

Software Project - fall 2014-15

Assignment 3

Due by 15.01.2014 23:55

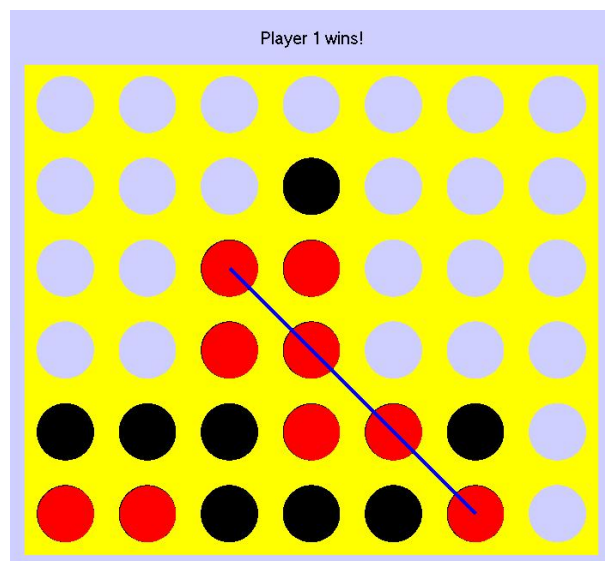
In this assignment, you are asked to write a C program simulating the Connect4 game. The program simulates a game between the user and the computer, where the computer uses a mini-max strategy, as illustrated below.

Provided are two header files – ListUtils.h and MiniMax.h which both contain prototypes which you will implement. An explanation regarding their contents will follow.

In the appendix you can read more about header files.

Background

Connect4 is a two-player game in which players take turns dropping colored discs from the top into a seven-column, six-row, vertically suspended grid. The pieces fall straight down, occupying the next available space within the column. The object of the game is to connect **four** of one's own discs of the same color next to each other vertically, horizontally or diagonally before your opponent.



The **mini-max** algorithm is a recursive algorithm for choosing the optimal strategy in an n -player, finite, zero-sum game, usually a two-player game. The algorithm operates on the **game tree**. The game tree for two players is a rooted tree whose nodes are states of the game and defined such that:

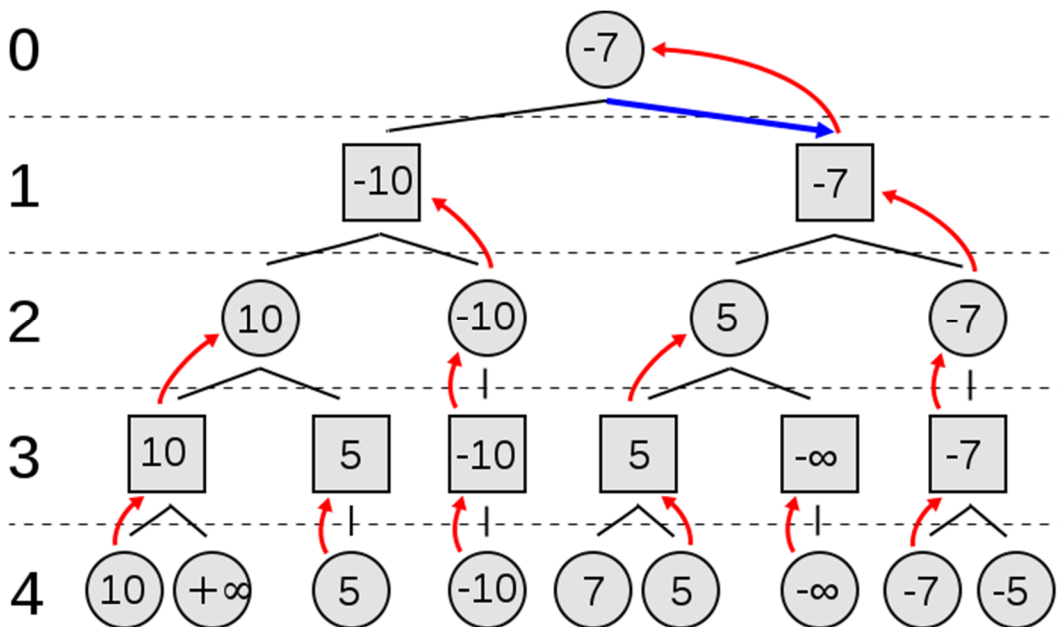
1. The root node represents the initial/current state of the game.
2. The children of each node v are the states that can be reached from v by a single move.
3. The leaves are the game's final states.
4. The tree levels with even depth are assumed to be player A's turns and the odd ones are assumed to be player B's turns.

