Helsinki University of Technology
Introduction to Knowlegde Engineering

A genetic window-placement algorithm

Bernhard M.J. Leiner, Stud.ID.: 53418L
bleiner@gmail.com

May 5, 2005

A lot of window managers for X11 have an intelligent window placement option. This basically means, that the window manager looks for a good place for a new window. The used fitness function to decide this is usually based on the caused overlap with already existing windows. It's quite easy to figure out a brute force algorithm wich tries $O(n^2)$ different places to find the one with the lowest overlap. This is ok for practical use since people tend to have less than about 10 windows on the same screen.

Instead of using a brute force method, one can look at this problem as an informed search and exploration. Genetic algorithms are usable for this class of problems. This paper discusses the implementation of such an algorithm and the gained results.

Contents

1. Introduction

The insperation for the algorithm described later on in this paper came from practical work with different window managers for the X11 window system. There are literally hundreds of different window managers available. One of the basic features of such a window manager is to map a newly created window to a place on the screen. From a certain point of view the best place would be the one that causes the least overlap with already existing windows on the screen. In practice the best placement option in terms of usability is usually not always the one to minimize overlap. Some applications remember their last position or demand always a certain placement. Errors, warnings, and dialog boxes should be mapped to the center of the screen to catch the attention of the user immediately and so on.

As a matter of this reasons most[1] of the available window managers that have some kind of *intelligent window placement* option, don't always try to minimize the overlap. This report ignores this practical issues and concentrates on finding the solution to the following mathematical problem:

> Given a bounding box that contains a number of rectangles with various sizes. Find a place for a new rectangle inside the bounding box that causes the least overlap with the already existing rectangles.

Nevertheless some of the constraints of the initial problem will still be considered. For example, valid coordinates are always pairs of integers since the resolution of the placement is limited by the resolution of the screen. *Screen* or *root window* will be used insted of the term bounding box and *window* for the rectangles inside the bounding box. The coordinate origin of the screen is — like in X11 — in the upper left corner and similar to that, windows are defined by their upper left corner, width and height.

Let's look at one approach to solve this problem: The basic idea is, that the newly mapped window is always aligned to two other windows to use the existing free place optimally without causing overlap. So each existing window adds two promising looking x-values. One to align the new window on it's left side and one on the right. The same happens for the y-value. This results in $2n$ possibilities for the x and y coordinate or $4n^2$ possibilities for a coordinate pair. This idea can be a bit optimized but the algorithm will always be in $O(n^2)$.

On the other side, the problem looks quite simple for a human eye. We have the capability to immidiately spot good places for the new window. From an AI point of view the whole the problem can be seen as an informed search and exploration problem. A class of algorithms to solve such problems are the so called *genetic algorithms* and are introduced in [1]. Such an algorithm can simulate the human capability to sort out bad looking x or y-values soon. A drawback is, that there is not guarantee that it will find the best solution but the results, as presented in section 3, are quite good and the algorithm runs in $O(n)$.

[1] at least all the author has personal experiences with

2. Implementation

The implementation of the algortihm follows the one introduced in the book except that there are no kinds of mutations. It was written in C and can be summarized as follows:

```
do {
    if (no generation yet) {
        create initial generation
    } else {
        create new generation (offsprings of the current one)
    }
    calculate the fitness of the population
    update the current best coordinate
} while (best overlap != 0 or maximum number of iterations)
place the new window on the best coordinate found
```

The source code for this main loop is shown in listing 1.

create initial generation (listing 2): The initial generation consist of $2n + 2$ coordinate pairs. Each of the n windows is used to calcualte 2 x and 2 y-values. On x-value to align the new window on the left and one to align it on the right. The computation of the y-values is analog but now the alignment is done according to the top and the bottom.

