AN ABSTRACT OF THE THESIS OF

Shankar Jothi for the degree of Master of Science in Computer Science presented on June 8, 2015.

Title: Evaluation of Parallel Monte Carlo Tree Search Algorithms in Python

Abstract approved: ____

Alan P. Fern

Monte-Carlo Tree Search (MCTS) is an online-planning algorithm for decision-theoretic planning in domains with stochastic and combinatorial structure. The general applicability of MCTS makes it an ideal first choice to investigate when developing planners for complex applications requiring automated control and planning. The first contribution of this thesis is to develop a new open-source Python MCTS planning library (PyPlan), which allows for fast prototyping of MCTS solutions in new domains. While the library can offer a final solution for applications that do not impose strict real-time constraints, for other applications the interpreted nature of Python may not meet the time constraints. This issue leads to the second contribution of this thesis, which is to study the effectiveness of various parallel versions of MCTS for speeding up the performance of PyPlan. We evaluate these algorithms on several complex planning problems using cloud-based resources. Our findings point to a preferred method for parallelizing PyPlan and

also indicates that some prior methods are quite ineffective due to particular aspects of the Python language.

Evaluation of Parallel Monte Carlo Tree Search Algorithms in Python

by

Shankar Jothi

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 8, 2015
Commencement June 2015

Master of Science thesis of Shankar Jothi presented on June 8, 2015.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

Shankar Jothi, Author

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Alan Fern for his continuous time and support throughout this work, his enthusiasm to keep experimenting wider and helping me learn to always ask the right questions. I would also like to thank him for funding this work to get results on time and accomplish my goals. My sincere gratitude to Dr. Prasad Tadepalli and Dr. Mike Bailey for accepting the invitation and taking time to be a part of my Graduate committee.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF ALGORITHMS

Chapter 1: Introduction

Monte Carlo Tree Search (MCTS) is a method used for solving planning problems in Artificial Intelligence especially in Combinatorial Games[1]. Upper Confidence Bounds applied to Trees (UCT) is the most widely used variant of MCTS. With the conjunction of UCT, MCTS has yielded phenomenal breakthrough in the domain of Go [2].

Recent works have explored the effectiveness of three main parallel variants of MCTS; root, tree and leaf parallelization and it has been shown that root parallelization method outperforms all the other variants[3]. Few cases were studied in which the tree parallel method outperforms root parallelization [3].

This method of constructing ensembles in parallel also proves to be efficient in time [4]. With the advent of GPU-based systems having huge computational capabilities, a hybrid GPU-CPU block-parallel method was introduced[5].

In this work, we have introduced a new open-source Monte Carlo planning library PyPlan written in Python. We evaluated the performance of each parallel method and also studied its effectiveness for speeding up the performance of our library. The experiments were run on three domains using the simulators from our library; Connect4, Yahtzee and Tetris.

We were able to show that the performance of the root-parallel method of parallelization in PyPlan is better than all the other methods and the block-parallel method performs nearly equal in certain cases. Due to specific constraints in implementation using Python, other parallelization methods proved ineffective in certain domains. We were able to show an increase in performance for the same methods in specific settings but we did not find conclusive evidence about the efficiency of these methods in domains with real-time constraints using PyPlan.

Chapter 2: Monte Carlo Tree Search with UCT

Monte Carlo methods involve in repeated sampling over the state-space to quantify the effectiveness of each action from the sampled state. Planning problems in AI especially combinatorial games are solved by constructing a look-ahead tree from the given state and searching for the best moves at each node. MCTS is an online planning algorithm that combines the concept of repeated sampling with random simulation and the precision of a tree search. The UCT variant of MCTS provides a way to intelligently bias between exploration and exploitation. Given a state value to UCT, it builds a look ahead tree over the state-space with the given state at the root node.

Each trajectory in UCT involves in selecting the next node based on UCB tree policy until a leaf node is reached, expanding the node, playing a random game until end from that node and then backtracking the reward values to all the nodes in that trajectory. Unlike other depth-bounded trees, UCT does not impose any bounds on the depth and it guarantees an optimal solution provided more number of trajectories. The essential part of UCT lies in the node selection phase which intelligently alters between exploring a new node and exploiting a previously explored node. An UCT tree with more depth implies that the bias was towards exploiting actions that looked better during past explorations and more width implies that the tree was explored more.

Each node in the UCT tree represents a state value and the edges represent an action taken from the parent node resulting in the corresponding child node. Every node in the tree stores its visit count and $Q(s, a)$ for each of the action values from that node's state and the player's turn at that node. The alternative depths in the tree correspond to a different player's turn in the order in which the game is played. At each node, an action with maximum value is chosen according to a given tree policy.

2.1 UCB Tree Policy

The first step in each trajectory is node selection. If all actions at a given node in the tree is not explored at least one time then a random action is chosen. If all the actions are explored at a node then the best action at this node is selected using a tree policy. UCT involves in applying Upper Confidence Bound (UCB) as the tree policy which is given by:

$$Q^{\oplus}(s, a) = Q(s, a) + c\sqrt{\frac{\log n(s)}{n(s, a)}} \tag{2.1}$$

The first term in the formula is the exploitation term which is used to bias the selection towards the actions that were previously explored and the second term to bias it towards unexplored actions. $Q(s, a)$ is the action value estimate of a at state s. $n(s)$ is the visit count of the state s and $n(s, a)$ is number of times the action a was explored previously from state s in all the previous trajectories. The value of the constant c is decided based on the domain. For a chosen domain, higher the value of c, more new actions are explored at a node. After finishing a rollout from a newly expanded node, the reward R from the rollout is backtracked to all the nodes in the current trajectory. The action value node is the mean value of the reward of all the trajectories that went through that node. The action value update after the end of a rollout for each node is as follows:

$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{n(s,a)}(R - Q(s,a)) \tag{2.2}$$

The selection phase continues until a leaf node is hit and if the selected node is a terminal node then the reward of that node is backtracked to all the nodes in the trajectory rather than executing the rollout phase. Every rollout has a horizon value which is used to stop simulations that take longer. The horizon value imposes a bound on the length of each rollout. This helps in certain domains like Tetris where each game could ideally take forever. Every trajectory in the tree incrementally expands the tree and the shape of the UCT tree could be altered by adjusting the value of the constant c. The algorithm for MCTS with UCT as tree policy is given in Algorithm 1.

2.2 Example simulation of UCT

There are four phases during one simulation episode in an UCT tree; node selection, expansion, simulation and backpropagation. Let us assume that we have a deterministic zero-sum domain and two agents playing in the domain. At the end of the game, a player's reward in this domain for winning is 1.0, losing is -1.0 and

Algorithm 1 UCTAgent

```
**Input:**$s_i \leftarrow$ Input state
$simc \leftarrow$ Simulation count
$h \leftarrow$ horizon
**Output:**Q value of all actions from input state
1: for i = 1 to $simc$ do
2:     $s_c \leftarrow s_i$
3:     $trajectory = []$
4:     while all actions of $s_c$ are explored do
5:         Add current state $s_c$ to $trajectory$
6:         $s_c \leftarrow$ Select next state using TreePolicy at $(s_c)$
7:     if $s_c$ is terminal then
8:         backtrack reward to all nodes in $trajectory$
9:     else
10:        $tempstate$ = Sample a new unexplored action and create a new state
11:        Add $tempstate$ to current state $s_c$ as a child state.
12:        $reward$ = Simulate a game randomly from $temp\_state$ till end or $h$
13:        Backtrack $reward$ to all nodes in $trajectory$
return $Q(s_i, a)$ for all actions from state $s_i$
```

0.0 if the game is a draw. The rewards are represented as a vector with dimensions equal to the number of players. The possible rewards that could be obtained are [1.0, -1.0], [-1.0, 1.0] and [0.0, 0.0]. The simulations start at the root node with state value equal to the given input state. In the node selection phase, if all actions at this node are not explored at least once then the next unexplored action is taken from the node. In the node expansion phase, a new node is created whose state value is equal to the new state recently obtained by taking an unexplored action in the previous state. The state visit counts are updated for the root node and the new node created. In the simulation phase, a random game is played from the newly expanded node till the end. However, if all actions are explored then the next node is selected based on the tree policy until a leaf node is reached. A leaf node is defined as the node at which at least one action remains unexplored. In the backpropagation phase, the reward value obtained at the end of the previous random rollout is passed to all the nodes in the current trajectory.

Figure 2.1 shows the first phase of a simulation. The value inside the nodes are the action values represented as a vector. The value to the left of each node represents the state visit count. The color inside the nodes represent the turn of the player at that node. Since the domain includes more than one player taking alternative turns, each node selects the maximum Q value during the node selection phase and the best action (action with highest Q value) at the root node is the action to be taken by the UCT agent. Let us assume that the UCT agent is player 1. At the root node, two actions could be taken but the first child node has an action value of 1.0 whereas the second child node has a value of 0.0 for player 1. So, at the end of node selection phase at the root node, according to the action-value estimate, first child node would be selected. At this node, there is one unexplored action present which is chosen. The trajectory is marked in red. Figure 2.2 shows the expansion and simulation phase. The newly expanded node is shown in blue color. After the end of a random rollout, the reward obtained is [0.0, 0.0] (Figure 2.3). Figure 2.3 shows the final tree structure and node values after backpropagating the reward value to all the nodes in the trajectory.

