Symmetry Breaking in the Alien Tiles Puzzle

Ian Gent*, Steve Linton†, and Barbara Smith‡

October, 2000

Report APES-22-2000

Abstract

We describe an application of symmetry breaking in constraint programming to a combinatorial puzzle based on the Alien Tiles game. We obtain a 40-fold run-time improvement over code with no symmetry breaking. We believe this is the first integration of an algebraic system like GAP and a constraint programming system like ILOG Solver.

Available from http://www.cs.strath.ac.uk/~apes/apesreports.html

* School of Computer Science, University of St Andrews, St Andrews, Fife, KY16 9SS, UK. ipg@cs.st-and.ac.uk
† School of Computer Science, University of St Andrews, St Andrews, Fife, KY16 9SS, UK. sal@cs.st-and.ac.uk
‡ School of Computing, University of Leeds, Leeds, LS2 9JT, UK. bms@comp.leeds.ac.uk

1 Introduction

We describe an application of symmetry breaking in constraint programming to a combinatorial puzzle based on the Alien Tiles game. This work has a number of interesting features

- we obtain a 40-fold *run-time* improvement over the code with no symmetry breaking.
- this was achieved without any new constraint programming techniques, but by direct application of the technique reported in [1, 3].
- it represents a large increase in the number of symmetries we handle explicitly, from 7 in the n-queens and LABS problems [3], to 1151.
- to calculate the 1151 symmetry functions, we directly applied the computational algebra system GAP [2]. We believe this is the first integration of an algebraic system like GAP and a constraint program system like ILOG Solver.
- while this is a successful application, it shows up the need to integrate GAP and constraint programming more closely. In this application the 1151 symmetry functions were handled naively as individual C++ functions. To handle orders of magnitude more functions, or this number of functions more efficiently, we will need new techniques.
- as a final point, even expressing the constraints of the problem without symmetry involved the use of GAP. This raises the interesting question of whether we can use symmetry more widely in constraint programming.

2 The Alien Tiles Problem

The Alien Tiles puzzle is available for play over the internet at www.alientiles.com. We addressed the combinatorial problem of finding the most difficult puzzle, in a certain sense.

For the Alien Tiles puzzle, you are presented with a square grid, with each square a given colour. Each grid square can be one of some number of colours, and the colours are ordered in a cycle, for example $Red \rightarrow Green \rightarrow Blue \rightarrow Red$. Moves in the puzzle are made by clicking on one of the grid squares. Each click rotates the colour by one in both the clicked square, and all other squares in the same row and column. Given a starting state and a goal state of the grid, the puzzle is to find a set of clicks of squares that achieves the goal.

Notice that the order of the clicks does not matter, so all we have to do is to decide how many times to click on each square. The puzzle reduces to arithmetic modulo the number of colours, c. Clicking adds one mod c to each square in the same row and column. Instead of colours the start and goal states are an assignment of integers mod c to the squares. For simplicity we assume from now on that the start state is all zeroes.

There are a number of interesting combinatorial questions about Alien Tiles, as well as the straightforward one of solving given positions. We outline one particular question.

With certain versions of the puzzle, if a goal state can be reached there are many equivalent ways of reaching it. For example, consider a 4x4 grid, with $c = 3$ colours. Consider clicking once in each square in the top row, and twice in each square in the second row. This adds 3 clicks to each square in the bottom two rows, so makes no difference mod 3. It adds 6 clicks to each square in the top row, and 9 clicks to each square in the second row. The net result is that no difference mod 3 is made by this set of clicks, and the final colours in the squares of the grid are the same as the starting colours. We can do the same for any two rows, and two columns, and any combination thereof. Therefore there are many equivalent versions of any given solution. Of all the equivalence class of solutions under these operations, there is some solution with minimum number of clicks over the whole grid. We can now ask: what is the absolute maximum number of clicks necessary to solve any (solvable) goal state? That is, which solvable state has the largest minimum number of clicks in its equivalence class of solutions?

