International Journal on Artificial Intelligence Tools
Vol. XX, No. X (2025) 1–22

JGraphs: A Toolset to Work with Monte-Carlo Tree Search-Based Algorithms

García-Díaz, V., Núñez-Valdez, E. R., García, C. G., Gómez-Gómez, A., & Crespo, R. G. (2020). JGraphs: A Toolset to Work with Monte-Carlo Tree Search-Based Algorithms. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems, 28(Supp02), 1–22. https://doi.org/10.1142/S0218488520400115

JGraphs: A Toolset to Work with Monte-Carlo Tree Search-Based Algorithms

Vicente García-Díaz

Department of Computer Science, University of Oviedo, Federico García Lorca 18, Oviedo, 33011, Spain
garciavicente@uniovi.es

Edward Rolando Núñez-Valdez

Department of Computer Science, University of Oviedo, Federico García Lorca 18, Oviedo, 33011, Spain
nunezedward@uniovi.es

Cristian González García

Department of Computer Science, University of Oviedo, Federico García Lorca 18, Oviedo, 33011, Spain
gonzalezcristian@uniovi.es

Alberto Gómez Gómez

Department of Business Administration, University of Oviedo, Viesques Campus, Gijón, 33204, Spain
albertogomez@uniovi.es

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Monte-Carlo methods are the basis for solving many computational problems using repeated random sampling in scenarios that may have a deterministic but very complex solution from a computational point of view. In recent years, researchers are using the same idea to solve many problems through the so-called Monte-Carlo Tree Search family of algorithms, which provide the possibility of storing and reusing previously calculated results to improve precision in the calculation of future outcomes. However, developers and researchers working in this area tend to have to carry out software developments from scratch in order to use their designs or improve designs previously created by other researchers. This makes it difficult to see improvements in current algorithms as it takes a lot of hard work. This work presents JGraphs, a toolset implemented in the Java programming language that will allow researchers to avoid having to reinvent the wheel when working with Monte-Carlo Tree Search. In addition, it will allow testing experiments carried out by others in a simple way, reusing previous knowledge.

Keywords: Monte-Carlo; Monte-Carlo Tree Search, Best-first search algorithm, Combinatorial game, Tic-Tac-Toe, JGraphs

1. Introduction

Monte-Carlo (MC) methods are algorithms that rely on repeated random simulations to provide generally approximate solutions. The main idea is to use randomness to solve problems that may be deterministic but that are too big to be calculated[1]. MC can be applied in many domains such as decision support systems[2] or learning analytics[3]. In fact, within the Artificial Intelligence community, MCTS has received increasing attention in the last years[4].

The accuracy and utility of MC simulations can be improved using tree-based search, leading to the Monte-Carlo Tree Search (MCTS). MCTS is a best-first search algorithm based on heuristics where pseudorandom simulations guide the solution of a problem. The goal is to find optimal decisions in a specific domain of knowledge by generating random samples in the decision space, at the same time that a search tree is generated according to the obtained results[5].

MCTS has been successfully applied in many contexts related to games such as: 1) Two-player games, being one of the greatest successes the 100-0 victory of AlphaGo Zero against AlphaGo, that had been the first program to defeat a world champion in the game of Go[6]; 2) Single-player games, proposed as a variant called Single-Player Monte-Carlo Tree Search (SP-MCTS) when first applied to the SameGame puzzle[7]; 3) Multi-player games, proposed as a variant called Multi-Player Monte-Carlo Tree Search Solver (MP-MCTS-Solver) when first applied to the Focus and Chinese Checkers games[8]; 4) Real-time games such as the NaïveMCTS algorithm applied to strategy games with good results compared to other alternatives when the branching factor grows[9]; or 5) Nondeterministic games, like in the case of Poker, with expected reward distributions[10].

However, MCTS has also been applied to other areas such as[11]: 1) combinatorial optimization; 2) constraint satisfaction; 3) scheduling problems; 4) sample-based planning; or 5) procedural content generation[5], to mention some of the most representative ones. The application areas grow continuously in all kinds of works but with special impetus in topics where artificial intelligence or the internet of things play an important role.

The motivation of this work is based on the premise that current libraries to work with MCTS are generally focused on very specific domain of knowledges like videogames (e.g., FUEGO[12] or Centurio[13]), material (e.g., MDTS[14]) or chemistry (e.g. ChemTS[15]) and lack of tools that facilitate analysis, debugging, visualization or interoperability of the solutions made by the researchers.

This works presents JGraphs, a toolset to work with MCTS-based algorithms. JGraphs is an extensible framework that allows developers to create applications based on MCTS in an agile and simple way. It provides several utilities that facilitate analysis, debugging, visualization and interoperability between applications, while offering some default implementations as well as extension mechanisms to make it easier for developers to tailor the code to their specific needs. With JGraph, developers will be able to focus on solving MCTS-based problems, which are already very complex, rather than reinventing the wheel repeatedly. In addition, the code is distributed openly, allowing new developments to be incorporated in the future.

The rest of this work is structured as follows: Section 2 introduces the main concepts to understand the proposal. Section 3 explains our proposal called JGraphs. Section 4 defines the use case scenario used to test the proposal. Section 5 applies the proposal to the use case scenario and shows how it works. Section 6 briefly compares JGraphs to other works. Finally, Section 7 indicates our conclusions and future work to be done.

2. Background

Although there are many implementations and hundreds of variations for MCTS, they share the basic idea of 4 different steps carried out cyclically[5] as shown in Fig. **1**.

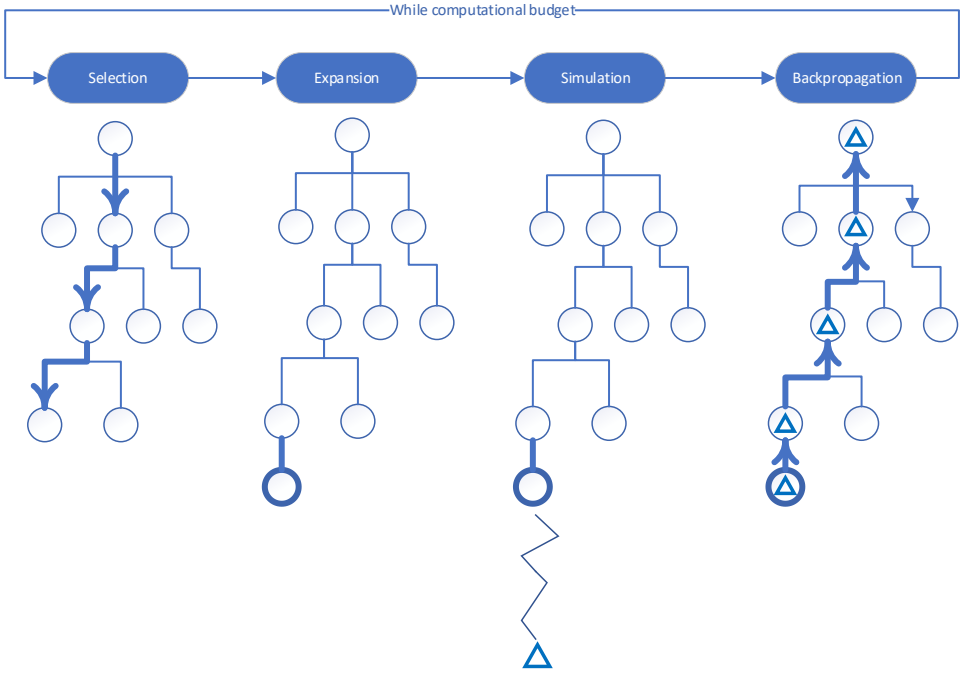

Fig. 1. Monte Carlo Tree Search general scheme

The following paragraphs explain what each of the main steps may consist of, as well as their importance in the implementation of a solution for MCTS. It is important to note that all the steps are executed if there is still computational budget. Such a budget can be based on any aspect of the system such as number of predefined iterations, memory, CPU usage or elapsed time. The fundamental idea is that the more iterations are performed, the more closely adjusted and close to reality the results will be.

