A Brute Force Algorithm for Hamiltonian Cycles in Square Grids

Suran Warnakulasooriya

Contents

1 Introduction **2**

1.1 The Königsberg Bridges . . . 2

1.2 Eulerian & Hamiltonian Cycles . . . 2

1.3 Number of Directed Graphs . . . 4

1.4 Time Complexity . . . 6

2 Pseudocode for Hamiltonian Cycles **7**

2.1 Why We Use Pseudocode . . . 7

2.2 Fallback & Initialization . . . 7

2.3 Sub-Functions . . . 9

2.4 `HamSquares` Function . . . 16

2.5 Results for the 2×2 Grid . . . 31

2.6 Results for the 4×4 Grid . . . 31

Abstract: Within a grid comprised of links and nodes arranged as a square matrix, paths can be made that traverse every node exactly once. These are Hamiltonian paths. Paths that end where they begin are cycles. Square grids with an odd number of nodes on each side do not possess Hamiltonian cycles. A brute force algorithm to find Hamiltonian cycles in square grids with an even number of nodes on each side is presented in this paper. By finding the number of neighbors per node, we can find all combinations of one to one linkages between nodes. The algorithm then traverses every path and logs the nodes it has reached. If every node in the list of visited nodes is unique and the path ends where it began, then the path is a Hamiltonian cycle. Once found, the cycles are listed and plotted. Using the algorithm, we find that a 2×2 grid posses only 1 unique Hamiltonian cycle and a 4×4 grid has 6. The algorithm would take over 90 years to compute Hamiltonian cycles for a 6×6 grid or greater. Therefore, finding Hamiltonian cycles with the brute force method takes exponential time and is practically infeasible when the side length has six or more nodes.

August 12, 2020

1 Introduction

1.1 The Königsberg Bridges

In the southwest corner of the Baltic Sea, nested between Poland and Lithuania, is the Russian exclave of Kaliningrad. In the 13th century, it was under Prussian rule and had the name of Königsberg. Here, the two islands Kneiphof and Lomse are connected to each other and the mainland via seven bridges (Figure 1). A challenge arose involving these bridges: is it possible to create a path that crosses each bridge exactly once? Reaching an island or mainland bank other than via a bridge or accessing any bridge without crossing to its other end is forbidden. The story of Hamiltonian cycles starts with the problem of Königsberg's seven bridges.

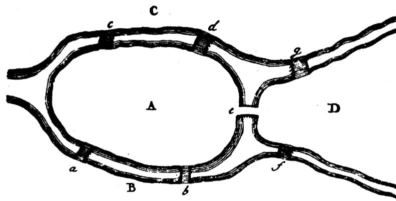


Figure 1: Euler's sketch of the Königsberg bridges (From Wikimedia Commons).

1.2 Eulerian & Hamiltonian Cycles

The Königsberg bridge problem was solved by Leonhard Euler in 1736, who accomplished a multitude of mathematical feats throughout his lifetime. He was the first to denote a function with the notation $f(x)$. Euler proved that the Königsberg bridge problem has no solutions. His proof required the spearheading of a new form of mathematics, named graph theory. Graph theory is the study of structures comprised of nodes (also called vertices or points) which are connected by links (also called lines or edges). We will use the terms *node* and *link* throughout this paper. With the terminology defined, Euler managed to simplify the banks and islands to four nodes, and the bridges to seven links (Figure 2). With the graph created, Euler proved that there is no path that crosses each bridge once. A solvable bridge problem would have at least one Eulerian path: a path that traverses every link exactly once.

Euler defined two types of graph: directed and undirected (Figure 2 falls under the latter). A graph is undirected if every link can be accessed from either node. Directed graphs have at least one link that can be accessed via only one of their nodes. The degree of a node is the number of links connecting to that node. Euler also defined the notion of connectedness: a graph is connected if it is self-contained, with no nodes or links being disconnected. Figure 3 is an example of a disconnected graph, with 3 self-contained systems. In the case of Figure 3, each self-contained graph is referred to as a component or subgraph, while the full figure is simply the graph or supergraph. Of course, if there are multiple components within the supergraph,

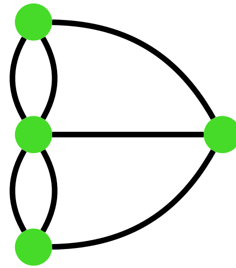


Figure 2: The Königsberg bridges when reduced to nodes and links.

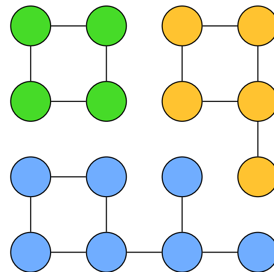


Figure 3: A disconnected graph: the three subgraphs are individual components while the full figure is the supergraph.

then Eulerian cycles do not exist, since regardless of which component we start on, we will not be able to traverse every other link. The graph depicted in Figure 2 is the supergraph, with a single component, so Eulerian paths are not immediately impossible.

In the case of the Königsberg bridges, the only links present are the bridges, so the nodes (landmasses) can only be accessed via a bridge. Euler observed that assuming the node is not at the beginning or end of the path, the number of times a landmass is entered and exited are equal. In order for each link to be traversed, every non-terminal node must have a nonzero even degree. There can be at most 2 nodes of odd degree for an Eulerian path to still exist, those nodes being the start and/or the end. All four landmasses have odd degree, and so Euler declared that no Eulerian paths exist in the problem of Königsberg's seven bridges.

Euler's solution to the bridge problem is considered as the first theorem of graph theory. Euler's lack of concern for the exact position of the nodes and attention to the links between them laid the groundwork for the mathematical branch of topology, which famously disregards the rigid shape of figures and focuses on the flow between the vertices.

Over a century later, in 1857, another graph theory-related question arose. It was a game invented by the Irish mathematician William Rowan Hamilton. He dubbed it the icosian game: a challenge to find a Hamiltonian cycle along the edges of a dodecahedron. A Hamiltonian path is quite similar to an Eulerian path, the only difference being that a Hamiltonian path must reach every node exactly once, regardless of how many times a link is crossed. A Hamiltonian cycle is simply a Hamiltonian path that ends where it starts.

1.3 Number of Directed Graphs

In this paper, we will write a brute force algorithm to compute and generate Hamiltonian cycles in a square grid, finding all possible directed graphs in a grid formatted as an $m \times m$ matrix. After finding these graphs, we must traverse them for m^2 steps to verify which of them are Hamiltonian cycles. To find the number of directed graphs, we must consider that each node has a certain number of neighbors. Any node in the grid can be one of three types: a corner, an edge, or a center. Corners, being blocked off in two directions, have two neighbors. Edges, blocked off in one direction, have three neighbors. And centers are not blocked off in any direction, and therefore have four neighbors.

