a City, a Settlement, a Road or Development Card and where to build. For other actions of the agent, as determining if the agent will play the *Monopoly* card and which resource will he monopolize, various heuristic strategies were created and are described in Chapter 4.4. We had to make many choices concerning each step of the implementation of MONTE CARLO TREE SEARCH, starting with setting a computational budget for the main loop of the algorithm (as seen in Algorithm 1) that consists of the following condition:

1. **time cut-off:** We set the cutoff time limit to 7 seconds, based on the size of the state space and which we believe is reasonable in order not to “break” the pace of the game and frustrate the other players.

Having set the computational budget we present the individual decisions and implementation choices made for each step. Below, we present the four main steps of the MCTS algorithm, the *Selection*, *Expansion*, *Simulation* and the *Backpropagation*, which are described in Chapter 2.6 The *Expansion* and *Simulation* steps of the algorithm are shared between the methods in contrast to the *Selection* and *Backpropagation* steps, which require different computations and thus each method has its own implementation.

4.3.1 Selection Step

The selection step is used for selecting the *most urgent expandable node*, in order to descend and build the tree. A node is *expandable* if it represents a non-terminal state and has unvisited children. Which node is the *most urgent*

is determined from the selection formula of the method, and usually corresponds to the node that maximises this formula. This step is implemented in the classes UCT, BAYESIANUCT and VPI according to the formulas described in Chapters 2.8.2.1, 2.8.2.2 and 2.8.2.3 respectively. For clarity, in each of the following sections we restate the uses of these formulas and briefly describe their implementation.

4.3.1.1 UCT

The selection formula for the UCT method is:

$$\text{Maximise UCT} = \bar{X}_j + 2C_p \sqrt{\frac{2 \ln n}{n_j}} \quad (13)$$

where \bar{X}_j is the average reward of the node j , n the number of times the current (parent) node has been visited, n_j the number of times child j has been visited and $C_p > 0$ is a constant.

Kocsis and Szepesvári [20], showed that the value $Cp = \frac{1}{\sqrt{2}}$ satisfies the Hoeffding inequality with rewards in the range $[0,1]$. The rewards returned from our simulation are in the range $[0.2,1]$ and so this value was chosen for our implementation. This parameter can be adjusted in order to lower or increase the amount of exploration that will be performed.

Each `TREENODE` maintains its number of visits and the rewards received from simulation and when the selection step is called, all the necessary values are calculated in order to compute the above UCT value. The child of the current node with the maximum value is returned in order to proceed to expansion, simulation or to the final selection of our move.

4.3.1.2 Bayesian UCT

The selection formula for BAYESIAN UCT is:

$$\text{Maximise } B_i = \mu_i + \sqrt{\frac{2 \ln N}{n_i}} \sigma_i \quad (14)$$

μ_i : Mean of an extremum (minimax) distribution P_i

σ_i : Square root of variance of P_i

n_i : Number of visits to node i

$N = \sum n_i$: Total trials of all arms

P_i : Each node i in the tree maintains a probability distribution over its true expected reward value.

The BAYESIAN UCT selection formula is a little more sophisticated than the standard UCT method and introduces a Bayesian framework that allows more precise estimation of the node values and uncertainties.

In this case, each `TREENODE` maintains a probability distribution P_i over its true expected reward. Before any simulation, leaf nodes are initialized with conjugate prior distributions fitting the reward distributions. If our payoffs were 0/1, we could choose the *Beta* distribution but in our case we chose to use *Dirichlet* priors which are described in Chapter 2.8.2.2.1. After simulations are performed, the results are combined with the priors in order to produce a posterior distribution.

For our selection step, we produce the posterior distribution via updating the *hyper-parameters* α of the *Dirichlet* distribution and calculate (14) by taking the μ_i and σ_i for each node that result from the updated *Dirichlet* distribution. The child of the current node that maximises the value B_i is returned in order to proceed with our actions.

4.3.1.3 VPI

The strategy of VPI is to choose the action that maximises:

$$E[q_{s,a}] + VPI(s, a) \quad (15)$$

where

$$VPI(s, a) = \int_{-\infty}^{\infty} Gain_{s,a}(x) Pr(q_{s,a} = x) dx \quad (16)$$

where the Gain computes what can be gained from learning the true value $q_{s,a}^*$ of $q_{s,a}$:

$$Gain_{s,a}(q_{s,a}^*) = \begin{cases} E[q_{s,a_2}] - q_{s,a}^*, & \text{if } a = a_1 \text{ and } q_{s,a}^* < E[q_{s,a_2}] \\ q_{s,a}^* - E[q_{s,a_1}], & \text{if } a \neq a_1 \text{ and } q_{s,a}^* > E[q_{s,a_1}] \\ 0, & \text{otherwise} \end{cases} \quad (17)$$

For a more detailed explanation of the VPI formulas see Chapter 2.8.2.3.

The computation of the integral on (16) depends on how we represent our distributions over $q_{s,a}$.

Now, in order to be able to take decisions in our setting, each `TREENODE` corresponding to a state s' , maintains a distribution over the quality value $q_{s,a}$ of having executed the action a that led from (parent) state s to s' . Intuitively, this value corresponds to the “quality” of following a path in the game tree downwards from s . When a decision needs to be taken at some `TREENODE`, the value of its children (along with the corresponding value of information entailed in visiting these children) is calculated.

The calculations needed are facilitated by the use of appropriate conjugate priors, enabling the easy update of the distributions mentioned above. Namely, the distribution over $q_{s,a}$ is a *Dirichlet*, described by a set of hyper-

parameters α , where each parameter α_i represents the frequency of seeing reward i , and these frequencies are updated through the observation of simulation results. For these calculations we first find the two best actions a_1 and a_2 from the set of available actions and we estimate the Q-value distributions using *sampling* on the existing *Dirichlet* distributions. So essentially, these *Dirichlet* are used in order to implement the sampling process used in *Model-Based Bayesian Exploration*[10]:

When we want to sample a vector θ according to the distribution $P(\theta | \xi)$ we use a simple procedure:

We sample values y_1, \dots, y_K such that each $y_i \sim \Gamma(\alpha_i, 1)$, where α_i is the concentration parameter in the index i of the α vector and $\Gamma(\kappa, \theta)$ is the *Gamma* distribution. Then we normalize to get a probability distribution [10].

Having these values, we calculate Eq.(17) and (15)), and we then select the child that maximises Eq. (15).

To sum up, one could view the implementation of *Value of Perfect Information* described above, as acting according to an underlined belief-state MDP, which belief states correspond to probability distributions over rewards. These are used in determining the best possible action based on our beliefs.

4.3.2 Expansion Step

Even though the expansion step is shared between the methods, we chose for it to be called inside of each method due to the classification of the SELECTION and EXPANSION step as TREEPOLICY. To implement the expansion step we created a class called EXPANSION, and we had to find the set of available actions of our agent in the given state and a create TREENODES according to the combinations of the selected actions. These actions include:

buying a *City, Settlement, Road* or *Development Card* and choosing where to build. The structures used by Thomas [35], provided a way to access a great part the action set for each state with the acquisition of the *potential cities, settlements and roads*. In our expansion, we also consider the acquisition of Development Cards and the only thing we need is to get the amount of available Development Cards from the game. It is important to note, that our agent does not know beforehand what kind of *Development Card* will arise and so there isn't an immediate reward concerning the acquisition of a Development Card. Before buying a piece or a card, there are also some other requirements that must be met. Basic element in defining the set of available actions is the set of resources that our agent has in his possession. So even if there are *potential pieces*, if we don't have the required resources (as defined in Chapter 1.2.2) we can't consider any possible action.

But if we do not have the required resources, can we acquire them through a bank or a port trade?

In that spirit, in order to explore its available moves, our agent takes into consideration his *potential pieces* and his *available resources*.

We start with a specific game piece type, and we check its *potential array*:

1. If the array is empty we move on to the next piece type and we continue until we find a piece type that has a non-empty array.

If all arrays are empty, then the current node is considered to be fully expanded; otherwise we have an empty actions set and we check if we have more than 7 resources in order to consider trading with a port, the bank, or other players in order to avoid the situation in which a "7" is rolled and we need to discard half of our resources.

If the node is fully expanded and we cannot add any more children, we proceed with the algorithm, in which case the *selection step* is applied

in order to select the best child, from which a *simulation* will be run. If the node has no children and we cannot add more due to the lack of pieces or resources, we assume that we do not have the ability to do anything in this turn, so we stop the algorithm.

2. If an array of a piece type is non-empty, we look at our available resources to assess if we have the necessary subset in order to buy and build that type.

- (a) In case we have the subset, we select the first piece in the *potential array* and we put it in a stack called BUILDINGPLAN, which will be part of the child TREENODE that we'll create.

Then we must temporarily remove the piece from the *potential pieces* and the needed resources from the *agent's current resources* in order to check for the ability to buy or build the next piece. The last part is very important, because a player can build any number of pieces he has the ability to in a turn-and from our experiments building more than one pieces per turn (if we can), yielded far better results.

- (b) In case we don't have the subset, we look into our available resources and consider the possibility of trading with a port or with the bank. If we have enough "*unnneeded*" resources that we can give in order to get the *missing resources* needed for building the piece, we put the piece in the BUILDINGPLAN and we proceed as in step (a). Otherwise, we conclude that we cannot build anything of that piece's type and we proceed to the next type.

As mentioned, the player can buy pieces, representing *Cities, Settlements, Roads* and *Development Cards*. Each piece needs different resources

and gives different advantages to the players.

In our implementation of the expansion step we made some decisions in order to reduce the space state and insert some kind of heuristic knowledge in the algorithm:

1. One set of decisions concerns the *number of actions to be explored and how to choose the next piece*. We decided the number of actions to expand to be the number of *potential cities and settlements* available for building and for each iteration we have an inside loop that considers roads to add, based on the number of *potential roads*. When a piece has been selected, it is placed in the appropriate *child vector*, in order to avoid looking the same pieces over and over again. In this way, not only do we add enough actions to the tree (from which the best will be selected through the *selection formulas*) but also choosing pieces by order of appearance guarantees that the algorithm is fast and that more time can be dedicated to the *simulation*, instead of searching for the best piece per type.
2. To reduce the state space, we decided to add some conditions to restrict the options concerning *Roads* and *Development Cards*:
 - (a) For *Road* pieces, we first look if we have the *Longest Road* card.
 - i. If we don't, we need to build more roads in order to obtain this card and the 2 *Victory Points* that accompany it.
 - ii. In case we do have the *Longest Road* card, we check if we have a spot in which we can build a settlement. In case we don't have, we need to expand our roads in order to "create" spots for possible settlements, so we consider adding roads to the child's building plan.

So, if we have the *Longest Road* card and we have a potential spot for settlement, we do not consider adding roads to the child’s building plan. In this way, we “tell” the agent to mainly consider building *Cities*, *Settlements* or buying *Development Cards*, which will give them more options.