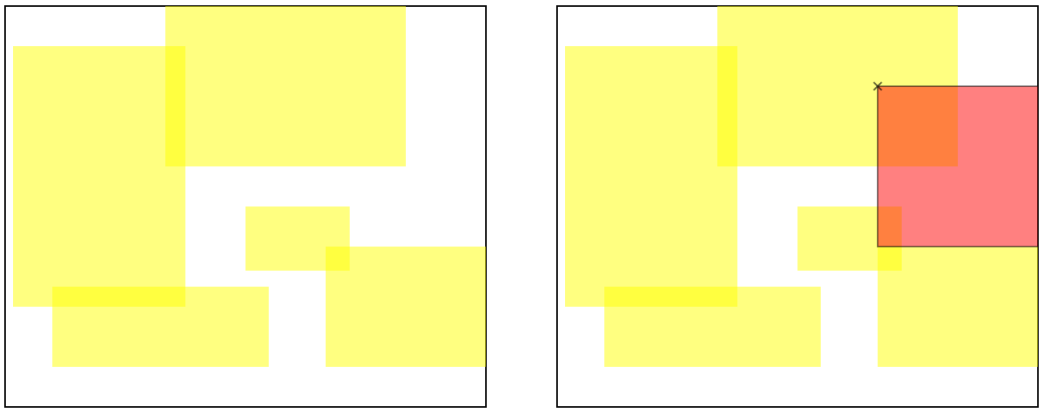
Additionally 2 other coordinate pairs are created. One of them is $(0, 0)$ for the most left x and the uppest y-value and the second one does the same for the ride side and bottom of the screen.

calculate the fitness (listing 3): The most natural way to calculate a fitness function for each coordinate would be the resulting overlap if the new window is mapped on it. Since 0 is the best according to this calcualation, the resulting value has to be inverted somehow to ensure that a *high* fitness value is good. This is done by subtracting the caused overlap area from the whole window area. The best value is now the area of the new window.

Furthermore there are some ad hoc measures in the code to ensure that no fitness value is negativ or 0. This means that also very bad choices have small chance to reproduce.

create new generation (part of listing 1 and listing 4): The new generation that eveloves from the old one has the same number of individuals $(2n + 2)$. The new coordinates are a combination of x and y values randomly choosen from individuals of the old generation[2] . Of course the function to choose a parent doesn't do that completly randomly but takes the fitness of all possible parents into account.

[2]This means that each new individual has in general two parents but it's also possible that two times the same parent was choosen and the child is an exact clone of an individual of the old generation.



(a) (b)

Figure 1: Example setting for the first round of tests. (a) shows the placement of the windows already on the screen (b) the best solution to map the new window onto the screen

update the current best (part of listing 1): An important fact in the whole problem is that it's impossible to know in advance how good the best solution will be. It is also not guaranteed that each new generation contains an individual that is at least as fit as the fitest individual of the former generations. As a matter of fact the algortihm keeps track of the best coordinate found so far.

The main loop for the genetic algortihm immediately terminates if a solution was found that causes 0 overlap with other windows or if a certain numbers of iteration has taken place. The placement of the new window will be done according to the best solution found during all iterations.

