CS 61B Project 2
Network (The Game)
Due 3pm Wednesday, March 22, 2000

Warning: This project is more time-consuming than Project 1. Start early.

This is a team project. Form a team of 2 or 3 people. No teams of 1 or teams of 4 or more are allowed.

Copy the Project 2 directory by doing the following, starting from your home directory. Don't forget the "-r" switch in the cp command.

```
mkdir pj2
cd pj2
cp -r $master/hw/pj2/* .
```

Suggested Timeline (if you want to finish on time)
==========

Design the classes, components, and interfaces (see "Teamwork").	March 3
Have working code for the easier components.	March 8
Have working code for identifying a network; progress on search.	March 15
Finish project.	March 21

Network
=======
In this project you will implement a program that plays the game Network against a human player or another computer program. Network is played on an 8-by-8 board. There are two players, "Black" and "White." Each player has ten chips of its own color to place on the board. White moves first.

	10	20	30	40	50	60	
01	11	21	31	41	51	61	71
02	12	22	32	42	52	62	72
03	13	23	33	43	53	63	73
04	14	24	34	44	54	64	74
05	15	25	35	45	55	65	75
06	16	26	36	46	56	66	76
	17	27	37	47	57	67	

The board has four goal areas: the top row, the bottom row, the left column, and the right column. Black's goal areas are squares 10, 20, 30, 40, 50, 60 and 17, 27, 37, 47, 57, 67. Only Black may place chips in these areas. White's goal areas are 01, 02, 03, 04, 05, 06 and 71, 72, 73, 74, 75, 76; only White may play there. The corner squares--00, 70, 07, and 77--are dead; neither player may use them. Either player may place a chip in any square not on the board's border.

Object of Play
==========
Each player tries to complete a "network" joining its two goal areas. A network is a sequence of six or more chips that starts in one of the player's goal areas and terminates in the other. Each consecutive pair of chips in the sequence are connected to each other along straight lines, either orthogonally (left, right, up, down) or diagonally.

The diagram below shows a winning configuration for Black. (There should be White chips on the board as well, but for clarity these are not shown.) Here are two winning black networks:

50 - 52 - 42 - 15 - 45 - 67
30 - 36 - 45 - 15 - 42 - 64 - 67

			BB		BB			_0
								_1
				BB	BB			_2
								_3
						BB		_4
	BB			BB				_5
			BB					_6
		BB				BB		_7
0_	1_	2_	3_	4_	5_	6_	7_	

An enemy chip placed in the straight line between two chips breaks the connection. In the first network listed above, a white chip in square 56 would break the connection to Black's lower goal.

Although more than one chip may be placed in a goal area, only one chip in each goal area can be used as part of a network. Neither of the following are networks, because they both make use of two chips in the upper goal.

```
30 - 50 - 52 - 42 - 64 - 67
50 - 52 - 30 - 36 - 45 - 67
```

A network cannot pass through the same chip twice, even if it is only counted once. For that reason the following is not a network.

```
50 - 52 - 42 - 15 - 45 - 42 - 64 - 67
```

A network cannot pass through a chip of the player's own color without turning a corner. Because of the chip in square 36, the following is not a network.

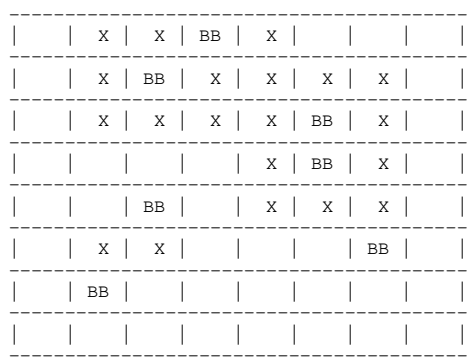
```
50 - 52 - 42 - 15 - 45 - 36 - 27
```

Legal Moves
==========
To begin the game, choose who is Black and who is White in any manner (we use a random number generator). The players alternate taking turns, with White moving first.

The first three rules of legal play are fairly simple.
1) No chip may be placed in any of the four corners.
2) No chip may be placed in a goal of the opposite color.
3) No chip may be placed in a square that is already occupied.

The fourth rule is a bit trickier.
4) A player may not have more than two chips in a connected group, whether connected orthogonally or diagonally.