- (b) For *Development Cards*, we restrict the agent’s ability to buy them. The way that this is done, is that we see what Development Cards we have in our hands, and if we have unplayed cards, the agent won’t buy a new one until this card is played. In this step we must take into consideration if the agent has *Victory Points* cards, which cannot be played. So, if our total number of development cards minus the number of Victory Points cards is greater than one, the agent won’t buy a new one. This is done to avoid the unnecessary and continuous purchase of Development Cards through the exploration of actions.

After the implementation of these heuristics, in order for the agent to determine his options concerning buying a road or buying a *Development Card* (the *Cities* and *Settlements* consideration does not change) , he must first look at the above “restrictions”. If none of these apply, then he will consider all the possible options for buying game pieces (*Cities*, *Settlements*, *Roads* and Development Cards).

4.3.3 Simulation

The simulation step is also shared between the methods, but this time by being a *policy* by itself, we chose for it to be called from the general TREE NODE class. The class SIMULATION implements the simulation steps of the algorithm, and these steps are described below.

Firstly, we need to define a cut-off condition for the simulation loop consisting of:

1. **end-of-game cut-off:**, if a player acquires 10 or more *Victory Points*, we have a winner and the simulation returns the result Δ in order to proceed to the backpropagation step.
2. **turn cut-off:**, if a game exceeds 60 turns, we stop the simulation loop and again we return the current result Δ for the backpropagation step. This number of turns was chosen by observing the turns needed for a game to end when agents play against each other. Usually, the game needs approximately 17-20 rounds and each round consists of 4 turns for the players. So each game need 68-80 turns to end. We chose to set the cut-off, a little less in order to be able to perform more simulations and get a better estimation of the probability distribution over the expected rewards.

Afterwards we needed to create some functions to reconstruct basic behaviors of the game. For that purpose, we created a function for simulating dice rolls according to the probabilities:

Dice result	Combinations	Probability
2	1	1/36
3	2	2/36
4	3	3/36
5	4	4/36
6	5	5/36
7	6	6/36
8	5	5/36
9	4	4/36
10	3	3/36
11	2	2/36
12	1	1/36
Total	36	1.000

Figure 21: Dice Outcome Probabilities.

A random generator is used to generate a number in the $[0,1]$ interval and according to that value we decide the corresponding dice result.

As the rules suggest, after each dice roll, the players get resources according to their pieces on the board. For each player we look for pieces adjacent to hexes with a number matching the simulated dice result and we temporarily add the resources gained by the dice roll. In the end of the simulation, all the gained resources are subtracted.

Then the player can consider taking actions. The first move belongs to our agent due to the *expansion step* and so we take the pieces from the selected child from which we start the simulation and we put them into the game. We proceed with the next player, for whom we generate moves according to the a simpler logic based on the expansion logic described in the previous section. For each turn and player action a new `TreeNode` is created -but it is not added to the tree-, and that `TREENODE` contains the “owner” number and the `BUILDINGPLAN` that the player decided. All the pieces are temporarily put into the game board by using the `PUTTEMPPIECE()` function of Thomas

and resources are subtracted from the sets of the player in order to “simulate” a normal game. This loop continues until one of the cut-off conditions is met in which case we restore the game to its original state in order to prepare for the next iteration of the MONTE CARLO TREE SEARCH algorithm, after the backpropagation step.

Even though the actions chosen by the simulation are usually random in order to be fast, games simulated in that way tend not to be realistic. This is why we chose to apply the expansion logic during move generation but by omitting the option for *Development Card* acquisition due to the complexity of its implementation. This complexity would lead to major delays in the simulation step due to the fact that we maintained the hidden elements of the game and we do not know what Development Card will come up, and if we included this option in the simulation step, a need for exploring all the possible Cards and planning actions for each card separately would be mandatory. Even if our agent acquired a *Development Card* in a previous round, we don’t consider its play during the simulation. The main reason for this decision is -again- related to the complexity that would arise. The consideration of when to play a Development Card and the resulting actions could produce unwanted delays to the simulation, i.e., when we play a *Soldier/Knight* card, we must move the robber to another hex and we must steal a resource at random from a player that has an adjacent City/Settlement to that hex. In that way valuable time from the simulation will be consumed in order to determine the new hex, the victim, and the random stolen resource. The same logic applies to the *Monopoly* card and the *Road Building* card, where in the former case we must *monopolize* a resource when the agent judges that will be of benefit to him and in the latter, we must choose the 2 best roads for placing.

4.3.4 Backpropagation

Each of the selected enhancements has a different backpropagation method due to the variables used in the different calculation formulas, and even though we could include these methods in the general `TREENODE` class, we decided to include them in each class separately for easier maintenance and upgrading purposes.

4.3.4.1 UCT

For the backpropagation method for *UCT*, the process is quite simple and straightforward.

The selection formula was showed in the previous section (see Eq. (13) above). After the simulation reaches a terminal node, all we have to do is backpropagate the result of each player to the appropriate *visited nodes*. The result is defined by the victory points of each player at the terminal node, divided by 10, to get reward values in the interval $[0.2,1]$. For each of the visited nodes of the simulation, starting from the terminal node, we access its ancestor and we “inform” the nodes about the result.

- If the node represents an action made by our agent during simulation we add the $\frac{\text{Agent Victory Points}}{10}$ to the existing total reward X_j of the node.
- If the node represents an action made by another player, we subtract that player's $\frac{\text{Player's Victory Points}}{10}$ from the total reward X_j of the node.

4.3.4.2 Bayesian UCT

The selection formula for the *BAYESIAN UCT* is seen in Eq. (14) above. So for the update of the values needed for the calculation of the Bayesian UCT value, we need to update the *hyperparameters* α of the *Dirichlet* distribution

for each node. So according to the result Δ of the simulation, we update the appropriate index value of the α vector:

$$\alpha_i = \alpha_i + 1 \tag{18}$$

Essentially the α vector acts as a pseudo-count of how many times we have seen each reward, so we update the number according to the result by adding 1 to the counter.

4.3.4.3 VPI

The update procedure for the VPI method is based on the update of the hyper-parameter vector α of *Dirichlet* priors. These hyper-parameters α_i , represent the frequency of seeing reward i and are used for sampling possible expected reward distributions, in order to determine which is the best action from the available. The update procedure is described in Eq.(18). The calculation of the best action requires the solving of the equations for selection for the VPI: Eq.(15),(16) and (17).

4.4 Heuristic Strategies

Apart from the MONTE CARLO TREE SEARCH implementation, which is described in Chapter 4.3 and is used for action selection (determine what we can buy: Cities, Settlements, Roads and Development Cards), various strategies **needed to be implemented** in order to cope with various situations of the game. These strategies derive from the characteristics of the game and are presented in detail below. The *JSettlers* framework already contains implementations for these strategies and we use many of them as they are, due to their good implementation. So, from the following strategies, the *Open-*

ing Build and *Robber* Strategies are those used in the *JSettlers* framework and the rest, the *Monopoly*, *Road Building* and *Discard* Strategies were separately created to suit the needs of our agent. These created strategies are included in the package HEURISTIC STRATEGIES and a different class was created for each for easier maintenance and modifications. The HEURISTIC STRATEGIES package is presented in detail in the form of a class diagram in Appendix B and specifically in Figure 39.

4.4.1 Opening Build Strategy

In the beginning of the game, each player places two settlements and one adjacent road to each settlement. The *Opening Build Strategy* is about the selection of the spots of the initial pieces. In this case, we chose to use the existing OPENING BUILD STRATEGY of the *JSettlers* framework, which uses the some metrics to decide where the initial settlements will be, as the building speed for a pair of initial settlements and if a settlement is near a port. The choice of roads at the initial placement phase is based on the best nearby potential settlements and so it favors spots near ports and spots with high numbers.

4.4.2 Robber Strategy

When a “7” is rolled or when a *Knight/Soldier* card is played, the player that rolled the “7” or played the card, must move a special piece called *The Robber*. When a robber is placed on a hex, that hex stops *production*, meaning that if the number on it, is rolled, the players with a piece adjacent to it, won’t get any resources. Not only that but the player that moved *The Robber* can select a *robber victim* from who he can steal one resource at random. The *JSettlers* framework contains a built-in class called ROBBERSTRATEGY,

which we also use for the selection of the best hex and the selection of the *robber victim*. To decide the best new hex for the *Robber*, we first calculate the estimated time of winning for each player and find the player with the least time. Then we search for the best way to obstruct that player, by looking the building estimates for each piece of the game for each of his hexes. The hex with the largest total building speed for all the pieces is chosen as the best hex, because in that way we essentially “delay” that player’s actions. If something goes wrong and the best hex is the current hex of the *Robber* we choose a hex at random.

The decision concerning the *robber victim* is made in a similar way, by calculating estimated time of winning, but this time only for the players that are possible victims (players with a piece adjacent to the selected hex). The player with the least time is selected and returned to the server. Again if we can’t decide on a *victim*, a random choice is returned to the server.

4.4.3 Monopoly Strategy

One of the available Development Cards that a player can buy is the *Monopoly Card*. When a player plays this card, he must select one type of resource. Then all the other players must give him **all** of the *Resource Cards* of this type that they have.

When the agent decides to play a card and that card is a *Monopoly Card*, he must select a resource. The MONOPOLYSTRATEGY class was created for the purpose of selecting the best resource. The agent takes into consideration his nearby ports (Specific and General) to determine the trade ratio with which he can trade and the available pieces and options for future actions and decides which resource is best to monopolize and if it’s the best time to play this card. The agent might decide that he will not gain as much as

needed if he plays the card right now, so he keeps it and will consider playing it in the next round.

4.4.4 Road Building Strategy

Another type of Development Card is the *Road Building* card, which gives the player the ability to place immediately 2 free roads on the board (according to the Game Rules). To cope with this situation we did not create a new class but instead when we play this card, a function is called to determine the best spots for the two roads based on the *Longest Road* potential and the ability that their combination provides in order to build more settlements.

4.4.5 Discard Strategy

When a seven is rolled and a player has more than 7 resources, he is obligated to discard half of his resources. If the number is odd, the number of card discards for the player is rounded down to the closest integer (i.e. if the agent has 9 resources, he must discard 4). In this case, even though *JSettlers* already contains a Discard Strategy we chose to create a new class `DISCARDSTRATEGY` to be able to discard resources according to our needs. The agent determines his current state (if he has the *Longest Road* card, if he has a spot to build a settlement, if he is able to upgrade to a city,...) based on the state of the game and then constructs a set of *needed resources* and a set of *“leftover” resources*. With these two sets and with the function `resourceRollStats()` of the *JSettlers* framework, which returns descending order how easy will it be to obtain a resource based on current state of the game and dice roll probabilities, we determine the resources to discard first from the *leftover resources* set. If the size of the discard set is smaller than the number of cards we need to discard, we look to the *needed resources* set to

fill in. If something goes wrong and the agent cannot correctly determine the appropriate resource set to discard, a function of SOCGAME is called, which provides a random set to discard based on our current resources.

