NPC engine: a High-Performance, Modular, and Domain-Agnostic MCTS Framework for Emergent Narrative

Stéphane Magnenat, Henry Raymond, David Enderlin, Sven Knobloch, Robert W. Sumner

Department of Computer Science, ETH Zurich
stephane.magnenat@inf.ethz.ch, henry.raymond@inf.ethz.ch, robert.sumner@inf.ethz.ch

Abstract

The quality of emergent narrative is strongly linked to the capabilities of the underlying simulation. A powerful way to drive character activity is to use a performant and dependable planner. In this paper, we present a domain-agnostic multi-agent Monte Carlo tree search planner implemented using the Rust programming language. The planner supports tasks of varying duration and a dynamic number of agents with heterogeneous properties. It also provides optional concurrency, allowing for scalable simulations of many agents planning in parallel on multiple threads. In addition to simulation-based rollout, it supports custom state value estimators and offers a basic adaptive implementation using neural networks. The planner also includes a variety of debugging features, such as the ability to plot the search tree. For easy adoption, we provide several documented application examples.

As defined by Ryan (2018), emergent narrative is "narrative that emerges out of computer simulation of character activity, [...]". The quality of emergent narrative is therefore inherently linked to the believably and sensibleness of character behavior within the simulation. Characters need to pick appropriate actions for the given circumstances, a task optimally performed by a planning algorithm. As a result, the strengths of the utilized planning algorithm fundamentally determine the apparent intelligence of agents, and in turn defines the applicability of a simulation to emergent narrative.

The past decade has seen tremendous improvements in the capabilities of planning algorithms and other AI agent control techniques, but only in specific applications. The most noteworthy examples – reaching super-human performance – all combine multiple techniques. For example, the best board game players AlphaGo, AlphaZero and MuZero combine Monte Carlo Tree Search (MCTS) planning and deep learning (Schrittwieser et al. 2020), while the best Starcraft 2 player AlphaStar combines reinforcement learning and neural networks (Vinyals et al. 2019). These successes were achieved by large teams focused on specific types of games. So far, this progress has not translated to a similar increase of agent intelligence in more general applications.

We believe a key reason to be the difficulty of developing industry-grade AI algorithms. For instance, implementing a correct and efficient MCTS planner is demanding, as debugging search algorithms is tedious due to large state spaces needing inspection. Moreover, efficiency and ease of use typically conflict with domain expressiveness. Small bugs can easily nullify the intelligence gain that the planner would otherwise offer. At times, bugs can be so hard to identify that they may lead to inaccurate research results.

The time invested into the development of reliable custom planners presents a significant cost that likely holds back research. This is especially true in the field of emergent narrative, where planning is central to character behavior. We believe that a generic and reliable planning framework would facilitate and encourage exploring new research questions in this field.

Related work

Traditionally, narrative planning is performed on the whole storyworld at once, for instance with a goal satisfaction planner (Ware and Young 2014). However, recently, there has been a growing interest in using optimization-based MCTS planners. In the context of murder puzzle story generation, Kartal and colleagues plan for the whole world but use MCTS in an iterative fashion, optimizing a believability metric and stopping when the goal is reached (Kartal, Koenig, and Guy 2014). More recently, taking a distributed approach, Jaschek and colleagues have explored per-agent MCTS planning (Jaschek et al. 2019). In that work, agents plan over a world described in linear logic, each independently optimizing their own reward, aiming at *belongingness and love*, *self-esteem and respect* and *self-actualization*. The murder mystery emerges from the interaction between these agents.

In this paper, we propose a generic software framework aiming at providing a solid foundation on which specific domains, such as world simulations or linear logic, can be implemented without having to worry about the implementation details of MCTS. There have been attempts to provide a general framework for tree-search–based AI players for some time (Pell 1996). A recent example is *general board game* (Konen 2019), that defines common interfaces for board games, game states and their AI agents. It offers multiple algorithms, including MCTS and temporal-difference reinforcement learning. However, all agents execute actions in turn and each action must last exactly one tick, hindering the applicability of this framework beyond board games. It

is written in Java and its abstractions are based on dynamic dispatch.

In the last few years, the Rust programming language[1] – which focuses on performance and memory safety – has increasingly been selected as the language of choice to build complex and dependable software (Fulton et al. 2021). In that context, several projects[2] have leveraged the rich type system offered by Rust to implement re-usable MCTS algorithms using generics. However, all focus on domains where a static number of agents have actions of equal duration. This lack of dynamism hinders the use of these planners in more complex domains, which is a requirement for many emergent narrative applications.