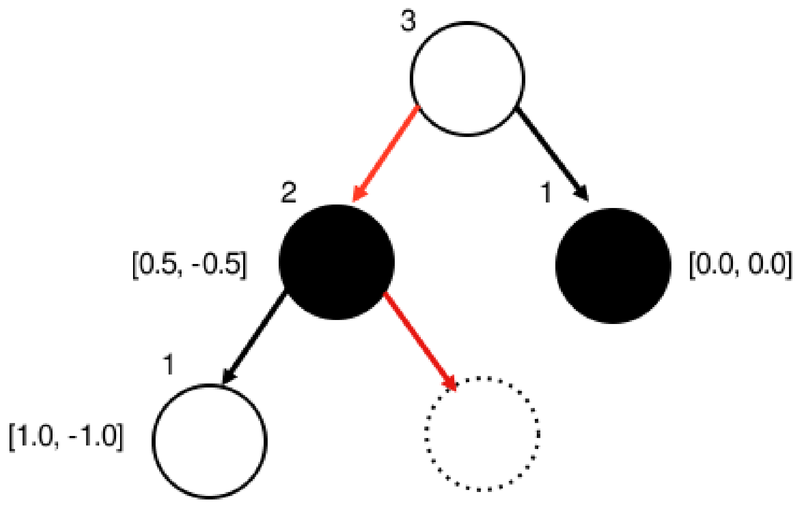


Figure 2.1: Selection and expansion

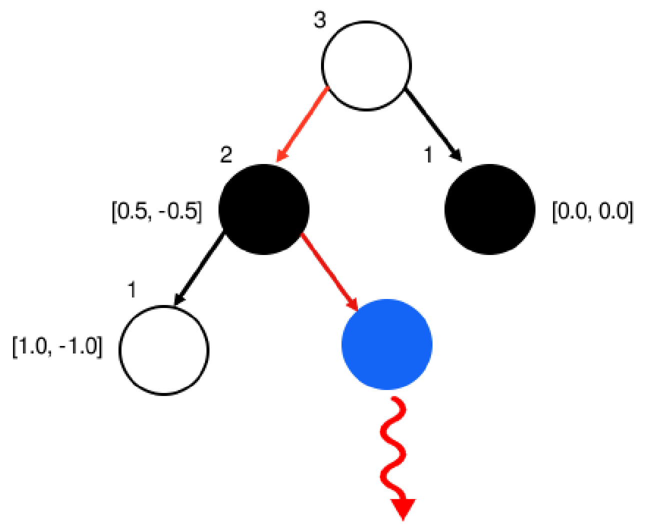


Figure 2.2: Simulation

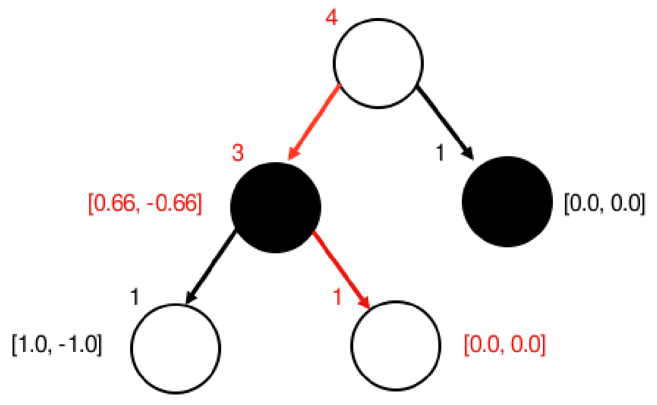


Figure 2.3: Backpropagation

Chapter 3: Parallel MCTS Algorithms

In this chapter, we discuss about the various ways of parallelizing Monte Carlo Tree Search. We have considered parallelizing the tree search on a symmetric multiprocessor (SMP) where every physical processing core shares the same main memory. Parallelization on MCTS could be applied on certain steps of a trajectory based on how the data is being used. We could construct more than one tree in parallel from the root node thereby parallelizing during node selection phase at the root node. Also when the simulation phase starts, the remaining parts of the tree remain idle. This enables us to start a new trajectory on the tree while the rollout happens. This chapter distinguishes the three main parallelization methods; root parallelization, leaf parallelization and tree parallelization. We have also included another parallelization method; block parallelization which was introduced as a hybrid CPU-GPU implementation.

3.1 Motivation for Multiprocessing

Due to simultaneous access of the same memory region, we might encounter strange results during the tree search. So, parallel execution needs the use of a mutex mechanism such as locks or semaphores to ensure consistency. Multithreading involves in parallel execution of threads sharing the same memory space. However, in Python, because of the presence of global interpreter lock (GIL), multithreading is not truely parallel. Python programs are interpreted. The interpreter needs to give its execution time for each thread by locking its execution space for that thread. If one python thread acquires the lock then all the other threads will be busy waiting to acquire the interpreter's lock to enter the execution space. This adds a lot of latency in the execution process. Hence we used multiprocessing in Python for our experiments. Multiprocessing in Python involves executing individual Python processes that execute in their own memory space. To share data between processes, we need to use any of the interprocess communication (IPC) methods. However, there is a disadvantage when it comes to multiprocessing because of the IPC overhead. To avoid this, we used the Manager model in Python which involves in creating a manager process that holds the tree data and every worker process communicates with the manager to retrieve the needed data and run in parallel. We used locks to make sure that only one process is working on a node as this ensures consistency.

3.2 Root Parallelization

Root parallelization consists of building multiple MCTS trees from the input state in parallel. This method is depicted in Figure 3.1. Each process receives a copy of the input state and then constructs the tree in parallel. Depending on each of the MCTS phases, each tree might get explored in a different way and the action-value estimate of each child at the root node would be different. After executing the given number of trajectories, all the action-value estimates for each child node from the root are averaged and the best child node (node with highest mean Q-value) is returned. This method does not involve in any interprocess communication as each tree is constructed with its own memory space and is the easiest of all the methods. The benefit of this algorithm is its time efficiency. As the number of parallel processes increase, the total number of trajectories are quickly generated.

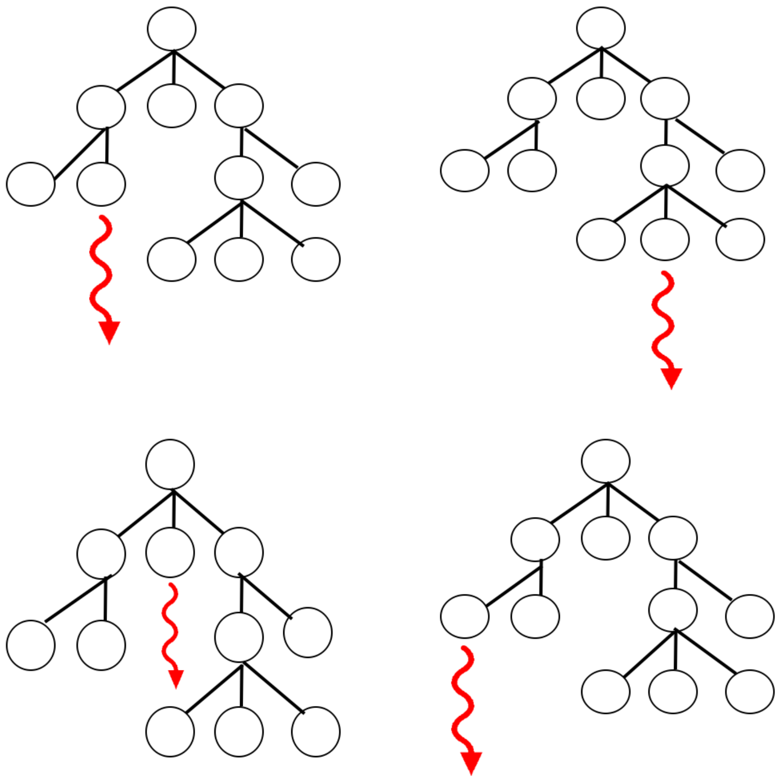


Figure 3.1: Root parallelization

Also root parallelization method eliminates the effect of the UCT constant setting. The sequential version of UCT could stay in the local optima for a long time. For smaller values, the UCT algorithm exploits a promising node more often than the other nodes and searches more deeply in the tree. For higher values, it could explore many nodes and searches more wider. If one random rollout ends in a positive reward then the sequential UCT for a lower constant setting would end up exploiting the same action again though the probability of that random rollout is really less. Multiple trees generated could ideally avoid getting stuck in such local optimum. The algorithm for Root parallelization is given in Algorithm 2.

Algorithm 2 RP-UCTAgent

```
   Input:$s_i \leftarrow$ Input state
   $simc \leftarrow$ Simulation count
   $h \leftarrow$ horizon
   $ens\_c \leftarrow$ Ensembles count
   Output:action
1: $ensemble\_rewards = []$
2: for i = 1 to $ens\_c$ do
3:     $action\_rewards \leftarrow run\_in\_parallel(UCTAgent(s_i, simc, h))$
4:     $ensemble\_rewards.push(action\_rewards)$
5: $avg\_rewards$ = Average of all action rewards in $ensemble\_rewards$
   return $argmax_a avg\_rewards(s_i, a)$
```

3.3 Leaf Parallelization

In Leaf parallelization, only one process runs the given number of trajectories on the tree. However, in the simulation phase, multiple games are played and the results of these games are backpropagated to all the nodes in the current trajectory. Leaf parallelization method also does not involve in any interprocess communication and there is only one process working on the tree. So there are no mutexes involved in this method. This method is depicted in Figure 3.2.

