Chomp.

Jeroen Laros

April 4, 2008

Contents

1 Introduction.

This is a document about the first assignment of the Artificial Intelligence course. Some (if not most) of the code isn't really part of the assignment, partly because the information reached me too late (in case of the (m x n) array, which is of variable size) and partly because I thought the assignment was more complicated than it was. For this reason the brute-force strategy and the stack are implemented.

2 Problem.

The game chomp is played by two players on a (m x n) bar of which the coordinate $(0, 0)$ is poisoned. Each player must take a bite from the bar until one of them is forced to eat the poisoned coordinate. An example on a (3 x 4) bar:

```
XXX
XXX
XXX
XXX
Player1 moves: 2 2
XXX
XXX
XX
XX
Player2 moves: 0 3
XXX
XXX
XX
Player1 moves: 1 1
XXX
X
X
Player2 moves: 0 1
XXX
Player1 moves: 1 0
Player1 wins.
```

We will use words like winning and losing in the following sense: A bar is either winning or losing. When you see a bar; you've already lost or won. Now, a bar is winning if and only if it has at least one losing child and a bar is losing if and only if it has only winning children. To complete the definition: the (1 x 1) bar is losing. Another definition is bite: by taking a bite from the bar at coordinate (i, j), ($i + j \neq 0$ and (i, j) is filled) all coordinates (k, l) ($k \geq i, l \geq j$) will disappear. And for completeness: a bar A is a child of another bar B if and only if B can be transformed to A by taking a bite.

3 Approach.

The tools used for tackling this problem are the following strategies: human input, random, brute force, several heuristics and two very simple dumb strategies. Let's review all of them in detail.

3.1 Strategy 1: Human input.

This function keeps asking input from the user until valid input is given.

3.2 Strategy 2: Random strategy.

This strategy takes a random bite from the bar.

3.3 Strategy 3: Brute force strategy.

Based on the definitions of winning and losing bar, this function recursively tries all moves until either a winning move is found, or until all options are exhausted. In the latter case it will make a random move.

3.4 Strategy 4: Heuristic strategy 1.

This strategy is based on the trivial cases. It the (n x n), (2 x n) and (n x 2) cases (and thereby also the (n x 1) and (1 x n) ones). If the bar is losing it will take a random move.

3.5 Strategy 5: Heuristic strategy 2.

This strategy is also based on the trivial cases, but has another approach, it tries not to leave a trivial bar. It uses the stack from the brute force strategy to try a move. If the bar is smaller than (3 x 3) all moves will leave a trivial case, so it will do a random move.

3.6 Strategy 6: Combined heuristic strategy.

This strategy combines the heuristic strategies 1 and 2.

3.7 Strategy 7: Dumb strategy 1.

This is a very simple strategy which always takes the left-lower square.

3.8 Strategy 8: Dumb strategy 2.

This is also a very simple strategy which always takes the biggest bite available ($(0,1) \oplus (1,0) \in bite$).

4 Implementation.

We use a (m x n) dynamical array of booleans as a bar. True is filled, false is empty. We use a stack to try out moves.

5 Experiments.

Let's first review some simple cases. The results are ordered in a matrix, vertical is Player1, horizontal is Player2. The results for strategy 1 and 3 are omitted because they are both not interesting. These are the results of a (3 x 8) bar. More results can be retrieved from the testrun.log file, or by running the testrun script.

```
m: 3 n: 8 Number of games: 100
   2     4       5       6       7       8
2 49    8       73      13      59      14
4 92    46      98      37      90      86
5 36    4       50      0       44      0
6 85    59      100     47      100     84
7 41    10      56      0       100     0
8 84    0       100     0       100     0

m: 3 n: 8 Number of games: 1000
   2     4       5       6       7       8
2 506   72      634     111     523     191
4 904   440     965     358     896     903
5 377   36      504     0       481     0
6 855   578     1000    472     1000    878
7 477   104     519     0       1000    0
8 847   0       1000    0       1000    0

m: 3 n: 8 Number of games: 10000
   2     4       5       6       7       8
2 5073  755     6516    1107    5232    1943
4 9067  4376    9611    3533    8855    9121
5 3588  371     5129    0       5075    0
6 8730  5878    10000   4777    10000   8823
7 4768  1145    4925    0       10000   0
8 8532  0       10000   0       10000   0

m: 3 n: 8 Number of games: 100000
   2     4       5       6       7       8
2 50356 7866    64072   10622   53034   20011
4 90558 44157   95987   35534   88408   91344
5 36534 3948    51081   0       50717   0
6 87237 58347   100000  47829   100000  88208
```

```
7 46966 11594   49279   0       100000  0
8 85684 0       100000  0       100000  0
```

