CS 540: Introduction to Artificial Intelligence
Homework Assignment #2: Search and Game Playing

Assigned: Thursday, September 25
Due: Wednesday, October 8

Late Policy:

Homework must be handed in by noon on the due date and electronically turned in by this same time.

- If it is 0 – 24 hours late, including weekend days, a deduction of 10% of the maximum score will be taken off in addition to any points taken off for incorrect answers.
- If it is 24 – 48 hours late, including weekend days, a deduction of 25% of the maximum score will be taken off in addition to any points taken off for incorrect answers.
- If it is 48 – 72 hours late, including weekend days, a deduction of 50% of the maximum score will be taken off in addition to any points taken off for incorrect answers.
- If it is more than 72 hours late, you will receive a '0' on the assignment.
- In addition, there are 2 'late days' you may use any time throughout the semester. Each late day has to be used as a whole – you can't use only 3 hours of it and "save" 21 hours for later use.

Collaboration Policy:

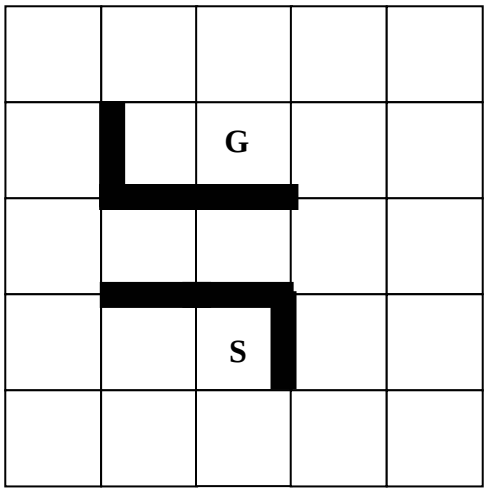
You are to complete this assignment individually. However, you are encouraged to discuss the general algorithms and ideas with classmates, TA, and instructor in order to help you answer the questions. You are also welcome to give each other examples that are not on the assignment in order to demonstrate 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: Informed Search

In the figure to the right is a simple maze on a 5 × 5 board. The task is to go from square S to square G by moving over a path of squares where consecutive squares must be horizontally or vertically adjacent, i.e., above, below, right or left. A bold line is a "wall" indicating that a move cannot be made across this line. Assume the successor function considers adjacent squares in the following precedence order: up, right, down, and left, though not all of these moves may be legal from a given square.



a) [10] Using **Depth-First Search**, incrementally number the squares in a copy of the above maze that are expanded (including the goal node if it is found). State **S** is expanded first, so mark that square with a "1." Assume **cycle checking** *is* done so that a node is *not* generated in the search tree if the grid square position associated with the node occurs somewhere on the path from this node back to the root node. Highlight the solution path found (if any), or explain why no solution is found.

b) [10] Apply **Uniform-Cost Search** and show the sequence of nodes expanded in a second copy of the above maze. Assume the cost of moving **Down** is 10 units, the cost of moving **Left** is 2, the cost of moving **Up** is 1, and the cost of moving **Right** is 4. Avoid repeated states by assuming you do *not* generate a node if that node's associated grid square has been previously **expanded** (i.e., use the general graph-search algorithm described in Figure 3.19). In the case of ties between non-siblings in the search tree, use FIFO (first in, first out) order to expand first the node that has been on the *fringe* list the longest. In the case of ties between siblings, consider them in the order Up, Right, Down, Left. For each cell expanded (including node **G**), label it with (i) a number indicating when it was expanded (position **S** should be marked "1"), and (ii) the cost, $g(n)$, of getting from **S** to node n. Highlight the solution path found (if any), or explain why no solution is found.

c) [10] In a third copy of the above maze, incrementally number the squares expanded using **Greedy Best-First Search** from **S** to **G**. Unlike (b), assume the cost of all moves is 1. Use as the heuristic function $h(n) = |xn - xg| + |yn - yg|$, where the grid square associated with node n is at coordinates (xn, yn) on the board, and the goal node **G** is at coordinates (xg, yg). Use the same method as in (b) for avoiding repeated states. Break ties in the same way as (b). For each node expanded, show in its square a number indicating when it was expanded (starting with "1" at **S**), and the

value of *h* applied at that node. Highlight the solution path found (if any), or explain why no solution is found.

d) [10] Perform **A* Search** using a fourth copy of the above maze to show the sequence of nodes expanded. Use the same heuristic function, *h*, defined in (c) and assume the cost of each move is 1. Use the A* algorithm in the lecture notes for handling repeated states. Break ties in the same way as (b). For each node expanded, show in its square a number indicating when it was expanded (starting with "1" at **S**), and the value of the evaluation function, *f*, applied at that node. Highlight the solution path found (if any), or explain why no solution is found.

