Checkers :: Monte Carlo Random Descent Playouts

Adam Levy // CS405 Intro to AI // Professor Jon Genetti // UAF Spring 2016

Table of Contents

Board Representation && Move Generation

Design Considerations

Goals

When I set out to write this checkers program I knew that stable and efficient board representation and move generation would be crucial for building a competetive AI. I wanted to design my board early and get as much testing and validation time in on that code as possible.

Mistakes in the board generation would likely lead to wasted time in the AI design process. Instability of the program could lead to crashes during AI development and training or worse, during the tournament.

In addition to stability, I knew that a small board representation would be important for deeper searches, both in terms of memory but more importantly for the speed benefits of moving less memory around. A board representation that can fit in registers allows for faster processing and stack frame changes that pass board states between functions.

Functional Paradigm

I decided to apply a functional programming paradigm to this project. I wanted a program that had no dynamic memory allocation and that used pure functions with no side effects. So there is no CheckerBoard object that mutates. Instead I use small bitboards that simply hold the state of the game. Bitboards do not get manipulated directly. Instead, the bitboards are passed to a move generation function which generates all the child bitboards and leaves the parent unchanged. In this way there are no mutable objects. There are only board states that can be used to generate their children via a deterministic function.

Code Correctness

I made a point to structure my code in such a way as to reduce programming mistakes. Bitboards are efficient but they are not very expressive or easy to visualize. I spent a lot of time sitting with a checker board and writing out bitboards by hand. I ended up writing a set of macro primitives for movements and jumps on the bitboard. The macros generate move and jump positions for all pieces simultaneously using bitwise operations. They also consider the turn and apply the approriate movement direction. This allowed me to write the move generation functions once from a single perspective. As I discuss further below, it also eliminates turn dependent branching.

BitBoard Struct

The Checkers board has 32 positions and each position can be in one of 5 different states: empty, red piece, red king, black piece, black king. This can be minimally held in three `uint32_t`, 32 bit unsigned integers, and one `bool` for the turn. Here is my representation:

```
struct BitBoard
{
    uint32_t red_pos  = RED_INIT_POS_BM;
    uint32_t blk_pos  = BLK_INIT_POS_BM;
    uint32_t king_pos = KING_INIT_POS_BM;
    bool turn = FIRST_TURN;
    bool operator==( const BitBoard & ) const;
    bool operator!=( const BitBoard & ) const;
};
```

The equality operator returns true if everything down to the turn is equal between the two boards being compared. Any difference is an inequality.

The BitBoard is initialized to the following:

```
#define RED_INIT_POS_BM    0xFFF00000
#define BLK_INIT_POS_BM    0x00000FFF
#define KING_INIT_POS_BM   0x00000000
```

The board numbering representation I use is below:

```
//                      00  01  02  03
//                      ----------------|
//   1C  1D  1E  1F     1C  1D  1E  1F  |   07
// 18  19  1A  1B       18  19  1A  1B  |   06
//   14  15  16  17     14  15  16  17  |   05
// 10  11  12  13       10  11  12  13  |   04
//   0C  0D  0E  0F     0C  0D  0E  0F  |   03
// 08  09  0A  0B       08  09  0A  0B  |   02
//   04  05  06  07     04  05  06  07  |   01
// 00  01  02  03       00  01  02  03  |   00
```

Where `0x00` is the least signficant bit and `0x1F` or `32` is the highest bit. In code,

```
bool bit(uint32_t B, int index){
    return ((0x00000001) & (B >> index));
}
```

The row and column indexes are used in the move generation. Rows and columns are retrieved using masks generated by macro functions.

```
#define ROW_MASK(row) (0xf << (4*(row)))
#define COL_MASK(col) (0x11111111 << (col))
```

One way to think about the bitboard in hex is that each hex digit represents a row where `0x1` is the left most square and `0x8` is the right most square.

Move Generation Implementation

The two main move generation functions are named,

```
vector<BitBoard> gen_children(const BitBoard & bb);
vector<BitBoard> follow_jumps(const BitBoard & bb, uint32_t follow_mask = 0xffFFFffF);
```

The `gen_children()` function starts by checking for jumps, since jumps must be taken if there are any available. It calls `follow_jumps()` without a follow mask so that all jumps are considered. Since jumps must be followed to their conclusion, `follow_jumps()` calls itself recursively on each possible jump it finds with a follow mask such that only continuation jumps are followed.

