Solving "Number Cross 4" (Jane Street Puzzle, May 2024)

A Go Solver That Relabels Split Regions to Solve the Grid Row by Row

Guillaume Lethuillier
glthr.com

June 3, 2024 · updated November 11, 2026

Series: Jane Street Puzzles — Part 1 of 2
Next: Solving "Knight Moves 6" (Jane Street Puzzle, October 2024) (glthr.com/solving-knight-moves-6-jane-street-puzzle-october-2024)

Abstract

In May 2024, Jane Street released a cross-number puzzle, **"Number Cross 4"** (official solution). My Go solver found its unique solution, `88243711283`, by building the 11-by-11 grid row by row; what makes this possible is relabelling the regions that shaded cells may split.

Introduction

This puzzle, an 11-by-11 grid described on Jane Street's "Number Cross 4" page, encompasses various types of constraints:

- **Row clues** with which numbers, formed by concatenating adjacent digits from non-shaded cells, must comply (*e.g.*, square numbers, palindromic numbers).
- **Rules governing the relationship between digits and regions**: every cell within a region must contain the same digit, and orthogonally adjacent cells in different regions must contain different digits.
- **Locations of shaded cells**, which need to be sparse and not result in single-digit numbers.

The difficulty of this multi-constraint puzzle comes from the fact that regions are ultimately defined by where the solver more or less arbitrarily places shaded cells.

Cite as: Guillaume Lethuillier, "Solving "Number Cross 4" (Jane Street Puzzle, May 2024)", glthr.com, June 3, 2024. https://glthr.com/solving-number-cross-4-jane-street-puzzle-may-2024
Interactive figures, animations and videos of the web page are shown here as still images or addresses.

Solution

I submitted the unique solution, `88243711283` (the sum of all the numbers), discovered by the solver on May 12.

CLUE											NUMBERS
square	1	1	1	2	2	2	3	3	4	4	4	105462^2
1 more than a palindrome	1	3	3	3	2	▨	3	4	4	4	▨	13331 + 1 3443 + 1
prime raised to a prime power	1	3	3	1	▨	7	3	4	4	4	9	11^3 857^2
sum of digits is 7	1	3	3	▨	1	0	0	4	1	1	▨	1+3+3 1+0+0+4+1+1
fibonacci	1	3	▨	1	4	4	▨	4	1	8	1	F_7 F_{12} F_{19}
square	1	4	4	4	▨	4	4	4	8	8	9	38^2 667^2
multiple of 37	7	4	4	4	4	▨	7	4	8	8	8	37 × 2012 37 × 2024
palindrome & multiple of 23	7	7	1	4	1	7	7	▨	9	8	9	23 × 335399 23 × 43
product of digits ends in 1	7	7	1	1	1	7	7	9	9	9	9	product 15752961
multiple of 88	▨	1	1	4	4	▨	7	9	9	9	2	88 × 13 88 × 909
1 less than a palindrome	4	4	4	4	4	3	▨	3	9	9	2	444444 − 1 3993 − 1

Initial Failed Attempt: Use of an SMT Solver

I initially attempted to solve the puzzle by modeling it as an SMT problem to leverage the Z3 solver.

However, this strategy proved unsuccessful for several reasons:

1. My solver is implemented in Go, and the main Z3 library available for this language is no longer maintained, leading to frequent crashes. Additionally, this library does not handle parallel calls well, making it unsuitable for concurrent operations.

2. Some of the clues posed significant challenges when modeled under these conditions. For example, how can digits be restricted to form one or more primes raised to a prime power (3rd-row clue)? More generally, how can the relationship between shaded cells, regions, and numbers be modeled?

3. The solving process was slower than expected for easily modelable clues (*e.g.*, generating square numbers), making it unfit to tackle the puzzle as a whole.

