Optimizing UCT for Settlers of Catan

Gabriel Rubin*, Bruno Paz† and Felipe Meneguzzi‡
Pontifical Catholic University of Rio Grande do Sul
Computer Science Department
Porto Alegre, Brazil
*Email: *gdearrud@gmail.com, †bruno.paz@acad.pucrs.br, ‡felipe.meneguzzi@pucrs.br*

***Abstract*—Settlers of Catan is one of the main representatives of modern strategic board games and there are few autonomous agents available to play it due to its challenging features such as stochasticity, imperfect information, and 4-player structure. In this paper, we extend previous work on UCT search to develop an automated player for Settlers of Catan. Specifically, we develop a move pruning heuristic for this game and introduce the ability to trade with the other players using the UCT algorithm. We empirically compare our new player with a baseline agent for Settlers of Catan as well as the state of the art and show that our algorithm generates superior strategies while taking fewer samples of the game.**

***Keywords*-Artificial Intelligence; Monte Carlo Tree Search; Settlers of Catan;**

I. Introduction

Board games are of great interest to the Artificial Intelligence community. The study of classical games such as Chess and Checkers motivated great developments in the area, as many AI techniques have been developed to improve the performance of an AI in these classic games. While dealing well with traditional games, these techniques are often unsatisfactory for modern strategic games, commonly called *Eurogames*, because of the greater complexity of these games when compared to traditional board games [1]. Newly developed techniques [2] have significantly improved the performance of an AI in the classic Chinese game Go, bringing new possibilities for the development of competitive agents for *Eurogames*.

Settlers of Catan [3] is a good representative of the *Eurogame* archetype, with gameplay elements that make it challenging for traditional tree search algorithms, such as Minimax: imperfect information, randomly determined moves, more than 2 players and negotiation between players. Most autonomous agent players available for this game have game-specific heuristics and have a low win-rate against human players.

Previous work showed that Upper Confidence Bounds for Trees (UCT) [4], a variant of Monte Carlo tree search prominently used in games such as Go [5], yields a high win rate when applied to Settlers of Catan with simplified rules against agents from the JSettlers implementation of the game [6]. JSettlers [7] is an open-source Java implementation of Settlers of Catan that includes implementations of AI agents that are frequently used as benchmarks for new game playing strategies [6], [8]. However, the strategies generated by this previous UCT implementation do not negotiate with other players [6] and was only tested on Settlers of Catan with simplified rules [6]. Given the importance of trade as a gameplay element and the challenges of implementing effective players of the game with unmodified rules, we aim to develop UTC-based strategies capable of overcoming these limitations and surpassing existing techniques for playing Settlers of Catan.

Thus this paper provides three main contributions. First, we modify the base UCT algorithm to use domain knowledge and optimize it to improve its win rate in Settlers of Catan without relaxing the game rules. Second, we develop a method for trading with other players using UCT, by implementing a *trade-optimistic* search and compare it to our solution using the base UCT algorithm with no trading. Finally, we also show how the agent can be improved by using Ensemble UCT [9], a parallel variation of the base UCT algorithm that improves win rates and response time.

II. Background

In Settlers of Catan, each player controls a group of settlers who intend to colonize an island. The game is a race for points: the first player to obtain 10 victory points wins. To obtain victory points, players must gather resources and build their colonies on the island. In the section below, we explain the fundamental rules of the game in some detail. For a more detailed explanation of rules, we encourage the reader to check the official rules [3].

A. Game Rules

The game board, illustrated in Figure 1, represents the island and its ocean through hexagons. Every hexagon can be either one of six different types of terrain, or part of the ocean. Each terrain type produces its own type of resource: *fields* produce *grain*, *hills* produce *brick*, *mountains* produce *ore*, *forest* produces *lumber*, and *pasture* produces *wool*. There is one special terrain that don't produce resources: the *desert*. Finally, on top of each terrain, there is a token with a number between 2 and 12, representing the possible outcomes out of 2 six-sided dice.

1) Buildings: There are 3 types of buildings: settlements, cities and roads. Each building has a price in resources and can give players victory points: roads cost 1 *brick* and 1 *lumber* and give no victory points; Settlements cost 1 *brick*, 1 *lumber*, 1 *wool*, and 1 *grain*, and are worth

2159-6662/17/31.00 ©2017 IEEE
DOI 10.1109/SBGames.2017.00026

Figure 1. The board of Settlers of Catan. This image include player settlements, roads, cities and other elements.

1 victory point; Cities cost 3 *ores* and 2 *grains*, and are worth 2 victory points.

Players can build settlements or cities on the intersection between 3 terrain hexagons in order to obtain the resources produced by them. These resources can then be used to buy more buildings. Players can only place settlements and roads adjacent to another one of their roads, and cities can only be placed on top of one of their settlements.