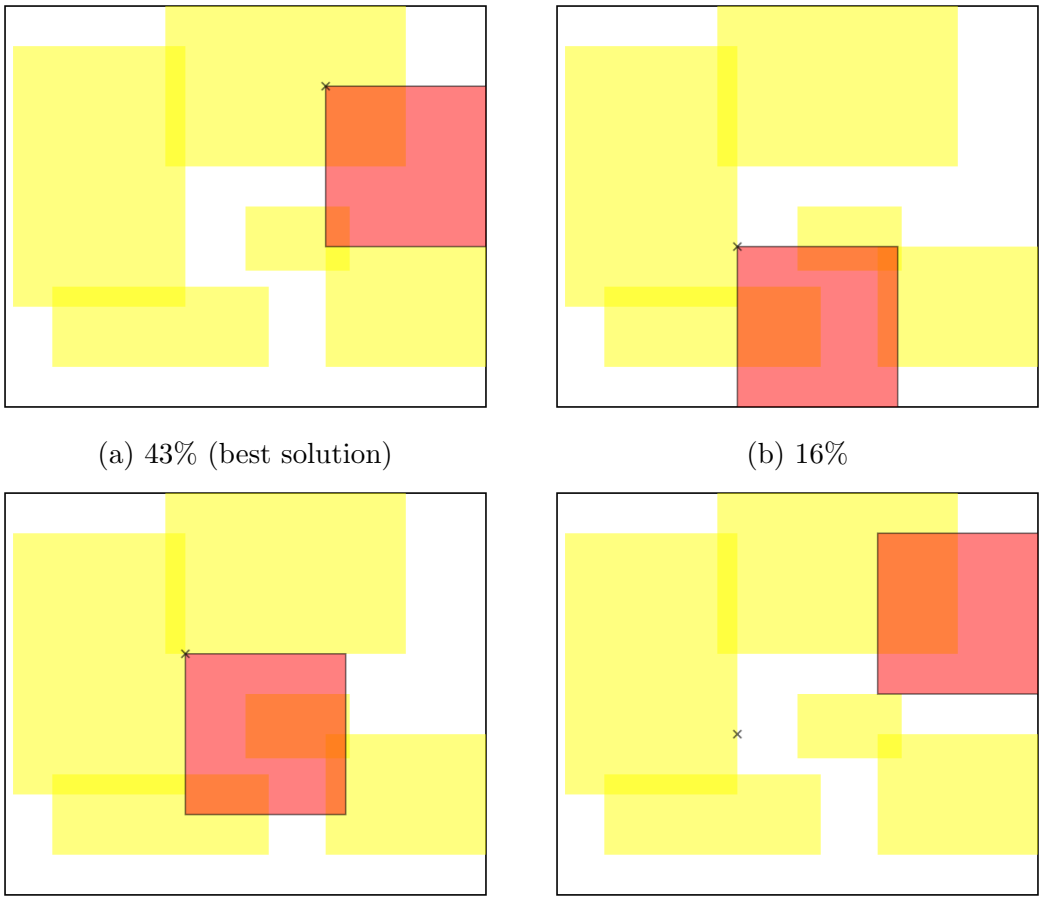
3. Results

The results were gained through 2 distinct test settings. First with just a single constructed configuration. Second with a large number of random settings.

3.1. Single test configuration

This single test configuration is shown in figure 1. It shows a picture of an example setting and the best solution via brute force.

Figure 2 shows the 4 most probable results of the genetic algorithm gained from 100 test runs. It indicates that the probability to find the best result is about 40 percent. Since there is no solution with zero overlap the genetic algorithm never terminates early, but always runs the maximal number of allowed iterations. The results in figure 2 where created with an maximal number of iterations of 5.



(a) 43% (best solution)

(b) 16%

(c) 12%

(d) 11%

Figure 2: The four most probable results of the genetic algorithm with their according probability. An interesting fact is that both solution (b) and (c) are sligthly worse than solution (d). They are prefered due to the initial generation.