Let's look at the mathematical functions $C_o(m)$, $E_d(m)$, and $C_e(m)$. These functions represent the number of corners, edges, or centers, respectively, in a square grid of side length m. With this background, we can make the following statement with ease:

$$C_o(m) = E_d(m) = C_e(m) = 0, \; m < 2.$$

If $m = 1$ then the single node in the grid fits none of the definitions of corner, edge, or center. And if $m < 1$, there is no grid at all. Now let's assume that $m \geq 2$. In this case there will always be 4 corners. An $m \times m$ square grid has 4 sides, each side having m nodes. Each side has 2 corners, so the number of edges per side is $m - 2$; therefore, the number of edges in the whole grid is $4(m - 2)$. Since this is a square grid, the number of centers will be a perfect square. The side length of the portion of the grid that contains the centers is $m - 2$, so there are $(m - 2)^2$ centers (see Figure 4). So if $m \geq 2$:

$$\begin{aligned} C_o(m) &= 4, \\ E_d(m) &= 4(m-2), \\ C_e(m) &= (m-2)^2. \end{aligned}$$

Then the number of possible directed graphs, $G(m)$, in a grid of side length m is given by

$$\boxed{G(m) = 2^{C_o(m)} \cdot 3^{E_d(m)} \cdot 4^{C_e(m)}}$$

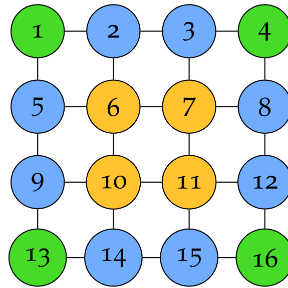


Figure 4: The corners, edges, and centers in a 4×4 grid. Corners, edges, and centers are marked green, blue, and yellow, respectively.

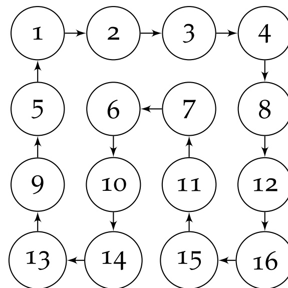


Figure 5: An instance of a 4×4 grid where every node is linked to only one neighbor.

The $m \times m$ directed graphs we will create will link every node to exactly one of their neighbors (Figure 5 is an example). This differs from a standard $m \times m$ grid, which links every node to each of their neighbors (Figure 6).

The total number of directed graphs given by the above expression for m values 2 through 6 is shown below.

m	$G(m)$
2	16
3	5,184
4	26,873,856
5	2,229,025,112,064
6	2,958,148,142,320,582,656

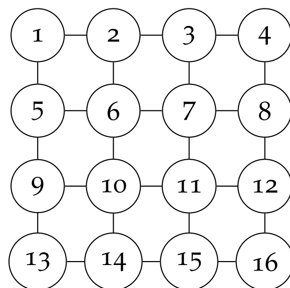


Figure 6: A standard 4×4 grid where every node is linked to every neighbor. This is in essence, the input grid for the algorithm. There are no diagonal links.

Since we are using $m = 4$ as an example throughout the majority of this paper, we will have to verify which out of the 26,873,856 paths are Hamiltonian cycles. This algorithm, in essence, creates every possible $m \times m$ directed graph there is, where every node links to exactly one neighbor, and then follows each graph for m^2 steps to verify if it is a Hamiltonian cycle.

An observation to be made here is that any square grid with an odd side length will have an odd number of nodes. A Hamiltonian cycle can be viewed as a path that leaves from a certain node and eventually returns to it after visiting every other node once. If the grid has an even side length, every move in a certain direction will have a corresponding move in the opposite direction. So for every move, there is an equal and opposite move that can be made without visiting some other node twice. A grid being odd entails that there will be a move that does not have an inverse that can be made without revisiting a node. Attempting to find Hamiltonian cycles in an odd grid therefore results in at best one node that must be reached via revisiting some other node. With this insight, we can come to the conclusion that square grids with an odd side length to not posses any Hamiltonian cycles.

1.4 Time Complexity

The time it takes to find $G(m)$ directed graphs is $\mathcal{O}(G(m))$, where $\mathcal{O}$ is the Big O notation. For every directed graph we create, we must follow it for m^2 steps to verify if it is a Hamiltonian cycle. We take m^2 steps because an $m \times m$ matrix contains m^2 nodes, and a Hamiltonian cycle traverses every node. Therefore, the time it takes to verify each path is $\mathcal{O}(m^2)$.

Suppose that $G(m) \cdot m^2 = V(m)$. $V(m)$ is now the number of moves needed to verify all $G(m)$ graphs. The total verification time therefore becomes

$$\mathcal{O}(G(m) \cdot m^2) = \mathcal{O}(V(m)).$$

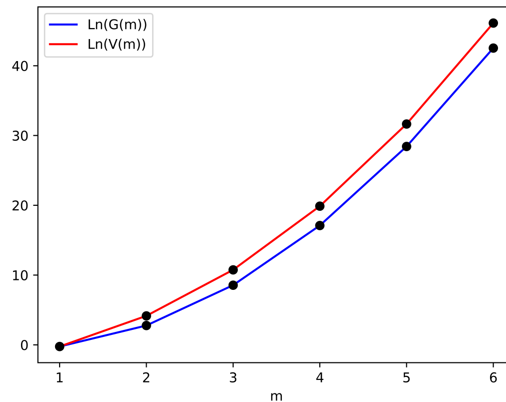


Figure 7: How $G(m)$ and $V(m)$ scale with m.

Creating and verifying $G(m)$ paths take the majority of our algorithm's runtime. As shown in Figure 7, $G(m)$ and $V(m)$ scale exponentially. Let's assume we are using a machine that can compute a path in one billionth of a second and also verify a path in one billionth of a second. If $m = 6$, it will take over 90 years to find every path through the square grid, and it will take over 3 millennia to verify all of them. Therefore, finding Hamiltonian cycles in a square grid of 6×6 or larger by this brute force method is not feasible.

2 Pseudocode for Hamiltonian Cycles

2.1 Why We Use Pseudocode

Regardless of what language we ultimately use, the logic of the program must be well-defined and should remain consistent across languages. This section will cover the Python code in English form and will be aided by flowcharts, diagrams, and snippets of the Python shell. We will refer to a Hamiltonian cycle as a hamcycle for brevity.

2.2 Fallback & Initialization

The major function of the program is named `HamSquares`, since we are finding hamcycles in a square grid. `HamSquares` requires the use of two built-in Python libraries: `itertools` for finding every possible path, and `matplotlib.pylot` to plot the hamcycles once they are found.

