Homework Assignment #2

Assigned: Wednesday, February 15th
Due: Tuesday, February 28th

Hand-In Instructions

This assignment includes written problems and programming in Java. Hand in all parts electronically by uploading them in *a single zipped file* to the assignment page on Canvas.

If you choose to work with a partner on the programming portion, you still need to complete and submit the written portion individually. All students must submit the written PDFs + a README.txt (with a partner's name if applicable) to receive full credit. At least one member of each partnership (or team of one) must submit the zipped, completed code in addition to the PDFS and README.

Your answers to each written problem should be turned in as separate pdf files labeled as <wisc NetID>-HW2-P<problem number>.pdf. You can use your program of choice – pencil and paper, word processor, LaTeX – as long as you show your work fluidly and neatly. If you handwrite your solutions to written problems, make sure to scan them in. *No photographed assignments will be accepted.*

For the programming problem, put *all* files needed to run your program, including ones you wrote and ones provided, into a folder called <wisc NetID>-HW2-P4.

Once you are finished, put your programming component folder and your two PDF files into a single directory. Zip it, name it <wisc NetID>-HW2, and upload it to the assignment Canvas page.

Late Policy

All assignments are due **at 11:59 p.m.** 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 a 100-point assignment is due on a Wednesday and it is handed in between any time on Thursday, 10 points 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 three (3) free late days may be used throughout the semester without penalty. Assignment grading questions must be raised with the instructor or a TA within one week after the assignment is returned.

Collaboration Policy

You are to complete the written portions of this assignment individually, and have the option to do the programming section with a partner. 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 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.

Problem 1: Hill-Climbing Search [17]

At a library open from 1:00PM to 6:00PM, there are five student workers -- A, B, C, D and E -- and five hourly shifts available. You need to assign the workers to hourly shifts, with one shift per student. Because the students have different classes, they are available in different timeslots.

This can be formalized as **a local search problem** as follows:

- Each state *s* is a *schedule* or permutation of the 5 workers [A, B, C, D, E]. It is assumed that the first worker in the list represents the student who works the 1:00PM shift, the second the 2:00PM shift, and so on. *No state has repeated workers*, but not all states satisfy the students'

availability requirements.

- Let the evaluation function f(s) *be the number of workers that are available to work in their assigned shifts at state s*. Because there are 5 workers, $0 \leq f(s) \leq 5$.

- The operator function neighbor(s) for a given state s should return all neighbors of s. Each neighbor differs by exactly one pair of swapped workers. For example, [A,D,C,B,E] is a neighbor of [A,B,C,D,E], because B and D have been swapped. Workers cannot be swapped with themselves, ie. A for A, so a state cannot have itself as a neighbor.

You will perform greedy (not stochastic hill-climbing, not simulated annealing) *hill-climbing search* from starting configuration [C, D, A, B, E]. The students' availability is as follows.

Worker	**1:00PM**	**2:00PM**	**3:00PM**	**4:00PM**	**5:00PM**
A			✓	✓	
B	✓				
C	✓				✓
D		✓	✓		✓
E		✓		✓	

a) [4] Given the neighbor(s) function describe above, how many *unique* neighbors does each state have? (By unique, I mean unique schedule configurations.)