4.5 Negotiation

Trading with ports or with the bank and negotiating with other players for resources is a very important aspect of the game. Many times a player does not have access to all the necessary resources in order to build pieces or buy development cards. Considering the importance of this feature, we gave our agent the ability to trade according to his resources and even consider trading with other players if a trade can be of benefit to him. Szita et al. in [4], deprived the agent of the ability to trade with other players, admitting though that this deprivation creates a “handicap” for their agent. In this thesis, we use the existing structures of the *JSettlers* framework in order to consider and initiate trades. In Chapter 4.5.1, we describe the logic used for trading with the ports and bank and in Chapter 4.5.2, a simple scheme that we created in order to trade with players.

4.5.1 Trading with Ports & Bank

Trading with ports and with the bank is an important part of the game for the purpose of acquiring the necessary resources in order to buy *game pieces*. As mentioned in the Game Rules section (Chapter 1.2.2), a player can trade with the bank with a ratio 4:1, meaning that he can give 4 resources of one type to acquire 1 resource of another. The players also have the ability to trade with two types of ports if they have an adjacent piece:

1. *General Ports*, in which the trade ratio is 3:1 and

2. *Type-Specific Ports*, in which, according to the specific type of resource a player can trade 2 resources of that type for 1 resource of another.

Our agent has the ability to carry out these transactions by checking the *Port Flags* array that exists in the *JSettlers* framework, in which if the agent has a piece (settlement or city) adjacent to a port, the appropriate array entry contains the TRUE flag. By checking this array the agent can determine the trade ratio for his transactions, something that is later used for trading with players. If all the values of the array are set to FALSE, the ratio is set to the default trade ratio of 4:1.

It is important to note that in each call of the trade functions, the trade ratio is re-defined in order to determine if a new settlement or city was built near a port.

This ability is not only limited to when the agent wants to make a move on the board, but also during simulation, where we make *fictitious trades* by subtracting and adding the resources according to the specified trade ratio.

4.5.2 Trading with Other Players

As mentioned in Chapter 1.3, compared to Szita et al. [4], where they do not allow their agent to accept or initiate trade with players, we implemented a simple negotiation scheme in the NEGOTIATION class.

Our implementation of negotiation, which we intend to further develop, consists of several functions, which provide the agent with the ability to consider what to trade with other players or if it will be of benefit to him to accept a proposed trade.

Our agent initiates trades with other players when there is the need to trade for acquiring resources. In Chapter 4.5.1 above, we described the

process of determining the trade ratio available to the agent.

After the trading ratio is defined and before the final transaction with the ports or bank, the agent considers trading with other players. The trade offer is not addressed to all players, but instead the agent, checks for the current state of the other players. If a player is close to victory and we see that the trade that we are proposing can be of benefit to him, we do not consider him for the offer. To determine if the trade is of benefit for another player, we use the existing structures and the specific state of the game to determine their options for future plans. For example, if a player has 9 Victory Points and has the ability to build a settlement, we won't propose an offer that can give him the ability to build that settlement and be declared a winner.

Even though this is an aheuristic technique, we experimentally found it to be of benefit to our agent, giving him access to resources with a smaller trading ratio.

Similar logic is also used when we want to determine if we'll accept an offer from another player. Obviously, the other player will make an offer that will help him with his current plans, but maybe it's an offer that can be of great benefit to us. So again, we take into consideration our possible actions and what can be done with the acquisition of the resources contained in the trade offer.

This *negotiation* scheme is very simple and is open to further development using *Game Theory*, *Machine Learning* and *argumentation* techniques and is part of our future plans, but we didn't want to deprive the agent of the ability of trading with other players and essentially alter the gameplay.

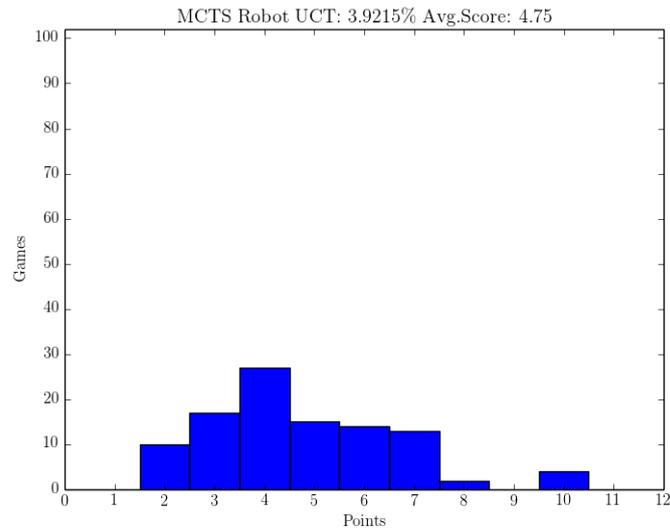
5 Experimental Results

For each different enhancement between *UCT*, *Bayesian UCT* and *Value of Perfect Information* of the selection step of the main MONTE CARLO TREE SEARCH algorithm, we tested our agent against the *JSettlers* agents. Below we present the results in order and provide some insights in why each enhancement produced these results.

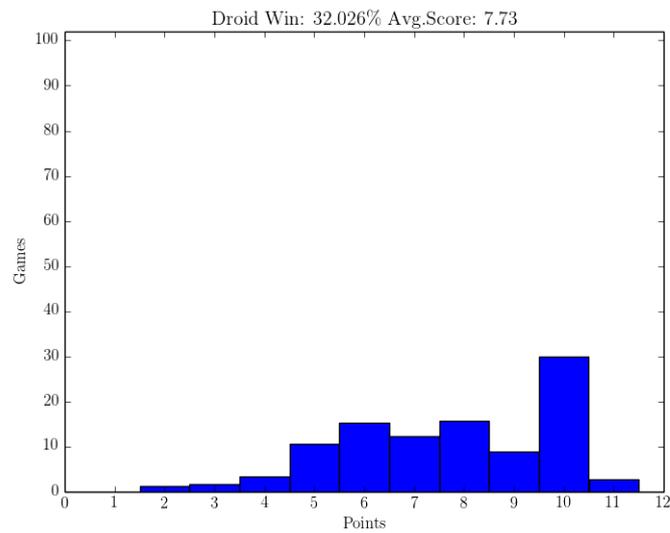
UCT

In Figure 22 we see the results concerning the standard MONTE CARLO TREE SEARCH Algorithm with the use of UCT in the selection step. We can see that the agent does not achieve a good average of Victory Points and only wins about 4% of the total games. The probability distribution of the rewards of this agent is centered towards the 4 Victory Points and rarely exceeds the 7 Victory Points. We can interpret our results based on the simplicity of the standard UCT approach, which is not appropriate for such a complex domain as the “Settlers Of Catan” and some kind of enhancement to the selection step must be used (As *Search Seeding* used by Szita et al[4]). Apart from the numerical values of the results one can observe the behavior of the agent during the game, in which many “unfit” moves are performed, i.e., ignoring the possibility of a road giving the agent the *Longest Road* card and the 2 Victory Points that go along with it. A great factor to consider and that can greatly affect the agent’s behavior is the number of simulations per move performed by our agent. For comparison with [4], where their agent performs 1000 to 10000 simulations, our agent only performs 120 to 130 simulations per move, creating a major handicap concerning the evaluation of the possible moves, leading to a selection of “bad” moves. Our future work includes the creation of a heuristic pruning in order to avoid

adding “bad” moves to the tree, reducing its size and making the algorithm faster.



(a) UCT agent/7sec/120iteration



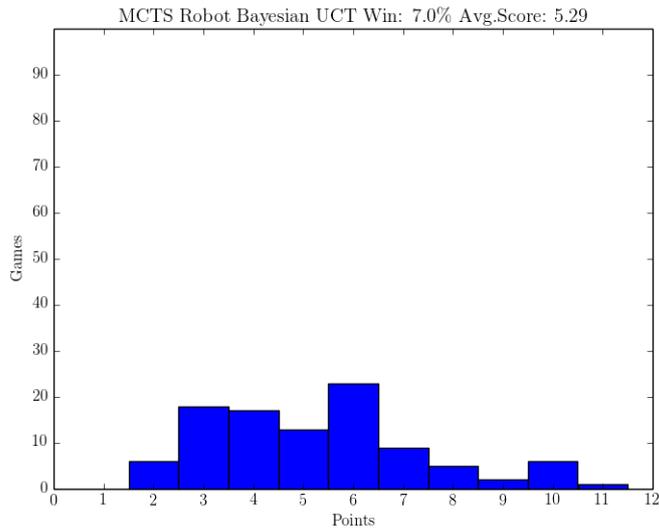
(b) JSettlers agent.

Figure 22: Results: UCT agent against JSettlers.

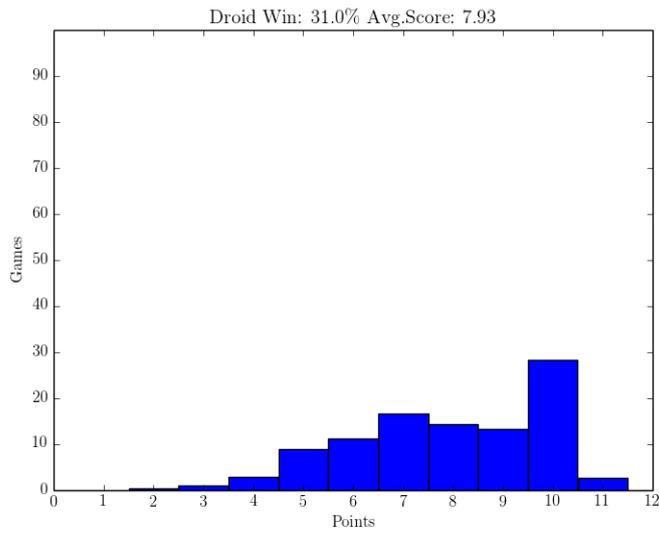
Bayesian UCT

In Figure 23 we see the results with the introduction of a Bayesian framework to the standard UCT approach. Due to the added complexity to the calculation of the selection values based on Eq. 14, we see that the agent performs even less simulations than the UCT approach but achieves better results.

In this case, the agent’s probability distribution of rewards is centered around 6 Victory Points, the average achieved score of the agent has risen from 4.75 to 5.29 and the agent’s winning percentage rose from 4% to 7%. Even with less simulations, the Bayesian framework allows a better estimation of the values of the tree nodes making the selection more effective. If we observe the agent’s behavior during plays, we can see that more “sophisticated” moves are performed, i.e. building settlements even when possible roads exist, but still there exists the possibility of performing “bad” moves, such as setting a road where no new possibilities or Longest Road potential exists. Even though the agent’s results are better with this method, room for improvement still exists and we believe that the number of simulations is again the main factor of this result. During the writing of this thesis, we test the agent’s behavior given more time to perform the MCTS algorithm. The results as the time increases are better, but not only there are some “timing” issues with the *JSettlers* framework but as the tree becomes larger, the selection step (mainly) consumes more time per iteration, creating a need to make the selection step more efficient.