2) Resource production: Resource production occurs in the beginning of each player's turn by rolling the 2 six-sided dice. Resources are then produced based on the outcome of the roll and the value depicted on top of the terrains on the board: any player with a settlement or city adjacent to a terrain with the same number as the dice roll, produces that terrain's resources, adding them to their hand. Settlements produce 1 resource per dice roll and cities produce 2 resources.

When a dice roll results in total of 7, all players that have more than 7 resources in their hand must discard half of them and move the robber. The robber is a special piece that blocks a terrain from producing during a dice roll. The robber starts the game at the *desert* terrain. Once the player rolls 7 and moves the robber to a terrain, that player can steal a random resource from other player whose settlement or city is adjacent to the robber's terrain.

3) Development cards and extra points: Players can also buy a card from the deck of development cards with resources. Each card costs 1 *ore*, 1 *wool*, and 1 *grain*. This deck have 5 types of cards in it, each one of these has different effects on the game: *Knight* cards can be used to move the robber; *Road Building* cards can be used to place 2 roads on the board; *Monopoly* cards can steal all resources from a specific type from all other players; *Year of Plenty* cards obtain any 2 resources; and *Victory Point* cards are worth 1 victory point at the end of the game.

There are 2 achievements that give victory points to the players during the game: The player with the longest continuous road gets 2 victory points, and the player with the largest army (largest number of knights cards used) also gets 2 victory points. These achievements are disputed during the match and cannot be shared between two players.

4) Trading: Players can trade resources with the game's bank or with other players. Trade rates are 4 to 1 with the bank and negotiable with other players. Players can only make trade offers during their turn. If a player decides to make no trade offer during its turn, then no other player can trade. Players can react to a trade offer by accepting it, declining it, or making a counter-offer.

There are ports in the game board that give players access to better trade rates with the bank. Players must place settlements or cities adjacent to these ports access points in order to use their trade rates. Each port have 2 access points. Ports are divided in 2 categories: *generic* ports have 3 to 1 rate for any resource, and *special* ports have a rate of 2 specific resources to 1. In the game board, there is a *special* port for each resource type and 4 *generic* ports, totalizing 9 ports.

B. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [10] is a modern algorithm that estimates the best move by iteratively constructing a search tree whose nodes represent the game states and edges represent the actions that lead from one state to another. Each node in the tree holds an estimated reward value $Q(v)$ and visit count $N(v)$.

At each iteration, the algorithm executes 4 steps, represented in Figure 2 [11]: Selection, Expansion, Simulation, and Backpropagation. The algorithm returns the estimated best move when a *computational budget* (i.e. time, iteration count or memory constraint) is reached.

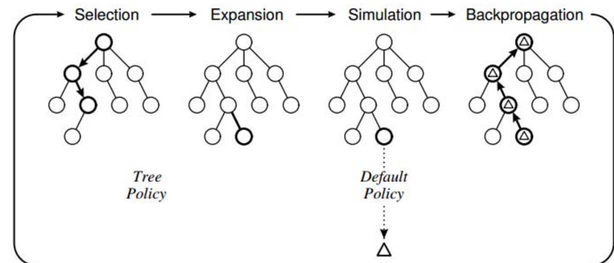

Figure 2. The 4 steps of the MCTS algorithm [11].

- The algorithm starts with the *selection* step, initially traversing the constructed search tree, using a tree policy π_T, this policy is dynamic and adapts as the tree grows. The algorithm traverses the tree from the root node using the π_T policy to select child nodes until it reaches a node with untried actions.
- The *expansion* step selects the node reached in the last step and choose an untried action at random to create a new child node.
- The *simulation* step uses a rollout policy π_R from the node created in the last step to select actions until it reaches a terminal state.
- The *backpropagation* step propagates the rewards obtained in the last step from the node created from the expansion step up to the root node of the search

tree, updating the visited nodes $Q(v)$ and $N(v)$ values.

C. Upper Confidence Bounds for Trees

UCT is a variation of MCTS that uses UCB1 [12], a method that solves the multi-armed bandit problem, as its tree policy π_T, balancing exploration and exploitation during the selection step. With this modification, UCT is shown to outperform the base MCTS algorithm on many games [11].

The action choice used in UCB1 is implemented using Equation 1, where $\overline{X}_j$ is the average reward obtained by choosing the action j, n_j is the number of times that action j was selected, n is number of visits the current node has been visited and C_p is a exploration value. The $\overline{X}_j$ term encourages exploitation, whereas the $C_p\sqrt{\frac{\ln n}{n_j}}$ term encourages exploration.

$$\arg\max_j \left(\overline{X}_j + C_p\sqrt{\frac{\ln n}{n_j}} \right) \quad (1)$$

The exploration value C_p can be adjusted to bias the selection towards exploration or exploitation. With C_p = 0, the exploration term is never taken into account and the selection is only based on exploitation. There is no predefined value for C_p, and it should be tuned for each implementation based on experimentation [11].