This fourth rule means that you cannot have three or more chips of the same color in a cluster. A group of three chips form a cluster if one of them is adjacent to the other two. In the following diagram, Black is not permitted to place a chip in any of the squares marked with an X, because doing so would form a group of 3 or more chips. (Of course, White's goal areas and the corners are also off-limits.)



There are two kinds of moves: add moves and step moves. In an add move, a player places a chip on the board (following the rules above). Each player has ten chips, and only add moves are permitted until those chips are exhausted. If neither player has won when all twenty chips are on the board, the rest of the game comprises step moves. In a step move, a player moves a chip to a different square, subject to the same restrictions. A player is not permitted to decline to move a piece (or "move from square ij to square ij").

A step move may create a network for the opponent by unblocking a connection between two enemy chips. If the step move breaks the network at some other point, the enemy does not win, but if the network is still intact when the chip has been placed back on the board, the player taking the step move loses. If a player makes a move that results in both players completing a network, the other player wins.

Your Task
=========
You job is to implement a MachinePlayer class that plays Network well. One subtask is to write a method that identifies legal moves; another subtask is to write a method that finds a move that is likely to win the game.

The MachinePlayer class is in the player package and extends the abstract Player class, which defines the following methods.

```
  // Returns a new move by this player.  Internally records the move as a move
  // by this player.
  public Move chooseMove();

  // If the Move m is legal, records the move as a move by the opponent and
  // returns true.  If the move is illegal, returns false without modifying
  // the internal state of this player.  This method allows your opponents to
  // inform you of their moves.
  public boolean opponentMove(Move m);

  // If the Move m is legal, records the move as a move by this player and
  // returns true.  If the move is illegal, returns false without modifying
  // the internal state of this player.  This method is used to help set up
  // "Network problems" for your player to solve.
  public boolean forceMove(Move m);
```

In addition to the methods above, implement two constructors for MachinePlayer.

```
  // Creates a machine player with the given color.  Color is either 0 (black)
  // or 1 (white).
  public MachinePlayer(int color)

  // Creates a machine player with the given color and search depth.  Color is
  // either 0 (black) or 1 (white).
  public MachinePlayer(int color, int searchDepth)
```

As usual, do not change the signatures of any of these methods; they are your interface to other players. You may add helper methods.

Your MachinePlayer must record enough internal state, including the current board configuration, to choose a good (or at the very least, legal) move. In a typical game, two players and a referee each have their own internal representation of the board. If all the implementations are free of bugs, they all have the same idea of what the board looks like, although each of the three uses different data structures. The referee keeps its own copy to prevent malicious or buggy players from cheating or corrupting the board. If your MachinePlayer is buggy and attempts to make an illegal move, the referee will grant the win to your opponent.

The forceMove() method forces your player to make a specified move. It is for testing and grading. We can set up particular board configurations by constructing a MachinePlayer and making an alternating series of forceMove() and opponentMove() calls to put the board in the desired configuration. Then we will call chooseMove() to ensure that your MachinePlayer makes a good choice.

Most of your work will be implementing chooseMove(). You will be implementing an algorithm for searching game trees, which will be described in Lecture 17. A game tree is a mapping of all possible moves you can make, and all possible responses by your opponent, and all possible responses by you, and so on to a specified "search depth." You will not need to implement a tree data structure; a "game tree" is the structure of a set of recursive method calls.

The second MachinePlayer constructor, whose second parameter searchDepth is the chosen search depth, is also used for debugging and testing your code.

A search depth of one implies that your MachinePlayer considers all the moves and chooses the one that yields the "best" board. A search depth of two implies that you consider your opponent's response as well, and choose the move that will yield the "best" board after your opponent makes the best move available to it.

The first constructor should create a MachinePlayer whose search depth you have chosen so that it always returns a move within five seconds. (We will use the second constructor to run tests that may need to search more deeply than your default.) You will probably want to reduce your search depth when all the chips have been placed, because there are generally many more possibilities for step moves than for add moves. However, only the first constructor is allowed to create a MachinePlayer that adjusts its own search depth; the second constructor must always create a MachinePlayer that searches to exactly the specified search depth.

