Deriving L-System Grammar from Real City Street Networks Using Pathfinding Algorithms

Aryo Wisanggeni - 13523100
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: arrrryow@gmail.com , 13523100@std.stei.itb.ac.id

***Abstract*— City street networks are inherently complex due to their organic and often unstructured development, influenced by geographic, historical, and planning factors. Modeling these networks accurately poses a significant challenge, both in terms of analytical complexity and computational cost. This paper explores am approach that combines uninformed pathfinding algorithms with Lindenmayer systems (L-systems) to simplify and compress complex urban networks into generative rule sets. Using blind search methods such as Breadth-First Search (BFS) and Depth-First Search (DFS), traversal patterns can be analyzed from real-world street layouts. These patterns are then encoded into compact, recursive L-system rules capable of regenerating the original network or extending it while preserving its underlying spatial logic. This technique offers data compression and enables lightweight generation of complex structures, making it ideal for applications in procedural content generation, simulation, and urban planning.**

Keywords—Pathfinding Algorithm; Deriving L-System; Real City Street Network

I. Introduction

City streets networks are highly complex by how they are often unorganized. They develop over long periods by geographical constraints, historical development, planning decisions, and various other factors. This results in unique patterns in different street networks all around the world as a result of those factors. Because of that, city streets present a challenge to model and simulate. Additionally, their complexity often demands substantial computational resources for storage and processing. However, accurately modeling these patterns holds significant value, especially for applications like in-game asset generation that requires to be lightweight so that users are able to run it smoothly while also be visually engaging to keep user attention. It's also crucial for future urban planning, allowing us to predict how street networks might evolve naturally without land-use restrictions. Consequently, simplifying these complex street patterns into more manageable, generative forms becomes a crucial endeavor.

Conversely, L-systems or Lindenmayer systems are a set of rules that were first developed to provide a formal description of how plants, fungi, or similarly shaped multicellular organism's growth can be modelled. These set of rules are often very simple, being only a string of usually not more than 20 characters. Yet recursively, they are able to model the growth of organisms that seemingly do not have an obvious pattern with high precision. This accuracy is partly because of nature's tendency to have fractal patterns that programmatically can be modelled with recursion. From the characteristics above, L-system's ability to produce high levels of complexity from minimal input makes them an interesting candidate for abstracting and generating spatial patterns, including those found in urban environments.

In respect to that, pathfinding algorithms are a fundamental component in analysing and navigating complex networks. These algorithms usually help in identifying efficient traversal routes within a graph of a network. But on the other hand, they also support analysis of a graph network as a whole by finding trends and key characteristics of the network. This is especially true for blind pathfinding algorithms that search the network without a heuristic leading them to a certain goal. Exploring without a goal in mind is the exact characteristic needed for the algorithm to completely explore the entire graph efficiently. Examples of these are Breadth First Search (BFS) and Depth First Search (DFS), one which goes layer-by-layer and another going through a route until it reaches an end to go back and try another route. Their searching strategy, alongside other blind pathfinding algorithm, simplifies the search process and give key insights on the network by not focusing on how they explore the graph, but by allowing developers to focus on heuristics on how to gain insight of the graph itself.

Given the complexity of city streets and the trouble it presents in simplifying it, alongside the l-system's inherent ability to generate complex patterns using simple recursive rules, a natural synergy emerges between these two domains. By using pathfinding algorithms to explore and extract traversal patterns from urban networks, we can transform the collected data into a rule-based grammar that reflects the spatial logic of real-world streets. This grammar can then be used to generate new street networks that are both natural in design and computationally lightweight. This approach has promising applications in fields such as procedural content generation for games and simulations and many other planning applications. It is with these considerations that inspires this paper to be made.

II. Theoretical Framework

A. Graph

Graph is a data structure that represents relationship between variables. A graph is made up of vertices (nodes) that are connected by the edges (lines). In applications using graphs, nodes represent discrete objects like people or places, while the edges describe a relationship between the discrete objects like routes from one place to another or people's relationships.

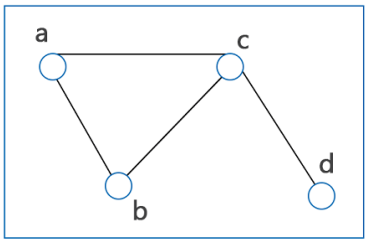

Figure 1. A graph with four vertices and four edges
Source: link

1. Types of Graphs based on Edge Weight

   a. Unweighted Graph

      Unweighted graphs are graphs that simply have the same weight for all edges, no matter how long/short they are even visually.

   b. Weighted Graph

      Weighted graphs are graphs that have values attached to every edge. These values represent the cost to use/go through said edge from one vertex to another. This value makes a path from A to B different depending on which path or set of edges is used and the weight on said edges.