2.1. *Selection policy*

The selection step is intended to find the best expandable node. A node is expandable if it is a non-terminal node and still has unvisited children. To find the best expandable node, there are different algorithms that go recursively from the root node (the current situation in the tree of states) to an expandable node. The most popular algorithm is called the Upper Confidence Bound for Trees (UCT)[16], that tries to address the exploration-exploitation dilemma in MCTS and provides good results in most contexts. However, there are other alternatives to carried out the selection policy such as the multi-objective variation presented in MO-MCTS[17].

This step is represented in the following snippet of pseudocode:

```
function selectionPolicy(n):
   while n is non-terminal do
```

```
    if n is not expanded then
        expansionPolicy(n)
    else
        n_next ← arg max_{n' children of n} R(n')/V(n') + C·sqrt(2 x Ln V(n) / V(n'))
return n_next
```

where *R(n')* is the obtained total reward moving through a child node *n'* from the point of view of a specific participant, *V(n')* is the total number of times the child node *n'* was visited during the process, *V(n)* is the total number of times the current node *n* was visited during the process, n_{next} is the next node to which the algorithm will be recursively applied until reaching the best expandable node, and *C* is a constant that needs to be adjusted empirically.

2.2. *Expansion policy*

The expansion step consists of adding a new node to the tree whenever we reach a non-terminal node with at least a child that has not yet been added to the tree. The nodes are added following some rules that may be included in the definition of the solution, trying to include first the most *promising* nodes. The definition of the most promising node depends on the context, although the most common way is to generate the different successors from a node and select each of them randomly. Thus, the expansion step is useful to expand the tree according to some actions.

2.3. *Simulation policy*

The simulation step is used to perform a simulation that reaches a terminal state to obtain a reward output value Δ, starting the process from the new node created with the expansion policy. The types of simulations can vary from random steps to sophisticated solutions based on different algorithms like Artificial Neural Networks[18] (ANN).

2.4. *Propagation policy*

The backpropagation step is intended to update the values of the node that initialized the simulation and its predecessors according to the reward value Δ obtained after the simulation. The reward value may be a discrete value, a continuous value, or multiple values depending on the domain. This step is represented in the following snippet of pseudocode:

```
function propagationPolicy(n, Δ):
   while n is not null do
      R(n) ← R(n) + Δ
      V(n) ← V(n) + 1
      n ← n''
```

where *n''* is the parent of node *n*, *R(n)* is the total accumulated reward for node *n*, and *V(n)* is the total number of times the node *n* was visited during the process.

3. Proposal

Fig. **2** shows an overview of JGraphs architecture, our proposal to provide a toolset for working with MCTS related problems, allowing researchers to focus on the specific problems instead of on the underlying technology to solve them. The following sections explain the different modules.

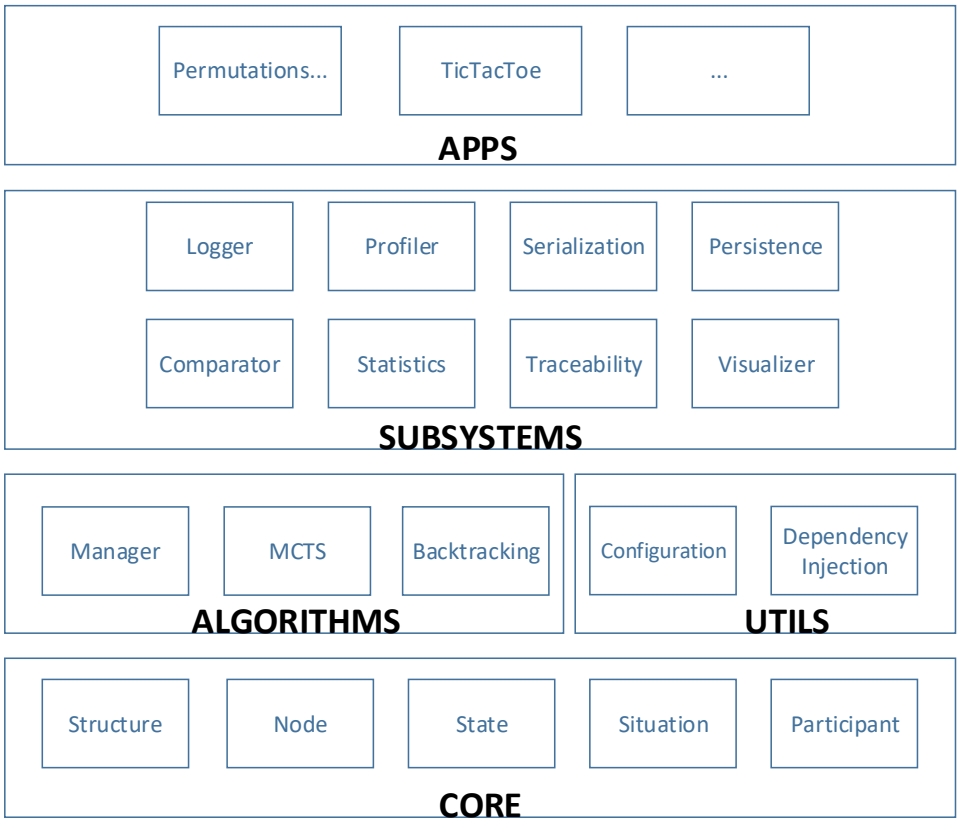

Fig. 2. JGraphs overview

3.1. *Core*

The Core refers to the main module, on which the others rely on. It is currently made up of 5 components.

3.1.1. *Structure*

This component is responsible for creating the skeleton on which the algorithms and solutions created with the framework will be based. Basically, an `AbstractStructure` class implements some basic methods to manage and store the nodes that will be inserted in graphs or trees. There are 4 different classes to work with graphs and trees:

- `Tree` and `Graph`. Default classes to work with trees and graphs respectively.
- `PTree` and `PGraph`. Parallelized version of the default classes.

- `SilentTree` and `SilentGraph`. Light versions (e.g., not storing nodes for further tree analysis) of the default classes.
- `SilentPTree` and `SilentPGraph`. Light and parallelized version of the default classes for performance improvement.

3.1.2. *Node*

This component is responsible for creating and managing the elements that will make up the structures, that is, the nodes. All the nodes store their predecessor and successor nodes and provide several methods to simplify the development. All the `Node` objects have a state.