Most functions take in a number of arguments and perform processes involving those arguments. In most cases the arguments must be of a specific type or else the function will not be able to

operate on them correctly. The `HamSquares` function takes in a single argument, m, which is the side length of the square grid in which the hamcycles must be found. The processes within `HamSquares` can only operate without error when m is an integer, so we must check if m is indeed an integer.

If m is an integer, the function will check if m is odd or less than 1. Square grids with an odd side length does not possess hamycles, and grids with side lengths less than 2 cannot be traversed. So, if $m < 1$ or if the division of m and 2 leaves a remainder of 1, then the function terminates and returns `None`. If m is a positive even, the function continues. If m is a string of an integer, then it will be converted into numeric form and the function continues. If m cannot be converted into numeric form, the function terminates and returns `None`.

Examples of arguments accepted by `HamSquares`:

$$2, ''4''.$$

Examples of arguments rejected by `HamSquares` (i.e., the function immediately returns `None`):

$$''hello'', 0, -1, 7, [1, 2, 3].$$

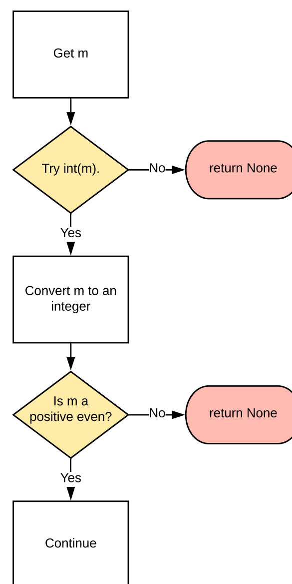


Figure 8: The fallback portion of `HamSquares` as a flowchart.

As an example, we will assume that we are calling `HamSquares` on the integer 4 (i.e., $m = 4$). This will pass through the fallback portion without any returns. The first variable is our currently

empty grid, which we will name `grid`. To populate `grid`, we create a list `n` of natural numbers up to m^2. We then create a new empty list `tempRow`. `tempRow` is the current row in the grid. Using a `for` loop, we append the next element in `n` to `tempRow`. If this element is a multiple of m, the contents of `tempRow` will be appended to `grid` as a list and `tempRow` will be cleared, since if the element in `n` is a multiple of m then we are at the end of that row. Once this process is over, `tempRow` is empty and will not be used again. `grid` is now a list of lists, where each sublist is a row and an index in a row is a column. Since this is a square grid, the length of each row (columns per row) and the number of rows will be both equal to m.

```
>>> n
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
>>> grid
[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]
```

It is difficult to visualize how we can traverse a list of lists when it is formatted in a straight line. The solution to this problem, among many others, is why we will define multiple smaller functions to be used by `HamSquares`.

2.3 Sub-Functions

We need a way to display `grid` in the form of a grid, not a line. For that we define our first sub-function, `display`, which takes a single list as its argument. The list does not need to be a list of lists for `display` to work. `display` simply prints each element in the list one at a time. Since the elements in `grid` are the rows, which are lists of their own, `display` will print each row one at a time (see Figure 9 for the flowchart).

```
>>> display([1,2,3,4])
1
2
3
4
>>> display(grid)
[1,2,3,4]
[5,6,7,8]
[9,10,11,12]
[13,14,15,16]
```

In essence, `grid` transforms from a list such as

$$[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]]$$

to

$$\begin{array}{l} [\ [\ 1\ ,\ 2\ ,\ 3\ ,\ 4\ ]\ , \\ \ \ [\ 5\ ,\ 6\ ,\ 7\ ,\ 8\ ]\ , \\ \ \ [\ 9\ ,\ 10\ ,\ 11\ ,\ 12\ ]\ , \\ \ \ [\ 13\ ,\ 14\ ,\ 15\ ,\ 16\ ]\ ]. \end{array}$$

This layout is identical to Figure 6, and we will substitute displaying `grid` with this figure for clarity.

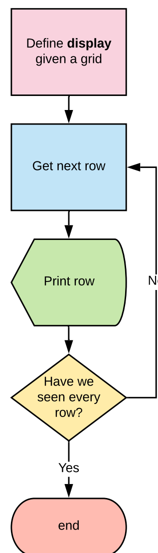


Figure 9: The `display` function as a flowchart.

Next we define `reverse`, a function that takes in a single list and returns the same list with its elements in reversed order. Python has the built-in command `list.reverse()`; however, this command is problematic. `list.reverse()` is an in-place function, it reverses the contents of the list itself instead of returning a reversed version, and therefore returns `None`. If we wanted to check the properties of the reverse of a list using `list.reverse()`, we would have to reverse the list again to return it to its original state. Calling `list.reverse()` in another function call or `if` statement is also a bad idea, since `None` is returned, not the reversed list. An example of this issue is shown below.

```
>>> L = [1,2,3,4]
>>> L.reverse()
>>> L
[4,3,2,1]
>>> print(L.reverse())
None
```

We want the function to simply return the reversed list without changing the list itself. For that, we define `reverse`. `reverse` creates an empty list `rev` to fill and return. A `for` loop moves backwards through the list and appends its elements to `rev`. After the loop is over, `rev` is returned as a completely reversed version of the original list. See the following example, as well as Figure 10 for the flowchart.

```
>>> reverse([])
[]
>>> reverse([1])
[1]
>>> L = [1,2,3,4]
>>> reverse(L)
[4,3,2,1]
>>> L
[1,2,3,4]
>>> print(reverse(L))
[4,3,2,1]
```

The next set of functions find the nodes above, below, left, and right of a given node. There is no need to find diagonal neighbors since diagonal travel is impossible (there are no diagonal links). There is one neighbor function for each direction (north, south, east, and west). Using the north function as an example, the function takes the row and column of the node as inputs and checks if the neighbor above is on the grid. If the node is on the top row, there is no higher row, so the function returns `False`. If `False` is not returned, the neighbor exists, so the function will return the north neighbor. The neighbor functions are named `N`, `S`, `E`, and `W` (refer to Figure 11 for the flowchart).