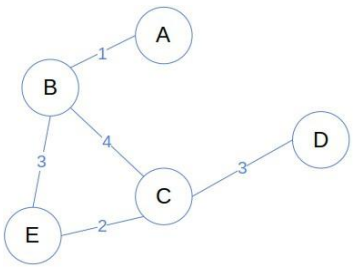

Figure 2. Weighted graph example. Source: link

B. Pathfinding Algorithms

Pathfinding algorithms are a group of algorithms with the goal of exploring a graph and finding the path from a starting point to destination point. The behavior and way of exploring the graph of each pathfinding algorithm differ and may also give different results. These distinctions have their own benefits and weakness in different problem cases. Among those different algorithms are also ones that give the most optimal path to the destination. In respect to that, pathfinding algorithms are split into two major groups: uninformed pathfinding and informed pathfinding [1]. Both of which have their own ways of finding the most optimal path.

To achieve the most optimal path or to help just finding a path toward the goal, a lot of pathfinding algorithm relies on evaluation functions f(n) to find their way towards it. The evaluation function, in a lot of algorithms, uses two other subfunctions to compound it. Those being g(n), the function to measure the cost of how far a path has gone from the start, and the other being h(n), a function to measure the cost of how much farther the goal is from the current position, usually estimated with a heuristic as most of the time we don't know how far the goal is. The composition of one or both g(n) and h(n) leads to the formation of f(n). The composition itself is determined by the algorithm that is being used.

1. Uninformed Search

   Uninformed search, also called blind search, are a group of pathfinding algorithms that find a path to the destination by blindly exploring the graph in hopes that it will get to the goal without any information about the goal while exploring [1]. It usually depends on a condition that is met when the goal itself is actually reached, but doesn't know if the search is getting closer or not to the goal while searching. Three common examples of this type of algorithm are Breadth First Search (BFS), Depth First Search (DFS), and Uniform Cost Search (UCS).

   a. Breadth First Search (BFS)

      BFS is an algorithm that searches one step at every alternative path before going deeper into one path. This approach can be represented well using layers, like the ones in figure 3, BFS will choose to explore the nodes at layer 1 first before going to layer 2. This approach allows it to guarantee the most optimal path towards the goal, as when the goal is found, it will already be at its shortest from the start. This search algorithm is completely blind and does not consider cost of each edge when traversing from node to node and is most suited for unweighted graphs. The biggest weakness of BFS is its high memory usage as it is very likely to explore most of the graph that does not need to be explored to reach the goal. In implementations, the queue data structure is used in order to achieve the behavior needed for this algorithm.

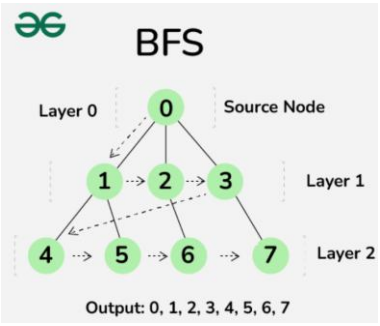

Figure 3. Diagram of BFS searching layer by layer. Source: link

b. Depth First Search (DFS)

DFS is an algorithm that explores as deep as possible along one path before backtracking to try alternative routes. As illustrated in figure 4, DFS begins at the starting node and proceeds by visiting a connected node, then another connected node from there, continuing down a single path until it either reaches the goal or a dead end. Only then does it backtrack to previous nodes to explore other unexplored paths. This approach is efficient in terms of memory, as it typically stores only the nodes along the current path in memory. However, DFS does not guarantee finding the most optimal or shortest path to the goal, especially in wide graphs with many branches, since it may miss closer solutions in favor of going deep. Additionally, DFS can become trapped in deep or even infinite paths without careful handling. This algorithm is also blind to edge cost and best suited for unweighted or exploratory searches. In implementations, DFS often uses a stack data structure, either explicitly or implicitly through recursion, to manage the nodes being visited.

