Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

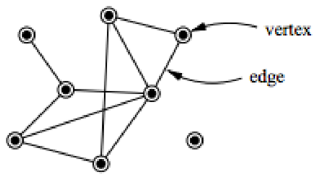
Student Copy
AIEP Send-off Scratch 2 Session 9

GRAPH SEARCH ALGORITHMS & BACKTRACKING

We live in the age of vast networks — in fact, among the most frequented websites are social networking sites. Statista reports that, as of November 2021, Facebook registered 11.74 billion visits every month, Instagram had 3.08 billion, and Twitter had 2.43 billion. These platforms allow us to interact with people virtually.

Another fascinating network is the air transport network, which is used to keep track of the routes of planes all around the globe. In this regard, it is valuable in the aviation industry for monitoring flight traffic, organizing routes in a cost-efficient way, and maximizing mobility across locations.

In computer science, networks are formally referred to as graphs. Note that the graphs which we will be discussing in this handout are different from line graphs, bar graphs, or pie charts. A graph in informatics has two components:



- Vertices: Also called nodes, these represent the "items" in the network. In social networks, the vertices are our user accounts while, in the air transport network, they are the airports
- Edges: These are the "links" that connect the vertices; an edge joins two vertices. In social networks, the edges can be friendships while, in the air transport network, they are the flight routes from one airport to another.

The study of graphs, termed graph theory, is beaming with many curiosities! But foremost among them is searching: figuring out a way from some starting point to arrive at some destination. To this end, there are two categories of graph search algorithms:

- Informed search: We know where the destination (or the goal) is located; thus, we can use this information to speed up our search. An example would be going from our house to our school (provided that we know both their locations).
- Uninformed (blind) search: We do not have any idea as to where the destination is located. An example would be trying to escape a labyrinth. In this handout, we will be focusing on the two core uninformed search algorithms: depth-first search and breadth-first search.

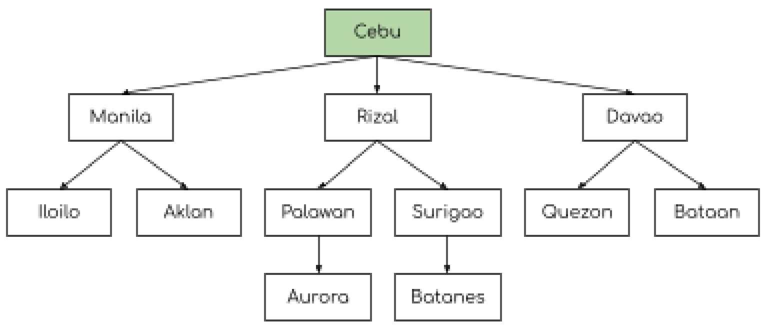
💡 Knowledge is Power

Albeit beyond today's lesson, you may be interested in learning about informed search algorithms, such as the greedy best-first and A* (pronounced "A star") search algorithms.

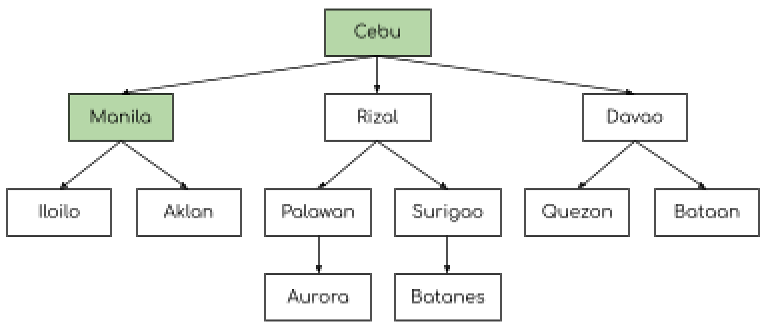
Depth-First Search (DFS) Algorithm

The idea of a depth-first search is to explore a path as deeply as possible before going back to try another path. This is best understood through an illustration[1]:

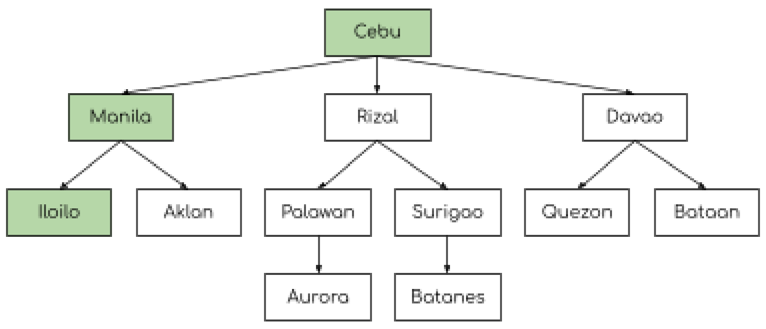
Start: Cebu **Destination**: Batanes



We start in Cebu.

