The Game of Sprouts

Graph theory and simulations

Zinnun Malikov

GHP July 2023

Simulation and path-finding algorithms

Introduction to the Game of Sprouts

About Sprouts:

- Sprouts is a mathematical game that was invented by John Horton Conway and Michael S. Paterson in 1967.
- It is a turn-based game played by two players, where the goal is to strategically draw and connect dots to create new lines.
- The game starts with a certain number of dots, n, and players take turns drawing lines between the dots, following specific rules

Introduction to the Game of Sprouts

Rules:

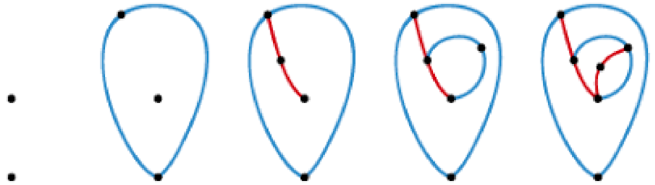
- Each turn consists of drawing a line between two spots (or from a spot to itself) and adding a new spot somewhere along the line.
- The line drawn may be straight or curved but must not self-intersect or cross any other line.
- The new spot cannot be placed on top of one of the endpoints of the new line.
- No spot may have more than three lines attached to it.
- You cannot touch a dot twice with one line then connect it to another.

Introduction to the Game of Sprouts

Terminology

- A **graph** is essentially a collection of vertices and edges that connect vertices.
- A **vertex** is a point on a graph.
- An **edge** is a line which connects two points and is incident to the vertices it touches.
- Two vertices are **adjacent** if there is a single edge connecting them.
- The **degree** of a vertex is the number of edges incident to that vertex.

Example of a Game



Simulating the Game

General Information:

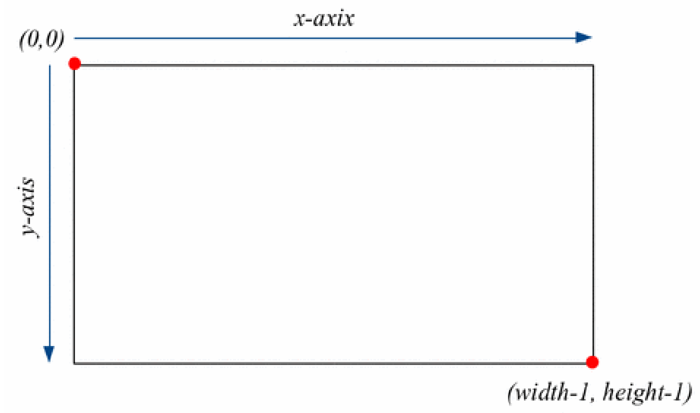
- The simulation of Sprouts runs in a Tkinter canvas using Python, allowing the user to draw curved lines.
- The main loop of the code consists of various functions to check if the rules of Sprouts are met.
- For example, a play only starts if the mouse is clicked near a point on the canvas with degree less than three.
- Moreover, a line is successfully stored only if it does not intersect any other lines and does not result in any degrees greater than 3 once drawn.

Simulating the Game

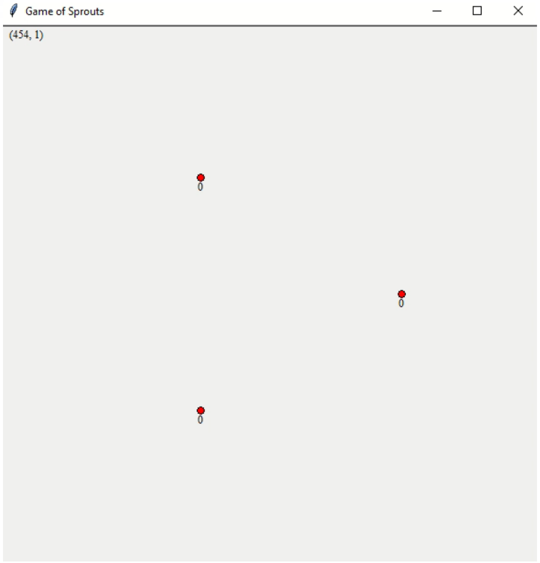
Point Display:

- All figures drawn in the simulation are stored in a coordinate point system.
- The default orientation for the Tkinter is origin $(0, 0)$ at the top left of the canvas and a point (m, n) m pixels right and n pixels down.
- Once a play is initiated, the coordinate position of the user's mouse is appended to a temporary junk storage list.
- If the play is valid, the list of coordinates is added to the main storage list for edges.
- Additionally, all vertices and their respective degrees are stored in separate lists.

