Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 29–34.

Object-oriented Approach of Search Algorithms for Two-Player Games

Márk Kósa, János Pánovics

University of Debrecen
Faculty of Informatics
`kosa.mark@inf.unideb.hu`
`panovics.janos@inf.unideb.hu`

Abstract

The theory of two-player strategic games is a part of the introductory course of Artificial Intelligence at the University of Debrecen. During the seminars, we demonstrate to the students the minimax and negamax algorithms with or without alpha-beta pruning.

In this paper, we present a possible object-oriented approach to implement these algorithms. It is about a Java class hierarchy which makes it easier to understand the operation of the algorithms described at the lectures of this subject, and helps our students prepare for their exams.

Keywords: artificial intelligence, state space representation, algorithms for choosing the next move, minimax algorithm, negamax algorithm, alpha-beta pruning, object-oriented programming, class diagram

MSC: 68T20

1. Introduction

The games in which the players do not have influence to the outcome of the game are called *gambles* (e.g. roulette, dice games). Research on these games acted as a base of probability theory in the 17th century. Contrarily, in *strategy games* (e.g. chess, bridge), the outcome of the game is actively affected by the players. Artificial intelligence deals with such games—among other topics. Strategy games can be classified by the following aspects:

- Considering the number of players, there are *two-player*, *three-player*, ..., *n-player* games.
- Considering the length of the game, there are *finite* games in which each player can choose from a finite number of moves, and each game terminates

in a finite number of moves. Those games which are not finite are called *infinite* games.

- Considering the sum of the players' gains and losses, there are *zero-sum* and *non-zero-sum* games. In zero-sum games, the sum of the players' gains and losses is zero.

- If a game has random factors, it is called *stochastic*, otherwise *deterministic*.

- In a game with *perfect information*, the players have all information related to the game at their disposal. A game with *imperfect information* does not have perfect information.

In the frame of the intorductory course of Artificial Intelligence, we deal only with finite, deterministic, zero-sum, two-player strategy games with perfect information. In the practice courses, we create state space representations for a couple of such games first, then implement them in Java programming language, and finally test them. To make the implementation easier, we developed a Java package which we can use to play an arbitrary game using an arbitrary search algorithm after correctly parameterizing it.

2. The Class Hierarchy

In this section we present the class hierarchy used in the practice courses. The recommended Java packages contain classes related to the state space representation and the algorithms for the choosing tje next move (see Figure 1).

The `Operator` class in the `allapotter` package is the abstract superclass of the operators for each game, while the `Allapot` class is the abstract superclass of the games themselves. As the properties of a state always depend on the particular game, we define only two attributes in the latter class. One of them is the set of all operators applicable to the states of the game, and the other one represents the player which is in turn in the current state. Independently of the game, each state knows if it is a final state, and if so what the result of the game is (which player wins or whether the game is a tie). We can also examine if an operator is applicable to the state or not, and we can give the state which derives from the current state by applying the operator. We can realize the human players' input by implementing the abstract `beker` method.

Classes implementing the algorithms related to games are placed in the `jatek` package. The superclass of the algorithms for choosing the next move is the abstract `LepesAjanlo` class. The attributes of these class store the *state*, the *depth* of the search, the suggested *operator*, the *utility functions's value* of the state reached by the suggested operator, and the *number of states* evaluated during the search. The values of these attributes will be computed by the concrete algorithms implemented in the derived classes. These algorithms can be compared to one another with the help of the number of evaluated states.