Contributions

We present a generic and modular implementation of a domain-agnostic multi-agent MCTS planner. Our implementation in Rust is self-contained, efficient, and integrates seamlessly with machine learning methods. It supports advanced features such as actions of varying duration and a dynamic number of agents over time. It also provides optional concurrency, allowing scalable simulations of many agents on multiple CPU cores. It is available at github.com/ethz-gtc/npc-engine. The repository also contains examples showcasing the use of the planner in a variety of domains.

Design

Figure 1 shows a simplified diagram of the core software architecture of our planner. It is built around a Rust struct `MCTS` that is generic over a trait[3] `Domain`, implemented as compile-time user code. Through static functions, the `Domain` allows the planner to retrieve the visible agents, obtain their possible actions, and compute the current value of a given agent in a specific state.

The actual state that is explored in the search tree is decomposed into two elements: a base `State` which is constant over the search tree, and a `Diff` that changes per node – depending on the executed actions – and represents the difference to the base state. These are implemented as associated types[4]. This split minimizes the amount of data copied and stored at each node. Combined with the use of generics for domain definition which allows for compile-time optimization, this design offers excellent performance.

The planner supports an arbitrary number of possibly-heterogeneous agents (for example a global bookkeeping agent along regular agents). It also supports the appearance and disappearance of agents, even within a single planning tree (for example a domain in which agents are born and die). As a result, the planner is usable to implement rich and complex domains without any change to the core algorithm.

The agents' actions (abstracted by a `Task` trait in the code) can last a variable length of time, specified as a number of abstract ticks. This length can change in function of the agent and the state. The user can write their own actions by implementing the `Task` trait. After execution, an action can return a subsequent forced action, meaning that the agent will not perform any planning at that step.

The planner keeps a queue of agents ordered by the tick in which their current action ends, hence the order of agents is not necessarily round-robin: a single agent might select multiple actions in a row, if other agents are performing longer-term actions. Planned-for actions might become invalid when they are about to be executed. In this case, the user can define the outcome: prune that action's subtree (as in board games), or re-plan (as in simulations).

During planning, in the MCTS expansion step, a new node is created and an expected value assigned to each agent. By default, the planner uses a standard simulation-based rollout. However, this can be substituted by custom state value estimators, for example based on machine learning. Our framework includes a basic neural network implementation with leaky ReLU activation function.

A planning domain is unlikely to be error-free from the start, and debugging tree algorithms is notoriously difficult. To aid in that task, we provide multiple debugging features including tracing the execution using Rust's logging functionality and plotting search trees as graphs[5] (Figure 2).

Planners are often used as part of a world update loop. To support that, we provide two `Executor` helpers: one for sequential planning and execution, as well as one for parallel multi-threaded planning and execution. For the latter, the planning time itself is reflected in the search tree through a built-in `Planning` task. Combined with each agent planning on a limited spatial and conceptual horizon, this approach allows for linearly scalable multi-agent simulations.

Examples and Experiments

Our planner includes several examples to illustrate its use and key capabilities. These can be run with the command `cargo run --example NAME` where `NAME` can be:

- `tic-tac-toe`: a simple implementation of the board game with command line input to play against the AI.
- `capture`: an AI vs AI capture-the-flag competition on a topological graph. This domain demonstrates actions of varying duration, a bookkeeping agent that replenishes collectibles, disappearance of agents upon death, and the use of the single-threaded executor utility. It also showcases emergent strategies, such as weakening the enemy agent at the right time in order to later kill it, optimizing long-term score.
- `learn`: a resource-collecting agent that plans with a low number of visits and uses a neural network for state evaluation during expansion, showcasing progressive self-improvement. This is achieved by using the root state and the post-planning root node value as training data for

[1] www.rust-lang.org

[2] For example, the mcts, arbor and board-game libraries.

[3] A *trait* in Rust corresponds to an *interface*, when used for dynamic dispatch, or to a C++ *concept*, when used in generics.

[4] A generic in Rust, used as an output type of a trait.