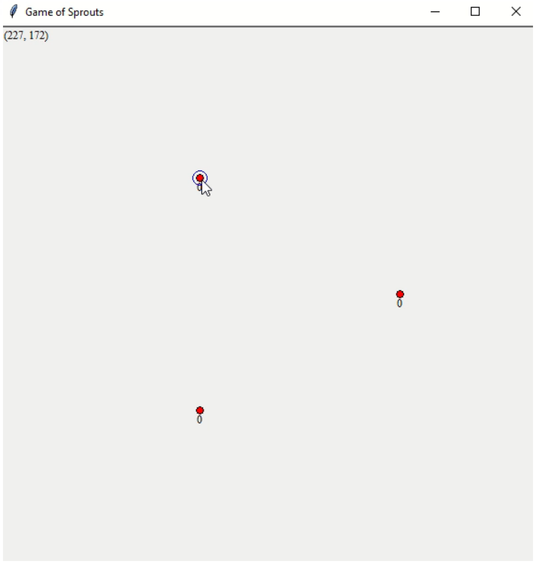
Simulation Coordinate System



Video Demonstration: **Invalid Moves**



Video Demonstration: Complete Game



Functionality

Data from Simulation:

- The simulation allows the user to play Sprouts, buts its most important feature is storing data representing the coordinates of the edges, the positions of the vertices, and the degrees of each vertex.

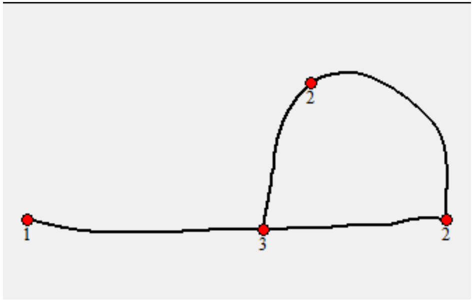
  edge = $[(x_1, y_2), \ldots, (x_i, y_i)]$, i is the number of points constructing an edge

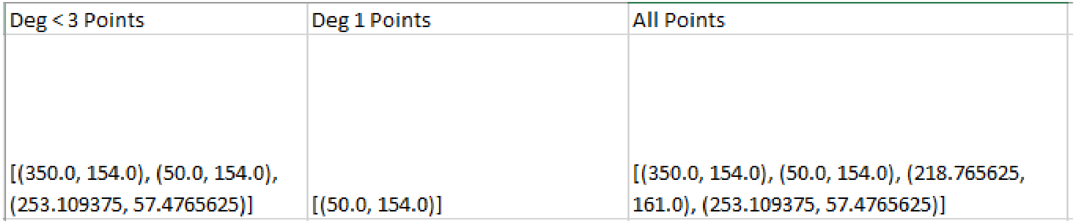
  vertices = $[(a_1, b_1), \ldots, (a_k, b_k)]$, k is the number of vertices

  degrees = $[d_1, \ldots, d_k]$

- These data are stored in a csv file to be used for the path-finding and path-drawing algorithms.

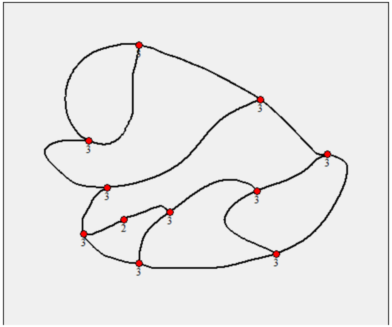
Data Display: Ex 1

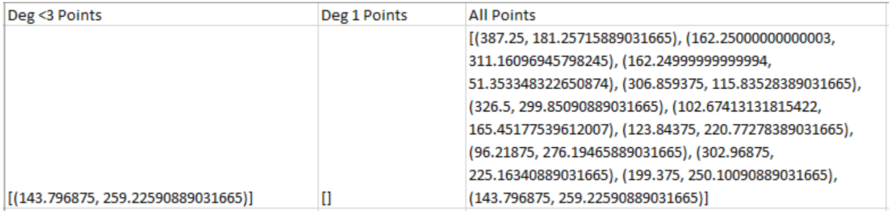




Deg < 3 Points	Deg 1 Points	All Points
[(350.0, 154.0), (50.0, 154.0), (253.109375, 57.4765625)]	[(50.0, 154.0)]	[(350.0, 154.0), (50.0, 154.0), (218.765625, 161.0), (253.109375, 57.4765625)]