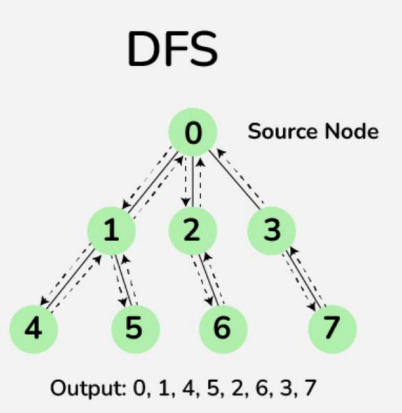

Figure 4. Diagram of DFS searching until reaches dead end. Source: link

c. Uniform Cost Search (UCS)

UCS is a pathfinding algorithm that expands the node with the lowest cumulative cost from the start, rather than by depth or arbitrary order. Unlike BFS or DFS, UCS takes into account the actual cost it takes to reach a node, denoted as g(n), where *n* is a particular node. This means that UCS prioritizes paths that are cheaper, not necessarily shorter in steps. At each step, the algorithm selects the node with the smallest *g(n)* value from a priority queue and continues expanding until the goal is reached.

When the goal is selected for expansion, UCS guarantees that the path found is the one with the lowest total cost, making it optimal for graphs with varying edge weights or costs like the ones in weighted graphs. However, its main weakness lies in performance. It may still explore a large portion of the graph, especially when many low-cost nodes must be processed before reaching the goal. Due to its similarities with BFS, in the case of an unweighted graph, UCS is the exact same as BFS.

2. Informed Search

Informed search, also called heuristic search, are a group of pathfinding algorithms that find a path to the destination by relying on a heuristic. The heuristic can be a form of a way to find the goal by various characteristics of the graph that "seems" to lead to the goal most efficiently. Thus, using heuristics, algorithms in this category utilizes h(n) to estimate the end. Two common algorithms using h(n) are Greedy Best First Search (GBFS) and A* algorithm

a. Greedy Best First Search (GBFS)

GBFS is a pathfinding algorithm that focuses solely on moving toward the goal as quickly as possible by using a heuristic estimate. It evaluates nodes based on the function f(n) = h(n), where *h(n)* is a heuristic that estimates the cost from node *n* to the goal. This means GBFS does not consider how far it has already traveled (no g(n) is used); instead, it chooses the next node that appears to be closest to the goal based on the heuristic. As a result, it can be very fast and often reaches the goal quickly, especially in open or sparse graphs.

However, this speed comes with a trade-off. GBFS is not guaranteed to find the optimal path, and it may make poor decisions in environments with misleading heuristics or many obstacles. The algorithm relies on a priority queue, selecting the next node with the lowest *h(n)* value. Its greedy nature makes it suitable for scenarios where speed is more important than accuracy, but caution must be taken with the heuristic design to avoid inefficient or incorrect routes.

b. A* Algorithm

A*, or A Star, is an algorithm that utilizes both g(n) and h(n) to form its cost function f(n). Effectively, this leads to f(n) = g(n) + h(n). Using both functions allows the algorithm to account for how far it has already traveled (g(n)) and estimate how much farther it needs to go (h(n)) to reach the goal. This combination enables A* to form paths that are not only goal-oriented but also cost-efficient. By balancing actual cost and heuristic estimation, A* avoids paths that stray too far (high g(n)) and avoids being overly optimistic about shortcuts (low h(n)).

As a result, A* is one of the few algorithms that can guarantee the optimal path, as long as the heuristic used is **admissible**. This makes it especially powerful and reliable in domains where both accuracy and efficiency are critical. A* typically uses a priority queue to expand nodes in

order of their f(n) value, always choosing the one with the lowest total estimated cost.

3. Heuristic Function (h(n))

Heuristic function, denoted as h(n), provides an estimate of the cost from a given node n to the goal. Unlike actual path costs (which are known from traversed paths), the heuristic is an informed guess used to guide the search more efficiently toward the target. Heuristics are particularly important in algorithms like A* and Greedy Best First Search (GBFS), where they influence which paths are prioritized during exploration.

The effectiveness of a heuristic greatly affects both the efficiency and accuracy of the search. A well-designed heuristic can significantly reduce the number of nodes expanded by focusing the search in more promising directions, whereas a poor heuristic may lead to inefficient or even incorrect results.

a. Admissible Heuristic

A heuristic is said to be admissible if it never overestimates the true cost to reach the goal. That is, for all nodes n: $h(n) \leq h^*(n)$. Where, h(n) is the estimated cost from node n to the goal and h*(n) is the actual minimum cost to reach the goal from n [2].

