How many loops can you slither around?

Nikoli the snake wants to slither along a loop through a four-by-four grid of points. To form a loop, Nikoli can connect any horizontally or vertically adjacent points with a line segment. However, Nikoli has certain standards when it comes to loop construction. In particular:

- The loop can never cross over itself.
- No two corners of the loop can meet at the same point.
- Once Nikoli has crossed the connection between two points, Nikoli can't cross it again (in either direction).

For example, the following two constructions are valid loops:

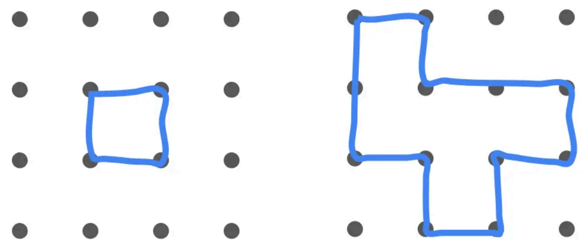


Figure 1: Left: 4-by-4 grid of dots. The middle four points are connected to form a blue square. Right. Another 4-by-4 grid of dots. 12 of the dots are connected to form a polygon.

Meanwhile, the following three constructions are not valid. The one on the left crosses over itself, the one in the middle has two corners that meet at a single point, and the one on the right requires Nikoli to pass over the same line segment twice.

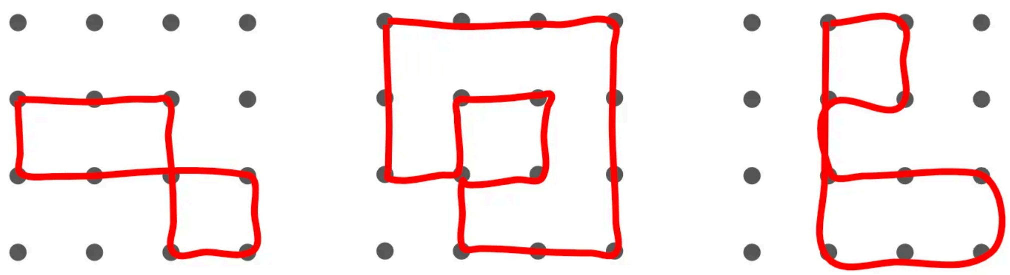


Figure 2: Left: 4-by-4 grid of dots. There's a red path shown, but it crosses over itself at one of the dots. Middle: Another 4-by-4 grid of dots. Another red polygonal path is shown, but two corners of the polygon coincide at a single point. Right: Another 4-by-4 grid of dots. This path is not a polygon, as it traverses the same edge between two points twice.

How many unique loops can Nikoli make on the four-by-four grid? (For any given loop, Nikoli can travel in two directions around it. However, these should still be counted as a *single* loop.

Solution

Every loop that Nikoli can make can be considered a **simple cycle** (a cycle with no repeated vertices except for the beginning and the ending vertex) on a 4×4 **grid graph**.

Python Code

The code below plots all the **213** cycles on a 4×4 grid graph.

```python
import networkx as nx
import matplotlib.pyplot as plt
import itertools

G = nx.grid_2d_graph(4, 4)
cycles = list(nx.simple_cycles(G))

grid_rows = 11
grid_cols = 20
num_grids = grid_rows * grid_cols
fig_width = 20
fig_height = fig_width * grid_rows / grid_cols
plt.figure(figsize=(fig_width, fig_height))
for i, cycle in enumerate(cycles):
    if i >= num_grids:
        break
    row = i // grid_cols
    col = i % grid_cols
    plt.subplot(grid_rows, grid_cols, i + 1)
    pos = {(x, y): (y, -x) for x, y in G.nodes()}
    nx.draw(G, pos, with_labels=False, node_size=10, edge_color='white')
    edges = [(cycle[j], cycle[j+1]) for j in range(len(cycle) - 1)]
    edges.append((cycle[-1], cycle[0]))
    nx.draw_networkx_edges(G, pos, edgelist=edges, edge_color='red')
plt.tight_layout()
plt.show()
```