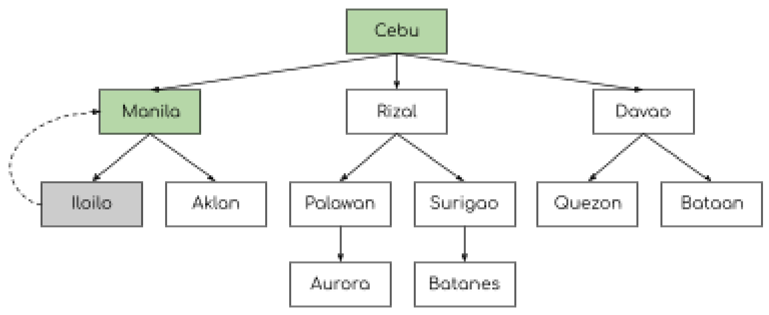


We travel to Manila.



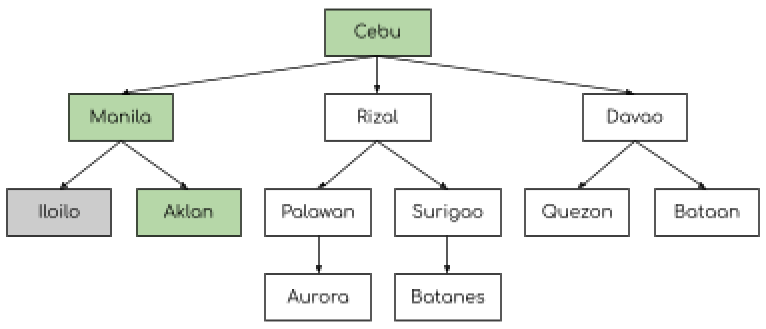
We travel to Iloilo.

We have reached a dead-end[2], signaling that we have explored the path as deeply as possible.



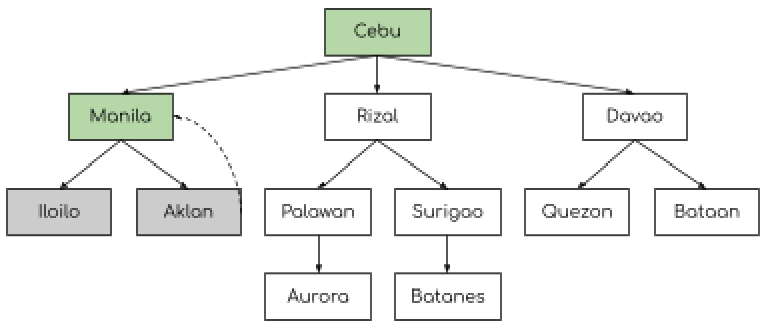
We return to Manila to explore the next possible path.

This act of "going back" is known as backtracking.

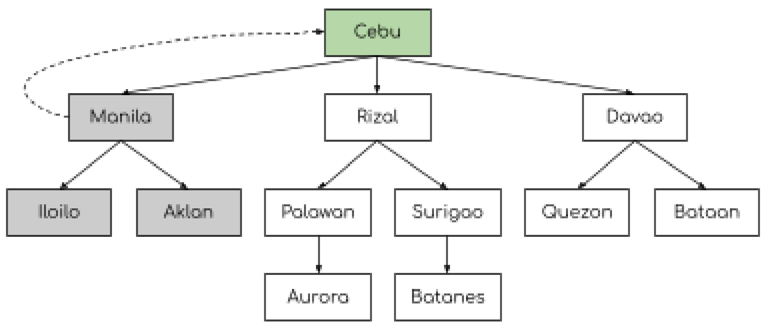


We travel to Aklan, but this is a dead-end anew.

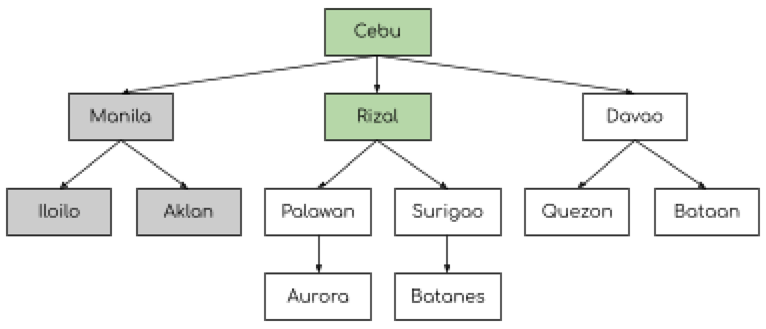
[1] In the interest of pedagogy, we will use a *tree* (a specific type of graph) to illustrate the search algorithms.
[2] In tree data structures, we call these dead-ends *leaf nodes* or simply *leaves*.



