1. 実験の目的

事前学習で TCP/IP や構造体、ポインタ、動的メモリ、make コマンドなどの基礎知識を学んで実践に活かす。実践ではネットワーク対戦型リバーシを作成することを通して、Socket を利用した TCP/IP プログラミングを実装する。

2. 原理

[TCP/IP について、TCP と UDP のちがい]

TCP/IP とはネットワーク上でコンピュータ同士が通信するときに使われる手順や約束事（プロトコル）の一種である。インターネット技術特別調査委員会（IETF）によって技術仕様がまとめられた RFC に基づいて規定されている。ネットワークアーキテクチャを国際的に標準化した OSI 基本参照モデルの各層と対応している。OSI 基本参照モデルは下位層から順に物理層、データリンク層、ネットワーク層、トランスポート層、セッション層、プレゼンテーション層、アプリケーション層と標準化されている。TCP/IP はこの中で下位層の 4 つが共通していて、上位層の 3 つはアプリケーション層にまとめられている。

TCP(Transmission Control Protocol)は信頼性を重視していてアプリケーション層の HTTP, FTP, TELNET, SMTP, POP3 などに利用される。一方 UDP（User Datagram Protocol）は信頼性よりもリアルタイム性を重視していて DNS, DHCP, NTP などに利用される。これは信頼性を担保しようとするとデータのチェックに時間がかかることによって信頼性とリアルタイム性の両立が不可能なので用途に合わせて使い分けられているということである。[2]

[構造体]

構造体とはプログラミング言語における変数の型の一つであり、任意の数と種類の変数を集めて管理するための変数である。以下に C 言語で構造体を使う際の形式を示す。

以下の形式で構造体を宣言する。

```
struct 構造体の名前{
    型 1 変数名 1;
    型 2 変数名 2;
    型 n 変数名 n;
}
```

構造体の変数を作成するときは main 関数の中で以下のように宣言する。

```
struct 構造体の名前 構造体変数名 = {値 1, 値 2, …, 値 n};  //  初期化
```

変数を使うときは以下のようにする。

```
構造体変数名.変数名 1 = 値 1;
構造体変数名.変数名 2 = 値 2;
構造体変数名.変数名 n = 値 n;
```

[ポインタ、関数ポインタ]

ポインタとは変数のアドレスを格納する変数である。アドレスとは変数のメモリ上に割り当てられる番地のことである。C 言語では変数名の先頭に「*」をつけることで宣言した変数がポインタであることを示す。配列の場合はインデックス（[n]）を省略することで配列の先頭のポインタを表すこともできる。またポインタではない変数のアドレスを取得する際には変数名の先頭に「&」をつける。int 型の整数 num のアドレスを pnum に入れるときには以下のようにする。

```
int num;      //  整数型の宣言
int *pnum;    //  整数型のポインタの宣言
pnum = &num;  //  変数のアドレスを代入
char str[10];     //  char 型の配列の宣言
char *pstr1;      //  char 型のポインタの宣言
char *pstr2;
pstr1 = &str[0];  //  アドレスの代入
pstr2 = str;      //  上の行と同じ意味
```

関数ポインタとは関数が格納されたアドレスのことである。「関数名(引数);」と同じ意味で「関数ポインタ名(引数);」と書くことができる。関数ポインタの中に異なるアドレスを入れることによって 1 つの関数ポインタによって複数の関数を用いることができる。関数ポインタは「返り値の型 (*func)(引数 1 の型, 引数 2 の型);」と宣言して「func *名前;」と初期化をして「名前 = &関数名;」と代入して使うことができる。以下に 2 つの関数を 1 つの関数ポインタで使用する例を示す。

```
int (*func) (int, int);  //  関数ポインタの型宣言
int add(int a, int b){ return a + b; }  //  関数 1
int times(int a, int b){ return a * b; }  //  関数 2
int main(){
    int num1 = 2; int num2 = 3; int ans;
    func *pfunc;   //  関数ポインタの初期化
    pfunc = &add;  //  関数 1 のアドレスを代入
    ans = pfunc(num1, num2);  //  5  //  関数 1 を使った
    pfunc = &times;  //  関数 2 のアドレスを代入
    ans = pfunc(num1, num2);  //  6  //  関数 2 を使った
}
```

