Improving MCTS

https://github.com/Xavier0301/JassMCTS

Xavier Servot

May 2019

1 Design specifications

1.1 General abstract

The MCTS Jass algorithm can be described generally by the following pseudo-code:

```
Input: s_0 the current state of the game, it the number of iterations
Let n_0 be the root node, associated with the sate s_0;
for i ← 1 to it do
    // Selection and expansion;
    path ← TreePolicy(n_0);
    n_l ← NodeForPath(path);
    // Simulation;
    score ← DefaultPolicy(n_l);
    // Backpropagation;
    Backpropagate(score, path);
end
return Card(BestChild(n_0));
```

In our case, the Tree Policy is an adaptation of the UCT algorithm, which is a selection method for the next child. Given a parent node n_i, the child node selected n_j at each iteration is the one that maximizes the value

$$V(n_j) = \frac{S(n_j)}{N(n_j)} + c\sqrt{\frac{2\ln(N(n_i))}{N(n_j)}}$$

as stated in the sixth week assignment.

The Default Policy simply maps any given pair of state and action (a, s) so as to make each action equiprobable: $\pi_{default}(a|s) = \frac{1}{|\mathcal{A}(s)|}$, where $\mathcal{A}(s)$ is the set of actions for a given state s.

The MCTS algorithm can be improved in many ways:

- **Speed**: speed up the decision process, i.e. lowering the execution time of its `cardToPlay`. This is has been achieved here by parallelizing the algorithm with multiple techniques.
- **Collective intelligence**: have multiple weaker predictors for the next best card to play, and aggregating the results to hopefully have a strong predictor. This ties in with one parallelization technique as we will see later.
- **Qualitative intelligence**: simply trying to approximate more closely the actual value of possible outcomes. This is done here by modifying the Tree Policy and by modifying the Default Policy.

1.2 Abstract: Concurrency in Java

For concurrency, we used `ExecutorService` from `java.util.concurrent` in the following way:

1. Instantiating the service:

   ```
   ExecutorService service = Executors.newFixedThreadPool(_);
   ```

2. Add a task to be ran on a thread by the service:

   ```
   service.execute(new Runnable() {...});
   ```

   or

   ```
   service.submit(new Callable<T>() {...});
   ```

3. Stop the service:

   ```
   service.shutdown();
   ```

4. Wait for the each task to be completed:

   ```
   service.awaitForTermination(_,_);
   ```

The implementation details for parallelizing will be specified for each improvement.

1.3 Leaf parallelization: MCTSLP

1.3.1 Abstract

Leaf parallelization means running multiple simulation from a given selected leaf. It is a from of **speed improvement**. The implementation is easy as no communication is required between the threads. We have to wait for each simulation to end before we can go on to the next iteration of the algorithm.
We call this algorithm the Monte Carlo Search Tree, Leaf Parallelized (MCTSLP).

1.3.2 First considerations

A naive way to implement leaf parallelization would be to instantiate a new service at each iteration of the algorithm, and wait for it to shut down before going on to the next iteration:

```
while(iterations-->0) {
    // select and expand
    ...

    ExecutorService service = Executors.newFixedThreadPool(threads)
        ;
    for(int i=0; i<threads; i++) {
        service.execute(new Runnable() {
            // simulate and backpropagate
            ...
        });
    }
    service.shutdown();
    service.awaitTermination(_,_);
}
```

However, this implementation showed 2.5 to 3 times longer execution than the original MCTS implementation:

Version	Execution time (ms)
Original MCTS	170
Single threaded naive MCTSLP	446

The reason as to why being that there is a sizeable computational cost overhead associated with creating a new `ExecutorService`. The workaround is to not rely on the `.shutdown()` method of the API, and use a `CountDownLatch` from `java.util.concurrency` instead:

```
ExecutorService service = new Executors.newFixedThreadPool(threads)
    ;
while(iterations-->0) {
    // select and expand
    ...
```

```
    CountDownLatch latch = new CountDownLatch(threads);
    for(int i=0; i<threads; i++) {
        service.execute(new Runnable() {
                public void run() {
                // simulate and backpropagate
                        ...
                        latch.countDown();
                }
        })
    }
    latch.await();
}
service.shutdownNow();
```

1.3.3 Setup for benchmark

The method used for testing the execution time of our class is to sequentially run `cardToPlay(_,_)` 100 times and taking the average of the execution times. The given turn state and hand do not change from simulation to simulation to increase consistency and decrease randomness in our execution times.