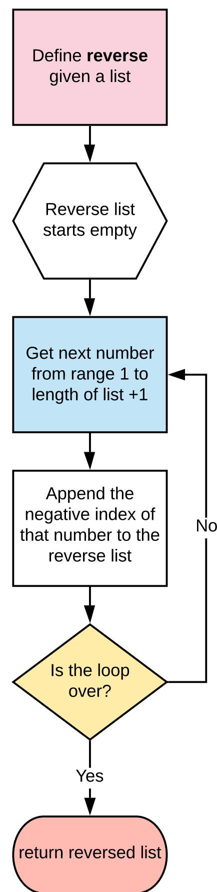


Figure 10: The `reverse` function as a flowchart.

```
>>> N(0,0)

>>> N(2,1)
6
>>> S(2,1)
14
>>> E(2,1)
11
>>> W(2,1)
9
```

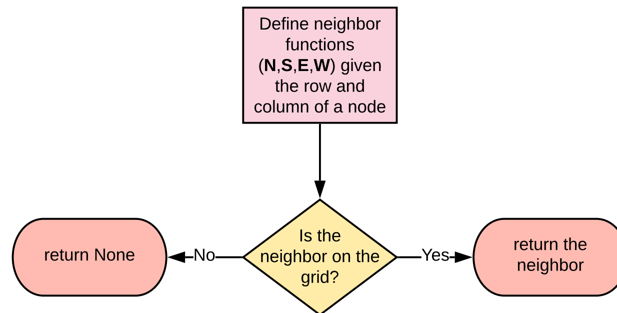


Figure 11: The neighbor functions as a flowchart.

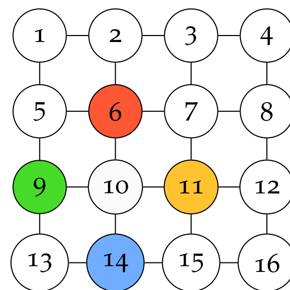


Figure 12: The neighbors of node 10 in a 4×4 grid.

Note that the colors in the snippet above only exist to connect the returns of the functions to the neighbors on the grid (Figure 12). Using these colors we can see how the neighbor functions view the grid when called on (2,1). Meaning the first column of the second row in `grid` (counting rows and columns from 0), which is node 10.

We need to find our paths before verifying if they are hamcycles. To find our paths, we create `nodeType`. When called, `nodeType` creates a list of the four neighbor functions, named `fcns`, and sets `score`, the number of neighbors found, to 0. The function iterates through `fcns` and applies the current neighbor function to the row and column that was provided. If the neighbor exists, `score` increases by 1. Once the iterating is over, `score` is checked to see if it is 2, 3, or 4. `nodeType` returns corner, edge, or center, respectively based on the result (see Figure 13 for the flowchart).

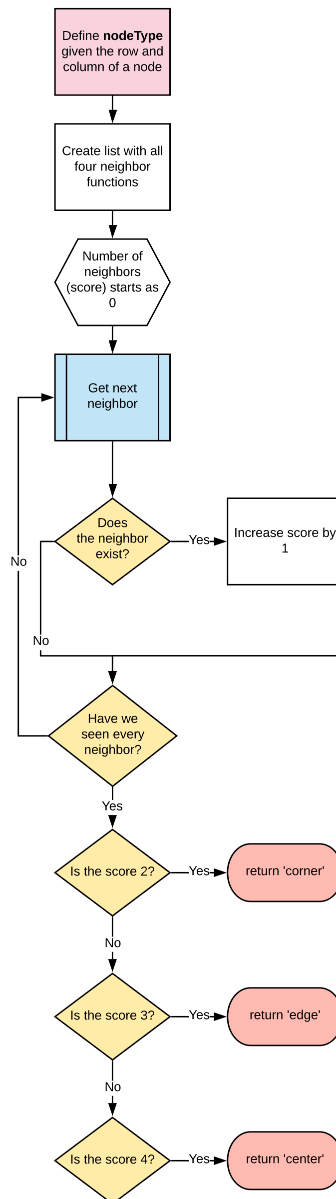


Figure 13: The `nodeType` function as a flowchart.

Last of the functions is `checkUnique`, to see if all elements in a given list are unique. When called, `seen`, the list of elements seen by the function, is created empty. The function gets the next element from the input list and checks if it is in `seen`. If the element has been seen before, the function terminates and returns `False`. Otherwise, the element is appended to `seen` and the next element is checked. If the loop has gone through every element, then it means none of those elements triggered a return, so the function returns `True` (see Figure 14 for the flowchart).

```
>>> checkUnique([1,1])
False
>>> checkUnique(["hello", "hello"])
False
>>> checkUnique([1,2,3,4,5,4,7])
False
>>> checkUnique([])
True
>>> checkUnique([1])
True
>>> checkUnique(["hello", "h"])
True
>>> checkUnique([1,2,3,4])
True
```

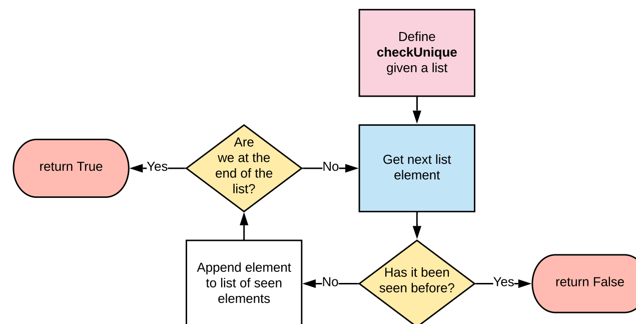


Figure 14: The `checkUnique` function as a flowchart.

2.4 `HamSquares` Function

With the fallback and sub-functions now defined, we can move on to the meat of `HamSquares`. A hamcycle travels through every node on the grid and ends where it starts, and therefore any hamcycle in list-form will be $m^2 + 1$ elements long. Using $m = 4$ as our example, the hamcycle list will be 17 elements long. `HamSquares` is a brute force function and must find every possible path (not necessarily Hamiltonian) that can be made in 17 moves, and then verify which of those paths are hamcycles. To find every path we must find every way to travel the graph.

We start again with `fcns`, as well as two new variables, `starts` and `dests`. `dests` is the list of nodes one can travel to from a given node. `starts` is every possible start position, meaning every node in `grid`, therefore `starts = n` (where `n` is the list of natural numbers up to m^2). Once again we use a loop to find every node. For each node, the list `neighbors` starts empty, and is populated by the neighbors of the current node, which are found by looping through `fcns`. `neighbors` is then appended to `dests`. Once this process is completed for every node:

$$\texttt{starts} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]$$

$$\texttt{dests} =$$

$$
\begin{array}{l}
[\ [5, 2]\ ], \\
\quad [6, 3, 1], \\
\quad [7, 4, 2], \\
\quad [8, 3,], \\
\quad [1, 9, 6], \\
\quad [2, 10, 7, 5], \\
\quad [3, 11, 8, 6], \\
\quad [4, 12, 7], \\
\quad [5, 13, 10], \\
\quad [6, 14, 11, 9], \\
\quad [7, 15, 12, 10], \\
\quad [19, 14,], \\
\quad [10, 15, 13], \\
\quad [11, 16, 14], \\
\quad [12, 15]\ ].
\end{array}
$$

`starts` and `dests` now compliment each other. Any index in `starts` corresponds with the same index in `dests`. Keep in mind that whenever we loop through each node, we find each node in the same order every time. For example, for the start node 1 (index 0 in `starts`), the possible destinations (neighbors) are 5 and 2 (index 0 in `dests`). If the start node is 9, the destinations are 5, 13, and 10. Now we can zip `starts` and `dests` into a dictionary named `tree`. Calling a node from `tree` will return its neighbors. Therefore the contents of `tree` will be

1: [5 , 2]
2: [6 , 3 , 1]
3: [7 , 4 , 2]
4: [8 , 3]
5: [1 , 9 , 6]
6: [2 , 10 , 7 , 5]
7: [3 , 11 , 8 , 6]
8: [4 , 12 , 7]
9: [5 , 13 , 10]
10: [6 , 14 , 11 , 9]
11: [7 , 15 , 12 , 10]
12: [8 , 16 , 11]
13: [9 , 14]
14: [10 , 15 , 13]
15: [11 , 16 , 14]
16: [12 , 15].