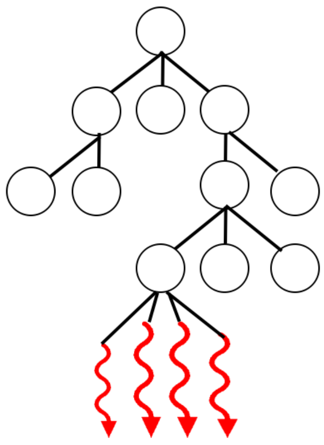


Figure 3.2: Leaf parallelization

A disadvantage with this method is that playing more than one game takes relatively more time. When run in parallel, the simulation phase has to wait for the rollout that takes the longest time among all the simulations. When the newly expanded state is near to a terminal state, playing n games in parallel would add more overhead for forking a new process and the random game play. Such states could be clearly marked as returning positive or negative reward with one or two rollouts rather than 8 or 10 parallel redundant rollouts. The algorithm for Leaf parallelization is given in Algorithm 3.

Algorithm 3 LP-UCTAgent

```
**Input:**$s_i \leftarrow$ Input state
$simc \leftarrow$ Simulation count
$h \leftarrow$ horizon
$tc \leftarrow$ leaf simulation count
**Output:**Q value of all actions from input state
1: for i = 1 to $simc$ do
2:     $s_c \leftarrow s_i$
3:     $trajectory = []$
4:     while all actions of $s_c$ are explored do
5:         Add current state $s_c$ to $trajectory$
6:         $s_c \leftarrow$ Select next state using TreePolicy at $(s_c)$
7:     if $s_c$ is terminal then
8:         backtrack reward to all nodes in $trajectory$
9:     else
10:        $tempstate$ = Sample a new unexplored action and create a new state
11:        Add $tempstate$ to current state $s_c$ as a child state.
12:        $reward$ = Simulate $tc$ parallel games from $temp\_state$ and return all rewards
13:        Backtrack $reward$ to all nodes in $trajectory$
return $Q(s_i, a)$ for all actions from state $s_i$
```

3.4 Tree Parallelization

Tree parallelization method involves in using one shared tree for all the parallel processes. Each process obtains a lock as needed and then executes the phases based on the information obtained from the tree. We use mutex-locks to lock various sections of the search tree. Depending on the mutex-lock location in the tree, we have two types of tree parallelization method; global mutex and local mutex.

3.4.1 Global Mutex

In global mutex method, the whole tree is locked during the selection, expansion and backpropagation phase. When a process enters the simulation phase, other processes enter the tree. This method ensures that only one process can access the search tree at a time. This method is shown in the Figure 3.3.

In this method, rollouts occur in parallel. The difference between this method and leaf parallelization is the key idea of rollouts from different leaf nodes rather than multiple rollouts starting from the same leaf node. A clear disadvantage of this method is time spent by the waiting process until the selection and expansion phases are over. This decreases the number of trajectories per second by a considerable factor. The algorithm for this method is given in Algorithm 4.

Algorithm 4 GM-UCTAgent

```
    **Input:**$s_i \leftarrow$ Input state
    $tcc \leftarrow$ Thread count
    $h \leftarrow$ horizon
    **Output:**Q value of all actions from input state
 1: for i = 1 to $tc$ do
 2:     $s_c \leftarrow s_i$
 3:     $trajectory = []$
 4:     Lock the node $s_i$
 5:     while all actions of $s_c$ are explored do
 6:         Add current state $s_c$ to $trajectory$
 7:         $s_c \leftarrow$ Select next state using TreePolicy at $(s_c)$
 8:     if $s_c$ is terminal then
 9:         backtrack reward to all nodes in $trajectory$
10:         Release the node $s_i$
11:     else
12:         $tempstate$ = Sample a new unexplored action and create a new state
13:         Add $tempstate$ to current state $s_c$ as a child state.
14:         $reward$ = Simulate a game randomly from $temp\_state$ till end or $h$
15:         Release the node $s_i$
16:         Backtrack $reward$ to all nodes in $trajectory$
    return $Q(s_i, a)$ for all actions from state $s_i$
```

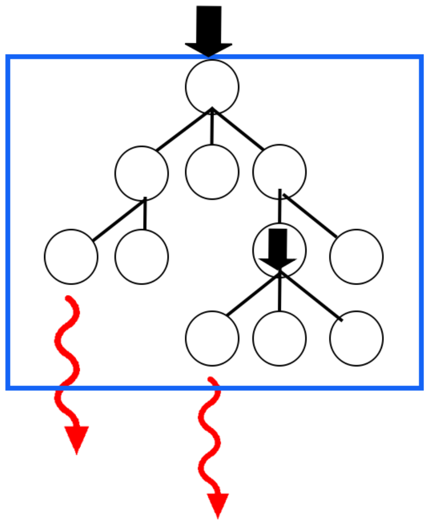


Figure 3.3: Tree parallelization with Global mutex

3.4.2 Local Mutex

Local mutex method involves in locking various sections of the tree. When a process p_1 enters the selection phase, the node is locked until the process selects a child node or starts a rollout. The process p_1 releases the lock when it selects the next node or expands a new node to start a random rollout. This ensures that no other process updates the values of the node while p_1 is working on it. After the process p_1 releases the lock on the node, any other process obtains the lock on the node. This method involves in frequent locking and unlocking of various nodes in the tree. This is a disadvantage in this method. This method is shown in Figure 3.4 and the algorithm is given in Algorithm 5.

Algorithm 5 LM-UCTAgent

```
    Input:$s_i$ ← Input state
    $tcc$ ← Thread count
    $h$ ← horizon
    Output:Q value of all actions from input state
 1: for i = 1 to $tc$ do
 2:     $s_c \leftarrow s_i$
 3:     $trajectory = []$
 4:     Lock the node $s_c$
 5:     while all actions of $s_c$ are explored do
 6:         Add current state $s_c$ to $trajectory$
 7:         Release the node $s_c$
 8:         $s_c$ ← Select next state using TreePolicy at ($s_c$)
 9:         Lock the node $s_c$
10:     if $s_c$ is terminal then
11:         backtrack reward to all nodes in $trajectory$
12:         Release the node $s_c$
13:     else
14:         $tempstate$ = Sample a new unexplored action and create a new state
15:         Add $tempstate$ to current state $s_c$ as a child state.
16:         $reward$ = Simulate a game randomly from $temp\_state$ till end or $h$
17:         Release the node $s_c$
18:         Backtrack $reward$ to all nodes in $trajectory$
    return $Q(s_i, a)$ for all actions from state $s_i$
```

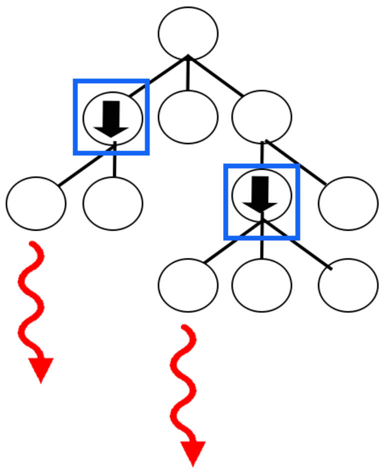


Figure 3.4: Tree parallelization with local mutex

3.5 Block Parallelization

Block parallelization is a hybrid CPU-GPU based parallel implementation of MCTS. It is also a combination of root and leaf parallelization methods. Multiple trees are generated by individual processes from the given state as the root node. Each process incrementally constructs the search tree in its own memory space. During the simulation phase, multiple rollouts are executed in parallel similar to the leaf parallelization method. The proposed method runs the parallel rollouts by assigning specific computation blocks in GPU to a process (running in CPU) which started the rollouts. This method is shown in the Figure 3.5. However, we run these rollouts on the CPU itself by adjusting the process count for the tree search and for rollouts. For example, let us consider the number of parallel rollouts during the simulation phase as 16. In a 32 core CPU, to execute 1024 trajectories, we use 16 cores with 64 trajectories each and then assigning the remaining 16 cores for the rollouts. This implementation is less time efficient when compared to root parallelization in which all the CPU cores are assigned for the tree expansion. The algorithm for this method is given in Algorithm 6.

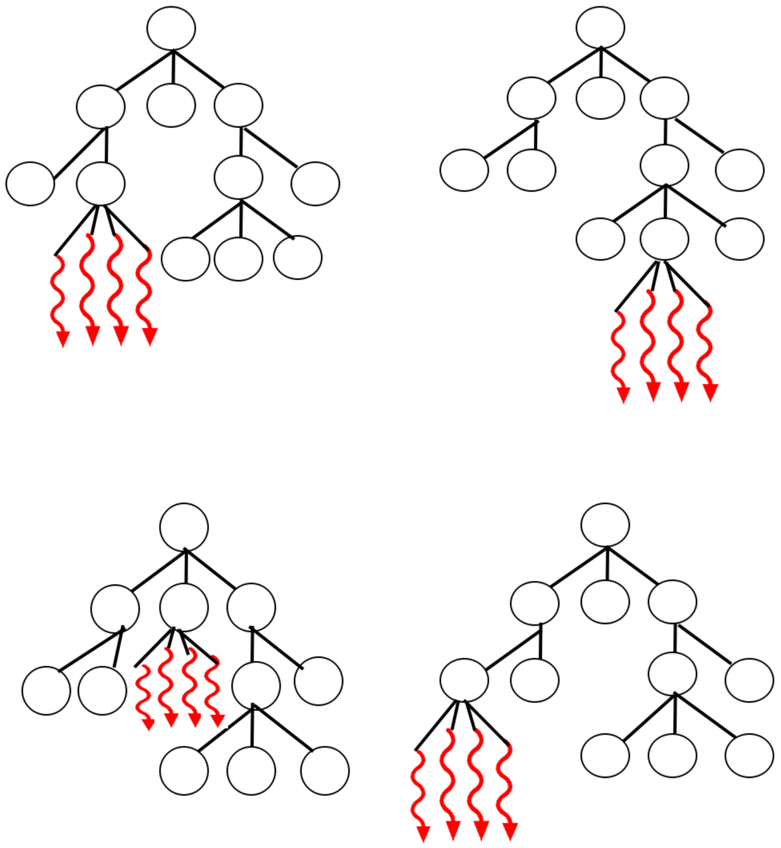


Figure 3.5: Block parallelization

Algorithm 6 BP-UCTAgent

```
    Input:$s_i$ ← Input state
    $simc$ ← Simulation count
    $h$ ← horizon
    $ens\_c$ ← Ensembles count
    $t\_c$ ← Leaf simulation count
    Output:action
1:  $ensemble\_rewards = []$
2:  for i = 1 to $ens\_c$ do
3:      $action\_rewards \leftarrow run\_in\_parallel(LP - UCTAgent(s_i, simc, h, tc))$
4:      $ensemble\_rewards.push(action\_rewards)$
5:  $avg\_rewards$ = Average of all action rewards in $ensemble\_rewards$
    return $argmax_a avg\_rewards(s_i, a)$
```