Specifically, we ignore the first execution in our implementation as I consistently saw an abnormally high execution time for the first `cardToPlay(_,_)`. This is replicated with a regular `MctsPlayer`, which means that this does not come from a feature of `ExecutorService`, which might have had a reduced computational cost overhead once it has already been instantiated once.

The method used to choose the sample size (100) was to increment the sample size from 1 to a point where multiple different simulations over the sample size yielded a very close average execution time.

We chose 10 000 iterations for MCTS and MCTSLP in each case.

1.3.4 Results

On my machine `Runtime.getRuntime().availableProcessors()` is set to 8. This value indicates the number of available processors in the Java Virtual Machine (JVM). It is recommended to use the number of available processors for multithreading. The results are given here:

Version	Execution time (ms)
MCTS	170
MCTSLP 2 threads	172
MCTSLP 4 threads	117
MCTSLP 8 threads (available processor)	94
MCTSLP 16 threads	72
MCTSLP 32 threads	63
MCTSLP 64 threads	60
MCTSLP 128 threads	62
MCTSLP 256 threads	70

I must add that it would be a reasonable design choice to discriminate against a high number of threads in our given implementation, as the actual number of iterations is the given number of iterations divided by the number of threads: `actualIterations = iterations/threads`, which means we might not have explored the tree enough to make an informed decision.

However, using a timeout approach (i.e. it stops the process after a given timeout has passed), discriminating against the higher number of threads might not be as important. Yet still, as there is a points of diminishing returns after 32 threads, we will be sacrificing much exploration against not a lot more number of simulations. The timeout approach seems more reasonable in our project as each player's `cardToPlay(_,_)` is set to be executed in 2s via a `PacedPlayer` anyway.

1.4 Root parallelization: MCTSRP

1.4.1 Abstract

Root parallelization means running multiple instances of the MCTS algorithm and aggregating the result in the end. It is a form of **speed improvement** and **collective intelligence**, as we can set each instance of the MCTS algorithm to be weaker predictors by setting the iterations to $\frac{iterations}{threads}$, while aggregating the results from multiple instances at the end.
Again, the implementation is pretty straightforward as no communication is needed between the threads. We aggregate the results by summing them.
We call this algorithm the Monte Carlo Search Tree, Root parallelized (MCTSRP).

1.4.2 Considerations

In practice, we will use different APIs than for Leaf Parallelization, because we each thread needs to return a result. We thus make use of the `Future` interface from `java.util.concurrent`, which provides the method `.get()`. We will use in particular a `Future<SimulationResult>` where `SimulationResult` is given by

```
class SimulationResult {
    int[] childrenPoints;
    int[] childrenTurnsPlayed;

    public SimulationResult(int[] totalPoints, int[]
        totalTurnsPlayed) {
        this.childrenPoints = totalPoints;
        this.childrenTurnsPlayed = totalTurnsPlayed;
    }
}
```

The length of the arrays is the number of children that the root node has and `childrenPoints[i]` designates the points collected in simulations by the i-th child of the root node. When we call the method `.get()`, it blocks until the thread returns a value, so we effectively wait for the longest thread to be done before we combine the results. This also means we won't need to use a `CountDownLatch`. The general structure of the method `cardToPlay(_,_)` is given by:

```
List<Future<SimulationResult>> futureResults = new ArrayList<Future
    <SimulationResult>>();
ExecutorService service = Executors.newFixedThreadPool(threads);

for(int i=0; i<threads; i++) {
    futureResults.add(service.submit(new Callable<SimulationResult
        >() {
        public SimulationResult call() {
            // selection, expansion, simulation, backpropagation
            ...
```

```
            return new SimulationResult(...);
        }
    }));
}
List<SimulationResult> results = new ArrayList<SimulationResult>();
for(Future<SimulationResult> futureResult: futureResults)
    results.add(futureResult.get());

service.shutdownNow();

int numberOfPossibilities = results.get(0).childrenPoints.length;

int[] summedPoints = new int[numberOfPossibilities];
int[] summedTurnsPlayed = new int[numberOfPossibilities];

// we sum over the set of MCTS instances
...

int bestChildIndex = getIndexOfBestChild(summedPoints,
    summedTurnsPlayed);

return state.trick().playableCards(hand).get(bestChildIndex);
```

where `getIndexOfBestChildMethod(_,_)` is specified by:

Input : $(S_1, \ldots, S_k)$ and $(N_1, \ldots, N_k)$ the list of points and turns
// the index of the best possibility;
let $j \longleftarrow -1$;
// the value of the best possibility;
let $v_* \longleftarrow -1$;
for $i \leftarrow 1$ **to** k **do**
  let $v_i \leftarrow \frac{S_i}{N_i}$;
  if $v_i > v_*$ **then**
    $v_* \longleftarrow v_i$;
    $j \longleftarrow i$;
end
Output: j

Algorithm 1: Index of max value

What do I mean by "we sum over the set of MCTS instances"? Given that the number of points and the number of turns played for the j-th child of the i-th instance are $S_{n_i,j}$ and $N_{i,j}$ respectively, we can defined the aggregated points of the j-th child as

$$S_j = \sum_{i=1}^{threads} S_{i,j}$$

and the aggregated number of turns played as

$$N_j = \sum_{i=1}^{threads} N_{i,j}$$

1.4.3 Results

The setup for the results is the same as for MCTSLP, including the same turn state and hand, which means that the result can be compared.

Version	Execution time (ms)
MCTS	170
MCTSRP 2 threads	91
MCTSRP 4 threads	54
MCTSRP 8 threads (available processor)	50
MCTSRP 16 threads	53
MCTSRP 32 threads	54
MCTSRP 64 threads	57
MCTSRP 128 threads	59
MCTSRP 256 threads	64

As a reminder, the results for MCTSLP are given by

Version	Execution time (ms)
MCTS	170
MCTSLP 2 threads	172
MCTSLP 4 threads	117
MCTSLP 8 threads (available processor)	94
MCTSLP 16 threads	72
MCTSLP 32 threads	63
MCTSLP 64 threads	60
MCTSLP 128 threads	62
MCTSLP 256 threads	70

We can see a vast improvement in speed from one to the other, in addition to on between MCTS and MCTSRP.

Again, we must discriminate against a higher number of threads, as each instance of MCTS run by thread is given $\frac{iterations}{threads}$ actual iterations. The best execution time is for 8 threads, which is not very high anyway, thus using more than that would be pointless on my machine.

We can make the same point in advocating for a timeout implementation for the player instead of a number of iterations for the same reasons as in MCTSLP.

1.5 Enhanced default policy

1.5.1 Abstract

In our original implementation of the MCTS algorithm, we set the default policy to be random ($\pi_{default}(a|s) = \frac{1}{\mathcal{A}(s)}$). While it's very easy to code and fast, it does not orient the simulations towards particularly good insights.
Replacing it means finding a fast way to make better than random decisions in average, i.e. it is a form of **qualititve intelligence** improvement. We implement in this section two algorithms that meet the criterion, the first being composed of a hand made heuristic, and the second implementing the minimax algorithm. In particular, we code two new players: `SimplePlayer` and `ShallowMinimaxPlayer`, which we will instantiate in the simulation function of a modified MCTS algorithm.

1.5.2 Considerations: hand-made heuristic (HandMade)

The motivations to create the heuristic presented here was to emulate a weak player, who does not have any intuition about the game and acts in the following way:

- If his teammate has already played a card:
  - He plays a more valuable one if he can.
  - He plays its weakest playable card otherwise.
- Otherwise if some of his card are better than all of the opponents':
  - He plays the most valuable one of these.
- If none of his cards is better
  - He plays his weakest card.

We refer to this algorithm as HandMade.

