Declaratively solving tricky Google Code Jam problems with Prolog-based ECLiPSe CLP system

Sergii Dymchenko
Independent Researcher
sdymchenko@progopedia.com

Mariia Mykhailova
Independent Researcher
michaylova@gmail.com

ABSTRACT

In this paper we demonstrate several examples of solving challenging algorithmic problems from the Google Code Jam programming contest with the Prolog-based ECL^{i}PS^e system using declarative techniques: constraint logic programming and linear (integer) programming. These problems were designed to be solved by inventing clever algorithms and efficiently implementing them in a conventional imperative programming language, but we present relatively simple declarative programs in ECL^{i}PS^e that are fast enough to find answers within the time limit imposed by the contest rules. We claim that declarative programming with ECL^{i}PS^e is better suited for solving certain common kinds of programming problems offered in Google Code Jam than imperative programming. We show this by comparing the mental steps required to come up with both kinds of solutions.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*; G.1.6 [**Numerical Analysis**]: Optimization—*Linear programming*

Keywords

Declarative programming, logic programming, constraint programming, linear programming, programming competitions

1. INTRODUCTION

Google Code Jam[1] (GCJ) is one of the biggest programming competitions in the world: almost 50,000 participants registered in 2014, and 25,462 of them solved at least one task.

For good results, competitors must think and code quickly – an individual round is usually 2–4 hours long and poses 3 or more problems. A solution is considered correct if it produces correct answers for all given test cases within a certain time limit (4 minutes for the "small" input and 8 minutes for the "large" one).

GCJ competitors can use any freely available programming language or system (including the ECL^{i}PS^e[2] system described in this paper). Most other competitions restrict participants to a limited set of popular programming languages (typically C++, Java, C#, Python). Many contestants participate not only in GCJ, but also in other contests, and use the same language across all competitions. When the GCJ problem setters design the problems and evaluate their complexity, they keep in mind mostly this crowd of seasoned algorithmists.

We show that some GCJ problems that should be challenging according to the problem setters' estimate can be easy or even trivial to solve declaratively using high-level programming languages like ECL^{i}PS^e and techniques like constraint logic programming and linear (integer) programming.

ECL^{i}PS^e CLP[7, 1] is an open-source software system which aims to serve as a platform for integrating various logic programming extensions, in particular constraint logic programming (CLP). ECL^{i}PS^e has a modern and efficient implementation of the Prolog programming language in its core, offers some Prolog extensions (like declarative loops [6]), contains several constraint programming libraries ('ic' for interval arithmetic, 'fd' for finite domains, 'grasper' for graphs), and interfaces to third-party solvers (Gecode, Gurobi, COIN-OR solvers, CPLEX) [3]

ECL^{i}PS^e implementation of TPK algorithm

We do not assume that the reader is familiar with ECL^{i}PS^e, so to give a feeling of the syntax we present and explain a program for a TPK algorithm.

TPK is a simple algorithm proposed by D. E. Knuth and L. T. Pardo [4]. It is used to demonstrate some basic syntactic constructs of a programming language beyond the "Hello, World!" program. The algorithm prompts for 11 real numbers $(a_0 \dots a_{10})$ and for each a_i computes $b_i = f(a_i)$, where $f(t) = \sqrt{|t|} + 5t^3$. After that for $i = 10 \dots 0$ (in that order) the algorithm outputs a pair (i, b_i) if $b_i \leq 400$, or (i, TOO LARGE) otherwise.

[1] `https://code.google.com/codejam`

[2] `http://www.eclipseclp.org/`

```
1 f(T, Y) :-
2     Y is sqrt(abs(T)) + 5*T^3.
3 main :-
4     read(As),
5     length(As, N), reverse(As, Rs),
6     ( foreach(Ai, Rs), for(I, N - 1, 0, -1) do
7         Bi is f(Ai),
8         ( Bi > 400 ->
9             printf("%w TOO LARGE\n", I)
10        ;
11            printf("%w %w\n", [I, Bi])
12        )
13    ).
```

Listing 1: TPK algorithm in ECL^{i}PS^e

Lines 1–2 define a predicate to calculate value of f. Lines 3–13 define predicate `main`. The name of this predicate is passed to ECL^{i}PS^e translator as a command-line parameter.