1) UCT Variations: Ensemble UCT [9] is a parallel variation of the UCT algorithm that can speed up the UCT search as well as improve its performance, with evidence that it can also outperform plain UCT in the game of Go [13]. This algorithm parallelizes the UCT search using *root parallelization* [13]: from a common root node, the algorithm creates p independent UCT trees, each in a separate thread, and expand them in parallel until a *computational budget* is reached. Then, the algorithm *merges* all root nodes and its children into a single tree. The nodes of the *merged* tree hold the total estimated reward $N(v)^E$ and total visit count $Q(v)^E$, calculated by Equation 2, where $Q(v)^i$ and $N(v)^i$ are the estimated reward and visit count of that node in tree i.

$$N(v)^E = \sum_{i=1}^{p} N(v)^i, \; and, \; Q(v)^E = \sum_{i=1}^{p} Q(v)^i \quad (2)$$

The best move is chosen from the root of the *merged* tree, using the same selection policy as UCT. Figure 3 illustrates the process done by this algorithm.

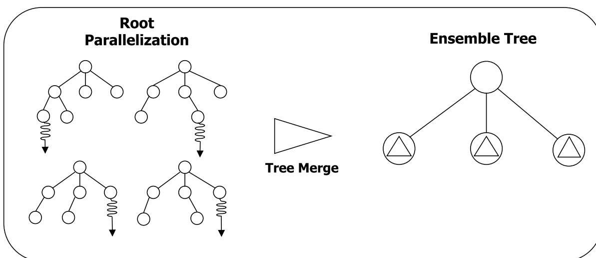

Figure 3. Ensemble UCT algorithm steps.

Sparse UCT [14] is a variation of the UCT algorithm that represents stochastic moves in the search tree as multiple nodes, where each node represents one possible outcome from taking that move. During the selection step of the algorithm, these nodes are chosen at random, and are also expanded at random during the expansion step of the algorithm, to simulate stochastic results.

III. Heuristics for UCT in Settlers of Catan

The use of domain knowledge was shown to improve the gameplay strength of UCT agents in many games [11]. In this section we describe the strategies we developed to improve the win rate of our UCT agent in the game of Settlers of Catan.

First, we describe our move pruning strategy that uses domain knowledge to reduce the algorithm's search space and compare it to a strategy developed in previous work. Afterwards, we introduce our trade-optimistic search method used by our agent to trade resources with other players.

A. Move pruning

The search space of Settlers of Catan is huge, with many legal moves per turn, and players can make multiple moves per turn. Previous work by Szita et al. [6] showed that an UCT agent can use domain knowledge to bias the tree search, decreasing the amount of rollouts spent by selecting *suboptimal* moves, increasing its playing strength.

1) Move pruning in previous work: In their work, Szita et al. introduce the concept of *virtual wins* to bias the tree search: at the start of the expansion step, their agent initialize both $Q(v)$ and $N(v)$ of the new child node to a predetermined *virtual* value. This value is set according to the move selected in the selection step: 20 for settlement-building, and 10 for city-building. Other moves don't receive *virutal wins*. By initializing both $Q(v)$ and $N(v)$ of these nodes to a greater value than other nodes in the tree, their agent explore them more often.

Their results show that their *virtual wins* heuristic increased their agents playing strength in a game of Settlers of Catan with rule changes [6], and that prioritize settlement-building and city-building is a viable strategy in Settlers of Catan. A possible explanation for the success of this strategy is that these moves give players more resource production and victory points, making them often preferable when compared to other moves available.

Nevertheless, we find in our tests that biasing the tree search with *virtual wins* is not enough: the agent usually spends too many resources in road-building and other actions, leaving few resources for settlement-building and city-building. Since these are the most expensive moves available, players must manage their resources carefully to be able to afford them. Spending resources on other moves can delay the opportunity of building cities and settlements, but accumulating resources in order to afford these moves can be risky, because of the discard rule, so waiting for the right moment to take these moves requires

some measure of luck. A player can play *safe* by always making these moves when they are affordable.

2) Our solution: In order to deal with this problem, we developed a move pruning heuristic that cuts all other moves whenever building a city or a settlement is possible, so that our agent take fewer risks and the penalty of losing half its resources impact it as little as possible. Our move pruning strategy also prioritizes cities over settlements since cities are worth two victory points and yields twice the resource production of a settlement, increasing the average resources gathered per turn, and consequently the amount of moves available to the agent per turn, as early as possible.

We use this same method in the π_R policy of the UCT algorithm to prune available actions, which are then selected at random. Random move selection increase rollout response time, allowing our agent to perform many rollouts at a given time, if a more complex heuristic were to be considered, rollout speed would be affected. In our experiments, we show that our move pruning strategy have better win rates compared to the *virtual wins* strategy when playing Settlers of Catan without rule changes.