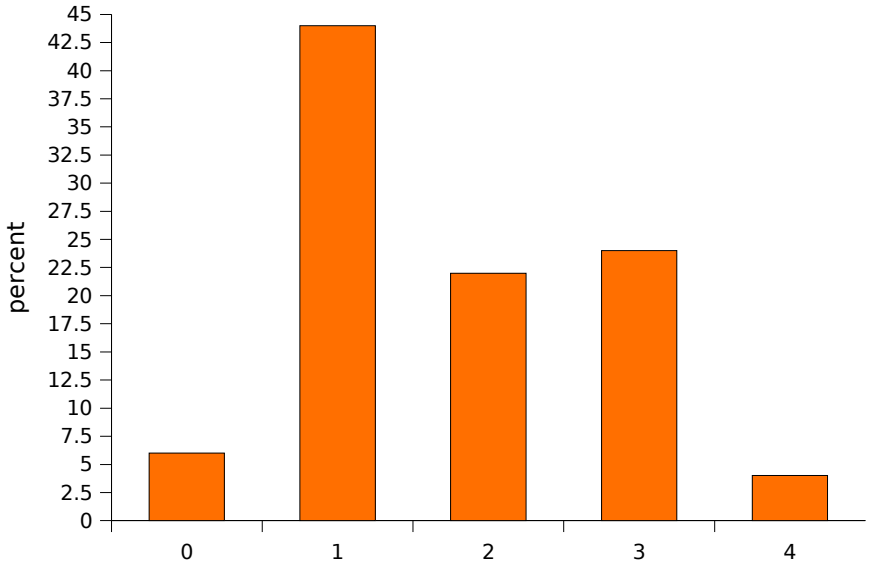


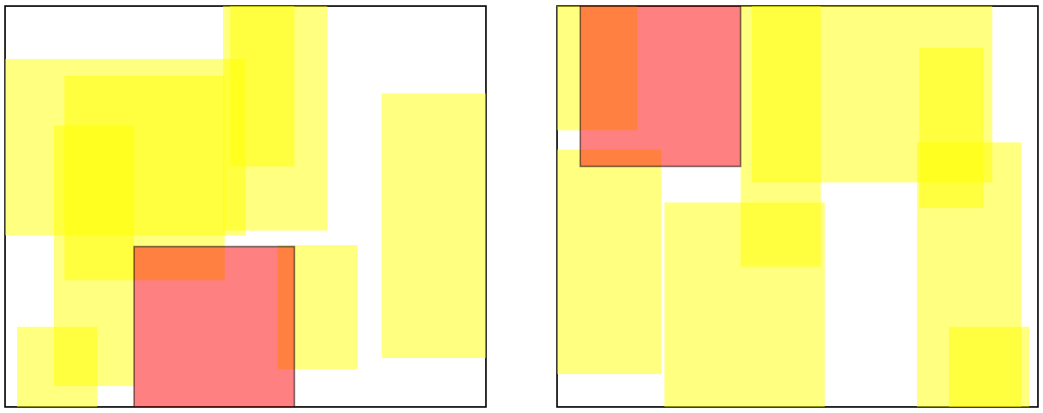
Figure 3: This figure shows that an increase in the number of generations would not result in much better results. The probability that generation 4 contains a fitter subject as the fittest subject of all older generations is already quite small.

An interesting question is if a higher number of iteration would give better results. This is not really the case. Figure 3 shows that after the first three iterations the algorithm seldomly finds a new better solution. It even happens sometimes (about 6% probability) in this configuarion that the best result is already found in the initial generation.

3.2. More random tests

To get a more general overview about the performance of the genetic placement algorithm a second round of test were done. This time with random starting configurations but still fixed windows sizes.

Complete randomization was also tried but lead to problems since the window distribution tends to center in the middle of the root window. This can be explained with the following example: Lets suppose the width of the window is a few pixels more the half of the screen width. No matter if this window is arranged on the very left side or on the very right side of the screen, it will always use space in the center. For smaller windows this is not guaranteed but the situation is somehow the same. Even if their position is equally distributed in x and y coordinate values the probability that they cover some area in the center is higher than on the borders of the screen. As a matter of fact rather small windows have been used in the following test and the distribution is not completly random. Some windows are enforced to be placed aligned to the root window borders.



(a) (b)

Figure 4: Two typical examples of pseudorandom initial window placement and mapping the new window with the genetic algorithm.

Listing 5 shows a part of the perl script that achieves that.

The tests have been done with 100 different starting configurations and each of those was run 100 times. The configuration file `wins_rand` was used as starting point for defining the sizes of the windows. Their position was randomized as described above. Two typical results are shown in figure 4.