A dictionary like `tree` has two parts: the list of keys and the list of values. Each key has an associated value. Calling a key in a dict is the same as calling an index in a list; instead of returning the item in that index, the value is returned. In the case of `tree`, the keys are the nodes and the values are the respective neighbors (see Figure 15 for the flowchart).

```
>>> L = [1,2,3,4] # a list
>>> L[1] # index 1
2
>>> L[3] # index 3
4
>>> T = {1:3, 4:2} # a dictionary linking 1 to 3 and 4 to 2
>>> T[1] # call value of key 1
3
>>> T[4] # call value of key 4
2
```

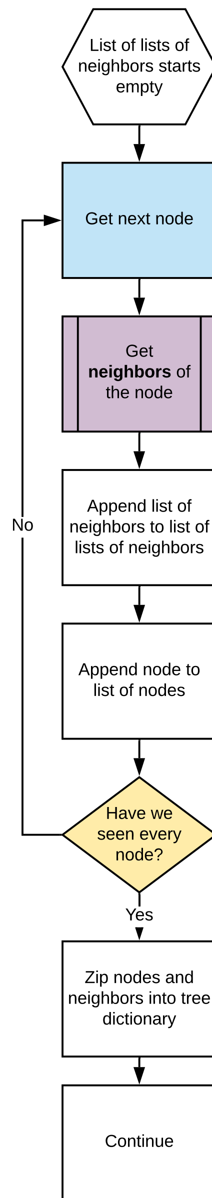


Figure 15: The process of making `tree` as a flowchart.

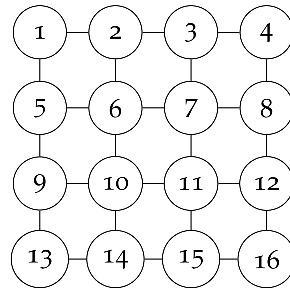


Figure 16: The graph made using the relations between the keys and values in `tree`. This is identical to Figure 6.

`tree` shows us the neighbors of each node. If we were to use `tree` to create a graph by linking keys to their values, that graph would appear as the figure above, which is identical to `display(grid)`.

Note that we are not generating any graph using `tree` but we can use `tree` to create one. `tree` provides the nodes (keys and values) and the links (relations between keys and values) necessary for a grid to form. And thus, `tree` is the grid itself in dict form. The graph that `tree` creates is undirected and gives us no way to decide what path to take. The graph, as shown above, is identical to Figure 6, which is how `HamSquares` already views it. To describe a single possible path, we need a directed graph. Specifically, we need all possible graphs where each link can only be traversed in a single direction. These graphs would be one to one instead of one to many, which means that every node is directed to one neighbor.

Creating these graphs brings us to the next phase of the `HamSquares` function. We start with the empty list `neighbornums`, destined to become a list of lists. The lists within `neighbornums` will be a mix of [0,1], [0,1,2], and [0,1,2,3]. The decision of which of these three lists to append lies on the type of each node. By calling `nodeType` on each node, we append one of the three lists depending on the result. If `nodeType` returns corner, edge, or center, we append [0,1], [0,1,2], or [0,1,2,3], respectively. [0,1] is the list form of `range(2)`, 2 being the number of neighbors that a corner has. Similarly, [0,1,2] is the list form of `range(3)`, 3 being the number of neighbors that an edge has. Centers follow the same rule and corresponds with [0,1,2,3]. Once this process is complete, `neighbornums` will be

$$[[0,1],[0,1,2],[0,1,2],[0,1],[0,1,2],[0,1,2,3],[0,1,2,3],[0,1,2],$$
$$[0,1,2],[0,1,2,3],[0,1,2,3],[0,1,2],[0,1],[0,1,2],[0,1,2],[0,1]].$$

With `neighbornums` initialized, we can create every possible directed graph. `neighbornums` is currently similar to `dests` in the sense that it tells us the type of each node but does not specify any paths. For this, we make the first and last call of `itertools.product`, which will find

every combination of list elements from `neighbornums`. In a practical sense, `itertools.product(neighbornums)` pulls one element out of each sublist in `neighbornums` and finds every possible combination of single elements from each sublist. The results are saved as the list `paths` (see Figure 17 for the flowchart). Referring back to Section 1.3, $26,873,856$ paths are being generated if $m = 4$. Since there are no hamcycles for grids with an odd side length, to spare time, it is more efficient to immediately return `None` if m is odd.

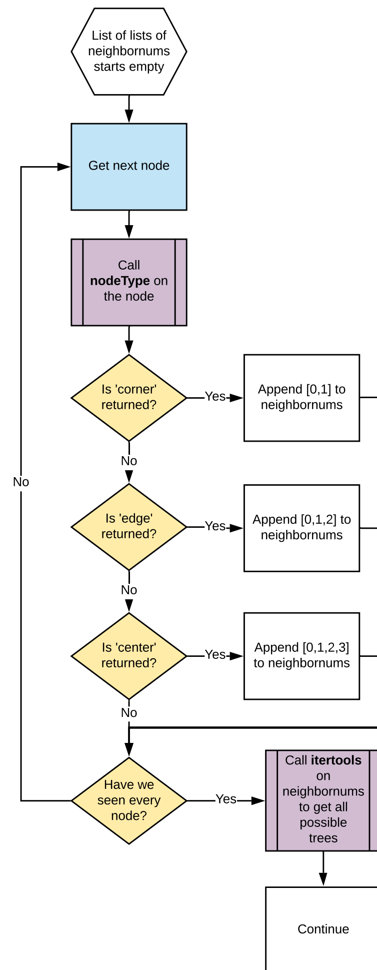


Figure 17: The process of making `paths` as a flowchart.