3.1.3. *State*

This component gives dynamic information about the changes undergone in a node. All the `State` objects store a group of `Participant` objects (i.e., external agents that interact with the state), along with scores for each of the participants, the number of times that the state was visited and the situation, that is a domain-specific representation of the state. The class is completely generic to be adapted to any `Situation` object.

3.1.4. *Situation*

This component is useful to represent a specific situation in the development of events. For example, when playing chess, it would be a board with the position of the pieces. It provides a graphical representation together with information such as what will be the following possible situations from a given one (e.g., next possible movements in the chess game), or when the process is finished (e.g., when checkmate or draw occurs). By default, `IntArraySituation` and `IntTableSituation` are provided to represents situations through an array of integers or a bidimensional array of integers. But new constructions can be easily added.

3.1.5. *Participant*

This component is responsible for managing the different agents that can interact in a process, as well as the turns of intervention. For example, it may be a process where there is only one participant (`SingleParcipantManager`) or a game for two players (`TwoParticpantsManager`).

3.2. *Algorithms*

The Algorithms module includes a manager, which facilitates working with the different algorithms that could be implemented. Currently, Backtracking (a basic depth-first search algorithm) and MCTS (the goal of this work) have been implemented.

3.2.1. *Manager*

To execute any algorithm, it is necessary to use an `AbstractManager` that is going to be in charge of different actions such as managing visualizers, statistics or traceability. It

also provides some utilities such as storing current solutions, or the number of steps performed until some point.

3.2.2. *Backtracking*

It provided a basic implementation of a brute force algorithm based on depth-first search. `BacktrackingOne` gives the first solution for any problem and `BacktrackingAll` gives all the solutions. These classes are mainly used for testing and benchmarking purposes.

3.2.3. *MCTS*

It provides a standard implementation of MCTS (`MCTS`) and a parallelized one (`PMCTS`) based on the Fork/Join Framework[19], along with various utility classes.

The `IBudgetManager` allows to specify the way to check when to stop a MCTS step. The default implementation `DefaultBudgetManager` considers number of iterations, memory used and/or time in seconds. Like almost everything in the system, it can be configured through code or through an external properties file.

There are many default implementations to define different policies for the MCTS algorithm. For example:

- The `UCBSelectionPolicy` implements the Upper Confidence Bound (UCB)[16] to determine which node will be the next one chosen in the tree, trying to get the best movement for a specific participant considering that each participant may have different scores for a same situation. It is based on the `ISelectionPolicy` interface.
- The `GeneralAllExpensionPolicy` creates a list with all the children of a given node and link them in order to maintain the order in the data structure. It is based on the `IExpansionPolicy` interface.
- The `RandomMovementSimulationPolicy` performs a simulation from a node to a final state in a random way. There may be many variations of this implementation that include, for example, heuristics for a certain domain of knowledge or even simulations performed with neural networks. All of them may be based on the `ISimulationPolicy` interface.
- The `UpdateAllPropagationPoliy` propagates the score after a simulation to the previously visited nodes in the tree. With the default implementation, nodes may have different scores depending on whether they resulted in a successful situation, an unsuccessful situation, or a neutral situation. In addition, it can consider that nodes may have different points of view depending on the participant (i.e., what is good for a participant may be bad for another one). It is based on the `IPropagationPolicy` interface.

3.3. *Apps*

The Apps module includes implementation for different scenarios. To begin with, we have implemented two scenarios used to test the utility of the different algorithms. A

permutations generator to work with Backtracking-based algorithms and a Tic-Tac-Toe[20] game engine to work with MCTS-based algorithms.

The TicTacToe game is going to be used as a use case in this work for working with MCTS. Besides, the Backtracking-based algorithm is not used to solve problems of a certain size, since it relies on what is known as brute force to search among all the possible alternatives through the search tree, offering complexities that are typically exponential or factorial. These complexities are not adequate for most board games, even those that at first glance seem affordable.

3.4. *Utils*

The Utils module provides some basic functionality that is useful for the different components. For example, a configuration subsystem is included to standardize the way in which applications are set up by developers. By default, it includes 4 modes of operation:

- Basic: To work with graphs, trees, nodes and states.
- Basic parallelized: To work with a parallelized version of the basic mode.
- Silent basic: To work with light versions (e.g., not storing nodes for further tree analysis) of graphs, trees, nodes and states.
- Silent basic parallelized: To work with a parallelized version of the silent basic mode.

In addition, it provides some utilities for dependency injection and configuration of the different subsystems through external properties text files.

3.5. *Subsystems*

The Subsystems module is responsible for the different components that provide advanced functionality in the system.

3.5.1. *Logger subsystem*

A logger subsystem is provided to standardize the way in which the logs created by developers are stored, including different levels and multiple possible configurations. The underlying technology is based on the Simple Logging Facade for Java (SLF4J)[21] and the Apache Log4j API[22]. The `DefaultLogger` class implements the `ILogger` interface, which allows 5 levels of severity (error, warn, info, debug, trace).

3.5.2. *Profiler subsystem*

A profiler subsystem is provided to standardize the way in which dynamic program analysis is carried out by developers, including time measurements to perform different tasks and the creation of textual and time series database recorders. The underlying technology is based on the SLF4J profilers. The `DefaultProfiler` class extends the `AbstractProfiler` class and implements the `IProfiler` interface.

3.5.3. *Serialization subsystem*

A serialization subsystem in provided to standardize the way in which graphs are converted to a textual representation. We have decided to use the de-facto industry standard, JavaScript Object Notation (JSON)[23], to provide the ability to create graph representations

that can be persisted, manually inspected, or later processed with another tool. `AbstractSerializer` serializes and deserializes most of the data automatically, regardless of the application case. Developers only must indicate the specific data structures they use in their situations (if they are not already included in `IntArraySituation` or in `IntTableSituation`).

3.5.4. *Persistence subsystem*

A persistence subsystem is provided to standardize the way in which graphs are stored for further manipulation. By default, `MemoryPersistence`, `FilePersistence` or `H2Persistence` are already provided to allow storing structures in memory, a text file or in a data base. H2[24] is a relational database management system written in Java, that can be embedded in applications or run in client-server mode. As in the case of the serialization subsystem, this operation can be performed at any time during execution, which can be a powerful debugging or interoperation tool.

3.5.5. *Comparator subsystem*

A comparator subsystem is provided to standardize the way in which graphs are compared to see the differences between them. The underlying technology is based on the implementation of RFC 6902 JSON Patch carried out by ZjsonPatch[a]. That is a standard format for describing changes to a JSON document. For example, it can be used to see the changes or to avoid manipulating a whole document when only a small part has changed. The `DefaultComparator` class implements the `IComparator` interface.

3.5.6. *Statistics subsystem*

A statistics subsystem is provided to facilitate developers to obtain information about the graph in a direct way. Developers can use a `checkpointEvent` in any moment to generate statistics about what is happening in the graph. All statistics will be stored in time series. The stored information includes: 1) total elapsed time; 2) depth of the tree; 3) width of the tree; 4) width of the explored tree; 5) number of nodes; 6) number of explored nodes; 7) number of nodes that have not been explored; 8) number of visits; 9) visits per node; 10) visits per explored node; 11) top visited nodes; and 12) top ranked nodes.