[malloc、free]

malloc とは構造体のメモリを動的に確保するために用いる関数である。動的にメモリを扱ったり確保したメモリ領域について関数を使って操作したりする際に用いる。確保したメ

モリは解放しない限りメモリを使ってしまうので使い終わった後に free 関数を用いて必ず解放する必要がある。
C 言語においてはヘッダーファイル「stdlib.h」内に宣言されているので include する必要がある。main 関数の外に以下のように宣言する。

```
#include <stdlib.h>
```

構造体を動的メモリ確保によって使用する場合は以下の例のようにする。以下の実装では malloc 関数を用いて構造体変数 s のサイズ文だけメモリを確保してポインタにキャストしている。使った後には free 関数を用いて使った動的メモリの解放をしている。

```
struct s{
    int num;    //  整数
    char *str;  //  文字列
}
int main(){
    s *entity;  //  ポインタ型の変数
    entity = (s*)malloc(sizeof(s));  //  動的メモリの確保
    entity->num = 0;  //  変数の初期化
    entity->str = (char*)malloc(sizeof(char) * 32);  //  変数の初期化
    //  処理
    free(entity->num);  //  メモリの解放
    free(entity);       //  メモリの解放
    return 0;
}
```

[make、Makefile]
make コマンドとは Linux 上でコンパイルのルールを記述した Makefile を用いて、プログラムのビルド作業を自動化するツールである。コマンドを自由に作ることができるという利点がある。ビルドするプログラムと同じ階層に「Makefile」という名前のファイルを作って任意の変数や依存関係などのルールを書くことによって Makefile を作ることができる。以下に具体的な設定の仕方を示す。

① Makefile の変数の設定

Makefile の中で用いる変数の設定は「変数名 = 定義」とする。変数名は慣例的に大文字を使う。

② Makefile 内のルールの設定

・任意に決められるコマンド名に:をつけて改行してタブを使ってからルールの内容を書く。
・変数を呼び出す際には「$(変数名)」とする
・make コマンドを実行するとルール内容まで表示されるが、先頭に「@」をつけることで非

表示にすることができる
・必要なファイルがあるときは「コマンド名:ファイル名」とする
・「コマンド名：コマンド1 コマンド2 … コマンドn」とすることで複数のコマンドを連続して呼び出すことができる。

③ make の使い方

使う際にはMakefileを定義したパスと同じ場所で「make」とすることで自分で定義したルールを適応することができる。[3]

3. どのタスクまで行ったか

記載されているタスクは全て実施した。AI に関してはマス目ごとの評価点で次の手を決めるのと候補数（後ほど説明）を評価点に次の手を決める2つのプログラムを作った。以下にタスクの内容とファイル名の関係を示す。

タスク2：task2_client.c, player.c
タスク3：task3_server.c
タスク4 マルチスレッド：task4_1_server_pthread.c
タスク4 DOS攻撃対策：task4_2_server_pass.c
タスク4 マス目AI：task4_3_server_AI1.c
タスク4 候補数AI：task4_3_server_AI2.c

4. 実装方法（アイデア・設計）

【タスク2】

① player.c ファイルの変更点

〇connect_to 関数

・echo_client.cからコピーした。サーバとの接続をするために用いる。必要なヘッダーファイルをincludeした。

〇remote_play 関数

・サーバにMOVEリクエストを送り、mpに次の手を入れるための関数
・サーバには”MOVE 盤面(64個) 手番¥r¥n”と送るのでbufには73バイト分確保している
・サーバからは”D3¥r¥n”のように次の手が返ってくるのでbuf_returnには4バイト分確保する配列を用意している
・盤面と手番の色の状態を見て、置けるところがあるならサーバにMOVEリクエストを送る。現在の手番の置けるところがないかつ次の手番の置けるところがあるときにはパスを選択する。
・write_boardをしてbufの中に先頭から64バイト分に盤面を入れる。その後、後ろに5つずらし、先頭に”MOVE ”をつける。その後ろには手番がWHITEのときは”0”、BLACKの

