Interface for Artificial Intelligence Project “gobang”

1) Description

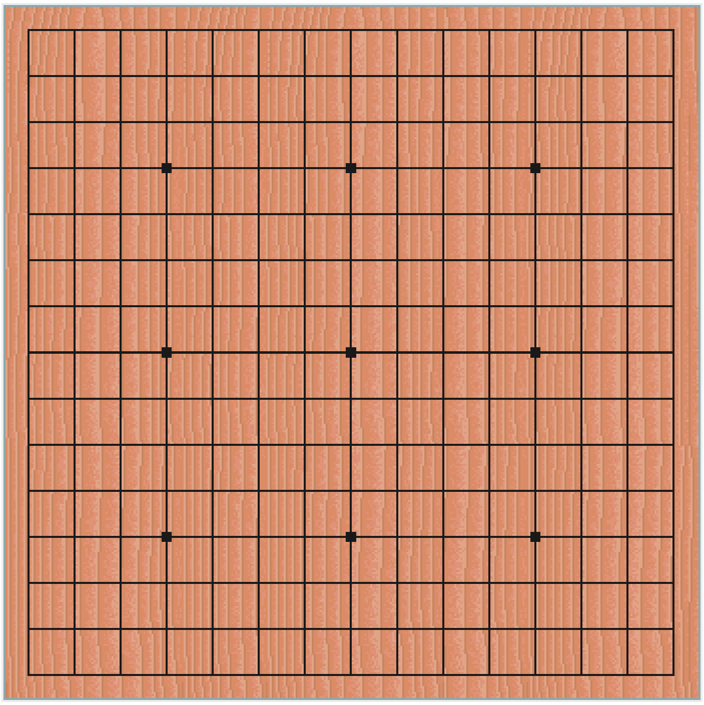
Implement a program “gobang” with artificial intelligence. The program provide two major mode: man-vs-machine and machine-vs-machine. No human action needed when the program runs in mode machine-vs-machine. When it is in mode machine-vs-machine, the program must connect to a server called “judge”. All steps will be recorded in a log, keeping as evidence. You can choose any program language to develop this program.

The definition of interface in this document is only for the mode machine-vs-machine.

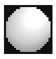
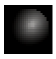
2) Criterion

1 UI

The UI for the chessboard is defined as follow:



And the chess can be:

White chess:  Black chess: 

All the chesses are put on the cross point of the row and col lines. The total numbers of rows and cols are both 15. The index of row and col is begin from 0 and end with 14. So (0, 0) for top-left and (14, 14) for bottom-down.

2 Communication between processes

We use socket to communicate between processes. The program ‘gobang’ can use socket to connect to the server ‘Judge’ when in mode machine-vs-machine. We use the network port 9527. A ‘Judge’ can only receive two gobang’s connection.

When testing the program, both client and server will be in the same machine.

So please set the default remote ip address value to ‘127.0.0.1’.

3 Definition for communicatoin interfaces

The following commands must be realized for the client and server.

All the commands are sent by byte(8 bits)

```
//define commands for the socket communication
#define   COMM_MSG_DONOTHING          0    //do nothing, ignore it!
#define   COMM_MSG_REJECTED           1    //last request is rejected
#define   COMM_MSG_ACCEPT             2    //last request is accepted
#define   COMM_MSG_FIRST              3    //go first when the game starts
#define   COMM_MSG_SECOND             4    //go second when the game starts
#define   COMM_MSG_GAME_REQUIRE_START 5    //request for game start
#define   COMM_MSG_GAME_START         6    //game start
#define   COMM_MSG_CHESS              7    //information for chess
#define   COMM_MSG_TIMEOUT            8    //time out when one peer takes too many time in a bout
#define   COMM_MSG_WIN                9    //win
#define   COMM_MSG_LOSE               10   //lose
#define   COMM_MSG_DRAW               11   //draw
```

The value of the chess color:

```
//define color for the chess
enum chess_color{
    Chess_Clr_Black = 0,
    Chess_Clr_White = 1,
}
```

The details for the commands:

- COMM_MSG_DONOTHING

Server or client can receive this command. Please ignore this command when received.