Line 4 reads a list into variable `As`. The input is a Prolog list (comma-delimited and bracket-enclosed). We preprocess the space-separated input with a simple 'sed' script to convert it to the Prolog format.

Line 5 stores the length of the input list in the variable `N` (the program can work with inputs of different lengths, not necessarily 11 as original TPK program), and stores original input numbers in reverse order in the variable `Rs`.

Line 6 is the head of ECL^{i}PS^e loop: we iterate simultaneously over every number in `Rs` and over $I = N - 1 \ldots 0$. `I`, `Ai` and `Bi` are local variables for every loop iteration. Line 7 simply assigns the value `f(Ai)` to `Bi`. This line demonstrates ECL^{i}PS^e support of using arithmetic predicates as functions – in many other Prolog implementations less clear `f(Ai, Bi)` must be used. Lines 8–10 are Prolog 'if-then-else' construct that outputs "TOO LARGE" or `Bi` depending of the value of `Bi`. `printf` is very similar to the corresponding C function. `%w` is a "wildcard" control sequence that can be used to output values of different types. The second argument of `printf` is a value or a list of values for substitution.

Indentation has no syntactic meaning in Prolog (or ECL^{i}PS^e), we use it for clarity.

2. THE PROBLEMS

In this section we show how tricky algorithmic problems can often be easily modeled and efficiently solved using declarative programming techniques and ECL^{i}PS^e. We chose a set of GCJ problems from different tournament stages to demonstrate different useful aspects of ECL^{i}PS^e: constraint programming library 'ic', linear programming library 'eplex', working with integers and floating point numbers.

To compare our declarative programming solutions with possible imperative solutions we compare the "mental steps" required to come up with the solution (counting number of lines of code would not be useful because the main challenge is to invent the solution, not to code it.) Our ECL^{i}PS^e programs solve both large and small inputs for each problem.

Triangle Areas[3]

"Triangle Areas" is a problem from the second round of GCJ 2008. The problem statement can be rephrased as follows: given integer N, M and A, find a triangle with vertices in integer points with coordinates $0 \le x_i \le N$ and $0 \le y_i \le M$ that has an area equal to $\frac{A}{2}$, or say that it does not exist.

"Triangle Areas" is almost perfect for solving with constraint logic programming. Variables are discrete, constraints are non-linear (so integer linear programming can not be used), and we are looking for any feasible solution. The problem is not too hard from purely algorithmic point of view, but correct implementation using conventional programming languages is very tricky. Many contestants who solved all other problems (including higher valued problems) failed to solve the large or both inputs for "Triangle Areas". At the same time, modeling this problem in ECL^{i}PS^e is almost trivial. To come up with an effective model we need to notice that one vertex of the triangle can be chosen arbitrarily. With this observation, the most convenient way to calculate the doubled triangle area is to place one vertex in $(0, 0)$; then $2S = |x_2 y_3 - x_3 y_2|$. (The same formula can be used in an imperative solution.)

For this problem we present complete source code of the solution. For subsequent problems we present only interesting parts: 'model' plus other problem-specific predicates.

```
1  :- lib(ic).
2  model(N, M, A, [X2, Y2, X3, Y3]) :-
3      [X2, X3] :: 0..N,
4      [Y2, Y3] :: 0..M,
5      A #= abs(X2 * Y3 - X3 * Y2).
6  do_case(Case_num, N, M, A) :-
7      printf("Case #%w: ", [Case_num]),
8      ( model(N, M, A, Points), labeling(Points) ->
9          printf("0 0 %w %w %w %w", Points)
10     ;
11         write("IMPOSSIBLE")
12     ),
13     nl.
14 main :-
15     read([C]),
16     ( for(Case_num, 1, C) do
17         read([N, M, A]),
18         do_case(Case_num, N, M, A) ).
```

Listing 2: Complete ECL^{i}PS^e program for the "Triangle Areas" problem

Line 1 loads interval arithmetic constraint programming library 'ic'. Lines 2–5 define the model with input parameters `N`, `M`, `A` and a list of output parameters `[X2, Y2, X3, Y3]`. `::` and `#=` are from 'ic' library. With `::` we define possible domains for `X2, X3, Y2, Y3` variables, and `#=` from 'ic' constraints both left and right parts to be equal and integer. After model evaluation `X2, X3, Y2, Y3` variables will not necessarily be instantiated to concrete values, but they will have reduced domains with possible delayed constraints and will be instantiated later with `labeling`.

