CS 2110 Fall 2022
Assignment 5
Game Search and Hash Tables

Table of Contents

Updates

The following changes have been made to these instructions since their initial release.

- Oct 23: Typo fixed.
- Oct. 21: Clarification added about (non-)use of `java.util`.
- Oct. 21: Link to Javadoc web page added.
- Oct. 21: One new TODO added for TranspositionTable, which would have been necessary to do anyway but is now more clearly specified.
- Oct. 20: More details added about the `Maybe` type.

Learning Objectives

- Implementing a hash table
- Practice implementing `hashCode()` and `equals()` methods
- Experience with game search and recursion
- Experience with null-free programming
- Experience reading and modifying a larger program
- Experience with nested classes and interfaces
- Designing test cases with good coverage

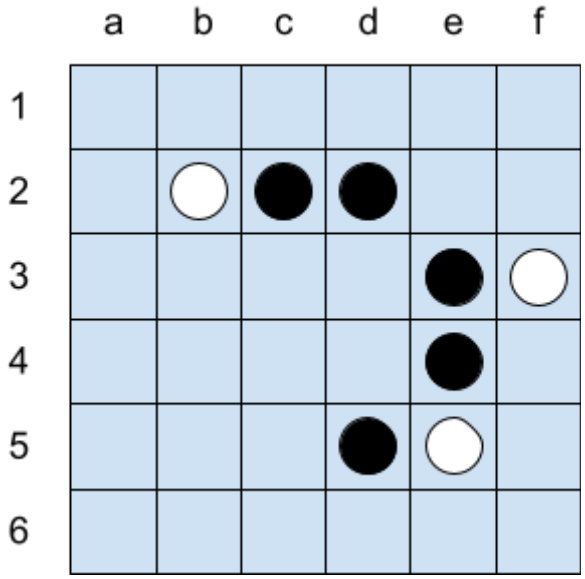
Introduction

In this assignment, we have given you an implementation of a simple two-player game, including a generic game search algorithm used by a computer player. Your job is to extend the program so it handles a more complex game while also implementing hash tables to improve the core search algorithm.

The game: Pente

The base game you receive as part of the code release is an implementation of Tic-Tac-Toe. Your job is to extend it into an implementation of the game Pente. Fortunately, Pente is similar to Tic-Tac-Toe, so most of the work has already been done for you.Tic-Tac-Toe and Pente are both (m, n, k)-games, which are board games in which two players take turns in placing a stone on an m-by-n board, and the winner is the first player who places k stones in a row, horizontally, vertically or diagonally.

Pente is played on a larger board than Tic-Tac-Toe, and the game can be won either by playing *five* pieces in a row, or by capturing ten of the opponent's pieces. You capture pieces by placing a piece so that it brackets a pair of the opponent's pieces between two of yours, either horizontally, vertically, or diagonally. Each captured piece is removed from the board, leaving its former position empty. That position can be played on again freely by either player. It is possible to capture more than one pair on a single turn by bracketing multiple pairs. For example, in the following board position, the white player can capture one pair by playing to position c6, or two pairs by playing to e2.



Pente can be played on any board that is at least 5 on a side. For this assignment you will implement it on an 8×8 board. Traditionally, Pente stones are placed on grid intersections rather than in the centers of the squares, but this is merely a cosmetic choice.

Minimax game search

Computer algorithms can often play games at or beyond a human level of capability by searching the tree of possible game states. In this tree, each node represents a possible state of the game (e.g., positions and colors of pieces), and the parent→child edges correspond to legal moves in the game: how a move transitions the game from one state to another. The leaves of the tree are ending states for the game, where one or other player wins (or, perhaps they are tied). Given the current state of the game, on their move a perfect player would choose one of the tree edges (moves) that makes it impossible for their opponent to steer the game toward a tree node that is a win for them.

For a sufficiently small game tree, like that of tic-tac-toe, it is possible to build the entire tree, and therefore to play perfectly. For more complex games, the game tree is far too large to search fully. Instead, a *heuristic search* algorithm is used, in which the game tree is searched incompletely (often to some fixed depth). The search is based on a *heuristic evaluation function*, which assigns a value to each game state. For example, game states with a positive value might appear to favor the current player, whereas states with a negative value appear to favor the opponent. How the heuristic function is defined depends on the game; modern heuristic functions are often discovered using machine learning, following the remarkable success of AlphaGo.

