Adnan Alhariri and Federico Caputo
Theory of Computation
17/05/2010

SAT SOLVER FOR TETRAVEX:

MINI PROJECT II


```
 _____________   _____________   _____________
|             | |             | |             |
|      6      | |      3      | |      6      |
|     \ /     | |     \ /     | |     \ /     |
|   8  X  8   | |   8  X  7   | |   7  X  7   |
|     / \     | |     / \     | |     / \     |
|      7      | |      1      | |      8      |
|_____________| |_____________| |_____________|
 _____________   _____________   _____________
|             | |             | |             |
|      7      | |      1      | |      8      |
|     \ /     | |     \ /     | |     \ /     |
|   3  X  4   | |   4  X  3   | |   3  X  4   |
|     / \     | |     / \     | |     / \     |
|      3      | |      0      | |      8      |
|_____________| |_____________| |_____________|
 _____________   _____________   _____________
|             | |             | |             |
|      3      | |      0      | |      8      |
|     \ /     | |     \ /     | |     \ /     |
|   9  X  0   | |   0  X  1   | |   1  X  2   |
|     / \     | |     / \     | |     / \     |
|      3      | |      7      | |      5      |
|_____________| |_____________| |_____________|
```

ADNAN ALHARIRI

FEDERICO CAPUTO

The game

Tetravex is an edge-matching puzzle game, born in the early nineties as an extra for Microsoft Windows 3.x.

The game is composed by a set of square tiles, usually 9 and carrying a number on each side, to be arranged on a 3x3 field in such a way that no adjacent tiles have mismatching numbers on their facing sides.

The problem

Our task was to generate a series of rules for an existing SAT solver so that the SAT solver would return a possible solution for a Tetravex.

The only simple constrain to be considered is that the tiles must have the same number on their facing sides: apparently rather easy., but it had to be put in a way that made sense for the SAT solver.

The first steps

The first step was creating the simplest possible tile set, solve the game manually and find a way to reverse engineer the rules of the field.

The first trial was to manually specify where each tile could be set in respect to all others, but this solution presented two problems: too many cases, especially for big fields, and rules too complex because of the SAT solver syntax; this solution was scrapped pretty quickly.

After that we found a simpler way to declare tiles relations, which consisted in specifying for each tile which other tiles were not correct neighbor: this way we ended up having a consistent number of rules, but the theory seemed to work, so we started working on the actual implementation.

The implementation

Of the four components needed for a fully working Tetravex solver (parser, encoder, SAT engine, UI), the easiest two were the parser and the UI (the SAT engine was already given as a starting point); the real challenge was, of course, developing the encoder, which was supposed to transform the Tetravex rules into SAT rules.

The parser simply reads a text file, where each line is composed of 4 numbers from 0 to 9 in the form wxyz, being the values of one tile in clockwise order from top; the number of lines must be either 4, 9 or 16 (actually, it could be any number n with $\sqrt{n} \in N$, but for more than 16 the solving times increase dramatically).

The UI is merely a console rendering of the puzzles, printed using ASCII characters.

During subsequent refinements, we devised three different versions of the dynamic part of the encoder (the static part is made of predefined constraints, and is the one that states the rule(s) only-one-tile-per-space and all-tile-on-field, while the dynamic part varies based on the specific tiles in use in the specific puzzle); the first two versions conceptually very similar to each other, even if the second already reduced the number of cycles from four inner loops to only three, while the third one follows a completely different mechanism, resolving in a notable speed increase.

The first and second version of the encoder both work in the same fashion: the encoder take every possible tile, put it in every possible position and check for neighbors incompatibility with any other tile in any other possible position for the first implementation or in the four adjacent positions for the second implementation; the second version is slightly faster, but the difference become noticeable only with relatively big Tetravex.

The third version of the encoder works with a completely different principle: instead of putting the tiles on the board and checking for feasible neighbors (like the first two did), it simply takes each tile and check for every other tile whether the first can sit to the left of the second and whether the first can sit atop of the second; at this point, the rules are absolute (contrary to the relative ones of the first two implementations), and the second part of the algorithm loops over the field generating the in-place rules for the SAT solver.

NOTE: all the three engines output EXACLTY the same set of rules (although in different order), but they take different times to do that.

IMPORTANT NOTE: although given as a black box, having the SAT solver work on the same set of rules generated via different procedures revealed one very important characteristic => the order of the rules MATTERS in terms of SAT solver solving speed.

The encoder complexity

The complexity analysis, in this case, only gives part of the solution, since we don't know the exact behavior of the SAT solver and are limited to analyze the encoder complexity.

The static part always has a complexity of

$$2n^2+n^2$$

where we have n tiles that can't occupy two spaces at the same time, and

where we have n spaces that can't be occupied by two tiles at the same time,

so, an overall complexity of $\mathbf{O(n^2)}$.