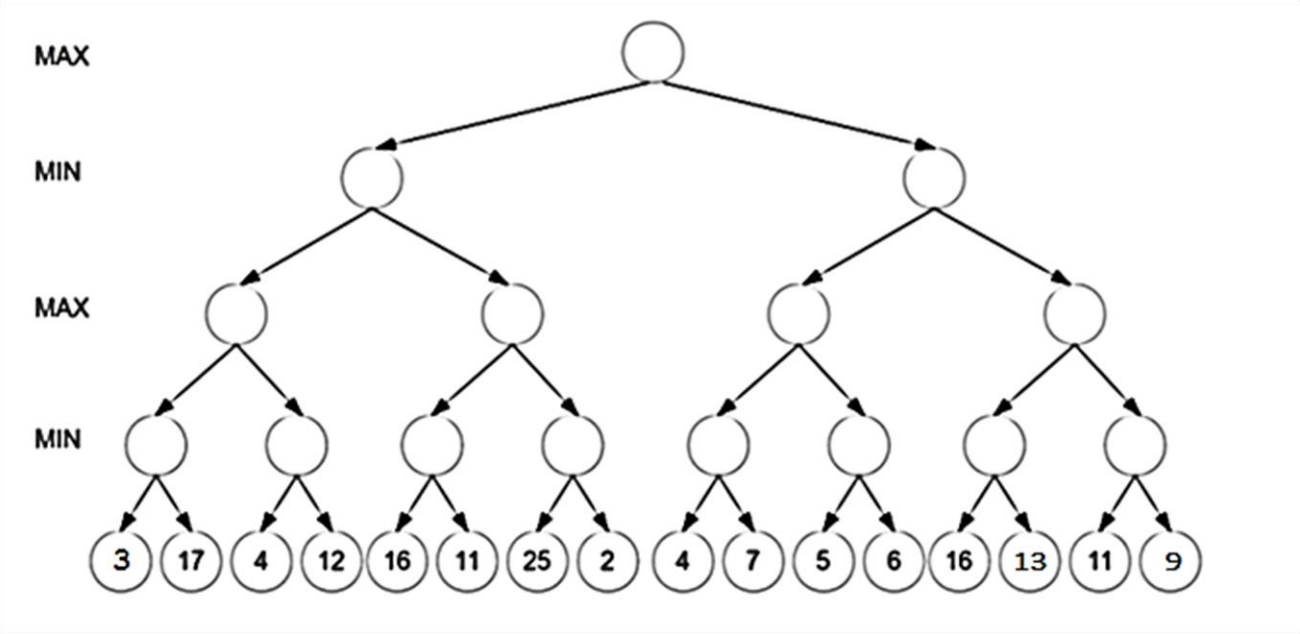
b) [3] How many *unique* schedule configurations are in the *state space?* Explain your reasoning in a sentence or less.

c) [10] Perform greedy hill-climbing search from initial configuration [C,D,A,B,E] until the algorithm terminates. In the event of a tie between two neighboring states, break the tie based on the forward alphabetical ordering of the neighbors' schedules.

At each iteration show 1. the current state 2. its neighbors and 3. the scoring function values of each neighbor. No credit will be given for only showing the schedule at the final state.

If the algorithm terminates at an optimal solution, explain (briefly) why that solution is optimal. If an optimal solution is **not** found, describe an optimal solution, explain (briefly) why it is optimal, and why the algorithm didn't find it.

Problem 2: Minimax Search [16]



Base your answers on the minimax and alpha-beta algorithms in the textbook, which are identical to the ones in the slides.

Above is a game-space graph.

a) [5] Use the minimax algorithm to show how the values at each leaf node percolate up the tree. Which of the two available moves is chosen by the computer, the right or the left?

b) [11] Run the alpha-beta pruning algorithm on the state space graph above (use a separate drawing from a), ***diagramming which branches are expanded and using X's to mark those that are pruned.***

   Indicate **the final alpha and beta values at each node** (this will change throughout for nodes with children).

Problem 3: Constraint Satisfaction [17]

Imagine that you manage a puppy kindergarten, which allows puppies of all breeds to attend. However, *small* dogs are not allowed to sit next to *large* dogs, for fear that Big Buddy will crush Snowball. However, medium dogs are allowed to sit next to large and small dogs.

Variables:

Formally, there are 7 dogs enrolled: (**A**kita - large, **B**asset Hound - medium, **C**hihuahua – small , **D**achshund - small, **E**lkhound - medium, **F**rench Bulldog – small , **G**olden Retriever - large).

Domain:

Each of these 7 dogs can take one of 7 available places represented as i, j coordinates. Two places in the grid aren't available for the dogs because of other objects in the room. The seats are arranged as follows:

Water Bowls	L	**B**
0,0	1,0	**Chew Toys**

Hence, d = { (0,0), (1,0), (1,1), (2,1), (2,2), (1,2), (0,2) }.

Constraints:

No two dogs can sit in the same place. No small dogs can sit next to large dogs.

a) [3] Draw a **constraint graph** with the variables [A,B,C,D,E,F,G] as *nodes* and canSitNextTo relationships as *edges*. That is, represent the dogs who *can* sit next to each other as connected in the graph. If two dogs cannot sit next to each other, do not connect them with an edge.

b) [4] Formalize the dogs that **cannot** sit next to each as binary numeric constraints relative to the each dog's current coordinate position. (See the n-queens example in the lecture slides.)

c) [8] Assume that an owner has requested that G sit at (1,1) and another

has requested B sit at (2, 1). No other dogs have been assigned places.