You are not required to understand how the search algorithm works in order to complete this assignment, because you have been given an implementation of a classic heuristic search algorithm, minimax search with alpha–beta pruning. This search algorithm, with some modifications, is used by the best modern AI game players, including AlphaGo. With an implementation of the search algorithm, a suitable heuristic evaluation function, and an implementation of the game rules that you will need to provide, the AI should be ready to play. A more detailed explanation of the search algorithm is given at the end for those curious.

Transposition tables

Part of your job is to improve the AI player by implementing a *transposition table*. Searching is very expensive, and the same game state may appear multiple times in the tree (since it could be reached by different sequences of moves, all starting from the initial state). A transposition table is a mechanism to avoid searching from the same game state more than once. Once searching has been done from a game state, the result is recorded in the transposition table so it can be looked up rapidly the next time it is encountered rather than repeating the search. Recording previously computed results is a powerful idea known more generally as *memoization*. The fastest implementation of a transposition table is as a hash table, so we will expect you to implement it

as a hash-table data structure, using chaining to resolve collisions and automatically resizing the data structure as new elements are added.

For the transposition table to be able to store game states, the class representing a game state must implement the `hashCode()` and `equals()` methods correctly and consistently. A bad implementation of `hashCode()` or `equals()` will lead to bad performance or even failure to work properly. Part of your job is to implement these methods for game states so the transposition table can work.

An implementation of a clustering estimator has been given to you for the transposition table. You should use this estimator to evaluate the quality of your hashcodes. If you run the game with the statistics reporting turned on, statistics on clustering will automatically be printed out. The clustering value should be near 1.0 if your hashcodes are generated in a way that avoids collisions. Larger values mean the hash table is less efficient.

Getting Started

Start by extracting the contents of `a5.zip` onto your computer, then open the contained `a5` folder in IntelliJ IDEA (you may see multiple nested `a5` folders after extracting; open the one that contains the `src` and `tests` subdirectories). Then open `src/a5/Main` from the project browser. You will probably see a banner in your editor saying "Project JDK is not defined"; click the "Setup SDK" link in the corner, then select your version 17 JDK. Now you're ready to code.

You also need to set up JUnit 5 to be able to run unit tests. Open `test/a5/ai/TranspositionTableTest`, and hover over the word "junit" appearing in imports (for example, in the line "`import org.junit.jupiter.api.Test;`"); click "More actions…" and choose "Add 'JUnit 5.8.1' to class path"; click "OK". Now you are ready to run unit tests.

Running the program. You should already be able to run the provided code and play Tic-Tac-Toe either between two human players, a human and a computer, or two computer players. Create a new Application run configuration with a main class of `a5.Main`. Remember to turn on assertions by going to Modify Options→Add VM Options and entering `-ea` in the "VM options" box.

The main program expects three main program arguments:

<game> <player1> <player2>

The *game* argument is either "tic-tac-toe" or "pente" (you are welcome to implement additional games as well!). The *player1* and *player2* arguments are either "human", "ai" or "ai2", where player *"ai2" uses the transposition table (hash table) you implement*. For example, running the game with arguments "tic-tac-toe human ai" will start a game where player 1 is human and

player 2 is a computer player; running it as “pente ai ai” will start up two computer players against each other.

The game board in the user interface requires your font to be monospace to display correctly. If you encounter related issues, go to office hours for help.

Below is a configuration example that runs a Tic-Tac-Toe game between two human players. Note that all of these arguments are case-insensitive.

The main program also accepts some optional program arguments to adjust its behavior. These optional arguments, if present, come *before* the main program arguments described above:

`-- help`	Print a usage message and quit
`-- showinfo`	Print out verbose statistics while AI players run
`-- timelimit <ms>`	Specify the maximum time for an AI player to make its move (overriding the default, 3000 ms = 3 s)

Reading the code. We recommend you to go through the codebase to get a sense of the whole structure, preferably together with your partner so you can discuss. The code is documented, and we also provide an overview of the code below.