Data Display: **Ex 2**



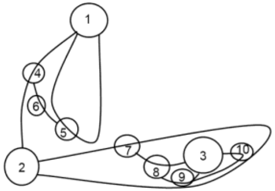


Deg <3 Points	Deg 1 Points	All Points
[(143.796875, 259.22590889031665)]	[]	[(387.25, 181.25715889031665), (162.25000000000003, 311.16096945798245), (162.24999999999994, 51.353348322650874), (306.859375, 115.83528389031665), (326.5, 299.85090889031665), (102.67413131815422, 165.45177539612007), (123.84375, 220.77278389031665), (96.21875, 276.19465889031665), (302.96875, 225.16340889031665), (199.375, 250.10090889031665), (143.796875, 259.22590889031665)]

Path-Existence Algorithms

Existence of Paths:

- Sprouts is said to end when no more valid paths can be drawn between two points.
- It is possible to eyeball solutions to the game, but how can you ensure no paths exist?

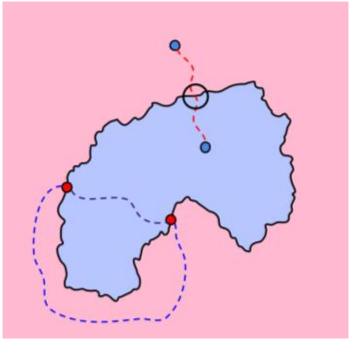


Path-Existence Algorithms

Idea:

- If two points are in the same region, or border the same region(s), then there must be a path entirely within the region(s) between those points that does not intersect any other line and region.

The two red points border the same two regions, so a path exists between these points through each region. However, the two blue points are in different regions, so a line between the two must cross more than one region.



Path-Existence Algorithms

Existence of Paths:

- Identify each distinct closed region G_i, as well as the outside area, in the graph resulting from a position in Sprouts.
- If two points are in the same region, or border the same region(s), then there must be a path entirely within the region(s) between those points that does not intersect any other line and region.
- Different regions can be determined using the flood-fill algorithm.

Path-Existence Algorithms

Flood-fill:

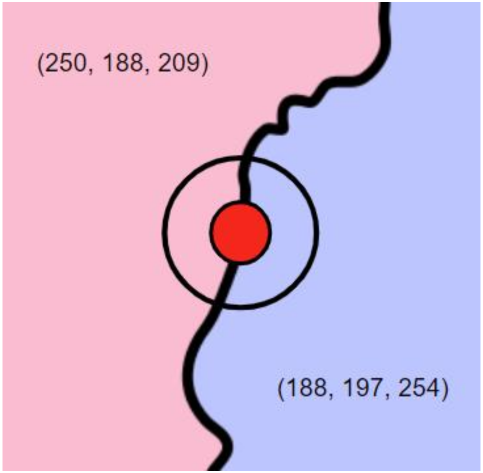
- Each closed region will have one unique color.
- If two vertices border at least one color in common, at least one path exists between these vertices.

Takeaway:

Let A and B be the set of colors two vertices border, respectively. A valid path exists between the vertices if $A \cap B \neq \emptyset$.