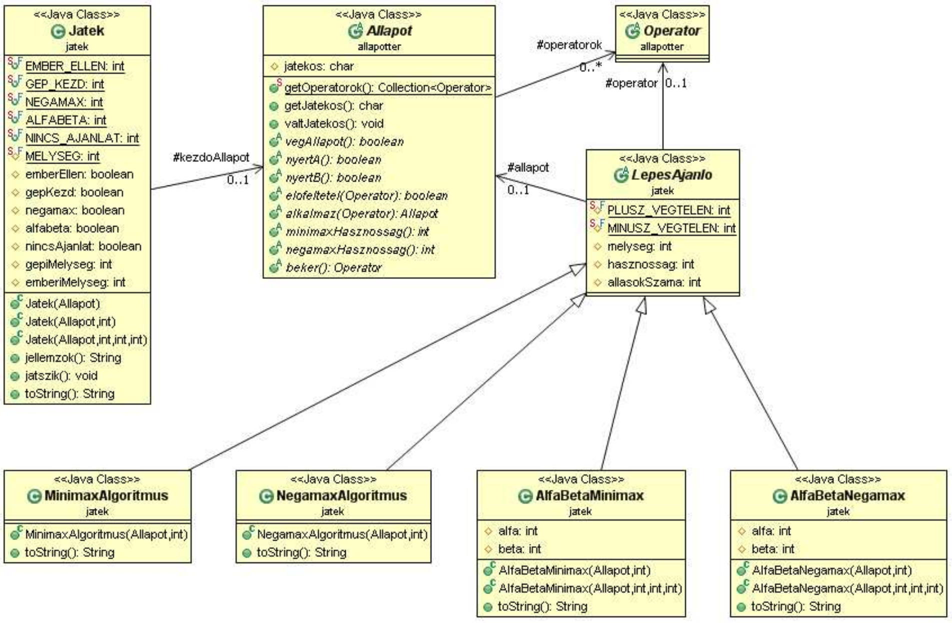
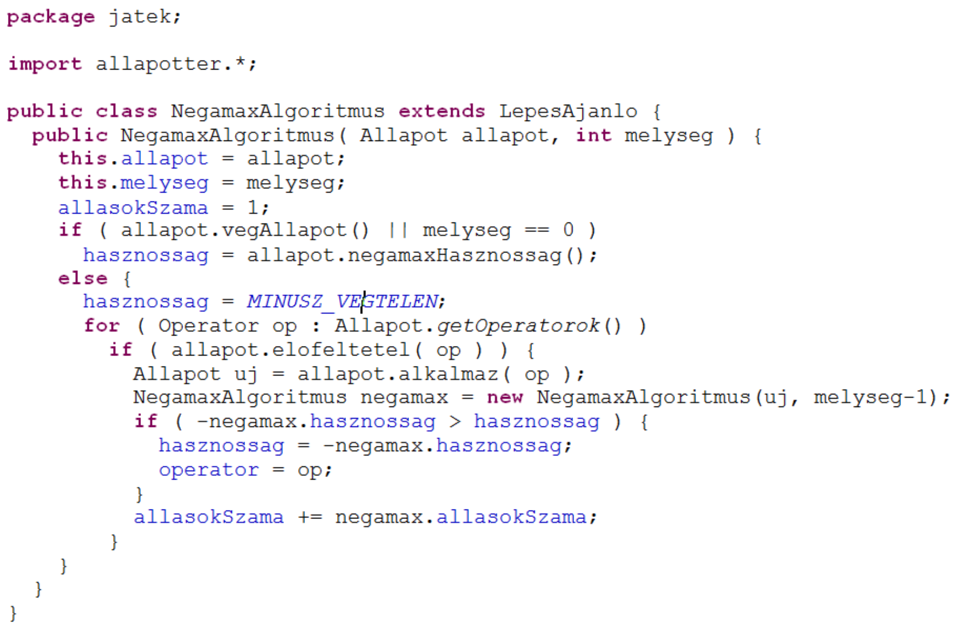


Figure 1: Class diagram of the search algorithms.

The concrete algorithms for choosing the next move work recursively by instatiating objects of the concrete algorithm classes. Because of this, there was no need for methods other than the constructors (see the source code of the negamax algorithm in Figure 2). Classes of algorithms using alpha-beta pruning have been extended by two attributes which represent the *alpha* and *beta* values that belong to each node of the game tree.

The `jatszik` method defined in the `Jatek` class is responsible for controling the game. The body of this method contains a loop which runs until the current state instance is a final state. While in the loop, it displays the current state and determines which player is in turn. If we play interactively and the human player is in turn, it will read the player's move, otherwise it instantiates an algorithm object and determines the computer's move using this object. Afterwards, it applies the operator representing this move in both cases and updates the current state.

We can customize the games and the algorithms for choosing the next move at the time of the game's instantiation. We can choose whether a human player plays against another human player or a computer player. In the latter case, we can also tell the algorithm which player will make the first move. We can set either minimax or negamax algorithm as the algorithm for choosing the next move and we can decide whether or not these algorithms should use alpha-beta pruning. We may ask for a suggestion for the human player. Similarly to the case of the computer's move, this suggestion will be computed by instantiating an algorithm object. The human player may either accept or reject this suggestion. Additionally,



```
package jatek;

import allapotter.*;

public class NegamaxAlgoritmus extends LepesAjanlo {
  public NegamaxAlgoritmus( Allapot allapot, int melyseg ) {
    this.allapot = allapot;
    this.melyseg = melyseg;
    allasokSzama = 1;
    if ( allapot.vegAllapot() || melyseg == 0 )
      hasznossag = allapot.negamaxHasznossag();
    else {
      hasznossag = MINUSZ_VEGTELEN;
      for ( Operator op : Allapot.getOperatorok() )
        if ( allapot.elofeltetel( op ) ) {
          Allapot uj = allapot.alkalmaz( op );
          NegamaxAlgoritmus negamax = new NegamaxAlgoritmus(uj, melyseg-1);
          if ( -negamax.hasznossag > hasznossag ) {
            hasznossag = -negamax.hasznossag;
            operator = op;
          }
          allasokSzama += negamax.allasokSzama;
        }
    }
  }
}
```

