Supplementary Information - In Pursuit of the Optimal City: Evaluation of Procedurally Generated City Layouts

Cameron Dobbie[1] and Anders Johansson[1*]

[1*]School of Engineering Mathematics and Technology, University of Bristol, Queens Road, Bristol, BS8 1QU, United Kingdom.

*Corresponding author(s). E-mail(s): a.johansson@bristol.ac.uk;

List of Tables

List of Figures

1 CitySprout Methodology

This section provides further detail on how CitySprout develops a city.

The full source code for CitySprout can be accessed in our open-source repository [1].

The `generateCity` function within the `CityGenerator` class is used to create the cities. The full list of inputs and output of this function are given in Table 1.

The generated cities, at all stages of their development, are stored as a `NetworkX` graph object [2]. This object type has been chosen because the `NetworkX` package allows for custom attribute types to be assigned to nodes. `NetworkX` also contains many graphical analysis algorithms which can be used to calculate metrics as discussed in the main paper.

`NetworkX` graphs are stored as a collection of nodes, and edges between them. In the case of CitySprout, each node and edge has a series of stored parameters which influence how the city expands. These parameters are described in Tables 2 and 3. Note that we will subsequently refer to the `roadType` and `nodeType` combination of a node as its "RN value".

Before a city is generated, a single "grammar" is selected. Each of these grammars has a series of production rules, governing how (and if) nodes with different RN values will develop with each iteration.

Within the definition of any individual grammar, each possible RN value can have any number of associated production rules. These rules specify what will happen when the L-system is applied to an existing node i. These rules can change i to a different node type, and can create any number of new "children" nodes j, making i the "parent" of these nodes. The position of these children is also determined by the rule, which specifies both the angles of deflection from the `incEdge` of i, and also the distance between i and each child, j. The angles can be either a single value, or a list of two limits, between which a random value will be selected. If a child is created, an edge will also be created from the parent to the child. Figure 1 illustrates how these angles and distances are calculated.

If any RN value has 0 associated rules, any node i with this RN value will remain in the same state for all subsequent iterations, and no children nodes j will be created from it. If it has one production rule, this will be automatically applied to i during an iteration. If it has multiple rules, each rule must have an associated `occurProb` which governs how likely it is to be selected during an iteration. Each rule may also have a `minDistances` condition, which prevents a production rule from being selected if i is too close to any other node of a specific RN value. The handling of this will be discussed subsequently. This condition allows the network to follow a desired spatial distribution and avoid overcrowding. The full list of parameters for each production rule is shown in Table 4.

In order that the graphs can later be analysed as road networks, the edges are grouped into roads. To achieve this, each edge is assigned a "road number", which is a unique identifier for each road. Each road consists of any number of connected edges. New edges are either added to existing roads or placed on a brand new road, according to the `newRoad` parameter of each production rule shown in Table 4.

Table 1 The inputs and outputs for the `generateCity` function.

Parameter	**Possible values**	**Meaning**
Inputs		
`grammar`	`grammars.Organic`, `grammars.Grid`, `grammars.Hex`, `grammars.Line` (or other `dict` of the correct form)	The grammar which will be used to generate the city
`seed`	Any `int` (default 42)	The seed which will be used
`intersectRadius`	Any `float` (default 0.8)	The minimum possible distance between nodes (any nodes closer than this will be joined according to check (2)
`maxWidth`	Any `float`, `None` (default `None`)	The maximum width of the city
`maxHeight`	Any `float`, `None` (default `None`)	The maximum height of the city
`plotType`	`"Map"`, `"Animation"`, `None` (default "Map")	The plot type which will be shown
`showNodes`	Any `bool` (default `False`)	Whether or not a circle will be displayed at each node
`nodeLabelType`	`"Node Type"`, `"Node Number"`, `"Road Number"`, `None` (default `None`)	The label type displayed at each node
`edgeLabelType`	`"Edge Weight"`, `"Road Number"`, `None` (default `None`)	The label type displayed at each edge
`complexityPath`	Any `str` representing a path, `None` (default `None`)	The path at which the complexity data will be stored
`maxRoadSegments`	Any `float` or `int`	The maximum number of edges in the graph before generation will stop
`maxRoadLength`	Any `float` or `int`	The maximum length of road before generation will stop (hm)
`population`	Any `float` or `int`	The maximum population of the city before generation will stop
Output		
`G`	`NetworkX` graph object	The city's road network

Table 2 The parameters stored for node i.

Parameter	**Meaning**
`roadType`	`m` (main), `l` (large), or `s` (small): the type of road node that i lies on
`nodeType`	`start`, `L` (leaf), `T` (twig), or `B` (branch): represents the stage of expansion of a node, i.e. starting node, yet to expand, finished expanding, etc.
`pos`	The coordinates of node i
`incEdge`	The direction vector of the edge between i's parent and i
`incRoadNumber`	The road number of the edge between i's parent and i

Table 3 The parameters stored for edge (i, j).

