Morpion Solitaire

Developer's guides

Quyen Linh TA

Ha Anh TRAN

1. Introduction

Morpion Solitaire, also known as Join Five, is a game played alone with a pencil and paper, and it is popular in several countries. A move in this game consists of drawing a cross and a line segment on an infinite square lattice. The line segment has to pass through exactly five consecutive crosses including the one that has just been placed. The objective is to make as many moves as possible starting from a given initial configuration. We call the number of moves the score. There are two variants of this game according to how two line segments can touch each other. Demaine et al. studied generalizations of the game and their computational complexity, and show that a generalized Morpion Solitaire is NP-hard and that its maximum score is hard to approximate. Another target of interest is the maximum scores or their lower and upper bounds. Recently, computing maximum scores was used as a test problem to evaluate the effectiveness of the Monte-Carlo tree search method, which has been attracting rising attention as a promising approach in game programming.

2. Execution program

Requirement

JavaFX SDK 19 installed [all modules expected]

Execution

You should be able to run the .jar file with the following command (from the command line)

```
$ >>
java --module-path C:\Java\javafx-sdk-19\lib --add-modules
javafx.controls,javafx.fxml,javafx.swing -jar out/artifacts/jar/MPSL.jar
```

Of course, you'll need to modify the module path to where you installed JavaFX.

In the case, your machine does not support JavaFX modules set-up runtime, please using Eclipse or any editor at your disposition to execute program.

3. Implementation

a. Nested Monte-Carlo Search

The basic idea of Nested Monte-Carlo Search (NMC) is to find a solution path of cities with the particularity that each city choice is based on the results of a lower level of the algorithm. At level 1, the lower level search is simply a playout (i.e. each city is chosen randomly). Figure 1 illustrates a level 1 Nested Monte-Carlo search. Three selections of cities at level 1 are shown. The leftmost tree shows that, at the root, all possible cities are tried and that for each possible decision a playout follows it. Among the three possible cities at the root, the rightmost city has the best result of 30, therefore this is the first decision played at level 1. This brings us to the middle tree. After this first city choice, playouts are performed again for each possible city following the first choice. One of the cities has result 20 which is the best playout result among his siblings. So the algorithm continues with this decision as shown in the rightmost tree. This algorithm is presented in Algorithm 1.

At each choice of a playout of level 1 it chooses the city that gives the best score when followed by a single random playout. Similarly for a playout of level n it chooses the city that gives the best score when followed by a playout of level $n - 1$.

b. Nested Rollout Policy Adaptation (NRPA)

The Nested Rollout Policy Adaptation algorithm (NRPA) is an algorithm that learns a playout policy. There are different levels in the algorithm. Each level is associated to the best sequence found at that level. The playout policy is a vector of weights that are used to calculate the probability of choosing a city. A city is chosen proportionally to exp(pol[code(node,i)]). pol(x) is the adaptable weight on code x. code(node,i) is a unique domain-specific integer leading from a situation node to its ith child. This is comparable with previous known learning of Monte-Carlo simulations. Learning the playout policy consists in increasing the weights associated to the best cities and decreasing the weights associated to the other cities. The algorithm is given in algorithm 2.

Algorithm 1 Nested Monte-Carlo search

```
nested (level,node)
if level==0 then
  ply ← 0
  seq ← {}
  while num_children(node) > 0 do
    CHOOSE seq[ply] ← child i with probability 1/num_children(node)
    node ← child(node,seq[ply])
    ply ← ply+1
  end while
  RETURN (score(node),seq)
else
  ply ← 0
  seq ←{}
  best_score ← ∞
  while num_children(node) > 0 do
    for children i of node do
      temp ← child(node,i)
      (results,new) ← nested(level-1,temp)
      if results<best_score then
        best_score ← results
        seq[ply]=i
        seq[ply+1...]=new
      end if
    end for
    node=child(node,seq[ply])
    ply←ply+1
  end while
  RETURN (best_score,seq)
end if
```