[5] Using graphviz's dot format: graphviz.org/doc/info/lang.html

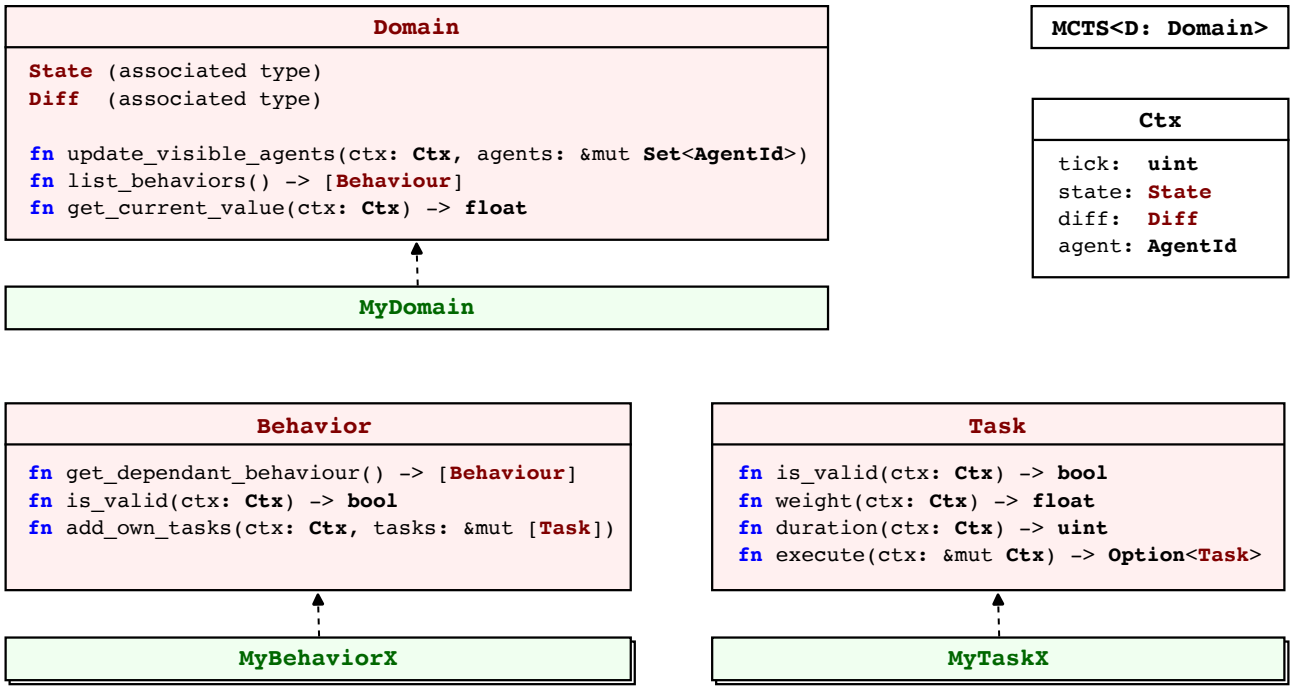

Figure 1: A simplified diagram of the core software architecture of the planner. Types are in bold font. Rust traits provided by the framework are in red, and user-defined structs implementing these traits are in green.

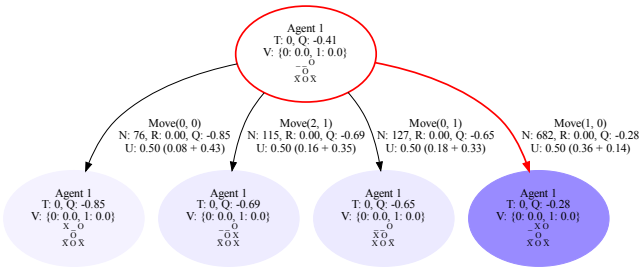

Figure 2: Example of graph debug output: the first layer of the search tree for tic-tac-toe at turn 3, including user-defined state visualization.

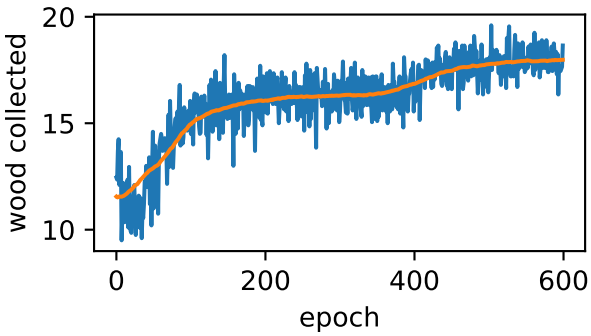

Figure 3: Self-improvement of a resource-collecting agent by training a neural network with previous root node values.

back-propagation–based learning (Figure 3). This simulation shows that over the course of 500 epochs, the performance of the agent (the amount of resources collected within a given period) improves by more than 50 %. This shows the value of combining heuristic-guided combinatorial search with machine learning, as seen in state-of-the-art AI research.