(a) Bayesian UCT agent/7sec/80iteration



(b) JSettlers agent.

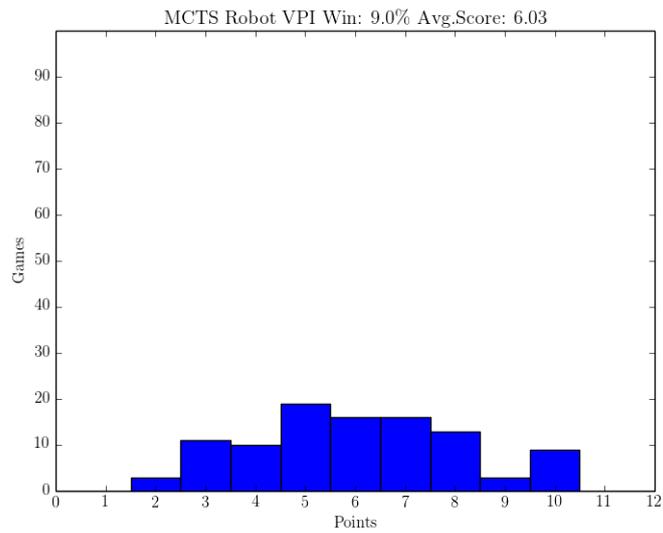
Figure 23: Results: BAYESIAN UCT agent against JSettlers.

Value Of Perfect Information

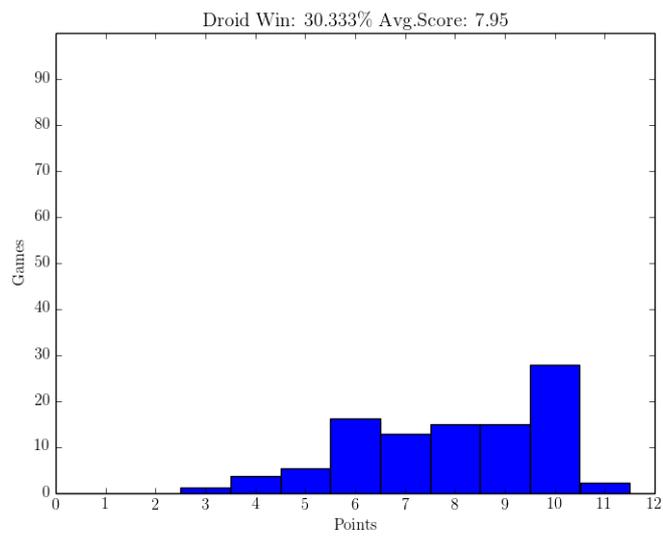
In Figure 24, we see the results of our last method, the Value Of Perfect Information. This method needs the most computational budget out of the

three methods. This complexity arises due to the need for “sampling” over the possible reward distributions using the *Dirichlet* priors of each node. This sampling is necessary for implementing the Value Of Perfect Information method, based on Eq.15,16 and 17, where sampling is used to calculate the last two equations and essentially the third. In our implementation we chose for the number of samples of possible reward distributions to be 100 and from experiments we concluded that in the current state of the implementation, this is the best number (when we improve the efficiency of the selection step, we may increase the number of samples).

Even though fewer simulations are performed (the number of simulations for the action selection in one turn dropped from 120 to 130 for UCT and 80 for Bayesian UCT to 10-20), the agent’s behavior is the best out of the three methods. The selection procedure of VPI is the most “sophisticated” compared to UCT and BAYESIAN UCT and the approximation of the expected reward for an action of the agent is very accurate despite the less simulations. And the results confirm this. Not only the winning rate rose from 7% to 9-10%, but also the average score rose from 5.29 to 6.03, closing the gap with the 6.43 average score of the *SmartSettlers* implementation. We believe that with the increase of time and simulations this method can outperform the *JSettlers* and *SmartSettlers* performance. The number of “bad” moves has dropped dramatically leading to better moves and better scores for the agent. An example of the agent’s behavior during a game is presented below in figure 25.



(a) VPI agent/7sec/12iteration



(b) JSettlers agent.

Figure 24: Results: VPI agent against JSettlers.



Figure 25: Value Of Information Gameplay.

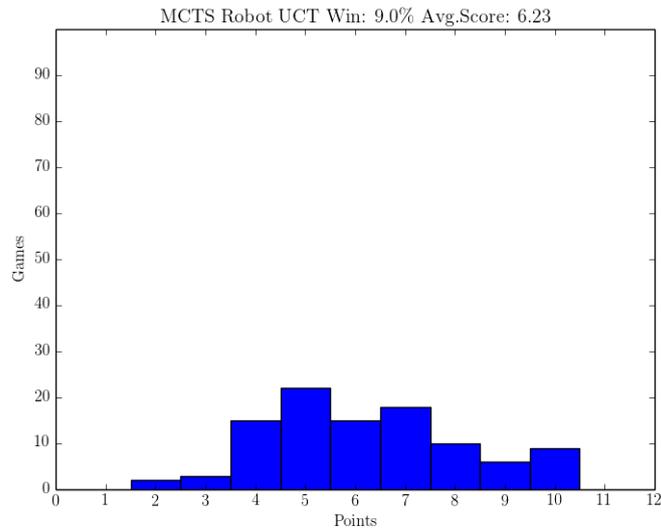
Methods Comparison

In this section, we tested the three different methods playing against each other in a setting with 2 MCTS agents and 2 *JSettlers* agents. So, below we present the agents behaviors when a UCT agent plays against a BAYESIAN UCT agent, when a UCT agent plays against a VPI agent and finally when a BAYESIAN UCT agent plays against a VPI agent.

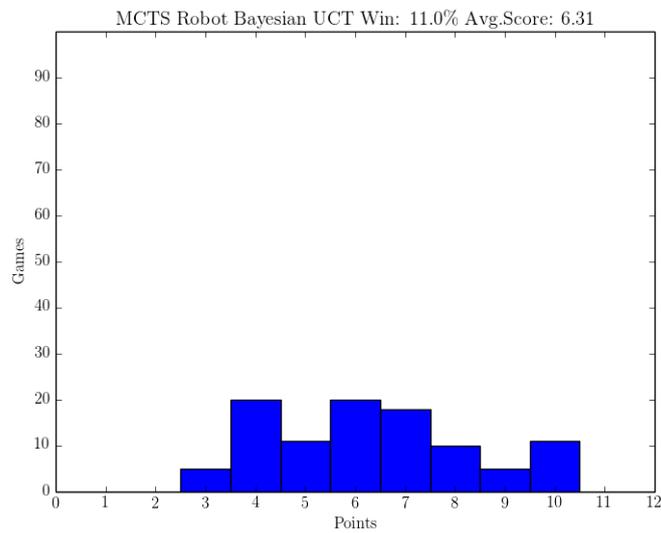
UCT against Bayesian UCT

In Figure 26, we see the results of the UCT against BAYESIAN UCT. The performance of the agents follow the one-method experiments above but with an increase in the total win rates of each. Specifically, the UCT winning rate from 4% rose to 9% and the BAYESIAN UCT rate rose from 7% to 11%.

We also see an increase to the average Victory Points of the agents, from 4.75 to 6.23 for UCT and from 5.29 to 6.31. The total winning rate for our agents is 20% against the JSettlers agents.



(a) UCT agent/7sec/120iteration



(b) Bayesian UCT agent/7sec/80iteration.

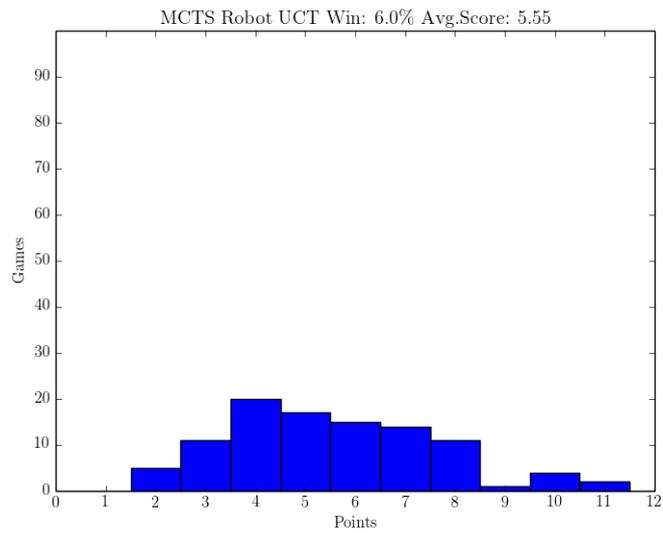
Figure 26: Results: UCT vs. BAYESIAN UCT.

UCT against VPI

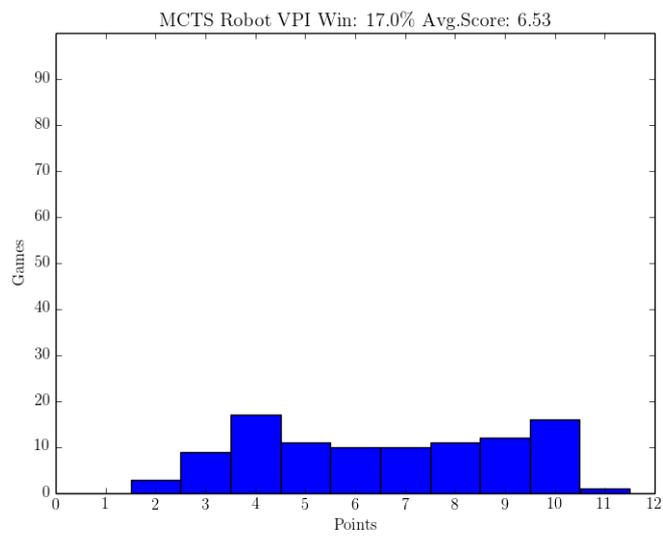
In this setting, we had two of our agents, one using the standard UCT approach and the other using the Value Of Perfect Information. The results are presented in Figure 27 and as with the above described setting of UCT against Bayesian UCT, we also see an increase in both agents performance in comparison to the one-agent experiments. This time, the UCT agent's winning rate only rose about 2% and the average score rose from 4.75 to 5.55. But there is a major increase in the performance of the VPI agent, rising his winning rate about 8%, from 9% to 17%, reaching to an average of 6.53 Victory Points for all games (from 6.03 in the one-method experiments). This point average even exceeds the average obtained by Szita et al.[4], where their agent with 1000 Simulations reached an average of 6.48 points.