Some results from this test runs:

- Since the genetic placement algorithm uses random selection the outcome is not deterministic. As already indicated in figure 2 there are usually several possible outcomes. But how much? The result ranges from a single one, usually caused by a perfect result with zero overlap in the initial generation, until 12 different results. The average was 2.66 different solutions for one configuration and 100 runs.

- The performance of the genetic algorithm during this 10000 runs is on average only 13% worse the the optimal solution via the brute force search. Figure 5 shows an overview. Usually the best solution is found with a high probability. The algorithm performs bad if there are several alternatives which are only slightly worse than the best. An example for such a case is shown in figure 6. What causes the bad performance in this case? First, the x value of 0 is really bad if the y value is also low since there are multiple overlapping windows in the upper left corner. So the low x value has bad chances to propagate. Furthermore there are two not too bad places more on the right.

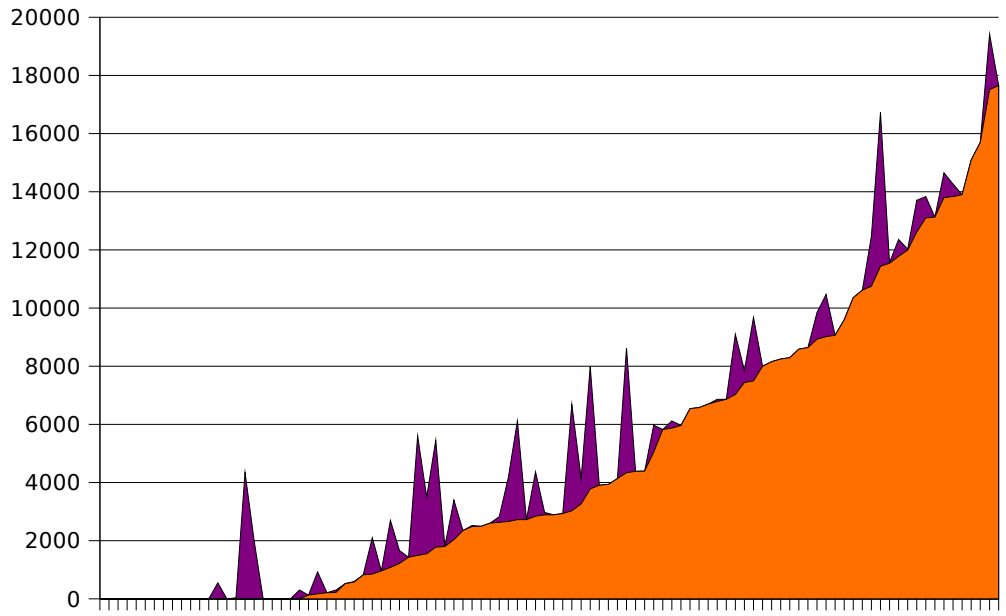


Figure 5: 100 different initial window configurations sorted ascending to the overlap of the best solution. The peaks indicate cases where the genetic algortihm can't find the best solution every time and the average resulting overlap is therefore higher than the best solution.

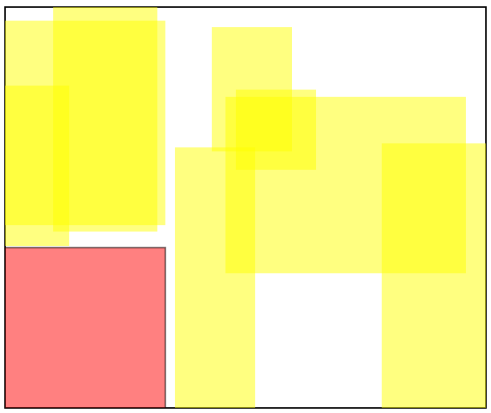


Figure 6: This is the configuration that causes the first significant peak in the graph in figure 5. Zero overlap is possible, as shown here, but the average perfomance is not as good.