The first implementation of the dynamic part had a complexity of

$$(n-1)^2n^2$$

where we have n tiles put over n possible spaces and compared with (n-1) in (n-1) possible spaces; working, but not very efficient, with an overall complexity of $\mathbf{O(n^4)}$.

The second implementation had a complexity of

$$4(n-1)n^2$$

where we have n tiles put over n possible spaces and compared with (n-1) in 4 possible spaces (4 neighbors); working and more efficient, with an overall complexity of $\mathbf{O(n^3)}$.

The third implementation had a complexity of

$$n(n-1)+m\sqrt{n}((\sqrt{n})-1)$$

with m = number of incompatibilities (worst case 4n)

where we have n tiles checked against (n-1) tiles plus $\sqrt{n}$ spaces times $(\sqrt{n})$-1 spaces times m rules, with an overall complexity of $\mathbf{O(n^2)}$.

So the best solution is $O(n^2)$ (static) + $O(n^2)$ (third implementation dynamic) = $\mathbf{O(n^2)}$.

SAT solver

The SAT solver is a black box, but from its behavior we extrapolated some intuitions regarding its algorithm.

Having three different engines that output the same rules in different order, we tested them all with the SAT solver; the result was exactly the same, but the computation takes different times.

The difference in computing time led us to deduce that the order of the rules greatly matters for the SAT solver.

After this discovery we wanted to see if there were ways to speed up the SAT solver improving or at least without crippling too much the encoder.

At first we tried to tweak the static part of the encoder to make it faster, but the result was a complete disaster: the solving time skyrocketed, almost doubling; the nature of the tweaking revealed a possible reason: if rules for the same tile are concentrated (pre-tweaking), the SAT is faster than when they are spread over the list of rules (post-tweaking).

After this consideration, we decided to try to sort the rules before writing them in the list; the result was excellent, we halved the computing time.

So, in conclusion we have an excellent encoding algorithm (according to java documentation, the sorting adds only $n\log n$, so it doesn't increase the complexity), and we were also able to reduce the computing time.

Rules

The rules we created for the SAT solver, despite their consistent number, are created starting from few and very simple logical premises; what follows is a short description of each rule used with their logical statement, their template and some examples to clarify their motivation.

"A tile can't occupy more than one space"

(~a v ~b)

(~tile_i_in_space_j v ~tile_i_in_space_k)

~tile_0_in_space_0 ~tile_0_in_space_1 $

~tile_0_in_space_0 ~tile_0_in_space_2 $

...

~tile_8_in_space_7 ~tile_8_in_space_8 $

"Each space can't hold more than one tile"

(~a v ~b)

(~tile_i_in_space_j v ~tile_k_in_space_j)

~tile_0_in_space_0 ~tile_1_in_space_0 $

~tile_0_in_space_0 ~tile_2_in_space_0 $

...

~tile_7_in_space_8 ~tile_8_in_space_8 $

"Each tile must stay in one space"

(a v b v ... v i)

(tile_i_in_space_k v tile_i_in_space_(k+1) v ... v tile_i_in_space_n)

tile_0_in_space_0 tile_0_in_space_1 ... tile_0_in_space_8

tile_1_in_space_0 tile_1_in_space_1 ... tile_1_in_space_8

...

tile_8_in_space_0 tile_8_in_space_1 ... tile_8_in_space_8

"Each tile can't stay next to incompatible ones"

(~a v ~b)

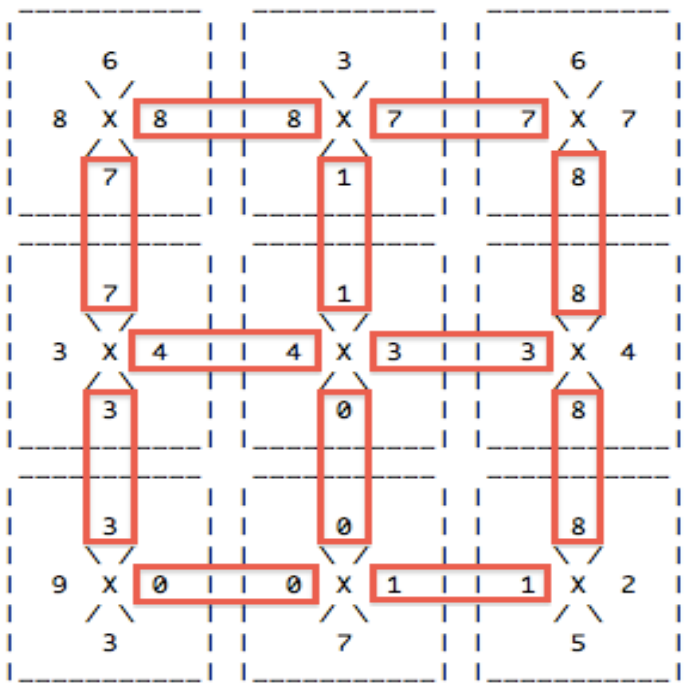
(~tile_i_in_space_j v ~tile_k_in_space_(next_to_j))

this set of rules depends strictly on the tiles that compose each puzzle, therefore they are different for each Tetravex.