Each sublist in `paths` is the blueprint for a directed graph, where arriving on one node means

that we can only travel to one other node. Using a loop, we find every path in `paths`. The first element in `paths` is `[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]`, meaning that taking the 0th element from each sublist in `neighbornums` is being used to create the graph. This is zipped with every node to create `simptree`, the 1 to 1 dict we are looking for. Visually, if the first path is all 0s, the values in `simptree` are the neighbors highlighted in pink, and each node is a key.

	0	1	2	3	
1:	[5 ,	2]			0
2:	[6 ,	3 ,	1]		0
3:	[7 ,	4 ,	2]		0
4:	[8 ,	3]			0
5:	[1 ,	9 ,	6]		0
6:	[2 ,	10 ,	7 ,	5]	0
7:	[3 ,	11 ,	8 ,	6]	0
8:	[4 ,	12 ,	7]		0
9:	[5 ,	13 ,	10]		0
10:	[6 ,	14 ,	11 ,	9]	0
11:	[7 ,	15 ,	12 ,	10]	0
12:	[8 ,	16 ,	11]		0
13:	[9 ,	14]			0
14:	[10 ,	15 ,	13]		0
15:	[11 ,	16 ,	14]		0
16:	[12 ,	15].			0

1:	5
2:	6
3:	7
4:	8
5:	1
6:	2
7:	3
8:	4
9:	5
10:	6
11:	7
12:	8
13:	9
14:	10
15:	11
16:	12

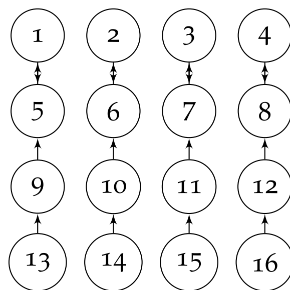


Figure 18: The directed graph created by `simptree`, where `path` is `[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]`. Every node links to exactly one neighbor. This graph is not a hamcycle.

Figure 18 is the corresponding directed graph created by `simptree`. Now being directed, we can check if each graph made by `simptree` contains a hamcycle.

Here is another iteration of `simptree` where `path = [0,2,0,1,2,3,1,0,1,0,3,2,0,1,1,0]`. The resulting directed graph is shown in Figure 19.

	0	1	2	3	
1:	[5 ,	2]			0
2:	[6 ,	3 ,	1]		2
3:	[7 ,	4 ,	2]		0
4:	[8 ,	3]			1
5:	[1 ,	9 ,	6]		2
6:	[2 ,	10 ,	7 ,	5]	3
7:	[3 ,	11 ,	8 ,	6]	1
8:	[4 ,	12 ,	7]		0
9:	[5 ,	13 ,	10]		1
10:	[6 ,	14 ,	11 ,	9]	0
11:	[7 ,	15 ,	12 ,	10]	3
12:	[8 ,	16 ,	11]		2
13:	[9 ,	14]			0
14:	[10 ,	15 ,	13]		1
15:	[11 ,	16 ,	14]		1
16:	[12 ,	15]			0

1:	5
2:	1
3:	7
4:	3
5:	6
6:	5
7:	11
8:	4
9:	13
10:	6
11:	10
12:	11
13:	9
14:	15
15:	16
16:	12

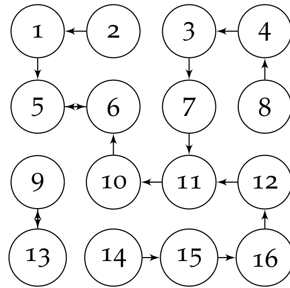


Figure 19: Another iteration of `simptree` as a graph, where `path` is [0,2,0,1,2,3,1,0,1,0,3,2,0,1,1,0]. This graph is not a hamcycle.

Here is a final example which results in a hamcycle. `path` in this case is [1,1,1,0,0,1,3,1,0,1,0,1,0,2,0,1].

	0	1	2	3	
1:	[5	**2**]			1
2:	[6	**3**	1]		1
3:	[7	**4**	2]		1
4:	[**8**	3]			0
5:	[**1**	9	6]		0
6:	[2	**10**	7	5]	1
7:	[3	11	8	**6**]	3
8:	[4	**12**	7]		1
9:	[**5**	13	10]		0
10:	[6	**14**	11	9]	1
11:	[**7**	15	12	10]	0
12:	[8	**16**	11]		1
13:	[**9**	14]			0
14:	[10	15	**13**]		2
15:	[**11**	16	14]		0
16:	[12	**15**].			1

1:	2
2:	3
3:	4
4:	8
5:	1
6:	10
7:	6
8:	12
9:	5
10:	14
11:	7
12:	16
13:	9
14:	13
15:	11
16:	15

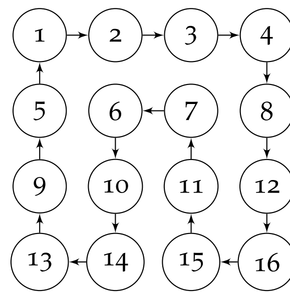
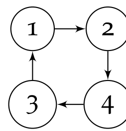


Figure 20: Another incarnation of `simptree` in graph form, where `path` is `[1,1,1,0,0,1,3,1,0,1,0,1,0,2,0,1]`. This graph is a hamcycle.

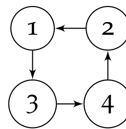
Since this is a loop, `simptree` will change every time, and so will the graph that it will practically create. Once the current `simptree` is formed, we traverse through it. A hamcycle visits every node, so any hamcycle in list form (if the first element is the last) will be $m^2 + 1$ elements long. The first element of this list will always be 1, since we always start traversing the list from node 1. With the first element always decided, we need to make m^2 moves to fully traverse `simptree`. Using a `for` loop through m^2 steps will allow us to make m^2 moves. First, we summon three new variables: `hamcycles`, `queue`, and `node`. `hamcycles` is an empty list that will contain all of the hamcycles we find. `queue` is an empty list that will contain the current path we make, and will be cleared once we create the next `simptree`. `node` is the current node we are on, which begins as 1. Using `simptree`, we travel to the next node and append that node to `queue`. Once the loop is over, we must check if `queue` is a hamcycle. In essence, we traverse `simptree` starting

from 1, moving to whichever node it tells us too, and fill `queue` with our positions.

