A genetic algorithm.

Jeroen Laros

June 8, 2003

Contents

1 Introduction.

This article is about an algorithm that tries to find a solution for the following problem: Let us say that we have an n x n board on which we can lay tiles of one particular shape. This shape is either the 1 x 1 tile, the 2 x 2 tile, or the L-shaped tile. We mark the upper-left square as a defining square of the tile. The problem is to find a tiling given a projection. In a projection we look al the number of defining squares on each row or column.

2 Problem.

Let us say we have an n x n board with $n = 5$, L-shaped tiles and the following projections: x: 0 2 0 1 0, y: 1 1 0 1 0. A possible solution is:

```
. X . . .
. X X X .
. . . X X
. X . . .
. X X . .
```

3 Approach.

To solve this problem we use a genetic algorithm which uses elitair selection, for the initiated: a (1, N) algorithm, which means that we select the best individual from the population and make N offsprings. The ',' means that this technique will be used even if the overall fitness decreases. We do allow invalid tiling, but they will have a very poor fitness.

4 Implementation.

We use a dynamic two-dimentional array as a genome, two arrays for the temporary phenotype.

4.1 Details.

The tiling itself is kept simple, we only use the defining square, even if the tiles are bigger than 1 x 1. To make sure the tiling will not be invalid, we look at the neighbourhood of the defining square. For example: an 2 x 2 tile must not have any neighbours, nor can it be on the n-th row or column. The L-shaped tile has the same restrictions, except it may have a neighbour on the upper-left or the lower-right side.

5 Results.

A simple test with a 4 x 4 board and 1 x 1 tiles with a population size of 10 finds a solution in about 30 generations. The same problem with a population size of 200 finds a solution in about 5 generations. Here is a little table:

```
     | 1   2   L
-----------------
10   | 30  272 235
50   | 12  109 91
100  | 9   67  67
500  | 4   21  20
1000 | 3   12  7
```

These are the results of a 4 x 4 board with '1 0 1 0' as x- and y-projection. The first column contains the population size, the second one contains the average (over 400 tests) number of generations needed to find a solution for the 1 x 1 tile, the third column is for the 2 x 2 tile and the fourth column is for the L-shaped tile.
And here is the table for a 6 x 6 board averaged over 10 tests:

```
     | 1      2      L
------------------------
10   | 91     151065 77952
50   | 47     61463  22958
100  | 55     36431  12716
500  | 23     13214  8841
1000 | 22     8752   5649
```