Due to these outcomes—probably not because of Z3 itself, but rather how I used it and the limitations of the Go library—I took a different approach. **I chose an iterative solving approach, where:**

- (identification) **each row satisfying the constraints is identified**, then
- (augmentation) **added to rows that are above it and that satisfy their constraints**, and

- (propagation) **alongside its own constraints (digits assigned to regions potentially connected to the next row), propagated to a new instance of the solver** (that is: a goroutine).

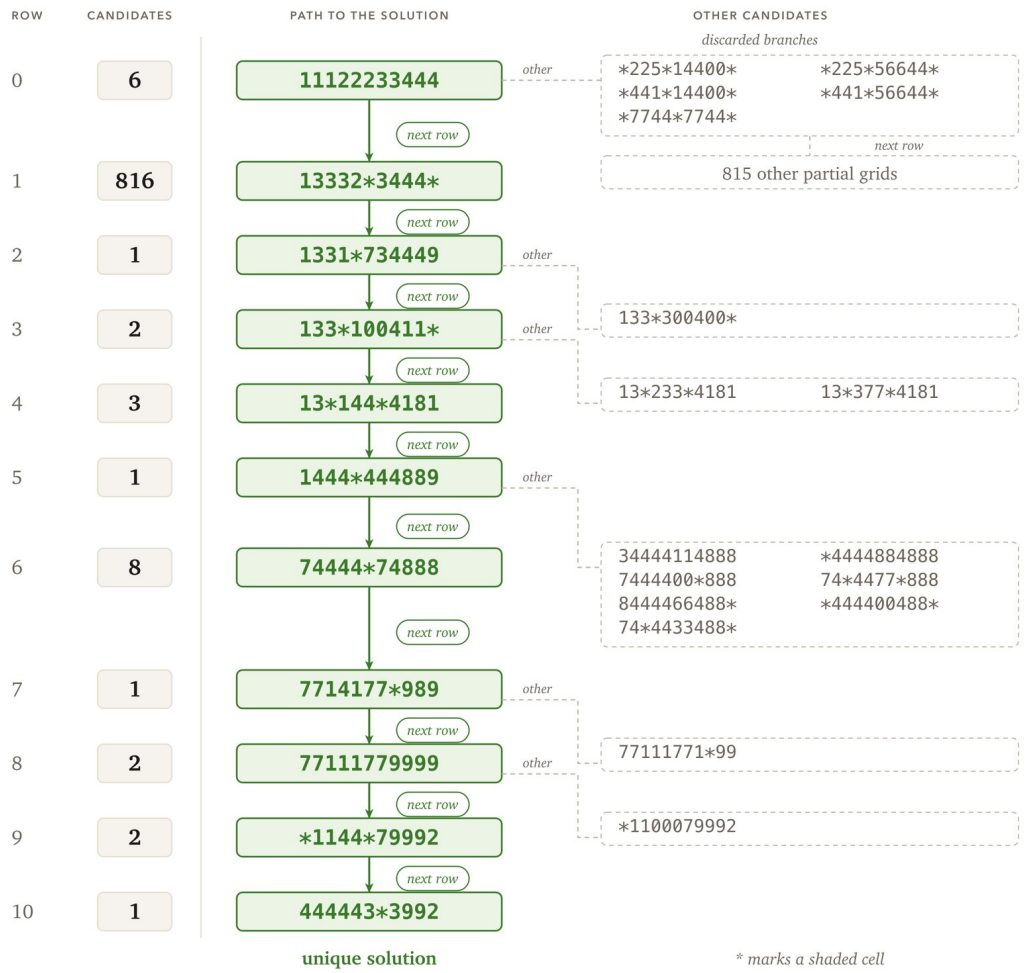
Successful Approach: Solve and Propagate Valid Rows

This second approach (essentially brute forcing but propagating the candidates forward instead of backtracking) proved successful. On a recent laptop, the solver—implementing this approach—traverses the entire valid (in terms of constraints) search space within approximately five seconds and finds the solution in about one second.