An admissible heuristic ensures that A* will always find the optimal path, because it guarantees the search will not bypass a better route in favour of a misleading shortcut. This property is critical when accuracy and path cost minimization are required.

C. L-Systems

L-systems were defined by Lindenmayer in 1968 as an attempt to describe the development of multi-cellular organisms, the L standing for Lindenmayer. L-systems provide a framework within which these aspects of development can be expressed in a formal manner [3].

In essence, l-systems are a set of rules and an axiom that allows the production of shapes. In traditional l-systems, the rules can be iterated multiple times to produce more complex shapes. L-systems do not have a set rule saying a character represent a specific movement, but it is up to the pattern-maker to figure out characters to put in their l-system and their meaning to achieve the shapes that they desire.

As an example of an l-system is the fractal tree. Consisting of the axiom: 0, and the rules: (1 → 11) and (0 → 1[0]0). Where 0 means to draw a short line, 1 means to draw a long line, "[" is to push the current position and angle into a stack while also turning 45° to the left, lastly "]" popping a position from the stack and turning 45° to the right. Below in figure 5, is the first few iterations of this l-system.

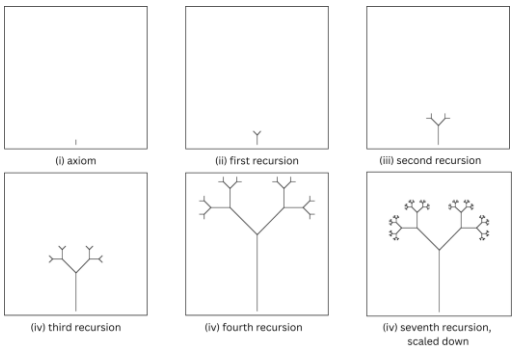

Figure 5. First few iterations of fractal tree l-system.
Source: link

This system's capability to produce intricate shapes and geometry has high potential in practical uses, particularly in graph mapping usually takes a lot of computing power and storage to house. Using l-system, it is possible to compress huge graphs to become just a set of rules and an axiom, allowing for compression and also allow further generation using the same rules iteratively beyond the default scope.

III. Problem Analysis

Urban street networks in real life are unorganized and most have no pattern and is hard to generalize. To effectively model these complex networks using l-system, there are a few limitations and definitions to this paper and experiment.

A. Urban City Networks

The data for modelling the real street networks will use the format developed and provided by OpenStreetMap (OSMNx) [4]. Their formatting models real geographical streets as nodes and edges. One interesting fact about the edge they use is that in itself are edges, this is to simulate curves within the edge other than the two nodes that is at the end of the lines. To simplify and lower computation cost, the main program will only use nodes from the maps provided by OSMNx. The edges of these maps will not be used and will only use nodes with edges being synthetic ones that connect one node to another based on the edges provided by the original map, but are straighten to simply. One weakness of this is that granularly, the curves in the original street network is not preserved, but the general shape of the city will be roughly preserved and is good enough for this research.

B. Pathfinding Algorithm

The pathfinding algorithm used for this will be uninformed searches. The reason being that there is no set goal for this exploration of the street graphs, rather the aim is to explore the entire map and find the characteristics of the network. To simplify, BFS and DFS will be used for their blind searching capabilities not using any cost function, which fits the specifications for the task. Both those algorithms will be using visited node memory as a way to avoid loops and multiple visits of a single node.

C. L-System

In order to build these networks using l-system, there needs to be several modifications to the traditional l-system for its use in this paper. The basics of l-system that will be present are the basic axiom and set of rules to present the network. The modifications come by adding the position (cartesian coordinates) of the original map into the l-system. In addition to that, the rounded distance and angle of the edges are also to be recorded. These steps are necessary to preserve the shape of the original map. This modified l-system to present the original map will be called **numerical l-system** in this paper

However, there will also be a **reference l-system** that is an l-system produced by simplifying the numerical l-system and plotted in a way that is similar to traditional system using fixed angles and distance. This l-system is utilized to model the trend of the street networks.

IV. Implementation

The implementation will use a main program to determine the flow of future users of the program, should there be further interest in this research. But below are the implementation steps necessary in order to derive the l-system of the street networks.