Algorithm 2 Nested Rollout Policy Adaptation

```
NRPA ($level$,$pol$)
if $level = 0$ then
  $node \leftarrow$ root
  $ply \leftarrow 0$
  $seq \leftarrow \{\}$
  while there are possible moves do
    CHOOSE $seq[ply] \leftarrow$ child i the with probability proportional to
    exp($pol$[code($node$,i)])
    $node \leftarrow$ child($node$, $seq$ [$ply$])
    $ply \leftarrow ply + 1$
  end while
  return (score ($node$), $seq$)
else
  $bestScore \leftarrow \infty$
  for N iterations do
    (result,new) $\leftarrow$ NRPA ($level - 1$, $pol$)
    if result $\leq bestScore$ then
      $bestScore \leftarrow$ result
      $seq \leftarrow$ new
    end if
    $pol \leftarrow$ Adapt($pol$,$seq$)
  end for
end if
return ($bestScore$,seq)

Adapt ($pol$,$seq$)
$node \leftarrow$ root
$pol' \leftarrow pol$
for $ply \leftarrow 0$ to length($seq$) - 1 do
  $pol'$[code($node$,$seq[ply]$)] += Alpha
  $z \leftarrow$ SUM exp($pol$[code($node$,i)]) over node's children i
  for children i of $node$ do
    $pol'$[code($node$,i)] -= Alpha $\times$ exp($pol$[code($node$,i)]) / $z$
  end for
  $node \leftarrow$ child($node$, $seq$ [$ply$])
end for
return $pol'$
```

Policies: A policy is a probability distribution p over a set of moves M that is conditioned on the current game state $s \in S$. For example, we often consider the uniform policy p0, which assigns equal probability to all the moves that are legal in state s. I.e. $p0(m|s) = 1\ |Ms|$. In this paper, we also consider policies probability distributions pW which are parameterized with a set of weights W. There is one real valued weight for each possible move, i.e. $W = wm1, \ldots, wm|M|$, and the probability $pW\ (m|s)$ is defined as follows:

$$p_W(m|s) = \frac{e^{w_m}}{\sum_{p \in M_s} e^{w_p}}$$

The softmax function enables to calculate the gradient for all the possible weights associated to the possible moves of a state and to learn a policy in NRPA using gradient descent. Finally, we also consider more complex policies πθ in which the probability of each move depends on a function of the state, represented using a neural network. Let fθ : S → R |M| be a neural network parameterized with θ, we can then define policy πθ as follows:

$$\pi_\theta(m|s) = \frac{e^{(f_\theta(s))_m}}{\sum_{p \in M_s} e^{(f_\theta(s))_p}}$$

As most Monte-Carlo based algorithms, Nested Monte Carlo Search (NMCS) and Nested Rollout Policy Adaptation (NRPA) both generate a large number of random sequences of moves. The best sequence according to the scoring function is then returned as a solution to the problem. The quality of the final best sequence directly depends on the quality of the intermediate random sequences generated during the search, and thus on the random policy. Therefore NMCS and NRPA have introduced new techniques to improve the quality of the policy throughout the execution of the algorithm. NMCS and NRPA are both recursive algorithms, and at the lowest recursive level, the generation of random sequences is done using playouts parameterized with a simple stochastic policy.

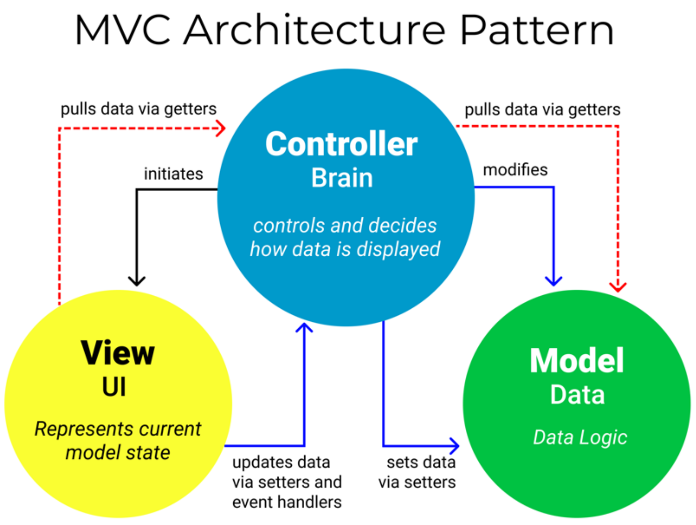
If the user has access to background knowledge, it can be captured by using a non-uniform policy (typically by manually adjusting the weights W of a parameterized policy pW). Otherwise, the uniform policy p0 is used. In NMCS, the policy remains the same throughout the execution of the algorithm. However, the policy is combined with a tree search to improve the quality over a simple random sequence generator. At every step, each possible move is evaluated by completing the partial solution into a complete one using moves sampled from the policy. Whichever intermediate move has led to the best completed sequence, is selected and added to the current sequence. The same procedure is repeated to choose the following move, until the sequence has reached a terminal state.