Regions	Digits
`aaabbbccddd`	`11122233444`
`aeeeb*cddd*`	`13332*3444*`
`aeef*gcdddh`	`1331*734449`
`aee*ijjdkk*`	`133*100411*`
`ae*lmm*dkno`	`13*144*4181`
`appp*dddnnq`	`1444*444889`
`rpppp*sdnnn`	`74444*74888`
`rrtpuss*vnw`	`7714177*989`
`rruuusswwww`	`77111779999`
`*uuxx*swwwy`	`*1144*79992`
`xxxxxz*@wwy`	`444443*3992`

The solver's final region labels and digits. An asterisk marks a shaded cell.

A significant factor contributing to this efficiency is how the puzzle's constraints are constructed, enabling the solver to converge quickly toward the solution. The following diagram illustrates this by showing the number of valid candidates at each level: the initial level (first row) generates just six eligible candidates. The subsequent level produces 816 such candidates. Subsequently, however, the number of valid candidates merely fluctuates between 1 (for four levels) and 8 (for one level).

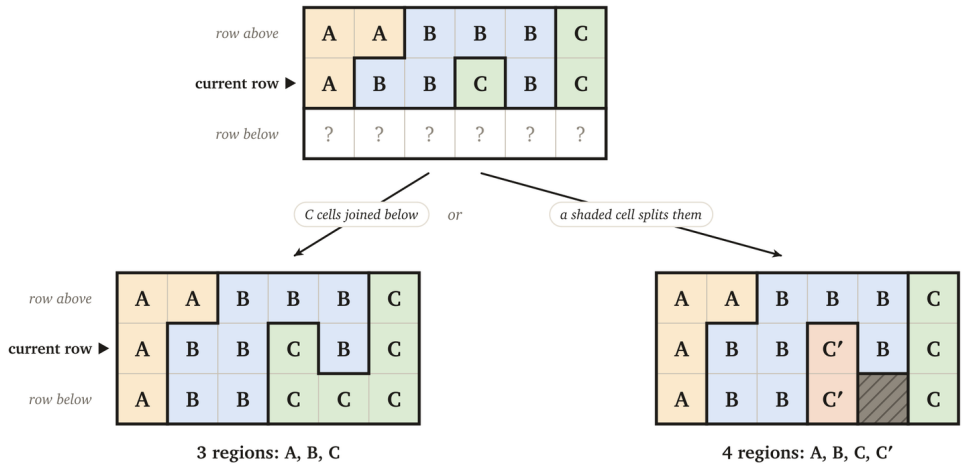


For this reason, a single-threaded implementation of this solver takes less than 30 seconds on similar hardware to solve the puzzle. Thus, in retrospect, it appears that parallelism is not crucial to solving this puzzle efficiently.

Solving Row by Row

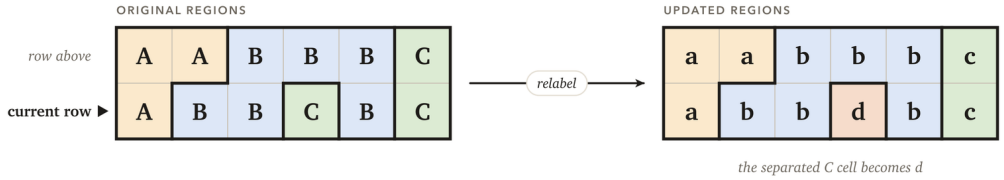
The initial challenge was determining whether this puzzle could be solved iteratively, row by row *(from top to bottom)*, or if a more global approach was necessary due to interdependencies between cells from different rows.