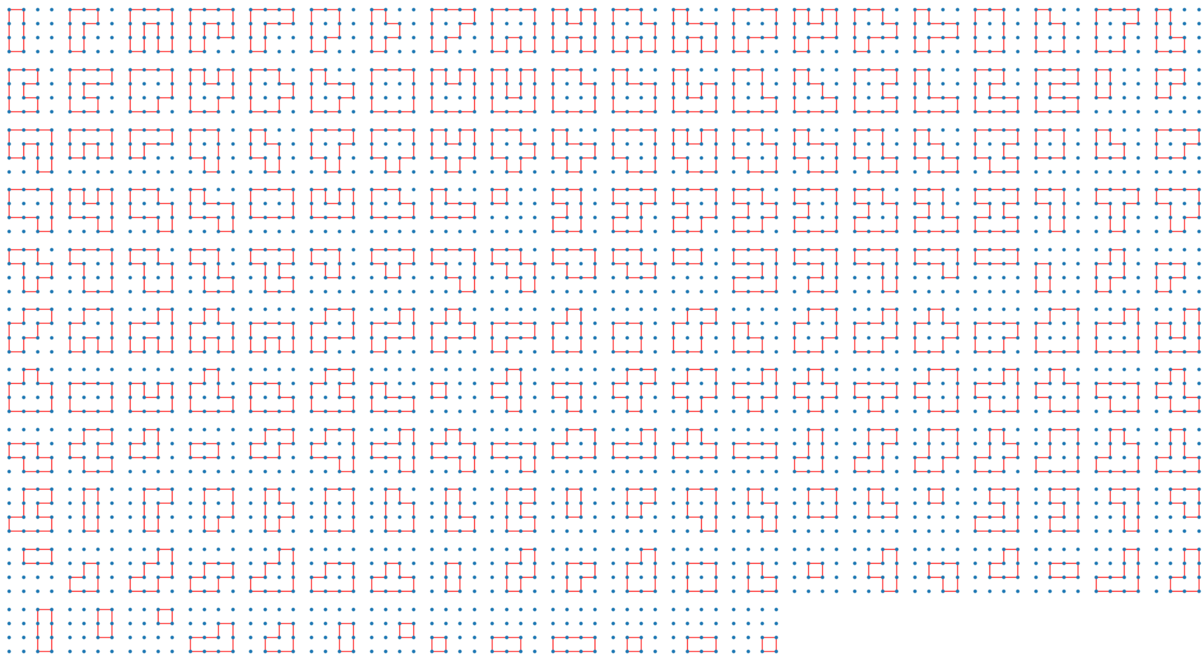


Figure 3: All unique loops Nikoli can make on a 4 × 4 grid

Slitherlink

In a Slitherlink, you connect adjacent points in a grid to form a loop that does not self-intersect, as described above. But what makes a Slitherlink a puzzle is that numbers are provided in some of the spaces between four grid points. These numbers specify how many of the four surrounding edges are present in the desired loop. Then it's entirely up to you to draw the loop.

For example, here's a puzzle I encountered on one of the previously linked sites, as well as the solution. (I marked red x's where I knew there couldn't be any edges—a helpful strategy when solving such puzzles.)

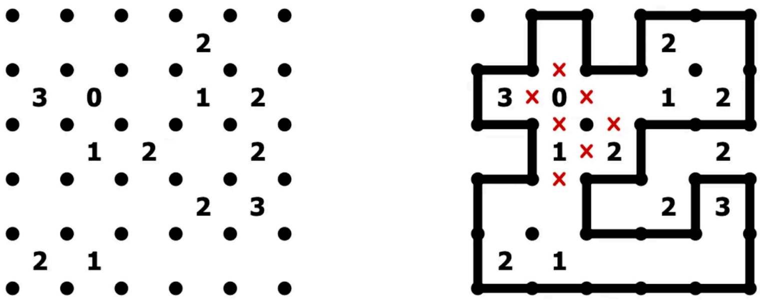


Figure 4: Left: A 6-by-6 grid of dots. Among them are several numbers ranging from 0 to 3. Right: The solved Slitherlink, given the numbers in the grid on the left.