Chapter 4: Prior Evaluations

Our work is an extension of the previous work that explored the various ensemble configurations of the Root parallel variant of MCTS parallelization. The work concluded that the ensembles in a parallel model almost always improve performance given a fixed amount of time per decision [4] and the exploration of domains included Connect4, Yahtzee, Havannah, Biniax and Backgammon. This work is a part of evaluation of parallelization methods of UCT.

Prior works to this evaluated various other methods of parallelization; Leaf parallelization and Tree parallelization [3]. However, the domain that was evaluated in this work limited itself to the popular Chinese board game Go. A multitude of combinatorial games, single-player games, real-time games, non-deterministic games and certain no-game applications were evaluated in different works [6]. But most of the works limited itself in the methods of parallelization on the evaluated domains.

The work on evaluating various methods of parallelization limited itself to the number of domains evaluated [3]. The evaluation of the methods of parallelization were based on few measures; Games-per-second (GPS) and Strength speedup. Each of these measure determines the relative efficiency of a parallel method when compared to the sequential version of UCT. GPS is the ratio of number of simulated games per second by the parallel method to the number of simulated games per second by the sequential method. Strength speedup is the measure of increase of time needed for the sequential UCT method to achieve the same efficiency as that of the parallel method.

The results of this work concluded that the root parallel methods performs the best and the leaf parallel variant performs bad compared to all the other methods. Also the normal version of tree parallel methods performs relatively better than leaf parallel with a strength-speedup of 3.3. However, there were certain cases in which tree parallel performs better than the root parallel method and might be useful in specific cases. Another work evaluated the performance of the tree parallel method without any locks thereby allowing redundant selections and simulations during node expansion. This resulted in a bad performance compared to the lock-based versions of tree parallel method [7].

Another method of parallelization, block parallelization is a hybrid CPU-GPU based parallelization technique [5]. The evaluation of this method was performed on the domain Othello and was compared to the performance of leaf parallelization. The work concluded that block parallelization performs better than the leaf parallelization method provided the hardware benefits of a GPU. Our extension from this work is to study the effectiveness of block parallelization run on a CPU-only environment, evaluate it on various other domains and compare its performance to other methods of parallelization.

Chapter 5: PyPlan - Planning Library for Python

PyPlan is a set of modular Monte-Carlo planning libraries for Python. The main goal of this project is to offer easy-to-use and flexible planning algorithms. Flexibility is offered in terms of input parameters to the algorithms and easily altering the parameters of execution environments. This library was built from the ground up to provide an API to easily create planning algorithms, domains and also easily interface between them. Using this, anyone could create various domains and agents, plug various other agents for the same domain and make effective comparisons. There are 11 agents and 5 domains. The list of available agents in the library are :

1. Uniform Rollout [8] [9]
2. Incremental Uniform Rollout [8]
3. e-Greedy Rollout [8]
4. Random [8]
5. UCT
6. Ensemble UCT (Root Parallel)
7. Tree Parallel (Local Mutex. With Virtual loss)

8. Tree Parallel (Local Mutex without Virtual loss)
9. Leaf Parallel
10. Block Parallel (Only on CPU)
11. Tree Parallel (Global Mutex)

The list of available simulators are :

1. Tic Tac Toe
2. Connect4
3. Yahtzee
4. Tetris
5. Othello

5.1 Architecture

The library is built in an object-oriented fashion to enable developers to easily use and extend. There are two main abstract-base classes in the library; Agent and Simulator. The other main class is the Dealer which executes the desired number of simulations and returns various statistics about each game that was played. The parameters for initiating a dealer class are the list of agents, simulator object, number of simulations, horizon value for each simulation and the output file to which the results are written. The structure of PyPlan is shown in Figure 5.1.

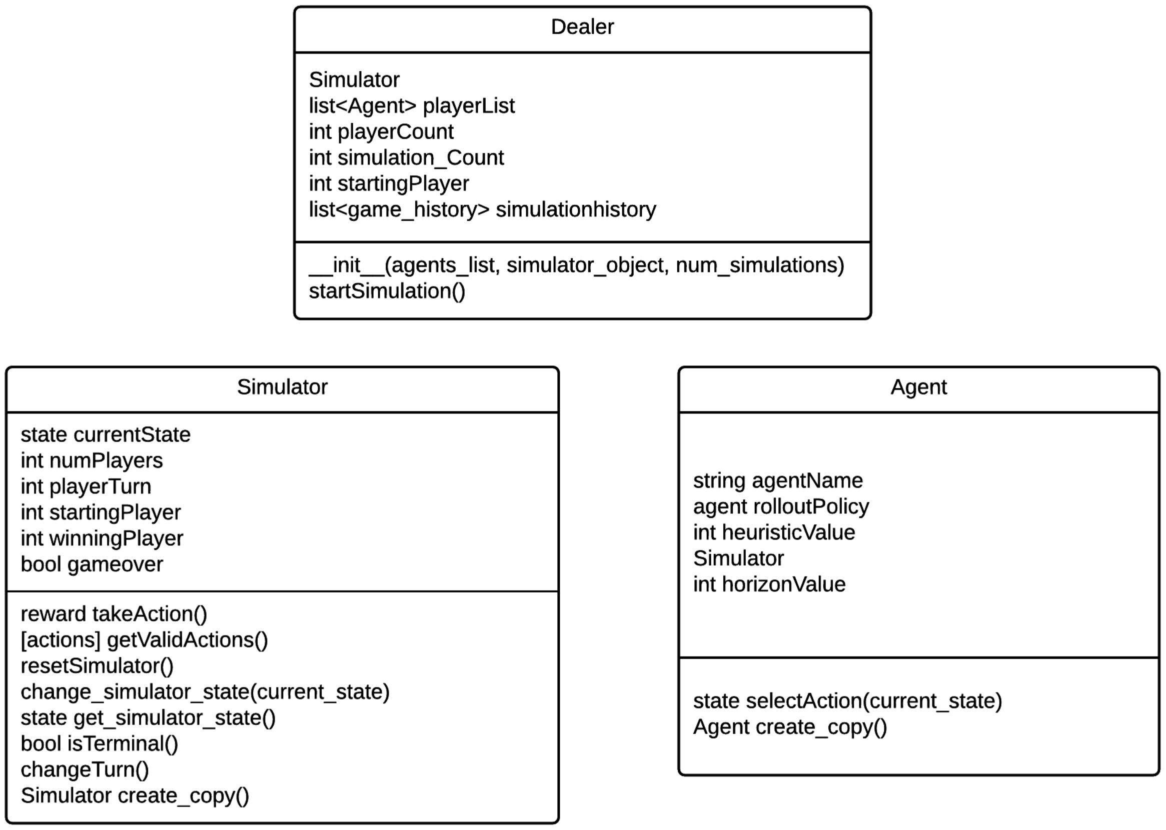


Figure 5.1: PyPlan Structure

The agent and simulator classes are inherited to create new planning agents and domains respectively. The agent class takes a copy of the simulator and rollout-policy during its instantiation. Every agent uses this copy of the simulator to run the rollouts. This simulator could be the copy of the actual simulator used in the game or a partial version of the simulator. All the other parameters for agent instantiation depends on the agent's characteristics. The simulator class stores the current state of the game and its status. The current state object holds the information about the state of the game and the player's turn. The dealer runs all the simulations. It stores the list of agents and a copy of the simulator. The dealer class executes the gameplay by calling each agent during its turn. It aggregates the statistics of each move by the agents and returns it back to the invoking function. The statistics of each move include the action taken by the agent and the reward obtained. The dealer also calculates the time taken for every move. Using this, we could calculate the overall statistics for the given number of simulations.

5.2 Running Simulations

PyPlan is a library built for developers to use the available planning agents very easily. It also enables easy configuration of these agents to different settings according to the problem. The Dealer class runs all the simulations and returns the statistics at the end. Depending on the verbose options, the dealer could be enabled to output each game's statistics or even every action. There are two ways in which PyPlan could be used to run simulations; using the API or using the Job Parser.