We can see that the absence of one of *JSettlers* droids, benefits our agents in both cases, but the VPI agent, as we would expect to see and in the last experiment, is confirmed to be the strongest out of our three agents. Compared to the previous experiment we see that the VPI agent does not "allow" the UCT agent to take advantage of the absence of a *JSettlers* droid, keeping the winning rate of the latter low.

With the performance of our agents in this setting, we have a total of 23% winning rate for our agents against the *JSettlers* droids.



(a) UCT agent/7sec/120iteration

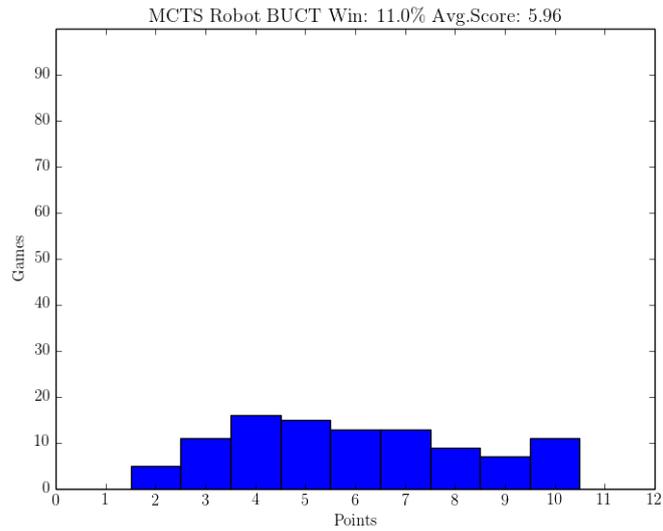


(b) VPI agent/7sec/12iteration.

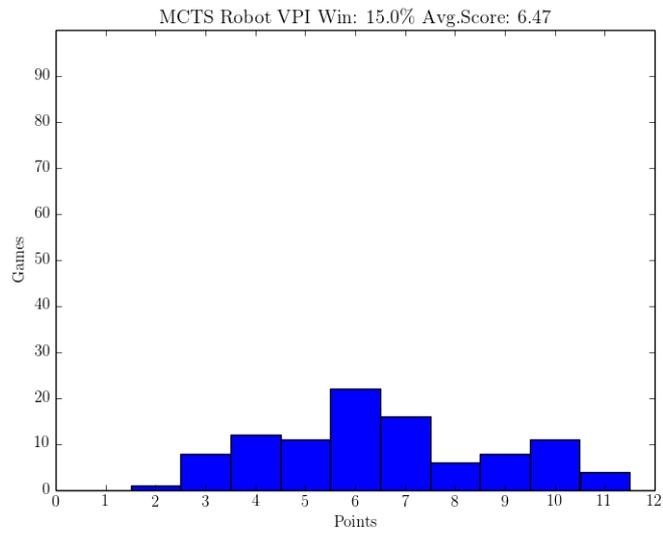
Figure 27: Results: UCT vs. VPI.

VPI against Bayesian UCT

In the last experiment, two of our agents compete against each other, one using the BAYESIAN UCT method and the other using the VPI method. Due to the fact that these two agents use 2 of our strongest implementations and based on the previous experimental results, we expect that the winning rate of the agents will rise. And our results in Figure 28 confirm our expectations. The BAYESIAN UCT agent, rose his winning rates from 7% to 11% compared to the one-method experiments but the winning rate is the same as with the BAYESIAN UCT-against-UCT experiment. On the other hand the VPI agent rose his winning rates from 9% of the one-method experiments to 15% but his winning rate is less than the VPI-against-UCT experiment, which was 17%. This result can be explained by the fact that in this experiment the VPI agent is competing against a stronger agent than the UCT case, and many times, his Victory was “stolen” by the BAYESIAN UCT agent, something that happened rarely against the UCT agent. So, when 2 of our stronger agents are competing against the *JSettlers* droids, they reach a total of 26% winning rate, that will definitely improve with the further development of the existing implementation. In Chapter 6.1, we mention our future plans for the development and the agent that we intend to implement in the next few months.



(a) Bayesian UCT agent/7sec/80iteration



(b) VPI agent/7sec/12iteration.

Figure 28: Results: BAYESIAN UCT vs. VPI.

6 Epilogue

We reached the last chapter of this thesis, which we dedicate in providing our main “research” ideas concerning our future work on this project (in Chapter 6.1) and in presenting our conclusions concerning the implementation of this newly founded algorithm in such a complex domain (in Chapter 6.2).

6.1 Future Work

MONTE CARLO TREE SEARCH, as the results of many implementations in many domains show, is a very promising method for creating an agent capable of defeating even an expert human player. There are many things that we are planning in order to make our agent stronger. These plans include many aspects of the existing implementation:

- **Experiments:**
 - We are currently performing experiments of our agents playing against random players in order to see their performance in such a setting.
 - We are planning to perform more experiments with the use of two of our agents that use the VALUE OF PERFECT INFORMATION method, against 2 JSettlers agents. We expect that in this setting the agents will take further advantage of the absence of one JSettlers agent and so, they’ll increase their winning rates further, exceeding even the BAYESIAN UCT against VPI setting.
- **MONTE CARLO TREE SEARCH:**

MONTE CARLO TREE SEARCH is a memory intensive algorithm. The more computational power, the better the results will be. One of our

future goals is to test the existing implementation in a stronger system and to log the results produced. Apart from this procedure, we are also have in mind to re-design some elements of the existing implementation in order to make our agent faster. We intend to rerun all the experiments after this modification, in order to study the immediate effect of performing more simulations, to the agent behavior.

- **Heuristic Strategies:**

Various heuristic strategies were created (and described in Chapter 4.4) in order to cope with situations of the game. These strategies, even though they are implemented based on the game and a player's logic, are yet to be tested in order to see how much they affect the behavior of the agent. So, we plan to test (and modify if needed) the created strategies to achieve even better results for our agent.

- **Heuristic cut-offs:**

Some heuristic strategies were implemented in order to reduce the state space and help the agent make better(?) decisions. The results of these strategies must be documented and if needed, "more guidance" must be injected to our agent to properly exclude beforehand any "bad" moves.

- **Negotiation:**

This is the domain in which we will really focus our research from now on. The "Settlers of Catan" gameplay is greatly based on the interaction between players and the negotiation in order to acquire resources. The first step is to enhance our NEGOTIATION scheme, using tools from *Machine Learning*, *Game Theory* and *Argumentation* techniques. The second step concerns the use of tools of *Natural Language Processing (NPL)*, in order to identify player strategies during the negotiations

phase (e.g. in online versions of the game). This plan is justified by the fact that there is a lot of “table-talk” during the game and the negotiations and one can and must identify the true intentions of his opponents.

6.2 Conclusions

Our intention was to implement a strong agent using the MONTE CARLO TREE SEARCH algorithm in the non-deterministic, partially observable, multi-player strategic board game “Settlers of Catan”. This task included the use and the enhancement of the *Tree Policy* of the MCTS algorithm, with methods such as the standard UCT approach, the BAYESIAN UCT, which is used for the first time in MCTS in “Settlers of Catan” and the VALUE OF PERFECT INFORMATION approach, which is used for the first time in conjunction with the MCTS algorithm. Apart from the MCTS implementation, in order for our agent to be able to respond to any situation of the game, we created various HEURISTIC STRATEGIES and implemented a simple NEGOTIATION scheme in order to trade with players. Our agent performs well against the existing *JSettlers* framework, which is considered to be one of the strongest computer implementations of the game. This framework is maintained and updated up to this day and we tested our agent against the most recent implementation. There is still room for improvements, especially in the VALUE OF PERFECT INFORMATION method, which displays the most potential, but also in a more efficient implementation of the algorithm. Our future work 6.1 includes all the modifications we are planning to implement, which -in our opinion- will improve the agent’s performance greatly. MONTE CARLO TREE SEARCH is a newly founded algorithm but the potential for creating an agent capable of winning even an expert human player exists,

and we think that we are in the right direction.

7 References

- [1] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intellig. and AI in Games*, pages 1–43.
- [2] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.
- [3] Jonathan Schaeffer and H. Jaap van den Herik. Games, computers, and artificial intelligence. *Artif. Intell.*, 134(1-2):1–7, 2002.
- [4] István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-Carlo Tree Search in Settlers of Catan. In *Proceedings of the 12th International Conference on Advances in Computer Games*, ACG’09, pages 21–32. Springer-Verlag, 2010.
- [5] Yoav Shoham, Rob Powers, and Trond Grenager. If multi-agent learning is the answer, what is the question? *Artificial Intelligence*, 171(7):365 – 377, 2007. Foundations of Multi-Agent Learning.
- [6] Peter Stone. Multiagent learning is not the answer. it is the question. *Artificial Intelligence*, 171:402–05, May 2007.
- [7] An overview of Catan Games. <http://www.catan.com/board-games>.
- [8] Game rules & almanac 3/4 players. http://www.catan.com/files/downloads/soc_rv_rules_091907.pdf.
- [9] Gerald Tesauro, V. T. Rajan, and Richard Segal. Bayesian Inference in Monte-Carlo Tree Search. 2012.

- [10] Richard Dearden, Nir Friedman, and David Andre. Model Based Bayesian Exploration. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, UAI'99, pages 150–159. Morgan Kaufmann Publishers Inc., 1999.
- [11] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1st edition, 1998.
- [12] J.M. Hammersley and D.C. Handscomb. *Monte Carlo Methods*. Methuen's monographs on applied probability and statistics. Methuen, 1964.
- [13] Sylvain Gelly and David Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artif. Intell.*, 175(11):1856–1875.
- [14] Cameron Browne. The Dangers of Random Playouts. *ICGA Journal*, 34(1):25–26, March 2011.
- [15] Rajeev Agrawal. *Sample mean based index policies with $O(\log n)$ regret for the multi-armed bandit problem.*, volume 27, pages 1054–1078. Applied Probability Trust, 1995.
- [16] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256.
- [17] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*. The AAAI Press.
- [18] Jeffrey Long, Nathan R. Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information monte carlo sampling in game tree search, 2010.

- [19] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293. Springer-Verlag, 2006.
- [20] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved Monte-Carlo Search. Technical Report 1, Univ. Tartu, 2006.
- [21] Richard Dearden, Nir Friedman, and Stuart Russell. Bayesian Q-Learning. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 761–768. American Association for Artificial Intelligence, 1998.
- [22] J. Wyatt. *Exploration and Inference in Learning from Reinforcement*. PhD thesis, University of Edinburgh, 1997.
- [23] Georgios Chalkiadakis and Craig Boutilier. Sequentially optimal repeated coalition formation under uncertainty. *Autonomous Agents and Multi-Agent Systems*, 24(3):441–484, 2012.
- [24] Konstantinos Babas, Georgios Chalkiadakis, and Evangelos Tripolitakis. You are what you consume: A bayesian method for personalized recommendations. In *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, pages 221–228. ACM, 2013.
- [25] W. T. L. Teacy, G. Chalkiadakis, A. Rogers, and N. R. Jennings. Sequential decision making with untrustworthy service providers. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems*, pages 755–762, May 2008.