ときは”X”をつける。最後には”¥r¥n”をつける。
・fwrite でサーバに buf に入れた MOVE リクエストを送る
・fread でサーバから返ってきた次の手を buf_return に入れる
・read_move で次の手を mp に格納する

〇make_remote_player 関数
・相手プレイヤーの次の手を考える際にサーバに接続するための関数
・_hidden_state には remort_play 関数で用いるためにファイルのポインタを入れる
・_play には remote_play のポインタを入れる
・_clean_up にはファイルを閉じてサーバに QUIT リクエストを送る myfclose_quit 関数のポインタを入れる

② task2_client ファイルの作成（single_play.c との変更点）
〇main 関数
・変数「is_net」を作り、相手プレイヤーをサーバに接続するかどうかのフラグとする。オプション引数の中に「-c」があるときに is_net = 1 とする。このときは make_remote_player 関数を呼び出す。引数はホスト名とポート番号としている。

【タスク 3】
・echo_server.c からコピーして serve 関数を変更する
〇serve 関数
・73 バイト分の buf を用意してクライアントからのリクエストを受け取る。受け取ったリクエストをターミナルに表示するために 71 バイト分の hyouzi に格納する。¥r と¥n の分だけ 2 バイト少なくしている。
・MOVE リクエスト、NOPE リクエスト、QUIT リクエストを受け取ったときでそれぞれ分岐させる。
・NOPE リクエストのときはクライアントに「OK¥r¥n」と返す
・QUIT リクエストのときはクライアントには何も返さずに while ループを抜ける
・MOVE リクエストのときは合法手を探す legal_moves を使うために board、color を受け取ったリクエストから取り出す。合法手は最大 64 手あるのでその分の大きさの配列 buf_return と何手あるかを格納する len を定義する。合法手の中からランダムに 1 つ手を選び、後ろに¥r¥n を付けてクライアントに返す。

【タスク 4】
〇サーバのマルチスレッド化
・echo_server_pthread.c からコピーして serve 関数をタスク 3 で作った serve 関数に置き

換えることによって実装した

〇攻撃対策
・バッファオーバーフローに備えるために echo_server と同じように serve 関数の最初に setbuf 関数の第 2 引数に NULL を指定してバッファリングオフを実装した。また、開いたファイルは関数の最後に必ず閉じるようにしている。
・DOS 攻撃に備えるためにパスワード認証を設ける。クライアント側で-p を引数に指定することでパスワード認証に対応した関数を呼び出すようにした。実装を変更したところについては以下に示す。クライアントからサーバにアクセスがあった時点でサーバでランダムにパスワードを生成してサーバに表示する。クライアント側でパスワードの入力を促す。入力されたパスワードをサーバに送り正しいパスワードかどうか確認する。一致しない場合はクライアントとの接続を終了して_play の中に-1 を入れてエラー終了させて、クライアント側の main 関数の中で game 関数の返り値を-2 にする。このときは Game finished のメッセージや終了時の盤面を表示しないようにする。