5.2.1 Using the API

The PyPlan API could be directly used to create agents, simulators and invoke the dealer to run specific number of simulations using these agents and the simulators. This approach is useful when we need to run few simulations to output the statistics of every action taken by the agent. We could also customize each parameter of the agents and simulator as needed. This method is depicted in the Figure 5.2.

```
1: simulator_obj = connect4simulator.
   Connect4SimulatorClass(num_players = 2)

2: agent_one = randomagent.RandomAgentClass(simulator =
   simulator_obj)
3: agent_two = uniformagent.UniformRolloutAgentClass(
   simulator = simulator_obj, rollout_policy = agent_one
   , pull_count = 10)
4: agent_three = uniformagent.UniformRolloutAgentClass(
   simulator = simulator_obj, rollout_policy = agent_two
   , pull_count = 10)

5: agents_list = [agent_three, agent_one]

6: dealer_object = dealer.DealerClass(agents_list,
   simulator_obj, num_simulations = 10, sim_horizon =
   100, results_file = "test.csv")
7: dealer_object.start_simulation()

8: results = dealer_object.simulation_stats()
```

Figure 5.2: Using PyPlan API

The first line in figure 5.2 shows the simulator object being created. Each of the simulator object are a class on its own. Here, we have created the connect4 simulator. The simulator objects takes the input of number of agents playing in the current domain. Three agents are created in the next three lines. We could see that the `agent_three` takes `agent_two` as its rollout agent which in turn takes `agent_one` for its rollout. This is a nested rollout case. We could observe the flexibility that is offered using this library. For every agent, the `time_limit` parameter is set at -1 to disable time limits. However, if the `time_limit` parameter is set, all other parameters are ignored (ex. number of simulations). The dealer object is then created in line 6. It takes the input of list of agents playing in the current domain, the simulator object, simulation horizon and the file name to which all the statistics are written. We could also set `verbose=True` if we want to display the output on the console. After running all the simulations, the dealer object stores the results. We could retrieve the results as show in line 8 by calling the `simulation_stats()` function. The returned value is an array of two objects; details about every action in all the games by each agent (including its reward), winning agent in every game.

5.2.2 Using the Job Parser

PyPlan includes a XML Job Parser using which sets of simulations could be run. This is useful to run large number of simulations with varying agents and simulators sequentially. This method is shown in Figure 5.3.

Figure 5.3 shows a number of simulations between different agent configurations. The structure of this file (jobs.xml) starts with a root node jobs tag. Every job is defined in a job node. The parameters of each job node are the same as that of its dealer object instantiation. Every agent is defined in a player node inside a job tag. The parameters of each player are same as its object's parameters during instantiation. The results of each job is printed on the console by default and to the file given in the parameter `output_file`. The number of simulations is given

```
<jobs>
    <job playercount="2" sim_count="500" sim_horizon
       ="100" game="yahtzee" output_file="Results/
       locm_yahtzee_256_g500.csv">
        <player number="5" num_simulations="1024"
           uct_constant="30" horizon="100" time_limit
           ="-1"/>
        <player number="8" num_simulations="8"
           threadcount="32" uct_constant="30" horizon
           ="100" time_limit="-1"/>
    </job>

    <job playercount="2" sim_count="500" sim_horizon
       ="100" game="yahtzee" output_file="Results/
       locm_yahtzee_1024_g500.csv">
        <player number="5" num_simulations="1024"
           uct_constant="30" horizon="100" time_limit
           ="-1"/>
        <player number="8" num_simulations="32"
           threadcount="32" uct_constant="30" horizon
           ="100" time_limit="-1"/>
    </job>
</jobs>
```

Figure 5.3: Using PyPlan Job Parser

using the parameter `sim_count` and the domain type using the `game` parameter.

5.3 Extending PyPlan

PyPlan could be extended to add more domains or more agents. The architecture of PyPlan includes the abstract base classes for simulators and agents from which each of the simulator and agent is inherited from. They have to meet certain requirements to qualify as an agent or simulator class and to easily interface with the dealer for running the simulations.

5.3.1 Adding new agents

The structure of the abstract base agent class is shown in the architecure (Figure 5.1). Every agent must specify definition for the methods `select_action()`, `get_agent_name()` and `create_copy()`. The simulator calls the method `select_action()` on a agent to get the action to be taken on the simulator by passing the current state to the agent. The instantiation parameters for every agent depends on its characterisitcs except a copy of simulator. The Figure 5.4 shows the agent constructors for an UCT agent and an uniform rollout agent.

```
# UCT AGENT
def __init__(self, simulator, rollout_policy,
   tree_policy, num_simulations, uct_constant=1, horizon
   =10, time_limit=-1):

# UNIFORM ROLLOUT AGENT
def __init__(self, simulator, rollout_policy, pull_count
    = 5, horizon=5, heuristic=0):
```

Figure 5.4: Agent class constructors

5.3.2 Adding new domains

Adding new domains to PyPlan requires relatively more work than adding new agents. The simulator of a every domain needs two classes to be defined along with it; a state class and an action class. Every State class must hold a private dictionary `current_state` with two values `state_val` and `current_player`. Python is a dynamically typed language. Hence the `state_val` could be dynamically assigned to different types of state values for the same domain as needed. `current_player` is the turn of the agent that needs to take the action on the `current_state`. Python assigns objects to the other by reference. So we need to define a `create_copy` method for all the custom objects created in PyPlan; simulator, state, action and agents. This method creates new copies of all the private variables and returns a new object. The `get_valid_actions()` method of the simulator returns all the possible actions from the current state of the simulator. The `take_action()` method returns the reward of taking the given input action object on the current state of the simulator. The abstract base class structure of the simulator is shown in Figure 5.1.

Chapter 6: Overview of Domains

In this section, we describe about the three domains used in our experiments; Connect4, Yahtzee and Tetris. The characteristics of each of the domain are discrete, fully observable and deterministic except Yahtzee which is stochastic. Figure 6.1 shows the comparison of all these domains.

Domain	Maximum players	UCT Constant	APS	SPA
Yahtzee	2	6.0	32 + Number of unscored categories	252
Connect4	2	0.8	7	1
Tetris	1	20.0	Number of rotational configurations * 10	1

Figure 6.1: UCT constant for Yahtzee.

6.1 Yahtzee

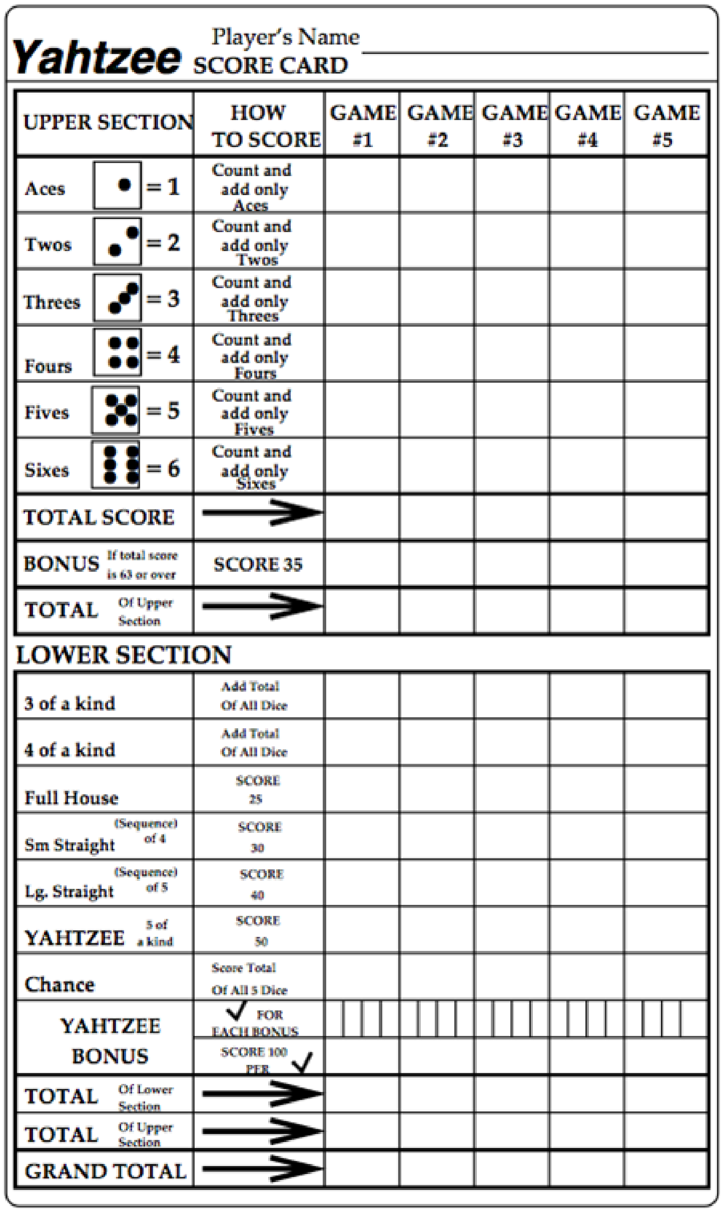
Yahtzee has a maximum player count of two in a game and it is stochastic. In our experiments, we have one agent playing the game. An optimal strategy or obtaining maximum score of 254.5896 in Yahtzee is present [10]. The gameplay of yahtzee involves in each player taking alternative turns to roll dices (6 faces) and score categories. There are five dices in the game and 13 categories to score. During a player's turn there are three rounds to roll the dices. In the first round, the player has to roll all the five dices and in the second and third round, the player could choose the dices to roll and the dices to keep. The player could also not roll after any round and choose to score a category. Once a category has been scored, the same category is not chosen again. The score categories has two sections; the upper and lower section. The upper section has 6 categories and the lower section has 7 categories. Each of the sections are explained below. The game is depicted in the Figure 6.2.