Perform a (pseudo) iteration of DFS search with forward checking from the current state.

Draw array that shows the remaining legal values (possible coordinates) for each variable (dog) with this configuration. *(Check the lecture slides on forward checking)*

Identify the most-constrained variable(s), the most-constraining variable, and the least-constraining value for the most constraining variable, and whether the algorithm would backtrack from this state.

d) [2] Can an optimal solution be reached from this state? Why or why not? If yes, give an optimal solution, if not, propose an alternative assignment of dogs to places.

Problem 4: Game-Playing Implementation [65]

This problem may be done either individually or in pairs. Find a teammate on your own or via Piazza. Write a Java program that plays the game Mancala. The game is two-player, turn-based, and uses a board with 12 small *bins* (6 for each player) and two *mancalas* (the large "scoring" bins, one for each player), and a set of small stones. The 6 small bins in front of you and the mancala on your right are yours. The ones across from you and to your left belong to the opponent.

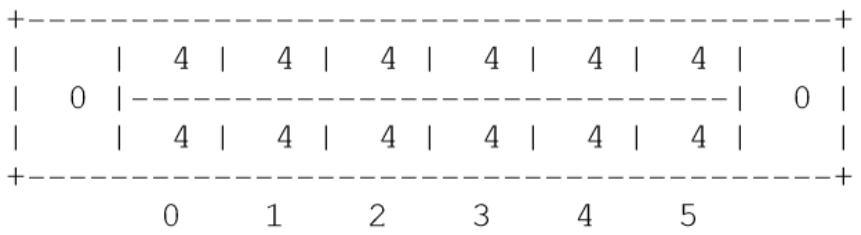
Mancala Resources

Odds are, some of you are unfamiliar with how to play Mancala. The rules are detailed below, but here are two versions that you can play online to familiarize yourself with the game dynamics and strategies (note that the rules for these versions might vary slightly from ours, though not by much):

- Mancala Snails!
- Mancala Web v2.0

Rules of Play

There are many different versions of rules for this game, but the ones we will use are as follows. The game starts with 4 stones (or sometimes 3 or 5) in each small bin. The object of the game is to collect the most stones in your mancala.

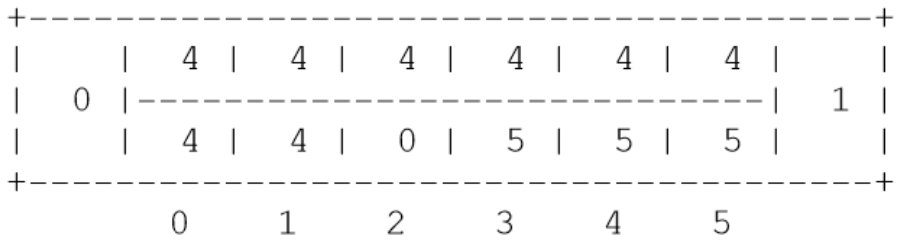


On your turn, you select one of your own bins. The bin must have one or more stones in it. You pick up all the stones in that bin and place one stone in each bin to the right (i.e., going counter clockwise).

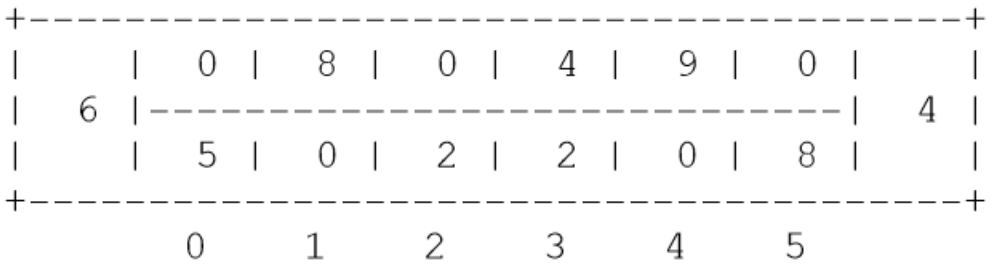
If you still have stones when you reach your mancala, then you put a stone in your mancala, and begin to place stones in each of your opponent's bins going back around.

If you still have stones when you reach your opponent's mancala, skip it and proceed through your bins again, and your mancala, etc., until you run out of stones.