〇サーバの手を AI で強化する
・リバーシの置く場所それぞれに評価点をつけて評価点が最も高いところを選択するプログラムを作った。serve 関数の中で以前は可能な手の中からランダムに選んでいたところを可能な手の中から評価点が最も高い手を選択する best_choice 関数を作って変更した。best_choice 関数では現在の盤面のみ見て、その中で元々設定したマス目ごとの評価点の中で最も高いところを選択させている。評価点は以下の論文を参考にした。[4]
・次打てる場所の数を候補数と名付けると、一般的にリバーシにおいては自分の候補数が多くて相手の候補数が少なくなるようにすれば強いと言われている。現在の盤面の中で「自分の候補数 - 相手の候補数」が最も大きくなる手を選ぶようにした。具体的には most_kouho 関数の中で現在可能な手の中から最善手を選ぶプログラムを書いた。この関数の中で現在の盤面から次の手を 1 つずつ打ってそのときの自分の色の候補数から相手の色の候補数を引いた数を評価点として一番評価点が高くなるときの手を採用するようにした。

5. 工夫点・改善点

〇工夫点
・1 回目の実験日では仕様や書き方があまり理解できずにタスク 2 の序盤で終了してしまったが、2 回目の実験日までに自分では解決しなくなるまで進めた。タスク 2 とタスク 3 のクライアントとサーバ間の実装をして、1 往復はできるようになった。しかし、サーバからクライアントの通信の後にゲームが強制的に終了してしまうというバグが自分では解決策が見つからなかったので次回 TA に聞いて解決することにした。実際に聞いて一緒にデバッグをしていたところ、クライアント側で元からある他の関数からコピーしてそのままになっ

ていた remort_play 関数の最後で fclose をしていることが原因だった。あんなに苦労していたのにこの 1 行をコメントアウトするだけで解決したので拍子抜けした。
・タスク 4 の Dos 攻撃対策としてパスワード認証をつけることを考えた。サーバの出力を見ないとパスワードがわからないという点において、実用性があるかどうかは正直微妙かもしれないが自分で試行錯誤して実装の仕方を考えた。特にパスワードが間違っていた際の挙動について、ゲームを終了させるのとゲームに関する情報を表示しないようにするのに苦労した。
・下の改善点に書いたように現在の盤面を見て置く場所の評価点のみに基づく AI では勝率 3 割に満たなかったが、自分と相手の候補数に基づく AI では 50 戦 42 勝で勝率 8 割まであげることができた。

〇改善点
・パスワードが一致しているかどうかを判断するのに苦労した。生成したパスワードを格納する char とクライアントから送られたパスワードを格納する char の一致のさせかたがわからなかった。出力して目で確認して同じであるように見えるのにもかかわらず同じと判断されなかったり、char の文字列を 4 と指定しているのに文字列の長さが変わっていたりした。C 言語の文字列の扱いがよくわかっていないので先頭から 4 文字の char が 1 つずつ正しいか確認するという怪しい方法で判断してしまった。
・パスワード認証が失敗した際にクライアント側に盤面を表示させたくなかったので、このファイルを実行するときはクライアント側は必ず後手になるように固定してしまった。これはパスワード認証を make_remort_player の中でやっていてクライアント側が先手の場合にサーバに手を送った後にしか接続を切ることができないことが原因である。本当は手番はランダムにしたかったが方法がわからなかった。
・AI の実装については現在の状態しか見ていない上に評価点の決め方がひっくり返る石のことは考えずに置いた場所のみしか考えていないのでほんの少しだけ知識をつけた小学生程度の実力しか出せていないはずである。本来は数手先までみてその評価点の合計と相手の打てる手の評価点の合計が低くなるようにするなど色々やれることはあったが今回は一番簡単だと考えた実装にとどめた。ランダムな手と 20 回の対戦をしたところ AI は 7 勝しかできず勝率は 5 割を超えることはないのでかなり弱い AI かもしれない。