A major difference between NMCS and NRPA, is the fact that NRPA uses a stochastic policy that is learned during the search. At the beginning of the algorithm, the policy is initialized uniformly and later improved using gradient descent based the best sequence discovered so far. The policy weights are updated using gradient descent steps to increase the likelihood of the current best sequence under the current policy. Finally, both algorithms are nested, meaning that at the lowest recursive level, weak random policies are used to sample a large number of low quality sequences, and produce a search policy of intermediate quality. At the recursive level above, this policy is used to produce sequences of high quality. This procedure is applied recursively. In both algorithms the recursive level (denoted level) is a crucial parameter. Increasing level increases the quality of the final solution at the cost of more CPU time. In practice it is generally set to 4 or 5 recursive level depending on the time budget and the computational resources available.

4. Structure Model-View-Controller (MVC)

In the Model-View-Controller (MVC) design pattern, the model represents the data and the business logic of the application, the view represents the user interface, and the controller handles the input and communicates with both the model and the view.

In Java, the MVC pattern can be implemented using a variety of approaches. One way to implement MVC in Java is to have the model be represented by a JavaBean, which is a Java object that has properties that can be set and retrieved using getter and setter methods. The view can be implemented using a Java Swing component or a JavaFX component, and the controller can be implemented as a separate Java class. The controller can listen for events on the view and update the model as necessary, and it can also update the view when the model changes.



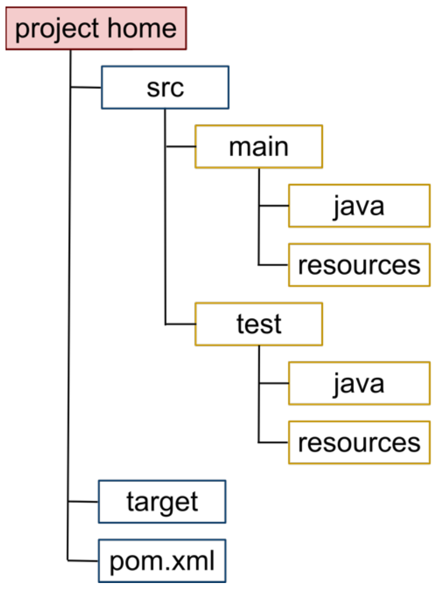
5. Maven project

Maven is a build automation tool used primarily for Java projects. Some benefits of using Maven include:

- Maven can manage the entire build process, including compilation, testing, and packaging.
- Maven uses a standard directory layout and a default build lifecycle, which makes it easy for users to understand how a Maven project is structured.

- Maven uses a declarative approach to build configuration, which means that users only need to specify what needs to be done, rather than how it should be done.
- Maven integrates well with other tools, such as continuous integration servers, making it easier to build and deploy projects.
- Maven provides a large repository of pre-built libraries and plugins, which can be easily included in a project's build.

Overall, Maven helps to make the build process more efficient and standardized, which can be particularly useful for larger projects with multiple developers.



6. Functional Implementation

a. NMCS Search

This is a recursive function that performs the Monte Carlo Search (MCS) algorithm on a given state. MCS is a heuristic search algorithm that uses random simulations to find the best possible action for a given state in a game or other decision-making process.

The function takes in three arguments:

- **state**: an object representing the current state of the game or decision-making process
- **level**: an integer representing the number of levels of recursion to perform
- **isCanceled**: a supplier function that returns a boolean indicating whether the search should be canceled

The function first checks if the recursion level is 0, in which case it returns the result of a simulation of the game from the current state. If the recursion level is not 0, the function enters a loop that continues until the current state is a terminal state (i.e., the game has ended) or the search is canceled.