References

[1] Stuart Russell and Peter Norwig *Artificial Intelligence - A Modern Approach* Prentice Hall, second edition 2003

[2] Cairo vector graphics library,
http://cairographics.org/

[3] wmii - Window manager improved,
http://wmi.modprobe.de/

A. Notes on the implementation

First of all I would like to mention some principial design issues. The whole project was written in C. This was done since the very original idea was to use the resulting intelligent window placement algorithm in a real existing X11 window manager which would have been *wmii* since I contributed a few lines of code to this project and I'm already familiar with it. And *wmii* [3] is written in C.

A.1. Software architecture

Figure 7 gives an overview over the software architecture. The *aiwin* modul (`aiwin.c`) reads a configuration file which contains the size of the screen, all existing windows (size and placement) and the size of the new window. The perl script `wincreator.pl` or `shuffle_wins.pl` in the `scripts` subdirectory can be used to create new random configurations.

The *placement* modul handels the calcualation of the placement of the new window. I have written 3 different algorithms: the genetic algorithm (`genetic.c`) which was described in the above sections, a brute force algorithm (`brute_force.c`) which tries all possible coordinates and therefore definitely finds the best solution and one that is a variation of the possible soultion that was mentioned in the Introduction of this report (`upper_left.c`).

utils (`utils.c`) contains various function to calculate overlaps between rectangles, sorting results and so on.

To display the results the *drawing* module can be used. It uses the cario vector graphics library [2] and supports output on the screen or into a PNG file.

A.2. Installation and running

The project was done parallel in a Linux (Debian) and BSD (OpenBSD) environment and has no special dependencies except the cairo graphics library (which is a compile time option). Note that `qsort()` is used, which is a addition to the standard C library. One thing that restricts the program to an Posix environment is that reading from `/dev/urandom` is used to get a seed for the random number generator.

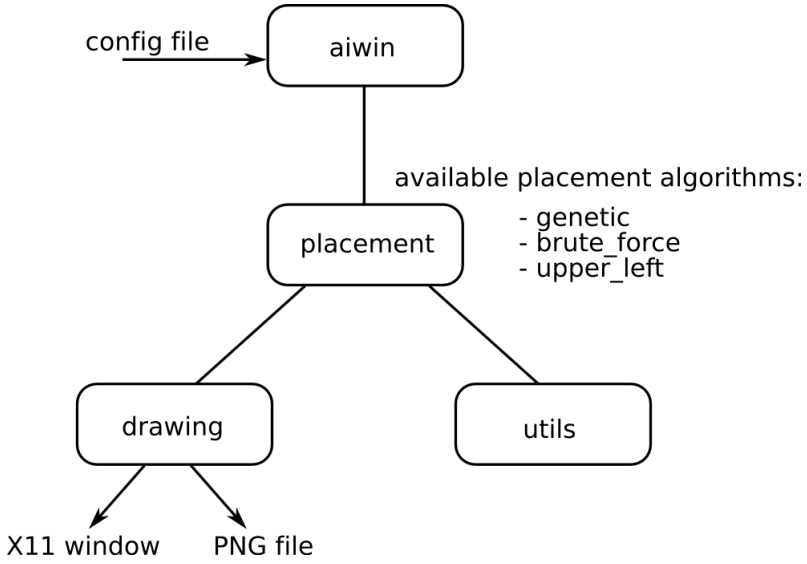


Figure 7: overal software architecture

Installation: Unpacking the file `aiwin.tgz` creates the directory `aiwin` that contains all source code files. A simple `make` should produce 3 executables. One for each placement algorithm. There is not install target in the Makefile.

Note that the default is to compile without cairo support. To activate it just uncomment the first two lines in the `Makefile`. Depending on he environment the include path might have to be changed.

The directory `report` contains all the pictures and LaTeX source files used to created this report. `scripts` contains a number of Perl scripts, shell scripts, configuration files, log files used for testing and analysing the performance. They are undocumented and just hacked together somehow!

Running: Just run one of the executables with a configuration file a first argument. There is an example configuration file `wins` that defines some windows. To use this configuration and the genetic placement algorithm: `./genetic wins`.

