

Software Project, Spring 2015 - Assignment 3

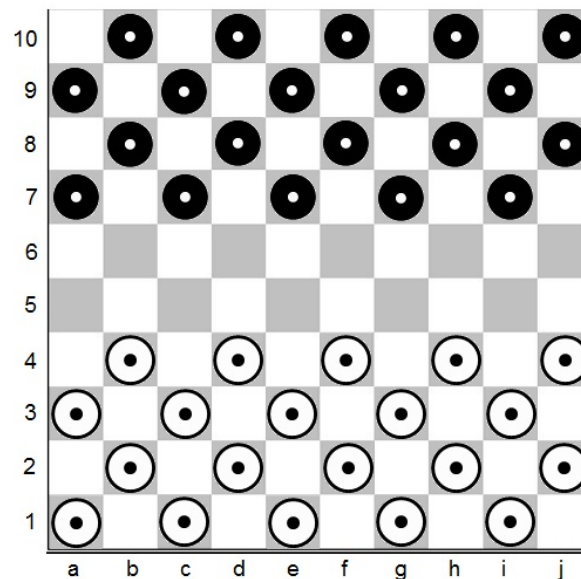
Due by 18.6.2015, 23:59

In this assignment, you are asked to write a C program simulating the Draughts game. The program simulates a game between the user and the computer, where the computer uses a minimax strategy, as illustrated below.

Make sure your implementation meets the code requirements that are specified on page 10.

Background

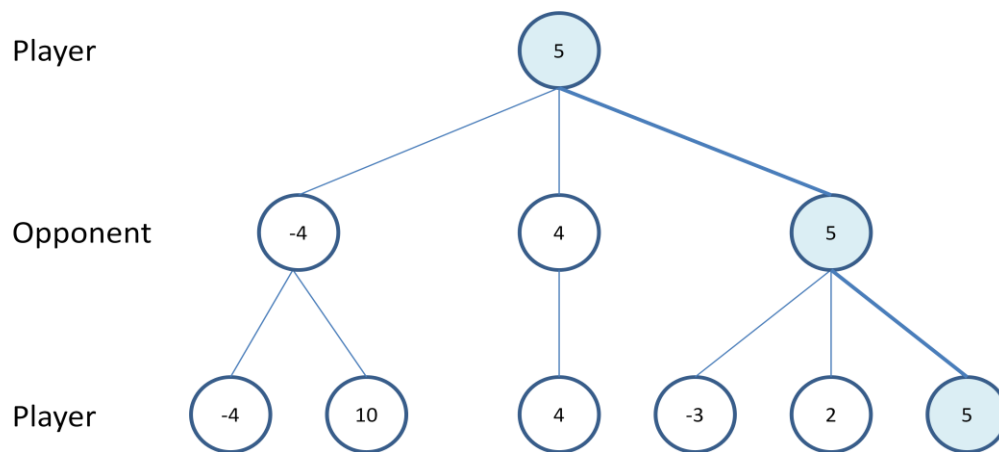
Draughts is a strategy game for two players which involve diagonal moves of uniform game discs and mandatory captures ("eats") by jumping over opponent discs. The game is played on a chess board of the size 10x10. The goal of each player is to capture all of its opponent's discs. Each game must have a winner.



A **minimax** algorithm is a recursive algorithm for choosing the next move in an n -player game, usually a two-player game. A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves.

What this means is that the computer calculates its possible moves, scores each move according to the game state it reaches and possible future states, all according to some scoring function illustrated below. It calculates future moves from each such move, recursively, along with a score for each resulting state, and then picks the best move for the current turn and performs it.