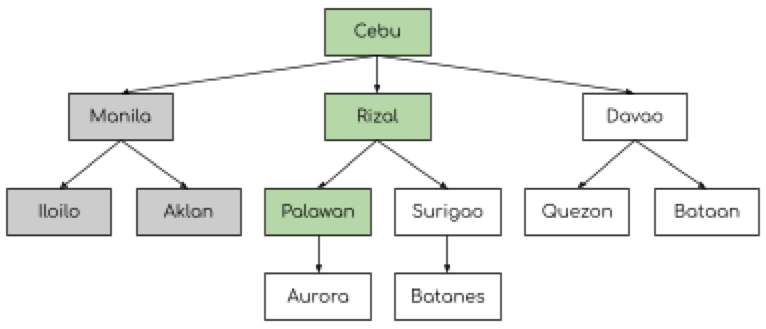
We backtrack to Manila.



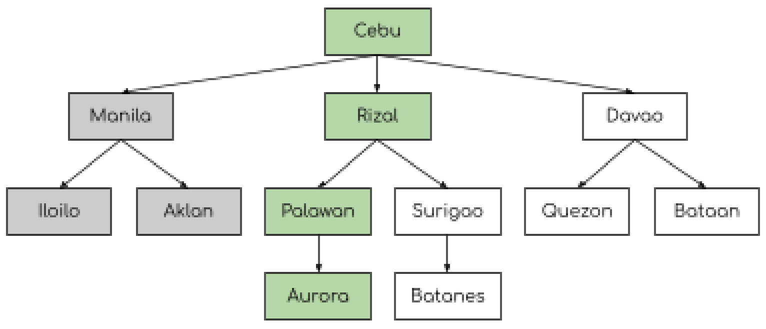
Since all paths from Manila have been explored, we backtrack to Cebu.



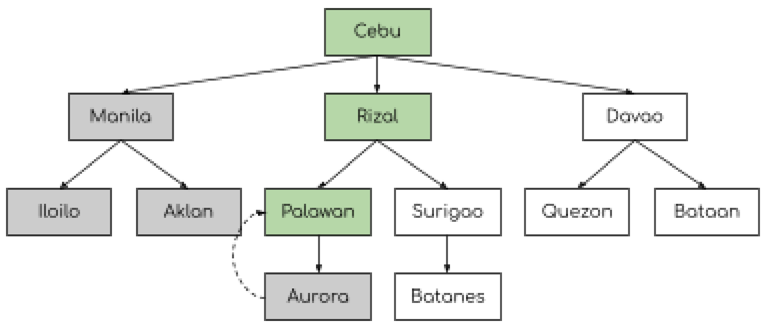
We travel to Rizal.



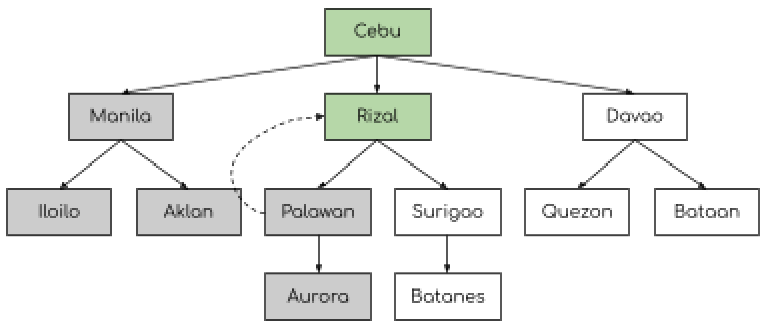
We travel to Palawan.



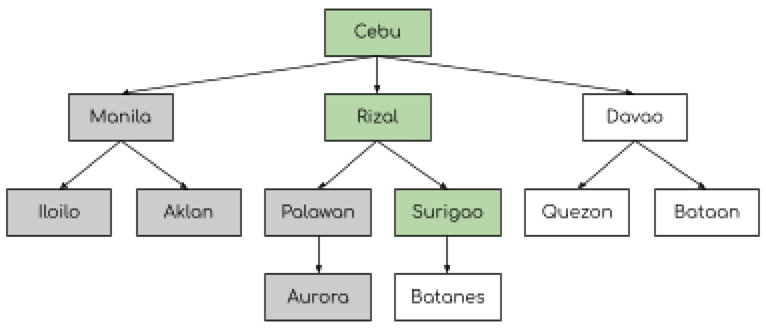
We travel to Aurora, but this is a dead-end.