Codebase Overview

Package `a5` contains one Main class and four sub-packages. Most of the classes you should not modify, but the classes colored green contain TODO items for you to complete. There is a Javadoc web page for the whole package, and the description below contains more information.

```
a5
├── Main.java
├── ai
│   ├── GameModel.java
│   ├── Minimax.java
│   ├── PenteModel.java
│   ├── TicTacToeModel.java
│   └── TranspositionTable.java
├── logic
│   ├── Board.java
│   ├── MNKGame.java
│   ├── Pente.java
│   ├── Position.java
│   └── TicTacToe.java
├── ui
│   ├── AIPlayer.java
│   ├── HumanPlayer.java
│   └── Player.java
└── util
    ├── GameResult.java
    ├── GameType.java
    ├── PlayerRole.java
    └── PlayerType.java
```

Here is an overview of all classes in this package:

- **`a5`**: the package of this assignment.
    - **`Main.java`**: the main program for `a5`. It contains code that parses user arguments and runs a new game.
    - **`logic`**: This package manages game states and maintains game logic for running (m, n, k)-games.
        - **`MNKGame.java`**: An abstract class that defines a running (m, n, k)-game. It maintains the current game board, player, and turn number. It also defines methods to let user query states and make moves. It provides an implementation of `hasEnded()` that checks if there are k stones in a row, horizontally, vertically, or diagonally, or if the board is full, which leads to a draw
        - **`Board.java`**: A mutable representation of an m-by-n board, in which each cell can be occupied by a player stone or be empty. States are represented compactly as a one-dimensional byte array to improve efficiency. You are asked to implement `equals()` and `hashCode()` so it can be hashed and stored in a transposition table.
        - **`Position.java`**: represents a position on the board.

        - **`TicTacToe.java`**: An implementation of Tic-Tac-Toe. It overrides `makeMove()` to implement the logic for a move in Tic-Tac-Toe.
        - **`Pente.java`**: An incomplete implementation of Pente. It maintains game states for an *8*-by-*8* version of Pente. You are asked to complete this implementation so that the game implements the rules of Pente, in which a player, in addition to what is allowed in standard (*m*, *n*, *k*)-games, can capture stone pairs and win by doing so five times in total. You will also need to implement `equals()` and `hashCode()` for these game states so that the transposition table can keep track of them correctly.
    - **`ui`**: this package defines players that interact with either a human player or an AI algorithm to generate the next move.
        - **`Player.java`**: An interface that defines an (*m*, *n*, *k*)-game player. It only contains one method, `nextMove()`, which is supposed to return the next move of the player.
        - **`HumanPlayer.java`**: An implementation of a Player which gets its moves from a human user. It prints out messages to ask for the user's next move, parses the input (using regular-expression searching), and checks the move's validity. It keeps asking until the user enters a valid move.
        - **`AIPlayer.java`**: An AI player that uses game search to generate moves. It is implemented generically so that any two-player game can be supported. Its constructor allows the user to specify game models and options on whether to enable features like alpha-beta pruning and a transposition table.
    - **`util`**: this package contains enums and classes that define constants and utility methods for other packages to use for convenience.
        - **`GameResult.java`**: An enum that defines the result of a game.
        - **`GameType.java`**: An enum that defines the type of game. It contains two types: Tic-Tac-Toe and Pente. It also contains some helper methods, including `fromString()`, which converts a string to a `GameType`, and `options()`, which returns a string listing all valid game types.
        - **`PlayerRole.java`**: A class that represents player roles. It defines two static `PlayerRole` objects, `FIRST_PLAYER` and `SECOND_PLAYER`, which represent the two player roles in an (*m*, *n*, *k*)-game. It also provides helper methods, including `nextPlayer()`, which returns the other player, and `boardValue()`, which returns the byte value representing this player's stones on the board.
        - **`PlayerType.java`**: An enum that represents the type of player. It contains three types: `HUMAN` for human players, `AI` for AI players without a transposition table, and `AI2` for AI players that enable the transposition table. It contains helper methods similar those in `GameType`.