6. 実行結果（実行例など）

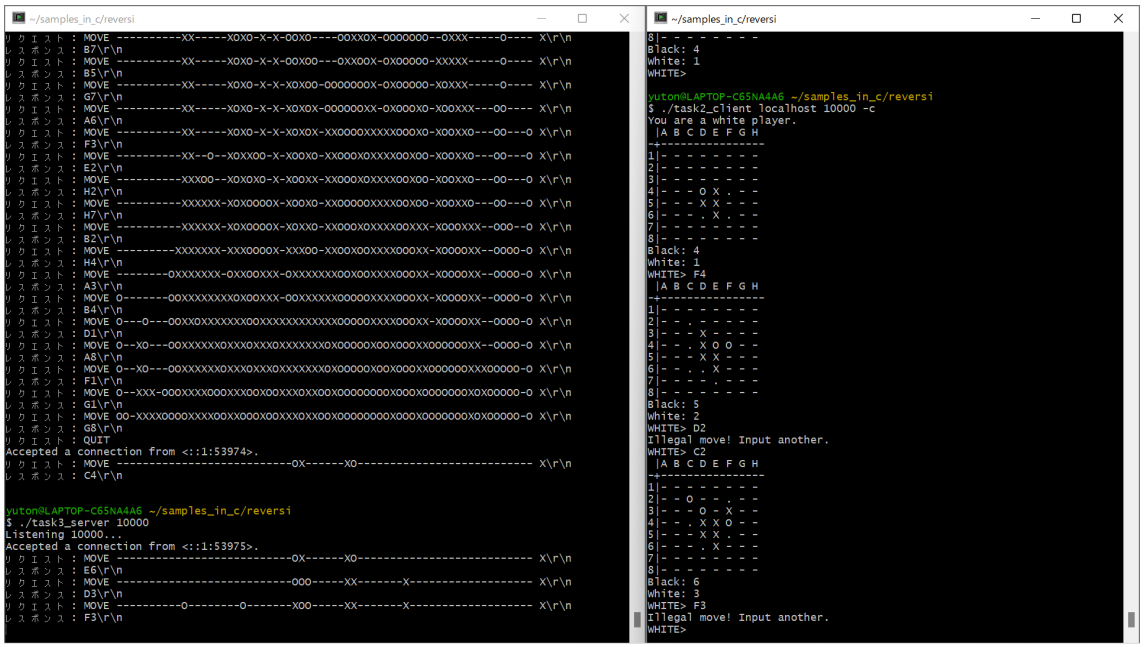

図 1. サーバとクライアントの通信（タスク 2、タスク 3）

図 1 はタスク 2 で実装したクライアント側とタスク 3 で実装したサーバ側を通信させているターミナルである。図 1 を含む以下の図では全て左半分でサーバ、右半分でクライアントを実行している。サーバの下の方（図 1 の左下）では実行し始めを映し、上の方（図 1 の左上）で実行し終わるところを映している。クライアントでは手で実際に入力している様子を映している。