Lines 6–13 define the `do_case` predicate to process a single input case. Line 7 outputs case number according to

[3]Problem link: `http://goo.gl/enHWlq`

the problem specification. Lines 8–12 are an 'if-then-else' construct that outputs point coordinates if it is possible to satisfy our `model` predicate and labels (assigns concrete values from the domain) all coordinate variables, or "IMPOSSIBLE" otherwise. Line 13 simply outputs a new line character.

Lines 14–18 define the `main` predicate that reads number of test cases `C` and for each test case reads `N, M, A` parameters and executes `do_case`.

We can formulate following mental steps needed to invent and implement this ECL^{i}PS^e solution:

1. *Notice that one vertex can be chosen as* $(0, 0)$.
2. *Recall or look up the formula for an area of a triangle.*
3. *Formulate the constraint programming model.*
4. *Code the constraint programming model.* For this problem, it requires 4 lines of straightforward code.

Let us compare this with a possible imperative solution in a convenient programming language that requires a more in-depth analysis of the problem. First observations will be the same. We will also note that it is impossible to find required triangle if $A > M \times N$, and for $A = M \times N$ triangle $(0, 0), (N, 0), (0, M)$ is a valid answer. Now, for $A < M \times N$ we can represent A as $M(A \operatorname{div} M) + (A \bmod M)$, $0 < A \operatorname{div} M < N, 0 < A \bmod M < M$. If we match this representation with the area formula, we can see that points $(0, 0), (1, M)$, and $(-A \operatorname{div} M, A \bmod M)$ form a triangle with area $\frac{A}{2}$. If we shift this triangle $A \operatorname{div} M$ units in positive direction along the x axis, we will get a triangle $(A \operatorname{div} M, 0), (A \operatorname{div} M + 1, M), (0, A \bmod M)$ that will match all the requirements.

The mental steps for this solution could be:

1. *Notice that one vertex can be chosen as* $(0, 0)$.
2. *Recall or look up the formula for an area of a triangle.*
3. *Figure out that for* $A > M \times N$ *there is no such triangle.*
4. *Find the solution for border case* $A = M \times N$.
5. *Come up with a representation of* A *that matches the area formula for a special triangle.*
6. *Shift this triangle to fit inside the boundaries.*
7. *Code the solution.* The code is even easier, check how A relates to $M \times N$ and output corresponding triangle.

Arguably, declarative solution in ECL^{i}PS^e needs simpler steps and leaves less space for a possible mistake.

Dancing With the Googlers[4]

"Dancing With the Googlers" is a problem from the qualification round of GCJ 2012. In this problem we consider triplets of integers from 0 to 10 which never contain numbers that are more than 2 apart. A triplet is surprising if it contains numbers that are exactly 2 apart. Given the list of sums of N triplets and the number of surprising triplets among them S, how many triplets can be high (have the highest number at least p)?

[4]Problem link: `http://goo.gl/JpQQYi`

One of the very useful features of ECL^{i}PS^e is that similar (sometimes identical) models can be used to solve the same problem using different solvers (for example, constraint programming 'ic' and linear programming 'eplex' [8]). Linear (integer) programming is almost always much more effective than constraint programming (especially when looking not just for any feasible, but for the optimal solution), but constraint programming has more expressive power because of possible usage of non-linear constraints and objectives. Some problems can be adequately modeled as linear (integer) programming problems, but it may not be obvious how to formulate the model in terms of linear constraints and objective – additional variables and specific linearizing "tricks" might be required [9, 2, 5]. So a useful technique is to solve small input of a GCJ problem using a constraint programming model (easier to formulate, but less effective), and then convert the constraint programming model into a linear programming model to solve large input using linear (integer) programming solver. Correctness of the less obvious linear programming model can be verified by comparing its results with results of the constraint programming model on the same (small) input.