The Move class in Move.java is a container for storing the fields needed to define one move in Network. It is not an ADT and it has no interesting invariants, so all its fields are public. It is part of the interface of your MachinePlayer, and you may not change Move.java in any way. If you would like to have additional methods or fields, feel free to extend the Move class; your MachinePlayer may return subclasses of Move without any fear.

Strategy
========
Where should you start? First, design the structure of your program (see "Teamwork" below). Then begin by writing a relatively simple MachinePlayer class that simply chooses some correct move, no matter how bad. These actions will give you partial credit on the project. Based on that foundation, you can implement something more sophisticated that incorporates strategy.

Game trees rely on an "evaluation function" that assigns a score to each board that estimates how well your MachinePlayer is doing. An evaluation function is necessary because it is not usually possible to search all the way to the end of the game, so you need to estimate your odds of winning if you make a particular move. Your evaluation function should assign a maximum positive score to a win by your MachinePlayer, and a minimum negative score to a win by the opponent. Assign an intermediate score to a board where neither player has completed a network. One of the most important but difficult parts of implementing game search is inventing a board evaluation function that reliably evaluates these intermediate boards. I leave you to your own wits for this.

You will need to invent an algorithm that determines whether a player has a winning network. A good place to look for clues is Section 9.3.1 of Goodrich and Tamassia, which describes depth-first search in graphs. It's not quite what you need for the job, but close enough that you'll be able to modify it.

To earn full credit, you must implement alpha-beta search, which will be discussed in Lecture 17. Alpha-beta search is a technique for "pruning" a game tree, so you don't need to search the entire tree. Alpha-beta search can be significantly faster than naive tree search. You can earn partial credit by implementing game tree search without pruning. If you can't get that working, you can earn a little bit of partial credit by looking ahead one move.

You will almost certainly want to use a separate class to represent game boards. One decision you will have to make is whether to create a new game board or change an existing one when you consider a move. The latter choice may be slightly faster, but it is likely to cause lots of hard-to-solve bugs if you're not extremely careful about how and when you manipulate game boards.

After the due date, we will hold a tournament pitting student MachinePlayers against each other. Participation in the tournament is optional and does not affect your grade. There is a time limit of five seconds (which will be checked by our refereeing software) on the time to perform one opponentMove() plus one chooseMove(). This will be strictly enforced in the tournament. The winning team will receive gift certificates to Amoeba Music.

This is a difficult project. Do not wait to start working on it. If you don't have the code that identifies legal moves implemented by March 8, you would be well advised to engage in neurotic spasms of worry. We will have autograder software set up to test your code for legal moves.

Teamwork (10% of grade)
========
Before you start programming, break the project up into multiple components (tasks). Decide what high-level methods and classes must be implemented, define the interfaces by which these methods and classes will communicate, and divide up the work among your team. Some possible components, which should be reasonably modular, are

1) determining whether a move is valid,
2) generating a list of all valid moves,
3) finding the chips that form connections with a chip of the same color,
4) determining whether a game board contains any networks for a given player,
5) computing an evaluation function for a board, and
6) performing tree search

The file GRADER provided in the pj2 directory includes a questionnaire, which you are required to submit. Once you've worked out your classes, components, and interfaces, write them down at the bottom of GRADERS. Your description should include:

- A list of the classes your program will need.
- A list of each of the "components" used in or by MachinePlayer, which might be similar to, but more detailed, than the list above. Which class will each component be implemented in? It may make it easier for you to work as a team if each module is in a separate class, but it's not required.
- A list of the interfaces by which your components call each other. Don't include every single method in your project--we will penalize you if you're not selective. Include the interfaces that create a clean separation between the different components you have enumerated. For each interface, provide (1) a method prototype and (2) a description of the behavior of the method/component (which should also be in the comments in your code).
- Who is assigned the task of implementing each component?

If you have defined your classes and interfaces well, you should be able to implement any one of the components without having decided how to implement any of the others. This will allow you to divide up the chores and work quickly as a team. You will probably need to change some of your design decisions as you go; be sure to modify the GRADERS file to reflect these changes.

Before you submit, make sure your GRADERS file tells us who _actually_ implemented each portion of your project. Although you must hand in GRADERS with your project, you must also hand in a printed version of GRADERS on which you have written "This is a truthful statement of how we divided the labor for this project." ALL of your team members must put their signatures under this statement. This statement is not due until after the project deadline, but you will not receive a grade until you turn it in.