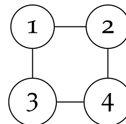
We first see if `checkUnique(queue[1:])` returns `True`. Calling `checkUnique` on the whole queue will always return `False` even if `queue` is a hamcycle, since if `queue` is a hamcycle, there is a 1 at the start and end. So we must call `checkUnique` on `queue` minus its first element. If `True` is returned, we see if the last element in `queue` is 1. If so, we make our final check. Hamcycles are closed undirected graphs, with no starting point nor direction. Each `queue` we make is directed, but a hamcycle is undirected, so if we find a `queue` that is a hamcycle, we must undirect it when we plot it (which is as simple as not placing any arrows on the links). An issue arises here: using $m = 2$ for simplicity, there are 2 cycles: [1,2,4,3,1] and [1,3,4,2,1]. The first cycle as a directed graph would be



And the second cycle would be



Removing the direction from both graphs gives



Both cycles are identical when stripped of their directions, and will result in duplicate plots. Since hamcycles are not directed, [1,2,4,3,1] and [1,3,4,2,1] must be counted as the same hamcycle. To prevent duplicates, we must check if `queue` is already in `hamcycles`, more specifically, the reverse of `queue`, since a path going clockwise will look the same as a path going counterclockwise if both are undirected. This is why we defined `reverse` to give us the reverse of any list. If `reverse(queue)` is in `hamcycles`, then `queue` is a duplicate and is not appended to `hamcycles`. If `reverse(queue)` is not in `hamcycles`, then `queue` is unique and is appended to `hamcycles` (see Figure 21 for the flowchart).

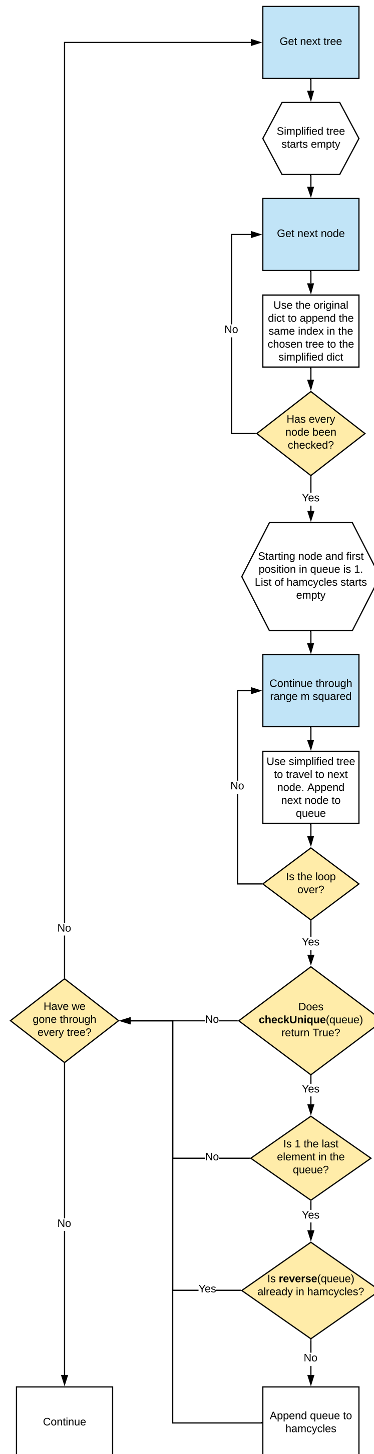


Figure 21: The process of populating `hamcycles` as a flowchart.

Lastly, we need to show what the hamcycles look like in the grid. Which requires `matplotlib.pyplot`'s ability to plot. The program refers to `matplotlib.pyplot` as `plt`. To save the images onto the local machine, we need to make sure that each plot has a different file name or else only the last plot will remain (since the file will refresh with each plot). To save different files, we create the variable `plot`: starting as 0, this variable will increase every time we generate a plot. To generate the plots, we loop through `HamSquares`. With every loop, we get the next element from `HamSquares`. The lists `x` and `y` start empty. Lopping through every node, we append the column and row to `x` and `y` respectively. Once `x` and `y` are fully populated, we can use `plt.plot()` to plot `x` and `y`. This plot alone is the hamcycle itself. We then plot the grid points on top of the cycle to clarify the positions of each node, as well as set the aspect ratio to 1:1 and remove tick marks. Once the plot is made, we save the figure with m and `plot` in its name, and then clear the figure to have a blank slate for the next plot. The directory looks like `Users/user/Desktop/cycle[m]-[plot].png` (see Figure 22 for the flowchart). Once every cycle has been plotted, we can return `display(hamcycles)` as the final result in the shell.

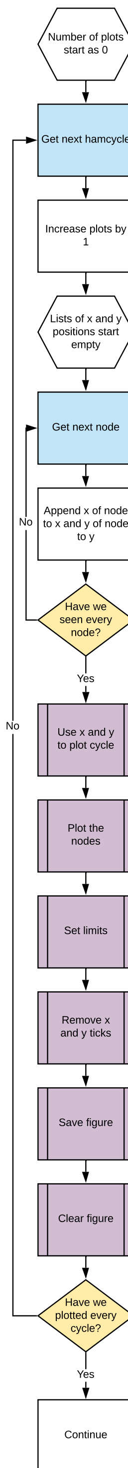


Figure 22: The process of plotting each cycle as a flowchart.

With every portion of `HamSquares` demystified, a full (but simplified) version of the full function can be expressed in a flowchart (see Figure 23).