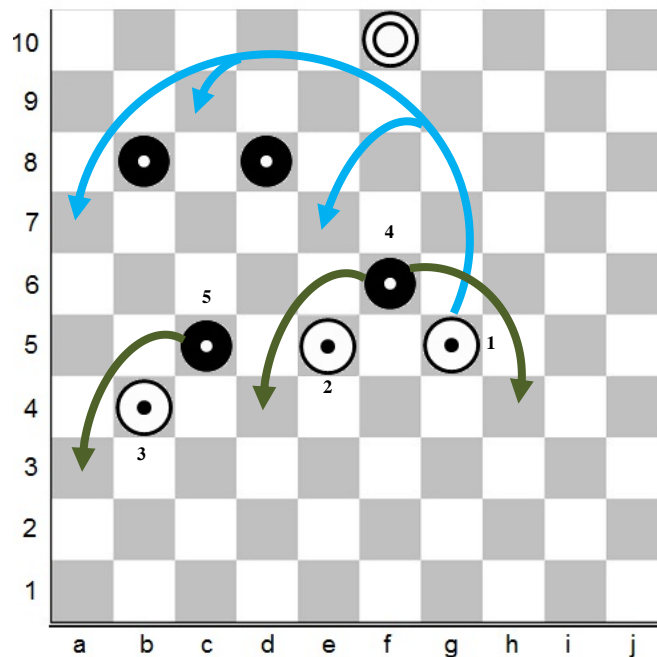
We can represent the results of the minimax algorithm as a graph $G = \langle V, E \rangle$ where the vertices are game states and their score, and the edges are possible moves from each state.



In the figure above, the game contains three legal moves for each turn, and the minimax algorithm calculates two steps ahead, with the score shown as the value of the node.

Draughts Rules:

1. The game is played by two opponents, each plays with either black or white discs. By default, white player starts first.
 2. There are two disc types in the game: man and king. The game starts with only man disks on the board. Each opponent has 20 man disks placed on the dark squares on two opposite sides of the board (4 rows for each player).
 3. Moves are allowed only on the dark squares, so the discs always move diagonally.
 4. Each player can only move its own discs:
 - a. Man: moves one step diagonally, and (may) capture opponents' discs by moving two consecutive steps in the same direction, jumping over the opponent's disc on the first step.
 - b. King: moves any distance along unblocked diagonals, and may capture an opposing man any distance away by jumping to any of the unoccupied squares immediately following it. **once the first capture in a turn is made, the king can proceed to a multiple capture the same as a man does it – capturing only adjacent disks.**
 5. Captured discs are removed from the board.
 6. **A man becomes a king once it reaches the farthest row from its initial position (for white man disks it's the 10th row and for black man disks it's the first row). After a man becomes a king, it cannot move or capture in the same turn.**
 7. Backward movements:
 - a. Man – allowed only for the purpose of a capture.
 - b. Kings – allowed without constraints.
 8. If a player is able to make a capture - the jump must be made. If more than one capture is available, the player must choose the move with the highest number of captures discs. If there are several possible moves, the player is free to choose whichever move he likes.
 9. A player wins the game when the opponent cannot make a move. This happens when the opponent's discs have been captured or blocked.
- The following example illustrates all possible moves for the white and the black player.



- If it's the white player's turn, the only possible move is for disk 1 to make 3 captures.
- If it's the black player's turn, it has 3 options – disk 4 can capture disk 1 or 2, and disk 5 can capture disk 3.

Draughts Program

The program Draughts starts a new game when it's executed. The program operates by reading commands from the user – and the game is played accordingly. The game contains two players: the user and the computer, and they play in turns.

User commands:

1. A command must be specified in its own line (lines are separated by new lines) and must include at least one non-whitespace character.
2. The command parameters (this includes the command name) are separated by at least one whitespace (excluding the ending '\n' character).

The program has two states: settings state and game state. In the settings state the user specifies all game computation setting (such as depth of minimax algorithm), and the game is being played in the game state.

As the first step, the program prints the message `"Welcome to Draughts!\n"` and then moves to the settings state (no user input is expected).

Settings state:

The program initializes the Draughts board. For your convenience, in `Draughts.c` you can find the implementation of two functions:

`print_board` – a function that prints the game board in the correct format to the console.