Bit Mask Macros

The majority of the bit manipulation is done using macros which manage the appropriate bitshifts. These macro defines are located in `bit_mask_init.h`.

To understand the following macros, first note that if you are the player whose pieces start on the least significant bits, a left shift is moving your pieces forward.

Also note that for all positions on the board, a bit shift of 4 always produces a valid move. The second possible move from a position is produced from a bit shift of either 3 or 5, depending on the row. And a few positions on the board only have one valid move.

The following bit masks mask off the positions from which a left or right shift of 3 or 5 is not a valid move. These are all based on the row and column masks given above. The bit representation of the masks is displayed for the move shift masks. The jump masks are for the same purpose: masking off the positions so that only the positions for which the given shift produces a valid move remain.

```
// MOVE SHIFT MASKS
#define LEFT3_MASK       ((~COL_MASK(0)) & (ROW_MASK(0) | ROW_MASK(2) | ROW_MASK(4) | ROW_MASK(6)))
//0b00001110000011100000111000001110
#define LEFT5_MASK       ((~COL_MASK(3)) & (ROW_MASK(1) | ROW_MASK(3) | ROW_MASK(5)))
//0b00000000001110000011100000111000;
#define RIGHT3_MASK      ((~COL_MASK(3)) & (ROW_MASK(1) | ROW_MASK(3) | ROW_MASK(5) | ROW_MASK(7)))
//0b01110000011100000111000001110000;
#define RIGHT5_MASK      ((~COL_MASK(0)) & (ROW_MASK(2) | ROW_MASK(4) | ROW_MASK(6)))
//0b00000000011100000111000011100000;
// JUMP SHIFT MASKS
#define LEFT7_MASK       ((~COL_MASK(0)) & (~(ROW_MASK(6) | ROW_MASK(7))))
#define LEFT9_MASK       ((~COL_MASK(3)) & (~(ROW_MASK(6) | ROW_MASK(7))))
#define RIGHT7_MASK      ((~COL_MASK(3)) & (~(ROW_MASK(0) | ROW_MASK(1))))
#define RIGHT9_MASK      ((~COL_MASK(0)) & (~(ROW_MASK(0) | ROW_MASK(1))))
```

Bit Shift Macro Functions

Using the masks above, we can apply the appropriate bitshifts to the bitboards and `bitwise or` them back together. These macro functions generate all the possible moves from the given bits set in the bitboard, `pos`, which is assumed to be a `uint32_t`. Move left is a shift left, which is forward for the player whose pieces start on the least significant bits.

```
// GENERATE MOVES OR PIECES CONCURRENTLY ON A BOARD
#define MOVE_LEFT(pos)      (((pos) << 4) | (((pos) &  LEFT3_MASK) << 3) | (((pos) &  LEFT5_MASK) << 5))
#define MOVE_RIGHT(pos)     (((pos) >> 4) | (((pos) & RIGHT3_MASK) >> 3) | (((pos) & RIGHT5_MASK) >> 5))
```

These shifts are always valid, and I use them to help isolate individual moves.

```
#define LEFT_4(pos)         ((pos) << 4)
#define RIGHT_4(pos)        ((pos) >> 4)
```

The jump shifts work pretty much the same way.

```
// GENERATE JUMP POSITIONS OR JUMP PIECES CONCURRENTLY ON A BOARD
#define JUMP_LEFT(pos)      ((((pos) &  LEFT7_MASK) << 7) | (((pos) &  LEFT9_MASK) << 9))
#define JUMP_RIGHT(pos)     ((((pos) & RIGHT7_MASK) >> 7) | (((pos) & RIGHT9_MASK) >> 9))
```