Initially, my intuition leaned towards requiring a global approach, given that the definition of regions in a row depends on the shaded cells in the rows above *and, critically, below*, as demonstrated in the following illustration:

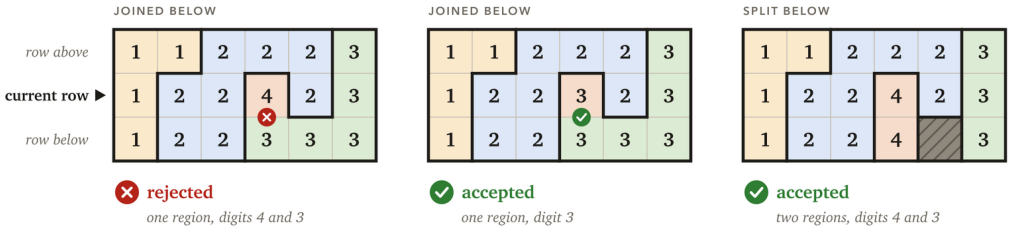


In this example, the current row (indicated by the arrow) may ultimately have *three* (`A`, `B`, and `C`) or *four* (`A`, `B`, `C`, and `C'`)—or even more—regions based on the presence of a shaded cell at a specific position within the row below that separates (or not) the `C` cells. This information cannot be determined when the current row is being solved: *it cannot be decided yet, as it depends on the placement of shaded cells on a row that has not been solved yet.*

Upon further consideration, I discovered that the originally joined regions could be *temporarily* considered disjoint at a row level, meaning they should not necessarily have the same digits. In other words, cells originally belonging to the same region could be assigned to different regions when *(a)* a top row is not connected to them and *(b)* they are separated by a region or a shaded cell in the current row. In this example, the original regions `A`, `B`, `C`, become `a`, `b`, `c`, and `d`.



That way, if they are actually connected by the row below (once it is the turn for the solver to consider the row below), any digit assignment that would violate the rule according to which regions should have the same digits would be excluded. Conversely, if they stay disjoint, nothing ties their digits: the two parts are never orthogonally adjacent, so they may hold the same digit or different ones.



This insight supported the idea that the puzzle could be solved on a row-by-row basis.

Before discussing how the solver traverses the search space, I must first clarify the generation of shaded cells and how regions are encoded.

Generating the Shaded Cells

The solver represents each configuration of shaded cells as a *mask* (a bitfield; a slice of booleans) where an unset bit `0` represents a digit, and a set bit `1` represents a shaded cell. For example, `00100000100` corresponds to three groups of digits and two shaded cells: one at x-position 2 and another at x-position 8.

Valid masks must comply with these rules:

1. Unset bits cannot be isolated (that is: `01---`, `---101---`, or `---10`) as the numbers must be at least 2 digits long.
2. Set bits cannot be horizontally adjacent as shaded cells are horizontally sparse (*i.e.*, `---11---`).
3. Set bits cannot be vertically adjacent as shaded cells are also vertically sparse (*e.g.*, top row `0010010` is incompatible with the bottom row `0010000` because of the shaded cells at x-position 2).

The masks are trivially generated by a function that builds the Cartesian product of sets of set and unset bits and filters out masks that do not satisfy the first two rules (the third one—vertical sparsity—can only be enforced, by a different function, between two rows).

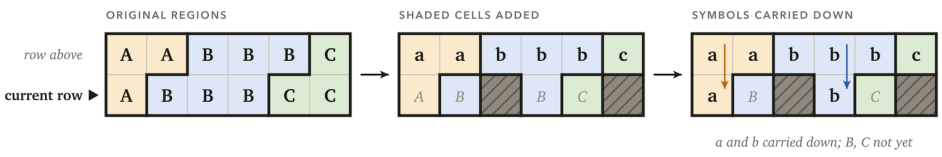
This function generates 54 valid masks per row.