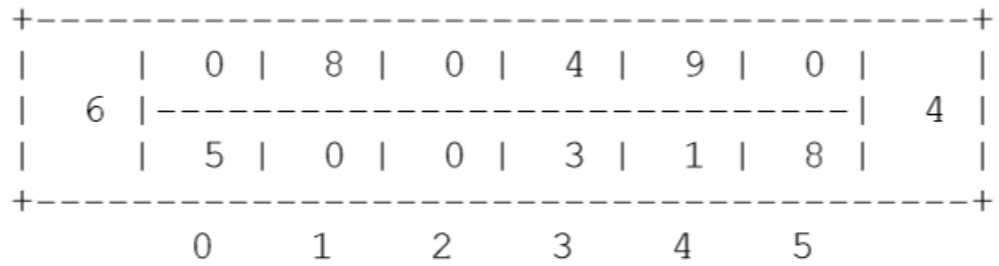
For example, if you have the first move, you could pick up the stones in bin 2, which results in this board:



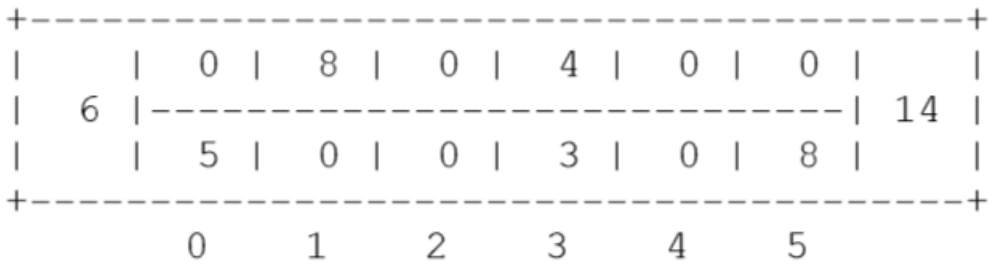
Note that your last stone ended up in your mancala. When this happens, you get to take another turn. Now suppose that later on in the game, the board looks like this:



This time when you choose bin 2, you get the following board:



Note that the last stone ended up in an *empty* bin *that belongs to you*. When this happens, you get to take that stone, plus *all* of the stones in the opponent's bin on the opposite side, and put them in your mancala:



And then it is your opponent's turn. The game is over when all the bins on one side of the board are empty. All the remaining stones go into the opponent's mancala (e.g. if Min has no more stones on her side of the board, then Max gets to put all of the remaining stones in his mancala, and the game is over). The

player with the most stones at the end of the game wins.

Writing a Mancala-Playing Agent
We have provided a skeleton program and a Mancala game framework, which handles all of the game board data structures, rule aspects of the game, and provides a GUI interface for playing against your agent. In this framework there is an *abstract* class defined in the file Player.java which defines all the methods necessary for some agent to interface with the game framework.

To compile everything, use this command:
% javac *.java

Your Programming Task
Your task is to write an xxxPlayer.java file, where xxx is your wisc username. For example, jdoe@wisc.edu's code would be written in jdoePlayer.java The skeleton of xxxPlayer.java is provided to you as studPlayer.java.

If you work with a partner, the Player file can be named using either partner's username and only one of you needs to submit your java files, but both of you need to submit a README.txt file.

It is important that your login name appears verbatim, and "Player" is capitalized, with no dashes or underscores or anything else. This file will implement the xxxPlayer class, which extends the *abstract* Player class. Here you will write code for the *abstract* methods defined in Player.java.

There are four things required for your implementation:

1. Minimax search
2. Alpha-beta pruning
3. Time management with iterative-deepening search (IDS)
4. A static board evaluation (SBE) function

Specifically, you must implement six functions in xxxPlayer class. They are:

1. *public void move (GameState state);*

2. *public int maxAction(GameState state, int maxDepth);*

3. *public int minAction(GameState state, int maxDepth);*

4. *public int maxAction(GameState state, int currentDepth, int maxDepth, int alpha, int beta);*

5. *public int minAction(GameState state, int currentDepth, int maxDepth, int alpha, int beta);*

6. *private int sbe(GameState state);*

More details on these six functions are given below.

1. **public void move(GameState state);**

This method will implement the IDS algorithm to update the protected data member move after each iteration of IDS. (The getMove() method then returns that data member to the class that is controlling the game environment.)