Existence of Paths

In the image below, the point borders two colors:



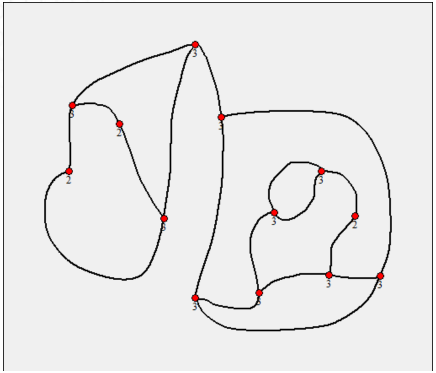
Path-Existence Algorithms

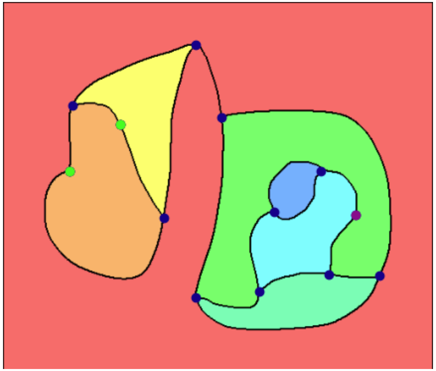
Flood-fill Algorithm

- Pressing CONTROL + I in the Sprouts game automatically saves a screenshot of the current game position, as well as updates the CSV data.
- The pixels of the image are analyzed and filled with different colors based on the flood-fill algorithm.
- In the end, every closed region is filled with a unique RGB color.

Flood-Fill

Image conversion using flood-fill





Path-Existence Algorithms

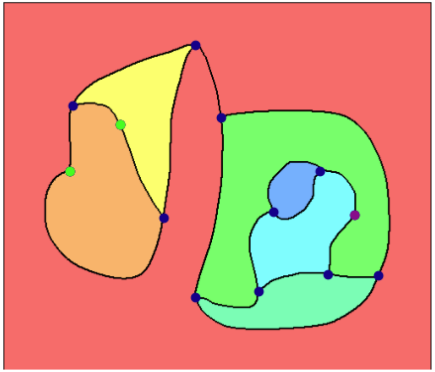
Python Algorithm

- A set of adjacent colors is created corresponding to each vertex with degree 2 or less.
- Combinations of two are created to be used to determine whether the vertices share a region.
- For each combination, the algorithm determines whether a valid path exists or not.

Flood-Fill

Path Existence Using Flood-Fill

```
Path does not exist from: (78.400390625, 200.0) to (418.853515625, 252.609375)
Path exists from: (78.400390625, 200.0) to (138.470703125, 144.140625) with route {(255, 178, 102)}
Path does not exist from: (418.853515625, 252.609375) to (138.470703125, 144.140625)
* * * * * * *
completed
```

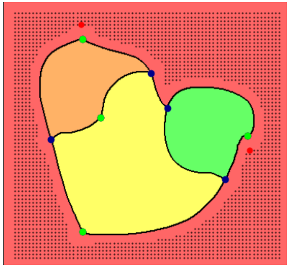
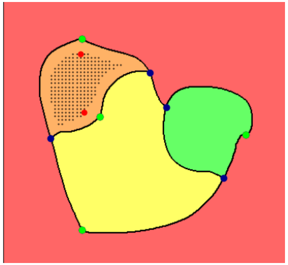
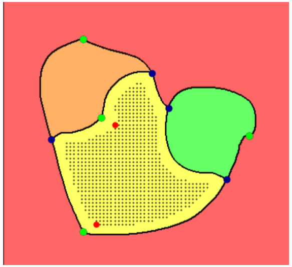


Path-Finding Algorithms

Connecting the Points: **non-closed, simple lines**

- Based on the existence of valid paths between vertices, an algorithm to connect these points can also be constructed.

- First consider the case of lines connecting different vertices. A grid of points can be drawn within the colored regions corresponding to the valid path.

Path-Finding Algorithms

Connecting the Points: **non-closed, simple lines**

- With the grid, the A-star algorithm can be implemented.
- A grid is created consisting of a matrix, where each element is either a 1 or 0.
- A 1 corresponds to a point that can be traversed (i.e. a point within the colored region), and a 0 represents point that cannot be traversed, meaning it is not in the colored region.
- Moves for the path are listed below:
  $(1,0), (-1,0), (0,1), (0,-1), (1,1), (1,-1), (-1,1), (-1,-1)$

Path-Finding Algorithms

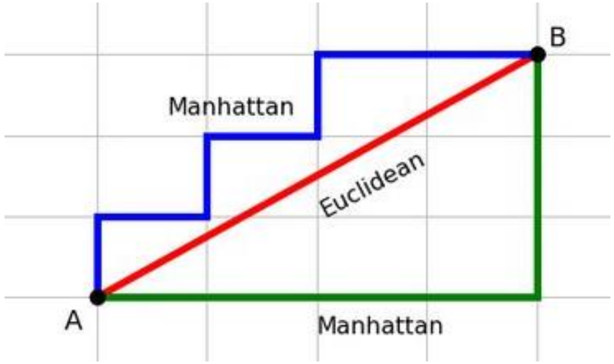
Connecting the Points: **non-closed, simple lines**