`init_board` – a function that initializes the board.

You can change these functions as you please, but you must maintain the output format (attached is a graphic representation of an initialized board).

5.
 - command: set <x,y> a b
 - description: this command places a disc of color a and type b in <x,y> position.
 - The value of **a** should be either *white* or *black*. The value of **b** should be either *m* (stands for man) or *k* (stands for king).
 - if **x** values is not within the range of 'a' to 'j' or **y** is not within the range of 1 to 10, or <x,y> is a white square, the program prints the message "Invalid position on the board\n" and the command is not performed.
6.
 - command: print
 - description: this command prints the current game board to the console.
7.
 - command: quit
 - description: this command terminated the program.
8.
 - command: start
 - description: after this command is entered the program move the **game state** and the user cannot enter settings command anymore.
 - In the following cases:
 - the board is empty
 - there are discs of only one color
 - there are more than 20 discs of the same color
 this message will be printed "Wrong board initialization\n" and the command will not be executed.

Game state:

In this stage of the program the game is actually being played. The players are the user and the computer. White player always starts first. In each turn, the current player performs its move and the game continues until one of the players wins or until the game is stopped by the user.

Move representation in the program:

We represent each move by two values:

- initial position
- a list of the positions where the disc visits during its move.

A regular move will have only one position – its destination. When a player captures several discs, its move will contain more than one position in the list.

For example, in the board from page 3 we have disc 1.

Its position is: <g,5>.

It has a single move, capturing <f,6>, <d,8>, <b,8>:

Computer's turn:

The computer makes its move using minimax algorithm with the depth parameter. After computer's move was performed the program prints the move in the following format:

"Computer: move <x,y> to <i,j>[<k,l>...]\n".

<x,y> is the position of the disc that moved and <i,j>[<k,l>...] represents the move that was performed (possibly with more than one stop, hence the optional argument [<k,l>...]).

Following this:

1. The updated board is being printed to the console.
2. The program checks if the computer won the game. If so, the message "`xxx player wins!\n`" is printed (XXX represents the color, either black or white) and the program terminates.
3. In case the game is still on, it's the user's turn now.

User's turn:

When it's the user's turn the message "`Enter your move:\n`" is printed and the program repeatedly gets commands from the user (while it's still the user's turn).

The following commands are available for the user at this stage:

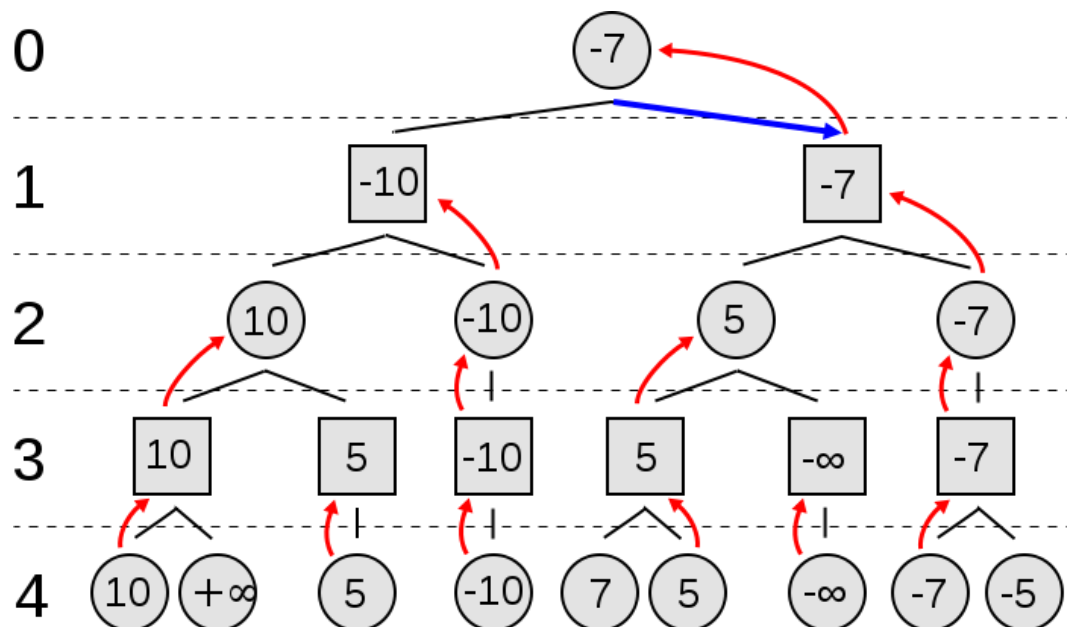
1.
 - command: move <x,y> to <i,j>[<k,l>...]
 - description: This command executes the user turn operating the move represented by <i,j>[<k,l>...] on the disc positioned in <x,y>.
 - example: move <e,3> to <g,5><c,7> (there are no spaces between <g,5> and <c,7>)
 - (1) if either one of the positions in the command is invalid, the program prints the following message "`Invalid position on the board\n`".
 - (2) if position <x,y> does not contain a disc of the user's color, the message "`The specified position does not contain your piece\n`" is printed.
 - (3) If the move is not legal for the disc in the position <x,y>, the message "`Illegal move\n`" is printed.
 - You must print only one error message for each command. The order of the tests is marked with (1), (2), (3) in the previous bullets.
 - If there were no errors and the move was executed:
 - a. The updated board is being printed to the console.
 - b. The program checks if the user won the game. If so, the message "`xxx player wins!\n`" is printed (XXX represents the color, either black or white) and the program terminates.
 - c. In case the game is still on, it's computer's turn now.
2.
 - command: get_moves
 - description: this command prints all possible for the player, each move in a new line (you can print them in any order you wish). A move is represented by the position of the disc and the move itself, as specified before.
3.
 - command: quit
 - description: this command terminated the program.

Minimax algorithm

One of the AI (Artificial Intelligence) algorithms extensively used in two-player games is Minimax.

The minimax algorithm uses a “scoring function”. This function, given an input board, produces a number. The more positive the number is, the better the board is in terms of player A; the more negative the number is, the better the board is in terms of player B. When the minimax algorithm is run for the computer, it is treated as player A. However, when the user inputs the command to suggest a move, the user is treated as player A.

With a scoring function and a way to find allowed moves for a particular board (game state), we can construct a tree like the graph shown in this figure:



Each node represents a game state, with the root node representing the current state of the game. From the root we recurse down the tree, creating nodes representing possible game states that can occur by each of the allowed moves, and stop when we reach a specified depth (determined by *minimax_depth* command, described above).

For example, at the beginning of the game the root node represents a new game state for a blank board, with a score of zero. It has 9 edges – each representing a possible move for the black player (the only discs that can move are discs from row 3 - note that the disc in <a,3> has only one move option). We then expand each state to its own child nodes – each representing the move for the white player. The white player also has 9 possible moves for each state, so we get a total of 81 **new** nodes (game states), and so forth until reaching depth equal to the set number of steps.

Important note: in every game stage you may have a different number of children, so you cannot assume anything about their number.

The algorithm continues to create the tree until we reach the maximum depth level, and we then need to apply the scoring function. This creates scores for all the “leaf” nodes of the tree, and we apply a simple strategy for the other nodes (consider the root as level 0):

- If the level corresponds to a move of the A player (an even-numbered level), we set the maximum of the children scores to their parent (since player A wants a high value as possible).
- If the level corresponds to a move of the B player (an odd-numbered level), we set the minimum of the children scores to their parent (since player B wants a low value as possible).

This process creates scores as in the figure above. When the recursion calculates all scores at level 1 (depth 1), the first level below the root, the final result is calculated – and the best move is chosen as the one that leads to a child with the optimal score.

Note: You may implement this algorithm in any way you please. Make sure you free all the allocated memory in the process.