5. Each Leaf has a value. Usually, a positive value means that player A has won and a negative value means that player B won. In this case we will call A the **max player** and player B the **min player**.

In theory, if player A can compute the whole game tree (which is finite) he can always figure out the optimal strategy using mini-max:

1. Calculate the *value* of each inner node according to its children:
 - a. If the node belongs to the max player (according to its depth) then the node's value is the maximum value out of its direct children.
 - b. If it is the min player's node then its value will be the minimum out of the children.
2. If we reach a leaf, then its value is the original value from the tree.
3. Now the optimal move for player A (assuming he is a max player) is to move from the root to the child which gives the highest value.

The logic behind this algorithm is that it assumes that the **max player** will always try to ensure himself the **highest** possible value and the **min player** will always try to ensure the **lowest** possible value. This is a **deterministic** algorithm for finding the optimal (not necessarily winning) strategy.



In practice (and in this exercise), the full game tree of a real game is often too large to compute/traverse. What we will do instead is use a **bound depth** when traversing the tree in order to limit the computation. We will adjust the algorithm such that when the maximal depth is reached, the value of the inner node will be **evaluated** by some scoring function rather than being fully computed.

This implementation of mini-max provides us with a **heuristic** solution – our strategy is not guaranteed to be optimal (unless the evaluation function always gives us the correct value), but we “hope” it will be close enough in order to defeat our opponent.

More information and explanations about mini-max and game trees will be provided in class.

When executing Connect4, the program starts a new game and waits for the user's input. The program operates by reading commands from the user – and the game is played according to it.

Each line starts and ends with a pipe '|' character. In between are 7 characters according to the state of the Connect4 board – space character if there's no disk in it, an 'O' character for the user discs, and 'X' character for the computer discs. Between each of these characters is a space ' ' character, including between the game board's edge (the pipe) on each side – for a total of 17 characters in each line - 2 pipe characters, 8 spaces, 7 disc characters (which might themselves be spaces).

The bottom (8th) line contains the column numbers – 1 through 7 in the columns corresponding to disc positions, with the other 8 characters for the line being spaces. Here are 2 examples – the beginning of a new game (left), and end of a game (right):

After printing the board, the program waits for a command from the user. If the user added a disc, the move is performed - and the program checks if the user had won. If he did, it prints the final board and outputs "Game over: you win.\n", otherwise it makes the move for the computer's turn by executing the mini-max algorithm, choosing his move accordingly and making it. After the computer's turn, the board is presented again to the user. If the computer won the program outputs: "Game over: computer wins.\n". In any case, the program then waits for a new command.

The program repeatedly gets commands from the user. A formal definition of a command follows:

- For receiving the user commands use `fgets(str, sizeof(str), stdin)`, where `str` is a pre-allocated `char *` string.

The following commands are handled by the program:

set_number_steps < num_steps >

This command sets the number of steps to look ahead in the mini-max algorithm (i.e. the max depth of the game tree traversal). A step represents a single turn (by a **single** player, not a pair of steps), starting with player A. Depth k means $\left\lfloor \frac{k}{2} \right\rfloor$ steps for player A, and $\left\lfloor \frac{k}{2} \right\rfloor$ steps for player B (see mini-max algorithm, below, for players A and B).

This command must be entered by the user before 'suggest_move' or 'add_disc' are entered, whenever the game starts (or restarts), but can be used later as well. Must be done in $O(1)$ time.