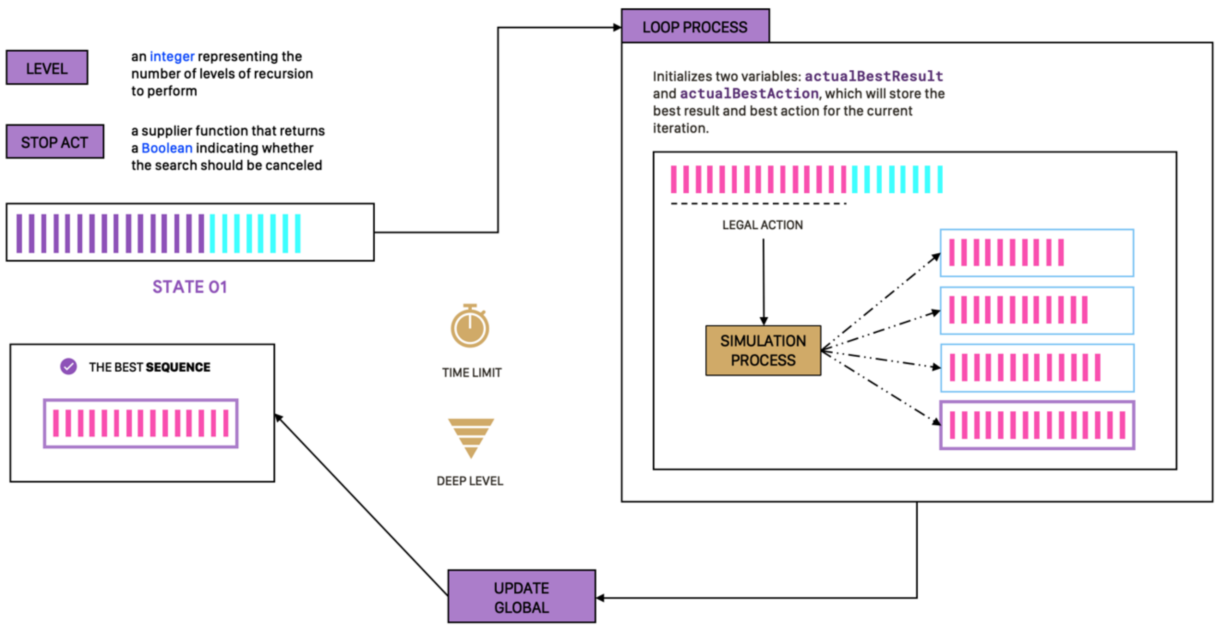
In each iteration of the loop, the function performs the following steps:

1) Initializes two variables: **actualBestResult** and **actualBestAction**, which will store the best result and best action for the current iteration.
2) Iterates over all legal actions in the current state. For each action:
    a. Takes the action and gets the new state (child)
    b. Simulates the game from the new state (child)
    c. If the score of the simulation is better than the current best score for the iteration, updates **actualBestAction** and **actualBestResult** with the action and simulation result, respectively.
3) If the best result for the current iteration is better than the best result for the whole search (or if it's the first iteration), updates the global best result with the current best result and adds the current best action to the list of visited nodes.
4) If the best result for the current iteration is not better than the global best result, gets the best action from the global best result and adds it to the list of visited nodes.
5) Takes the best action for the current iteration and gets the new state (child).
6) Returns the global best result when the loop is finished.

