CS 540: Introduction to Artificial Intelligence
Homework Assignment #4: Game Playing

Assigned: Nov. 30, 2011
Due: Dec. 14, 2011

Hand-in Instructions

This homework assignment includes written problems and programming in Java. Hand in hardcopy of the requested written parts of the assignment. All pages should be stapled together, and should include a cover sheet on top of which includes your name, login, class section, HW #, date, and, if late, how many days late it is. Electronically hand in files containing the Java code that you wrote for the assignment (see course Web page for instructions. ASSIGNMENT_NAME is HW4).

Late Policy

All assignments are due **at the beginning of class** on the due date. One (1) day late, defined as a 24-hour period from the deadline (weekday or weekend), will result in 10% of the total points for the assignment deducted. So, for example, if an assignment is due on a Wednesday and it is handed in between Wednesday 11 a.m. and Thursday 11 a.m., a 10% penalty will be deducted. Two (2) days late, 25% off; three (3) days late, 50% off. No homework can be turned in more than three (3) days late. Written questions and program submission have the same deadline. A total of two (2) free late days may be used throughout the semester without penalty.

Assignment grading questions must be raised with the instructor within one week after the assignment is returned.

Collaboration Policy

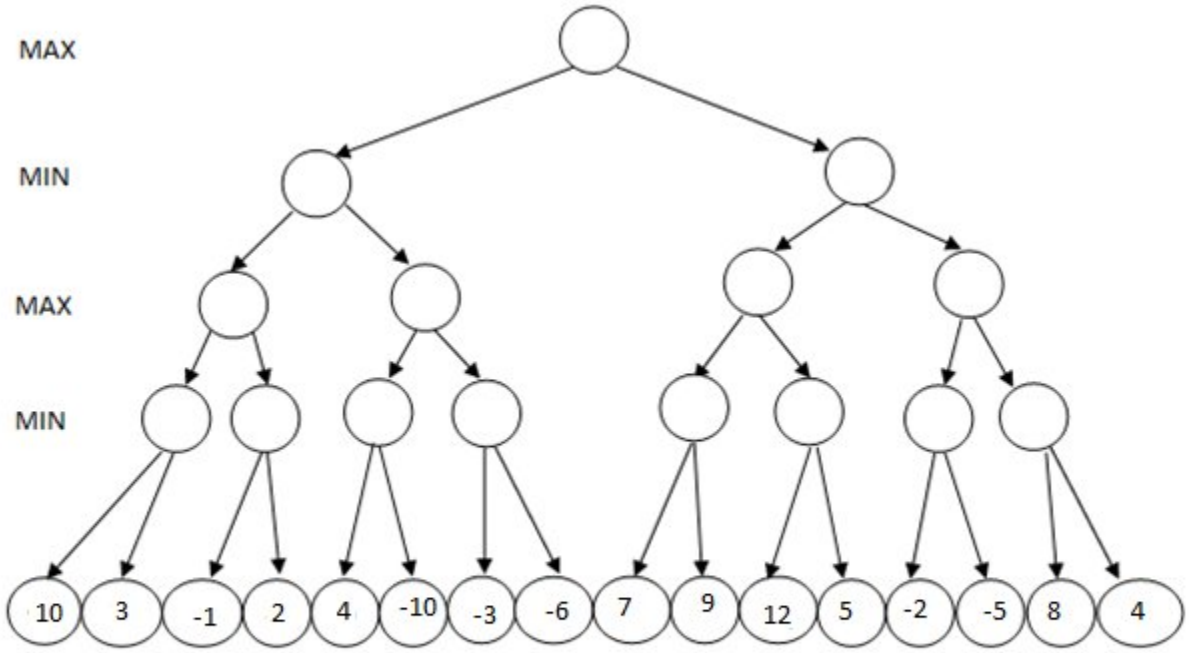
You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TAs, and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate how to solve problems. But we require you to:

- not explicitly tell each other the answers
- not to copy answers or code fragments from anyone or anywhere
- not to allow your answers to be copied
- not to get any code or help on the Web