3 Alien Tiles as a Constraint problem

To encode our question as a constraint satisfaction problem, we observe that we can work not with the goal states, but with the solutions. That is, the variables and values will be the number of clicks made in each square in the solution: from this we can read off the resulting goal state.

The main part of the encoding is to restrict solutions to minimal elements of each equivalence class. As an example of the constraints in the problem, if we add 1 to each number in the top row of the grid, mod 3, and we add 2 in the second row, the sum of the numbers in the entire grid must not decrease: if it did decrease we would not be at a minimal element in a class. To guarantee minimality, we have to consider all combinations of rows and columns.

It is not at all trivial to calculate all combinations of row and column sums. Instead of doing this laboriously by hand, we wrote a GAP program to calculate the set of constraints. The result is a set of 728 lines of C++ code for use with the ILOG Solver library. We find it interesting that the constraint program itself was written in part by the use of computational algebra.

Finally the problem can be encoded by asking for the set of numbers satisfying the constraints making it a minimal element, but having largest sum, i.e. it becomes an optimization problem.

4 Symmetry Breaking in Alien Tiles

While the construction of the program using GAP is interesting, it is not our main focus. There is a lot of symmetry remaining in the problem.

Given a solution to our problem, we can swap any two rows and obtain an equivalent solution. The same applies to any two rows. Finally, we can reflect the grid in a diagonal. With 4! = 24 row and column operations, and one reflection, we get a group of $24x24x2 = 1152$ symmetries acting on the set of solutions.

To exclude symmetric solutions, we use the ILOG Solver code written by Gent and Smith [3]. The requirement on the programmer is to write a C++ function representing each symmetry in the group (except for the identity). Given that, the code guarantees not to return two symmetrically equivalent solutions.

In this example, encoding the 1151 symmetry functions necessary would be almost impossible by hand. Again, we were able to write a GAP program to calculate the group and output the 1151 C++ functions. The key function in the GAP program is the following. It calculates the group given three permutations which are known to generate it.

```
#
# thegroup(n) returns a group of permutations of [1..n^2] which are
# the natural symmetries of the Alien tiles board. The group should
# have order 2*(n!)^2
#

groups := [];

thegroup := function(n)
    local p1,p3,p5;
    if not IsBound(groups[n]) then
        p1 := PermList(List([1..n^2],
                    i->cellno(coords(i,n)[1], coords(i,n)[2] mod n+1,n)));
        p3 := PermList(List([1..n^2],
                   function(i) local x; x := coords(i,n)[2]; if x = 1 then
                      x := 2; elif x = 2 then x := 1; fi;
                      return cellno(coords(i,n)[1], x,n); end));
                      p5 := PermList(List([1..n^2], i-> cellno(coords(i,n)[2],coords(i,n)[1],n)));
                      groups[n] := Group(p1,p3,p5);
    fi;
    return groups[n];
end;
```

The first function output by the GAP program is:

```
static IlcInt x1p[] =
           {0, 1, 2, 3, 4, 5, 6, 7, 12, 13, 14, 15, 8, 9, 10, 11};
IlcConstraint x1 (IlcIntVarArray vars, IlcInt i, IlcInt j)
{ return vars[x1p[i]] == j;}
```

This function returns the equivalent of `vars[i] == j` under the symmetry which exchanges the last two rows of the grid, so that `vars[8], vars[9], vars[9, vars[11]` are exchanged with `vars[12], vars[13], vars[14], vars[15]` respectively. The implementation is not very efficient: for instance, this function returns `vars[0] = j` as the symmetric equivalent of itself, and this will be treated as a separate constraint, even though it is identical to a constraint already in place. However, the implementation is correct and achieves the aims we had for symmetry breaking; no (partial or complete) assignment which is symmetric to one already considered will ever be explored.

5 Results

We tested the 4x4 3-colour problem with and without the symmetry functions.