Parameter	**Meaning**
`weight`	Used only for plots. Unique to each `roadType` and used to plot different sized roads with different line thicknesses
`boundaryBox`	The axis-aligned boundary box of edge (i, j)
`roadNumber`	The road number of edge (i, j)

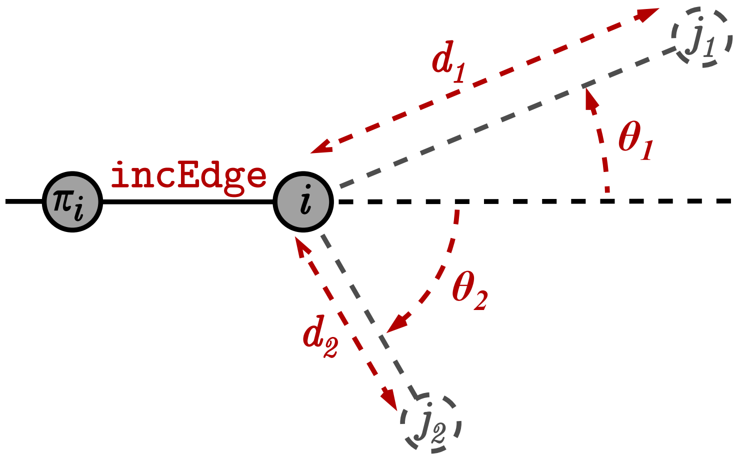

Fig. 1 Diagram showing how the positions of new nodes are calculated. The `incEdge` shown is that of node i, and is the line between i and its parent, π_i. j_1 is the first new node, and d_1 and θ_1 are the corresponding distance and deflection angle respectively. Similarly for the second new node, j_2. Note that in this case, θ_2 will take a negative value, as the deflection angle is always measured in the anticlockwise direction.

Table 4 Parameters defining each production rule.

Parameter	**Data type**	**Meaning**
`occurProb`	`float`	The relative probability of this rule being chosen. Used when there are multiple possible production rules for the same *RN* value.
`changeNodeTo`	`str`	Node type to change node to after expansion. This allows nodes to expand only once, rather than every iteration. Possible values shown in Table 2.
`thetas`[1]	`float`	Angle of deflection of new edge from expanding node's `incEdge`. See Figure 1.
`randDirection`[1]	`bool`	`True` means the angle of deflection has a 50% chance to change sign.
`lengths`[1]	`float`	The distance between the expanding node and the child node(s) (before any checks are performed). See Figure 1.
`newRoadTypes`[1]	`str`	The road type of the child node(s). Possible values shown in Table 2.
`newNodeTypes`[1]	`str`	The node type of the child node(s). Possible values shown in Table 2.
`newRoad`[1]	`bool`	`True` means the new edge(s) will lie on a brand new road. `False` means the new edge(s) will lie on the same road as the expanding node's incoming edge.
`minDistances`[1]	`dict`	A dictionary of *RN* values and the minimum allowed distance between the expanding node and a node of the corresponding *RN* value. If there is a node of the given *RN* value too close to the expanding node, another rule will be selected (if possible).

[1]These parameters can also be a list of values, with all of these lists being the same length, corresponding to the number of nodes created. The first value in each list is for the first child node, and so on. This allows for nodes to expand into multiple nodes rather than just one.

When the generateCity function is called, a single node with a roadType of m and a nodeType of start is created at position $(0, 0)$. The following algorithm is then applied to this node, i:

1. Check the RN value of node i
2. Check the production rules for this RN value. If no production rule exists, leave the node unchanged and skip the remaining steps. If one production rule exists, select it. If multiple production rules exist, randomly select one based on their occurProbs.
3. Check if the minDistances condition for this production rule is satisfied. If not, randomly select another production rule for this RN value (if possible). Continue searching for a rule until the minDistances condition of the selected rule is satisfied
4. Change node type of i according to the production rule
5. Check if production rule requires generation of child nodes j
6. For potential positions of any child nodes j, perform checks (1) and (2)
7. Place any new nodes and/or edges, and calculate and store their parameter values

Note that it is important that for each RN value, there is at least one rule that has no minDistances condition. Otherwise, during step 3, it is possible that no rule will be found where this condition is satisfied. In this case, a random rule will be selected regardless of its minDistances condition. This may lead to unexpected behaviour.

The initial iteration will create a certain number of new nodes and edges.

Another iteration is then applied to this updated graph. Each iteration involves applying the above algorithm to every node in the current graph.

The network expansion will continue until one of the stopping conditions is met. These conditions are input when the generateCity function is called, and can be any combination of: a maximum number of iterations, a maximum number of edges, a maximum total road length, and a maximum population. If multiple stopping conditions are selected, the generation will stop as soon as any of these are met.

Checks (1) and (2), described in step 6 of the algorithm, ensure the graph is cyclic and that the desired spatial distribution of nodes is achieved. They work as follows:

Check (1) ensures that the potential new edge (i, j) being created between existing node i and the potential new node j does not cross over an existing edge. If it does, the position of j will be updated to this intersection point.

If (i, j) crosses over several existing edges, the new node j is placed at whichever of these intersections is closest to the parent node i. The workings of Check (1) are shown in Figure 2.