The maxDepth of the IDS algorithm starts from 1 and increments by 1 at each iteration until interrupted due to the time limit. Inside each iteration, you need to do a Minimax search with maxDepth.

When move(GameState state) is called, the bottom row of the board is your side. Be sure that when you start each new iteration of IDS you get a copy of the current state before changing it.

2. **public int maxAction(GameState state, int maxDepth);**

This is a wrapper function for Minimax search for ease of use. The caller of this function should be the Max player. The detailed descriptions of input and output are given below:

@GameState state: The game state for the current player

@int maxDepth: The maximum depth you can search in the search tree

@return: Return the best move that leads to the maximum SBE value. You may alter this to return the (best move, SBE value) if you want.

3. **public int minAction(GameState state, int maxDepth);**

This function is similar to public int maxAction(GameState state, int maxDepth) except this function returns the best move for the Min player.

4. **public int maxAction(GameState state, int currentDepth, int maxDepth, int alpha, int beta);**

This function will actually do the Minimax search with Alpha-Beta pruning. The detailed descriptions of input and output are given below:

@GameState state: The game state we are currently searching

@int currentDepth: The current depth of the game state we are searching

@int maxDepth: The maximum depth we can search. When current depth equals maxDepth, we should stop searching and call the SBE function to evaluate the game state

@int alpha: alpha value for alpha-beta pruning

@int beta: beta value for alpha-beta pruning

@return: Return the best move that leads to the child that gives the maximum SBE value; return the move with the *smallest index in the case of ties.*

It is important to note that we will also call the SBE function to evaluate the game state when the game is over, i.e., when someone has won the game.

5. **public int minAction(GameState state, int currentDepth, int maxDepth, int alpha, int beta);**
6.

This function is similar to public int maxAction(GameState state, int currentDepth, int maxDepth, int alpha, int beta) except this function returns the best move for the Min player.

7. **private int sbe(GameState state);**

This function takes a game state as input and returns its SBE value. You may implement a simple SBE method as follows: Return the number of stones in the mancala of the current player minus the number in the mancala of the opponent. You may want to design a better SBE if you want to gain more points for the homework.

Running the Program

In the provided framework is the Mancala class, which has the main method and allows you to just play the game, play against your agent, or pit two agents against each other by using the command:

% java Mancala (xxxPlayer) (xxxPlayer)

The arguments are the agent class names. With no parameters, it is a two human player game. With one argument, you play against the provided agent, and with 2 arguments, the agents will play each other.

The match gives the agents a 10-second time limit by default in which to decide on moves, so alpha-beta pruning and IDS are going to be essential to searching for good moves. *For debugging, the time limit can be adjusted through the TIME_LIMIT variable in the Mancala.java class.*

The search may be interrupted by the game environment when the time limit up, which is why you store the best move so far in the data member move.

TIPS:

1. Start early. No cheating!
2. **Write a random player to test your code! Use the java.util.random library to help you. A properly written minimax AI should consistently beat a random player.**
3. If you are using javac to compile, then put wood-texture.jpg into the src folder. If you are using Eclipse, wood-texture.jpg needs to be outside the src folder.
4. You will probably want to refer to the pseudocode on page 170 in the textbook but you will need to add a depth limit (as in the example from class) for IDS purposes.
5. The GameState class has a copy constructor and an applyMove() method. You'll want to use these.
6. We recommend that you do *not* modify the files we've provided. Your agent *must* be compatible with the framework for grading.
7. All you need to submit are xxxPlayer.java and any additional helper classes you may have written, together with a README.txt file.

Grading

Your agent will play against several of our own agents and your grade will be based on the match results. More specifically:

- If your agent can beat our random agent, which makes a random move at each step, you will get half the total points.

- If your agent can beat our simple agent (implemented using a simple SBE), you will get 100% of the points. We will test two cases, one where your agent moves first and one where our agent moves first. You will get full points if your agent wins either of the two cases.

- Finally, we will hold an automated tournament where we play your AIs against each other. If your AI finishes in the top 10%, you get 10 extra credit points on this assignment. If your AI finishes in the remainder of the top 25%, you get 5 extra credit points on this

assignment. *There's no penalty whatsoever for doing badly in this tournament, so focus on getting the core algorithm right before coming up with a killer SBE.*

Have fun!