Note: The maximum number of steps allowed is 7.

suggest_move

This command executes the mini-max algorithm for the **user**, and calculates the best move according to it and suggests it to the user. After executing the mini-max algorithm (described below), the program prints the following message to the user: "Suggested move: add disc to column [i].\n", where i is the suggested column to place the disc in, ranging from 1 to 7.

Note: This command doesn't perform any move.

add_disc < col_num >

This command executes the user turn by placing a disc in the specified column. Immediately after, the computer turn is played by executing the mini-max algorithm for the computer, and adding a disc to the column suggested by it. The program then prints the following message to the user: "Computer move: add disc to column [i].\n". The program then prints the game board to the user (as explained above).

quit

This command frees the memory allocated for the game and exits the program with exit code 0.

restart_game

This command restarts the game by clearing the board and starting a new game, starting with the user's turn. The program outputs the message "Game restarted.\n" and then prints a blank board for the new game (the same as when starting the program).

Mini-max algorithm

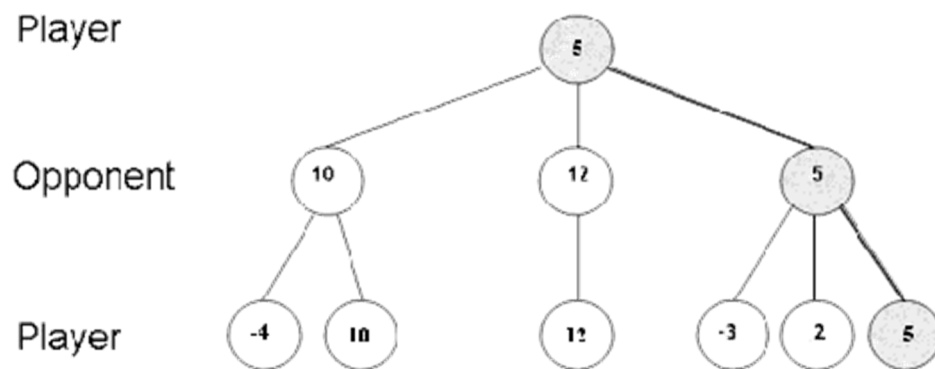
In this exercise, you will implement a **generic version** of mini-max, defined by the header file MiniMax.h which contains a prototype for a single function named **getBestChild**. The same definition will also be used in the final project, although the implementation will be slightly different. You will implement this function in a source file named MiniMax.c which will not depend on any other source or header file, except MiniMax.h and the standard C libraries.

Note: Since dependencies in C are transitive and because MiniMax.h includes ListUtils.h, your code will depend also on ListUtils.h and its implementation.

The game tree will **not** be computed before mini-max is executed – the children of each node will be computed on-the-fly whenever it is reached, considering the supplied 'maxDepth' parameter.

Since the function is generic, it takes several function pointers as input in order to do the game specific logic: evaluate state scores, compute each node's children, de-allocate memory, etc... The representation of each state is left up to the caller of `getBestChild` – it is passed as an opaque `void*` pointer. See the header file's documentation for full details.

In the exercise zip file you will also find `MiniMaxDemo.c` – a simple program which demonstrates the usage of the mini-max algorithm. The implemented example uses the same game tree described in the following image:



In the figure above, you can see an example of a game tree with depth 2 – the values at the leaves are given and the values at the inner nodes are calculated by the mini-max algorithm. Note that in this example, player A is min player.

Note: The demo program will work only when compiled with your implementations of `MiniMax.c` and `ListUtils.c`.

You will also implement a specific usage of the mini-max algorithm for the connect4 game. For that, you will need to define a representation for the game state, an evaluation/scoring function for states and a function for computing possible child states from a given state.