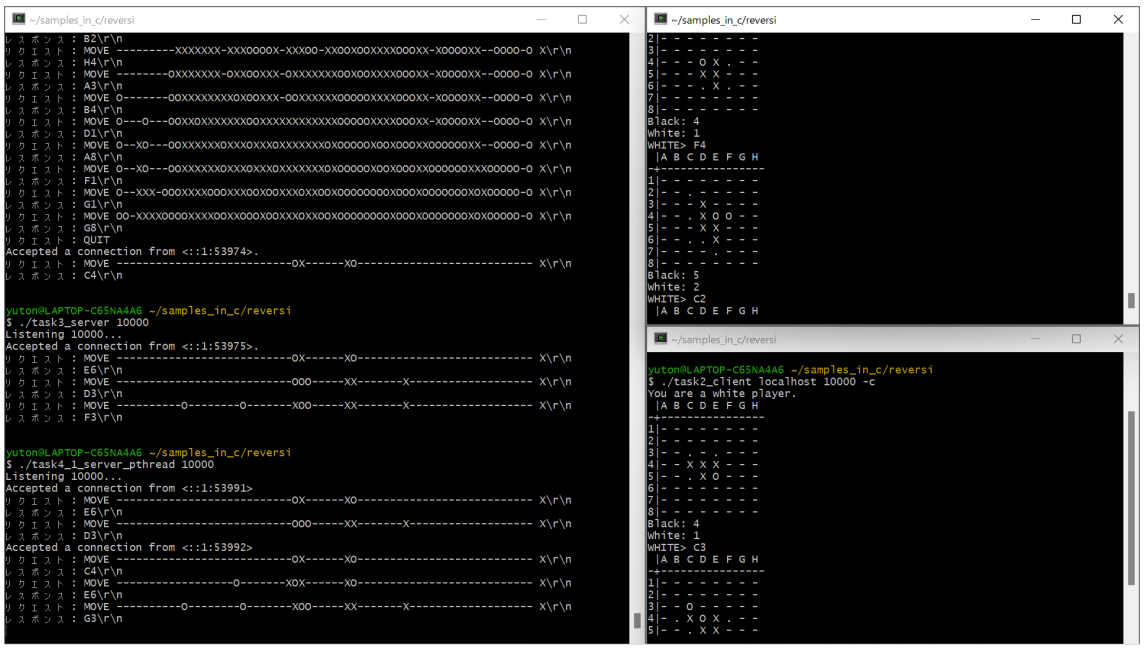

図 2. マルチスレッドの通信（タスク 4）

図 2 はタスク 4 で実装したサーバをマルチスレッドに対応させたターミナルである。図 2 の左側でクライアントを 2 つ起動させて、1 つのサーバと接続させている。今回は 1 つめのクライアントで 2 手進めてから 2 つめのクライアントとの接続を開始して、以降では交互にクライアントから手を打たせたときの様子である。

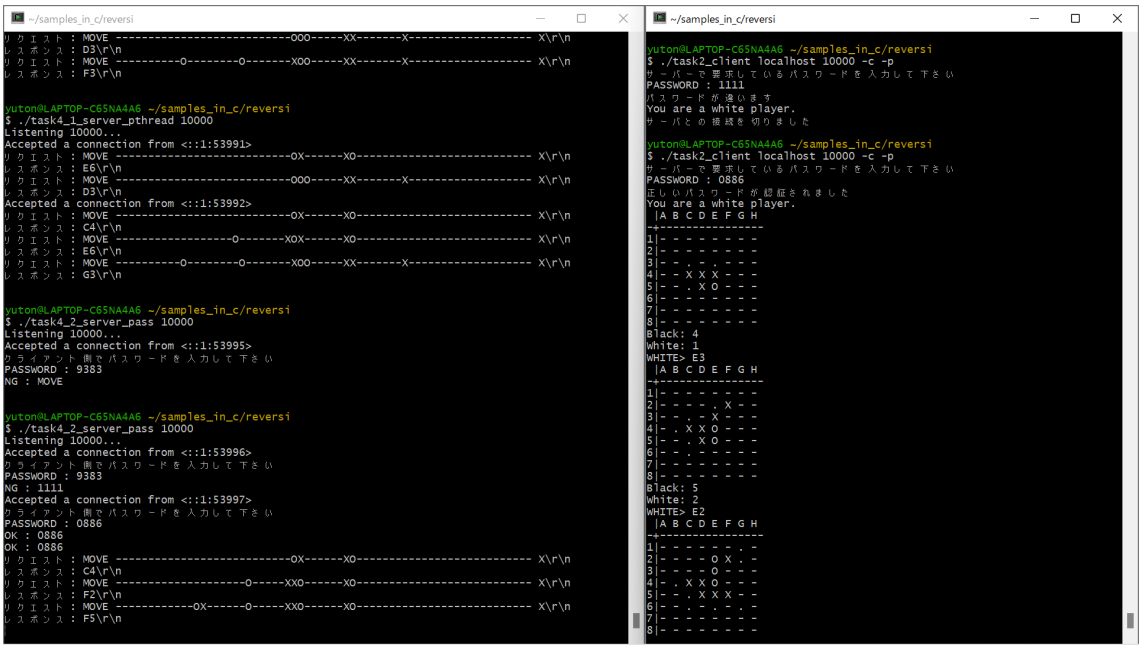

図 3. DOS 攻撃対策のパスワード認証（タスク 4）

図 3 はタスク 4 の DOS 攻撃対策としてパスワード認証を設けた際のターミナルである。クライアントからの接続があった際にサーバ側でパスワードを表示してクライアントにパスワードの入力を求める。入力が間違っている場合にはクライアントの通信を切断して、入力が正しい場合は今まで同様にリバーシを始めている。