A. Program File Structure

```
.
├── analysis
│   ├── __init__.py
│   ├── lazy_pattern_extractor.py
│   └── lsystem_maker.py
├── classes
│   ├── branch_node.py
│   └── __init__.py
├── main.py
├── playground.py
├── testing
│   ├── data_structure_analysis.py
│   └── node_analysis.py
└── util
    ├── download_graphml.py
    ├── draw_lsystem.py
    ├── __init__.py
    ├── pattern_plot.py
    └── save_load.py
```

Figure 6. File Structure of the min program

Most of the implementation is in analysis directory. The script lazy_pattern_extractor.py contains the implementation of BFS and DFS algorithms to extract patterns from graphml raw data. While lsystem_maker.py is the script that generates numeric and reference l-system.

B. Data Structure

In order to build the l-systems, an analysis on the city network is needed. This is to find general patterns while also preserving the shape of the network itself. To model this, there are two classes used.

```python
class Branch:
    def __init__(self, target_id, angle, distance):
        self.target_id = target_id
        self.angle = angle
        self.distance = distance
```

Figure 7. Branch class definition

```python
class LocationNode:
    def __init__(self, id, position):
        self.id = id
        self.position = position
        self.branches = []
```

Figure 8. LocationNode class definition

The main class used is the LocationNode. It represents a single node in the original map with some modifications. The only attribute that is preserved and needed is the id and position. There is also a list of branches that is present in the LocationNode, the list consists of Branch objects.

Branch class is the synthetic edges between each node, but to facilitate the l-system drawing needs, it is presented as a member of LocationNode and has the attribute angle and distance to other nodes. Angle in particular refers to the angle with the positive x-axis as the reference point. Positive angles in this context is the turn degree to the left, while negative angles is the turn degree to the right. In addition to that, to avoid calculation errors because proximations, the target node id is also stored in the Branch class to simplify and ensure an edge is between two valid nodes.

C. Deriving Patterns from City Networks using Pathfinding Algorithm

To build the patterns and find characteristic of the networks, BFS and DFS algorithm will be used. The pattern will take form in the two classes presented in part A, which are LocationNode and Branch class.

1. BFS Algorithm

```
Function BFS_Extract_Patterns(Graph G,
Dictionary positions):
  Initialize visited as empty set
  Initialize patterns as empty list
  node_to_edges ← Preprocess_Edges(G)

  For each node start in G.nodes:
    If start is in visited or start not in
node_to_edges:
      Continue to next node

    Initialize queue with start node
    Add start to visited set

    While queue is not empty:
      s ← Dequeue from queue
      pos_s ← positions[s]
      solution ← new LocationNode(s, pos_s)

      For each edge (u, v, key, data)
connected to s in node_to_edges[s]:
        other ← v if u == s else u

        If other not in positions:
          Continue to next edge

        pos_other ← positions[other]
        dist ← EuclideanDistance(pos_s,
pos_other)
        angle ← ComputeAngle(pos_s,
pos_other)
        branch ← new Branch(other, angle,
dist)

        If branch is not already in
solution.branches:
          Append branch to solution.branches
```

```
        If other not in visited:
          Add other to visited
          Enqueue other to queue

    Append solution to patterns

  Return Remap_Node_IDs(patterns)
```

Figure 9. Pseudocode for BFS algorithm to extract patterns.

2. DFS Algorithm

```
Function DFS_Extract_Patterns(Graph G,
Dictionary positions):
  Initialize visited as empty set
  Initialize patterns as empty list
  node_to_edges ← Preprocess_Edges(G)

  Define Recursive Function DFS(s):
    pos_s ← positions[s]
    solution ← new LocationNode(s, pos_s)

    For each edge (u, v, key, data) in
node_to_edges[s]:
      other ← v if u == s else u

      If other not in positions:
        Continue to next edge

      pos_other ← positions[other]
      dist ← EuclideanDistance(pos_s,
pos_other)
      angle ← ComputeAngle(pos_s, pos_other)
      branch ← new Branch(other, angle, dist)

      If branch not in solution.branches:
         Append branch to solution.branches

      If other not in visited:
         Add other to visited
         Call DFS(other)

    Append solution to patterns

  For each node start in G.nodes:
     If start not in visited AND start in
node_to_edges:
    Add start to visited
    Call DFS(start)

  Return Remap_Node_IDs(patterns)
```

Figure 10. Pseudocode for DFS algorithm to extract patterns.

D. Deriving Numeric L-System

Generating the numeric l-system first is by converting the patterns produced before into strings of rules. This is done by encoding certain steps with certain characters. The encoding is listed in figure 11.