In general, each state will have up to 7 children (less if one of the columns is full or none if the game is over). For example, in the beginning of the game, the root of the game tree is the blank board and it has seven children, each having seven children of their own – so at depth 2 we already have 49 “leaves” and a total of 57 nodes in the tree. Note that since we allocate and de-allocate child nodes on-the-fly, the whole game tree is never kept entirely in memory, even if we traverse it all.

You may include whatever information you need in the state representation, e.g. what disc appears in each slot or who's turn it is to play. The state's score should not be computed in advance, only by the evaluation function or according to child nodes.

Important note: For the sake of consistency, the children of each state should be ordered increasingly by their column number. Also, the mini-max implementation should always prefer a child with a smaller index, in case of ties: i.e. if $V(child_i) = V(child_j)$ and $i < j$ then $child_i$ should be preferred.

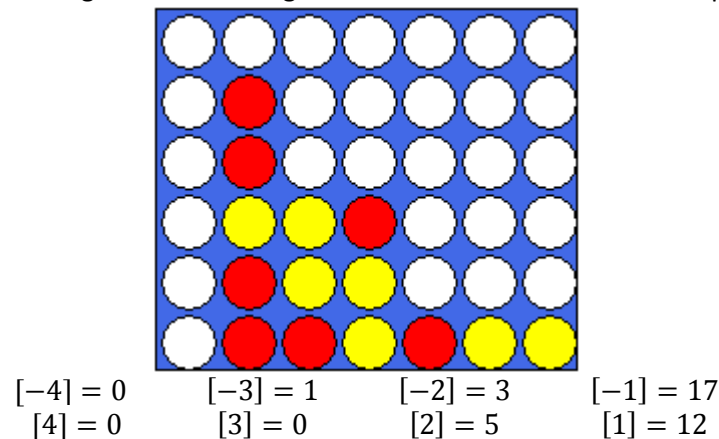
Evaluation Function

The scoring function goes over the board and outputs a score according to the spans of discs it finds. To find the score, we go over all possible spans of 4 discs on the board (a “span” is

any vertical, horizontal or diagonal sequence of 4 slots which may or may not contain discs). For each span, we count the number of discs it contains for each player. We count player A's discs as positive: +1, and player B's discs as negative: -1. This process gives us, for each span, a score number between -4 and +4.

We then aggregate the results in a table going from -4 to +4. Each cell of the table counts the number of times we encountered a span of that size. We can ignore spans of size 0.

For example, here is a game board along with its score table where the **red** player is A:



Once we have this table calculated according to all possible spans, we can calculate the board's score:

- If there's any +4 or -4 score, that means a player has won – we can immediately return a maximum/minimum evaluation according to it: 100,000 or -100,000.
- In any other case, we'll calculate the score according to a constant weight vector: Our weight vector will be defined as $w = (-5, -2, -1, 1, 2, 5)$. Let $v = ([-3], [-2], [-1], [1], [2], [3])$ be the vector derived from our table then the scoring function gives $f(v) = \langle v, w \rangle = \sum_{i=1}^6 v_i w_i$

For the above board the score is:

$$1 \cdot (-5) + 3 \cdot (-2) + 17 \cdot (-1) + 12 \cdot (1) + 5 \cdot (2) + 0 \cdot (5) = -6$$

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

Hint: The scoring function can be used both to calculate the score and to determine the winner (if any).

Data structures

Since in this mini-max implementation the graph need not be computed in advance, you will only have to implement a list data structure for this exercise. It will be a **linked list** whose fundamental functions are defined in the header file ListUtils.h. In the header file you will see prototypes for the **List** structure and various functions which operate on this abstract structure such as: append, tail, destroy, etc...

The list representation is left totally up to you. Your only requirements are that it would be a reasonable and efficient implementation of a linked list that obeys the documentation in the header file.

Note that the list structure defined by the functions in the header file is non-modifiable in the sense that you can only add elements to the list, but cannot insert an element into a specific location, remove or rearrange the elements.

You will implement the List structure and functions in a source file named ListUtils.c which will not depend on any other source or header file, except ListUtils.h and the standard C libraries.