Figure 2: Java source of negamax algorithm.

we can give different depths of search for computing the computer's move and the human's suggestion.

The strength of the suggestion is affected by two factors. One of these factors is the depth of the search, and the other is the efficiency of the utility function applied to each state. Depending on the branching factor of the game tree, the time required to compute the next move may increase exponentially by increasing the depth of the search. That's why it is worth to write as efficient utility function as we can in the class of the actual game. If we can write a utility function that can determine for every state of the game how much that state is good for each of the players, then there is no point in setting the depth of the search to a value greater than one. However, the coin has two sides. It may take a very long time to evaluate a state, so what we really have to minimize is the product of the number of evaluated states and the time spent for evaluating each state.

3. Implementation of a Particular Game

In this section we present a very simple version of the Nim game as an example. The game is played by two players, A and B. They take turns adding 1, 2, or 3 to an integer which has an initial value of 0 at the beginning of the game. The player who reaches 21 wins.

When implementing this game, we should first create the operator which has

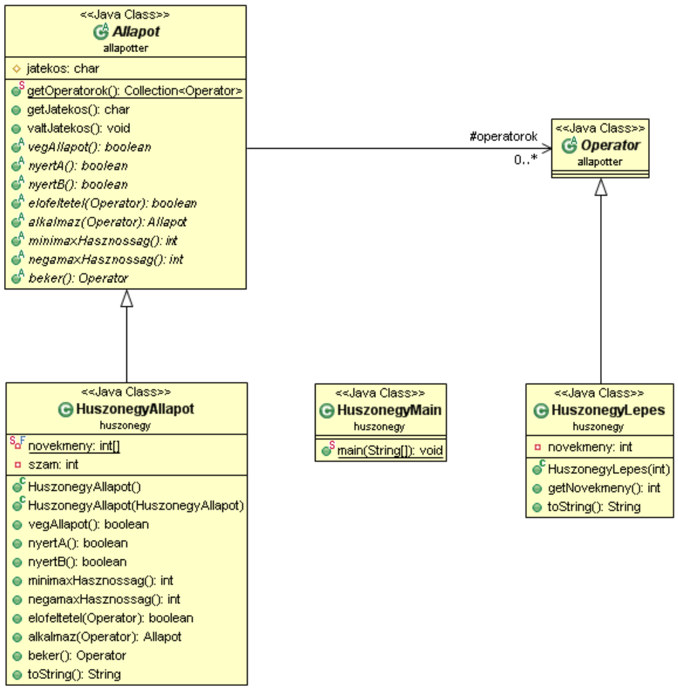


Figure 3: Class diagram of a very simple Nim game.

only one argument, the increment. The superclass of the class representing the operator will be the abstract `Operator` class.

The class representing the states of the game will be derived from the abstract `Allapot` class. We declare an *integer* as the only attribute of the game. Creating the methods which check the final state, the winner, the operator application preconditions and generate the new state produced by the operator is self-evident. The heuristic information used for evaluating the states may be implemented in only one of the methods `minimaxHasznossag` and `negamaxHasznossag`, as the other method can be easily implemented based on that.

The game itself will be instantiated in a different class which contains only the main program. We give the settings of the game along with the instantiation. After this, the game can be started and played by calling the `jatszik` method.

References

[1] Stuart Russel, Peter Norvig, *Mesterséges intelligencia modern megközelítésben*, Panem, 2005.

[2] Fekete István, Gregorics Tibor, Nagy Sára, *Bevezetés a mesterséges intelligenciába*, LSI Oktatóközpont, 1990.

[3] *Mesterséges intelligencia* (szerk.: Futó Iván), Aula Kiadó, 1999.

Márk Kósa
János Pánovics
University of Debrecen
Faculty of Informatics
H–4032
Egyetem tér 1.