6 References.

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

7 Appendix.

The code in this section is just the important parts. The first part contains the datastructures for the puzzle.

```
#include <stdio.h>   // printf, fscanf
#include <stdlib.h>  // malloc, free
#include <math.h>    // abs
#include "puzzle.h"

// Global variables.
int *projection[2],
    *givenprojection[2],
```

```
    size = 0,
    restrictions = 0;

// Private prototypes.
int restrict(board, int, int);

// Private functions.
// This function is responsible for giving an invalid tiling a severe
// punishment.
int restrict(board b, int i, int j) {
  int ret = 0;

  if (restrictions) {
  if ((i == size - 1) || (j == size - 1))
    return size * size; // The other schemes can not have tiles on the brim.
  if (i) {
    if (j) {
      if (b[i - 1][j - 1])
        ret += size;
      if (b[i][j - 1])
        ret += size;
      if ((b[i + 1][j - 1]) && (restrictions == 1))
        ret += size;
    }//fi
    if (b[i - 1][j])
      ret += size;
    if ((b[i - 1][j + 1]) && (restrictions == 1))
      ret += size;
  }//if
  if (b[i + 1][j])
    ret += size;
  if (b[i][j + 1])
    ret += size;
  if (b[i + 1][j + 1])
    ret += size;
  }//if
  return ret;
}//restrict

// Public functions.
// Allocate memory for a board.
board makeboard(int x) {
  board b = (board)malloc(x * sizeof(bool *));

  for (int i = 0; i < x; i++) {
    b[i] = (bool *)malloc(x * sizeof(bool));
```

```
    for (int j = 0; j < x; j++)
      b[i][j] = (bool)(rand() % 2);
  }//for
  return b;
}//makeboard

// Free the memory used up by the board.
void freeboard(board *b) {
  for (int i = 0; i < size; i++)
    free(b[i]);
  free(b);
}//freeboard

// Initialize the solution datastructures.
void init(int x, int shape) {
  size = x;
  restrictions = shape;
  projection[0] = (int *)malloc(x * sizeof(int));
  projection[1] = (int *)malloc(x * sizeof(int));
  givenprojection[0] = (int *)malloc(x * sizeof(int));
  givenprojection[1] = (int *)malloc(x * sizeof(int));
}//init

// Print a board.
void printboard(board b) {
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++)
      if (b[i][j])
        if (restrictions == 1) {
          printf("X X ");
          j++;
        }//if
        else
          printf("X ");
      else
        if (i) {
          if (b[i - 1][j])
            printf("X ");
          else if (j && b[i - 1][j - 1])
            printf("X ");
          else
            printf(". ");
        }//if
        else
          printf(". ");
    printf("\n");
```

```
  }//for
}//printboard

// Translate a board into a projection set(genotype -> phenotype mapping).
void makeprojections(board b) {
  for (int i = 0; i < size; i++) {
    projection[0][i] = 0;
    for (int j = 0; j < size; j++)
      if (b[i][j]) {
        projection[0][i] += restrict(b, i, j) + 1;
      }//if
  }//for
 for (int i = 0; i < size; i++) {
   projection[1][i] = 0;
   for (int j = 0; j < size; j++)
     if (b[j][i])
       projection[1][i]++;
 }//for
}//makeprojections

// Ask the user for a projection set.
void askprojections(void) {
  for (int i = 0; i < size; i++) {
    printf("Give a number for x-projection[%i]: ", i);
    fscanf(stdin, "%i", &givenprojection[0][i]);
  }//for
  for (int i = 0; i < size; i++) {
    printf("Give a number for y-projection[%i]: ", i);
    fscanf(stdin, "%i", &givenprojection[1][i]);
  }//for
}//askprojections

// Calculate the difference between two projection sets.
int difference(void) {
  int diff = 0;

  for (int i = 0; i < 2; i++)
    for (int j = 0; j < size; j++)
      diff += abs(givenprojection[i][j] - projection[i][j]);
  return diff;
}//difference

// Free the memory used by the solution datastructures.
void destroy(void) {
  free(projection[0]);
  free(projection[1]);
```

```
  free(givenprojection[0]);
  free(givenprojection[1]);
}//destroy
```

This part is the genetic algorithm itself.

```
#include <stdlib.h> // malloc
#include <stdio.h> // printf
#include "puzzle.h"
#include "ga.h"

// Global variables.
board *population = NULL;
int pop_size = 0,
    genome_length = 0,
    board_size = 0,
    best = 0;

// Private prototypes.
void mutate(board),     // Mutate a board (genome).
     cp(board, board);  // Copy one board onto another.
int select(void);       // Select the best board.

// Private functions.
// Mutate a board. Each bit on the board has the propability of 1 / gen_length
// to get flipped.
void mutate(board b) {
  for (int i = 0; i < board_size; i++)
    for (int j = 0; j < board_size; j++)
      if (!(rand() % genome_length)) {
        if (b[i][j])
          b[i][j] = false;
        else
          b[i][j] = true;
      }//if
}//mutate

// Select the best individual from the population and return it's fitness.
int select(void) {
  int result = genome_length,
      temp = 0;

  for (int i = 0; i < pop_size; i++) {
    makeprojections(population[i]);
    temp = difference();
    if (temp < result) {
      result = temp;
```

```
      best = i;
    }//if
  }//for
  return result;
}//select

// Copy one board (genome) onto another.
void cp(board out, board in) {
  for (int i = 0; i < board_size; i++)
    for (int j = 0; j < board_size; j++)
      out[i][j] = in[i][j];
}//cp

// Public functions.
// Allocate memory for the population.
void init_pop(int p_size, int b_size) {
  population = (board *)malloc(p_size * sizeof(board));
  pop_size = p_size;
  board_size = b_size;
  genome_length = b_size * b_size;

  for (int i = 0; i < p_size; i++)
    population[i] = makeboard(b_size);
}//init_pop

// Keep repeating the selection and mutation process until the fitness can
// not be improved any more.
void evolve(void) {
  int progress = genome_length,
      trace = 0;

  while (progress) {
    for (int i = 0; i < pop_size; i++)
      mutate(population[i]);
    progress = select();
    for (int j = 0; j < pop_size; j++)      // Elitair selection.
      cp(population[j], population[best]);
    trace++;
  }//while
  printf("Solution found after: %i generations.\n", trace);
  printboard(population[best]);
}//evolve

// Free the memory used by the population.
void destroy_pop(void) {
  for (int i = 0; i < pop_size; i++)
```

```
    free(population[i]);
  free(population);
}//destroy_pop
```