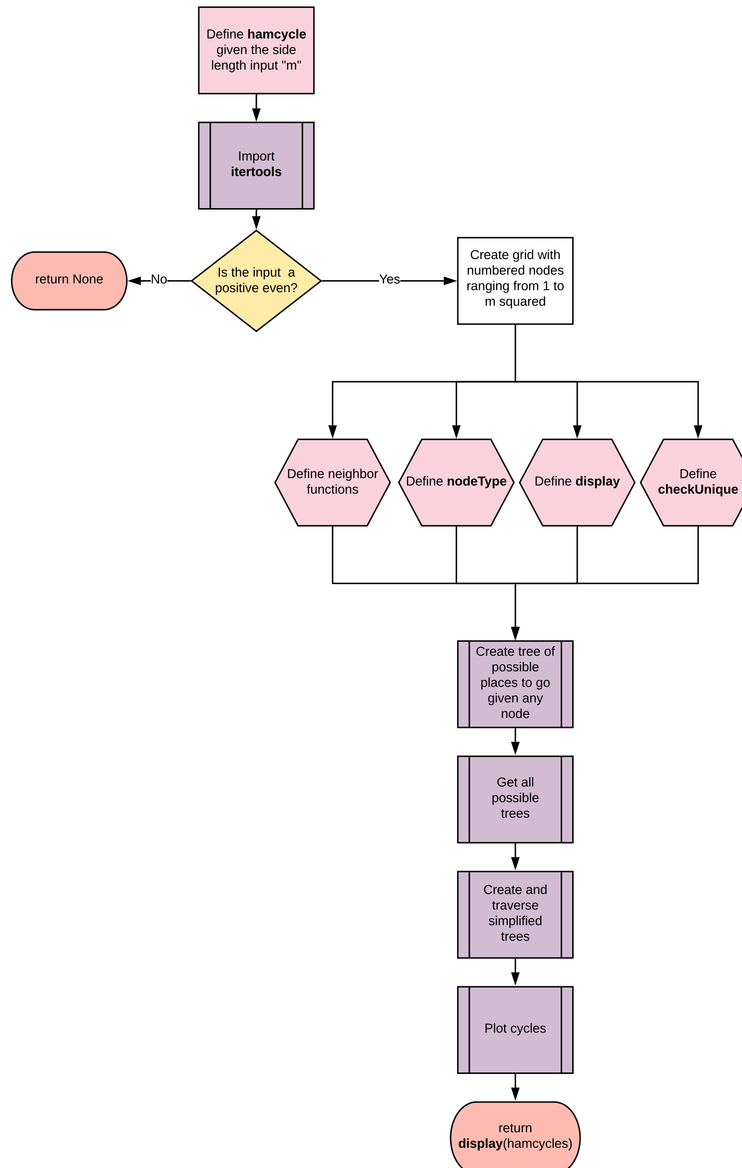


Figure 23: A simplified form of the `HamSquares` function as a flowchart.

2.5 Results for the 2×2 Grid

```
>>> HamSquares(2)
There is 1 Hamiltonian cycle in a square grid of side length 2.
[1,3,4,2,1]
```

When `HamSquares(2)` was run on a macOS version 10.15.4 with 16GB RAM/2.3 GHz Intel Quad-Core i7, the running time was roughly half of a second.

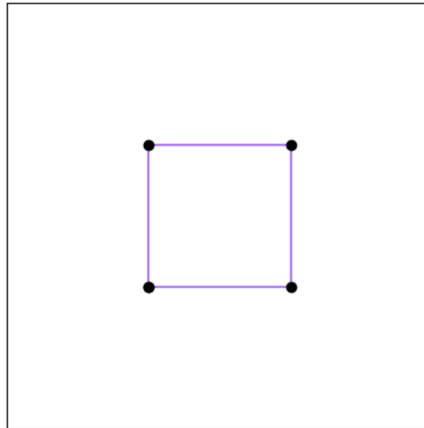


Figure 24: The hamcycle [1,3,4,2,1].

2.6 Results for the 4×4 Grid

```
>>> HamSquares(4)
There are 6 Hamiltonian cycles in a square grid of side length 4.
[1, 5, 9, 13, 14, 15, 16, 12, 8, 4, 3, 7, 11, 10, 6, 2, 1]
[1, 5, 9, 13, 14, 10, 11, 15, 16, 12, 8, 4, 3, 7, 6, 2, 1]
[1, 5, 9, 13, 14, 10, 6, 7, 11, 15, 16, 12, 8, 4, 3, 2, 1]
[1, 5, 9, 13, 14, 15, 16, 12, 11, 10, 6, 7, 8, 4, 3, 2, 1]
[1, 5, 6, 10, 9, 13, 14, 15, 16, 12, 11, 7, 8, 4, 3, 2, 1]
[1, 5, 6, 7, 11, 10, 9, 13, 14, 15, 16, 12, 8, 4, 3, 2, 1]
```

When `HamSquares(4)` was run on the same machine described in the results for 2×2, it took about one and a half minutes to execute completely.

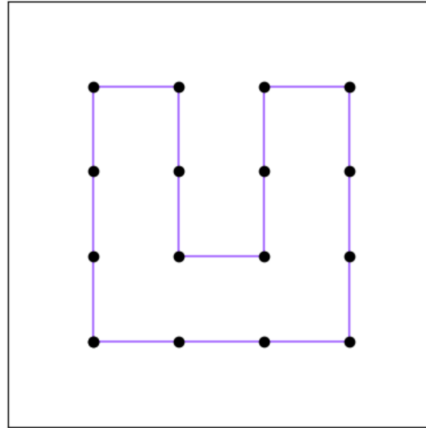


Figure 25: The hamcycle [1, 5, 9, 13, 14, 15, 16, 12, 8, 4, 3, 7, 11, 10, 6, 2, 1].

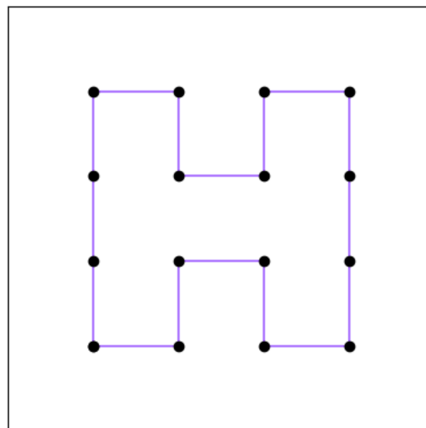


Figure 26: The hamcycle [1, 5, 9, 13, 14, 10, 11, 15, 16, 12, 8, 4, 3, 7, 6, 2, 1].

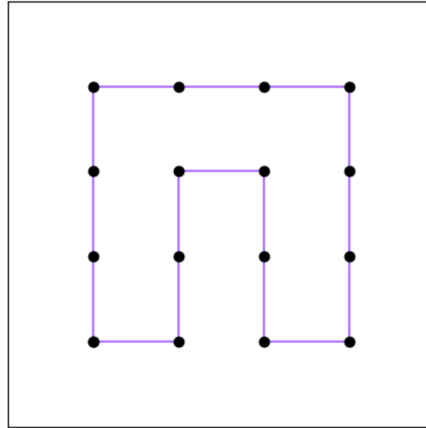


Figure 27: The hamcycle [1, 5, 9, 13, 14, 10, 6, 7, 11, 15, 16, 12, 8, 4, 3, 2, 1].

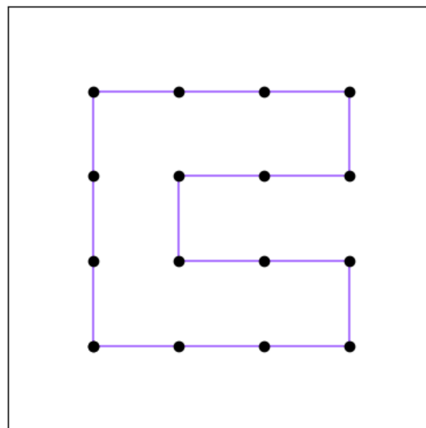


Figure 28: The hamcycle [1, 5, 9, 13, 14, 15, 16, 12, 11, 10, 6, 7, 8, 4, 3, 2, 1].

Figure 29: The hamcycle [1, 5, 6, 10, 9, 13, 14, 15, 16, 12, 11, 7, 8, 4, 3, 2, 1].

Figure 30: The hamcycle [1, 5, 6, 7, 11, 10, 9, 13, 14, 15, 16, 12, 8, 4, 3, 2, 1].