

Game Trees

- in this section of the course we will look at the following topics (and not in this order and not sequentially):
 - C programming
 - game tree searching
 - minimax, alphabeta
 - Othello
 - Chess
 - evaluation functions

Game Trees

- C programming will normally cover the first third of the lecture
 - here we assume the student is familiar with control, sequence and iteration syntax
 - we will cover pointers, structures, parameters and macros
 - enough to aid understanding of the Othello game code

Othello

- a game which has a **simple rule set** (<http://www.britishothello.org.uk/rules.html>)
- black moves first and places a piece on the board
 - it must trap one or more white piece between the new black piece and a previous black piece
 - each trapped piece now becomes a black piece
 - pieces can be trapped in any one or more of the eight possible straight lines emitting from the new piece
- the game ends when neither black or white can move
 - the winner is the player with the most pieces

Othello

- chosen Othello as the rules are simple
 - however the game is not so simple
- relatively easy to program a weak player
 - compared to a chess program

Othello and programming

- key is board representation
 - needs to be compact, so taking copies of the current board position is not expensive
 - needs to provide an easy mechanism for exploring all new moves
 - needs to provide an easy mechanism to determine a good move from a potentially bad move

Rule and turn based games

- normally are programmed by constructing a game tree and searching it to a given depth
 - chess, othello, naughts and crosses, draughts
 - each level (called a *Ply*) represents all the moves which can be made by a particular colour
- as long as the computer can evaluate the board position accurately and it can examine the game tree to a greater depth than a human then it should be able to win..
 - however, game trees normally grow exponentially at each level
 - evaluating a board position mid way through a game is not always accurate

Non deterministic rule based games

- are games where searching the entire game tree would be impossible and therefore some "tricky" programming must be employed
 - often termed *heuristics*, rules of thumb which AI programmers use to short circuit full searching of the game tree
- include Othello and Chess
 - Chess has an estimated 10^{120} moves

Othello board representation in C

- many ways to crack a nut but here is one simple data structure which is compact and relatively fast:
- ```
typedef unsigned long long BITSET64;

struct board {
 BITSET64 Colours;
 BITSET64 Used;
}
```
- 16 bytes of data, each bit of `Colours` determines whether it is white, a 1, or black, a 0
- `Used` determines whether a piece has been played in that position

## Othello board representation in C

- the positions of the bits in the two bitsets refer to the board positions:

|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |

## Data structure Pros/Cons

- Pros
  - compact, copying will be quick
  - each BITSET64 maps onto a fundamental C data type
  - relatively fast to search for legal moves
  - easy to evaluate a board position
- Cons
  - searching for legal moves makes for tricky programming
  - may not be so fast when searching for legal moves
    - must search 64 bits
    - depends on how fast the processor is at bit testing

## Bit manipulation functions

```

/*
 * IN - return 1 or 0 depending whether, bit, is in, set
 */
static __inline__ int IN (BITSET64 set, int bit)

```

```

/*
 * INCL - set, bit, in, set.
 */
static __inline__ void INCL (BITSET64 *set, int bit)

```

## Bit manipulation functions

```

/*
 * EXCL - unset, bit, in, set.
 */
static __inline__ void EXCL (BITSET64 *set, int bit)

```

- obviously to create and initialise an empty set you use:

```

BITSET64 set = 0;

```

**IN function**

```

static __inline__ int IN (BITSET64 set, int bit)
{
 if (sizeof(BITSET64) > sizeof(unsigned int)) {
 unsigned int *p = (unsigned int *)&set;

 if (bit >= sizeof(unsigned int)*8)
 /* high unsigned int */
 return (p[1] >> (bit-(sizeof(unsigned int)*8))) & 1;
 else
 /* low */
 return (p[0] >> bit) & 1;
 }
 else
 return (set >> bit) & 1;
}

```

**INCL function**

```

/*
 * INCL - set, bit, in, set.
 */
static __inline__ void INCL (BITSET64 *set, int bit)
{
 if (sizeof(BITSET64) > sizeof(unsigned int)) {
 unsigned int *p = (unsigned int *)&set;

 if (bit >= sizeof(unsigned int)*8)
 /* high unsigned int */
 p[1] |= 1 << (bit-(sizeof(unsigned int)*8));
 else
 /* low */
 p[0] |= 1 << bit;
 }
 else
 (*set) |= 1 << bit;
}

```

**EXCL function**

```

/*
 * EXCL - unset, bit, in, set.
 */
static __inline__ void EXCL (BITSET64 *set, int bit)
{
 if (sizeof(BITSET64) > sizeof(unsigned int)) {
 unsigned int *p = (unsigned int *)&set;

 if (bit >= sizeof(unsigned int)*8)
 /* high unsigned int */
 p[1] &= ~(1 << (bit-(sizeof(unsigned int)*8)));
 else
 /* low */
 p[0] &= ~(1 << bit);
 }
 else
 (*set) &= ~(1 << bit);
}

```

**Tutorial**

- download the tiny Othello [source code](http://flopsie.comp.glam.ac.uk/download/c/o64bit.c) <<http://flopsie.comp.glam.ac.uk/download/c/o64bit.c>>, compile and run it
- enable USE\_CORNER\_SCORES in the source code, does it play a better game or not
  - what is this option doing?
  - why is it better or worse?