These take the guesswork out of remembering which shift direction is forward for which side. For my board representation Black's pieces are on the least significant bits.

```
// FORWARD GIVES VALID POTENTIAL MOVES FROM THE GIVEN PIECES (pos)
#define BLK_FORWD(pos)      MOVE_LEFT(pos)
#define BLK_FORWD_4(pos)    LEFT_4(pos)
#define BLK_JUMP(pos)       JUMP_LEFT(pos)
// BACKWARD GIVES PIECES THAT COULD HAVE MOVED FROM THE GIVEN POSITIONS (pos) OR KING MOVES
#define BLK_BCKWD(pos)      MOVE_RIGHT(pos)
#define BLK_BCKWD_4(pos)    RIGHT_4(pos)
#define BLK_JUMP_BACK(pos)  JUMP_RIGHT(pos)

// SAME FOR RED
#define RED_FORWD(pos)      MOVE_RIGHT(pos)
#define RED_FORWD_4(pos)    RIGHT_4(pos)
#define RED_JUMP(pos)       JUMP_RIGHT(pos)
#define RED_BCKWD(pos)      MOVE_LEFT(pos)
#define RED_BCKWD_4(pos)    LEFT_4(pos)
#define RED_JUMP_BACK(pos)  JUMP_LEFT(pos)
```

Finally, these macros allow me to always write the code from the perspective of the player whose turn it is, without worrying about the direction of movement with respect to the board.

```
#define RED 0
#define BLK 1

#define IS_RED(turn)        (0xffffffff * !(turn))
#define IS_BLK(turn)        (0xffffffff *  (turn))

// REMOVE THE GUESS WORK AND SAVE REDUNDANT CODE
#define FORWD(turn, pos)       ((BLK_FORWD(pos)     & IS_BLK(turn)) | (RED_FORWD(pos)     &
IS_RED(turn)))
#define FORWD_4(turn, pos)     ((BLK_FORWD_4(pos)   & IS_BLK(turn)) | (RED_FORWD_4(pos)   &
IS_RED(turn)))
#define FORWD_JUMP(turn, pos)  ((BLK_JUMP(pos)      & IS_BLK(turn)) | (RED_JUMP(pos)      &
IS_RED(turn)))

#define BCKWD(turn, pos)       ((BLK_BCKWD(pos)     & IS_BLK(turn)) | (RED_BCKWD(pos)     &
IS_RED(turn)))
#define BCKWD_4(turn, pos)     ((BLK_BCKWD_4(pos)   & IS_BLK(turn)) | (RED_BCKWD_4(pos)   &
IS_RED(turn)))
```

```
#define BCKWD_JUMP(turn, pos)  ((BLK_JUMP_BACK(pos) & IS_BLK(turn)) | (RED_JUMP_BACK(pos) &
IS_RED(turn)))

#define KING_ME_ROW_MASK(turn) ((ROW_MASK(7)        & IS_BLK(turn)) | (ROW_MASK(0)        &
IS_RED(turn)))
```

This eliminates the code inside the turn dependent if statement in the move generation code. I believe this to help avoid branch misprediction and work with CPU's prefetching of instructions. I need to perform timing tests with this code compared against code that branches once based on turn. However, since the opponent's moves are generated but masked off, my implementation has to discard half of all the results of the inplace bitshift operations it performs. That could make it slower in fact. But no way to know unless it's timed and tested.

Putting It All Together

I won't go through the entire move generation code. But I want to show two parts that demonstrate how I use the above masks to generate all child boards somewhat simultaneously. First note that I initially assign `play_pos` and `oppo_pos` to the appropriate `uint32_t` from the input `BitBoard` according to the turn. The following code can be found in `checkerboard.cpp`

```
uint32_t occupied = (play_pos | oppo_pos);
uint32_t empty = ~occupied;

uint32_t movers      = BCKWD(turn, empty) & play_pos & ~bb.king_pos;
uint32_t king_movers = (BCKWD(turn, empty) | FORWD(turn, empty)) & play_pos & bb.king_pos;
```

In words, *a position has a mover if it can be reached from an empty position and contains a play piece*.

