mctreesearch4j: A Monte Carlo Tree Search Implementation for the JVM

Larkin Liu[1] and Jun Tao Luo[2]

1 University of Toronto **2** Carnegie Mellon University

Summary

We introduce *mctreesearch4j*, a Monte Carlo Tree Search (MCTS) implementation written as a standard JVM library following key object oriented programming design principles. This modular and extensibile implementation of MCTS provides a powerful tool to enable the discovery of approximate solutions to complex planning problems via rapid experimentation. *mctreesearch4j* utilizes class inheritance and generic types to standardize custom algorithm definitions. In addition, key class abstractions are designed for the library to flexibly adapt to any well-defined Markov Decision Process (MDP) or turn-based adversarial games. Furthermore, *mctreesearch4j* is capable of customization across a variety of MDP domains, consequently enabling the adoption of MCTS heuristics and customization into the core library with ease.

Statement of Need

Open source implementations of Monte Carlo Tree Search exist, but have not gained widespread adoption. Implementations of MCTS have been written for Java Virtual Machine (JVM), C/C++, and Python, yet many do not provide easy access to heuristics, nor do they implement extensible and modular state action space abstractions ([Cowling et al., 2012](#)) ([Lucas, 2018](#)). *mctreesearch4j* aims to build on the performance advantages of a compiled language families, such as JVM, while simultaneously being extensible and modular. Therefore, many improvements and research experiments could be performed by extending and modifying the capabilities of the base software. *mctreesearch4j* is designed to enable researchers to experiment with various MCTS strategies while standardizing the functionality of the MCTS solver and ensuring reliability, where the experiments are reproducible and the solver is compatible with common JVM runtimes.

In the domain of probabilistic planning, extensible, reproducible, and modular software packages do exist in the solution space of Reinforcement Learning. We take inspiration from software libraries such as Acme ([Hoffman et al., 2020](#)) and Tensorforce ([Schaarschmidt et al., 2018](#)), notably from its design concept. Such libraries function as a domain independent software library to generate solutions for a variety of MDP's. The MDP's are defined separate from the core library, and communicate via a common interface. Such a design concept has also been incorporated to *mctreesearch4j* within the solution space of MCTS for probabilistic and deterministic planning.

Monte Carlo Tree Search

Monte Carlo Tree Search primarily makes use of a deterministic selection of actions and stochastic simulations to estimate the reward function of well defined Markov Decision Process

---

(MDP). MCTS is a tree search adaptation of the UCB1 Multi-Armed Bandit Strategy (Auer et al., 2002). In more detail, the MAB does not change state with each action taken, and is primarily focused on optimizing exploration - exploring new actions - versus exploitation - obtaining rewards from known actions. MCTS, is the extension, where we optimize for exploration vs. exploitation for dynamic states contingent on any actions taken in the action space. We present a more detailed discussion in (Liu & Luo, 2021).

The MCTS algorithm is distinctly divided into 4-phases, *Selection*, *Expansion*, *Simulation*, and *Backpropagation*, which are illustrated in Fig. 1. In *Selection*, a policy deterministically selects which action to play, based on previously expanded states, guided by some exploration constant, C. In the *Expansion* phase, states that are unexplored, represented by a leaf node, are added to the search tree. Subsequently, in the *Simulation* phase, a simulation is stochastically played out. Finally, *Backpropagation* propagates the final reward of either a terminal state, or a node at a specified depth limit, back to the root node. This 4-phase process is repeated until a maximum number of iterations or a convergence criteria is established.
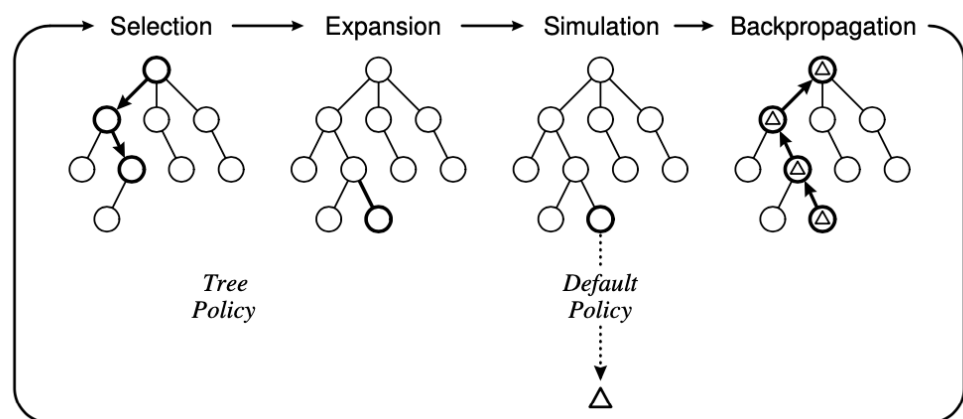