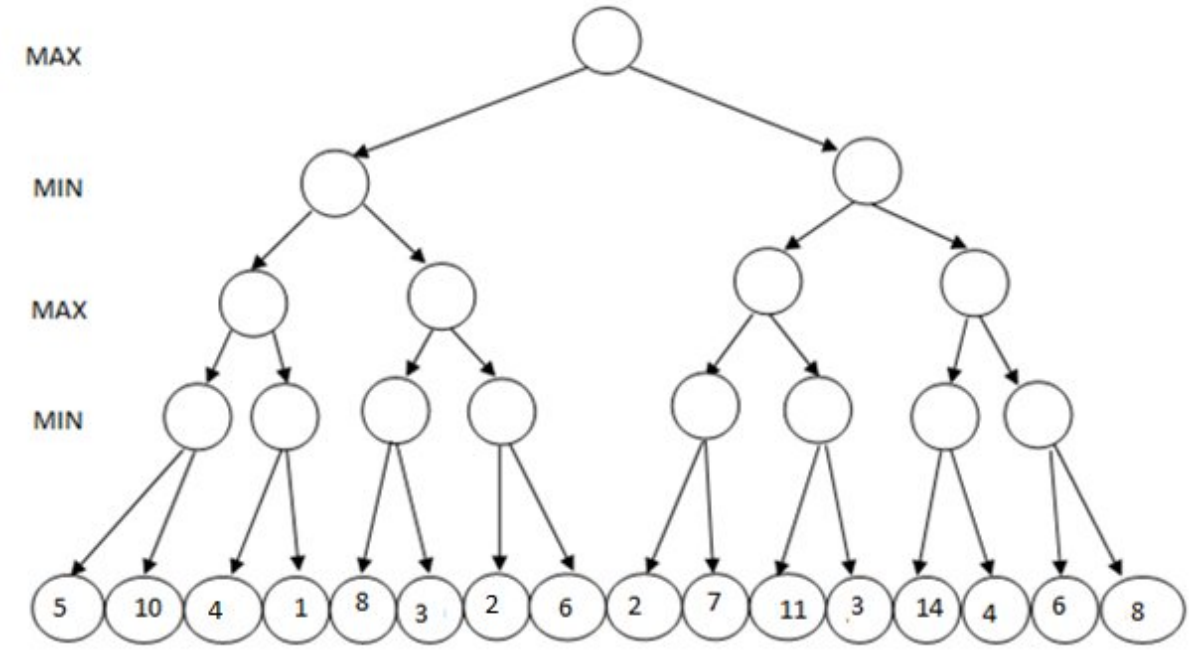
In those cases where you work with one or more other people on the general discussion of the assignment and surrounding topics, we suggest that you specifically record on the assignment the names of the people you were in discussion with.

Question 1 [20]: Game Playing

(a) Use the Minimax algorithm to compute the minimax value at each node for the game tree below.



(b) Use Alpha-Beta Pruning to compute the minimax value at each node for the game tree below, assuming children are visited left to right. Show the alpha and beta values at each node. Show which branches are pruned.



Question 2 [20]: Hill-Climbing

We would like to solve the n-Queens problem using a greedy hill-climbing algorithm. The n-queens problem requires that we place all n queens on the board so that none of them can attack any other; so we cannot have 2 queens in the same row, column or diagonal. Each state corresponds to a complete assignment of a row number from 1 to n to each of the n column variables C1 – Cn. The successor operator *Succ*(*s*) generates all neighboring states of *s*, which we define as all total assignments which differ by exactly one variable's row value. So, for example, given n=2 the state with assignments {C1 ← 1, C2 ← 2} has two neighboring states {C1 ← 2, C2 ← 2} and {C1 ← 1, C2 ← 1}. When breaking ties, order the variables from left to right, and the values for the variables from bottom to top of the board and pick the neighboring state that comes first in the ordered list of states.

(a) Define an evaluation function for the states such that the goal state has the highest value when the evaluation function is applied to it.

(b) If you have *n* rows and n column variables, how many neighboring states does the *Succ*(*s*) function produce?

(c) What is the total size of the search space, i.e., how many possible states are there in total? Assume again that there are *n* rows and n columns.