Scoring Function

The scoring function goes over the board and outputs a score according to the discs on the board and the player.

Winning score for a board is 100, and losing score is -100.

If there is no winner, each disc is given a score – man gets 1 point and king gets 3 points.

For player A, we sum of the score for the disc of his color, and subtract the score of its opponent's discs.

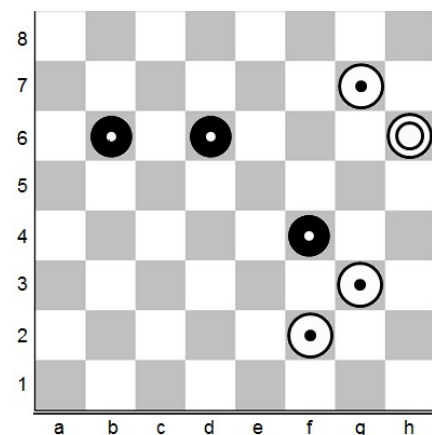
For example, the score of the following board for the white player is:

White discs: 3 man + 1 king = 6

Black discs: 3 man = 3

Total score: 6 – 3 = 3.

Notice that winning score is higher than possible discs sum (maximum is 60), guaranteeing that if a winning move exists, its score will top all other moves (same is for losing).



The scoring function should operate in $O(n)$, where n is the total size of the board.

Running Example

```
nova 52% ./Draughts
Welcome to Draughts!
Enter game settings:
clear
set <c,3> white m
set <c,5> black m
set <e,5> black m
print
|-----|
10| | | | | | | | | | |
|-----|
9| | | | | | | | | | |
|-----|
8| | | | | | | | | | |
|-----|
7| | | | | | | | | | |
|-----|
6| | | | | | | | | | |
|-----|
5| | | M | | M | | | | |
|-----|
4| | | | | | | | | | |
|-----|
3| | | m | | | | | | | |
|-----|
2| | | | | | | | | | |
|-----|
1| | | | | | | | | | |
|-----|
  a  b  c  d  e  f  g  h  i  j
start
Enter your move:
get moves
<c,3> to <d,4>
<c,3> to <b,4>
Enter your move:
move <c,3> to <b,4>
|-----|
10| | | | | | | | | | |
|-----|
9| | | | | | | | | | |
|-----|
8| | | | | | | | | | |
|-----|
7| | | | | | | | | | |
|-----|
6| | | | | | | | | | |
|-----|
5| | | M | | M | | | | |
|-----|
4| | m | | | | | | | | |
|-----|
3| | | | | | | | | | |
|-----|
2| | | | | | | | | | |
|-----|
1| | | | | | | | | | |
|-----|
  a  b  c  d  e  f  g  h  i  j
Computer: move <c,5> to <a,3>
|-----|
10| | | | | | | | | | |
|-----|
9| | | | | | | | | | |
|-----|
8| | | | | | | | | | |
|-----|
7| | | | | | | | | | |
|-----|
6| | | | | | | | | | |
|-----|
5| | | | | M | | | | |
|-----|
4| | | | | | | | | | |
|-----|
3| M | | | | | | | | | |
|-----|
2| | | | | | | | | | |
|-----|
1| | | | | | | | | | |
|-----|
  a  b  c  d  e  f  g  h  i  j
Black player wins!
nova 53% █
```

Notes:

1. You may assume that all input numbers are within the range of an unsigned int.
2. You may assume that the arguments for all command are always passed correctly, i.e. you have enough arguments and their order is as expected.
3. Whenever the user should type the player color (white, black) of disc type (man, king), you may assume the input is correct.
4. Whenever the user should type a position on the board, you may assume it will be formatted correctly, meaning <character, number>. You cannot assume that the position will be valid (the position <p,9> is formatted correctly, and you should print an error message in this case).
5. Failure of standard functions: in case a standard function fails (*malloc*, *scanf*, etc) an error message must be printed to the standard error (which can be done using *perror*, or using *fprintf* to *stderr*). The error message is "Error: standard function <function name> has failed\n". This is the **only** edge-case when you **must free** the memory allocated and exit the program.
6. In any stage, if the user tries to execute an illegal command, the following message is printed("Illegal command, please try again\n");
7. You are provided with the functions that print all possible error messages in the programs, as well as two functions to initialize and print the game board. You may change them as you please, just make sure you maintain the output format.
8. You are provided with a header file (Draughts.h) that contains all the messages that should be printed during the program. Use it to avoid typos and bad formatting.