Encoding the Regions

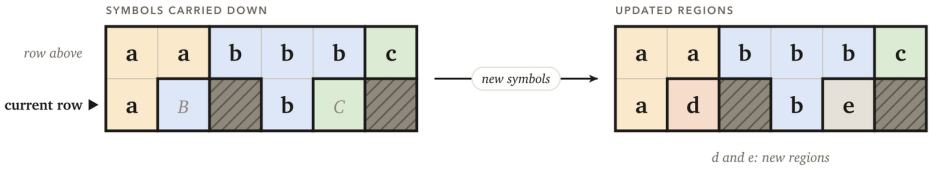
The original regions are defined in a configuration file and denoted using uppercase letters, ranging from `A` to `M`. Updated regions are defined row-by-row by updating the original regions based on a given configuration of shaded cells—as explained above. These regions are symbolized by characters that have not yet been utilized to represent the original regions (`a` to `z`, *etc.*).

Regions are updated as follows:

- Initially, when cells in the current row are connected with those in the preceding row—by considering the original regions—the updated symbols representing these regions in the upper row are carried down to the lower row.

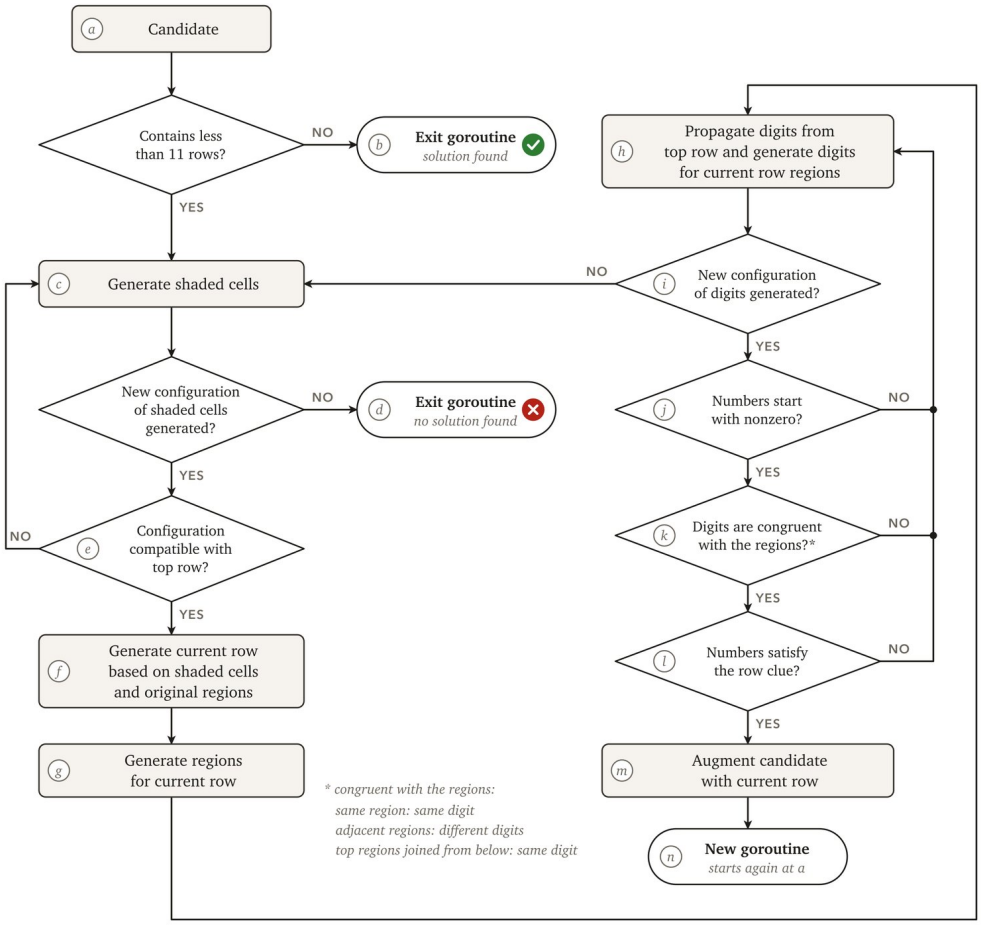


- Subsequently, unassigned regions are assigned symbols not used so far. Cells belonging to the same original region and adjacent to each other are allocated the same symbol. They belong to distinct regions if originally connected but horizontally separated by a region or a shaded cell. This does not preclude them from possessing identical digits if it is subsequently determined that they are connected from below.