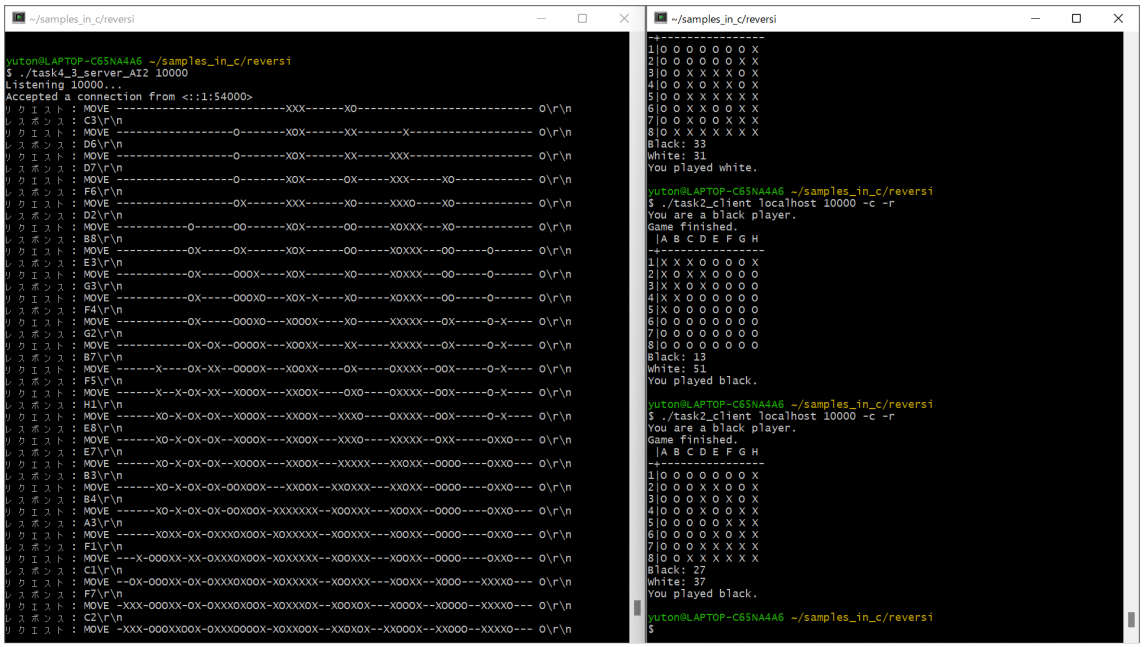

図 4. 候補数 AI の複数回実行（タスク 4）

図 4 はタスク 4 のサーバ側の AI の実装に関して候補数を元に評価点をつける方式を採用した際のターミナルである。クライアント側で引数に「-r」を入れることによって手入力ではなくランダムに打てるように task2_client.c を変更した。図 4 では 3 回実行して 3 回とも AI が勝っている。体感では 8 割ほど AI が勝っているようだ。

<参考文献>

[1]実験指針 C

[2]栢木厚, “令和 4 年 イメージ&クレバー方式でよくわかる 栢木先生の基本情報技術者教室”, 藤沢奈緒美, 株式会社技術評論社, 2022

[3]@mizcii, Qiita “【初心者向け】Makefile 入門”, https://qiita.com/mizcii/items/cfbd2aa17f6b7517c37f

[4]塩田好, “リバーシの評価関数について”, 近畿大学理工学部情報学科卒業研究報告書 (2012)