Constraint logic programming model for this particular problem is easy to formulate in ECL^{i}PS^e (based on the fact that logic values of arithmetic constraints – 0 or 1 – can be used in other arithmetic constraints as integers):

```
1  :- lib(ic).
2  :- lib(branch_and_bound).
3  model(S, P, Points, Triplets, GtP) :-
4      length(Points, N),
5      length(Triplets, N),
6      ( foreach(Triplet, Triplets),
7          foreach(Point, Points),
8          fromto(0, SPrev, SCurr, S),
9          fromto(0, GtPPrev, GtPCurr, GtP),
10         param(P) do
11         Triplet = [Min, Med, Max],
12         Triplet :: 0..10,
13         Min #=< Med, Med #=< Max,
14         Max - Min #=< 2,
15         Max + Med + Min #= Point,
16         SCurr #= SPrev + (Max - Min #= 2),
17         GtPCurr #= GtPPrev + (Max >= P ) ).
18 find(Triplets, GtP) :-
19    flatten(Triplets, Vars),
20    Cost #= -GtP,
21    bb_min(labeling(Vars), Cost,
22           bb_options{strategy: dichotomic}).
```

Listing 3: Constraint programming solution for "Dancing With the Googlers"

Lines 1 and 2 load 'ic' constraint programming library and a library for branch-and-bound search [3]. Lines 3–17 define constraint programming model with input parameters S (number of surprising triplets), p, and `Points` (list of triplet sums), and output parameters `Triplets` (list of possible triplets values) and `GtP` (number of high triplets). Lines 4 and 5 make `Triplets` a list of the same length as `Points`.

Lines 6–17 form an ECL^{i}PS^e loop. Lines 6–10 are loop header; `foreach` lines loop over triplets and points simultaneously, and `fromto` lines collect number of possible sur-

prising triplets (constrained to equal S after loop end) and number of possible high triplets (returned as `GtP`). Lines 11–17 are loop body: they provide the representation of a triplet as minimum, medium and maximum elements to simplify calculation of constraint expressions for surprising and high triplets.

Lines 18–22 define the predicate `find` that finds concrete values of `Triplets` and `GtP` using `bb_min` from the `branch_and_bound` library. In line 19 `Triplets` list is flattened to `Var`, because `labeling` works only with flat lists or arrays. `bb_min` minimizes the objective, so in line 20 we define the objective (`Cost`) as negative of (`GtP`) that we want to maximize.

An integer linear programming model is very similar, but finds solution much faster and solves large input.

```
1 :- lib(eplex).
2 model(S, P, Points, Triplets, GtP) :-
3     integers(GtP),
4     length(Points, N),
5     length(Triplets, N),
6     ( foreach(Triplet, Triplets),
7         foreach(Point, Points),
8         fromto(0, SPrev, SCurr, S),
9         fromto(0, GtPPrev, GtPCurr, GtP),
10        param(P) do
11        Triplet = [Min, Med, Max],
12        Triplet $:: 0..10, integers(Triplet),
13        Min $=< Med, Med $=< Max,
14        Max + Med + Min $= Point,
15        Surprise $:: 0..1, integers(Surprise),
16        Max - Min $=< 1 + Surprise,
17        SCurr $= SPrev + Surprise,
18        G $:: 0..1, integers(G),
19        Max $>= G * P,
20        GtPCurr $= GtPPrev + G ).
21 find(GtP) :-
22     eplex_solver_setup(max(GtP)),
23     eplex_solve(_),
24     eplex_var_get(GtP, typed_solution, GtP),
25     eplex_cleanup.
```

Listing 4: Integer programming solution for "Dancing With the Googlers"

This integer linear programming model uses `eplex` library and requires two additional sets of integer variables 0..1 compared to the constraint programming model. `Surprise` and `G` are indicator variables local to loop iteration for "is triplet surprising" and "is triplet high", and their use allows to linearize the constraints. Unlike the `ic` library, `eplex` doesn't deduce that variables must be integer from integer domain bounds, so variables have to be declared as integers explicitly. This solution doesn't require branch-and-bound search, because integer linear programming solver already produces the optimal solution.