In the exercise zip file you will also find ListUtilsDemo.c – a simple program which demonstrates the usage of various list functions. This program accepts a single command line argument – an integer denoting the number of elements (integers) to place in the list. It then creates a list with the given number of elements, prints out it's content to the screen and finally de-allocates the used heap memory.

Note: The demo program will work only when compiled with your implementation of ListUtils.c.

Running Example

```
nova 112% ListUtilsDemo 10
0 1 2 3 4 5 6 7 8 9
nova 113% MiniMaxDemo
The best child is at position 2 with value 5.
nova 114% Connect4

| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
-----
  1 2 3 4 5 6 7
add_disc 1
Error: first command must be set_number_steps.
set_number_steps 7
suggest_move
Suggested move: add disc to column [4].
add_disc 4
Computer move: add disc to column [4].

| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
-----
  1 2 3 4 5 6 7
add_disc 3
Computer move: add disc to column [5].

| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
-----
  1 2 3 4 5 6 7
set_number_steps 0
Error: number of mini-max steps must be between 1-7.
set_number_steps 1

Error: command must have at least one non-whitespace character.
quit
nova 115% █
```

Error Handling

You can assume that for an integer parameter (column number, number of steps) the user gives a non-negative integer less than $2^{32} - 1$. Also, you can assume that the user writes commands with the exact number of parameters specified in this document.

In the following cases **except number 1**, your program must print to the **standard output** (which can be done by *printf*), continue, and not exit. The program must print only **one** error message, even if more have occurred. Print the error with the smaller serial number (i.e., prefer 1 to 2, 3, etc.).

1. You must check if a standard function fails (*malloc*, *scanf*, etc). In this case, an error message must be printed.
The error message must be printed to the **standard error** (which can be done using *perror*). 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 with a code different from 0.

Note: you do not need to check if *perror* or *printf* succeeded.
2. The program must check that every command has a limited number of characters. Otherwise, the program cannot store the command string into a string variable without causing an overflow buffer error. Thus, the program checks that user command is less than or equal to 40 characters (including the command whitespaces, excluding '\n'). Otherwise, the program prints to the **standard output** (can be done using *printf*): `"Error: command length must be less than or equal to 40 characters.\n"`.
3. If the user enters an empty line command (i.e. it contains only whitespace characters, besides the new line character) - the program prints to the standard input `"Error: command must have at least one non-whitespace character.\n"`.
4. The user's first command must be *set_number_steps*. Otherwise, the program prints `"Error: first command must be set_number_steps.\n"`.
5. In *set_number_steps* < *num_steps* >: If the given number of steps is not between 1 and 7, the program prints `"Error: number of mini-max steps must be between 1-7.\n"`.
6. In *add_disc* < *col_num* >: If the given column number is not between 1 and 7, the program prints `"Error: column number must be in the range 1-7.\n"`.
7. In *add_disc* < *col_num* >: If the given column number refers to a column that's full (already contains 6 discs), the program prints `"Error: column [col_num] is full.\n"`, where 'col_num' is the given column number.
8. If, after any turn, the board is full and the game isn't over (there's no winner), the program prints the game board and then prints `"Game over: board is full and there is no winner.\n"`. From this point on, the program should behave as if the game is over.
9. You must check that the user writes a command only from the above command list. Otherwise the program prints `"Error: command [given_command] not supported.\n"`, where 'given_command' is the first sequence of characters until the first whitespace character.
10. In case the game is over (the user or the computer won on the last move, or the board is full), the only commands allowed are **restart_game** or **quit**. In case the user enters any other command, the program prints `"Error: game is over.\n"`.