Question 2: State Space and Heuristics in Search

You are designing software to coordinate a pair of robots that drive around a building. Each robot has a radio transmitter it uses to communicate with base stations. You discover that when two robots come too close together, there are interference problems. The problem you have to solve is how to plan a path such that both robots can get to their respective destinations without their radio transmitters interfering.

The building plan is given to you as a graph, where every node in the building graph denotes a particular position in the building (we discretize the positions). Two nodes in the graph are connected by an edge if there is a way for a robot to travel directly between the corresponding positions in the building. All edges in the building graph have the same cost. For robot A you are given a start and destination node (a,b); similarly, for robot B you are given a start and destination node (c,d). We require that robot A and robot B at any point in time must be at least ***t*** nodes apart from each other. Finally, at any particular point in time, only one robot is allowed to move across an edge.

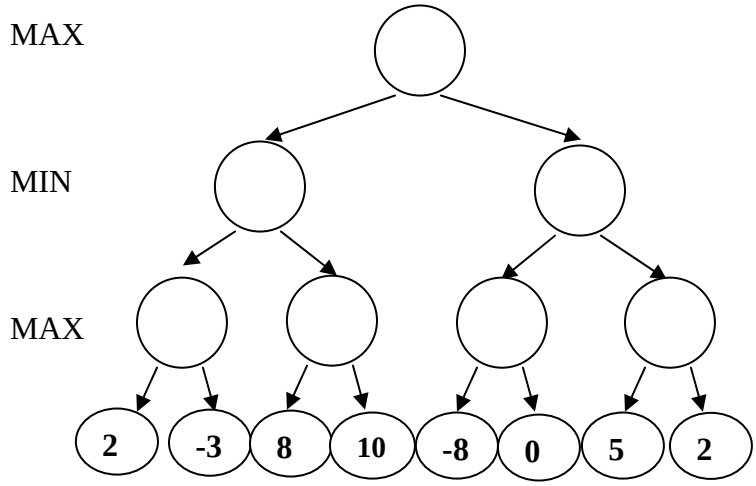
The question is whether you can apply a search algorithm we covered in class to schedule the robots so they both arrive at their respective destinations without their radio transmitters interfering. (Assume robot A is at node a, and robot B is at node b, in the beginning.)

a) [5] Describe how to construct the states for your state space, and the possible legal moves between two states in your state space.

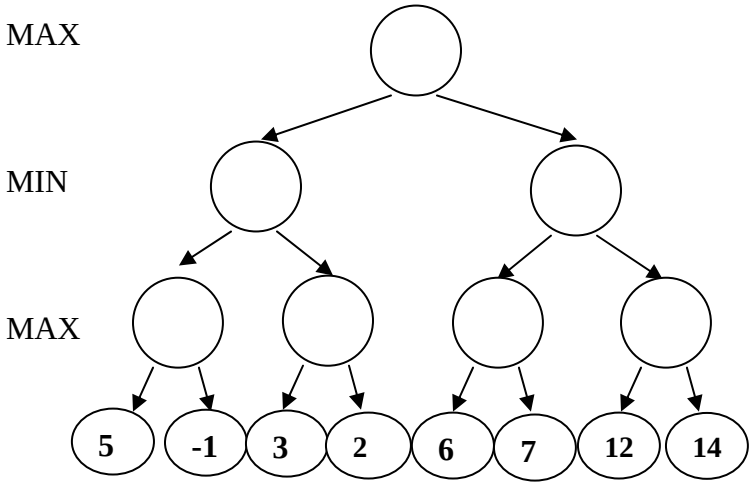
b) [5] Suppose you want to find the schedule for your robots with the least number of total edges traveled. You want to use A* but need to decide on a heuristic function ***h***; what heuristic would you use? Describe the best possible heuristic you can think of and argue why it is good.

Question 3: Game Playing

a) [5] Use Minimax to compute the game value at each node for the game tree below.



b) [5] Use Alpha-Beta Pruning to compute the game value at each node for the game tree blow. Show the alpha and beta values at each node. Show which branches are pruned.



Question 4: [40] Checkers

For this question you will 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. We have written most of the classes necessary, so the amount of code you'll have to write is not excessive. You can download the provided code from the course webpage.

Existing Framework

The following section describes the existing classes and how you should integrate your code with them.

`Checkers.java` :: This is the class responsible for driving the Checkers program. **You will have to modify the section of code that is clearly labeled**, but you are not required to modify anything else from this class (unless you want to).

`CheckerBoardConstants.java` :: This class contains all the constants used by the provided code. You should become familiar with these constants, especially the ones that are used by methods from the Utils class.