- [26] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Rapport de recherche, INRIA, 2006.
- [27] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.
- [28] Markus Enzenberger and Martin MÅijller. Fuego – an open-source framework for board games and go engine based on monte-carlo tree search, 2009.
- [29] Tapani Raiko and Jaakko Peltonen. Application of uct search to the connection games of hex, y, *star, and renkula! In *Proc. of the Finnish Artificial Intelligence Conference (STeP 2008)*, pages 89–93, August 2008.
- [30] F. Teytaud and O. Teytaud. Creating an upper-confidence-tree program for havannah, 2009.
- [31] M.H.M. Winands, Y. Bjornsson, and J.-T. Saito. Monte carlo tree search in lines of action. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):239–250, Dec 2010.
- [32] Y. Sato, D. Takahashi, and Reijer Grimbergen. A shogi program based on monte-carlo tree search. *ICGA Journal*, 33:80–92, 2010.
- [33] Kazutomo Shibahara and Yoshiyuki Kotani. Combining final score with winning percentage by sigmoid function in monte-carlo simulations. In Philip Hingston and Luigi Barone, editors, *CIG*, pages 183–190. IEEE, 2008.

- [34] M. Pfeiffer. Reinforcement learning of strategies for settlers of catan. 2004.
- [35] Robert Shaun Thomas. *Real-time Decision Making for Adversarial Environments Using a Plan-based Heuristic*. PhD thesis, Northwestern University, 2003.
- [36] R. Fisher and W. Ury. *Getting to Yes*. Penguin Books.
- [37] Simon Lucas. Minimal one-page implementation of MCTS. <http://mcts.ai/code/java.html>.

A Code Changes in the JSettlers Framework

This appendix describes the changes needed in the original *JSettlers* framework in order to integrate our agent. For each change, we briefly describe its logic and for a better classification we describe the changes per class.

SOCDevCardSet.java:

We created new function `getAmountOld()` to return the amount of cards available to our agent for immediate play. A card is labeled OLD after a round has passed since the acquisition.

SOCServer.java:

- We had to create a new type of text message, which we set as `*START*` in order to start the game only with bots.

```
/*
 * code for starting a game with just robots
 */
else if (cmdTxtUC.startsWith("*START*")){

    this.handleSTARTGAME(c, new SOCStartGame(ga.getName()));

}
```

- The string `rname = MCTS_Robot_ + (i + 1);` was added for our player name, which is used in `SOCPlayerLocalRobotRunner.createAndStartRobotClientThread(rname, strSocketName, port);`
- We reduced the “rcount” of the robots counter by 4 in order to create 4 robots with MONTE CARLO TREE SEARCH.

- We added Monte Carlo Tree Search Robot Parameters, which are the same with other agents but we pass the number 2 in the strategy type in order to define the use of MCTS.

SOCFaceButton.java:

- We changed the number of robot faces (variable NUM_ROBOT_FACES) from two to three, in order to add a face for our own agent.

SOCRobotDM.java:

- We created a new constant (MCTS_STRATEGY=2) in order to differentiate our agent from the others.
- We added a new case “MCTS” in order to call our code for implementing Monte Carlo Tree Search.
- After the decision for our *Building Plan* for this round, we check to see if we have a *Road Building* card and if so we call the ROADBUILDINGSTRATEGY in order to choose two roads.

SOCRobotBrain.java:

- We added a new case SOCRobotDM.MCTS_STRATEGY to assign the appropriate faceid our agent.
- When the agent needs to discard, our DISCARDSTRATEGY is called in order to decide on the discard set.
- If we have a *Monopoly* card and the agent is ours, we decide if we’ll play the card based on the heuristic strategy MONOPOLYSTRATEGY.

- When the game state is “Waiting For Monopoly” (when a discovery card is played and we are waiting for the monopoly choice of the agent), an *if* statement was added in order to choose the resource from our heuristic strategy MONOPOLYSTRATEGY.
- Wherever, the NEGOTIATOR class of the original framework is used, there is an *if* statement allowing access only to the original agents and not ours.
- The *tradeWithGame(SOCPossiblePiece targetPiece)* was created, which is used to call our functions for trading with ports and with the bank.
- When a trade is proposed to our agent by another player, we added an *if* statement in order to call the *considerTrade* function.

B Class Diagrams

In this appendix, we present the Class Diagrams of our implementation and Class Diagrams concerning the integration of our implementation in the existing *JSettlers* framework. We begin with the analysis of our classes and then we continue with the integration diagrams.

B.1 Code Creation

The created packages are the MCTS, DISTRIBUTION and HEURISTICSTRATEGIES.