Mental steps for the ECL^{i}PS^e solution could be:

1. *Formulate triplet representation and constraints for an individual triplet.*
2. *Code constraint programming model.* At this point we can solve the small input and make sure that our implementation is correct.
3. *Apply linearization tricks to convert non-linear constraints to integer linear.*
4. *Code integer linear programming model.*

For an imperative solution we have to notice that the lowest sum of numbers in a high unsurprising triplet is $3p-2$ (for a triplet $p, p-1, p-1$), and in a high surprising triplet is $3p-4$ (for a triplet $p, p-2, p-2$), but only if $p >= 2$ (otherwise the triplet won't be surprising). After this, we count the triplets which are high even when unsurprising N_{high}, and the triplets which can be high if they are surprising N_{surp}. The answer is $N_{high} + min(N_{surp}, S)$.

Mental steps for this imperative solution could be:

1. *Notice that unsurprising triplet is high if its sum is* $\geq 3p-2$*, and surprising triplet is high if its sum is* $\geq 3p-4$ *and* $p \geq 2$.
2. *Implement iteration over triplets and calculation of* N_{high} *and* N_{surp}.

Compared to our declarative solution, the imperative solution needs fewer steps, but the first step requires some math insight into the problem. Besides, our declarative approach provides an alternative solution for the small input which can be used to validate the solution for the large input.

Star Wars[5]

"Star Wars" was one of the harder problems from round 2 of GCJ 2008, yet it can be almost trivially modeled and solved as a linear programming problem.

The essence of the problem statement is: you are given a set of N 4-tuples of integers x_i, y_i, z_i, p_i. Find the minimal possible Y for which exists a triplet x, y, z such that for each original tuple $|x_i - x| + |y_i - y| + |z_i - z| \leq p_i Y$.

Direct translation of the problem statement to a model results in non-linear constraints, but these can be easily converted to linear constraints using the fact that $|X| \leq Max$ is equivalent to a pair of linear constraints $X \leq Max$ and $-X \leq Max$ [5].

```
1 :- lib(eplex).
2 model(Xs, Ys, Zs, Ps, X, Y, Z, P) :-
3     P $>= 0,
4     ( foreach(Xi, Xs), foreach(Yi, Ys),
5         foreach(Zi, Zs), foreach(Pi, Ps),
6         param(X, Y, Z, P) do
7       +(Xi - X) +(Yi - Y) +(Zi - Z) $=< Pi * P,
8       +(Xi - X) +(Yi - Y) -(Zi - Z) $=< Pi * P,
9       +(Xi - X) -(Yi - Y) +(Zi - Z) $=< Pi * P,
10      +(Xi - X) -(Yi - Y) -(Zi - Z) $=< Pi * P,
11      -(Xi - X) +(Yi - Y) +(Zi - Z) $=< Pi * P,
12      -(Xi - X) +(Yi - Y) -(Zi - Z) $=< Pi * P,
13      -(Xi - X) -(Yi - Y) +(Zi - Z) $=< Pi * P,
14      -(Xi - X) -(Yi - Y) -(Zi - Z) $=< Pi * P ).
15 find(P) :-
16     eplex_solver_setup(min(P)),
17     eplex_solve(_),
18     eplex_var_get(P, typed_solution, P),
```

[5]Problem link: `http://goo.gl/DtpEQl`

```
19      eplex_cleanup.
```
Listing 5: Linear programming solution for "Star Wars"

Line 1 loads linear programming library 'eplex'. Lines 2–15 define a linear programming model with input arguments `Xs, Ys, Zs, Ps` (lists of x_i, y_i, z_i, p_i), and output parameter `P`. Other parameters of the `model` predicate (`X, Y, Z`) can be useful for debugging. Line 3 constrains the value of `P` to be non-negative using `$>=` from 'eplex'. Lines 4–6 define a header of ECL^{i}PS^e loop: execute the loop body for each x_i, y_i, z_i, p_i, and do not treat variables `X, Y, Z, P` as local for the loop iterations. Lines 7–14 specify the linearized problem constraints.

Lines 15–19 define the auxiliary predicate `find` that sets up the goal for `eplex` solver (minimize P), runs the solver, gets the typed value of P, and cleans up the solver environment for the following test cases.

Possible steps to come up with this solution:

1. *Express the problem constraints in linear form.*
2. *Formulate linear programming model.*
3. *Code linear programming model.* Trivial after the formulation.

The first step towards a traditional algorithmic solution is to notice that we can use binary search to find the smallest possible solution Y_s: all $Y > Y_s$ will satisfy the constraints, and all $Y < Y_s$ will not. Thus we make a transition from searching for the smallest Y to checking whether a certain Y satisfies the constraints.