3.5.7. *Traceability subsystem*

A traceability subsystem is provided to allow developers to run algorithms step by step, making it possible to stop executions and display data deemed appropriate during stops. The `DefaultTraceability` class implements the `ITraceability` interface, which allows to stop executions at any point.

3.5.8. *Visualizer subsystem*

A visualizer subsystem is provided to graphically or textually represent the graphs/trees created during an execution. The `IVisualizer` interface has 3 methods, which are used

[a] zjsonPatch: https://github.com/flipkart-incubator/zjsonpatch (last visited at April 13, 2020).
[b] graphviz-java: https://github.com/nidi3/graphviz-java (last visited at April 13, 2020).

to make visualizations at different times (in each iteration, in each movement and when the process ends). The underlying technology for graphical representation is based on the Graphviz open source graph visualization software[25] and its implementation for Java through the `graphviz-java`[b] library.

4. Use case definition

We will show the framework through a classic use case: the Tic-Tac-Toe game[20]. It is a combinatorial game that represents the group of problems to which MCTS has been most often applied[26]. Like Tic-Tac-Toe, combinatorial games have the following properties[27]:

- Two players. There are typically only considered two players, although may not be the case.
- Zero-sum. The gain or loss of a player is exactly balanced by the loss or gain of the other player.
- Perfect information. The state of the game is fully observable to the players.
- Deterministic. There is no randomness in the development of future states.
- Sequential. Players move sequentially and in turns.
- Finite. The number of movements must be always finite.

4.1. *Tic-Tac-Toe game*

The Tic-Tac-Toe game (a.k.a. noughts and crosses, or Xs and Os) is a popular paper-and-pencil game for two players who, in turns, place Xs and Os respectively on a board that typically has a size of 3x3 cells. The player who is first able to place his marks along an entire row, an entire column or one of the main diagonals, wins the game. **Fig. 3** shows an example of a game with 7 moves in which the player with Xs wins.

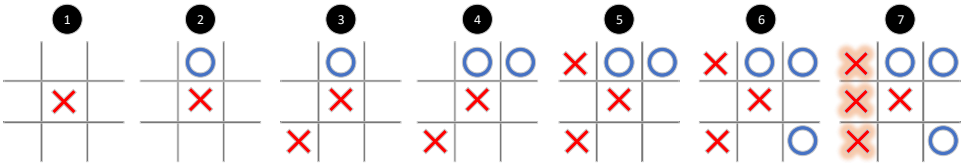

Fig. 3. Example of complete Tic-Tac-Toe game

It is a very simple game that is used for pedagogical uses and as a benchmark in artificial intelligence[28]. In this game, if both players play perfectly, the result of the game will be a draw. In total, there can be a total of 765 different situations on the board, with a total of 26,830 different possible games, given the different alternative movements that can be done.

4.2. *Using JGraphs for solving the Tic-TacToe game*

With JGraphs, designing any MCTS-based solution is a very simple task in which we only must consider 3 main actions (the complete source code is included in the project):

4.2.1. *Module configuration*

Since MCTS-based architectures can be setup in multiple ways, we created a `TicTacToeModule` class file in which we define our configuration by just overriding

the `configure()` method. That class extends `DefaultModuleConfiguration` to provide specific configuration for any problem. For this, some of the classes that are already included are used. However, others could be used, or new implementations could be created by extending the existing ones.

Fig. **4** shows all the code that is needed to configure the following items:

- The selection policy is `UCBSelectionPolicy`.
- The expansion policy is `GenerateAllExpansionPolicy`.
- The simulation policy is `RandomMovementSimulationPolicy`.
- The propagation policy is `UpdateAllPropagationPolicy`.
- The budget manager is `DefaultBudgetManager`.
- The participant manager is `TwoParticipantsManager`.
- The max value node is `ScoreMaxValueNode`.

```
@Override
protected void configure() {
    super.configure();
    bind(ISelectionPolicy.class).to(UCBSelectionPolicy.class);
    bind(IExpansionPolicy.class).to(GenerateAllExpansionPolicy.class);
    bind(ISimulationPolicy.class).to(RandomMovementSimulationPolicy.class);
    bind(IPropagationPolicy.class).to(UpdateAllPropagationPolicy.class);
    bind(IBudgetManager.class).to(DefaultBudgetManager.class);
    bind(IParticipantManager.class).to(TwoParticipantsManager.class);
    bind(IMaxValueNode.class).to(ScoreMaxValueNode.class);
}
```

Fig. 4. Example module configuration for the Tic-Tac-Toe game

4.2.2. *Situation definition*

Since every problem is going to be different, it is necessary to define how each of the situations in the development of the process is going to be presented. For this case, we provide a `TicTacToeSituation` class that extends the `IntTableSituation` abstract class. The abstract class already provides all the necessary infrastructure to represent any problem with numbers inserted in a bidimensional structure, so, we only had to override 3 methods with less than 70 lines of code in total.

- `createNewSituation()`. It relies on the super class to create a copy of a game situation, that is, copying the values of its attributes.
- `nextSituations()`. It creates a list of possible new situations from a given situation. That is, the possible next movements in the game from a given position.
- `checkStatus()`. It gives a value to indicate the status of a situation with the following possibilities: 1) one of the players has won; 2) the game ends with a draw; or 3) the game is in progress.

4.2.3. *Serialization definition*

Some actions such as serialization or persistence of graphs require some information that is specific to each situation in the graph. However, in most cases it will be enough to rely on the core JGraph structure. For example, to play Tic-Tac-Toe, the

`TicTacToeSerializer` is created extending the abstract class `IntTableSerializer`. In this case, the abstract class already provides all the necessary infrastructure to serialize any problem with numbers inserted in a bidimensional structure, so, we only need to create a constructor with 1 line of code initializing an instance of `TicTacSituation`.

5. Use case execution

We intend to show an example simple enough to facilitate understanding of how the platform works. We have created a game by training the two players at the same time, using only 3 iterations for each movement/execution and with a random simulation policy. A different configuration of the framework with more iterations and other policies would result in perfect games, but it would make more difficult to understand the behavior. Fig. **5** shows how the pseudo-random game played between the two participants has been carried out.

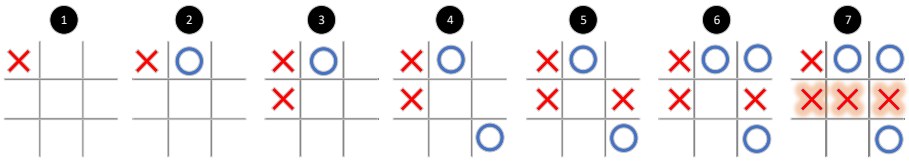

Fig. 5. Pseudo-random game played with JGraphs with 3 iterations per movement

5.1. *Logger*

Fig. **9** shows an example of the output with the `DefaultLogger` class. It allows to directly inspect and view anything in the graph: all the nodes, a specific node, a specific state or a specific situation. In this example, the content of the last node of the game is shown. Since it is a final state, it has not successors but a predecessor. It also has a state in which the player 1 is moving, with only 1 visit (the last movement) and the corresponding score for each player. In Addition, the situation of the game for that state is shown. It is the level 7 (seventh movement of the game), with a status of 1, that means that the player 1 won.

```
Node:
        Id: a9e75487-08ca-4ba2-99aa-5247f4aa96c3
        Predecessor: 6a392eb7-1980-421e-8f03-72c8dabe90b2
State:
        Id:             09c7012b-3925-4097-a155-c7a34cbd2645
        Participant:    1
        VisitCount:     1
        WinScore:       [1.0 0.0]
Situation:
        Level:          7
        Status:         1
        Values:         1 2 2
                        1 1 1
                        0 0 2
```