- The A* algorithm uses a combination of the actual cost g to reach a node and an estimated cost h from the current node to the goal node. The heuristic function guides the search towards the goal by providing an estimate of the remaining cost.
- The algorithm explores the nodes with the lowest total cost $f = g + h$ first, which tends to prioritize paths that are closer to the goal.
- Afterward, the grid points used in the path are connected and highlighted to form a path.

Path-Finding Algorithms

Connecting the Points: **non-closed, simple lines**

$$h = |x_1 - x_2| + |y_1 - y_2|$$



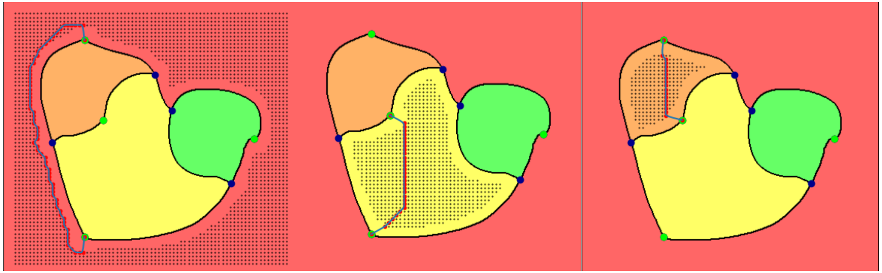
Path-Finding Algorithms

Connecting the Points: non-closed, simple lines

- Python provides an implementation of the priority queue, which is used to store and manage the open nodes during the A* algorithm.
- The priority queue ensures that the node with the lowest f is always selected for expansion.
- In the A* algorithm, the child and parent nodes are tracked using the **parent** dictionary.
- For each node, the **parent** dictionary stores the parent node from which the current node was reached.
- This information is used for reconstructing the path once the goal node is reached.

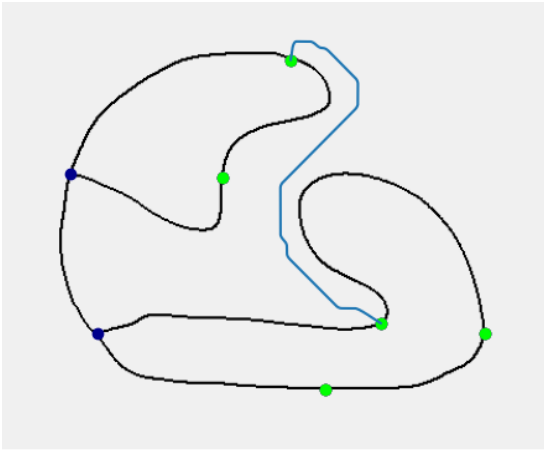
Path-Finding Algorithms

Non-closed, Simple Lines: Examples



Example 1

Basic Path Example:



Path-Finding Algorithms

Connecting the Points: **closed lines**

- Based on the existence of valid paths between vertices, an algorithm to connect these points can also be constructed.
- Next, consider the case of lines connecting the same vertices.

Path-Finding Algorithms

Connecting the Points: **closed lines**

- For the case of loops, where a vertex connects to itself, color analysis is not needed.
- For loop to be valid, the starting vertex must have degree one or zero. Additionally, the loop must not intersect another line, which is where the difficult comes in.
- Define a pre-determined shape for the loop, such as a tear-drop shape consisting of an arc and two lines.