The constraints can be converted to linear form in the same way as in declarative solution. However, instead of solving a full linear programming problem, we do some more analysis to check whether solution exists in $O(1)$ time (see the official contest editorial[6]).

Possible mental steps for this solution:

1. *Make transition from optimization problem to binary search combined with constraints feasibility check.*
2. *Express the problem constraints in linear form.*
3. *Perform constraints analysis to simplify feasibility check.*
4. *Implement feasibility check.*
5. *Implement binary search.*

Linear programming solution in ECL^{i}PS^e needs fewer steps, and the steps themselves are less complex.

Mine Layer[7]

This problem from the GCJ World Finals 2008 turned out to be very tricky for the contestants, with the smallest success rate from the all problems of the round: only 42% of the finalists who submitted answers for the large input got it right.

The problem describes a rectangular grid with odd number of rows, in which each square contains either one mine or no mines. Based on it we build a grid of integers: each integer is the total number of mines in corresponding square and eight adjacent squares. The task is: given the grid of mine counts, find the maximum possible number of mines in the middle row of the original grid.

An efficient integer programming model for "Mine Layer" is relatively straightforward to come up with and code in ECL^{i}PS^e:

```
1  :- lib(eplex).
2  model(Clues, Mines, MiddleSum) :-
3    dim(Clues, [R, C]),
4    dim(Mines, [R, C]),
5    ( foreachelem(Mine, Mines) do
6      Mine $:: 0..1,
7      integers(Mine) ),
8    ( multifor([I, J], 1, [R, C]),
9      param(R, C, Clues, Mines) do
10       ( multifor([Di, Dj], -1, 1),
11         fromto(0, Prev, Curr, S),
12         param(I, J, R, C, Mines) do
13         ( I + Di > 0, I + Di =< R,
14           J + Dj > 0, J + Dj =< C ->
15           Curr = Prev + Mines[I + Di, J + Dj]
16         ;
17           Curr = Prev
18         )
19       ),
20       Clues[I, J] $= S
21   ),
22   ( for(J, 1, C),
23     fromto(0, Prev, Curr, MiddleSumExpr),
24     param(Mines, R) do
25       Curr = Prev + Mines[R // 2 + 1, J]
26   ),
27   integers(MiddleSum),
28   MiddleSum $= MiddleSumExpr.
29 find(MiddleSum) :-
30   eplex_solver_setup(max(MiddleSum)),
31   eplex_solve(_),
32   eplex_var_get(MiddleSum,
33     typed_solution, MiddleSum),
34   eplex_cleanup.
```
Listing 6: Integer programming solution for "Mine Layer"

Line 1 loads 'eplex' library. Lines 2–29 define integer programming model with input parameter `Clues` (2-D grid) and output parameters `Mines` (2-D array of possible mine positions, 1 if mine is present and 0 otherwise) and `MiddleSum` (count of mines in the middle row). Line 3 gets number of rows (`R`) and columns (`C`) from the `Clues` array, and line 4 defines the same dimensions for the `Mines` array. Lines 5–7 define domain for each element of the `Mines` as 0 or 1.

Lines 8–20 contain two nested `multifor` loops (which are syntactic sugar for several `for` loops). For each grid square (`multifor([I, J], 1, [R, C])`) the loops construct expressions for number of mines in the square itself and in the neighboring squares, and then constrain this expression to agree with the value in the `Clues` array. Checks in lines 13 and 14 prevent accessing values outside of the grid for border squares.

Lines 21–27 construct expression for sum of values in the

[6] `http://goo.gl/ndjZhg`
[7] Problem link: `http://goo.gl/6ZyZGc`

Table 1: Running times for small (4 minutes time limit) and large (8 minutes time limit) inputs[8]

Problem	Library	Small	Large
Triangle Areas	ic	0.2s	1.4s
Dancing With the Googlers	ic	0.2s	timeout
Dancing With the Googlers	eplex	0.3s	1.7s
Star Wars	eplex	0.2s	0.4s
Mine Layer	eplex	0.2s	2.9s

middle row of the `Mines` array (count of mines in the middle row), and constrain value of `MiddleSum` to be equal to value of this expression and to be integer. It is not necessary to explicitly enforce `MiddleSum` to be integer because it is a sum of integer elements of `Mines`; but integer specification allows to output the answer directly as integer, without decimal point.