- **`ai`**: This package is the most complicated. It contains the components of an AI algorithm based on minimax search.
  - **`GameModel.java`**: An interface that defines a game. It is used by the minimax search algorithm to: generate legal moves, determine if a game has ended, apply moves and generate game states, and evaluate a game state heuristically.
  - **`TicTacToeModel.java`**: An implementation of a game model for TicTacToe. It evaluates a game state by counting how many consecutive pairs of stones are not fully blocked: #(consecutive unblocked pairs of the current player) - #(consecutive unblocked pairs of the opponent).
  - **`PenteModel.java`**: An implementation of a game model for Pente. It evaluates a game state by counting a weighted sum of the number of consecutive unblocked k stones where $1 <= k <= 4$, and the number of captured pairs.
  - **`Minimax.java`**: A generic implementation of minimax search which can be applied to any two-player game. This implementation has the option to use alpha-beta pruning to decrease the number of nodes the algorithm needs to evaluate in its search. You don't need to understand the details of how this algorithm works, but you can see that when enabled, it uses a transposition table to memoize the minimax evaluation of encountered states.
  - **`TranspositionTable.java`**: An implementation of the transposition table. It is implemented as a hash table that maps from a game state to the search depth and minimax evaluation of that state to the state's heuristic value and the recorded depth. You must implement a chaining-based hash table that supports adding/updating and querying values. Unlike a standard hashmap interface, the depth associated with a memoized value matters. The search should only use values computed using at least the necessary depth.

Also in the code release is the package `cms.util.maybe`, which implements the `Maybe` type described earlier. You should use but not modify it.

Try to understand the purpose of each component, and clear up any confusion by asking questions in office hours or on Ed Discussions. We next discuss two programming patterns that you will see in the codebase.

Programming without null

The special value null is a major source of bugs in Java programs. It is tempting to use it to indicate the absence of a value, but it is too easy to forget to check for the possibility that a value is null—which leads to bugs that are difficult to find. Using null is sometimes a good choice when performance is critical, but usually it is better to avoid it. Fortunately, there are good alternatives. For example, the Java class `Optional<T>` allows representing a value that is either `T` or is absent. In this assignment we will use a similar type, `Maybe<T>`, included in the code release, which has the advantage over `Optional<T>` that it produces a safer, checked exception if used as a `T`. Some of the important methods and idioms for using `Maybe` are the following:

- `Maybe.none()` : produces a `Maybe<T>` that does not contain a value.
- `Maybe.some(v)` : produces a `Maybe<T>` that contains the value `v` of type `T`.
- `m.get()` : if `m` is a `Maybe<T>`, this method returns the contained value of type `T`, or throws a *checked* exception `NoMaybeValue` if there is no contained value. This exception is designed to be fast to handle.
- `m.isPresent()` : returns whether there is a contained value.
- `for (T v : m) {...}` : this loop executes 0 or 1 times, depending on whether the `Maybe m` holds a value of type `T`. If the loop body does execute (once), variable `v` is bound to the value contained in the `Maybe`.

Several more useful methods are provided by `Maybe<T>`, which mostly correspond to methods provided by `Optional<T>`. Many are convenient to use only with lambda expressions, however, which we are not expecting you to use yet (though you may). For example, a statement of the form

```
m.thenDo(v -> {...})
```

executes the statements represented by the "…" if `m` is present, and with the variable `v` bound to the value in the `Maybe`. Similarly, the following two expressions allow conditionally evaluating different expressions based on whether m is present:

```
m.then(v -> {...}) .orElse(...)
m.then(v-> {...}).orElseGet(() -> {...})
```

The `orElse` version always evaluates the "else" expression even if the `Maybe` is present, so use `orElseGet` if that result is not a simple value.

Model objects for generics

The code for the search algorithm is implemented in a very generic way. It is parameterized over two types: a type representing a state of the game (type parameter `GameState`) and a type representing a move in the game (type parameter `Move`). To implement search over the two supported games, the generic search algorithm is then instantiated differently. For Tic-Tac-Toe, it is instantiated with `GameState` and `Move` equal to the types `TicTacToe` and `Position`; for Pente, it is instantiated on the types `Pente` and `Position`.
One challenge in implementing generic code is accessing operations of the actual types being used. The search algorithm is implemented using a pattern called the Concept pattern. In this pattern, a separate interface is used to describe the operations of the parameter types, and a separate *model object* provides those operations. We've seen this pattern already in the `Comparator` interface used for sorting and tree data structures. In the code we have given you, the `GameModel` interface similarly describes the operations that the search algorithm needs to use on `GameState` and `Move` values. The classes `TicTacToeModel` and `PenteModel` then define the operations that the algorithm actually uses, including the heuristic evaluation function for the two games. You should not need to modify these classes, although you may find it interesting to try changing the evaluation to improve the AI player. We will not be asking you to submit this code, however.