(d) Consider the 6-Queens problem again and come up with a non-goal state (i.e., a non-satisfying assignment) that is on a *plateau* in our hill-climbing space (fill in 1 to 6

for each variable):

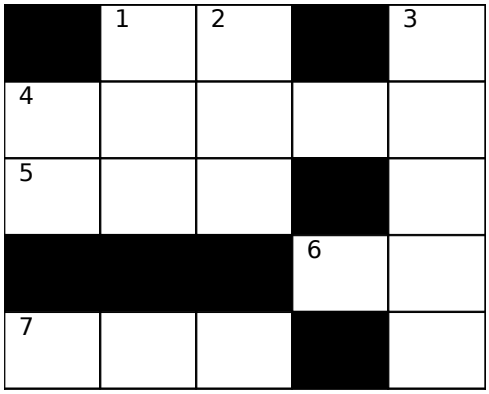
C1 =

C2 =

C3 =
C4 =
C5 =
C6 =

Question 3 [20]: Constraint Satisfaction Problems

This question will use the CSP algorithm on a crossword puzzle problem. Below is a list of words and the crossword puzzle matrix and your task is to find the position that each word has to occupy. The numbers correspond to the locations that the words should start. Say we formulate this problem so that we have a variable i-(ACROSS/DOWN) representing the position and the direction in which the word should be placed. The list of variables thus formed is also given below. Remember each word can be used only once in the puzzle.



LIST OF WORDS: AT, EAR, FED, ICE, IF, SCENE, SEATS, WED.

LIST OF VARIABLES: 1ACROSS, 1DOWN, 2DOWN, 3DOWN, 4ACROSS, 5ACROSS, 6ACROSS, 7ACROSS.

(a) Formulate the domain of values that each variable can take and the list of binary constraints for this problem. Hint: Consider each variable as an array of letters and denote positions in variables using the notation variableName[position].

(b) Show the result of the Forward Checking algorithm with Dynamic Variable Ordering using the most-constrained variable heuristic. When breaking ties, order the variables using the order given in the list above and then order the values in alphabetical order.

Question 4 [40]: Checkers

For this question you are to implement an “intelligent" computer player for the game of Checkers. If you do not know the rules of Checkers, you can find an explanation at http://www.jimloy.com/checkers/rules2.htm

You will create two Checkers players for this assignment. First, you will create a basic alpha-beta player, which uses alpha-beta pruning with iterative deepening. After creating the alpha-beta player, you will create a competitive player by building upon and increasing the competitiveness of your alpha-beta player. As part of creating the competitive player, you may implement a better search algorithm, devise a better static board evaluator, or use any technique of your choice. Your competitive player will be entered into a class tournament, from which the students with the best players will win prizes and extra credit points.

You are provided framework code that contains most of the code necessary, so the amount of code you'll have to write is not large. The framework code is available at the course webpage.

The Framework

Packages

The framework provided with this assignment is divided into several packages. Packages are a namespace mechanism in Java, which serves to increase organization and reduce name clobbering of classes. For more information, see http://java.sun.com/docs/books/tutorial/java/package/packages.html
The packages included with the framework are:

`cs540.checkers` : Provides the core functionality of the framework. All other packages revolve around `cs540.checkers`. This package is described in detail below.

`cs540.checkers.ui` : Implements a user interface to this framework. This user interface allows an interactive player to select moves through `cs540.checkers.ui.HumanPlayer`.

`cs540.checkers.demo` : Demonstrates the use of this framework with `cs540.checkers.demo.RandomPlayer` which chooses moves at random.

In addition, each student is assigned their own namespace in `cs540.checkers.<username>` where `<username>` is the CS login name of the student. Students are required to place any code they write or modify in that package so that student files are not clobbered. Throughout the rest of this document we will abbreviate classes without the `cs540.checkers` at the front. For example, `cs540.checkers.demo.RandomPlayer` will be referred to as `demo.RandomPlayer`

Data Structures

First, you should become familiar with how this framework represents the following characteristics of a Checkers game:

Side The sides Red and Black are represented as `int` values of `0` and `1` respectively. In addition, some classes use a side called `NEITHER` that represents an unknown, intermediate, or indeterminate side. This coding is defined by the fields `RED` and `BLK` and `NEITHER` in `CheckersConsts`

Table 1: Representation of the sides of a Checkers game.