Generator

Having some extra time on our hands, and being tired of searching the internet for new Tetravex puzzles to test, we decided to program a Tetravex generator.

We started looking at a solved puzzle of 3x3, and highlighted the important parts, namely the values that had to be identical for the puzzle to be valid:



From these premises, we decided that the easiest way to find a pattern was to "straighten" the Tetravex in a way that could actually be usable to our purposes, and a leftToRight-topToBottom sequence turned out to be optimal:

636 888777 718 718 344334 308 308 900112 375

Looking carefully at both the sequence and the puzzle, we were able to isolate a pattern:

636 8**(88)(77)**7 **[(718) (718)]** 3**(44)(33)**4 **[(308) (308)]** 9**(00)(11)**2 375

or simply

6368**(88)(77)**7**[(718)(718)]**3**(44)(33)**4**[(308)(308)]**9**(00)(11)**2375

which means (with x being a random number):

xxxx(aa)(bb)x(abc)(abc)x(aa)(bb)x(abc)(abc)x(aa)(bb)xxxx

now, considering the side of the puzzle is 3 (in this specific case):

xxx**|**x(aa)(bb)x**|**(abc)(abc)**|**x(aa)(bb)x**|**(abc)(abc)**|**x(aa)(bb)x**|**xxx

with this insight, we can easily device a pattern:

(*side*)x|x[(*side*-1)couples of identical numbers]x|2 identical sequences of numbers of length *side*|

x[(*side*-1)couples of identical numbers]x|2 identical sequences of numbers of length *side*|

x[(*side*-1)couples of identical numbers]x|(*side*)x

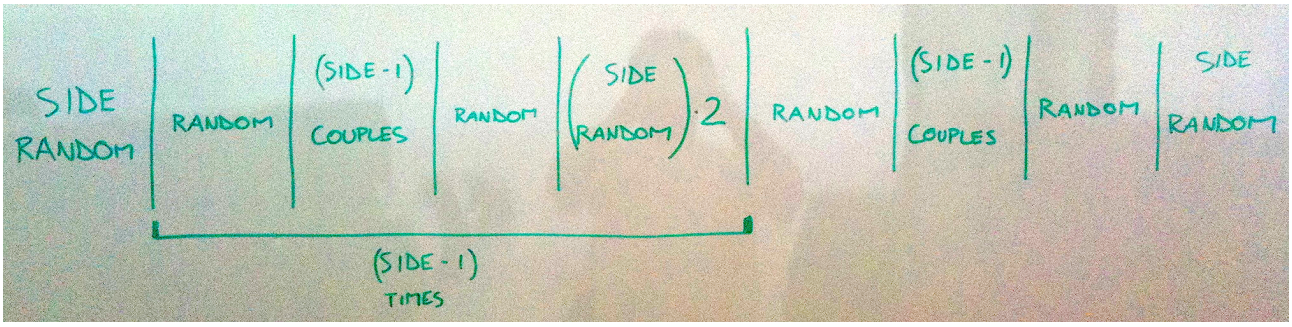
which can be generalized:

`(side)x`

`(side-1)(x[(side-1)couples]x | ((side)sequence)2)`

`x[(side-1)couples]x | (side)x`

(picture below might be more comprehensible)



So, as long as we can generate a sequence of random numbers that matches these criteria, we are generating a valid (solvable) Tetravex puzzle.

Of course, we started testing it immediately with our working engine, and the results confirmed our theory: we have a valid Tetravex generator.

How to use it

The application is freely downloadable at http://code.google.com/p/tetrasat/

It needs a working implementation of miniwrapper and to know the absolute path of the project, to be set through the variable *command* in *Solver*.

Our application has two independently executable components: *Generator* and *Solver*.

The *Generator* has to be launched with two parameters: the first one is the name of the file where the Tetravex definition has to be saved, and the second one is the side length of the desired puzzle. The *Generator* creates a file that contains a list of tiles for a valid Tetravex puzzle with the requested side length (the format is the same described above for the parser, for the obvious reason that it has to be readable from the parser itself).

The *Solver* has to be launched with one parameter: the name of the file where the definitions are. The *Solver* returns a printout of the tiles available, if it is solvable or not, and if it is it prints a possible solution and the time needed to compute it.