Tasks

You have two major tasks, which can be conducted largely in parallel. However, we strongly encourage your whole group to be involved in both tasks.

Task 1: TranspositionTable

This task requires you to implement a hash table that is used by the minimax search algorithm.

1. Complete `a5.ai.TranspositionTable` (4 TODOs).
   Goal: Implement a chaining-based hash table that maps a game state to a search depth and a heuristic evaluation of that state using the recorded depth.
   Some components, such as the type of stored values, are provided, so read the document and code carefully and think of how you would incorporate them when implementing the hash table.
   You may not use classes from the package `java.util` to implement this class.
   - TODO 1: Implement the `classInv()` method for your transposition table. You may also add more conditions to the class invariant, but you should document them if so.
   - TODO 2: Complete the constructor of an empty transposition table.
   - TODO 3: Implement looking up the value given a game state. It should return `Maybe.none()` when no appropriate value is found corresponding to the given state.

    - TODO 4: Implement adding/updating an entry in the table.
    - TODO 5: implement a method to grow the hash table to contain at least a certain number of buckets.
2. To test the transposition table implementation, we need to first complete the methods `equals()` and `hashCode()` for `a5.logic.Board`. Make sure that `equals()` returns true when comparing two boards that share states that are equivalent for the purpose of game search. Test your implementation by completing TODOs in `a5.logic.BoardTest`.
3. Test your implementation by writing unit tests in `a5.ai.TranspositionTableTest`. At this point, you might not have implemented Pente, so you can use Tic-Tac-Toe state as the game state. An example is provided. Notice how to use `assertThrows` to expect one exception.
4. Now you can enable the transposition table in our AI for Tic-Tac-Toe.
   For example, run `a5.Main` with arguments "Tic-Tac-Toe ai ai2" to run a game with two AI players where the second AI player enables the transposition table. For a small-scale game like Tic-Tac-Toe, this optimization doesn't make a difference (it's easily a draw), but you should see unusual behavior if your implementation is very buggy.
5. After you have finished implementing Task 2, write more unit tests in `a5.ai.TranspositionTableTest`, using Pente game state.
6. Run Pente games with an AI player that enables the transposition table.
   Enabling the transposition table allows the algorithm to search more useful game states by avoiding re-exploring equivalent states. We provide you a way to monitor the information of the search when executing the program: specify the optional argument `--showinfo` to the program. For example, arguments `--showinfo Pente ai ai2` will run a Pente game with two AI players where the second AI player enables the transposition table while printing out search information for both players.

Task 2: Pente

You need to implement the game rules of Pente. To make your implementation job easier, an abstract class `MNKGame` already provides functionality shared between Tic-Tac-Toe and Pente. The easiest and best way to implement Pente is to inherit from `MNKGame` and override its behavior.

1. Complete `a5.logic.Pente` (7 TODOs).
   Pente is an (m, n, k)-game with additional rules specified above. Your job is to make sure your implementation enforces these rules in addition to the rules of a standard (m, n, k)-game. Think of what additional information you need to keep track of and to represent it in code.
    - TODO 1: Complete the constructor. Initialize the field members you defined.

- TODO 2: Complete the copy constructor. This constructor is used by the search algorithm to create copies of the game state that can be used for searching without affecting the real state of the game.
- TODO 3: Complete method `makeMove()`. In a standard (*m*, *n*, *k*)-game, a move is simply placing a stone on the board (see `TicTacToe`'s implementation, for example). In Pente, a move can also removing other stones from the board. So make sure the board after the move looks like what's expected.
- TODO 4: Complete `capturedPairsNo()`. It returns the number of pairs captured by the given player.
- TODO 5: Complete `hasEnded()`. In addition to the rules of a standard (*m*, *n*, *k*) game, a Pente game also ends when one player captures five pairs of stones.
- TODO 6: Complete `stateEqual()`. This method implements state equality for the class `Pente`. However, it should only compare the parts of the game state that matter for searching.
- TODO 7: Complete `hashCode()`.