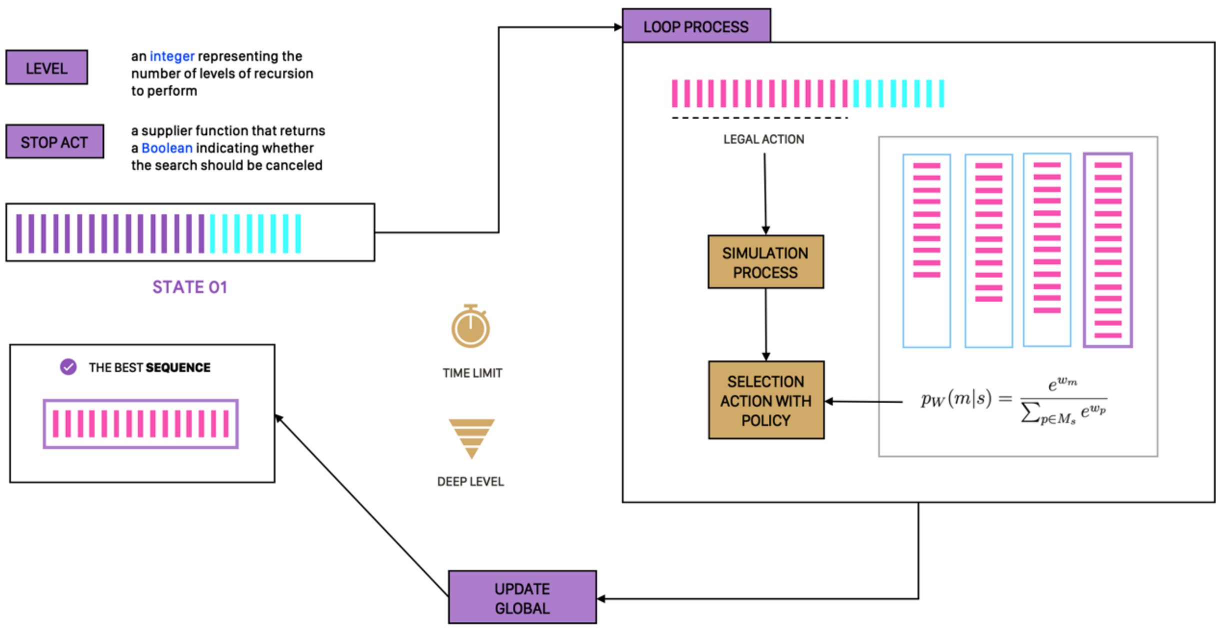
b. NRPA

To be implementing a recursive function called **_searchNestedPolicy** that takes in a state, a level, a policy, and a supplier of Booleans as input. The function uses the input level to determine how many levels of recursion should occur. The function has a base case where it returns a result if the input level is less than or equal to 0 or if the supplier's Boolean value is true. Otherwise, the function initializes a best result variable to a default value and then enters a loop that iterates N times. On each iteration, the function calls itself with updated input values and compares the returned result to the current best result. If the returned result has a higher key value, the function updates the best result and adapts the policy using the state and the best new result. Finally, the function returns the best result.

NESTED MONTE CARLO SEARCH


LEVEL
an integer representing the number of levels of recursion to perform
STOP ACT
a supplier function that returns a Boolean indicating whether the search should be canceled
STATE 01
THE BEST SEQUENCE
TIME LIMIT
DEEP LEVEL
LOOP PROCESS
Initializes two variables: actualBestResult and actualBestAction, which will store the best result and best action for the current iteration.
LEGAL ACTION
SIMULATION PROCESS
UPDATE GLOBAL

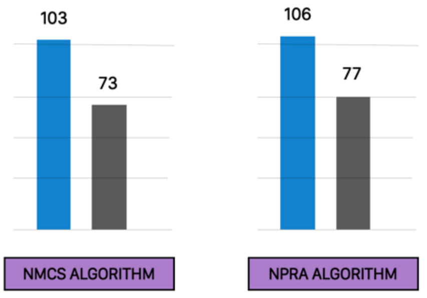
NESTED ROLLOUT POLICY ADAPTATION


LEVEL
an integer representing the number of levels of recursion to perform
STOP ACT
a supplier function that returns a Boolean indicating whether the search should be canceled
STATE 01
THE BEST SEQUENCE
TIME LIMIT
DEEP LEVEL
LOOP PROCESS
LEGAL ACTION
SIMULATION PROCESS
SELECTION ACTION WITH POLICY
$p_W(m|s) = \frac{e^{w_m}}{\sum_{p \in M_s} e^{w_p}}$
UPDATE GLOBAL

NMCS & NRPA
HIGHEST SCORE

🟦 GRID 5T

⬛ GRID 5D



BEST GRID