3) Implementation: Our move pruning method is shown in Algorithm 1, along with its usage by the UCT algorithm during the tree's expansion step, represented by the EXPAND function on line 1, where: v is the node to be expanded, a is an action, $A(v)$ is a list of untried actions from state $s(v)$, v' is a child node, $A(v')$ is the list of available actions from state $s(v')$, and $a(v')$ is the action that led to state $s(v')$. Our method MOVEPRUNING is represented on line 8, where: s is a game state, A_a is a list of actions, and auxiliary functions GETPOSSIBLECITIES, GETPOSSIBLESETTLEMENTS, and GETOTHERPOSSIBLEACTIONS, return a list of actions available from state s.

Algorithm 1 Move pruning method

```
1:  function EXPAND( v ) returns a node
2:      choose a ∈ untried actions from A(v)
3:      add a new child v' to v
4:          with s(v') = APPLYACTION( s(v) , a )
5:          and a(v') = a
6:          and A(v') = MOVEPRUNING( s(v') )
7:      return v'

8:  function MOVEPRUNING( s ) returns a list of
    actions
9:      A = empty
10:     A ← GETPOSSIBLECITIES( s )
11:     if A is not empty then
12:         return A
13:     A ← GETPOSSIBLESETTLEMENTS( s )
14:     if A is not empty then
15:         return A
16:     return GETOTHERPOSSIBLEACTIONS( s )
```

B. Trade-optimistic search

Previous implementations of UCT for Settlers of Catan did not consider trading with other players [6], a gameplay element that can boost the playing capabilities of an agent in this game. Trading with other players in Settlers of Catan is a challenging problem since it can benefit an opposing player, and estimating the impact of a trade can be difficult without knowing the opponents resources. However by not trying to trade at all, a player could be *starved* of resources for many turns, lowering its chances of remaining competitive in the game. Our solution deals with two trading cases separately: reacting to other players trade offers, and making trade offers for other players.

Our agent can react to trade offers with a regular UCT search with two options from the root node: accept or decline the trade offer, without making counter-offers. Finding the counter-offer that is most likely to be accepted by our opponent while being beneficial for our agent is difficult, since we don't know exactly what resources our opponent have in its hand, so we decided to leave this feature out of our trading strategy. We find that this approach is acceptable for reacting to trade offers.

1) Trade offering through optimistic search: We propose an *optimistic* method for creating trade offers that uses the UCT search to estimate what trades are most beneficial to our agent. After rolling the dices, our agent simulates the ability to afford any available move by trading spare resources with other players: our agent labels moves that are not currently affordable, but could be afforded via trading, as *trade-optimistic* moves, and consider them as affordable moves during the UCT search. If our agent can't afford a move and this move is not affordable via trading, it is not considered during the UCT search. If the UCT search selects a *trade-optimistic* move as the best move to be taken, our agent make trade offers to other players in order to afford that move.

In this method, our agent only makes trade offers with 1 to 1 resource rate. Trades with this rate are more likely to be accepted by other players, and even if not all trades are successful, our agent is still *closer* to afford the chosen *trade-optimistic* move. These trade offers are directed to all other players, to increase our agent's chances of obtaining all the resources needed to afford the selected *trade-optimistic* move. Therefore, our agents consider that a move is affordable via trading if it has the same amount of resources in its hand than the amount of resources needed to afford that move, even if they are not of the correct type. For example, if our agent have 2 resources in its hand out of the 5 needed build a city (i.e. 2 *ores*), it would need another 3 spare resources, from any type, in order to consider city-building as a *trade-optimistic* move: to get the remaining 3 resources needed to build a city (i.e. 1 *ore* and 2 *grains*), our agent needs to make 3 trades.

2) Implementation: Algorithm 2 shows our trade-optimistic search method in pseudocode, as well as how it utilizes the UCT search. Before starting the UCT search, on line 27, our agent adds the *trade-optimistic* moves to the list of moves available from the root node of the UCT tree.

These *trade-optimistic* moves are obtained by calling function GETTRADEOPTIMISTICMOVES, where: v is a node, p is the current player, $R(p)$ are the resources of player p, and $R(a)$ is the price of a. Equation $\sum R(p) >= \sum R(a)$ is used on line 38 of this function, to check if an action a is affordable via trading, by comparing the number of resources in the current player's hand with the number of resources needed in order to afford a, without considering resource types.

With the updated list of moves available, the UCT search is performed. If the UCT search selects a *trade-optimistic* move as the best move to be taken, our agent puts all trades needed in order to afford the chosen move in a trade queue, on line 10. These trades are obtained by calling function GETTRADESNEEDED, where: p is the current player, a is a *trade-optimistic* move, Δ_R is a list with the resources player p needs in order to afford a, $R(p)$ are the resources of player p, $R(a)$ is the price of a, A_T is a list of trade offers, and a_T is a trade offer action.

Our agent tries all trades in the queue. Even if one trade fails, it continues to try trades from the queue until the queue is empty, as shown on line 6. After all trade offers were made, our agent do another UCT search, without any trade-optimistic moves, by calling function TRADEUCT with parameter *opt* = $False$. This second UCT search is needed since it don't consider trade-optimistic moves, and if any trades were successful, the resources of our agent will have changed, invalidating the results of the first UCT search. Our agent ignores any counter-offers from other players, so that the trade queue strategy is preserved. Trading with other players is tried only once per turn, to avoid trading loops: this control is made with the local flag *canTrade*.