We can use these to generate the move locations. Ultimately we have to isolate the individual valid movers and their respective valid move locations and mask appropriately to create the valid child boards. In order to avoid iterating through each individual bit and asking if it contains a mover, I used nested for loops and iterated by row and column. This way if a row happens to contain no pieces, the entire row can be skipped.

```
uint32_t movers_remaining      = movers;
uint32_t king_movers_remaining = king_movers;
for (size_t r = 0; r < 8; r++){
    if (ROW_MASK(r) & (movers_remaining | king_movers_remaining)){
        // MOVES ON THIS ROW
        for (size_t c = 0; c < 4; c++){
            uint32_t p_piece = COL_MASK(c) & ROW_MASK(r) & (movers_remaining | king_movers_remaining);
            if (p_piece){   // MOVE FOUND: a valid move can be made from a piece on r,c
                // find individual valid move locations from here
                ...
```

As movers are found they are checked off so that if no more movers remain in a row, the inner for loop can easily check and break early.

```
                ...
                // uncheck move location from remaining moves
                if(is_king){
                    king_movers_remaining &= ~p_piece;
                } else{
                    movers_remaining &= ~p_piece;
                }
            }
```

```
            // no remaining moves in row
            if (!(ROW_MASK(r) & (movers_remaining | king_movers_remaining))){
                break;
            }
        }
    }
}
```

I could have nested the for loops the other way: columns then rows. But I felt that a completely empty row of length 4 is more likely than a completely empty row of length 8.

The jump generation works much the same way, only the follow mask is applied so that only the continuation of previous jumps, if any, are followed.

Checkers AI

Challenges of Neural Networks

Early on in the project I had some concerns about the Blondie24 genetic algorithm training method for evolving a neural network. The major issue I saw with it was how time and resource intensive the training is. This makes catching bugs and iterating the design take a long time. There are parameters that affect how fast and how effectively the neural network will evolve and you won't know how good the values you picked were until much later.

Although there are much faster ways to train neural networks they all are forms of supervised learning which means that they require example data. That means you would need a large dataset of example board states to be properly labeled with their relative strength. Early on in the project I thought about how to generate such a data set.

Another shortcoming of a simple feedforward neural network or multi layer perceptron (MLP) is its inability to see structural relationships in the input data. For example, simple MLP's do not perform well on most image datasets. There are correlations and other more complex relationships between neighboring pixels in images that are missed by the MLP since there is nothing in its neuron structure that associates pixels that are near each other. For an MLP, any relevant correlations between pixels must be completely learned and represented by the parameters of the fully connected layers.

Researchers have instead turned to Convolutional Neural Networks (CNN) for images which use low level layers of neurons that are fully connected only in local regions of the input. The locally connected sets of neurons feed into other locally connected sets of neurons in the layer above and ultimately feed into a set of fully connected layers. This allows the neural network to preserve structural aspects of the image from the outset. The weights of the neurons then only have to find relevant patterns within thier locally connected set of inputs.

I wanted to generate data for a CNN, which lead me to writing some simple code for random game playouts and some simple CUDA kernels for data processing.

Generating Training Data

One method that I experimented with to generate training data was to playout psuedo random games. On each move, a random number would be drawn. If the number was even, the next move would be randomly selected. Otherwise, the move would be selected using a simple piece count evaluation with minimax. I saved the data of these game playouts.

Later, I wrote CUDA kernels that converted the saved game playouts from their consice `uint32_t` BitBoard representation to an expanded 3 x 8 x 8 tensor of floats with values of ones and zeros. I generated a label based on the outcome of each game. I interpolated from 0 to 1 or -1 (depending on win or loss) from the start to the end of the game. The idea was that earlier states in the game are less certainly a win. I was hoping that a CNN might be able to see some patterns across all of the playouts.

I didn't have time to follow through on exploring training CNN on this data. I mention it here because it lead to my exploration of CUDA, Monte Carlo Tree Search, and my much simplified version which I refer to Monte Carlo Random Descent.

Monte Carlo Random Descent

In one sentence, Monte Carlo simulations use randomness to explore a space. In a Monte Carlo simulation, a large number of

paths are randomly explored through the space. You could imagine this like ants spreading out randomly in the search for food. Some paths result in an optimum location in the space, just like some ants randomly stumble on food. As some directions in the space are found to be *good* those paths get followed and further explored with a higher probability.