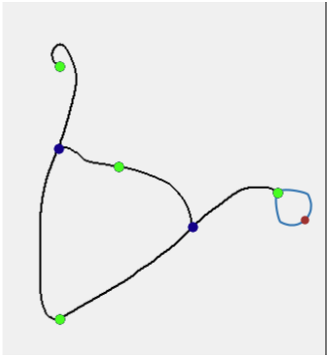
Path-Finding Algorithms

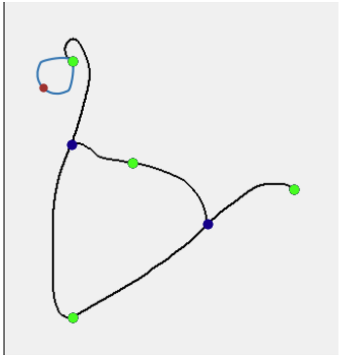
Connecting the Points: **closed lines**

- The algorithm analyzes the pixels of the image within a gradually decreasing radius, starting at 40 pixels and ending at 20 pixels.
- The tear-drop can have four orientations, analogous to the positions of each quadrant of the coordinate system (top right, top left, bottom left, bottom right)
- If there is a black pixel within a given position and radius, a loop is not drawn.
- A loop is drawn once the first successful iteration is met.

Path-Finding Algorithms

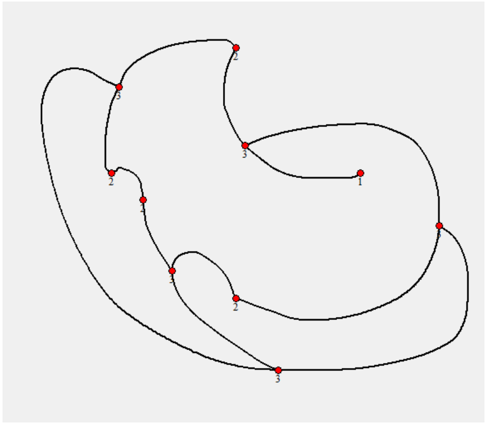
Closed Lines





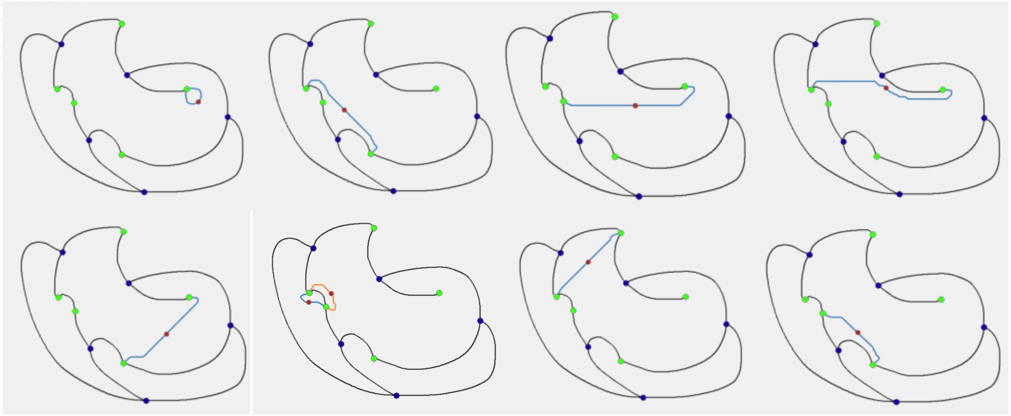
Example

Sprouts Game to Analyze:



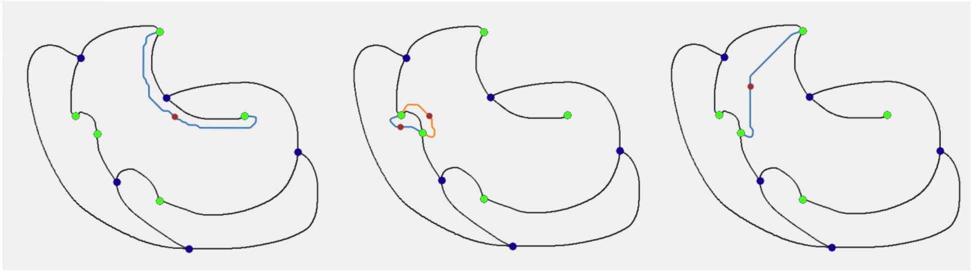
Example

Output:



Example

Output (continued):



Summary

Overview:

- Python simulation of the Game of Sprouts
- Flood-fill algorithm to determine existence of solutions for Sprouts
- Path-finding algorithm to construct valid edges

Next Steps

Expanding on the Project:

- Optimizing and refining algorithms
- Running simulations of the game to gather data about possible strategies