And these are from a (38 x 24) bar.

```
m: 38 n: 24 Number of games: 100
   2     4       5       6       7       8
2 54    4       66      4       50      8
4 94    48      100     40      95      100
5 45    0       55      0       47      0
6 98    52      100     45      100     100
7 50    5       53      0       100     0
8 95    0       100     0       100     0

m: 38 n: 24 Number of games: 1000
   2     4       5       6       7       8
2 517   43      624     60      520     41
4 960   515     985     416     948     1000
5 374   18      501     0       464     0
6 952   602     1000    505     1000    1000
7 480   52      536     0       1000    0
8 953   0       1000    0       1000    0

m: 38 n: 24 Number of games: 10000
   2     4       5       6       7       8
2 4966  490     6265    557     5109    468
4 9543  5035    9836    4035    9470    9973
5 3711  174     4964    0       4464    0
6 9432  5967    10000   5020    10000   9973
7 4891  530     5536    0       10000   0
8 9547  0       10000   0       10000   0
```

And an extremely large bar:

```
m: 101 n: 98 Number of games: 100
   2     4       5       6       7       8
2 53    5       67      8       59      1
4 98    53      99      45      98      100
5 33    0       47      0       46      0
6 94    57      100     50      100     100
7 41    2       54      0       100     0
8 99    0       100     0       100     0
```

From these results we can at least suspect that the strategies perform similarly no matter the size of the board. Now let's look at a matrix more closely. For simplicity the results are normalized to percents.

```
m: 3 n: 8
```

```
   2     4       5       6       7       8
2 50    8       64      11      53      20
4 91    44      96      36      88      91
5 37    4       51      0       51      0
6 87    58      100     48      100     88
7 47    12      49      0       100     0
8 86    0       100     0       100     0

m:38 n:24
   2     4       5       6       7       8
2 50    5       63      6       51      5
4 95    50      98      40      95      100
5 37    2       50      0       45      0
6 94    60      100     50      100     100
7 49    5       55      0       100     0
8 95    0       100     0       100     0
```

The first thing that catches our eye is the diagonal. especially the 44 at (4, 4), which seems to converge to 50 when the bar gets bigger. The 100 at (7, 7) and the 0 at (8, 8) are easy to explain. The 0 is because the game always ends in two moves, and the 100 is because there is an even amount of squares. Just look at these results:

```
m: 9 n: 11 Number of games: 1000
   2     4       5       6       7       8
2 494   49      615     50      504     116
4 948   498     976     395     931     972
5 371   19      503     0       458     0
6 938   589     1000    503     1000    969
7 496   69      542     0       0       0
8 892   0       1000    0       1000    0
```

In this case there is an odd amount of squares and at (7, 7) there is a 0, as expected. Another thing that is curious is that strategy 6 performs worse than strategy 4 when matched to the random strategy, while the overall performance of strategy 6 seems better. We also see that for bigger boards, this isn't the case anymore. Now instead of extrapolating all sorts of values, lets try to combine these results to something more understandable:

```
   2     4       5       6       7       8
2 X     --      +       --      +-      --
4 ++    X       ++      -       ++      ++
5 -     --      X       --      +-      --
6 ++    +       ++      X       ++      ++
7 +-    --      +-      --      X       --
8 ++    --      ++      --      ++      X
```

Here we have omitted the data on the diagonal and indicated a significant advantage with + and a huge advantage with ++, the +- means about equal. The rest is obvious.

