Halma board game

- the board consists of a grid of 16 by 16 squares
- the game may be played by two or four players
  - we will only consider two player game
- each player's camp consists of a cluster of adjacent squares in one corner of the board
  - in two-player games, each player's camp is a cluster of 19 squares
  - in the four-player game, each player's camp is a cluster of 13 squares
- see ⟨http://en.wikipedia.org/wiki/Image:Halma4.svg⟩

Halma board game

- each camp exists in the board corner
- each player has a set of pieces in a distinct colour, of the same number as squares in each camp.

Halma Objective

- is to move all your pieces to occupy the opposing camp
  - which is diagonally opposite to your own
- Victorian board game
- a move consists of moving one piece:
  - either one square (assuming it is empty)
  - or hopping over another piece (of any colour)
    - a piece may hop multiple times in any direction and it leaves any hopped pieces alone

Kangaroo Halma

- variation - to speed the game up, allows the hopping piece to take giant hops over any piece in any direction
  - as long as all squares either side of the hop are empty
  - each hop must be symmetrical
  - you can still perform multiple hops during one move for one piece
  - each hop may be of different symmetrical lengths

How can you program the game?

- need to decide which data structures to use
- need a board representation
- maybe use two 256 bit sets
  - one to determine whether a position is occupied
  - one to determine whether black or white

How can you program the game?

```
#define SQUARES_ON_BOARD (16*16)
#define BITS_PER_BYTE 8

typedef int h_bitset[SQUARES_ON_BOARD /
                     (sizeof(unsigned int) * BITS_PER_BYT

typedef struct {
   h_bitset occupied;
   h_bitset colour;
} h_board;
```


Efficiency

- problem with the previous method
- to search for legal moves would require searching the complete 256 bitset
  - maybe we could utilise a list of pieces array as well
  - gives us a fast method to find each piece
  - at the expense of having to maintain both data structures

Revised halma board data structures

```
#define SQUARES_ON_BOARD (16*16)
#define BITS_PER_BYTE 8
#define PEGS_PER_COLOUR 19
#define WHITE 0
#define BLACK 1
#define NO_PLAYERS 2

typedef int h_bitset[SQUARES_ON_BOARD /
                     (sizeof(unsigned int) * BITS_PER_BYT

typedef struct {
   h_bitset occupied;
   h_bitset colour;
   unsigned char position[NO_PLAYERS][PEGS_PER_COLOUR];
} h_board;
```

Revised halma board data structures

- how many bytes are used by the first version of the halma board data structure?
- how many bytes used by the second?
- is this an acceptable tradeoff?

Board manipulation routines

- need a function to initialise the board
- need a function to test whether a square is empty
- need a function to effect a move

Initialisation

```
void board_init (h_board *b)
{
   add_piece (b, WHITE, 0, 0);
   add_piece (b, WHITE, 1, 1);
   add_piece (b, WHITE, 2, 2);
   add_piece (b, WHITE, 3, 3);
   add_piece (b, WHITE, 4, 16);
   add_piece (b, WHITE, 5, 17);
   add_piece (b, WHITE, 6, 18);
   add_piece (b, WHITE, 7, 32);
   add_piece (b, WHITE, 8, 33);
   add_piece (b, WHITE, 9, 34);
   add_piece (b, WHITE, 10, 48);
   add_piece (b, WHITE, 11, 49);
   add_piece (b, WHITE, 12, 50);
   add_piece (b, WHITE, 13, 64);
   add_piece (b, WHITE, 14, 65);
}
```


add_piece

```
void add_piece (h_board *b, int colour, int piece, int sq
{
   INCL(b->occupied, square);
   if (colour == WHITE)
      EXCL(b->colour, square);
   else
      INCL(b->colour, square);
   b->position[piece] = square;
}
```

INCL

```
void INCL (h_bitset *b, int element)
{
   int array_element = element/
                       (sizeof(unsigned int) * BITS_PER_B
   int bit_in_element = element % sizeof(unsigned int);
   b[array_element] |= (1 << bit_in_element);
}
```


EXCL

```
void EXCL (h_bitset *b, int element)
{
   int array_element = element/
                       (sizeof(unsigned int) * BITS_PER_B
   int bit_in_element = element % sizeof(unsigned int);
   b[array_element] &= (! (1 << bit_in_element));
}
```


IS_IN

```
int IS_IN (h_bitset *b, int element)
{
   int array_element = element/(sizeof(unsigned int)
                       * BITS_PER_BYTE);
   int bit_in_element = element % sizeof(unsigned int);
   return (b[array_element] & (1 << bit_in_element)) != 0
}
```