Algorithm 2 Trade-optimistic search

```
1: var queue = empty
2: var canTrade = False

3: function GETBESTMOVE( s_0 ) returns an action
4:     if queue is not empty then
5:         canTrade ← size(queue) <= 1
6:         return DEQUEUE(queue)
7:     a ← TRADEUCT( s_0 , canTrade )
8:     if a is trade-optimistic then
9:         trades ← GETTRADESNEEDED(p(s_0), a)
10:        ENQUEUE(queue, trades)
11:        return DEQUEUE(queue)
12:    else
13:        canTrade ← True
14:        return a

15: function GETTRADESNEEDED( p, a ) returns a list of actions
16:    A_T = empty
17:    Δ_R ← GETMISSINGRESOURCES(R(p), R(a))
18:    for each resource r ∈ needed resources Δ_r
19:         create trade action a_T
20:              with give(a_T) = random resource from R(p)
21:              and get(a_T) = random resource from Δ_R
22:         A_T ← A_T + a_T
23:    return A_T

24: function TRADEUCT( s_0 , opt ) returns an action
25:    create root node v_0 with state s_0
26:    if opt is True then
27:        A_opt ← GETTRADEOPTIMISTICMOVES(v_0)
28:        A(v_0) ← A(v_0) + A_opt
29:    while within computational budget do
30:        v_n ← TREEPOLICY( v_0 )
31:        △ ← SIMULATIONPOLICY( s(v_n) )
32:        BACKUP( v_n , △ )
33:    return a(BESTCHILD(v_0))

34: function GETTRADEOPTIMISTICMOVES( v ) returns a list of actions
35:    p ← GETCURRENTPLAYER(s(v))
36:    A_opt = empty
37:    for each action a ∈ legal actions A_L from s(v)
38:        if p can't afford a and ∑R(p) >= ∑R(a)
39:            A_opt ← A_opt + a
40:    return A_opt
```

IV. IMPLEMENTATION AND EXPERIMENTS

Our implementation consists of: a client for the JSettlers server; our base UCT agent implementation and its variations; and our own Settlers of Catan simulator that is used to simulate the game during UCT rollouts more efficiently than the JSettlers server implementation. We implemented all of our algorithms using Python 2.7 and designed the code to be easy to modify and adapt for new strategies and experiments without sacrificing performance.

In the following sections, we detail our implementation, the experiments we carried out and their results. We first explain how we implemented UCT so that its tree correctly represent the possible states in Settlers of Catan. We also include technical details of our implementation, including limitations and possible upgrades. Finally, we detail how we developed our experiments and compare results obtained in each experiment.

A. UCT Agent implementation

In this section, we describe the details of our base UCT agent implementation for Settlers of Catan, and how we designed it to deal with the game's *imperfect information* and *stochastic* moves, so it can play Settlers of Catan without any modification or simplification of its rules. Our UCT agent was designed to be a standalone agent, capable of playing Settlers of Catan matches in the JSettlers server against humans or other agents through our JSettlers client.

In order to keep memory usage as low as possible during the construction of the search tree, our agent serializes the game state of new nodes added to the tree, and deserializes them when necessary. Since the repeated seri-

alization/deserialization process can take some processing time, it also slows down the rollout response time. In order to deal with this, we divided game states into two categories: states that are deserialized when accessed and states that are kept deserialized. A threshold value Δ_S controls states categories, if a node visit count is greater than Δ_S, its game state is kept deserialized, otherwise, it is kept serialized. We found in our tests that Δ_S = 20 is a value that balance memory usage reduction and extra processing time. With this addition, our agent memory utilization dropped considerably, while maintaining rollout response time.

Our agent keeps track of resources obtained by opponents during the game, until one of the following events occurs to an opponent: it discards half resources; it steals a resource; or it has a resource stolen. In these cases, our agent labels that player's resources as unknown. We use this information to deal with *imperfect information* before the UCT search: any unknown resource an opponent has in its hand is determined at random at the root node: unknown cards are given a random value. Since this process of randomly guessing unknown cards can affect the quality of the estimate made by the algorithm, we also considered the possibility of using the Sparse UCT approach of adding all possible resource combinations for unknown resources to the tree. Nevertheless, given the stochastic nature of the sampling performed by UCT, adding all possible resources to the tree will increase the tree branching factor, and our agent would consequently need more rollouts to make strategic decisions.

We implemented Sparse UCT [14]to represent the stochastic results of dice rolls, so that all possible dice roll results are taken into account during the construction and exploration of the search tree. Instead of using a uniform random function to select dice results or expand nodes from a dice roll, we use the same simulation of dice roll used in our simulator, to correctly simulate dice results. The only downside of implementing Sparce UCT in our search tree is that each dice roll move spawns multiple nodes, which increase the search tree's branching factor and, consequently, the amount of rollouts needed to make strategic decisions.