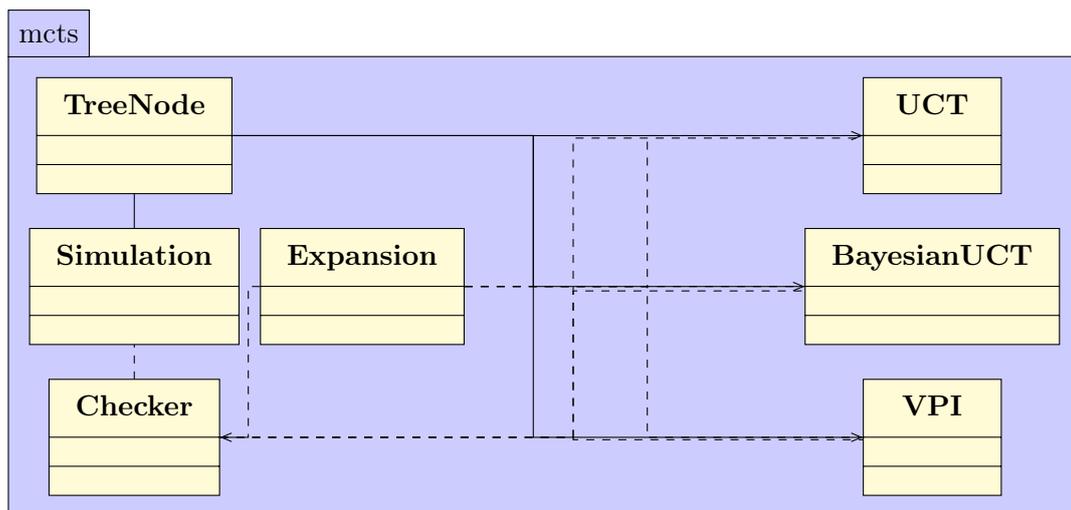


Figure 29: Class Diagram: MCTS Package

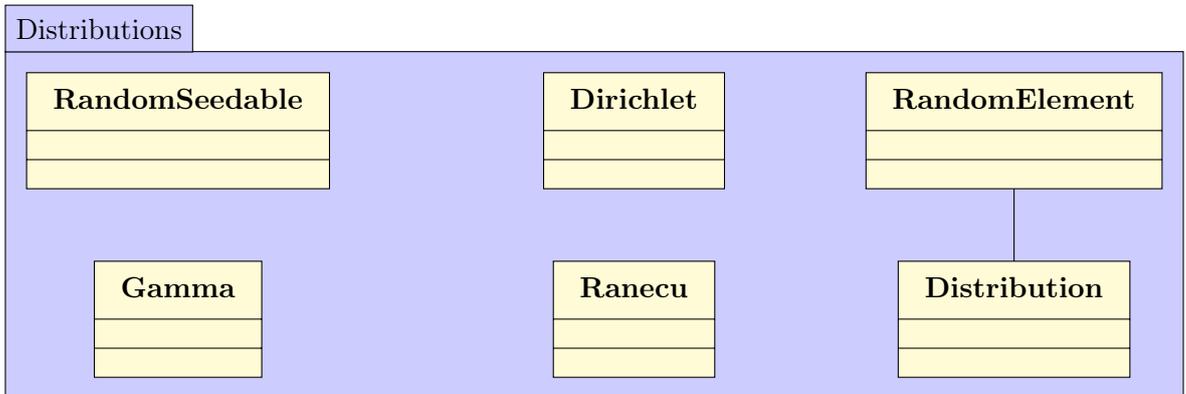


Figure 30: Class Diagram: Distribution Package

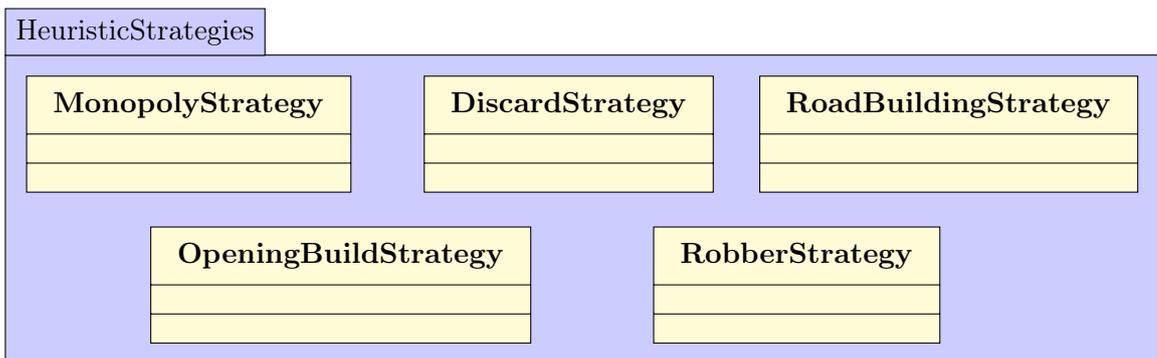


Figure 31: Class Diagram: Heuristic Strategies Package



Figure 32: Class Diagram: TreeNode

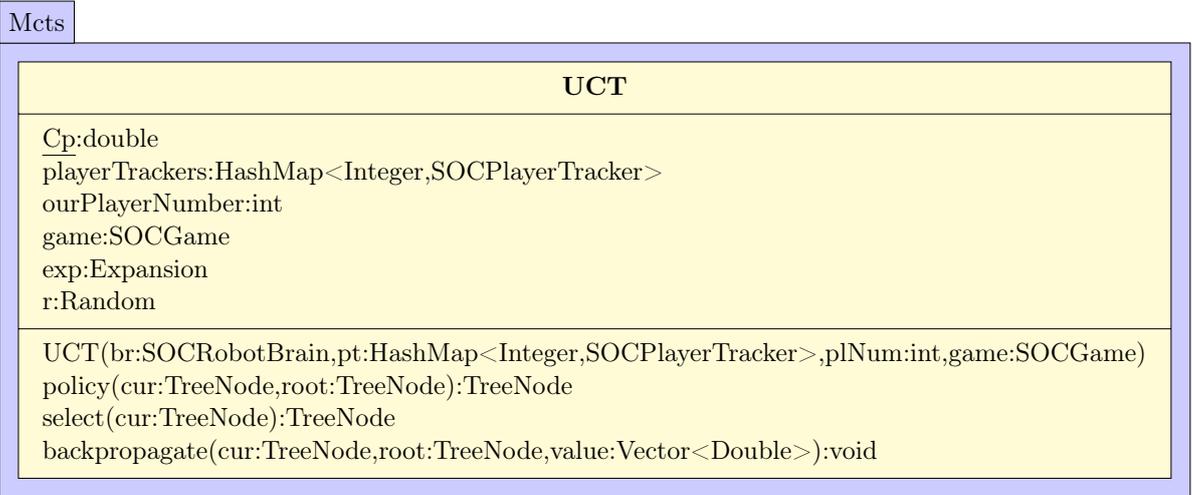


Figure 33: Class Diagram: UCT

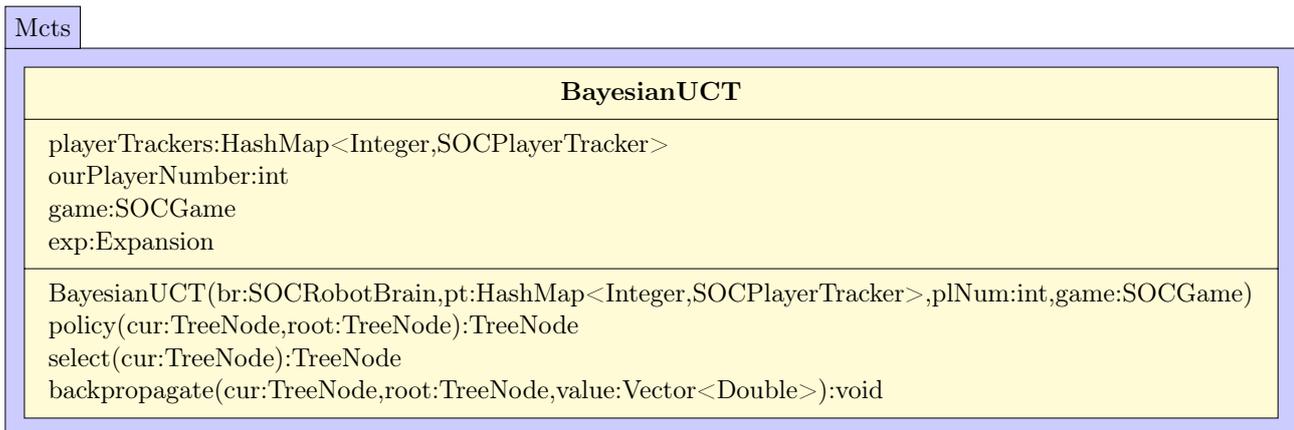


Figure 34: Class Diagram: Bayesian UCT

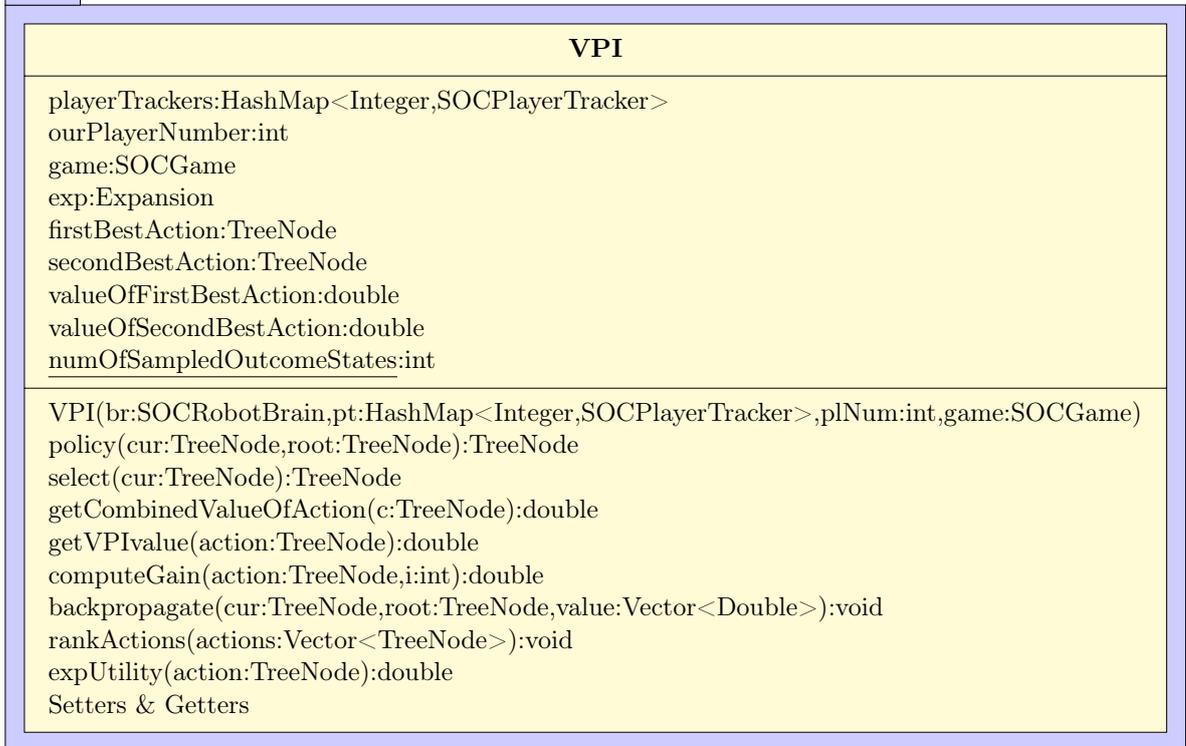


Figure 35: Class Diagram: VPI



Figure 36: Class Diagram: Expansion



Figure 37: Class Diagram: Simulation

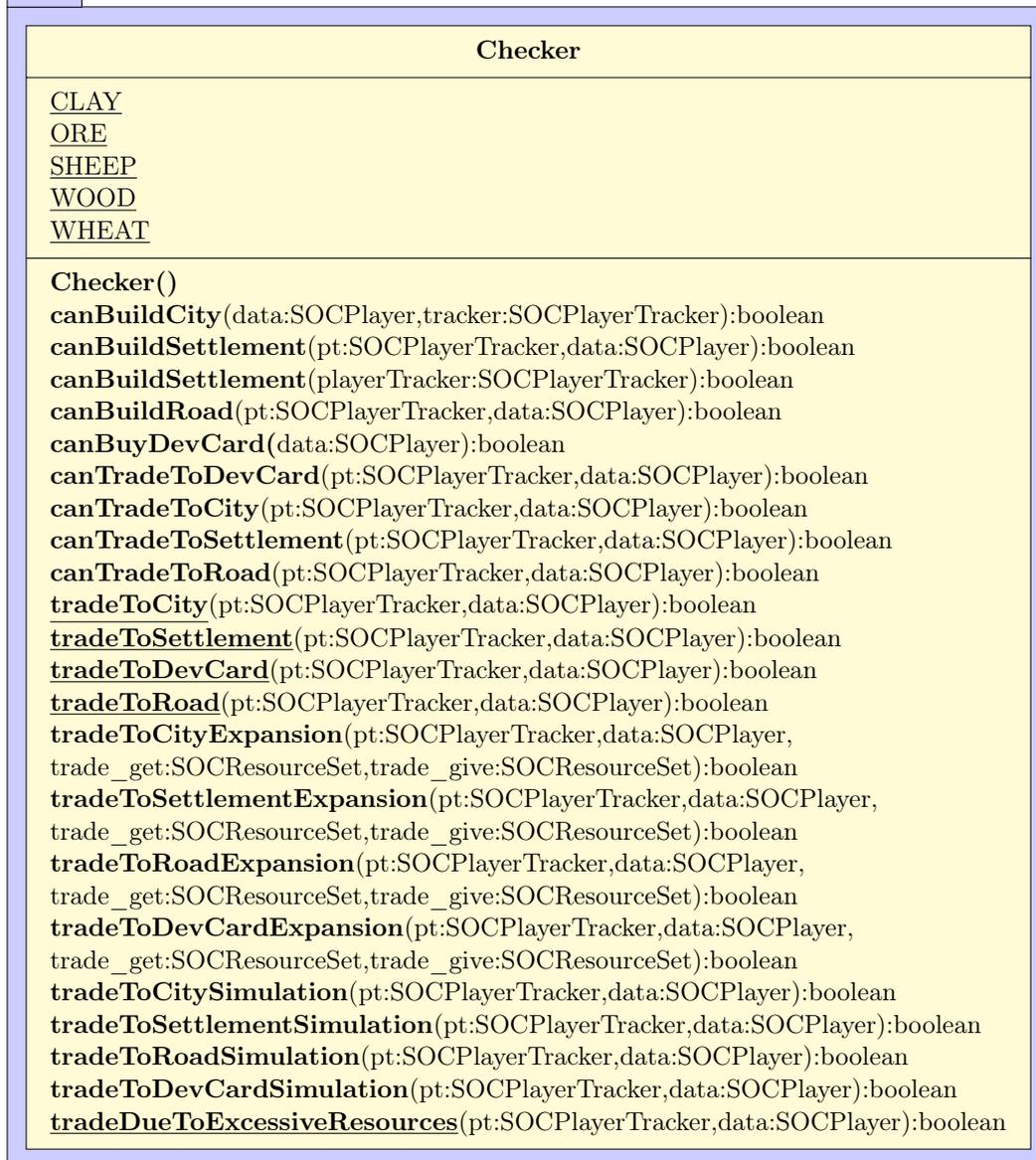


Figure 38: Class Diagram: Checker

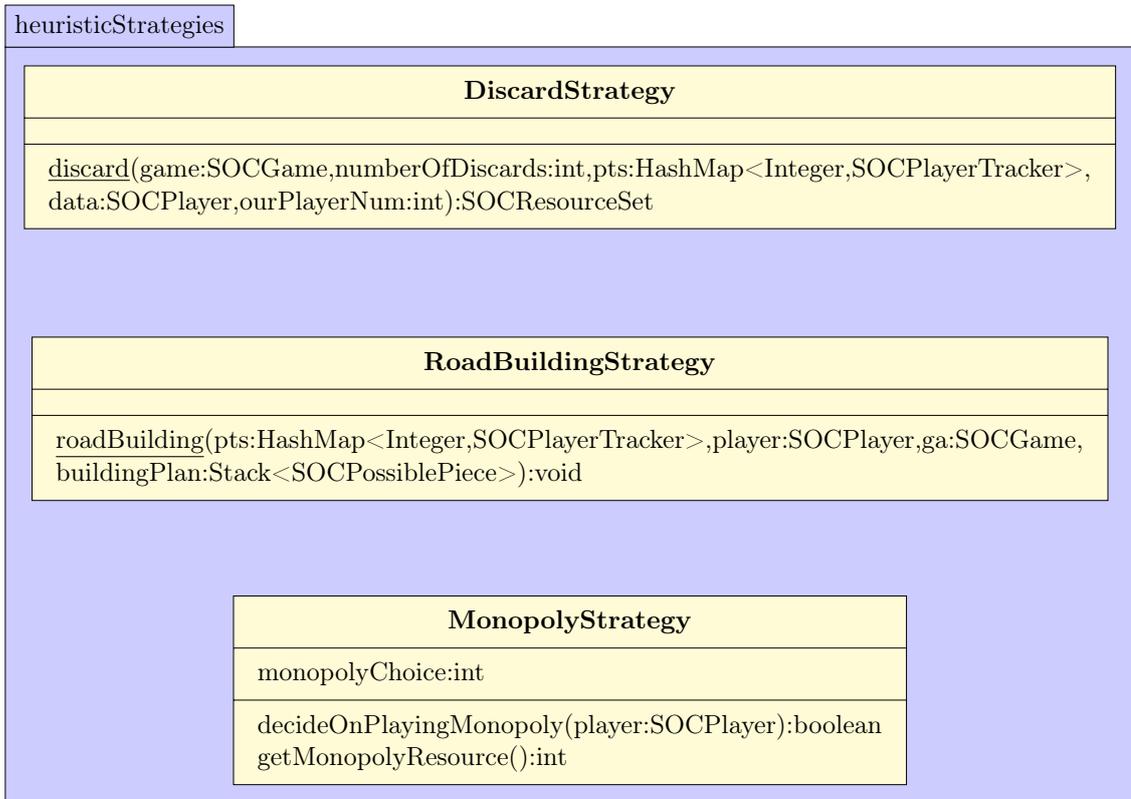


Figure 39: Class Diagram: Heuristic Strategies

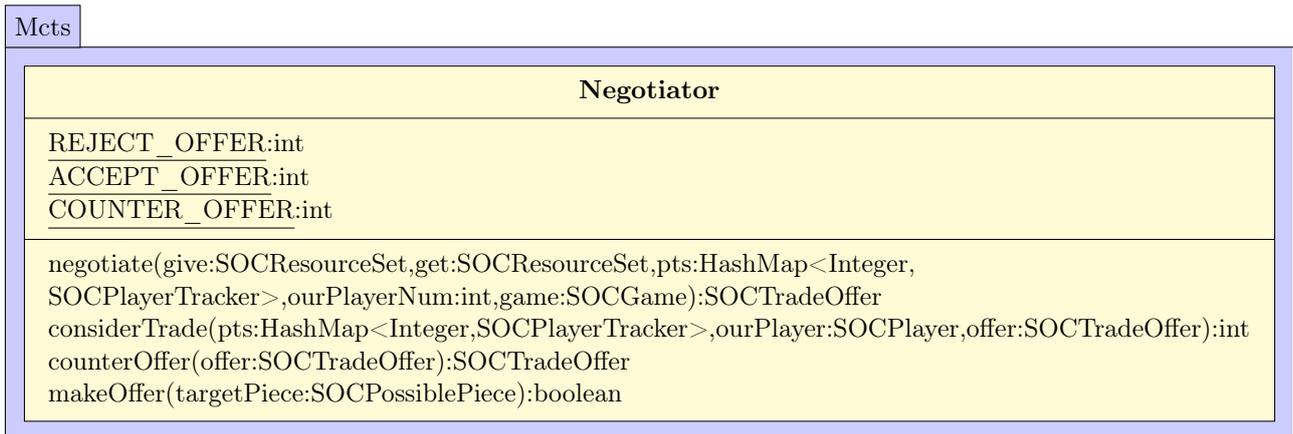


Figure 40: Class Diagram: Negotiator