No	Character	Meaning
1	F<n>	Go forward as far as n distance unit
2	+<n>	Turn left as far as n degree unit
3	-<n>	Turn right as far as n degree unit
4	[	Push current position and angle to stack
5	]	Pop position and angle from stack
6	X<n>	Do rule n

Figure 11. L-system encoding and meaning.

To derive the numeric l-system, we will follow a simple procedure. For each LocationNode, we will make a single rule. The rule will iterate through the branches in the LocationNode. For each branch, first record the angle using the symbol "+" or "-", then record the distance with "F", lastly if the rule of node at the other branch is not empty, then add "[Xid]" to the rule. Repeat the sequence before until all branches are recorded.

The next procedure is to clean up the raw branches that are produced. First is to delete all rules that are empty, these are produced because of nodes that do not have branches on them alongside deleting any reference toward them in the rules. Another clean up to do is by reindexing the id that the rules use. Before it was using the id from the LocationNode, but from the simplification and to make the id more meaningful, the id on the rules are reindexed from 1 to n with no gaps, n being the number of rules produced.

Lastly for the axiom of the system itself, it will consists of all rules that was generated alongside their position from the patterns first produced by the raw data.

E. Deriving Reference L-System

For reference l-system, it will be derived using the filtered numeric l-system. The biggest difference is that the numbers after the characters "F", "+", and "-" will be deleted for generalization. Then after another round of generalization, the rules be reindexed again as to not have two identical rules. id is also stored in the Branch class to simplify and ensure an edge is between two valid nodes. This remapping is done by noting the new generalized forms and assign new ids to the new shapes.

V. Testing and Analysis

The main program has several options, as seen in figure 12. With the main creation being in option 1-3. Option 1 is to download the dataset from OpenStreetMap[4] database, option 2 is to generate the patterns or LocationNodes from the downloaded dataset, while option 3 is to generate the l-sytem from the pattern that is produced before. Below are screenshots of the main program.

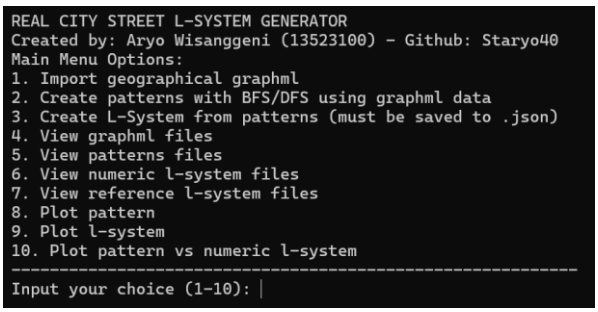

Figure 12. Main program options

```
DOWNLOADING REAL GEOGRAPHICAL STREETS
Please enter a location in a format like:
   - 'Bandung, Indonesia'
   - 'New York City, New York, USA'
   - 'Tokyo, Japan'
ⓘ The more specific the location, the more accurate the result.

Input location: Medan, Indonesia
This might take a while (a few minutes)
Downloading...
Download successful
Press anything to go back
```

Figure 13. Option 1: downloading "Medan, Indonesia" from OSMNx database, documentation

```
CREATE PATTERN WITH BFS/DFS
Current graphml files:
 - bandung.graphml
 - jakarta.graphml
 - Medan.graphml
 - new-york.graphml
------------------------------------------------------------
Enter the name of file to process pattern (exact): bandung.graphml
```

```
USE BFS OR DFS?
1. BFS
2. DFS
3. Exit
Algo: 1
```

Figure 14. Option 2: creating patterns from raw OSMNx data

```
PATTERN PROCESSED!
Using BFS took 0.666334867477417 seconds
Pattern - Node: 19424, Branch: 34080
Save pattern as json (y/n)?
```

Figure 15. Option 2: result of creating pattern

```
PROCESS L-SYSTEM WITH EXTRACTED PATTERN
Current pattern files:
 - bandung.json
------------------------------------------------------------
Enter the name of file to process (exact): bandung.json
```

Figure 16. Option 3: input pattern file for processing into l-system

```
L-SYSTEM PROCESSING
Which type of L-System to produce?
1. Save numeric L-System (similar to actual geo shape)
2. Save reference L-System (for seeing trends in traditional L-System)
3. Exit
Input (1-3):
```