If cairo support is compiled in, additional to some output on stdout a window will appear to show the results. *This is currently broken*[3]! To get the graphic output into a PNG file just add a filename as second parameter: `./genetic wins out.png`.

A.3. Code listings

This section contains listings of important parts of the source code of the genetic placement algorithm.

[3]Due to a lack of time (and interest) I couldn't fix that for this version. Please just use the PNG output backend.

Listing 1: the main loop of the genetic algorithm (`genetic.c`)

```
/* start with the genetic algorithm */
do {
    if (generation_count == 0) {
        /* create initial generation */
        new_population (wins, num_w, c, num_c);
    } else {
        /* create new generation (offsprings of the current one) */
        printf("create new generation ...\n");
        for (j = 0; j < num_c; j++) {
            new_pop[j].x = select_random (c, num_c, 'x');
            new_pop[j].y = select_random (c, num_c, 'y');
        }
        replace_old_generation (c, new_pop, num_c, gendata);
    }

    /* calculate the fitness of the population */
    printf("calculate the fitness of generation %d..\n", generation_count);
    calc_fitness(wins, num_w, c, num_c);

    printf("best overlap so far: %d\n", best.overlap_area);
    /* update the current best coordinate */
    if (compare_overlap (&best, &c[0]) > 0) {
        best.x = c[0].x;
        best.y = c[0].y;
        best.overlap_area = c[0].overlap_area;
        printf("best overlap replaced with: %d\n", best.overlap_area);
    }

    generation_count++;
} while (!(best.overlap_area == 0) &&
        (generation_count < MAX_GENERATIONS));

/* place the new window on the best coordinate found */
wins[num_w - 1].x = best.x;
wins[num_w - 1].y = best.y;
```

Listing 2: create initial generation (`genetic.c`)

```
/**
 * creates a starting population of coordinates
 */
static void new_population (win_t *wins, int num_w, coor_t *c, int num_c)
{
    /**
     * how much starting points (parents) do we need? good question!
     * let's take all possible good x and all possible good y values
     * and use them for creating starting points
     *
     * How many good x values do we have? Try to align with existing
     * windows on the left or the right side -> 2 values for each win
     * + 0 and (width - wins[num_w - 1]) too align with the root window
     * The same is valid for y values
     *
```

```
     * So we have 2 * (num_w - 1) + 2 = 2 * num_w start coordinates
     */
    int i;

    for (i = 0; i < (num_w - 1); i++) {
        c[2*i].x = wins[i].x - wins[num_w - 1].w;
        c[2*i].y = wins[i].y - wins[num_w - 1].h;

        c[2*i + 1].x = wins[i].x + wins[i].w;
        c[2*i + 1].y = wins[i].y + wins[i].h;
    }

    adjust_coor(c, 2 * (num_w - 1), wins[num_w - 1].w, wins[num_w - 1].h,
                width, height);

    /* finally add upper left (0,0) and lower right (this is the "+2") */
    c[2*i].x = 0;
    c[2*i].y = 0;
    c[2*i + 1].x = width - wins[num_w - 1].w;
    c[2*i + 1].y = height - wins[num_w - 1].h;
}
```