Monte Carlo Tree Search (MCTS) is an application of this technique for exploring a game tree. It has been applied to game AI for board games such as Go, Arimaa, Scrabble, Backgammon, Bridge and Checkers. MCTS uses repeated random playouts from a given board state to asses move quality. The move with the highest proportion of wins from the random playouts is preferred. MCTS treats the game tree as the "multi-armed bandit" problem. Once enough completely random playouts have passed through a leaf node, that node is expanded and its children are played out. Again, the child node with the best win percentage is expanded, and so on. This results in an asymmetrically expanded game tree, where the best moves get expanded first so that more playouts go through those nodes, improving their statistics.

I believe implementing MCTS on the GPU requires a tree structure and some mechanism to periodically check in on the playouts and expand preferred nodes. First I instead decided to focus on implementing the random playouts on the GPU.

Cuda Implementation

To implement a Cuda kernel to perform random playouts I first needed to make sure that my move generation code would work on the GPU. The main incompatability was the use of `std::vector`. You cannot use the standard library in device code on the GPU. So I needed a mechanism that would pass arrays of unknown numbers of child BitBoards between functions. I used a struct that carried a pointer to an array and a size variable. The array was dynamically allocated at a size of 30 BitBoards. The calling function is responsible for deleting that memory. This was the one place where I couldn't avoid dynamic memory management.

```
struct BitBoardArray
{
    BitBoard_gpu *  bb_ary = 0;
    size_t      size = 0;
};
```

I also created a `BitBoard_gpu` struct, identical to `BitBoard`. The reason for this was to keep the C++ `checkerboard.cpp` compilation unit seperate from the CUDA compilation so that my original functions could be easily used in other branches of the git repo. This is required for the BitBoard struct so that the equality operators can be compiled for the GPU device code. Any functions that will be called from the GPU require the `__device__` label. This code can be found in `checkerboard_gpu.hpp`.

```
struct BitBoard_gpu
{
    uint32_t red_pos  = RED_INIT_POS_BM;
    uint32_t blk_pos  = BLK_INIT_POS_BM;
    uint32_t king_pos = KING_INIT_POS_BM;
    bool turn = FIRST_TURN;
    __device__ bool operator==( const BitBoard_gpu & ) const;
    __device__ bool operator!=( const BitBoard_gpu & ) const;
    __device__ __host__ BitBoard_gpu & operator=( const BitBoard & );
};

__device__ BitBoardArray gen_children_gpu(const BitBoard_gpu & bb);
__device__ BitBoardArray follow_jumps_gpu(const BitBoard_gpu & bb, uint32_t follow_mask = 0xffFFFffFF);
__device__ size_t bit_count_gpu(uint32_t i);
```

cuRand

Using psuedo random numbers on the GPU requires the use of Cuda's curand library. There are a number of different random number generation algorithms and the options of the library can be overwhelming. But the process is relatively straight

forward.

To generate a random number from a kernel you need a `curandState` object on the GPU. The `curandState` object holds the random state, like the seed, sequence and offset. Two `curandState` objects set to the same seed, sequence, and offset generate the exact same random numbers. So each thread needs its own `curandState`. Each `curandState` must be initialized differently.

```
__global__ void setup_kernel(curandState *state, ullong r_offset)
{
    ullong idx = threadIdx.x + blockDim.x * (blockIdx.x * gridDim.y + blockIdx.y);
    ullong sequence = threadIdx.x;
    ullong seed = (idx + 1) * r_offset;
    curand_init(seed, sequence, r_offset, &state[idx]);
}
```

The kernel for random descent generates integers uniformly between 0 and the number of child boards less one.

Playout Loop

To keep the game playouts below a certain length I used a simple piece count using `bit_count_gpu()` to award the win on games that went on too long.

I also added cycle detection when kings are in play to prevent randomly descending through king cycles.