`CheckerPlayer.java` :: This is the class that each of your Checkers players must extend. It contains an abstract `calculateMove` method. You must implement that method in any class that extends `CheckerPlayer`. This method is where you will focus your attention for this assignment. You may write other methods and classes to aid the code you write in that method, but everything you do should revolve around `calculateMove`. There is also a `chosenMove` variable that you will be updating within the `calculateMove` method. If you don't update this variable, your player will lose. You should set the `chosenMove` variable every time you find a better move during the calculation because the calculate move method will be interrupted after some amount of time (MOVE_DELAY).

`Location.java` :: This class is used to represent a location on the checkerboard. A location is a row and a column. The rows and columns are indexed starting at 0. Row 0 is at the top and row 7 is at the bottom. Column 0 is at the left and column 7 is at the right.

`Move.java` :: This class is used to represent a move on the checkerboard. One move consists of locations including the starting location, the ending location, and possibly intermediate locations in the case of multiple jumps.

`RANDOMPlayer.java` :: This class has been provided as an example of a Checkers player. You should look at the code for this class to help you understand what you need to do. You can also have your computer player play the random player. You should be able to beat the random player every time if you have any reasonable player.

`Utils.java` :: This class contains various methods that you should find helpful. You should read through the documentation for this class carefully. You may also find it helpful to look through the code for some of the methods in this class. Most things are already done for you, and you can just call one of the methods from this class. For example, you can get all possible moves the red player can make if you pass in the board state and an indicator for the red player by calling `Utils.getAllPossibleMoves(boardState, RED_PLAYER)`.

Requirements

Your task is to create two players that can interface with the existing Checkers framework:

1. Implement a basic alpha-beta player that uses the provided static board evaluator (SBE) defined in `Utils.scoreCheckerBoard` to estimate the value of a non-terminal board state. The skeleton of this class is provided for you in `AlphaBetaPlayer.java`. Modify this file by filling in the `calculateMove` method with code that performs alpha-beta pruning to calculate the next move. You must use the `Utils.getAllPossibleMoves` method to retrieve the possible moves, and you must examine those moves in order, starting with the move at index 0 in the vector. You must implement alpha-beta pruning using an iterative deepening search that continues until the time limit is up. If the program is NOT in debug mode, you should NOT print out any additional information. If the program is in debug mode, after you complete each depth limit, you must print out:

```
'Done with depth limit x'
'I pruned the tree y time(s)'
```

For example:

```
Done with depth limit 1
I pruned the tree 0 time(s)
Done with depth limit 2
I pruned the tree 1 time(s)
Done with depth limit 3
I pruned the tree 6 time(s)
Done with depth limit 4
I pruned the tree 13 time(s)
```

This would be what is printed out if the player was able to reach a depth limit of 4 during the allotted time.

You must conform to these specifications for the purposes of grading. We will be comparing your output to the output produced by our alpha-beta player, and they should only differ in the depth limit that was reached, which will depend on the efficiency of the implementation.

2. Implement a second alpha-beta pruning player that uses your own static board evaluation function to estimate the value of non-terminal board state. Write the code for your static board evaluator function within your player class. For this class, you will name it `YOURLOGINAlphaBeta.java`. This class will extend the `CheckerPlayer.java` class. You can use `AlphaBetaPlayer.java` as the skeleton for this class.

   Look at `Utils.scoreCheckerBoard` to get some idea of the basic SBE function. Be warned that `Utils.scoreCheckerBoard` is a very simple SBE function, which uses only the number of red and black pieces on board. You must add more features and combine them to create a novel and effective SBE. In your documentation report write a description of your SBE function, explaining each feature in terms of what it computes and why it's included. Also give comments on how your SBE performs against the `basicAlpha` player or people, citing strengths and weaknesses that it seems to exhibit.

   In addition to creating your own SBE, you may also modify or extend the basic alpha-beta search procedure to make it better. See Chapter 6 for some ideas on this. Note: You are *not* required to modify alpha-beta to get full credit for this player, but it should help you improve your performance in the class tournament (see below).

   As in Homework #1, compilation, during grading, will be done by executing the command 'javac *.java' at the command prompt on the Linux machines. After compilation, the code will be executed with the line

```
java Checkers -red_player -black_player [-delay 1000] [-step] [-debug]
```

   - `red_player` is the specification of who you want to be the red player

   - `black_player` is the specification of who you want to be the black player

     The possible players are:
     - `random` (this is the random player provided for you)
     - `samplePlayer` (this is the player provided for you for illustration purposes)
     - `basicAlpha` (this is the alpha-beta player that you will create using the provided SBE function given in `Utils.scoreCheckerBoard`)
     - `YOURLOGIN` (this is your "tournament" player, which is your best player using your own SBE, alpha-beta and whatever other improvements you want to make)
     - `human` (this is a player controlled by human clicks)

   - `delay 1000` is the option that specifies how much time (in msec) you will give the computer player to think. This example will give computer player 1 second to think.