Make File

In this exercise you will have to create your own make file using gcc commands. You may not use any program (e.g. Eclipse or Visual Studio) other than a text editor to create the make file. Your make file will be based on the way you choose to divide your source and header files. However, you must meet the following requirements:

1. Every gcc command must include the flags:
`-Wall -g -std=c99 -pedantic-errors`
2. The following targets must be included in the make file:
 - Connect4 – to create the Connect4 executable.
 - ListUtilsDemo – to create the ListUtilsDemo executable.
 - MiniMaxDemo – to create the MiniMaxDemo executable.
 - ListUtils.o – to create the ListUtils.o file from ListUtils.c.
 - MiniMax.o – to create the MiniMax.o file from MiniMax.c.
 - all – to create all the above.
 - clean – to remove all the files that may be generated by **make** from the file system, using the 'rm' command.
3. Your make file must be human readable and concise (use macros).
4. Make sure that all the dependencies are well-defined.

Important Notes

1. You may add more source and header files as you see fit, but, the main function for Connect4 must be in a source file named Connect4.c.
2. Your code should be divided into modules (i.e. separate source and header files), in a reasonable way.
3. Use *valgrind* to detect memory errors and leaks in your program. We shall do the same.
4. You should do your best to avoid code duplication.
5. Beware of magic numbers (e.g. 7).

Submission guidelines

**The assignment must be submitted in the specified format.
Otherwise, it will not pass the automatic tests.**

The assignment zip file includes the header files ListUtils.h and MiniMax.h, the demo source files ListUtilsDemo.c and MiniMaxDemo.c, an example of the Partners.txt file 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 only in Moodle**.

You should submit a tar file named ***ex3.tar***.

The tar file will contain the following files:

- ListUtils.h – as given in the zip file.
- MiniMax.h – as given in the zip file.
- ListUtilsDemo.c – as given in the zip file.
- MiniMaxDemo.c – as given in the zip file.
- ListUtils.c – should contain the implementations for the prototypes declared in ListUtils.h.

- MiniMax.c – should contain the implementations for the prototypes declared in MiniMax.h.
- Connect4.c – must contain the Connect4 program main function.
- Any other source or header files you may have written.
- Your make file.
- Partners.txt file according to the following pattern:
 <first username>
 <first id number>
 <second username>
 <second id number>

An example of **Partners.txt** is provided in the zip file.

If you have all the files ready in the same directory, execute the following command from this directory to create the submission file:

```
tar cvf ex3.tar <list_of_files>
```

If you want to verify your tar was created properly, you can extract the files using:

```
tar xvf ex3.tar
```

After the files are extracted you should compile and test your code again.

Compilation

Your exercise **must pass** the GCC compilation test, which will be performed by running the "make all" command in a UNIX terminal window.

Check your code

Input/output examples are provided in the zip file. You should check your code by using the "diff" function.

Code submission

Only one of the partners should submit the exercise via Moodle. In case both partners submit, one of the submissions will be arbitrarily selected and graded – **this cannot be appealed.**

Submission Penalty

In rare cases where automatic testing fails (giving a grade of zero) due to a *submission error* and the student provides a *simple workaround*, the submission will be graded again with an automatic penalty of 15 points – **this cannot be appealed.**

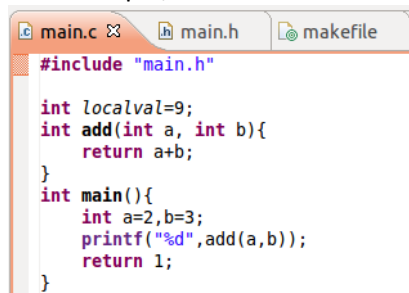
Good Luck!

Appendix: Header files

The header file is similar to the *"interface"* abstract type that exists in the Java language: it does not contain implementations but may contain function prototypes, structures, external variables (that can be used in other files) and other kinds of declarations.

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

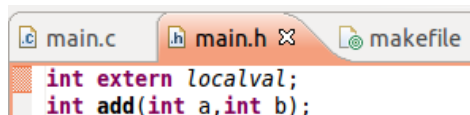
For example, assume that the following file *"main.c"* is implemented in the following file:



```
#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 copied 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 to using header files. For example, header files allow a programmer to use types and call functions defined in the header file without knowing the implementation details. This improves readability of code, increases modularity of projects, and decreases compilation time.