Listing 3: calculate the fitness (`genetic.c`)

```
/**
 * calculates the fitness and the threshold for each coordinate in the
 * array also sorts the coordinates according to their fitness
 * (overlap value)
 */
static void calc_fitness (win_t *wins, int num_w, coor_t *c, int num_c)
{
    int i;
    long fitness_sum = 0;
    gendata_t *gen_dat;
    calc_overlaps (wins, num_w, c, num_c);
    /* invoke qsort (GNU addition to the standard C library) */
    qsort (c, num_c, sizeof (coor_t), compare_overlap);
    printf("(%d,%d): %d\n", c[0].x, c[0].y, c[0].overlap_area);

    /* define the fitness as: window_area - overlap_area */
    for (i = 0; i < num_c; i++) {
        gen_dat = (gendata_t *)c[i].data;
        gen_dat->fitness = win_area - c[i].overlap_area;

        /* prevent negativ fitness value and keep a small chance for
         * very bad coordinates to reproduce. */
        if (gen_dat->fitness < 0) {
            gen_dat->fitness = 0.05 * win_area;
        } else if (gen_dat->fitness < 0.1 * win_area) {
            gen_dat->fitness = 0.1 * win_area;
        }

        gen_dat->threshold = fitness_sum;
        fitness_sum += gen_dat->fitness;
```

```
    }

    /* the (g)libc random number generator is a bit unflexible ... convert
     * the integer threshold into a double from 0 to 1 */
    for (i = 0; i < num_c; i++) {
        gen_dat = (gendata_t *)c[i].data;
        gen_dat->threshold_d = (double) gen_dat->threshold /
                               (double) fitness_sum;
    }
}
```

Listing 4: choose a random parent (`genetic.c`)

```
/**
 * selects a random member of the current population (by taking the
 * thresholds introduced by its fitness into account) and returns either
 * its x or y value (according to the parameter select)
 */
static int select_random (coor_t* c, int num_c, char select)
{
    double random = drand48 ();
    int i;
    int return_num = -1;
    gendata_t *gen_dat_low;
    gendata_t *gen_dat_high;

    for (i = 0; i < (num_c - 1); i++) {
        gen_dat_low = (gendata_t *)c[i].data;
        gen_dat_high = (gendata_t *)c[i + 1].data;
        if ( (random >= gen_dat_low->threshold_d) &&
             (random < gen_dat_high->threshold_d) ) {
            return_num = i;
            break;
        }
    }
    if (return_num == -1)
        return_num = num_c - 1;

    switch (select) {
        case 'x':
            return c[return_num].x;
            break;
        case 'y':
            return c[return_num].y;
            break;
    }
    return 0; /* prevent compiler warning */
}
```

Listing 5: random windowplacemnt with some restrictions (`shuffle_wins.pl`)

```
for (my $i = 1; $i <= 100; $i++)
{
    my $filename = "wins_" . "$i";
```

```
    open(FILE, ">$filename") or die "error opening file!\n";
    printf FILE "$width $height\n";
    foreach (@wins)
    {
        my $w = $_->{'w'};
        my $h = $_->{'h'};

        # force 40 % of the windows to be aligned on the
        # left or right border (20% left, 20% right)
        my $rand_tmp = int (rand (100));
        if ($rand_tmp < 20) {
            $rand_x = 0;                            # left
        } elsif ($rand_tmp < 40)
        {
            $rand_x = $width - $w;                  # right
        } else
        {
            $rand_x = int (rand ($width - $w));     # random
        }

        # force 40 % to be aligned on the top or bottom
        $rand_tmp = int (rand (100));
        if ($rand_tmp < 20) {
            $rand_y = 0;                            # top
        } elsif ($rand_tmp < 40)
        {
            $rand_y = $height - $h;                 # bottom
        } else
        {
            $rand_y = int (rand ($height - $h));    # random
        }

        printf FILE "$rand_x $rand_y $w $h\n";
    }
    close FILE;
}
```

B. Feedback

I can't comment on the course since I never attended it. En puhun suomea. (I hope this is correct)

The assignement was quite interesting. Not only programming the genetic algorithm but especially running the test and looking at the sometimes strange results and trying to find a reson for the behaviour. As you might have noticed I used this assignment also to get familiar with cairo. This has of course nothing to do with knowledge engineering but creating fancy graphics is always funny.

A thing that I would to different the next time: Don't use C as main programming language! This whole assignement would have been a good opportunity to learn the basics of a new programming language (I still don't know C# or Phyton).

Final remark: The program was written in a way to make it possible for *me* to get results. It's not user friendly and it's really easy to crash everything by introducing syntax errors into configuration files or some thing like that. Sorry for that.