B. Client and Simulator implementation

In order to test our agents with the JSettlers agent, we implemented a client that is able to connect to the JSettlers server and start a game with three other JSettlers agents. Figure 4 shows the interface of the JSettlers server during a match.

Our client sends and receives messages from the JSettlers server: it updates the current game state with data received from server, and sends our agent's actions back to the server. The JSettlers server don't send all game information to our client, *imperfect information*(i.e. other player's resources) are kept hidden in its messages.

We also implemented an very efficient Settlers of Catan simulator in our client to perform the UCT rollouts. It represents game states and simulates the game through

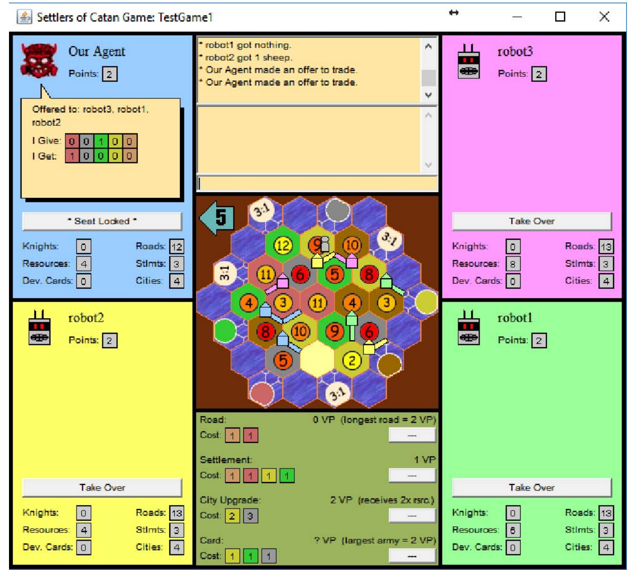

Figure 4. Screenshot of a match between our agent playing a game against 3 JSettlers agents in a JSettlers server.

an action system, each action represents a move in the game and its used by the simulator to modify game states. Games are simulated by selecting legal actions at random from a given state and applying them to that state, repeating this process until a terminal state is reached. Our agent utilizes this simulation method to perform UCT rollouts, using our move pruning method to prune legal moves.

Our simulator is able to simulate approximately 65 games per second in a modern PC, which in our experiments was an Intel i7-4702MQ CPU, with 4 cores at 2.2Ghz, and 16 Gigabytes of RAM. Our Ensemble UCT agent with 1,000 rollouts is almost as fast as the JSettlers agent. However, we found in our tests that running 10,000 rollouts per UCT search can be very slow, especially without the Ensemble UCT parallelization. Therefore, we decided to limit rollouts to 10,000 for our agent.

C. Experiments and results

We tested various different agent configurations in games where our agent plays against three JSettlers agents. We carried experiments on the following agent configurations:

- *PlainUCT*: Default UCT algorithm without any heuristic.
- *VW-UCT*: UCT algorithm with virtual wins, like described by [6].
- *MP-UCT*: UCT using our move pruning heuristic.
- *MP-EnsembleUCT*: Ensemble UCT using our move pruning heuristic. This agent runs n rollouts divided to a number of parallel UCT trees p, where each tree runs its share of the total rollout count.
- *MPT-EnsembleUCT*: This agent is the same as the *MP-EnsembleUCT*, but its capable of trading via our *trade-optimistic* search method.

Since previous work [6] shows that seating order can introduce an unknown bias to the agents performance, we randomized seating order for all tests to mitigate any seating bias.

1) Pruning heuristics comparison: First, we compared the win rate of *PlainUCT*, *VW-UCT*, and *MP-UCT*, using 1,000 rollouts with: 0; 0.25; 0.5; 0.75; and 1.0 as the exploration value C_p. *PlainUCT* use no heuristic to prune or select moves, and serves as a baseline for the other two agents. Figure 5 illustrates the results of this experiment with the error bars showing the standard deviation of win rate over 100 matches against three JSettlers agents.

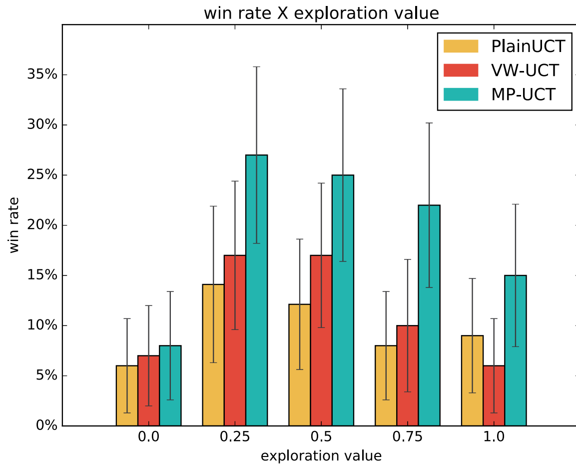

Figure 5. Comparison between agent win rates in games against three JSettlers agents, with varying exploration values.