Figure 17. Option 3: choosing which l-system to generate

```
Generating L-Systems...
Input filename: bandung_numeric
Successfully saved Numeric L-System
Produced rules: 19424
First 5 rules:
1: -142F22[X2]-43F89[X3]+135F154[X4]+74F155[X5]
2: +38F22[X1]+136F147[X6]
3: +137F89[X1]-48F160[X7]+45F73[X8]
4: -45F154[X1]+44F66[X9]+136F110[X10]-126F26[X6]
5: +44F47[X11]-106F155[X1]
Press anything to go back
```

Figure 18. Option 3: generating numeric l-system

```
Generating L-Systems...
Input filename: bandung_reference
Successfully saved Referential L-System
Produced rules: 6438
Non-reference rules: 19
Reference rules: 6419
First 5 nonreference rules:
1: (-1, '-F')
2: (-1, '+F')
3: (-1, '+F[]-F')
4: (-1, '+F[]+F')
5: (-1, '-F[]+F[]+F')
First 5 reference rules:
21: (2, '+F[X2023]')
22: (3, '-F[X2046]+F')
23: (4, '+F[X23]+F[X2029]-F[X2023]')
24: (5, '+F[X20]-F')
25: (6, '+F[X23]+F')
Press anything to go back
```

Figure 19. Option 3: generating reference l-system

In terms of pathfinding algorithm comparison of BFS and DFS, they both produce the same results, only BFS is faster than DFS for exploring entire graphs. Reason being DFS needs more time to explore new paths, especially when it is deep in one rout, the backtracking to go to another path takes too long, compared to BFS that has no trouble finding the next route to go to thanks to the queue data structure. Result of this comparison can be seen in figure 20.

```
$ python src/playground.py compare_algorithm bandung.graphml
Finished loading graphml data
Finished BFS pattern extraction in 0.5846140384674072
Finished DFS pattern extraction in 0.8918795585632324
Comparison of BFS vs DFS pattern extraction
--------------------------------------------
BFS - Nodes: 19424, Branches: 34080
DFS - Nodes: 19424, Branches: 34080
BFS and DFS results are consistent.
```

Figure 20. Algorithm comparison of BFS and DFS

Options 4-7 are to list the files in each directory to simplify checking for the users. While option 8-10 are to plot patterns and l-systems that are produced. The plotting of some large cities can take a long time, around a few minutes. Below are test results from plotting patterns and produced l-system from Bandung City dataset.

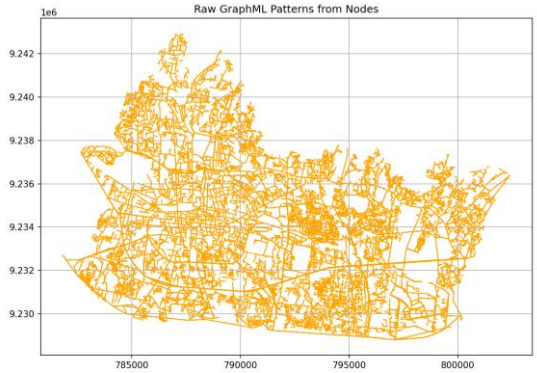

Figure 21. Option 8: generated plot from raw pattern of bandung.json.

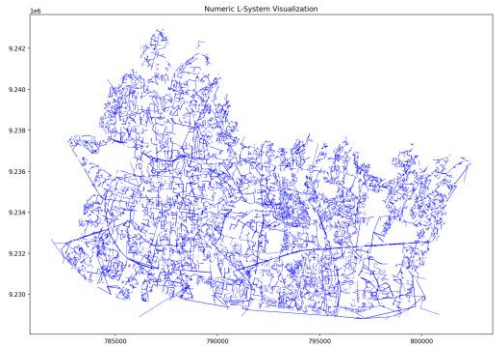

Figure 22. Option 9: generated plot from numerical l-system from bandung.json

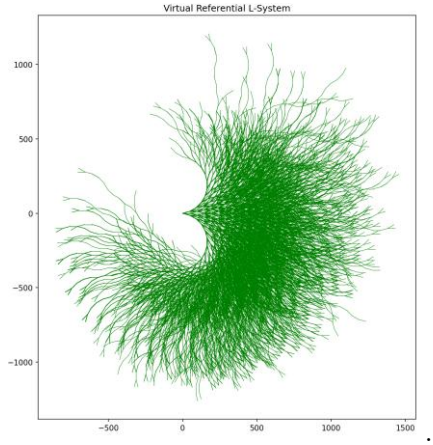

Figure 23. Option 9: generated plot from reference l-system from bandung.json

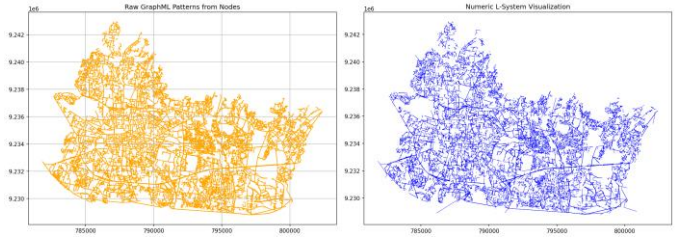

Figure 24. Option 10: side-by-side comparison of plot generated raw pattern and from numeric l-system from bandung.json