Upper section

1. Ones - Get maximum ones on the dices as possible
2. Twos - Get maximum twos on the dices as possible
3. Threes - Get maximum threes on the dices as possible
4. Fours - Get maximum fours on the dices as possible
5. Fives - Get maximum fives on the dices as possible
6. Sixes - Get maximum sixes on the dices as possible

Lower section

1. Three of a kind - Get the same number on three dices. Points are the sum of all the dices after the roll.



Yahtzee Player's Name ____________

SCORE CARD

UPPER SECTION	HOW TO SCORE	GAME #1	GAME #2	GAME #3	GAME #4	GAME #5
Aces ⚀ = 1	Count and add only Aces					
Twos ⚁ = 2	Count and add only Twos					
Threes ⚂ = 3	Count and add only Threes					
Fours ⚃ = 4	Count and add only Fours					
Fives ⚄ = 5	Count and add only Fives					
Sixes ⚅ = 6	Count and add only Sixes					
TOTAL SCORE	→					
BONUS If total score is 63 or over	SCORE 35					
TOTAL Of Upper Section	→					

LOWER SECTION

3 of a kind	Add Total Of All Dice					
4 of a kind	Add Total Of All Dice					
Full House	SCORE 25					
Sm Straight (Sequence) of 4	SCORE 30					
Lg. Straight (Sequence) of 5	SCORE 40					
YAHTZEE 5 of a kind	SCORE 50					
Chance	Score Total Of All 5 Dice					
YAHTZEE BONUS	✓ FOR EACH BONUS					
	SCORE 100 PER ✓					
TOTAL Of Lower Section	→					
TOTAL Of Upper Section	→					
GRAND TOTAL	→					

Figure 6.2: Yahtzee score sheet[11].

2. Four of a kind - Get the same number on four dices. Points are the sum of all the dices after the roll.

3. Full house - Get a three of a kind and a pair. 25 points.

4. Small straight - Get four numbers in sequence. 30 points.

5. Large straight - Get five numbers in sequence. 40 points.

6. Chance - Sum of the numbers on the dice.

7. Yahtzee - Five of a kind. Get same number of all the five dices. 50 points.

After the end of the first roll, the player has an option of rolling the dices again or scoring a category. There are 32 combinations in which the player could choose to keep certain dices and roll the other and at the beginning there are 13 categories left to be scored. So the total number of actions at a given state is the sum of the number of unscored categories and the dice roll combinations. The maximum total score that could be achieved in this version of Yahtzee is 340. The objective of this game is to maximize the total score of all the categories.

6.2 Connect4

Connect4 is a game of two players and is a relatively easier deterministic domain. The board of connect4 is a grid of width 7 and height 6. The gameplay of Connect4 involves in each player taking alternative turns to put their respective coins on the board. The board is fixed on its bottom such that a coin inserted at the top of a column falls down to lowest unoccupied position. The objective of the game is to connect four coins of a same color in a row, column or the diagonals. The possible outcomes of the game are win by any one of the players (+1.0 for win. -1.0 for losing) or a draw (0.0) when no more position could be occupied by any player. The game is depicted in the Figure 6.3.

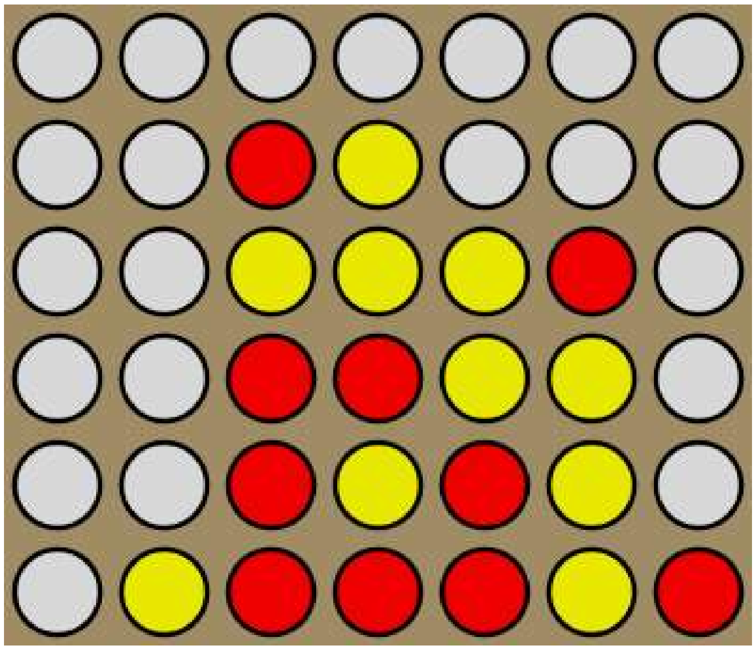


Figure 6.3: Connect4 board.

The maximum number of actions at a given state is 7 in this game as there are a total of seven columns in the grid and the coins fall down to the lowest unoccupied position. Our simulator is really fast compared to various other normal implementations of Connect4. We have used the bitboard representation of the board and bit shifting to check for terminal states and winning positions. The entire board is represented in two 64 bit integers; one for each player in the game.

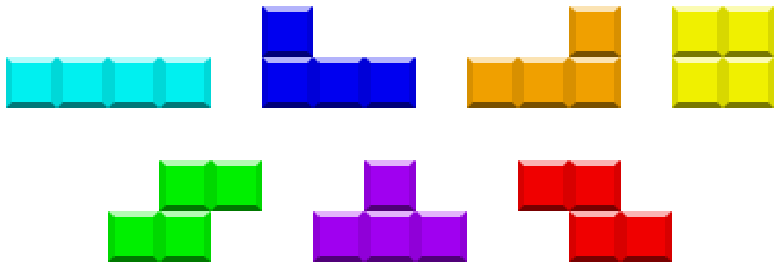


Figure 6.4: Tetris - Tetrominoes.

Because of this implementation, our simulation times decreased by large factors.

6.3 Tetris

Tetris is a single agent domain. It is a popular tile-matching puzzle game. Our implementation is based on tile-filling on the tetris board. The board is a grid of width 10 and height 20. The gameplay of tetris involves in the player filling up the tetrominoes on the grid as they fall down from the top of the tetris grid. There are seven types of tetromino pieces as shown in the Figure 6.4.

Each piece could be rotated 90 degrees clockwise and creating new shapes after each rotation. Every piece has a number of rotational configuration. For example, a piece that is square in shape has 1 rotational configuration as all the rotations would end up in the same shape. From a given state comprising of the current board, the current piece and the next piece, a total of 10 different positions are possible for a given rotational configuration of a piece. Hence the total number of possible actions for a given piece is given by 10 * the total number of rotational configurations for the current piece. A game of tetris is shown in the Figure 6.5.

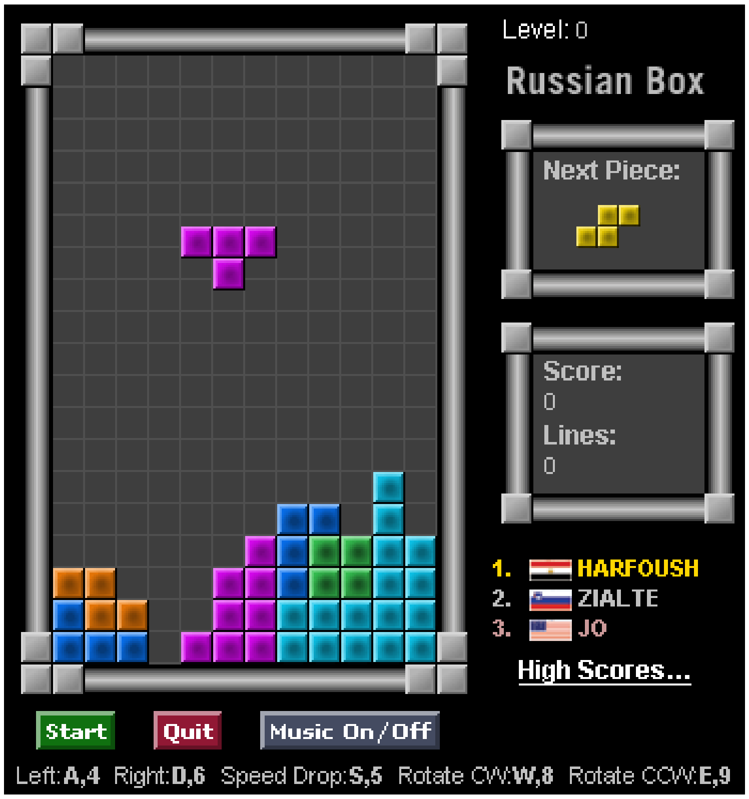


Figure 6.5: Tetris game simulator.

Chapter 7: Results

The main objective of this thesis is to understand the potential for parallel Monte Carlo Tree Search algorithms to speedup our Python based library. We have shown the performance of each of the algorithms; leaf parallelization, root parallelization, tree parallelization with local mutex and global mutex and block parallelization in the domains Connect4, Yahtzee and Tetris. Section 7.2 shows the overall results of these algorithms playing against each other in the domain Connect4 and individually in the other domains; Yahtzee and Tetris. The next section provides information about specific observations about each of these algorithms in subsection.

7.1 Experimental Set-up

Our experiments were run on an AWS machine with 32 cores Intel(R) Xeon(R) CPU E5-2680 v2 at 2.80GHz and 60 GB of memory. The Python interpreter version is 2.7.6. For our experiments, we tuned the parameters of each algorithm to run using all the available cores. Our initial experiment was to find the optimal UCT constant value for each of the domains under consideration. We ran the sequential UCT version by setting the UCT constant at the lowest reasonable value and gradually increased in specific factors until the point the curve begins to drop. We ran only the sequential UCT agent in the case of single agent domains like Yahtzee and Tetris with time limit per move as 1.0 second. The reward curve for various values of UCT constant in Yahtzee and Tetris are depicted in Figure 7.1 and 7.2 respectively.