Lines 28–32 define the `find` predicate that sets up the goal for `eplex` solver (maximize `MiddleSum`), runs the solver, gets the typed value of `MiddleSum`, and cleans up the solver environment for the following test cases.

Possible steps to come up with this solution:

1. *Notice that all constraints and objective are linear.*
2. *Formulate integer programming model.*
3. *Code integer programming model.* The code is short, but uses a lot of ECL^{i}PS^e syntax constructs.

The key observation for an algorithmic solution is that we don't need to reconstruct the exact grid of mines to get the answer, it's enough to be able to count mines in certain sections of the grid. To find the number of mines in a certain section of the grid, we can split this section into 3×3 blocks of squares and add up numbers in the central squares of each block. If the section width or height is not divisible by 3, blocks along the sides of the original grid can have width or height of 2, so that the number in the "central" square of the block still contains the total number of mines in the block. The solution itself is easy to implement and only requires careful handling of different sizes of the grid. For a full solution see the official editorial[9].

Mental steps:

1. *Notice that instead of reconstructing the grid of mines we can count mines in certain sections of the grid.*
2. *Figure out how to count mines in any* 3×3 *block.*
3. *Figure out how to count mines in* 2×3*,* 3×2 *or* 2×2 *block along the border of the grid.*
4. *Figure out how to split the whole grid or its parts in countable blocks various sizes of grid.*
5. *Implement the counting.*

Our declarative solution has fewer steps, and they are much easier to perform, as they don't require much insight into the problem.

[8]Results were obtained on a 64-bit Linux machine with Intel Core i7-4900MQ CPU @ 2.80GHz and 16GB RAM using ECL^{i}PS^e 6.1 #191. For linear (integer) programming free COIN-OR solvers bundled with ECL^{i}PS^e were used.

[9]`goo.gl/K2dfPi`

3. CONCLUSIONS

Many GCJ problems that are hard to solve in time-restricted and stressful competition environment can be relatively easily modeled and solved in ECL^{i}PS^e. We gave several examples of such problems, and our declarative solutions for them require simpler and often fewer mental steps than possible imperative solutions in a language like C++ or Java. Running times of our ECL^{i}PS^e programs are several orders of magnitude smaller than the time limit imposed by GCJ rules (table 1).

Other modern declarative high-level tools can also be successfully used to solve competitive programming problems: answer set programming tools, satisfiability modulo theories solvers, etc. This can be a topic of further research.

4. REFERENCES

[1] K. R. Apt and M. Wallace. *Constraint Logic Programming Using ECLiPSe*. Cambridge University Press, New York, NY, USA, 2007.

[2] J. Bisschop. AIMMS – optimization modeling. `http://www.aimms.com/downloads/manuals/optimization-modeling/`, 2012.

[3] P. Brisset, H. El Scakkout, T. Frühwirth, C. Gervet, W. Harvey, M. Meier, S. Novello, T. Le Provost, J. Schimpf, K. Shen, et al. ECLiPSe constraint library manual. release 6.1. `http://www.eclipseclp.org/doc/libman.pdf`, 2014.

[4] D. E. Knuth and L. T. Pardo. *The Early Development of Programming Languages*. Stanford University, Computer Science Department, 1976.

[5] lp_solve 5.5.2.0 reference guide. Chapter "Absolute values". `http://lpsolve.sourceforge.net/5.5/absolute.htm`, 2013.

[6] J. Schimpf. Logical loops. In P. Stuckey, editor, *Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 224–238. Springer Berlin Heidelberg, 2002.

[7] J. Schimpf and K. Shen. ECLiPSe – from LP to CLP. *Theory Pract. Log. Program.*, 12(1-2):127–156, Jan. 2012.

[8] K. Shen and J. Schimpf. Eplex: Harnessing mathematical programming solvers for constraint logic programming. In P. van Beek, editor, *Principles and Practice of Constraint Programming – CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 622–636. Springer Berlin Heidelberg, 2005.

[9] H. Williams. *Model Building in Mathematical Programming*. Wiley, 5th edition, 2013.