1.5.3 Considerations: Minimax algorithm (MinMax)

The intuition behind Minimax is basically that at each step, the opponent tries to minimize the value you get while you try to maximize it. The value is defined as the difference of the score of the team's player between before and after one trick.

In other words in this algorithm, we consider every possibility for only one trick, and we find the best card taking into account that each time the opponent plays he will try to play a card that minimizes your score over the set of his playable cards. Considering every possibility is done with a graph, like for MCTS.

We call the player `ShallowMinimaxPlayer` because the Minimax Algorithm could have any depth we want, yet we constrain it to have depth at most 4 (i.e. one full trick at most) to gain speed. The algorithm is in $O(b^d)$, where d is the depth and b is the number of possibilities, which explains why it is so important for our implementation to be shallow. We will thus define the *terminal depth* as the depth for which the trick is full.

Initially, I implemented the Minimax algorithm about the same way MCTS was implemented (i.e. using the same data structure), but I quickly realised that the performance where not great because of how we had to backtrack to compute the values of the nodes in the tree.

I decided to replace the initial implementation by a more efficient dimension 2 tensor, where the i-th row of the tensor corresponded to nodes of depth i. Yet the pseudocode for MiniMax stays the same:

Input: s_0 the current state of the game
Let t be a tree;
Let n_0 be the node associated with the sate s_0;
Add n_0 to the tree as the root node;
Populate(t);
ComputeValues(t);
return GetBestChild(n_0);

In particular, computing the values is done differently depending on the depth:

- If the depth is terminal, the value of a node is its state's score minus the initial state's score:

```
return score.totalPoints(teamId) - initialScore.totalPoints(
    teamId);
```

- If the depth is not terminal, the node has children.
  - If `node.state.nextPlayer()` is of opponent team, then the value is the minimum value of its children:

```
double minValue = 2000;
for(int i=0; i<children.size(); i++) {
    if(children.get(i).value < minValue)
        minValue = children.get(i).value;
}
return minValue;
```

  - If `node.state.nextPlayer()` is of our team, then the value is the maximum value of its children:

```
double maxValue = -1;
for(int i=0; i<children.size(); i++) {
    if(children.get(i).value > maxValue)
        maxValue = children.get(i).value;
}
return maxValue;
```

In our final implementation, the tree is of from

```
List<List<MinimaxNode>> tree = new ArrayList<List<MinimaxNode>>();
```

where `MinimaxNode` is of form

```
class MinimaxNode {
    TurnState state;
    PlayerId ownId;

    // We keep track of the index as we use a dimension 2 tensor
    // instead of a dimension 3 one.
    int childrenStartIndex;
    int childrenEndIndex;

    ...
}
```

We refer to this algorithm as MinMax.

1.5.4 Setup for benchmark

We ultimately want to know two things about each new default policy:

- How good it is against the random one.
- How fast it is against the random one.

We simulate lots of games where one team consists only of players implementing a new default policy (namely `ShallowMinimaxPlayer` or `SimplePlayer`), while the other consists of players playing at random (`RandomPlayer`).

How good a policy is is going to be measure by the percentage of win against the players playing at random, while we will measure the average time it takes for a new policy player to compute its `cardToPlay`. The following interface defines what is necessary for the benchmark:

```
public interface PlayerBenchmarkable {
    public int getNumberOfWins();
    public long getTotalExecutionTime();
    public long getTotalNumberOfExecutions();
}
```

I simulated 10 000 games for `ShallowMinimaxPlayer` and 100 000 games for `SimplePlayer`.

1.5.5 Results

The results are

Algorithm	Wins (%)	Execution time (ns)
MinMax	84.6	424 000 ns
HandMade	84.7	405 ns
Random	-	167 ns

I can safely say that HandMade beating the percentage wins of MinMax was very unexpected, and somewhat difficult to explain for me. Maybe it comes from the fact that Jass is a game where teams have little interest in minimizing the opponent's score, but where it is more about maximizing the coefficient

$$\frac{\text{own team points}}{\text{opponent team points}}$$

rendering the value used in our implementation of MinMax useless. I tested for this on a 1000 games and here is the result:

Algorithm	Wins (%)	Execution time (ns)
Fraction MinMax	92.6	432 000 ns

which is more of what is expected considering that the HandMade performed so well. While the new results are more promising, MinMax is still a slow algorithm compared to Random. If we were to replace the default policy by a minmax heuristic policy, we would have an MCTS algorithm that takes more than 2000 as long to compute than originally, which is unacceptable.