- `ecosystem`: a multi-threaded predator-prey simulation where many agents with limited horizon plan in parallel. This builds on our generic mechanism for extracting a local state out of a global state, and shows that our framework can scale to large simulations and take advantage of multi-core architectures. In several instances, we observed uninvolved people describing small stories such as “the tiger stalked the cow until it was trapped”. This shows that our system is rich enough to provide the base material for meaningful curation of emergent narrative.

Finally, the `scenario-lumberjacks` folder comprises a full experimental setup for running emergent theory of mind experiments, as described in our previous study (Raymond et al. 2020).

Discussion

Our work aims to be a solid framework on which the research community and the game industry can explore the field of emergent narrative. Though the game industry has traditionally used game engines based on C# and C++, Rust is increasingly being adopted, led by capable game engines

such as Bevy[6] and an ecosystem of companies[7]. Moreover, Rust integrates with traditional languages[8], allowing for planning logic to be implemented in Rust within an existing game engine.

In the future, it would be valuable to re-implement some domains from the literature using our framework, such as Jaschek et al. (2019). However, many papers do not offer enough details to do so. We hope that releasing our work as open-source software can foster such initiatives.

Currently, our action execution model is deterministic. This is a limitation, and we plan to add the possibility of multiple outcomes with associated probabilities. This could be implemented by inserting, after each action execution, a stateless dispatch node whose children are the possible future states – the links holding the probabilities of each outcome.

Conclusion

We provide an open-source, generic MCTS planner framework in Rust, a modern performance-oriented programming language. Focusing on scalable multi-agent simulations, this planner is optimally suited to simulate character activity for emergent narrative. We hope that it will serve as an accelerator for the deployment of modern AI technology within and beyond academia and foster innovative applications.

Acknowledgements

We thank Aydin Faraji, Patrick Eppensteiner, Nora Tommila, and Heinrich Grattenthaler for their contributions.

References

Fulton, K. R.; Chan, A.; Votipka, D.; Hicks, M.; and Mazurek, M. L. 2021. Benefits and drawbacks of adopting a secure programming language: rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, 597–616.

Jaschek, C.; Beckmann, T.; Garcia, J. A.; and Raffe, W. L. 2019. Mysterious murder-mcts-driven murder mystery generation. In *2019 IEEE Conference on Games (CoG)*, 1–8.

Kartal, B.; Koenig, J.; and Guy, S. J. 2014. User-driven narrative variation in large story domains using monte carlo tree search. In *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*, 69–76.

Konen, W. 2019. General Board Game Playing for Education and Research in Generic AI Game Learning. In *2019 IEEE Conference on Games (CoG)*, 1–8.

Pell, B. 1996. A Strategy Metagame Player for General Chess-like Games. *Computational Intelligence*, 12(1): 177–198.

Raymond, H.; Knobloch, S.; Zünd, F.; Sumner, R. W.; and Magnenat, S. 2020. Leveraging efficient planning and lightweight agent definition: a novel path towards emergent narrative. In *12th Intelligent Narrative Technolgies Workshop, held with the AIIDE Conference (INT10 2020)*.

[6] bevyengine.org

[7] For example Embark Studios and Dims

[8] See rustcxx and rnet for C++ respectively C# integration.

Ryan, J. 2018. *Curating Simulated Storyworlds*. Ph.D. thesis, University of California Santa Cruz.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; Lillicrap, T.; and Silver, D. 2020. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839): 604–609.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; Oh, J.; Horgan, D.; Kroiss, M.; Danihelka, I.; Huang, A.; Sifre, L.; Cai, T.; Agapiou, J. P.; Jaderberg, M.; Vezhnevets, A. S.; Leblond, R.; Pohlen, T.; Dalibard, V.; Budden, D.; Sulsky, Y.; Molloy, J.; Paine, T. L.; Gulcehre, C.; Wang, Z.; Pfaff, T.; Wu, Y.; Ring, R.; Yogatama, D.; Wünsch, D.; McKinney, K.; Smith, O.; Schaul, T.; Lillicrap, T.; Kavukcuoglu, K.; Hassabis, D.; Apps, C.; and Silver, D. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782): 350–354.

Ware, S. G.; and Young, R. M. 2014. Glaive: a state-space narrative planner supporting intentionality and conflict. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.