Fig. 6. Last node information printed with the logger

5.2. *Profiler*

Fig. **7** shows an example of the output with the `DefaultProfiler` class. It allows to dynamically analyze the execution of the program and get information such as that the initialization of the process was carried out in 392 milliseconds and the execution of the game in 55. The whole process took less than half a second.

```
+ Profiler [Jgraphs]
|---+ Profiler [ProfilerTest]
    |-- elapsed time [initialize the process]   392,686 milliseconds.
    |-- elapsed time          [play the game]    55,557 milliseconds.
    |-- Subtotal               [ProfilerTest]   452,853 milliseconds.
|-- elapsed time           [ProfilerTest]   452,853 milliseconds.
|-- Total                       [Jgraphs]   453,348 milliseconds.
```

Fig. 7. Execution information obtained from the profiler

5.3. *Serialization*

Fig. **8** shows an example of the output with the `TicTacToeSerializer` class. The serializer is intended to create a JSON of the entire graph so that it can be used by other internal tools (such as the persistence subsystem) or other tools external to the toolset. The example represents just the first node in the game, which still has no movement made. When the game ends this node has been visited a total of 21 times and after all the simulations it has registered 14 points for player 1 and 7 for player 2, giving the first player the clear favorite. This node has 9 possible successors that correspond to the 9 squares in which player 1 can start by placing the X.

```
{
   "predecessors": [],
   "name": "Node1",
   "successors": [
      "a60c70b1-ce9d-4279-8e39-a83694e0b061",
      "60c842e3-4669-4ce6-b99b-f8d1bf669984",
      "c488d3c6-3284-4c30-bc8b-5ed9ea0ea4eb",
      "231e7d26-2d12-44c5-ae67-4d41762a40c4",
      "60f6ec39-1722-4529-ba4d-56076a9121be",
      "3620e1ad-e688-4169-afe5-93c7271b9c4b",
      "8587c5df-a15e-419b-b098-d2984b90d9bd",
      "fc53e404-f2e3-4e60-8bc8-33b088c04418",
      "0332fe70-7cc0-48ba-bcd2-0dfe4915aa18"
   ],
   "id": "1603abd3-c3e5-4923-b2a7-064ad2b8df33",
   "state": {
      "currentParticipant": 0,
      "visits": 21,
      "scores": [
         14,
         7
      ],
      "id": "a9b52c27-5cd5-494b-8ea6-6f66f85078af",
      "situation": {
         "level": 0,
         "values": [
            [0,0,0],
            [0,0,0],
            [0,0,0]
         ],
         "n": 3
      }
   }
},
```


Fig. 8. First node information serialized

5.4. *Persistence*

Fig. **9** shows a part of what can be seen with the H2 database management system for the graph created as an example. Like the other classes that can also persist the graph information both in memory and on disk, the `H2Persistence` class is based on the `TicTacToeSerializer` serializer. Nothing more is required to persist the data in the different alternatives.

```
SELECT * FROM JGRAPHS
```

ticTacToe	[{"predecessors":[],"name":"Node1","successors":["a60c70b1-ce9d-4279-8e39-a83694e0b061","60c842e3-4669-4ce6-b99b-f8d1bf669 93c7271b9c4b","8587c5df-a15e-419b-b098-d2984b90d9bd","fc53e404-f2e3-4e60-8bc8-33b088c04418","0332fe70-7cc0-48ba-bcd2-0 6f66f85078af","situation":{"level":0,"values":[[0,0,0],[0,0,0],[0,0,0]],"n":3}}},{"predecessors":["1603abd3-c3e5-4923-b2a7-064ad2b8df33 12af77a81a57","a61903cb-8de3-4e91-b182-ba98732423de","b172eea4-77d8-4fb1-8a2d-6c988488808f","9de79910-639b-4ec8-b0b3- {"currentParticipant":1,"visits":19,"scores":[13.5,5.5],"id":"a5f27065-aeb1-4d35-adfa-74b1ff1f5c1a","situation":{"level":1,"values":[[1,0,0] f8d1bf669984","state":{"currentParticipant":1,"visits":1,"scores":[0,1],"id":"25a203d6-3609-4c5c-b9c2-391f329e20ba","situation":{"level" 5ed9ea0ea4eb","state":{"currentParticipant":1,"visits":1,"scores":[0.5,0.5],"id":"20d020de-8214-4545-8e02-e520dafae33a","situation":{" ae67-4d41762a40c4","state":{"currentParticipant":1,"visits":0,"scores":[0,0],"id":"d2de5236-d8e1-4bb7-a71a-2a63e409425d","situation" ba4d-56076a9121be","state":{"currentParticipant":1,"visits":0,"scores":[0,0],"id":"15587d83-ded5-4226-a523-d0ddc69fe7b1","situation" afe5-93c7271b9c4b","state":{"currentParticipant":1,"visits":0,"scores":[0,0],"id":"699f8852-f30f-4bf0-b8ab-98d11adc0ce5","situation":{"l b098-d2984b90d9bd","state":{"currentParticipant":1,"visits":0,"scores":[0,0],"id":"e8038835-6b37-46be-98cd-e56918100184","situation" 8bc8-33b088c04418","state":{"currentParticipant":1,"visits":0,"scores":[0,0],"id":"8fe50658-fbb9-4343-9b47-7c1604a52d57","situation": bcd2-0dfe4915aa18","state":{"currentParticipant":1,"visits":0,"scores":[0,0],"id":"7cbbc17f-3389-40a5-bab5-b84887e3b754","situation": 48e5f49a883b","6e2e5fca-1368-4115-b54e-07d912b8319c","45b59018-ed7b-4e54-a0ca-cde8de741ebd","e89386ef-1639-464e-bc40- 4ad7-8a21-d756b0e834bf","state":{"currentParticipant":2,"visits":16,"scores":[11.5,4.5],"id":"54751c57-f679-46ba-8b3b-c13170d510f9" a793-4f35-a700-f921857aed1f","state":{"currentParticipant":2,"visits":1,"scores":[0,1],"id":"0e6bfe84-da67-44bf-aeb6-cd130be89dcb","'

Fig. 9. Information persisted in the H2 database

5.5. *Comparator*

Fig. **10** shows an example of the output with the `DefaultComparator` class that implements the RFC 6902 JSON Patch. We compared two nodes where the only difference can be seen marked with a circle in the situations:

- In the new situation, the level is 6 instead of 7. In the output we can see the change with the op "replace".
- In the new situation, one of the values in the table is 0 instead of 1. From the point of view of the comparator that fact is seen as the inclusion of a new element 0 and the deletion of an element 1.

Using the RFC 6902 JSON Patch, JGraph automatically provides a standardized way to compare different graphs, allowing to carry out studies and debugging in a simple and independent way, regardless of the problem.