Figure 1: Monte Carlo Tree Search Key Mechanisms. (Browne et al., 2012)

Design Principles

mctreesearch4j is designed follow three key design principles.

- **Adaptability**: Adaptability is defined as the ability for MDP domain to be easily integrated into the *mctreesearch4j* framework using provided class abstractions. Our implementation seeks to simplify the adoption of MCTS solutions for a variety of domains.
- **JVM Compatibility**: We design a library that is fully compatible with the Java Virtual Machine (JVM), and consequently functional with any JVM languages, ie. Java, Scala, Kotlin etc.
- **Extensibility** We design to achieve a high degree of extensibility and modularity within the framework. Extensibility is defined as the ability for key mechanisms to be reused, redefined, and enhanced, without sacrificing interoperability.

mctreesearch4j serves as a framework to standardize the development of Monte Carlo Tree Search development in a research setting, while simultaneously serving as a production-ready software for deployment of MCTS for the JVM ecosystem. *mctreesearch4j* is specifically designed to be extensible, standardized, and modular. This standardization is key to enabling scientists and engineers to better reproduce results of AI research, and furthermore reduce the iteration speed of research into novel algorithms in AI.

Domain Abstraction

The main abstraction that is used to define an MDP problem is the abstract class defined in *MDP Abstraction*, using generic type variables. Each of the methods correspond to specific behaviour of a discrete MDP. In *mctreesearch4j* we use generic types `StateType` and `ActionType` to represent the MDP states and actions respectively. This abstract class has five members that must be implemented. These abstract class methods define the functionality of an MDP. The MDP abstraction will be used by MCTS solvers to compute the optimal policy. The MDP interface can be written in any JVM language, we use Kotlin and Scala for this paper, with the Scala implementation from (Liu, 2021).

MDP Abstraction

```kotlin
abstract class MDP<StateType, ActionType> {
    abstract fun transition(StateType, ActionType) : StateType
    abstract fun reward(StateType, ActionType?, StateType) : Double
    abstract fun initialState() : StateType
    abstract fun isTerminal(StateType) : Boolean
    abstract fun actions(StateType) : Collection<ActionType>
```

Solver Design

mctreesearch4j provides a default implementation known as `GenericSolver`, and an alternate `StatefulSolver`. The abstract `Solver` serves as the base class for both versions, and defines a set of functionalities that all solver implementations must provide as well as a set of extensible utilities. Similar to the MDP abstraction, the solver uses a set of type parameters to provide strongly typed methods that unify and simplify the solver implementation. The programmer is free to select the data type or data structure that best defines how states and actions are represented in their MDP domain.