Side	Field name	`Int` value
Red side	`RED`	0
Black side	`BLK`	1
Neither side	`NEITHER`	2

Checkers piece Checkers pieces are represented as `int` using the fields `RED_PAWN` and `BLK_PAWN` and `RED_KING` and `BLK_KING` in `CheckersConsts` (see Table 2). This representation also includes `BLANK` which is not a Checkers piece but represents an empty square.

Table 2: Representation of the Checkers pieces.

Checkers piece	Field name	`int` value
Red pawn	`RED_PAWN`	0
Black pawn	`BLK_PAWN`	1
Blank square	`BLANK`	2
Red king	`RED_KING`	4
Black king	`BLK_KING`	5

Note that the least two significant bits of a Checkers piece is the side the piece is owned by. For example, the owner of `BLK_KING` ($5 = 0101_b$) is `BLK` ($1 = 0001_b$). Likewise, the owner of `BLANK` ($2 = 0010_b$) is `NEITHER` ($2 = 0010_b$).

Location The two-dimensional Checkers board is indexed in row-major order. When viewed from `RED`, the squares of a Checkers board are indexed left-to-right before top-to-bottom, starting from 0 (see Figure 1).

Figure 1: The locations on a Checkers board.

BLK							
	01		03		05		07
08		10		12		14	
	17		19		21		23
24		26		28		30	
	33		35		37		39
40		42		44		46	
	49		51		53		55
56		58		60		62	
RED							

Board state A board state is represented by a one-dimensional `int[]` array whose indices are the 64 board locations. The element at index `i` is the `int` representing the Checkers piece at location `i`; the element is `BLANK` if location `i` is blank.

Move A move is represented by the sequence of locations through which a Checkers piece travels. An immutable move, which cannot be modified once constructed, is represented by the class `Move`. The class `MutableMove` represents a modifiable move. Both `Move` and `MutableMove` are subclasses of the generics collection `java.util.List<Integer>`

Contents of `cs540.checkers`

Each of the classes in the `cs540.checkers` package may be categorized into one of two groups. The first group deals with all the plumbing necessary to run a Checkers game, including clocks for time control, an event loop to progress the game, and facilities to control player threads. You do not need to understand or modify most of these classes. The classes in this first group are:

CheckersModel Maintains the state information of a Checkers game, including the board state, whose turn it is, and time accounting.

CheckersController Progresses a Checkers game by acting on an instance of `CheckersModel` This class controls computation threads and enforces time controls.

Checkers Parses command line options, initializes the Checkers game, and launches the user interface. This class contains `main()`

CheckersPlayer Selects moves to be played on a Checkers board. Both your alpha-beta player and your competitive player will extend this class. An example of how to extend this class is given by `demo.RandomPlayer`

CountdownClock Describes the methods associated with a countdown clock. This

interface is implemented by `DefaultCountdownClock`

GameClock Describes the methods of a game clock. This interface is implemented by `Default-GameClock`

The second group describes the structures described in the Data Structures section above, and contains utility methods that operate on those structures. The classes of this group are used throughout the rest of the framework, and you are expected to become familiar with what they do and how they function. They are:

CheckersConsts Contains the piece and side representations. You should statically import the static fields of this class by declaring
`import static cs540.checkers.CheckersConsts.*;`
at the beginning of your source files.

Move Represents an immutable move as a sequence of locations.

MutableMove Represents a modifiable move as a sequence of locations.

Utils Provides static methods that operate on the structures described in the Data Structures section above. You will use this class frequently, so you should carefully read through the documentation for this class.

BoardState Provides an object-oriented representation of a board state, which is otherwise stored as an `int[]` array. This class is provided for convenience, and you are not required to use this class. If you elect to use this class, read its documentation carefully.

Evaluator Describes the interface of a static board evaluator. This interface is implemented by `SimpleEvaluator` among others. You are not required to use this interface. Note that the member method `eval()` returns a score from RED's point of view; you must negate the score to obtain the score from BLK's point of view.

For an example of how these classes are used, see `demo.RandomPlayer` and `demo.DemoPlayer`

Directory Structure

The ant version of this framework uses the following directory structure:

`./` Base directory. The Ant configuration files `build.xml` and `build.properties` reside here.