This check is performed by first looking at every existing edge, and creating a subset of these in which each edge has an axis-aligned bounding box that intersects with the bounding box of the potential new edge (i, j). Any edge which is not in this subset cannot possibly intersect with (i, j), as shown by the proof in Section 2. For each of these potential edges in the subset, CitySprout calculates the intersection point of this edge and the infinite line which is the extension of (i, j). If this intersection point falls on line segment (i, j), the location of this intersection point is recorded. If the intersection point does not lie on (i, j), it is disregarded. Once all potential edges in the subset have been checked, the intersection closest to i is determined by

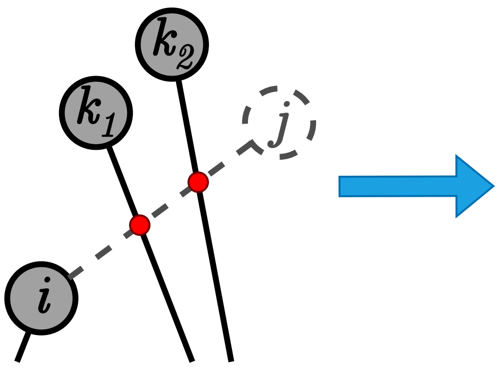

Fig. 2 Diagram showing how check (1) turns a "bridge" into an intersection, to preserve the planar element of the city and also create a cycle. Edge (i, j) intersects with two existing edges, as shown by the red dots. The position of potential node j is therefore adjusted to the intersection which is closest to its parent i.

calculating the Euclidean distance between i and each intersection using Pythagoras' theorem. This closest intersection point becomes the new location of node j.

Without this check the network would have many instances of roads crossing over one another ("bridges"), rather than intersections, meaning the network would not be planar. The joining of nodes onto existing edges creates cycles within the graph, making it into an inter-connected network rather than an acyclic tree.

Check (2) measures whether a potential new node j is too close to any existing node k, with this minimum allowed distance specified by an `intersectRadius` parameter which can be selected when the generator is called. The default value of this radius is 0.8 arbitrary units. If the new node j is too close to any existing node k, rather than this new node j being created, an edge will simply be made from i, j's parent, to k. The workings of Check (2) are shown in Figure 3.

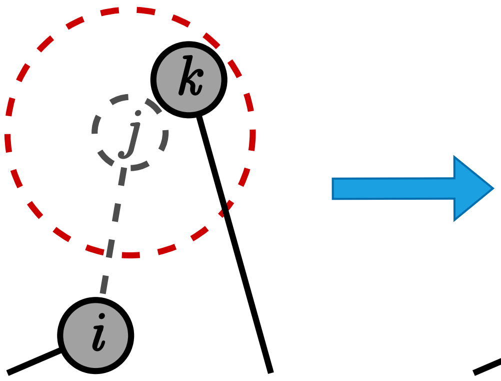

Fig. 3 Diagram showing how check (2) prevents node overcrowding. When an existing node k is within the `intersectRadius` of a potential new node j, j is not created and an edge is instead created from i to k. This `intersectRadius` is represented by the red circle.

This check is performed by first checking the distance between new node j and every single existing node. If one of these distances is less than the `intersectRadius`, the new node creation will be cancelled and instead an edge will be created between parent i and this node. Note that as the network becomes large, the sheer quantity of these distance calculations makes performing this check slow.

Without this check, the city would become overly dense with nodes and edges, with them being placed far closer together than is realistic. This check also boosts the inter-connectivity of the graph by creating cycles, in a similar way to Check (1).

It is important that Check (2) is still applied even if Check (1) has already moved the position of the new node. Otherwise, a node may be placed on an existing edge as a result of Check (1), but this new position may be too close to an existing node. The use of Check (2) subsequently to Check (1) would join the new node on to this existing node.

2 Proof that if two line segments intersect, the axis-aligned bounding boxes containing the entirety of these two lines must also intersect

Let AB and CD be two line segments in the xy plane, and bb_{AB} and bb_{CD} be the corresponding axis-aligned bounding boxes of these lines.

Assuming AB and CD intersect, there must be a point P that lies on both AB and CD.

By definition, bb_{AB} contains all of the points on the line segment AB, and similarly bb_{CD} contains all of the points on the line segment CD.

Since P lies on AB, it must fall inside bb_{AB}. Since P lies on CD, it must also fall inside bb_{CD}.

The only way P can lie in both bb_{AB} and bb_{CD} simultaneously is if bb_{AB} and bb_{CD} have an area of overlap, or an intersection.

Therefore, if two line segments intersect, their bounding boxes must also intersect.

QED

References

[1] Dobbie, C.: CitySprout. GitHub. https://github.com/camdobbie/CitySprout/ (2024)

[2] Hagberg, A.A., Schult, D.A., Swart, P.J.: Exploring network structure, dynamics, and function using networkx. In: Proceedings of the 7th Python in Science Conference, pp. 11–15. SciPy, Pasadena (2008). https://doi.org/10.25080/tcwv9851