1	2	4	6	8	16
23.08	32.45	38.87	**58.63**	51.88	48.06

Figure 7.1: UCT constant for Yahtzee.

1	5	10	15	20	25
236.43	257.52	263.88	292.12	**303.00**	297.72

Figure 7.2: UCT constant for Tetris.

In case of the domain Connect4, we set one of the agents to be sequential UCT with UCT constant $c = 0.5$. The second agent is also sequential UCT with varying values of UCT constant. We set the best value of UCT constant for Connect4 based on the win-ratio curve of the second agent for varying values of the UCT constant. We set the sequential UCT agent in all the domains to play 100 games in each constant setting with 1024 simulation count limit per move. The values of UCT constant in Connect4 are shown in Figure 7.3.

0.4	0.6	0.8	1.0	1.2
0.30	0.56	**0.92**	0.87	0.83

Figure 7.3: Win-ratio for UCT Constant - Connect4.

7.1.1 Implementation details

The configuration of every agent as declared in the *jobs.xml* file is given in the Figure 7.4.

```
# SEQUENTIAL UCT
<player number="5" uct_constant="0.8" horizon="100"
   time_limit="1.0"/>

# TREE PARALLEL - LM
<player number="8" threadcount="32" uct_constant="0.8"
   horizon="100" time_limit="1.0"/>

# LEAF PARALLEL - LP
<player number="9" num_threads="16" uct_constant="20"
   horizon="100" time_limit="1.0"/>

# BLOCK PARALLEL - BP
<player number="10" ensembles="16" threadcount="16"
   uct_constant="0.8" horizon="100" time_limit="1.0"/>

# ROOT PARALLEL - RP
<player number="6" ensembles="32" uct_constant="0.8"
   horizon="100" parallel="1" time_limit="1.0"/>

# TREE PARALLEL - GM
<player number="11" threadcount="32" uct_constant="0.8"
   horizon="100" time_limit="1.0"/>
```

Figure 7.4: Agent configurations for Connect4

The Tree parallel variant of MCTS involves in simultaneously accessing the same tree by all the working processes. The multiprocessing library in python provides the necessary API for our experiments. The disadvantage in multiprocessing is that the processes do not share the same memory. If we use IPC methods and communicate between the processes (32 processes in our case), it would add a huge overload for our simulations. So we used the Manager model provided by Python. In this model, one Manager process holds the data about the tree. All the read and write over the search tree is executed via the manager process. Python makes it simple enough for the developers to directly define the manager process as a class object and access it using its functions. Figure 7.5 depicts the structure of this implementation.

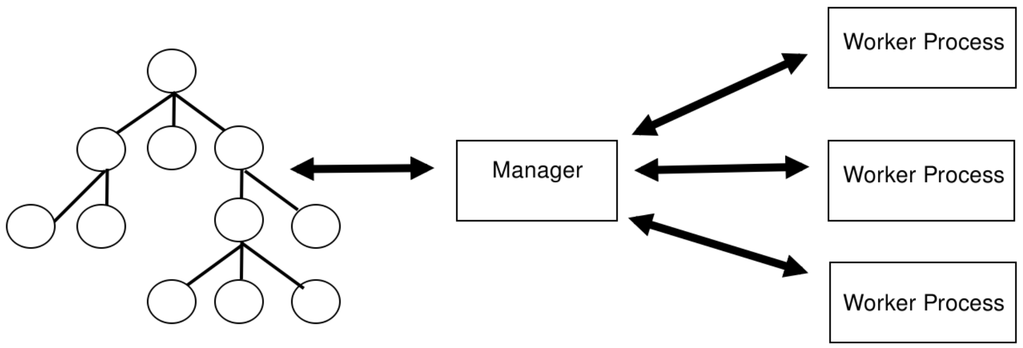


Figure 7.5: Manager process.

7.2 Overall Results

In this section we have given the results of all the simulations in various domains. A total of six agents are considered in our experiments; LP - Leaf Parallel, LM - Tree parallel with Local Mutex, GM - Tree parallel with Global mutex, BP -

Block parallel, RP - Root parallel and UCT - Sequential version of UCT. Figure 7.6 - 7.10 shows the performance of each agent against all the other in the domain Connect4. The time limit per move is set to be at 1.0 second and 2.0 seconds.

Figure 7.6 shows two tables for the agent LM in two different time setting respectively. The win-ratio for all the agents are presented in the same way. In each table, there is one row and five columns and each value indicates the win-ratio of the agent in the row of the cell against the agent in the column. The table is presented in a way to easily understand the comparison of a particular algorithm against all the other for a particular time setting.

A total of 100 games were played for each against all the other agents in Connect4 and 500 games for each agent in Yahtzee and Tetris. After the results of time limit per move set at 2.0 seconds, the agents do not show any improve in performance. Hence we did not conduct more experiments with higher value of time limits.

The results of all the simulations are shown in the Figures 7.6 - 7.12. Figure 7.13 shows the variation of simulation counts for each algorithm in a game of Connect4 and each value in a column shows the change of simulation counts as the game progresses to finish (from first row to the last).

7.2.1 Time and memory analysis of Tree parallelization

UCT, in general, does not impose a depth or time bound. As the given time for decision increases, the simulation count increases providing a better decision. In

	GM	RP	BP	LP	UCT
LM	0.56,0.43	0.0,1.0	0.04,0.96	0.02,0.97	0.04,0.96

	GM	RP	BP	LP	UCT
LM	0.54,0.45	0.0,0.99	0.05,0.93	0.14,0.84	0.02,0.98

Figure 7.6: Connect4 win ratio - LM. 1 second (Top) and 2 seconds (Bottom) per move.

	LM	RP	BP	LP	UCT
GM	0.43,0.56	0.0,1.0	0.02,0.96	0.04,0.96	0.03,0.97

	LM	RP	BP	LP	UCT
GM	0.45,0.54	0.0,1.0	0.03,0.96	0.89,0.07	0.02,0.96

Figure 7.7: Connect4 win ratio - GM. 1 second (Top) and 2 seconds (Bottom) per move.

	GM	LM	BP	LP	UCT
RP	1.0,0.0	1.0,0.0	0.84,0.16	0.8,0.2	0.89,0.11

	GM	LM	BP	LP	UCT
RP	1.0,0.0	0.99,0.0	0.93,0.07	0.97,0.01	0.78,0.18

Figure 7.8: Connect4 win ratio - RP. 1 second (Top) and 2 seconds (Bottom) per move.

	GM	LM	RP	LP	UCT
BP	0.96,0.02	0.96,0.04	0.16,0.84	0.46,0.54	0.47,0.53

	GM	LM	RP	LP	UCT
BP	0.96,0.03	0.93,0.05	0.07,0.93	0.63,0.32	0.60,0.37

Figure 7.9: Connect4 win ratio - BP. 1 second (Top) and 2 seconds (Bottom) per move.

	GM	LM	RP	BP	UCT
LP	0.04,0.96	0.97,0.02	0.2,0.8	0.54,0.46	0.42,0.54

	GM	LM	RP	BP	UCT
LP	0.07,0.89	0.84,0.14	0.01,0.97	0.32,0.63	0.42,0.54

Figure 7.10: Connect4 win ratio - LP. 1 second (Top) and 2 seconds (Bottom) per move.

	Score
LM	201.66 ±26.15
GM	196.93 ±23.79
RP	506.05 ±95.48
BP	348.34 ±58.43
LP	330.87 ±60.42
UCT	326.10 ±60.73

	Score
LM	214.75 ±38.27
GM	209.02 ±31.24
RP	538.92 ±93.71
BP	393.41 ±52.66
LP	360.81 ±58.93
UCT	341.22 ±66.51

Figure 7.11: Tetris scores. 1 second (Top) and 2 seconds (Bottom) per move.

	Score
LM	65.225 ±25.27
GM	65.04 ±24.68
RP	75.02 ±22.60
BP	67.335 ±25.82
LP	58.24 ±22.03
UCT	54.12 ±20.70

	Score
LM	84.73 ±26.23
GM	83.92 ±22.45
RP	112.51 ±21.77
BP	94.29 ±22.331
LP	67.44 ±25.37
UCT	62.07 ±22.702

Figure 7.12: Yahtzee scores. 1 second (Top) and 2 seconds (Bottom) per move.

this section, we study the usage of time and memory by the tree parallelization variant for a given time limit. The structure of a UCT node is shown in the Figure 7.14.

By executing $sys.getsizeof()$ on each node on the tree after construction (Connect4), we obtained the average size of a node as 75 bytes ($\pm 5 bytes$). In case of simulation counts nearing 700,000 (root parallel version), the size of the entire tree is less than 200 MB.

When communicating with the manager process, more objects are passed between the manager and the working processes as the simulation count increases. This adds up to a considerable amount in the time limit per move but not as high as the other parameters.

Figure 7.15 shows the split of time between all the activities performed by

RP	LM	LP	GM	UCT	BP
15854	148	1392	139	1192	456
18497	154	1344	140	1172	490
20319	153	1328	143	1371	509
21655	178	1312	140	1404	529
22466	184	1328	143	1461	553
24446	178	1328	142	1718	607
29618	203	1328	147	1745	568
29097	161	1312	140	1777	610
31100	152	1312	163	2436	
31715	181	1312	173	3131	
34729	178	1296	145	4162	
38783		1280	144	6010	
49877		1296	160	15091	
80154		1296	169	21960	
114651		12832	171	44059	
158019		27209	176	78964	

Figure 7.13: Comparison of simulation counts (Connect 4 - 1 second per move.)