Your design of classes and interfaces with be worth about 10% of your grade.

Running Network
==========
You may run Network from your pj2 directory with one of the following commands.

  java Network
    A human player competes against a very naive machine player that makes
    random legal moves. Use this to learn to play the game.

  java Network humans
    Two human players compete against each other. The two players share the
    same mouse.

  java Network machine
    A human player competes against your MachinePlayer.

  java Network machines
    Your MachinePlayer competes against the random machine player.

  java Network machine machines
    Your MachinePlayer competes against itself.

Submitting your Solution
==========
Be sure that you have answered all the questions in GRADER before submitting.
Don't forget that it's worth 10% of your grade.

Designate one member of your team to submit the project. If you resubmit, the
project should always be submitted by the same student. If for some reason a
different partner must submit (because the designated member is out of town,
for instance), you must send cs61b@cory a listing of your team members,
explaining which of them have submitted the project and why. Let us know which
submission you want graded. If you've submitted your project once, or even
written a substantial amount of code together, you may not change partners
without the permission of the instructor.

The designated teammate only: Change (cd) to your pj2 directory, which should
contain the player directory (package), which should contain your
MachinePlayer.java and any other Java files it depends upon. Type "submit
pj2". You won't be able to submit Move.java and Player.java, because you're
not allowed to change them.

Grading
=======
Your project will be graded in part on correctness and the quality of moves
chosen by chooseMove(). This grading will be done using automatic test cases.
Be sure the following statements apply to your chooseMove().

  1) forceMove and opponentMove return true if the given move is legal.
  2) forceMove and opponentMove return false if the given move is illegal.
  3) chooseMove returns only legal moves.
  4) If a winning move exists, chooseMove selects one. (This will happen
      automatically if you are searching one level of the game tree.)
  5) If you cannot win in this step, but can prevent your opponent from
      winning during its next move, chooseMove selects a move that does this.
      (This will happen automatically if you are searching two levels of the
      game tree.)
  6) Your player can beat a "random" player that picks a random legal move.
      We have supplied such a player for you to test against. Any reasonable
      search strategy should accomplish this.

You will also be graded on style, documentation, efficiency, and the use of
encapsulation.

  1) Each method must be preceded by a comment describing its behavior
      unambiguously. These comments must include descriptions of what each
      parameter is for, and what the method returns (if anything).
      Some methods serve as entry points to the "components" you designed when
      you began the project. The prototypes and behavioral descriptions of
      these methods are interfaces, and should be included in GRADERS.
  2) All classes, fields, and methods must have the proper public/private/
      protected/package qualifier. We will deduct points if you make things
      public that could conceivably allow a user to corrupt the data structure.
  3) There are no asymptotic limits on running time. However, part of your
      job is to avoid using inefficient algorithms and data structures. If
      your MachinePlayer takes substantially longer than 10 seconds to search
      to a depth of two on a 277 Soda machine, we will scrutinize your
      submission for inefficient algorithms and data structures.
  4) You should have divided up the tasks into well-defined components in your
      GRADER file and in your software.
  5) We will deduct points for code that does not match the following style
      guidelines.

  - Classes that contain extraneous debugging code, print statements, or
    meaningless comments that make the code hard to read will be penalized.
  - Your file should be indented in the manner enforced by Emacs (e.g., a
    two-space indentation inside braces).
  - All if, else, while, do, and for statements should use braces.
  - Your solution should produce no compiler warnings (e.g., uninitialized
    variables).
  - All classes start with a capital letter, all methods and (non-final) data
    fields start with a lower case letter, and in both cases, each new word
    within the name starts with a capital letter. Constants (final static
    fields) are in all capital letters.
  - Numerical constants with special meaning should always be represented by
    all-caps "final static" constants.
  - All class, method, field, and variable names should be meaningful to a
    human reader.
  - Methods should not exceed about 100 lines; any method that long can
    probably be broken up into logical pieces. The same is probably true for
    any method that needs more than 6 or 7 levels of indentation.
  - Avoid unnecessary duplicated code; if you use the same (or very similar)
    ten lines of code in two different places, those lines should probably be
    a separate method call.
  - Programs should be easy to read.

Finally, we will be looking at your code to see whether you have implemented
game tree search, and whether you use alpha-beta pruning.