6 Conclusion.

From the findings above we may conclude strategy 6 is by far the best, number 4 isn't bad either and strategy 5 is plain bad (except in combination with 4). Strategy 7 is also bad, presumably because it stretches the game to such a length that another strategy gets the chance to finish. Strategy 8 may seem to perform quite good on some occasions, but it isn't stable enough. Because the matrix above is anti-symmetrical we may assume that in general no strategy benefits from being the first to move.

7 References.

For the assignment itself visit: http://www.liacs.nl/ kosters/AI/chomp.html.

8 Appendix.

chomp.cc:

```
  #include <iostream.h> // cout, cin
  #include "chomp.h"

  // Element for the stack.
  struct stackelement{
    bool **b;
    stackelement *next;
  };//stackelement

  // Global variables.
  stackelement *top = NULL;
  bool **bar = NULL;
  int m_max = 0,
      n_max = 0;

  /* Private functions. */
  // Prototypes.
  inline bool **newbar(void);          // Allocate memory for a bar.
  inline void freebar(bool **),        // Free memory from a bar.
              cpbar(bool **, bool **); // Copy one bar to another.

  // Allocate memory for a bar.
  inline bool **newbar(void) {
```

```
  bool **b = new bool *[m_max];

  for (int i = 0; i < m_max; i++)
    b[i] = new bool [n_max];
  return b;
}//newbar

// Free memory from a bar.
inline void freebar(bool **b) {
  for (int i = 0; i < m_max; i++)
    delete b[i];
  delete b;
}//freebar

// Copy one bar to another.
inline void cpbar(bool **b1, bool **b2){
  for (int i = 0; i < m_max; i++)
    for (int j = 0; j < n_max; j++)
      b1[i][j] = b2[i][j];
}//cpbar

/* Public functions. */
// Reset the bar.
void resetbar(void) {
  for (int i = 0; i < m_max; i++)
    for (int j = 0; j < n_max; j++)
      bar[i][j] = true;
}//resetbar

// Initialize the bar.
bool makebar(int m, int n){
  if ((m < 2) || (n < 2))
    return false;

  m_max = m;
  n_max = n;
  bar = newbar();
  resetbar();
  return true;
}//makebar

// Throw away the bar.
void delbar(void){
  freebar(bar);
}//delbar
```

```
// Print the bar.
void printbar(void) {
  for (int j = 0; j < n_max && bar[0][j]; j++) {
    for (int i = 0; i < m_max && bar[i][j]; i++)
      cout << "X";
    cout << endl;
  }//for
}//printbar

// Take a bite from the bar.
bool bite(int i, int j) {
  if (!((i < m_max) && (j < n_max) && (bar[i][j]) && (i + j != 0)))
    return false;

  for (int x = i; x < m_max && bar[x][0]; x++)
    for (int y = j; y < n_max && bar[x][y]; y++)
      bar[x][y] = false;
  return true;
}//bite

// Determine if the game is ended.
bool end(void) {
  if (bar[1][0] || bar[0][1])
    return false;
  return true;
}//end

// Determine the left border.
int leftborder(int j) {
  int i = 0;

  while ((i < m_max) && (bar[i][j]))
    i++;
  return i;
}//leftborder

// Determine the lower border.
int lowerborder(int i) {
  int j = 0;

  while ((j < n_max) && (bar[i][j]))
    j++;
  return j;
}//lowerborder

// Push the bar on the stack.
```

```
  void push(void) {
    stackelement *element = new stackelement;

    element->next = top;
    top = element;
    top->b = newbar();
    cpbar(top->b, bar);
  }//push

  // Throw away the top of the stack.
  void pop(void) {
    stackelement *element = top;

    top = top->next;
    freebar(element->b);
    delete element;
  }//pop

  // Copy the top of the stack to the bar.
  void restore(void) {
    cpbar(bar, top->b);
  }//restore

strategies.cc:
  #include <iostream.h> // cout, cin
  #include "misc.h"
  #include "chomp.h"
  #include "strategies.h"


  /* Private functions. */
  // Prototypes.
  inline void randommove(int, int, int, int, int); // Do a random move.
  inline bool directmove(int, int, int); // Do a direct move.
  void dontreduceto(int, int, int); // Try not to leave an (n x n), (2 x n) or