How many distinct Slitherlink puzzles can you create on a four-by-four grid of points? Each puzzle consists of a placement of numbers (from 0 to 4) between grid points, and must result in exactly one loop. Note that multiple distinct puzzles can result in the same loop, but again, each puzzle itself can have only one loop solution.

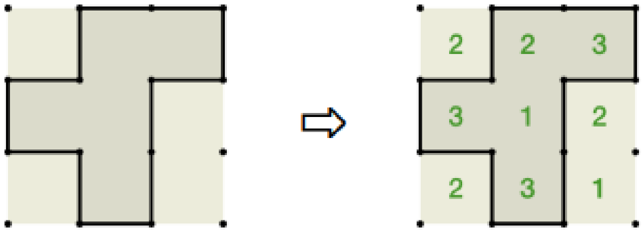
Solution

Here are two ways in which we could tackle the problem:

- We could start with a random configuration (there are 6^9 configurations) of numbers and check whether the configuration yields a unique loop.
- We could start with one of the loops that we identified in the previous section and identify all the different configurations of numbers that result in **only** that loop.

The first approach seems much more complicated and doesn't leverage the work that we have done in identifying all possible valid loops so we will take the second approach.

We start with a loop and then calculate the number of edges that this loop has in each cell of the grid. Let us call this the **canonical signature** of the loop.



We immediately notice the following:

- For each loop, every valid Slitherlink puzzle that leads to that loop has to be a subset of the canonical signature.
- For each loop, there can be no valid Slitherlink puzzle that is not a subset of the canonical signature.
- To ensure that a specific subset (signature) of the canonical signature of a loop leads to only that loop, we need to ensure that the subset does not match any of the subsets of the canonical signatures of all the other loops.

So here are the steps of the algorithm to generate the number of Slitherlink puzzles for a loop:

1. Generate the canonical signature of a loop.
2. Generate all the subsets/signatures of the canonical signature.
3. For each signature, increment the puzzle count by one if it doesn't match any of the signatures of the other loops.

Code walkthrough

In the function below, we take the size of the grid as an input and generate all edges for each cell identified by (r, c) where $0 \leq r, c \leq n - 1$.

```
def edges_per_cell(n):
    cell_edges = {}
    for i in range(n-1):
        for j in range(n-1):
            cell_edges[(i,j)] = [((i,j),(i,j+1)),((i,j+1),(i+1,j+1)),
                                 ((i+1,j+1),(i+1,j)), ((i+1,j),(i,j))]
    return cell_edges
```

In the function below, we take the cycle/loop represented as a list of adjacent nodes $[(r_1, c_1), (r_2, c_2), \ldots, (r_k, c_k)]$ where $0 \leq r_k, c_k \leq n$ and all the edges in the grid and return the **canonical signature** of the loop represented as a set of $3-$tuples of the form (r, c, l) where r is the row of the cell, c is the column of the cell and l, the number of edges of the loop in that cell.

```
def cycle_sig(cycle, all_cells_edges):
    edges_per_cell = defaultdict(list)
    cycle_edges = [(cycle[j], cycle[j+1]) for j in range(len(cycle) - 1)]
    cycle_edges.append((cycle[-1], cycle[0]))
    for cell, cell_edges in all_cells_edges.items():
        edges_per_cell[cell] = []
        for cell_e in cell_edges:
            for cycle_e in cycle_edges:
                if (cell_e[0], cell_e[1])==cycle_e or (cell_e[1], cell_e[0])==cycle_e:
                    edges_per_cell[cell].append(cell_e)
    num_edges_per_cell = set()
    for cell, edges in edges_per_cell.items():
        num_edges_per_cell.add((cell[0], cell[1], len(edges)))
    return num_edges_per_cell
```

In the function below, we generate the non-empty subsets of the canonical signature of all loops and store each subset/signature as a frozenset of $3-$tuples.

```
def all_cycles_sigs(all_cells_edges):
    def all_subsets(s):
        s_list = list(s)
        subsets = set()
        for r in range(1, len(s) + 1):
            for combo in combinations(s_list, r):
                subsets.add(frozenset(combo))
        return subsets

    sigs = {}
    for cycle in cycles:
        sigs[cycle] = all_subsets(cycle_sig(cycle, all_cells_edges))
    return sigs
```