However, the HandMade is very promising: while it is has lesser wins than Fraction MinMax, its execution time for computing the card on average is 405 ns while it takes 167 ns to compute the card for the random policy. Given the execution time of MCTS in the previous sections, we know we can afford to have a simulation that is 2.5 times as long as originally.

1.6 Enhanced tree policy

1.6.1 Abstract

The tree policy is the heuristic for node selection. The *regret* of a player is the expected loss due to not playing the best card. We are using the UCB1 heuristic for our tree policy, which selects the node that maximizes

$$\textbf{UCB1}(n_j) = \frac{S(n_j)}{N(n_j)} + c\sqrt{\frac{2\ln(N(n_i))}{N(n_j)}}$$

It can be shown that UCB1 guarantees an expected regret of $O(\log n)$ (shown by Auer, Cesa-Bianchi and Fisher (2002)), while a lower bound for regret has been established at $\Omega(\log n)$ by Lai and Robbisn (1989), i.e. UCB1 is optimal up to a constant.
Its authors Auer, Cesa-Bianchi and Fisher provided an other algorithm called UCB1-Tuned, which they claimed provided better results in practice. The tree policy under the UCB1-Tuned heuristic selects a node which maximizes:

$$\textbf{UCB1-Tuned}(n_j) = \hat{\mu_j} + c\sqrt{\frac{\ln(N(n_i))}{N(n_j)}\min(\frac{1}{4}, \hat{\sigma_j}^2 + \sqrt{\frac{2\ln(N(n_i))}{N(n_j)}})}$$

where $\hat{\mu_j}$ is the empirical reward and $\hat{\sigma_j}$ is the empirical variance.
We implement UCB1-Tuned and provide a new reward, thus making an improvement of **qualitative intelligence**.

1.6.2 Motivations: a new reward

We will explain our intention regarding the new reward, but most of the implementation details will be present in the next section.

We saw in the previous section that modifying the reward from the team's score to the fraction $\frac{\text{own team points}}{\text{opponent team points}}$ provided significant improvements in the result. We choose to do the same for MCTS.

1.6.3 Considerations: UCB1-Tuned and the new reward

For UCB1, we only had to memorize the sum of rewards, represented by the mapping S, and the number of turns played, represented by the mapping N. We could compute the empirical reward as

$$\hat{\mu_j} = \frac{S(n_j)}{N(n_j)}$$

However, given that the new empirical reward should be the average fraction of scores, we know that we can't just memorize the score and the opponent score

$(\overline{S})$ and compute the empirical value like $\frac{S(n_j)}{\overline{S}(n_j)}$ because that would be wrong.

We need to directly memorize the sum of fractions in a mapping T, and compute the empirical reward as

$$\hat{\mu_j} = \frac{T(n_j)}{N(n_j)}$$

Another problem comes with having the new UCB1-Tuned heuristic, because we need to compute the empirical variance $\hat{\sigma}^2$. Now, we recall that for two random variables X, Y, we have

$$\mathbb{V}[\frac{X}{Y}] = \mathbb{E}[(\frac{X}{Y})^2] + \mathbb{E}[\frac{X}{Y}]^2$$

Thus we also need to memorize the sum of squared fractions in a mapping U, and the variance becomes:

$$\hat{\sigma}^2(n_j) = \frac{U(n_j)}{N(n_j)} - (\frac{T(n_j)}{N(n_j)})^2$$

Which means the heuristic can be rewritten as:

$$\textbf{UCB1-Tuned}(n_j) = \frac{T(n_j)}{N(n_j)} + c\sqrt{\frac{\ln(N(n_i))}{N(n_j)} \min\{\frac{1}{4}; \frac{U(n_j)}{N(n_j)} - (\frac{T(n_j)}{N(n_j)})^2 + \sqrt{\frac{2\ln(N(n_i))}{N(n_j)}}\}}$$