The optimal solutions found are exactly the same in both cases, as they should be - but one run takes 2325 secs. (without symmetry constraints) and the other takes only 57 secs. (with symmetry constraints). As we should expect, symmetry-breaking only makes a difference when solutions are getting hard to find, and makes most difference in proving optimality, i.e. proving that there is no solution with cost 11. The reduction in the number of backtracks (fails) is greater still: with symmetry-breaking, the number of fails is reduced to less than 1% of the number without. A more efficient implementation of the symmetry-breaking functions and the way they are handled might achieve a greater reduction in runtime, but the reduction in the number of fails would not change.

Symmetry-breaking also uses more memory, as might be expected: without symmetry-breaking, the memory consumption reported by Solver when each solution is found is constant, at 1.2Mb; with symmetry-breaking it varies, with a maximum of about 13Mb. Again, it is likely that a smarter implementation could improve this.

5.1 Results without Symmetry-Breaking

```
Cost:0
Number of fails               : 0
Elapsed time since creation   : 0.972

Cost:1
Number of fails               : 0
Elapsed time since creation   : 0.989

Cost:2
Number of fails               : 0
Elapsed time since creation   : 0.999

Cost:3
Number of fails               : 0
Elapsed time since creation   : 1.027

Cost:4
Number of fails               : 0
Elapsed time since creation   : 1.041

Cost:5
Number of fails               : 0
Elapsed time since creation   : 1.075
```

```
Cost:6
Number of fails               : 0
Elapsed time since creation   : 1.092

Cost:7
Number of fails               : 0
Elapsed time since creation   : 1.133

Cost:8
Number of fails               : 0
Elapsed time since creation   : 1.154

Cost:9
Number of fails               : 290
Elapsed time since creation   : 10.386

Cost:10
Number of fails               : 866
Elapsed time since creation   : 31.185

Maximum Best Solution : 10
 [0] [0] [0] [0]
 [0] [0] [1] [1]
 [0] [1] [0] [1]
 [2] [1] [1] [2]]
Number of fails               : 57664
Elapsed time since creation   : 2324.96
```

5.2 Results with Symmetry-Breaking

```
Cost:0
Number of fails               : 0
Elapsed time since creation   : 3.509

Cost:1
Number of fails               : 0
Elapsed time since creation   : 3.736

Cost:2
Number of fails               : 0
Elapsed time since creation   : 3.774

Cost:3
Number of fails               : 0
Elapsed time since creation   : 3.949

Cost:4
Number of fails               : 0
Elapsed time since creation   : 3.987

Cost:5
Number of fails               : 0
Elapsed time since creation   : 4.175

Cost:6
Number of fails               : 0
Elapsed time since creation   : 4.223

Cost:7
Number of fails               : 0
Elapsed time since creation   : 4.443

Cost:8
```

```
Number of fails               : 0
Elapsed time since creation   : 4.495

Cost:9
Number of fails               : 29
Elapsed time since creation   : 11.557

Cost:10
Number of fails               : 116
Elapsed time since creation   : 18.585

Maximum Best Solution : 10
 [0] [0] [0] [0]
 [0] [0] [1] [1]
 [0] [1] [0] [1]
 [2] [1] [1] [2]]
Number of fails               : 499
Elapsed time since creation   : 56.589
```

To find the goal state corresponding to this longest minimal solution, given that the starting state is all red, for each cell we take the sum (mod 3) of the numbers in the corresponding row and column, and convert the result into a colour (where 0 = red, 1 = green, 2 = blue), giving:

```
B B B G
G G R B
G R G B
R G G B
```

There are 19 unique solutions requiring 10 clicks, i.e. solutions which cannot be transformed into each other by any of the 1152 symmetries.

References

[1] R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *Proceedings, CP-99*. Springer, 1999. LNCS 1713.

[2] The GAP Group, Aachen, St Andrews. *GAP – Groups, Algorithms, and Programming, Version 4*, 1998. (`http://www-gap.dcs.st-and.ac.uk/~gap`).

[3] I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In W. Horn, editor, *Proceedings of ECAI-2000*, pages 599–603. IOS Press, 2000.