Fig. 10. Comparison between 2 different situations

5.6. *Statistics*

Fig. **11** shows some interesting statistics about the process carried out. It includes the depth and width of the tree, together with information such as the number of nodes, the visits and the best nodes after the simulations from the point of view of both participants. As expected, the nodes in the first levels of the tree are the most visited and ranked ones.

```
*****************************STATISTICS*****************************
Depth of the tree: 8
Width of the tree: 9
Width of the explored tree: 3
Number of different situations: 43
Number of nodes: 43
Number of explored nodes: 22
Number of nodes that have not been explored: 21
Number of visits: 105
Visits per node: 2,441860
Visits per explored node: 4,772727
Top visited nodes
          Node: Node1 (1603abd3-c3e5-4923-b2a7-064ad2b8df33)        Visits: 21        Scores: [14.0 7.0 ]         Total scores: 21,000000
          Node: Node2 (a60c70b1-ce9d-4279-8e39-a83694e0b061)        Visits: 19        Scores: [13.5 5.5 ]         Total scores: 19,000000
          Node: Node11 (dc8162d0-b505-4ad7-8a21-d756b0e834bf)       Visits: 16        Scores: [11.5 4.5 ]         Total scores: 16,000000
          Node: Node20 (6e2e5fca-1368-4115-b54e-07d912b8319c)       Visits: 13        Scores: [10.5 2.5 ]         Total scores: 13,000000
          Node: Node31 (112e62a6-e604-4326-b073-002c8d4c837e)       Visits: 10        Scores: [8.0 2.0 ]          Total scores: 10,000000
          Node: Node34 (bbc9fb96-9363-4cc4-9f6d-4d83873dc0bc)       Visits: 7         Scores: [7.0 0.0 ]          Total scores: 7,000000
          Node: Node37 (6a392eb7-1980-421e-8f03-72c8dabe90b2)       Visits: 4         Scores: [4.0 0.0 ]          Total scores: 4,000000
          Node: Node3 (60c842e3-4669-4ce6-b99b-f8d1bf669984)        Visits: 1         Scores: [0.0 1.0 ]          Total scores: 1,000000
          Node: Node4 (c488d3c6-3284-4c30-bc8b-5ed9ea0ea4eb)        Visits: 1         Scores: [0.5 0.5 ]          Total scores: 1,000000
          Node: Node12 (6140f300-a793-4f35-a700-f921857aed1f)       Visits: 1         Scores: [0.0 1.0 ]          Total scores: 1,000000
Top ranked nodes
          **Participant 1
          Node: Node1 (1603abd3-c3e5-4923-b2a7-064ad2b8df33)        Visits: 21        Scores: [14.0 7.0 ]         Total scores: 21,000000
          Node: Node2 (a60c70b1-ce9d-4279-8e39-a83694e0b061)        Visits: 19        Scores: [13.5 5.5 ]         Total scores: 19,000000
          Node: Node11 (dc8162d0-b505-4ad7-8a21-d756b0e834bf)       Visits: 16        Scores: [11.5 4.5 ]         Total scores: 16,000000
          Node: Node20 (6e2e5fca-1368-4115-b54e-07d912b8319c)       Visits: 13        Scores: [10.5 2.5 ]         Total scores: 13,000000
          Node: Node31 (112e62a6-e604-4326-b073-002c8d4c837e)       Visits: 10        Scores: [8.0 2.0 ]          Total scores: 10,000000
          Node: Node34 (bbc9fb96-9363-4cc4-9f6d-4d83873dc0bc)       Visits: 7         Scores: [7.0 0.0 ]          Total scores: 7,000000
          Node: Node37 (6a392eb7-1980-421e-8f03-72c8dabe90b2)       Visits: 4         Scores: [4.0 0.0 ]          Total scores: 4,000000
          Node: Node17 (b89e197c-8522-4eaa-8fa7-e564d3fbd264)       Visits: 1         Scores: [1.0 0.0 ]          Total scores: 1,000000
          Node: Node21 (45b59018-ed7b-4e54-a0ca-cde8de741ebd)       Visits: 1         Scores: [1.0 0.0 ]          Total scores: 1,000000
          Node: Node26 (aec02f03-ee84-4e17-90b5-3bc56ab82207)       Visits: 1         Scores: [1.0 0.0 ]          Total scores: 1,000000
          **Participant 2
          Node: Node1 (1603abd3-c3e5-4923-b2a7-064ad2b8df33)        Visits: 21        Scores: [14.0 7.0 ]         Total scores: 21,000000
          Node: Node2 (a60c70b1-ce9d-4279-8e39-a83694e0b061)        Visits: 19        Scores: [13.5 5.5 ]         Total scores: 19,000000
          Node: Node11 (dc8162d0-b505-4ad7-8a21-d756b0e834bf)       Visits: 16        Scores: [11.5 4.5 ]         Total scores: 16,000000
          Node: Node20 (6e2e5fca-1368-4115-b54e-07d912b8319c)       Visits: 13        Scores: [10.5 2.5 ]         Total scores: 13,000000
          Node: Node31 (112e62a6-e604-4326-b073-002c8d4c837e)       Visits: 10        Scores: [8.0 2.0 ]          Total scores: 10,000000
          Node: Node3 (60c842e3-4669-4ce6-b99b-f8d1bf669984)        Visits: 1         Scores: [0.0 1.0 ]          Total scores: 1,000000
          Node: Node12 (6140f300-a793-4f35-a700-f921857aed1f)       Visits: 1         Scores: [0.0 1.0 ]          Total scores: 1,000000
          Node: Node19 (61b96f67-7e3d-45db-9680-48e5f49a883b)       Visits: 1         Scores: [0.0 1.0 ]          Total scores: 1,000000
          Node: Node4 (c488d3c6-3284-4c30-bc8b-5ed9ea0ea4eb)        Visits: 1         Scores: [0.5 0.5 ]          Total scores: 1,000000
          Node: Node27 (8ed2e671-7826-4b95-898c-926eb1075fe1)       Visits: 1         Scores: [0.5 0.5 ]          Total scores: 1,000000
*********************************************************************
```

Fig. 11. Statistics about the performed process

5.7. *Traceability*

The basic traceability subsystem just allows to stop the process at any point during the development. That way, the developer can take note of the information that can be obtained with other subsystems and take time to understand what is happing during the process. Pressing the space bar, the process continues.

5.8. *Visualizer*

It is possible to visualize movements with different levels of granularity, depending on the needs. For example, Fig. **12** shows how JGraphs automatically creates a trace of what happens in each iteration of a given movement. In case (a), the MCTS algorithm randomly generated one of the possible descendants placing the X just in the center, since at the beginning of the game, X (player 1) could be placed in any place. After that, a random play is done, and the result is a tie (result = 0). According to the configuration of the algorithm, given a tie, both players get 0.5 points. In case (b), the algorithm randomly placed X in the upper-left corner of the board. After the random play, player 2 wins, so she receives 1 point. In case (c), the algorithm tries another option but player 1 loses again after the random

play, so player 2 receives another point in that node. Finally, after the 3 iterations, player 1 decides to move to the best available option, that in that case is just to Node6 (it has a total of 0.5 points while in the other options have 0 points).

In this case, we are using just the default visualization, so the developer does not need to do anything to get it. However, developers may decide to change the way they see the generated nodes and their information.