2. Unit-test your implementation by putting test cases in `a5.logic.PenteTest`.
3. Run Pente games and see if the games go as expected. If you have finished an implementation of the transposition table, you can also check that the transposition table is working well by looking at the statistics reported when you use the `--showinfo` option.

   The clustering estimate for the transposition table should be low: around 1. And if the search is able to reach a depth of 3 or 4, you should expect to see a "hit rate" for the transposition table around 20% or higher. The hit rate is the fraction of searches that can be skipped because the transposition table already records their outcome. If you are not achieving that search depth, make sure expensive asserts are turned off or commented out.

Task 3: Summary document

You must write a short document briefly describing the following aspects of your work:

- Implementation strategy: describe how you went about implementing the assignment and how the work was divided between the partners.
- Testing strategy: how did you perform testing and design test cases?
- Known problems: are there any things that do not work?
- Time spent on the assignment, in hours.
- Comments on the assignment.

In the source release you will find a template file named `summary.txt`, which you can edit and put these answers into.

Getting help and feedback

Smoke testing: Smoke testing will be available. Look for the appearance of the assignment **A5 Smoke Test**, which is where submissions should be made to run the smoke test on your code.

Questions: If you do not know where to start, are stuck, do not understand testing, or are lost, please use Ed Discussions to ask questions. There will be a pinned A5 FAQ post that you should keep an eye on even when you don't have questions, because it may answer questions even before you have them. Don't be afraid to see someone on course staff. This can be in the form of an instructor, TA, or consultant.

Collaboration Policy

You may do this assignment alone or with *one* other person. If you do this assignment with another person, you must *work together* with both partners contributing. If you have worked with a partner on a past assignment, we encourage you to try working with someone new, though this is not required. Form a group on CMSX *as soon as you decide to work together* to formalize that agreement (if any files are submitted before confirming your group, then the group cannot be formed). *Both* partners must do something to form the group: one invites, the other accepts.

Pair Programming. As in Assignment 4, we recommend trying *pair programming*, in which the partners sit together, with one partner acting as the *pilot* driving the keyboard and mouse—and the other as the *navigator,* guiding and critiquing. Here is an effective workflow: first, the pilot proposes a method signature and specification. The navigator then critiques: is the spec clear? They iterate until both agree on the spec. Only when they agree does the pilot write the code. It is then the job of the pilot to convince the skeptical navigator that the code meets the spec. You should switch off the roles of pilot and navigator.

Academic Integrity. With the exception of your CMSX-registered partner, you may not look at anyone else's code, in any form, or show your code to anyone else, in any form. You may not look at solutions to similar previous assignments in 2110. You may not show or give your code to another person in the class. You can talk to others, but your discussions should not include writing code and copying it down.

Submission

Hopefully you now have a nice implementation of Pente that you can have fun playing!

Make sure to update the file summary.txt with information like the time you spent and your group's identification. Then upload the following **seven** files: `TranspositionTable.java`, `TranspositionTableTest.java, Board.java`, `BoardTest.java, Pente.java`, `PenteTest.java`, and `summary.txt` to **Assignment 5** on CMSX before the deadline. Before you submit, make sure you have made a group on CMSX with your partner!