   - `step` is the option for making the players move on the click of the mouse. You should only use this option if you are watching two computer players play each other.

- `debug` is the option that will set the debug flag to true in the `Utils` class. You can then choose to only print when the program is in debug mode.

For example,

```
java Checkers -basicAlpha -human -delay 5000
```

will run a game with red being controlled by your basic alpha-beta player, which is given 5 seconds per move, and the black player is controlled by a human player

```
java Checkers -YOURLOGIN -YOURLOGIN -delay 5000 -debug
```

will start a game with both red and black players using your player that uses your own SBE. Both players are given 5 seconds to make a decision, and the debug flag in the `Utils` class is set to true.

```
java Checkers -YOURLOGIN -basicAlpha -delay 5000 -step
```

will run a game with red being your tournament player and the black being the basic alpha-beta player that uses the provided SBE function. Both players are given 5 seconds to make a decision, and each player does not start computing its decision until you click the board with the mouse.

Note:

In the past, there are several known confusions:

1. When running Checkers with two automated players (i.e. not "-human"), the user interface normally will not show the very last move, i.e., when one player would win. The most obvious symptom is that you never see "RED wins" or "BLACK wins!" after the winning move is reported as chosen on the user interface. It just doesn't show the final board state. However, it will print out a message in the command line window.
2. The assignment requires you to use `Utils.scoreCheckerBoard()` to score non-terminal board states in your AlphaBetaPlayer implementation. However, be aware that `Utils.scoreCheckerBoard()` returns the score from the RED's standing point of view and thus, you need to negate it for the BLACK's standing point of view.
3. Lastly, there is a suggestion of what to do when your tree is no longer deep (e.g., when you're about to win or lose) so you don't need the full MOVE_DELAY milliseconds to make your decision. We suggest that you should explicitly return after `chosemove` is set.

Initial Board State

The initial state of the board is specified in the file called `boardState`. This allows you to easily set up different situations to test your player.

'-' means a blank
'r' means a red pawn
'R' means a red king
'b' means a black pawn
'B' means a black king

For example, this is the common starting board state:

```
- b - b - b - b
b - b - b - b -
- b - b - b - b
- - - - - - - -
- - - - - - - -
r - r - r - r -
- r - r - r - r
r - r - r - r -
```

To test various situations, just modify the `boardState` file. For example, in the board state below, if the turn belongs to the red player, the best red player move is clearly to move the lower red king in between the two black kings.

```
- - - - - - - -
- - R - - - - -
- - - - - - - -
- - - - R - B -
- - - - - - - -
- - - - B - - -
- - - - - - - -
- - - - - - - -
```

`myAlpha` player's SBE can be evaluated and modified by you using specific board situations until you find one that, hopefully, picks the right move in various circumstances.

What to Turn In

Electronically:

Turn in your basic alpha-beta pruning algorithm that you defined in the file `AlphaBetaPlayer.java` Also turn in all of the classes that you created associated with your player that uses your own SBE function. All these classes should begin with your login in all capital letters. This will allow us to place all students' codes in the same directory during the Checkers tournament. For example, if Mr. Smith wants to create a class called `MySearchObj`, he should call it `SMITHMySearchObj`. Copy it along with any other java files necessary to compile and run your program on the Linux machines into your own private handin directory, say called hw2-handin

Then, execute the following command from the directory containing your own handin directory:

```
handin –c cs540-SECTION –a hw2 –d hw2-handin
```

where SECTION is 1 or 2, depending on your CS540 section. For more information on how to electronically hand in your code, see the course web page. Be sure your code will compile and run on the departmental Linux machines; if it does not, we will not be able to grade it.

Paper:

Hand in in class hard copy of your answers to Questions 1, 2, and 3, a printout of your well-documented source code for Question 4, and a report on your static board evaluation function and any search-related features that you implemented in your "tournament" player. Describe the performance of your best player against humans and your `basicAlpha` player, including observations on the strengths and weaknesses of your player. Staple all pages together and put your name, your login, and the date on the top of the first page.

Checkers Tournament

We will run a Checkers tournament using each student's best player, which will be assumed to be named `YOURLOGINPlayer.java, e.g., SMITHPlayer.java.` All games will be run by the TA. The students with the best results in the tournament will receive prizes and 10 extra credit points for the assignment.