`src/` Source directory. All source files of classes in the `cs540.checkers` namespace belong here. Source files of classes in sub-namespaces belong to subdirectories of `src` For example, classes of `cs540.checkers.xyz` reside in `src/xyz/`

`src/ui/` Source directory for the `cs540.checkers.ui` namespace.

`src/<username>/` User directory. This directory is reserved for any files you create, i.e., files in the `cs540.checkers.<username>` namespace. You may need to manually

create this directory.

`build/` Build directory. This directory contains compiled classes and other temporary files.

The eclipse version of this framework uses the following directory structure:

`./` Base directory. This contains the src/ and lib/ directories

`src/` Source directory. This has a subdirectory names cs540/

`src/cs540/checkers` All source files of classes in the `cs540.checkers` namespace belong here.

`src/cs540/checkers/ui/` Source directory for the `cs540.checkers.ui` namespace

`src/cs540/checkers/<username>/` User directory. This directory is reserved for any files you create, i.e., files in the `cs540.checkers.<username>` namespace. You may need to manually create this directory.

Toolchain

Two versions of this framework have been created, one for use with Eclipse and the other for use with Apache Ant. Use one of these or any other system you prefer for developing your code. Ant is a build system similar to `Make`. While `Make` uses a `Makefile` to configure its targets, Ant uses `build.xml`. For more information, see http://ant.apache.org/manual/index.html

To compile this project using Ant, type "`ant`" in the base directory. This invokes the default target `build` which compiles all source files in `src/` into class files in `build/` These compiled class files are then linked together into the JAR file `./Checkers.jar` You may run this file with the command `java -jar Checkers.jar` Additional Ant targets are also provided for your convenience. The `clean` target removes all compiled class files in `build/` The `javadoc` target generates a local copy of the documentation for all classes in this framework. This documentation will be placed in `doc/api/` To invoke these targets, type `ant <target>` Note that typing `ant` by itself invokes the default target `build`

In Eclipse any files organized in a package are found in a subdirectory of `src/` that corresponds with that package name, e.g., package `cs540.checkers` files are in `src/cs540/checkers/` A separate tar file is provided that is set up for use with Eclipse. Also, see the documentation on using Eclipse with this assignment.

Execution

After building the framework, run it by typing (from the base directory):

```
java -jar Checkers.jar <red_player> <blk_player> [options]
```

where `red_player` and `blk_player` are the abbreviated class names of the red and black players. The abbreviated class name is the fully-qualified name of the class minus the "`cs540.checkers.`" at the beginning and "`Player`" at the end. For example, the abbreviated class name of `cs540.checkers.demo.RandomPlayer` is `demo.Random` and setting `red_player := demo.Random` puts the random player on `RED`.

The following optional parameters are recognized by the Checkers program:

`--turntime <turnLimit>` Specifies how long, in milliseconds, players are allowed to think per turn. Not enforced against interactive players. (Default = 1000)

`--step` Require a mouse click before the start of each turn. By default, turns start automatically with a small delay.

`--verbose` Verbosely print output. Classes may access this option through the field `Utils.verbose`

`--initbs <fileName>` Set the board state specified in `fileName` as the initial board state. This parameter is useful for debugging. See `Utils.parseBoardState` for the format expected in `fileName`

`--initside <side>` Specify which `side` moves first. This option is used for debugging. (Default = `RED`)

For example, the command

```
java -jar Checkers.jar ui.Human <username>.AlphaBeta --turntime 3000
```

will run a game between an interactive player (RED) and your alpha-beta player (BLK). The alpha-beta player is given 3 seconds per move.

Requirements

Your task is to create two Checkers players that interface with the provided framework. First, you should create the alpha-beta player, which uses alpha-beta pruning with iterative deepening to search a game tree. This player serves to demonstrate your knowledge about basic methods in searching a game tree. Next, you will improve the competitiveness of the alpha-beta player using techniques of your choosing; this player is referred to as your competitive player. A class tournament will be run using the competitive player for each student, from which the top scorers will receive prizes and extra credit points.

Alpha-Beta Player

The first of the two players you will implement, the alpha-beta player, searches using alpha-beta pruning. You must use the `getAllPossibleMoves` method to retrieve the list of possible moves at each non-terminal node, and you must examine these moves in order. Use the `SimpleEvaluator` SBE to evaluate terminal nodes.