The difference between solvers lies in their respective memory utilization of abstract node types to track states during MCTS iterations. The default `GenericSolver` provides a leaner implementation, where actions are tracked and no explicit states are stored permanently. The states tracked with `GenericSolver` are dynamic and the MDP transitions must be rerun when traversing the search tree during selection in order to infer the state. The `StatefulSolver` keeps an explicit record of the visited states, and additional information, such as terminality and availability of downstream actions. The extra overhead of storing the state explicitly in the MCTS node, allows the algorithm to optimize its search using information from previously visited states. This is particularly useful for deterministic games, where a re-computation of the MDP transition is not necessary to determine the state of the agent after taking a specific action. This results in different implementations of the *Selection* step, while maintaining identical implementations of *Expansion*, *Simulate* and *Backpropagation*.
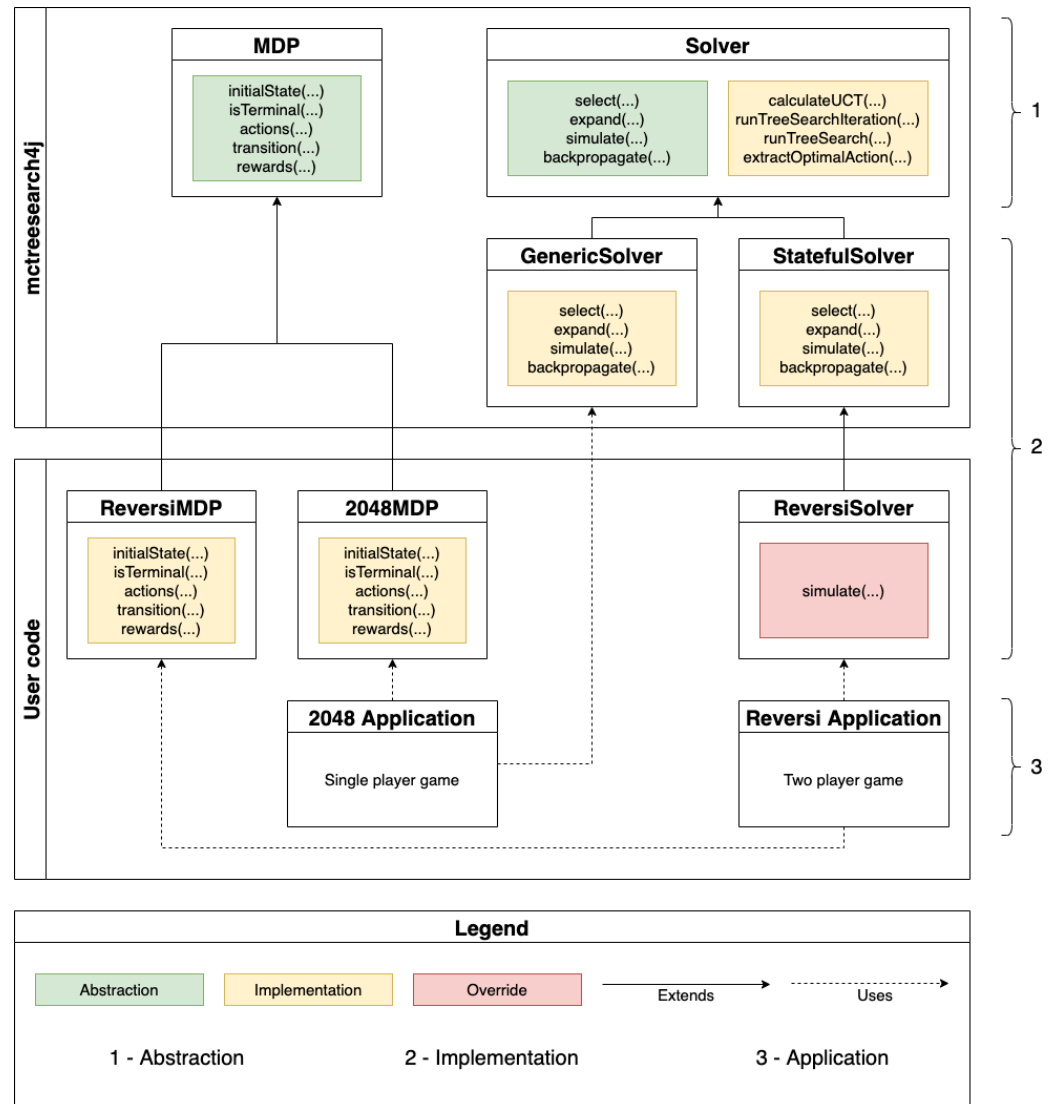
Figure 2: Software Architecture.