                                    //  (n x 2) bar.
  bool brutemove(int, int),     // Recursively try all moves.
       trivials(int, int, int); // Handle the (n x n), (2 x n) and (n x 2) cases.

  // Do a random move.
  inline void randommove(int player, int i_max, int j_max) {
    int i = 0,
        j = 0;

    while (!bite(i, j)) {
      i = rand() % i_max;
```

```
    j = rand() % j_max;
  }//while
  cout << "Player" << player << " moves: " << i << ' ' << j << endl;
}//randommove

// Recursively try all moves.
bool brutemove(int player, int depth) {
  int i_max = leftborder(0),
      j_max = 0;

  if (end())
    return false;

  for (int i = 0; i < i_max; i++) {
    j_max = lowerborder(i);
    for (int j = 0; j < j_max; j++) {
      push();
      if (bite(i, j)) {
        if (!brutemove(player, depth + 1)) {
          if (depth)
            restore();
          else
            cout << "Player" << player << " moves: " << i << ' ' << j << endl;
          pop();
          return true;
        }//if
        restore();
      }//if
      pop();
    }//for
  }//for
  return false;
}//brutemove

// Do a direct move.
inline bool directmove(int player, int i, int j) {
  bite(i, j);
  cout << "Player" << player << " moves: " << i << ' ' << j << endl;
  return true;
}//directmove

// Handle the (n x n), (2 x n) and (n x 2) cases.
bool trivials(int player, int i_max, int j_max) {
  int i1_max = leftborder(1),
      j1_max = lowerborder(1);
```

```
  if (i1_max > 1) {                   // The (n x n) heuristic.
    if (i_max == j_max)
      return directmove(player, 1, 1);
  }//if
  else {
    if (i_max < j_max)
      return directmove(player, 0, i_max);
    else if (i_max > j_max)
      return directmove(player, j_max, 0);
  }//else
  if (i_max == 2) {                   // The (2 x n) heuristic.
    if (j1_max == j_max)
      return directmove(player, 1, j_max - 1);
    else if (j1_max + 1 < j_max)
      return directmove(player, 0, j1_max + 1);
  }//if
  if (j_max == 2) {                   // The (n x 2) heuristic.
    if (i1_max == i_max)
      return directmove(player, i_max - 1, 1);
    else if (i1_max + 1 < i_max)
      return directmove(player, i1_max + 1, 0);
  }//if
  return false;
}//trivials

// Try not to leave an (n x n), (2 x n) or (n x 2) bar.
void dontreduceto(int player, int i_max, int j_max) {
  bool loop = true;
  int it_max = 0,
      jt_max = 0,
      mt_max = 0,
      nt_max = 0,
      i = 0,
      j = 0;

  while (loop) {
    i = 0;
    j = 0;
    push();
    while (!bite(i, j)) {
      i = rand() % i_max;
      j = rand() % j_max;
    }//while
    it_max = leftborder(0);
    mt_max = lowerborder(it_max - 1);
    if (it_max != mt_max) {               // We're not leaving a (n x n) bar.
```

```
      jt_max = lowerborder(0);
      if (!((mt_max == 2) && (jt_max == 2) && (i_max > 1))) { // (2 x n).
        nt_max = leftborder(jt_max - 1);
        if (!((nt_max == 2) && (it_max == 2) && (j_max > 1))) // (n x 2).
          if (!((i == 1) && (j == 1)))
            loop = false;
      }//if
    }//if
    if ((i_max < 3) && (j_max < 3))
      loop = false;
    restore();
    pop();
  }//while
  directmove(player, i, j);
}//dontreduceto

/* Public functions. */
// Ask for input.
void imove(int player) {
  int i = 0,
      j = 0;

  while (!bite(i, j)) {
    printbar();
    cout << "Player" << player << " (i j): ";
    cin >> i >> j;
  }//while
}//imove

// Generate a random move.
void rmove(int player) {
  int i_max = leftborder(0),
      j_max = lowerborder(0);

  printbar();
  randommove(player, i_max, j_max);
}//rmove

// Try all moves.
void bmove(int player) {
  int i_max = leftborder(0),
      j_max = lowerborder(0);

  printbar();
  if (!brutemove(player, 0))
    randommove(player, i_max, j_max);
```

```
}//bmove

// Generate move based on the trivials heuristic.