Below is the while loop used in the random descent kernel for playing out the game. The complete code can be found in `mcmc.cu`.

```
while(children.size && n_moves < MAX_MOVES){
    n_moves++;
    size_t b = children.size;
    frand = curand_uniform(&localState);
    int irand = frand * b;

    BitBoard_gpu cc = children.bb_ary[irand];

    // Cycle detection
    if ((cc.king_pos ^ bb.king_pos) && (parent.king_pos ^ gparent.king_pos)){
        if (bb.turn == BLK){
            if ((cc.blk_pos & cc.king_pos) == (gparent.blk_pos & gparent.king_pos)){
                continue;
            }
        } else{
            if ((cc.red_pos & cc.king_pos) == (gparent.red_pos & gparent.king_pos)){
                continue;
            }
        }
    }

    gparent = parent;
    parent = bb;
    bb = cc;

    delete [] children.bb_ary;
    children = gen_children_gpu(bb);
}
// Select winner ...
```

Cuda Error 6: Cuda Timeout

One major issue I had to work around was the timeout enforced by the OS on kernal calls to the GPU. My code was running close up against this limit even just for initializing the `curandState` variables. I found the maximum number of `curandState` variables I could initialize in a single kernal call on my Macbook Pro was about 31k. So this limited the max number of playouts I performed on a single kernel call.

Launching The Kernel

For each move I generated the child boards on the CPU and copied them over to the GPU. I had my kernel run evenly on all child boards.

I launched blocks of 1024 threads. I launched blocks in a grid of X blocks for each N child boards. Where X was however many threads could be run with the available number of `curandState` variables. That means that each child board had X * 1024 playouts.

For each move I ran my kernel as many times as I could within the time limit. The kernel adds the number of wins for each board to its respective slot in the array `d_wins`.

Once the time is up the wins are copied back to the host and the boards are sorted by win count on the CPU. The board with the highest win count is selected.

```
while (time - start_time < limit){
    random_descent<<<blocks,NUM_ITERS>>>(d_state, d_boards, d_wins, player);
    num_iter++;
    checkCudaErrors(cudaDeviceSynchronize());
    // Sort boards by win count, lowest to highest
    ullong new_time = system_clock::to_time_t(system_clock::now());
    if (new_time - start_time < 15 - elapsed_time){
        checkCudaErrors(cudaMemcpy(h_wins,
                           d_wins,
                           num_children * sizeof(ullong),
                           cudaMemcpyDeviceToHost));
    }
    elapsed_time = new_time - time;
    limit = 15 - elapsed_time;
    time = new_time;
    cout << "Time elapsed: " << time - start_time << endl;
}
thrust::sort_by_key(h_wins, h_wins + num_children, &children[0]);
```

Analysis

The MCRD I implemented outperformed other trained neural networks in the AI class. It appeared to make intelligent moves overall and held to reasonable strategies like keeping the back row unmoved until necessary. It seems to position its pieces well and then let the opponent walk into poor situations. From the start it was apparent that the MCRD algorithm often resulted in moves that forced advantageous trades. The algorithm appeared to play equally well as red or black.

MCRD Limitations

One major and obvious limitation of MCRD is how computationally expensive it is. When running on the GPU my computer comes to a standstill. Moreover much of the playouts are wasted since playouts are performed equally on all child boards regardless of their relative strengths. This means that a lot of computation is wasted playing out boards that will clearly not result in the max number of wins when really the top two or three contenders might benefit more from additional playouts to reduce the variance and possible overlap of their win percentages. This is what the MCTS is supposed to help with.

Another limitation, or at least a serious question, of MCRD is how much strategic play can be obtained from random playouts? It seems unlikely to me that random playouts can consistently beat sound strategy like that from master checkers players or programs like Chinook. I have yet to pit my checker program against a human or Chinook. I plan to set up my AI to work with

my GUI this summer.

Future Improvements

Proportional Playout

One simple way to improve MCRD is to have each thread choose its assigned starting board state with probability equal to the win percentage and randomly select a different starting board state otherwise. I implemented this before the second round of the tournament but I was running into the timeout error I mentioned above so I couldn't use it.

I don't think this will significantly improve the quality of play in most cases but there will be edge cases where the two top boards are close enough in win % that additional playouts will help determine the move with the true advantage.

MCRD Trained Neural Network

Ultimately I want to get back to implementing the CNN I discussed above using training data that I generate. Instead of making up the label values, I will instead create a database of the results of MCRD from certain board states and use their values to train the CNN. It would be interesting to see the result of psuedo random playouts that were partially guided by the neural network.