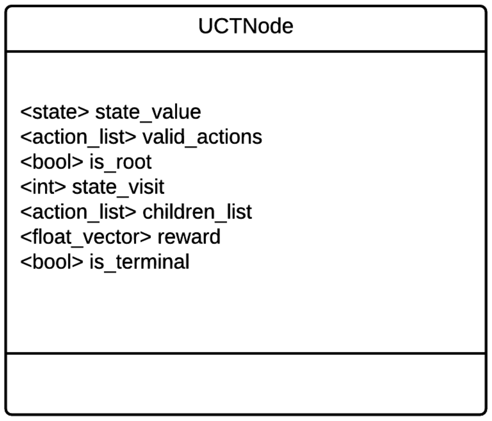


Figure 7.14: Structure of a UCT Node

all the processes in both the variants of Tree parallelization method. It could be observed that less than 25% of the given time per move is actually used for tree expansion which is the primary reason for the reduced simulation counts.

We calculated the process start time by measuring the time before the process start call and the time at which the first line of the worker process code and measuring the difference between them.

The wait time is measured by calculating the time difference between the time at which the process enters the section to obtain the lock and the section after acquiring the lock. The interprocess communication times are measured using the elapsed time for every call to the manager process. The calls to the manager are to lock and obtain a node data, lock and backtrack values, create a new node and finally to obtain the root node data.

Method	Total Time	start	wait	calls	tree
LM	1.0	0.197	0.437	0.118	**0.248**
GM	1.0	0.183	0.552	0.104	**0.161**

Figure 7.15: Tree parallel with Local mutex - Time analysis (32 processes)

7.3 Agent specific observations

Figure 7.13 shows the variation of simulation counts in a game of Connect4 for all the agents in our experiment. In this section, we discuss about the various observations of the performance of each agent in all the domains.

7.3.1 Root Parallel

From the Figures 7.8, 7.11, 7.12 we could observe that Root parallel version performs better than all the variants. The unique characteristic of root parallel variant that avoids local optimum as a result of constructing separate trees in parallel from the given input state is effective in this case.

By using the optimal setting for UCT constant, all the trees generated in the root parallel variant gives useful information for the decision at the root node. We are constructing 32 trees in parallel (1 process / core / tree). There are no communication between each of the processes.

Given a limited amount of time, root parallel clearly outperforms all the other variants and hence we could easily observe that this is the best form of parallelization.

7.3.2 Tree Parallel with Local Mutex

The performance of Tree parallel variant with local mutex is shown in the figures 7.6, 7.11, 7.12 on the domains Connect4, Tetris and Yahtzee respectively. The local mutex version involves in locking and releasing various sections of the tree frequently.

From the Figure 7.13 we could observe that the simulation counts are very low compared to all other variants and it performs bad compared to the sequential UCT version. The detailed analysis of the time per move usage is given in the previous Section 7.2.1.

The repeated calls to the manager process for node selection, process initiation and waiting time for locks adds up to more than 70% of the total time given per move (Section 7.2.1). Also, when a process selects a node and releases the lock, the next process entering the node selects the same next node in the trajectory. This adds up a lot of redundant simulations.

However, for simulators with longer execution times (Yahtzee), this method performs nearly equal to that of other methods. The effect of added time for activities other than tree expansion is low when using simulators with longer execution times. Hence, the local mutex method performs better than in the domain Connect4.

7.3.3 Tree Parallel with Global Mutex

The global mutex variant of Tree parallelization involves in locking the whole tree until the current process working on the tree enters the simulation phase. We could observe that a lot of time is used up in busy waiting until the current process enters the simulation phase unlike the local mutex version where the next process enter after the current process is done with selection. This accounts for the reduced number of simulations as shown in Figure 7.13.

This version also exhibits the same disadvantages as the local mutex version and performs relatively bad in case of the fast simulator domain Connect4 as shown in Figure 7.7.

However, in case of Yahtzee, the same effect of local mutex method could be observed where the simulator takes comparatively longer for each simulation and hence performs better (Figure 7.11 and 7.12).

7.3.4 Block Parallel

Block parallelization is a combination of root and leaf parallelization. Hence the construction of multiple trees in parallel prevents it from occasional rollouts and the effect of the constant setting. But we also execute parallel rollouts at the leaf node.

For each tree we execute 16 rollouts in parallel and also we have 16 trees constructed in parallel. This is the reason for the reduced simulation counts as shown in Figure 7.13. For 1.0 second per move time setting, block parallel does not execute enough simulations to make a better decision and hence it performs bad when compared to leaf parallel (Connect4 - Figure 7.9).

However with 2.0 seconds per move, the simulation count per tree increases which enables a better decision while aggregating the root node data from all the trees. This is observed in Connect4 - Figure 7.9 (BP vs. LP - 1.0 second and 2.0 seconds).

Also the sequential version of UCT performs better compared to block parallel variant in Connect4 as seen in Figure 7.9. The simulation count of sequential UCT increases in large factors after few moves down the game compared to leaf parallel.

However, block parallel clearly shows a better performance in Yahtzee and Tetris because of longer simulator execution times and the ensemble effect of avoiding a local optima. In such cases, the performance increases with increase in time per move. But the block parallel variant performs better than the tree parallel variants in all the time settings.

7.3.5 Leaf Parallel

Figure 7.10, 7.11 and 7.12 depicts the performance of leaf parallel variant of UCT parallelization compared to other agents. Leaf parallel agent is executed similar to the normal version of UCT except at the leaf node where multiple rollouts are executed in parallel.

In our experiments, we have set the leaf rollout count as 16. So during the simulation phase, 16 rollouts are executed in parallel. The leaf parallel version performs nearly the same as the sequential version and shows no improvement because of the condition that the simulation phase in leaf parallelization is done only when the longest rollout of all the 16 is finished. So when the game is near the end, we have many redundant simulations executing in parallel. At such stages, waiting for two such parallel rollouts could give better information than all 16 rollouts.

Despite these effects, we could observe that leaf parallel agent performs better than the Tree parallel agents where the time spent for steps excluding tree expansion is relatively more as shown in Figure 7.10, 7.11 and 7.12.

7.4 Discussions

The root parallel version performs better than all the variants of parallelization of MCTS. Given a fixed amount time, root parallelization turns out to be the effective way to parallelize Monte Carlo Tree Search. The win-ratio of root parallel method in the domain Connect4 remains near 1.0 against all the other methods.

For simulators with longer execution times, root parallel variant led to a better final score in the domains of Yahtzee and Tetris compared to the other methods.

Tree parallel methods exhibit bad performance compared to other methods. But this is because of the added redundancy with multiprocessing which resulted due to the absence of true multithreading in Python. In case of yahtzee, tree parallel method performs nearly equal to other methods. Hence the actual performance of the tree parallel method is inconclusive.

Block parallel method, as a combination of root parallel and leaf parallel methods, performs better with increase in time per move. With more time per move, block parallel method clearly wins against leaf parallelization.

Leaf parallel performs better than tree parallel only in cases of fast simulator domains. However, in most of the cases, leaf parallel performs marginally better or equal to that of the sequential method.

Chapter 8: Summary

We have evaluated the performance of five parallelization techniques of Monte Carlo Tree Search using the open-source Monte Carlo planning library PyPlan; Root parallelization, Leaf parallelization, Tree parallelization with local mutex, Tree parallelization with global mutex and Block parallelization. These experiments were conducted on three domains; Connect4, Yahtzee and Tetris.

In this thesis, we have introduced a new Python based Monte Carlo planning library PyPlan which enables easier and efficient prototyping of planning problems in Python. The library offers easier ways to use various built-in planning agents and simulators and also extend it efficiently. However, the characteristics of the underlying design of the Python language imposed various constraints in implementation of certain parallel methods which may not be suitable for certain domains with real-time constraints.

These constraints led us to study the characteristics and evaluate the various parallelization methods to improve the performance of our library. Our findings indicate that the root-parallel method of parallelizing MCTS in PyPlan performs better than all the other methods in domains with real-time constraints. Also, the block-parallel method performs better in domains with relaxed time limits. Results about tree-parallel method are inconclusive because of all the added time in redundant operations other than tree expansion.

Future work could explore the effect of better ways of working around such implementation constraints caused by Python and determine the cases in which other methods could perform as equal or better than the root parallel method.

Bibliography

[1] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game ai. In *AIIDE*, 2008.

[2] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[3] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *Computers and Games*, pages 60–71. Springer, 2008.

[4] Alan Fern and Paul Lewis. Ensemble monte-carlo planning: An empirical study. In *ICAPS*, 2011.

[5] Kamil Rocki and Reiji Suda. Parallel monte carlo tree search on gpu. In *SCAI*, pages 80–89, 2011.

[6] Cameron B Browne, Edward 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. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[7] Markus Enzenberger and Martin Müller. A lock-free multithreaded monte-carlo tree search algorithm. In *Advances in Computer Games*, pages 14–20. Springer, 2010.

[8] Alan Fern. CS 533 Intelligent Agents and Decision Making. Lecture topics and Reading. http://web.engr.oregonstate.edu/ afern/classes/cs533/lectures.html.

[9] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[10] James Glenn. An optimal strategy for yahtzee. *Loyola College in Maryland, Tech. Rep. CS-TR-0002*, 2006.

[11] Matthew Crumley. Regular Yahtzee Game Rules / Instructions. http://yahtzee-rules.com/, 2011.