Customization

mctreesearch4j maintains a high degree of modularity. For example, the key mechanisms highlighted in Fig. 1 can be rearranged, allowing scientists to experiment with different versions of MCTS. Furthermore, the internal functionality of each of the key mechanisms can be redefined. And complex meta heuristics and search algorithms can be implemented to add specific advantages to the solver for specific MDP domains. All of these features are important to give researchers the flexibility to design bespoke MCTS algorithms for their respective research.

Though the default MCTS implementation works well in many scenarios, there are situations where knowledge about specific problem domains can be applied to improve the MCTS performance. Improvements to MCTS, such as heuristics driven simulation, exploit domain knowledge to improve solver performance. We demonstrate that a Reversi AI that uses heuristics derived from (Guenther & Klinik, 2004) is able to outperform the basic MCTS implementation contained in *mctreesearch4j*. These heuristics are programmed via extensibility points

in the *mctreesearch4j* solver implementation, where the key mechanisms can be altered or augmented. In our Heuristic Implementation Example, we introduce the `heuristicWeight` array, a 2D array storing domain specific ratings of every position on a Reversi board representing the desirability of that position on the board. The negative numbers represent a likely loss and positive numbers representing a likely win, again as represented in Fig. 3.

This value is taken into consideration when traversing down the simulation tree. The `heuristicWeight` array adjusts the propensity to explore any position for both agents based on the heurisitc's belief about the desirability of the position. To alter the MCTS simulation phase we override the `simulate()` method and create a new definition for it. The application of this `heuristicWeight` only requires minor alterations to the `simulate()` method, as illustrated in *Heuristic Implementation Example*.

Heuristic Implementation Example

```kotlin
override fun simulate(node: NodeType): Double {
    /*... Original Simulation code ...*/
    while(true) {
        val validActions = mdp.actions(currentState)
        var bestActionScore = Int.MIN_VALUE // Begin heuristic
            var bestActions = mutableListOf<Point>()
            for (action in validActions) {
                val score = heuristicWeight[action.x][action.y]
                if (score > bestActionScore) {
                    bestActionScore = score
                    bestActions = mutableListOf(action)
                }
                if (score == bestActionScore) {bestActions.add(action)}
            }
        val randomAction = bestActions.random() // End heuristic
        val newState = mdp.transition(currentState, randomAction)
        /*... Original Simulation code ...*/
    }
}
```
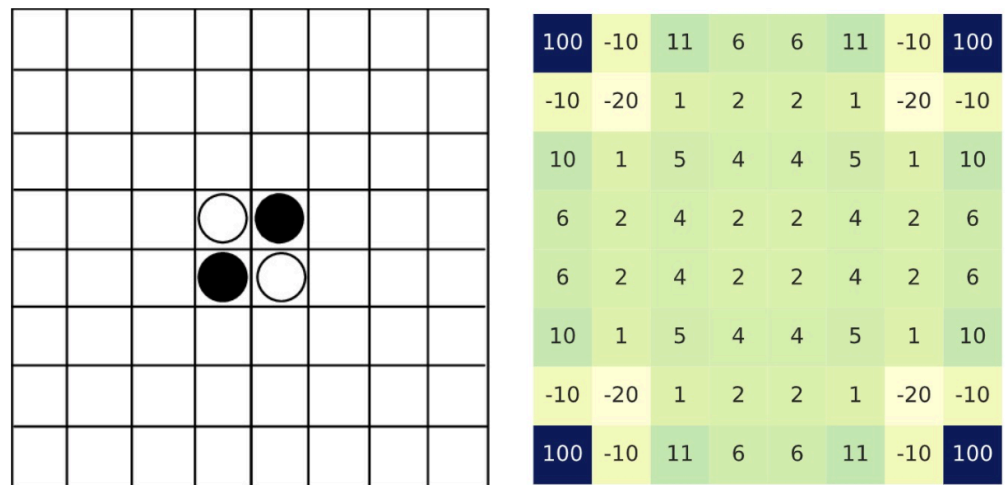

Figure 3: Reversi Heuristic.