- COMM_MSG_REJECTED

Client can receive this command. When the server received a command that is an invalid value or a value that provides invalid operation, the server will send the client this command. 4 byte data will follow this command, the data is descripted as follows(byte stands for 8 bits):

```
//define reject_info for COMM_MSG_REJECTED
struct reject_info{
    byte reason;      //can be REJECT_REASON_UNWANTED or REJECT_REASON_INVALID_CHESS
    byte solution;    //can be SOLUTION_NOACTION or SOLUTION_ACTION_REPEAT
    byte parameter;   //can be one of the command begin with COMM_MSG_
    byte reserved;    //not used
};
```

The values of the reason can be REJECT_REASON_UNWANTED or REJECT_REASON_INVALID_CHESS . The details are listed bellow:

```
//type of reject reasons
//unwanted command, please solve this by reference to field solution of reject_info
```

```
#define REJECT_REASON_UNWANTED          1
//the position of the chess is invalid, e.g. a taken position, position outside the chessboard,
forbidden step for first-go player
#define REJECT_REASON_INVALID_CHESS     2
```

The values of the solution can be SOLUTION_NOACTION or SOLUTION_ACTION_REPEAT. The details are listed bellow:

```
//type of reject reason solutions
#define SOLUTION_NOACTION            0    //no action needed
#define SOLUTION_ACTION_REPEAT       1    //please repeat this action with required parameters
```

The values of the paramter can be values start with COMM_MSG_. The possible values are listed bellow:

```
//define commands for the socket communication
#define  COMM_MSG_DONOTHING          0    //do nothing, ignore it!
#define  COMM_MSG_FIRST              3    //go first when the game starts
#define  COMM_MSG_SECOND             4    //go second when the game starts
#define  COMM_MSG_GAME_REQUIRE_START 5    //request for game start
#define  COMM_MSG_CHESS              7    //information for chess
```

The field reserved is not used.

- COMM_MSG_ACCEPTED

Client can receive this command. When the server received a valid command, the server will send the client this command.

- COMM_MSG_FIRST

Server or client can receive this command. When it is received by server, it will send COMM_MSG_ACCEPTED to the source client and set this client go first when the game starts. And send the other client a command COMM_MSG_SECOND. When it is received by client, the client set itself go first when the game starts. The client that connects to server as the first one will be setted with the property 'go first' by default.

- COMM_MSG_SECOND

Server or client can receive this command. When it is received by server, it will send COMM_MSG_ACCEPTED to the source client and set this client go second when the game starts. And send the other client a command COMM_MSG_FIRST. When it is received by client, the client set itself go second when the game starts. The client that connects to server as the second one will be setted with the property 'go second' by default.

- COMM_MSG_GAME_REQUIRE_START

Server can receive this command. When there are two clients connect to the server and the game is not begun or is already finished, this request will be accepted., then the client will receive a message COMM_MSG_ACCEPTED. Otherwise the client will receive a message COMM_MSG_REJECTED. When the server send COMM_MSG_ACCEPTED, it will send COMM_MSG_GAME_START to both clients.

- COMM_MSG_GAME_START

Client can receive this command. When the client received this command, it must set the chessboard to initial state. The First-go client then send command

COMM_MSG_CHESS to the server for the first step.

- COMM_MSG_CHESS

Both server and client can receive this command. This command is followed by 8 byte data. The data is as follows:

```
//define the information for each step
struct chess_info{
    byte index;         //index for the chess, the client do not need to fill this field
    byte color;         //color of the chess
    byte row;           //row of the chess
    byte col;           //col of the chess
    byte reserved[4];   //reserved, just ignore it
};
```

The server received this command and 8 bytes data, process it, and send COMM_MSG_ACCEPTED to client if the data is correct. Then this command and data will be sent to both client. Otherwise the source client will receive COMM_MSG_REJECTED. Please resend correct command and 8 byte data to the server when this happens.

- COMM_MSG_TIMEOUT

Client can receive this command. It is used to restrict the response time of the program. When the client do not response to server for a long time, the client will receive this command and receive command COMM_MSG_LOSE as will. And the other client will receive command COMM_MSG_WIN. Game is over.

The default value for TIMEOUT is 10 seconds.

- COMM_MSG_WIN

Client can receive this command. When client received this command, the other client will receive command COMM_MSG_LOSE. Game is over.

- COMM_MSG_LOSE

Client can receive this command. When client received this command, the other client will receive command COMM_MSG_WIN. Game is over.

- COMM_MSG_DRAW

Client can receive this command. When client received this command, the other client will receive command COMM_MSG_DRAW. Game is over. Result is draw.

Please implement these commands strictly when in mode machine-vs-machine.

4 Log

As a server, the 'Judge' will record every step for the chess by using log. The format of the log is as follows:

```
//format of the log file
log_chess     index1    color1    row1     col1     time1
log_chess     index2    color2    row2     col2     time2
…
log_chess     indexN    colorN    rowN     colN     timeN
log_result    reasons
```

And the values for log_chess and log_result are:

```
//definition for log type
enum log_type{
    log_chess      = 0, //type for chess informaiton
    log_result     = 1, //type for result, an reason will follow this field
};
```

When game is over, this log will be generated, keeping as evidence.