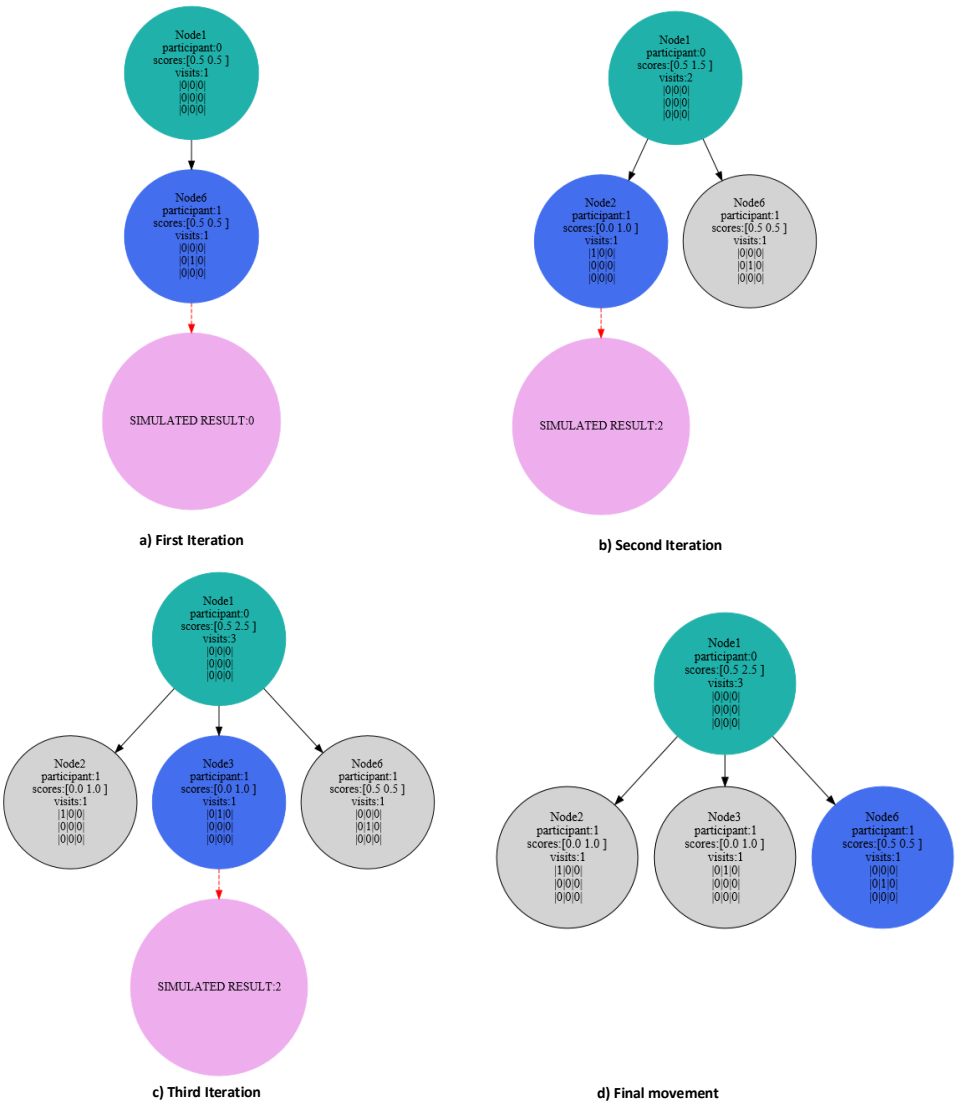

Fig. 12. State of the graph during the first movement

After the final movement of player 1, now player 2 should move. Fig. **13** shows just the final second movement. In that case, since we are using just 3 iterations (a very low value) and since player 1 started in a good position (just in the center), the 3 random plays (simulations) estimate that player 1 is going to be the winner regardless of the movement of player 2. So, player 2 selects Node11 but she may take Node12 or Node13 since all of them offer the same result.

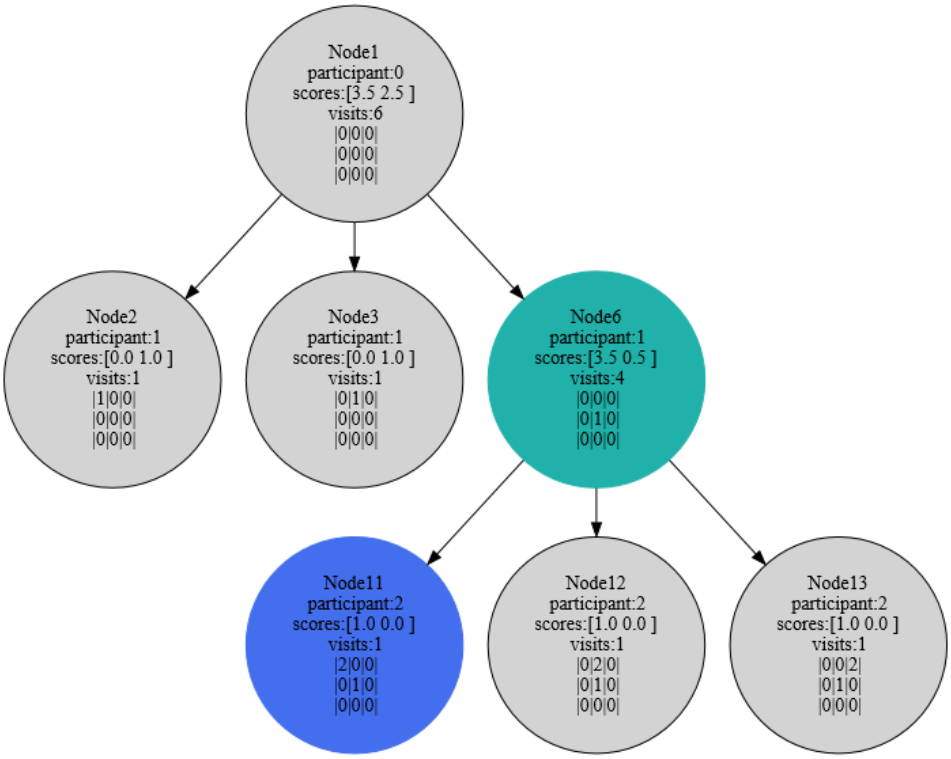

Fig. 13. State of the graph at the end of the second movement

6. Related Work

JGraphs is an open source software that allows to facilitate the development of new research in MCTS-related contexts. It reduces the learning curve and the enormous effort developers must make to start working in this field if they must develop all the code from scratch without reusing state of the art knowledge. It is intended to be a base from which to develop, test, and run new and old algorithms.

Although there are other works that may have similar goals, to the best of our knowledge, all of them are usually closely linked to the context for which they were created and do not offer the JGraph functionalities previously described, making the comparison very complex. Some of the most well-known libraries for working with MCTS are commented below.