void h1move(int player) {
  int i_max = leftborder(0),
      j_max = lowerborder(0);

  printbar();
  if (!trivials(player, i_max, j_max))
    randommove(player, i_max, j_max);
}//h1move

// Generate move based on the 'don't reduce to' heuristic.
void h2move(int player) {
  int i_max = leftborder(0),
      j_max = lowerborder(0);

  printbar();
  dontreduceto(player, i_max, j_max);
}//h2move

// Generate move based on both heuristics.
void hmove(int player) {
  int i_max = leftborder(0),
      j_max = lowerborder(0);

  printbar();
  if (!trivials(player, i_max, j_max))
    dontreduceto(player, i_max, j_max);
}//hmove

// Always take the left-lower square.
void d1move(int player) {
  int i = leftborder(0) - 1,
      j = lowerborder(i) - 1;

  printbar();
  directmove(player, i, j);
}//d1move

// Always take the biggest bite available.
void d2move(int player) {
  int i_max = leftborder(0);

  printbar();
  if (i_max < 2)
```

```
      directmove(player, 0, 1);
    else
      directmove(player, 1, 0);
  }//d2move

assign1.cc:
  #include <iostream.h> // cout, cin
  #include "chomp.h"
  #include "strategies.h"

  // Global variables.
  void (*func1)(int) = NULL,
       (*func2)(int) = NULL;

  /* Private functions. */
  // Prototypes.
  void chooseboardsize(void),  // Ask for the board size.
       *setplayer(int),        // Ask for the strategy of a player.
       choosestrategies(void), // Ask for the stategies.
       play(void);             // Match two strategies.
  int numberofgames(void);     // Ask for the number of games.

  // Ask for the board size.
  void chooseboardsize(void) {
   int m = 0,
       n = 0;

    while (!makebar(m, n)) {
      cout << "Choose a board size (m n): ";
      cin >> m >> n;
    }//while
  }//chooseboardsize

  // Ask for the strategy of a player.
  void *setplayer(int player) {
    int choice = 0;

    cout << "Choose from the following strategies:" << endl
         << "1: Human input." << endl
         << "2: Random strategy." << endl
         << "3: Brute force strategy." << endl
         << "4: Heuristic strategy 1." << endl
         << "5: Heuristic strategy 2." << endl
         << "6: Combined heuristic strategy." << endl
         << "7: Dumb strategy 1." << endl
         << "8: Dumb strategy 2." << endl
```

```
       << "The strategy for Player" << player << " (1-8): ";
  cin >> choice;
  switch (choice) {
    case 1:
      return imove;
    case 2:
      return rmove;
    case 3:
      return bmove;
    case 4:
      return h1move;
    case 5:
      return h2move;
    case 6:
      return hmove;
    case 7:
      return d1move;
    case 8:
      return d2move;
  }//switch
  return NULL;
}//setplayer

// Ask for the stategies.
void choosestrategies(void) {
  while (!func1)
    func1 = (void (*)(int))setplayer(1);
  while (!func2)
    func2 = (void (*)(int))setplayer(2);
}//choosestrategies

// Ask for the number of games.
int numberofgames(void) {
  int x = 0;

  while (!x) {
    cout << "Choose the number of games (>0): ";
    cin >> x;
  }//while
  return x;
}//numberofgames

// Match two strategies.
void play(void) {
  while (!end()) {
    func1(1);
```

```
      if (end()) {
        cout << "Player1 wins." << endl;
        return;
      }//if
      func2(2);
    }//while
    cout << "Player2 wins." << endl;
  }//play

  // Main.
  int main(void) {
    int x = 0;

    chooseboardsize();
    choosestrategies();
    x = numberofgames();
    for (int i = 0; i < x; i++) {
      play();
      resetbar();
    }//for
    delbar();
    return 0;
  }//main
```

misc.cc:

```
  #include "misc.h"

  /* Public functions. */
  // A pseudo-random number generator.
  unsigned int rand(void) {
    static unsigned long seed = 1;

    seed = (seed * 1103515245) + 12345;
    return ((unsigned int)(seed >> 16) & 2147483647);
  }//rand
```