1.7 Summing it up

1.7.1 Considerations

We showed many ways to improve the MCTS algorithm. We now want to incorporate them all in a single class `ImprovedMctsPlayer`:

- I chose to use root parallelization instead of leaf parallelization. The results show that root parallelization is more promising in terms of execution time. Specifically, it took 50 ms to run MCTSRP compared to 94 ms for MCTSLP on a large sample of a special case. Most importantly, leaf parallelization would be useless when using a deterministic default policy, instead of the current stochastic one.
- The algorithm selects the nodes with the UCB1-Tuned heuristic, which takes the variance of the values into account and not just the empirical mean. The results for this heuristic are not clearly better than for the UCB1 heuristic: the number of games that I could simulate on my computer was too low to draw a clear conclusion.
- The algorithm incorporates the improved default policy `SimplePlayer`, which has roughly 85% wins against the previous Random Walk. This new default policy should orient the simulations towards a more accurate representation of a game of Jass.

2 Future work

Many things can still be done to improve the MCTS algorithm for Jass. I tried ranking them from easiest to harset.

1. Implement a version where the algorithm stops after a given timeout, instead of a given number of iterations. This is especially good in our case: every player is a PacedPlayer, which means we can give 2s of timeout to the MCTS players without hurting the flow of the game.

2. Using another (more powerful) machine to make simulations. Sleeping with a struggling computer is only fun for so long. (Better) results could be obtained that way.

3. Making simulations to maybe determine a better constant c for the UCB1-Tuned heuristic.

4. Implement more avanced parallelization techniques such as tree parallelization with global and local mutexes.

5. Try to use an AMAF score alongside the UCB1-Tuned score, with something like α-AMAF.

6. Try different selection methods (other than UCB1 and UCB1-Tuned). Try stochastic ones.

7. I did not look into improving backpropagation, but I know it is something to be tested.

8. Use online knowledge to improve the default policy, like MAST.

9. A very hard problem would be to use offline knowledge to improve the default policy, like a neural network (cf AlphaZero)

3 References

1. "A Survey of Monte Carlo Tree Search Methods" by *Cameron Browne, Member, IEEE, Edward Powley, Member, IEEE, Daniel Whitehouse, Member, IEEE, Simon Lucas, Senior Member, IEEE, Peter I. Cowling, Member, IEEE, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton*; in IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 1, MARCH 2012.

   http://mcts.ai/pubs/mcts-survey-master.pdf

   Gave me a high level view of the algorithm.

2. "Comparison of Different Selection Strategies in Monte-Carlo Tree Search for the Game of Tron" by *Pierre Perick, David L. St-Pierre, Francis Maes and Damien Ernst.*

   http://www.montefiore.ulg.ac.be/ dlstpierre/publications/tron2012.pdf

   Helped me figure understand the tree policies better.

3. "Algorithms for the multi-armed bandit problem" by *Volodymyr Kuleshov and Doina Precup* in the Journal of Machine Learning Research 1 (2000).

   https://www.cs.mcgill.ca/ vkules/bandits.pdf

   Gave me a better understanding of the tree policies. More generally enabled me to get the basics right.

4. "Online Knowledge Enhancements for Monte Carlo Tree Search in Probabilistic Planning" by *Marcel Neidinger*

   https://ai.dmi.unibas.ch/papers/theses/neidinger-bachelor-17.pdf

5. "On Monte Carlo Tree Search and Reinforcement Learning", by *Tom Vodopivec, Spyridon Samothrakis and Branko Ster* in the Journal of Artificial Intelligence Research 60

   https://pdfs.semanticscholar.org/3d78/317f8aaccaeb7851507f5256fdbc5d7a6b91.pdf

   Allowed me to bridge the knowledge in RL and the knowledge in MCTS

6. A lecture form the University of Washington:

   https://courses.cs.washington.edu/courses/cse599i/18wi/resources/lecture19/lecture19.pdf

   Shows some nice theoretical considerations about the regret of UCT. Explains the enhancements made by the Deepmind team on AlphaZero.