In addition to being able to generate the same city and also record the trend of the street network. A lot of compression was done by making the l-system. For comparison, below is the size of the pattern json file and its numeric l-system json file size.

No	Dataset	Pattern size	L-System size	Compress
1	Bandung	5,293 KB	758 KB	85.69%
2	Jakarta	20,038 KB	1,440 KB	92.81%
3	New York	16,096 KB	1,540 KB	90.43%

Figure 24. Compression rate of a few sample cities.

VI. Conclusion

In conclusion, the combination of uninformed pathfinding algorithms and L-systems offers a powerful method for analyzing and compressing complex street networks. By leveraging blind search algorithms such as BFS and DFS, the network graph can be explored to uncover structural patterns.

Once these patterns are identified, they can be encoded into L-system rules. These rules are string-based instructions that capture the basics of the network's layout. Not only does it allow for the reconstruction of the original network through iterative application but also data compression. A network that originally requires thousands of bytes to store can instead be represented by a concise set of L-system rules, which significantly reduces storage requirements and computational overhead during regeneration. Other than that, this L-system-based approach also provides a compact and interpretable representation of the network's tendencies. This insight is particularly valuable when extending existing networks. Planners and designers can use the extracted L-system rules to generate new segments that align with the existing urban pattern, maintaining consistency.

This technique is especially beneficial in domains where procedural generation of complex structures is needed under tight computational constraints, such as in simulation, game development, urban planning, or modeling of biological and natural systems. By capturing the logic behind complex structures in a set of simple, repeatable rules, L-systems enable the generation of rich and believable forms without the need for heavy data storage or computation.

Ultimately, this work demonstrates that through the integration of pathfinding and generative systems like L-systems, it is possible to not only compress but also meaningfully understand and extend complex networks. This dual benefit makes the approach both efficient and insightful, offering practical applications across a wide range of fields.

Project Output Links

Youtube Video Link : link

Github Repository Link : link

Acknowledgment

The completion of this paper could not have been done without the help of IF2211 lecturers, especially Dr. Ir. Rinaldi Munir, M.T. whom has taught class K02 2025 for Algorithm Strategies class and provided a great amount of learning resources for students to study from online. Alongside the lecturers, the author would also like to thank the classmates of K02 for the support and fun times shared during the semester, as without them, surviving and finishing all courses in this semester would not have been possible. Lastly, a special thanks for the authors family for supporting the author mentally and financially throughout the semester. The author hopes this paper to serve as a useful reference, not only for others interested in the relevant field of study but also as a resource for the author's future work.

References

[1] Dr. Ir. Rinaldi Munir, M.T., Route Planning Part 1, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf, accessed at 23 June 2025

[2] Dr. Ir. Rinaldi Munir, M.T., Route Planning Part 2, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf, accessed at 23 June 2025

[3] Lindenmayer Systems (L-System), https://archive.nptel.ac.in/content/storage2/courses/106106049/downloads/Lindenmayer%20Systems%20.pdf, accessed at 23 June 2025

[4] Kevin Curran, University of Ulster, OpenStreetMap, https://www.researchgate.net/publication/262206447_OpenStreetMap, accessed at 23 June 2025

[5] Humera Farooq, et al., An Approach to Derive Parametric L-System Using Genetic Algorithms, https://www.researchgate.net/publication/221365144_An_Approach_to_

Derive_Parametric_L-System_Using_Genetic_Algorithm, accessed at 23 June 2025

[6] Surya Sujarwo, et al., Implementation of L-System in Procedural City Generation using Java, https://research-dashboard.binus.ac.id/uploads/paper/document/publication/Proceeding/ComTech/Vol.%2001%20No.%202%20Desember%202010/08%20-%20Surya%20Surjawo%20.pdf, accessed at 23 June 2025

STATEMENT

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 1 Juni 2025

Aryo Wisanggeni 13523100