Code Requirement:

1. You should use structs to implement entities in the program. Use a struct at least for:
 - Location in the board
 - Game move.
2. While generating possible moves for user, you must use a linked list. You can implement the move struct as a list node, or use another struct for list node and set the data type to be move.

Makefile

The **make all** command must build the **Draughts** executable and the *.o files.

you must create your own makefile with *gcc* commands.

Your makefile **should not** be generated by any automatic makefile program (this includes eclipse). You must use *-Wall -ansi -pedantic -errors* flags in the *gcc* commands.

The **make clean** command must remove all created files (using the *rm* command).

Submission guidelines

The submitted assignment must print all output in the specified format. Otherwise, it will not pass the automatic checks.

The assignment zip file includes a make file for compilation, some submission file examples and input/output examples. Questions regarding the assignment can be posted in the Moodle forum. Also, please follow the detailed submission guidelines carefully.

Moodle Submission:

This assignment will be submitted in Moodle.

You should submit a zip file named **username_ex3.zip**, username stands for you Moodle username.

The zipped file will contain.

- All your source files.
- The makefile you were provided with.
- partners.txt file according to the pattern provided in the exercise zip file. (do not change the pattern of this file).

Compilation

Your exercise should pass the GCC compilation test, which will be performed by running the "make all" command in a UNIX terminal window on nova.cs.tau.ac.il.

Check your code

Input/output examples are provided in the zip file. You should check your code by using the "diff -w" function (we use -w flag to ignore white spaces).

Code submission

Although the submission is in pairs, every student must submit it separately. The exercise files of both partners must be identical.

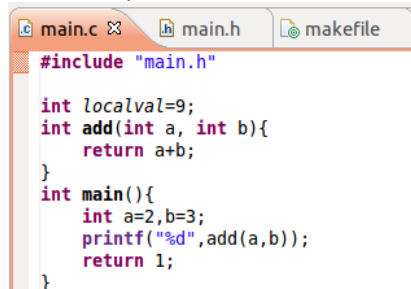
Good Luck!

Appendix- Header files

The header file is similar to the "*interface*" abstract type learned in JAVA course: it contains the ".c" file function prototypes, in addition to structures (such as linked-list) and external variables (that can be used in other files).

Note: usually, the main function prototype (i.e. "*int main()*") is not included in the header file.

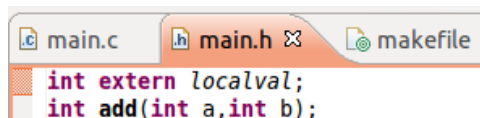
For example, assume that the following file "*main.c*" is implemented in the following file:



```
#include "main.h"

int localval=9;
int add(int a, int b){
    return a+b;
}
int main(){
    int a=2,b=3;
    printf("%d",add(a,b));
    return 1;
}
```

Then its header file, "*main.h*" might be in the following format:



```
int extern localval;
int add(int a,int b);
```

Using the `#include "main.h"` macro command in the beginning of *main.c* will make the contents of *main.h* be included in *main.c* after the preprocessor stage. That means that the *gcc* compiler will create a file that is equivalent to

```
int extern localval;
int add(int a, int b);

int localval=9;
int add(int a, int b){
    return a+b;
}
int main (){
    return 1;
}
```

There are many advantages using header files. For example, header files allow a programmer to use types and call functions defined in the header file without reading the implementation details. This improves readability of code, increases modularity of projects, and decreases compilation time.