Solving Process

My solver proceeds like this:



The fourteen steps, one by one

- *(a)* The solving function, initiated in a goroutine, begins by examining a prospective candidate containing 0 to 11 valid rows.
- *(b)* If the candidate contains 11 rows, terminate the present exploration since the candidate is a solution. *(Solved).*
- *(c)* Otherwise, iterate through all viable arrangements of shaded cells (as described above).
- *(d)* If no such configurations remain, exit the current process: the candidate cannot possibly yield a solution.
- *(e)* Compare the present configuration of shaded cells with the one in the row above. If two shaded cells are vertically adjacent, return to step *(c)*, as vertical sparsity is required for shaded cells.
- *(f)* Instantiate the current row based on the original regions and the current mask.
- *(g)* Generate the regions for the current row by first propagating the regions of interconnected cells from the row above and, secondly, creating new regions for the remaining cells (as described above).
- *(h)* Reuse the digits used in the row above for connected regions. For the row's own regions, generate the Cartesian product of the sets containing digits from 0 to 9 so that each new region is assigned a digit.
- *(i)* If no new Cartesian product can be generated, go back to *(c)* to try a new mask.
- *(j)* If the generated digits result in at least one number starting with zero, revert to step *(h)* until all numbers start with nonzero digits.
- *(k)* Verify that digits from adjacent regions are orthogonally different, digits from the same region are identical, and that top row regions retrospectively connected to the current row are identical (see above). If not, return to step *(h)* until they meet this condition.
- *(l)* Ensure that the numbers comply with the row clue. If not, revert to step *(h)*.
- *(m)* At this stage, the row satisfies all constraints (regarding regions, digits, and clues). Consequently, the candidate can be augmented with this new row.
- *(n)* A new solving goroutine is subsequently spawned to try solving the augmented candidate, starting from step *(a)*.

Conclusion

While solving this puzzle, I came to the conclusion that:

- The current Z3 library for Go is suboptimal. I would love to see a more comprehensive implementation for Go.
- It would also perhaps have been easier to use a language like Python with more robust SMT libraries. I believe some of the people who solved the puzzle may have used this approach.
- I implemented a multithreaded solver because Go makes it so convenient to use goroutines, but, in retrospect, it was unnecessary.

- Solving parts of the puzzle and propagating augmented candidates forward is a good alternative to the classic tree exploration and backtracking combination, as it simplifies the programmatic flow in some cases, especially in a multithreading context.

Series and Companion Articles

Jane Street Puzzles (glthr.com/series/jane-street-puzzles)

#	Article	Address
1	**Solving "Number Cross 4" (Jane Street Puzzle, May 2024)** (this article)	glthr.com/solving-number-cross-4-jane-street-puzzle-may-2024
2	Solving "Knight Moves 6" (Jane Street Puzzle, October 2024)	glthr.com/solving-knight-moves-6-jane-street-puzzle-october-2024

How to cite this article

> Guillaume Lethuillier, "Solving "Number Cross 4" (Jane Street Puzzle, May 2024)", glthr.com, June 3, 2024.
> https://glthr.com/solving-number-cross-4-jane-street-puzzle-may-2024

For BibTeX:

```
@misc{lethuillier2024number,
  author = {Lethuillier, Guillaume},
  title  = {Solving "Number Cross 4" (Jane Street Puzzle, May 2024)},
  year   = {2024},
  month  = jun,
  url    = {https://glthr.com/solving-number-cross-4-jane-street-puzzle-may-2024},
  note   = {glthr.com}
}
```

This document was typeset with LaTeX from the article published at https://glthr.com/solving-number-cross-4-jane-street-puzzle-may-2024, which remains the reference version (as updated on November 11, 2026).