Implement this player using iterative deepening, which expands the depth of the game tree by one level at each iteration. This expansion will continue until a maximum depth of `depthLimit` or until the time is up, whichever comes first.

Additionally, you must count the number of subtrees pruned by the alpha-beta search. This prune count will be used to grade your alpha-beta player for correctness.

Implement this alpha-beta player as a subclass of `CheckersPlayer` in `calculateMove()`. Do not place the chosen move in the return value of `calculateMove()`; instead, set the chosen move via `setMove()`. The game controller

will access the chosen move by calling `getMove()`. Note that each step of iterative deepening may produce a different best move. Therefore, you should call `setMove()` after each iteration so that when the game controller interrupts the computation thread on which `calculateMove()` runs, `getMove()` returns the best move found so far.

Place this alpha-beta player in class `cs540.checkers.<username>.AlphaBetaPlayer` and name the source file `src/<username>/AlphaBetaPlayer.java`

Skeleton A skeleton of the alpha-beta player is provided in `src/demo/AlphaBetaPlayer.java` This skeleton is straightforward to complete: first, move the file into `src/<username>/` and change the `package` declaration at the top of the file to read:

```
package  cs540.checkers.<username>;
```

All that remains is to complete the `calculateMove()` method with code that performs alpha-beta pruning.

Grading The TA will grade the correctness of your alpha-beta player by comparing its number of pruned subtrees to that of a reference alpha-beta player. For this to work, your alpha-beta player needs to implement the `GradedCheckersPlayer` interface, which provides the `getPruneCount()` method. Implement this method to return the number of pruned subtrees during the most recent iteration of iterative-deepening. You may assume that `getPruneCount()` will not be called while your player is calculating.

Competitive Player

After creating your alpha-beta player, create a second player that you will optimize for performance. You may incorporate any techniques you wish for this competitive player, which will eventually be used for the class tournament.

Begin by extending your existing alpha-beta player. The alpha-beta player uses a very simple SBE function, which uses only the number of red and black pieces remaining on the board. You must write a better SBE that uses more information from the board state. Modify your competitive player to use this new SBE.

In addition to creating your own SBE, you may also improve the alpha-beta search procedure. See Chapter 5 in the textbook for some ideas on doing this. You are not required to modify the search procedure to get full credit for this player, but it should help you improve your performance in the class tournament.

Implement your competitive player in class `<Username>Player` of namespace `cs540.checkers.<username>` and name the source file `src/<username>/<Username>Player.java` Note that `<Username>` is `<username>` with the first character capitalized.

Grading Your competitive player will receive credit based the effort you put into it. You will receive full credit if you implement "several" features on top of the alpha-beta player. At least one of these features must be an improved SBE.

Additionally, you must write a report describing your competitive player. At a minimum, this report should contain:

- a description of your SBE function, explaining what it computes and why it is good
- descriptions of any other features your player uses
- comments on how your competitive player performs against the alpha-beta player or people, citing strengths and weaknesses that it seems to exhibit

The Tournament

The TAs will run a Checkers tournament with each student's competitive player, whose abbreviated class name will be assumed to be `<username>.<Username>` For example, `smith.Smith` if username is `smith` All games will be run by the TAs. Extra credit points will be given to the top students in the class tournament.

Miscellaneous

Namespaces Be sure that any files you write or modify are in the `cs540.checkers.<username>` namespace and the `src/<username>/` directory. All other files are automatically discarded before grading. A simple way to test whether this namespace requirement is satisfied is by copying the `src/<username>/` directory into a fresh source tarball and verifying that everything works correctly.

Verbosity Both your alpha-beta player and your competitive player should print debugging statements to `stdout` when the `--verbose` flag is on. You may access the status of this flag by referencing the field `Utils.verbose` Your players should, at a minimum, print the best move and its score at the end of each iterative deepening step. Conversely, your program should not print anything if `Utils.verbose` is `false`.

Hand-In Instructions

Code Hand in the contents of `src/<username>/` to the HW4 handin directory.

Report Submit the report for your competitive player in hardcopy form in class on the due date. If you have multiple pages, staple them together. Put your name, login, and the date on the top of the first page.