Liu et al., (2022). mctreesearch4j: A Monte Carlo Tree Search Implementation for the JVM. *Journal of Open Source Software*, 7(70), 3804. 5
https://doi.org/10.21105/joss.03804

Results

When the MCTS solver is accurately selecting the optimal solutions, it will continue to compel the agent to explore in the optimal subset of the state space, and reduce its exploration in the non-optimal subset of the state space. We provide a quick example in the MDP Domain of GridWorld, detailed in (Liu & Luo, 2021). The cumulative number of visits corresponding to the optimal policy is proportionally increasing with respect to the number of MCTS iterations. Whereas for non-optimal solutions, the cumulative visits are significantly lower because the solver will avoid visiting the non-optimal state subspace.
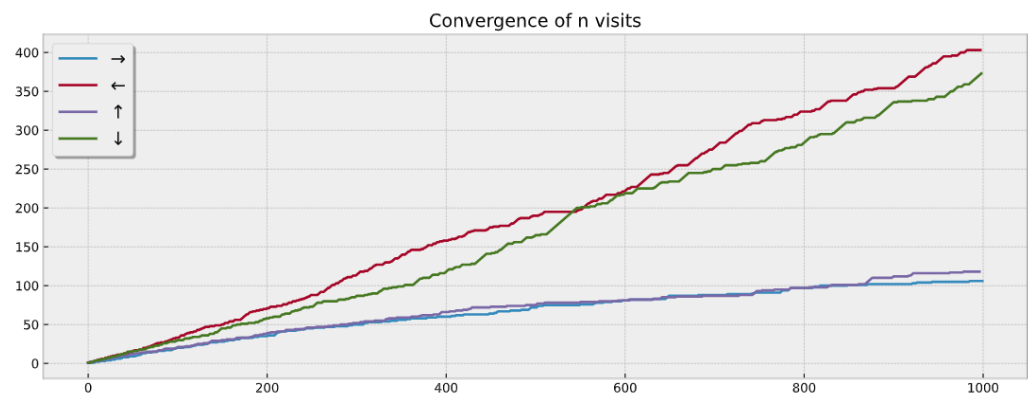

Figure 4: Convergence of visits.

Real World Applications

mctreesearch4j is designed to run on the JVM, and therefore is useful in a research setting where JVM is intended to be used alongside either the implementation of the game or the core AI engine. Typically, for state-of-the-art AI, MCTS is used in combination with other Deep Learning algorithms, where MCTS drives the exploration of hypothetical games to be played out in order for the agent to make better decisions.

MCTS is typically combined with Deep Learning strategies to achieve state-of-the-art performance for complex games whose solutions are typically intractable via standard tree-search methods. For example in AlphaGo [silver:2016], MCTS is combined with two types of neural networks, namely value networks and policy networks, to achieve state-of-the-art performance in the game of Go, never achieved before. Similarly, in the area of control theory, MCTS acting in combination with Value and Policy Networks to produce state and action space distributions, have shown impressive performance in probabilistic planning for self-driving cars, as demonstrated in Tesla's AI day event [tesla:2021]. The utility of MCTS illustrates the acceptance of MCTS into the standard repetoire of both probablisitc and deterministic planning. And as the need for MCTS grows among industry and research, the impetus to create MCTS software libraries that are standardized, reliable, and can integrate well into general software ecosystems, such as JVM, is increasing.