Our results show that all three agents benefit from more exploitation, with exploration values between 0 and 0.25. For the following experiments, we used 0.25 exploration value for all agents, since both *MP-EnsembleUCT* and *MPT-EnsembleUCT* are based on our *MP-UCT*, which had better win rates with $C_p = 0.25$.

These results also show that, compared to the *virtual wins* heuristic used by the *VW-UCT* agent, our *MP-UCT* agent can achieve superior win rates in games against 3 J-Settlers agents: with $C_p = 0.25$, our agent have about 10% more wins than the *VW-UCT* agent with the same configurations. With about 26% win rate, the *MP-UCT* agent with 1,000 rollouts per search has roughly the same playing strength of a JSettlers agent, since at this win rate, it has won about as much games as its three JSettlers opponents.

The performance of the *VW-UCT* agent in our results is slightly different than that observed in previous work [6], with about 10% less wins. We believe that this difference is due to the different settings in our test: their tests were made on Settlers of Catan with rule changes (i.e. no *imperfect information*), while our tests were conducted on games with complete rules. There are also implementation differences that might led to slightly different results.

2) MP-UCT variations comparison: The following experiments focus on the different variations of our *MP-UCT* agent: *MP-EnsembleUCT* and *MPT-EnsembleUCT*, capable of trading. We compared these agents performance in matches against three JSettlers agents, and the results of these matches are summarized in Table I: with win rate expressed in percent $\pm$ at a 95% confidence interval. We compared the three agents with 1,000 rollouts and 10,000 rollouts. The first column of Table I shows what agent was tested, followed the number of rollouts used by that agent, and the win rate of that agent in our experiments. For the ensemble agents, we used parallel UCT count $p = 10$, so it runs 10 parallel UCT searches of 100 rollouts for 1,000 rollouts, and 10 parallel UCT searches of 1,000 rollouts for 10,000 rollouts, to make the ensemble tree. All agents were tested using $C_p = 0.25$.

Table I shows that *MP-EnsembleUCT* agent has about 7% higher win rates than the base *MP-UCT* agent at 1,000 rollouts, and roughly 3% higher win rates at 10,000 rollout count. We believe that this advantage shows that by combining various independent UCT searches, each with different trajectories through the search tree, the ensemble tree have less variance then a single UCT tree with only one set of trajectories [9]. We believe that this difference is more pronounced with fewer rollouts, and as rollout count rises, the trajectories of the separate search trees tend to converge to a similar path. The major advantage of *MP-EnsembleUCT* comes with the agent's response time: *MP-EnsembleUCT* with $p = 10$ and 10,000 rollouts was about 3 times faster than the base *MP-UCT* agent with 10,000 rollouts in our test machine. Precise speed advantages were not measured as they can vary from one machine to another.

Our results also show that with 10,000 rollouts, these three agents are clearly superior to the JSettlers agent. Our trading agent *MPT-EnsembleUCT*, in particular, have an expressive superiority, winning 58.2% of all games played with 10,000 rollouts. Even at a low rollout count, with 1,000 rollouts, this agent was able to win 40% of all games, a slightly better result than the 38.4% win rate of the base *MP-UCT* agent with 10,000 rollouts. This shows that our *trade-optimistic* search method did boost the playing strength of the *MP-UCT* agent considerably. It should be noted that against players that don't consider trading, the *MPT-EnsembleUCT* agent's playing strength will be the same as the *MP-EnsembleUCT* agent, since the trading capability is the only difference between both agents.

Finally, Figure 6 illustrates the win-rates of every agent configuration in games against three JSettlers agents, using 1,000 rollouts per search and exploration value $C_p = 0.25$. In this comparison, it becomes clear that our heuristics can greatly improve the base UCT agent playing strength, even at the low rollout count of 1,000, specially *MPT-EnsembleUCT*, that has a great advantage over the others, since it is the only variation that considers trading.

V. Conclusions and Future Work

We developed two domain-dependent heuristics, the move pruning, that uses domain knowledge to prune the

Table I
AGENT WIN RATE COMPARISON IN GAMES AGAINST THREE JSETTLERS AGENTS.