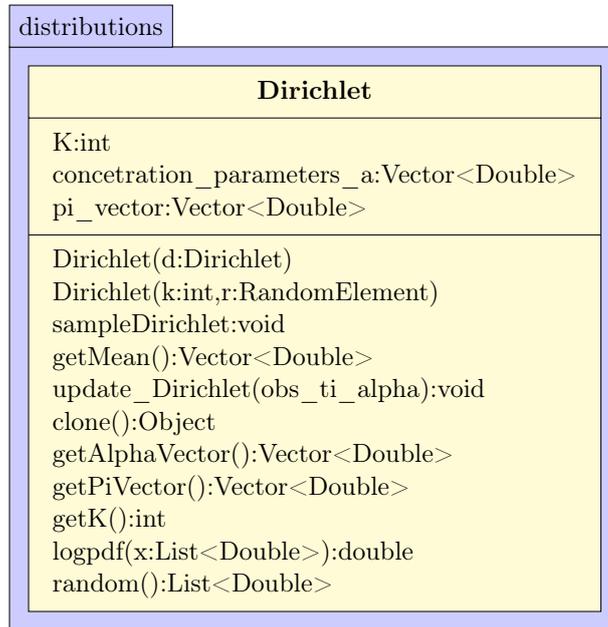


Figure 41: Class Diagram: Dirichlet

B.2 Code Integration

In order for our agent to work properly, we had to “connect” our implementation to the existing *JSettlers* framework. The two classes that are responsible for agent actions and planning are the SOCROBOTBRAIN and SOCROBOTDMM classes. SOCROBOTBRAIN is used to call the SOCROBOTDMM, to determine the future plans, but is also responsible for the communication with the GAME class in order to perform heuristic strategies as MONOPOLYSTRATEGY and others. Below, we present the two main diagrams for our code integration. One for the planning of the MCTS (Figure 42) and one for the integration of our heuristic strategies (Figure 43).

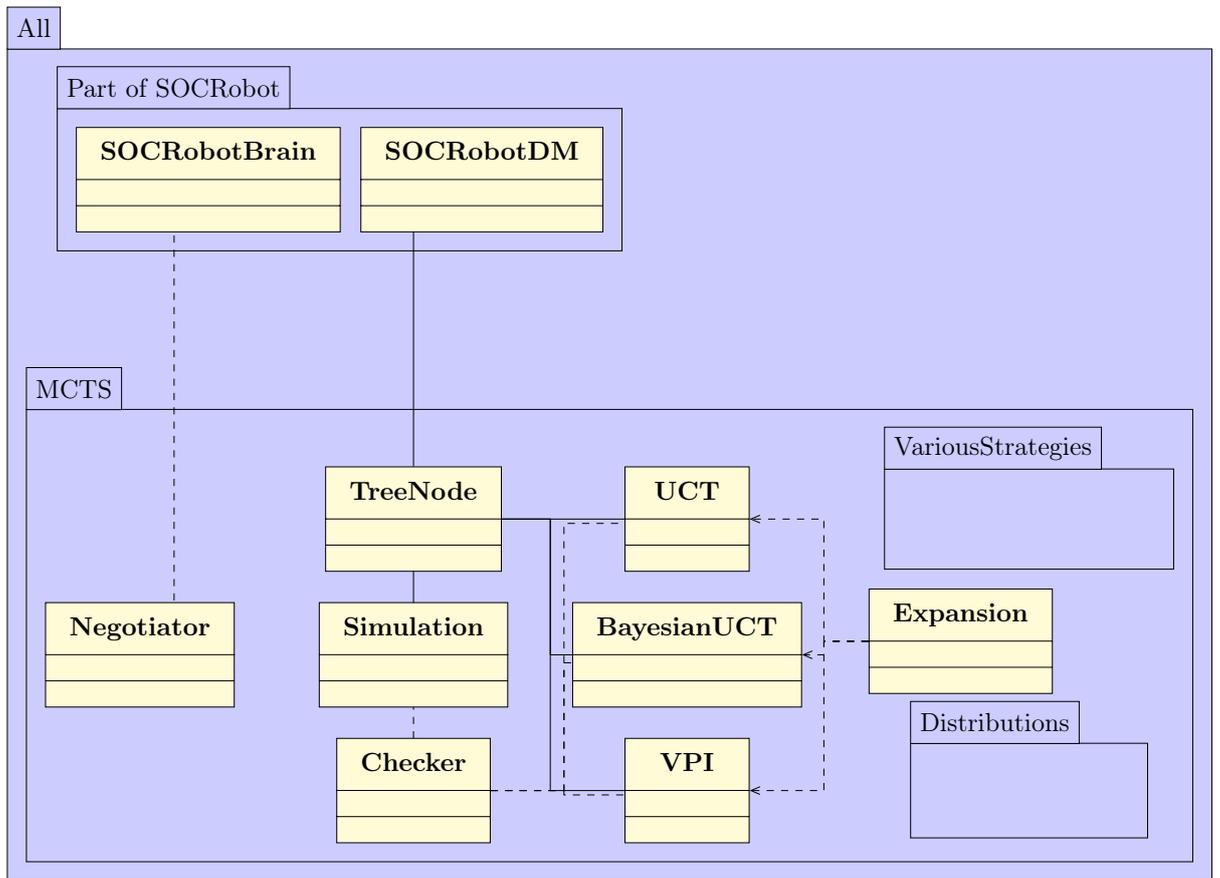


Figure 42: Class Diagram: Integration

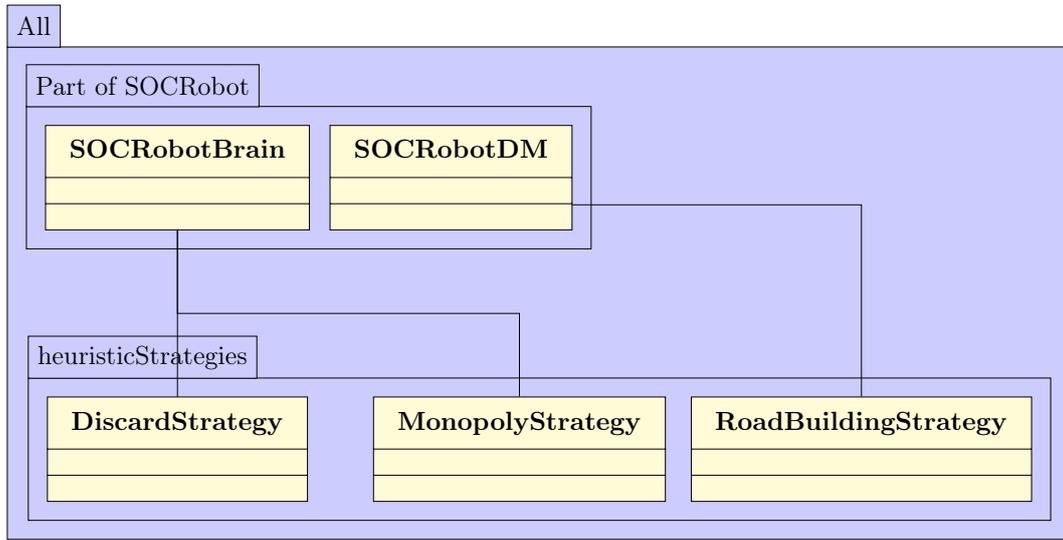


Figure 43: Class Diagram: Integration Heuristic