In the function below, we precompute for each loop, the set of non-empty subsets of the canonical signatures of all the **other** loops. This is for optimizing the time to check whether a signature for a loop matches any of the signatures of all the other loops.

```
def other_cycles_sigs(all_cycles_sigs):
    all_other_sigs = {}
    for cycle in cycles:
        sigs = set()
        for c in cycles:
           if c != cycle:
                for s in all_cycles_sigs[c]:
                    sigs.add(s)
        all_other_sigs[cycle] = sigs
    return all_other_sigs
```

In the function below, we implement the algorithm described at the beginning of the section for every cycle/loop.

```
def num_puzzles_per_cycle(cycles, all_cells_edges):
    cycles_sigs = all_cycles_sigs(all_cells_edges)
    other_sigs = other_cycles_sigs(cycles_sigs)
    puzzles_per_cycle = defaultdict(int)
    for cycle in cycles:
        for sig in cycles_sigs[cycle]:
            if sig not in other_sigs[cycle]:
                puzzles_per_cycle[cycle] += 1
    return puzzles_per_cycle
```

Python Code

Here is the complete code for identifying all the **41433** Slitherlink puzzles on a 4×4 grid.

```
import networkx as nx
from collections import defaultdict
from itertools import combinations

def edges_per_cell(n):
    cell_edges = {}
    for i in range(n-1):
        for j in range(n-1):
            cell_edges[(i,j)] = [((i,j),(i,j+1)),((i,j+1),(i+1,j+1)),
                                 ((i+1,j+1),(i+1,j)), ((i+1,j),(i,j))]
    return cell_edges


def cycle_sig(cycle, all_cells_edges):
    edges_per_cell = defaultdict(list)
    cycle_edges = [(cycle[j], cycle[j+1]) for j in range(len(cycle) - 1)]
    cycle_edges.append((cycle[-1], cycle[0]))
    for cell, cell_edges in all_cells_edges.items():
        edges_per_cell[cell] = []
        for cell_e in cell_edges:
            for cycle_e in cycle_edges:
```

```python
                if (cell_e[0], cell_e[1])==cycle_e or (cell_e[1], cell_e[0])==cycle_e:
                    edges_per_cell[cell].append(cell_e)
    num_edges_per_cell = set()
    for cell, edges in edges_per_cell.items():
        num_edges_per_cell.add((cell[0], cell[1], len(edges)))
    return num_edges_per_cell


def all_cycles_sigs(all_cells_edges):
    def all_subsets(s):
        s_list = list(s)
        subsets = set()
        for r in range(1, len(s) + 1):
            for combo in combinations(s_list, r):
                subsets.add(frozenset(combo))
        return subsets

    sigs = {}
    for cycle in cycles:
        sigs[cycle] = all_subsets(cycle_sig(cycle, all_cells_edges))
    return sigs


def other_cycles_sigs(all_cycles_sigs):
    all_other_sigs = {}
    for cycle in cycles:
        sigs = set()
        for c in cycles:
           if c != cycle:
                for s in all_cycles_sigs[c]:
                    sigs.add(s)
        all_other_sigs[cycle] = sigs
    return all_other_sigs


def num_puzzles_per_cycle(cycles, all_cells_edges):
    cycles_sigs = all_cycles_sigs(all_cells_edges)
    other_sigs = other_cycles_sigs(cycles_sigs)
    puzzles_per_cycle = defaultdict(int)
    for cycle in cycles:
        for sig in cycles_sigs[cycle]:
            if sig not in other_sigs[cycle]:
                puzzles_per_cycle[cycle] += 1
    return puzzles_per_cycle

G = nx.grid_2d_graph(4, 4)
cycles = list(nx.simple_cycles(G))
all_cell_edges = edges_per_cell(4)
total_num_puzzles = 0
for c,n in num_puzzles_per_cycle(cycles, all_cell_edges).items():
    total_num_puzzles += n
print(total_num_puzzles)
```