Agent	UCT rollouts	Win Rate
MP-UCT	$1,000$	$26.1\% \pm 7.14\%$
MP-EnsembleUCT	$1,000$	$32.8\% \pm 6.31\%$
MPT-EnsembleUCT	$1,000$	$40.0\% \pm 6.81\%$
MP-UCT	$10,000$	$38.4\% \pm 8.64\%$
MP-EnsembleUCT	$10,000$	$41.3\% \pm 7.66\%$
MPT-EnsembleUCT	$10,000$	$58.2\% \pm 7.09\%$

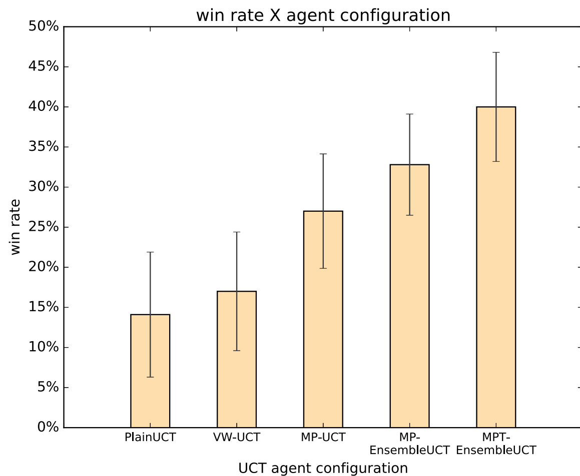

Figure 6. Comparison between agent win rates in games against three JSettlers agents, with 1,000 rollouts per search.

game tree, and the *trade-optimistic* search that utilizes the UCT algorithm in order to trade in Catan. These heuristics provide substantial improvements to MCTS-based methods for the Settlers of Catan Game without rule changes.

Previous work found that UCT could effectively play Settlers of Catan with rule changes (i.e. with no *imperfect information*) and that heuristic strategies (*virtual wins*), could improve an UCT agent performance in-game. Our results show that in games without rule simplifications (i.e. with imperfect information: unknown opponents resource cards and development cards), our own *move pruning* heuristic strategy outperforms the *virtual wins* strategy.

However, our move pruning strategy is very restrictive and there are cases were it leads to *suboptimal* moves, especially near the end of the game. If the agent is competing for the largest road with another player and both are tied with eight or more victory points, this strategy will favor cities and settlements over roads, leading our agent to lose the largest road points. We intend to develop a less rigid heuristic in the future as well as to increase the number of games that our simulator is able to execute per second, so that our agent can execute more rollouts per UCT search. We also intent to find the exact exploration value that maximizes our agent's win rate. In our tests, we set the exploration value $C_p = 0.25$, but the real *optimum* value could be different. Our results show that this value is between 0.0 and 0.25.

In our experiments, the Ensemble UCT agent had slightly better win rates compared to the regular UCT agent, while having better response times. Because of this, we find that this version of UCT is better suited for Settlers of Catan than the base UCT algorithm. In future work, we intend to investigate how to reach an optimal configuration of this algorithm for this game, such as the number of parallel trees p for 1,000 and 10,000 rollouts.

Finally, our results show that our *trade-optimistic* search heuristic increases the competitive strength of our agent against JSettlers agents, increasing our agents win rate and average points per game. These results show that an effective trading strategy can have significant impact in an agent gameplay capabilities and is fundamental for the game of Settlers of Catan. There are features, such as making counter-offers, that could be implemented into our heuristic, and we intend to further develop this heuristic in the future. We also intend to investigate the performance of this trading strategy against other agents and human players in future work.

REFERENCES

[1] D. Robilliard and C. Fonlupt, *Towards Human-Competitive Game Playing for Complex Board Games with Genetic Programming*. Cham: Springer International Publishing, 2016, pp. 123–135.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[3] I. Mayfair Games, "Catan - 5th edition, game rules and almanac," 2015. [Online]. Available: http://www.catan.com/files/downloads/catan_-5th_ed_rules_eng_150303.pdf

[4] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Proceedings of the 17th European Conference on Machine Learning*, ser. ECML'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 282–293.

[5] S. Gelly and Y. Wang, "Exploration exploitation in Go: UCT for Monte-Carlo Go," in *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, Canada, Dec. 2006.

[6] I. Szita, G. Chaslot, and P. Spronck, "Monte-carlo tree search in settlers of catan," in *Proceedings of the 12th International Conference on Advances in Computer Games*, ser. ACG'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 21–32.

[7] R. S. Thomas, "Real-time decision making for adversarial environments using a plan-based heuristic," Ph.D. dissertation, Evanston, IL, USA, 2003, aAI3118622.

[8] M. Guhe and A. Lascarides, "Game strategies for the settlers of catan," in *2014 IEEE Conference on Computational Intelligence and Games*, Aug 2014, pp. 1–8.

[9] A. Fern and P. Lewis, "Ensemble monte-carlo planning: An empirical study," in *Proceedings of the Twenty-First International Conference on International Conference on Automated Planning and Scheduling*, ser. ICAPS'11. AAAI Press, 2011, pp. 58–65.

[10] R. Coulom, *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 72–83.

[11] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, March 2012.

[12] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2-3, pp. 235–256, May 2002.

[13] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik, *Parallel Monte-Carlo Tree Search*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 60–71.

[14] R. Bjarnason, A. Fern, and P. Tadepalli, "Lower bounding klondike solitaire with monte-carlo planning," in *Proceedings of the Nineteenth International Conference on International Conference on Automated Planning and Scheduling*, ser. ICAPS'09. AAAI Press, 2009, pp. 26–33.