One key industry application of *mctreesearch4j* is the fact that it is a JVM software, which allows it to integrate into both legacy and modern JVM software. One key application can be in mobile gaming, which has a large developer base in Android, largely written in Kotlin within the JVM. *mctreesearch4j*'s lightweight design with guaranteed reliability and JVM compatibility, tested extensively on both Java 8 and Java 11, make it a prime candidate for robust AI in turn-based games on mobile devices.

Future Contributions

Future work on *mctreesearch4j* could potentially involve further optimization, particularly of the memory consumption of the algorithm. Additional speed improvements can be improved via multi-threading and parallelization, which are fully supported on the JVM.

The fact that *mctreesearch4j* is based in the JVM can however also serve as a downside, since Java based programming languages are not typically used for research problems within AI. That, combined with JVM language's added complexity when compared to scripting languages such as Python, can result in *mctreesearch4j* experiencing a slow adoption in a pure research setting. Nevertheless, the key design principles that *mctreesearch4j* was designed on, could easily be ported into another scripting based language, more commonly found in AI research, namely Python. *mctreesearch4j* is designed to serve as a foundational template for research in MCTS for the JVM, and it can also serve as a reliable importable library for JVM applications which implement MCTS.

Conclusion

In closing, *mctreesearch4j* presents a framework which enables programmers to adapt an MCTS solver to a variety of MDP domains. This is important because software application was a main focus of *mctreesearch4j*. Furthermore, *mctreesearch4j* is fully compatible with JVM, and this design decision was made due to the excellent support of class structure and generic variable typing in Kotlin, and other JVM languages, as well as support for mobile applications. Yet most importantly, *mctreesearch4j* is modular and extensible, the key mechanism of MCTS are broken down, and the programmer is able inherit class members, redefine and/or re-implement certain sections of the algorithm while maintaining a high degree of MCTS standardization.

Acknowledgements

References

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, *47*(2-3), 235–256. https://doi.org/10.1023/A:1013689704352

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P. I., Tavener, S., Perez, D., Samothrakis, S., Colton, S., & al., et. (2012). A survey of monte carlo tree search methods. *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI.*

Cowling, P., Powley, E., & Whitehouse, D. (2012). Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and Ai in Games*, *4*, 120–143. https://doi.org/10.1109/TCIAIG.2012.2200894

Guenther, M., & Klinik, J. (2004). *A new experience: O-thell-us – an AI project.* https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/O-Thell-Us/Othellus.pdf; University of Washington.

Hoffman, M., Shahriari, B., Aslanides, J., Barth-Maron, G., Behbahani, F., Norman, T., Abdolmaleki, A., Cassirer, A., Yang, F., Baumli, K., Henderson, S., Novikov, A., Colmenarejo, S. G., Cabi, S., Gulcehre, C., Paine, T. L., Cowie, A., Wang, Z., Piot, B., & Freitas, N. de. (2020). Acme: A research framework for distributed reinforcement learning. *arXiv Preprint arXiv:2006.00979*. https://arxiv.org/abs/2006.00979

Liu, L. (2021). Connect 4 implementation in scala. In *GitHub repository*. https://github.com/larkz/connect4-scala, commit = 4549c00398e7987814d83a2bd0760bbaedeb879b; GitHub.

Liu, L., & Luo, J. T. (2021). An extensible and modular design and implementation of monte carlo tree search for the JVM. *Preprint*. https://doi.org/doi:10.20944/preprints202107.0622.v1

Lucas, S. (2018). MonteCarloTS - java. In *GitHub repository*. https://github.com/DrumerJoe21/MonteCarloTS; GitHub.

Schaarschmidt, M., Kuhnle, A., Ellis, B., Fricke, K., Gessert, F., & Yoneki, E. (2018). LIFT: Reinforcement learning in computer systems by learning from demonstrations. *CoRR*, *abs/1808.07903*. http://arxiv.org/abs/1808.07903