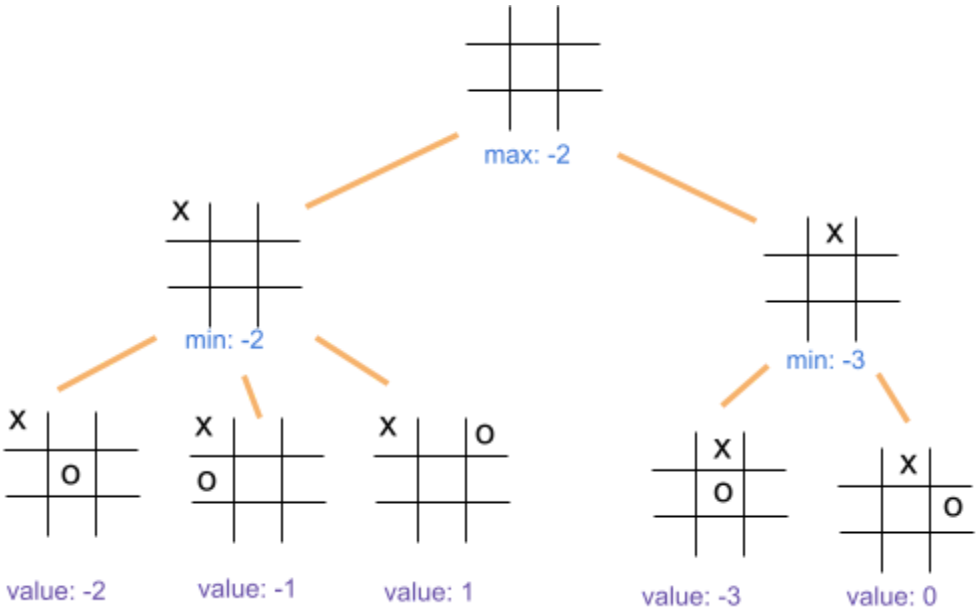
Good luck, and have fun!

Appendix

Minimax Game Search

While you don't need to understand it, here is a brief explanation of the minimax game search algorithm. In game search, the game tree is searched only partially, perhaps to some maximum depth. At the deepest reached nodes (call them *search leaves*), the *heuristic evaluation function* is used to assign a value to the game state, capturing how favorable that game state appears to the current player. The heuristic evaluation function takes on some maximum value *WIN* when the current player is definitely going to win the game, and a minimum value *LOSE* when the other player is definitely going to win. Often it is defined in such a way that its value from one player's perspective is the negative of its value from the other player's, so *LOSE* = –*WIN*. The algorithm we have given you works this way. If the game search is able to search the game all the way to the leaves of the full game true, then the value of the game state is known completely: either *WIN*, -*WIN*, or 0 (in case of a tie). More typically, it is not possible to search all the way to an final game state, in which case the heuristic evaluation function assigns some best guess at who is ahead, as a value between -*WIN* and *WIN*.

For example, the diagram below shows a small part of the game tree of the Tic-Tac-Toe game. The root has an empty board, the next layer has various moves by the "x" player, that layer's node's children show responses by "o", and so on. Suppose that the game search reaches only the nodes shown. At the leaves of the search, the heuristic evaluation function assigns game states a value from the perspective of the player "x", shown in purple.



The value of a node that is not a search leaf depends on whose move it is. If it is the current player, the value is the maximum of the value over all child nodes, since the current player gets to choose their move; but if it is the opponent's move, the node value is the *minimum* value (from the perspective of the current player) over all child nodes. Therefore the value of a node is found by alternately taking maxima and minima of values over the searched part of the tree.

For example, in the diagram above, the leaf states correspond to choices by the "o" player. Therefore, their parent nodes have a value that is the minimum of all the values of the children. And player "x" has the choice at the root node, so the value of that node should be the *maximum* of the values of its children. Obviously player "x" should steer play toward the left child from the root since it leads to better outcomes.

A two-player game search algorithm recursively computes this *minimax* tree value for each searched node, producing an estimate of the best possible move from the current state; the algorithm simply chooses the move with the largest minimax value.

Alpha–beta pruning can effectively double the search depth, making AI players much more powerful. It works by modifying minimax search so that the search algorithm is only interested in finding values between specified minimum and maximum values. At the top level of the tree, the minimum and maximum values are initially -*WIN* and *WIN* respectively. If at some node, a move is found that is at least as good as the maximum value of interest, no more moves need to be considered at that node in the tree, pruning away part of the search. As better moves are found from a particular node, the best value seen so far, improving on the minimum value, is passed to the recursive search from the opponent's perspective as increasingly tighter bounds on the maximum value for that recursive search. For example, in the game tree above, once the value of the left child of the root is known to be -2, there is no point to searching parts of the right subtree corresponding to values smaller than -2. So the call to search under the right child passes a tighter maximum value of -(-2) = 2, potentially pruning away subtrees. In practice, alpha–beta pruning roughly doubles search depth while computing the same game value as a full, unpruned minimax traversal.