FUEGO[12] is an open source framework since 2008, that is intended for developing game engines for full-information two-player board games, focusing on the Go game. The code is written in C++ and is based on 5 different components: 1) GTPEngine, that is a game-independent implementation of a text-based protocol for controlling a Go engine; 2) SMARTGAME, that is a set of useful game-independent functionality for two-player games on square boards, including an implementation of the alpha-beta search and the MCTS search algorithms, assuming both of them that player moves can be represented by positive integer values; 3) GO, that is mainly an implementation of a Go game board, keeping the history of all moves played; 4) Opening book, that includes functions to read opening positions from a data file and match the book against the current position; and 5) Other libraries and applications for playing Go. Focusing on the implementation of MCTS in Fuego, it includes the UCT and the Rapid Action Value Estimation[29] (RAVE) algorithms. They also provide some statistics such as the number of simulations per second of the length of the simulations, that may be included in JGraph in the future. However, Fuego is strongly focused on Go. Thus, MCTS is specifically implemented for playing Go with domain-specific policies and heuristics.

Centurio[13] is a general game open-source playing system that based on Java, including different strategies based on MCTS. Centurio is focused on multi-theading as well as cluster-computing. It uses the Answer Set Programming (ASP), that is a declarative problem-solving paradigm to encode a game as a logic program. At the time of writing this work, Centurio does not appear to be available anymore.

ChemTS[15] is a Python library that explores the chemical space by combining MCTS and ANNs. The library is focused on the design of molecules without any predetermine fragments and with some specific properties such as gap or energy. The library can be extended to obtain other functionalities such as designing molecules to target specific proteins.

MDTS[14] (Materials Design using Tree Search) is another Python library that is intended for automatic materials design. As in ChemTS, the library is fully accessible to be extended providing new functionalities. MDTS solves structure determination of substitutional alloys with composition constraints. Authors compare MCTS and an efficient Bayesian optimization implementation.

7. Conclusions and Future Work

In this work, we presented JGraphs, a toolset to make work easier with MCTS-based algorithms. JGraphs is an extensible framework with several utilities that facilitate analysis, debugging, visualization and interoperability between applications, while providing some default implementations as well as extension points to adapt the source code to specific needs. The main goal is for developers to focus on the problems instead of the code to solve them. To the best of our knowledge, this is the first general purpose framework offered to help solving problems in a field that has been booming lately, and for which developers ended up creating developments from scratch in most cases.

The source code of this project is distributed openly and can be found at https://github.com/vicegd/jgraphs. We think that open-source software can accelerate

research on this field. We will continue working on this project to incorporate new options and functionalities. Thus, future work will be focused on further developing JGraphs, with new algorithms and new use cases not just restricted to games. For example, we will deal with how to use it in real-time industrial scenarios. For that, one of our priorities is to design a mechanism to dynamically define what is the "best expandable node", changing the implementation to recalculate it without stopping the system.

References

1. Metropolis N, Ulam S. The monte carlo method. *J Am Stat Assoc*. 1949;44(247):335-341.
2. Khiat S, Djamila H. A temporal distributed group decision support system based on multi-criteria analysis. *Int J Interact Multimed Artif Intell*. 2019;5(7):7-21.
3. Navarro ÁAM, Ger PM. Comparison of clustering algorithms for learning analytics with educational datasets. *Int J Interact Multimed Artif Intell*. 2018;5(2):9-16.
4. Bertsimas D, Griffith JD, Gupta V, Kochenderfer MJ, Mišić V V. A comparison of Monte Carlo tree search and rolling horizon optimization for large-scale dynamic resource allocation problems. *Eur J Oper Res*. 2017;263(2):664-678.
5. Browne CB, Powley E, Whitehouse D, et al. A survey of monte carlo tree search methods. *IEEE Trans Comput Intell AI games*. 2012;4(1):1-43.
6. Silver D, Schrittwieser J, Simonyan K, et al. Mastering the game of go without human knowledge. *Nature*. 2017;550(7676):354-359.
7. Schadd MPD, Winands MHM, Van Den Herik HJ, Chaslot GM-B, Uiterwijk JWHM. Single-player monte-carlo tree search. In: *International Conference on Computers and Games*. ; 2008:1-12.
8. Nijssen JPAM, Winands MHM. Enhancements for multi-player Monte-Carlo tree search. In: *International Conference on Computers and Games*. ; 2010:238-249.
9. Ontanón S. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In: *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*. ; 2013.
10. den Broeck G, Driessens K, Ramon J. Monte-Carlo tree search in poker using expected reward distributions. In: *Asian Conference on Machine Learning*. ; 2009:367-381.
11. García CG, Valdez ERN, Díaz VG, Bustelo BCPG, Lovelle JMC. A review of artificial intelligence in the Internet of Things. *Int J Interact Multimed Artif Intell*. 2019;5(4):9-20.
12. Enzenberger M, Muller M, Arneson B, Segal R. Fuego—an open-source framework for board games and Go engine based on Monte Carlo tree search. *IEEE Trans Comput Intell AI Games*. 2010;2(4):259-270.
13. Möller M, Schneider M, Wegner M, Schaub T. Centurio, a general game player: Parallel, Java-and ASP-based. *KI-Künstliche Intelligenz*. 2011;25(1):17-24.
14. M. Dieb T, Ju S, Yoshizoe K, Hou Z, Shiomi J, Tsuda K. MDTS: automatic complex materials design using Monte Carlo tree search. *Sci Technol Adv Mater*. 2017;18(1):498-503.
15. Yang X, Zhang J, Yoshizoe K, Terayama K, Tsuda K. ChemTS: an efficient python library for de novo molecular generation. *Sci Technol Adv Mater*. 2017;18(1):972-976.
16. Kocsis L, Szepesvári C. Bandit based monte-carlo planning. In: *European Conference on Machine Learning*. ; 2006:282-293.
17. Wang W, Sebag M. Multi-objective monte-carlo tree search. In: ; 2012.
18. Hassoun MH, others. *Fundamentals of Artificial Neural Networks*. MIT press; 1995.
19. Lea D. A Java fork/join framework. In: *Proceedings of the ACM 2000 Conference on Java Grande*. ; 2000:36-43.
20. Beck J, Beck J. *Combinatorial Games: Tic-Tac-Toe Theory*. Vol 114. Cambridge University Press; 2008.
21. Cheng F. The Platform Logging API and Service. In: *Exploring Java 9*. Springer; 2018:81-86.

22. Gupta S. *Logging in Java with the JDK 1.4 Logging API and Apache Log4j*. Springer; 2003.
23. Severance C. Discovering javascript object notation. *Computer (Long Beach Calif)*. 2012;45(4):6-8.
24. Soares J, Preguiça N. Database engines on multicores scale: A practical approach. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ; 2018:2335-2340.
25. Ellson J, Gansner E, Koutsofios L, North SC, Woodhull G. Graphviz—open source graph drawing tools. In: *International Symposium on Graph Drawing*. ; 2001:483-484.
26. Rocki K, Suda R. Large-scale parallel monte carlo tree search on GPU. In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. ; 2011:2034-2037.
27. Berlekamp ER, Conway JH, Guy RK. *Winning Ways for Your Mathematical Plays, Volume 4*. AK Peters/CRC Press; 2004.
28. Pilgrim RA. Tic-tac-toe: introducing expert systems to middle school students. *Acm Sigcse Bull*. 1995;27(1):340-344.
29. Xiong-xing RUI, Yi-li W. Application of UCT-RAVE algorithm in multi-player games with imperfect information. *Comput Eng Des*. 2012;(3):60.