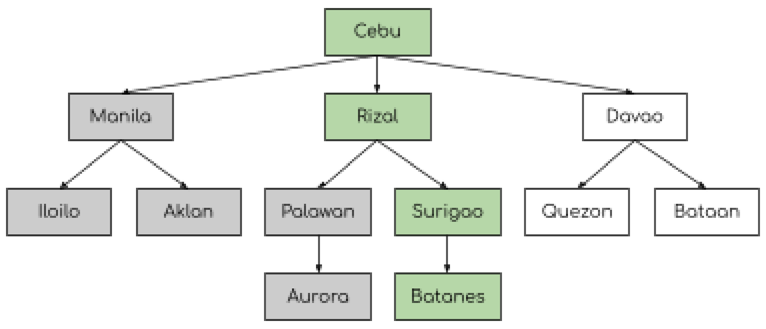
We backtrack to Palawan.



Since all paths from Palawan have been explored, we backtrack to Rizal.



We travel to Surigao.

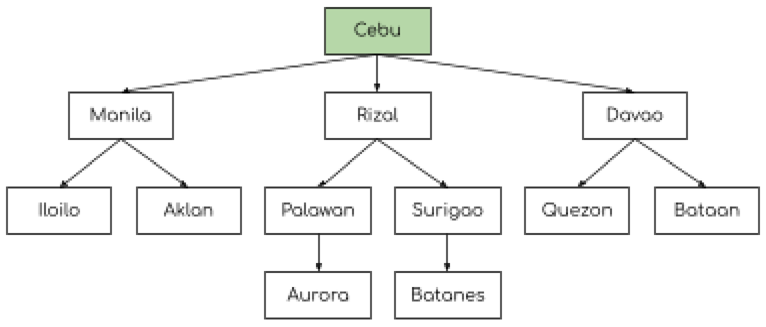


We finally reach Batanes!

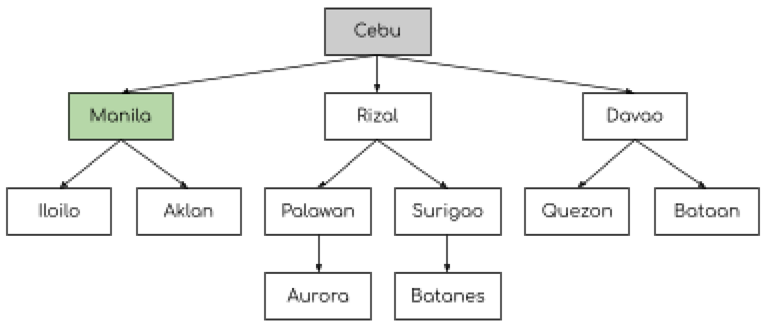
Breadth-First Search Algorithm

The idea of a breadth-first search is to explore all the "neighbors" before going deeper. Again, this is best understood through an illustration.

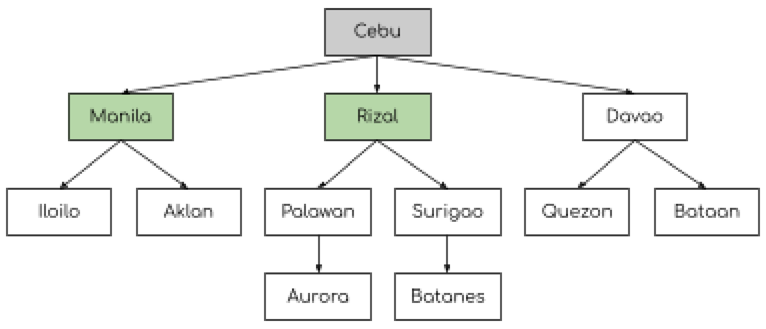
Start: Cebu **Destination**: Batanes



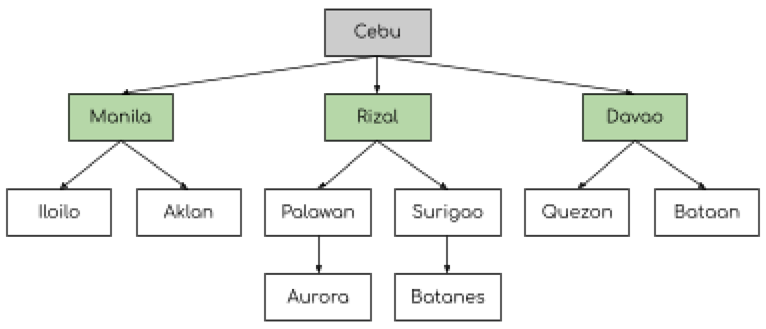
We start in Cebu.



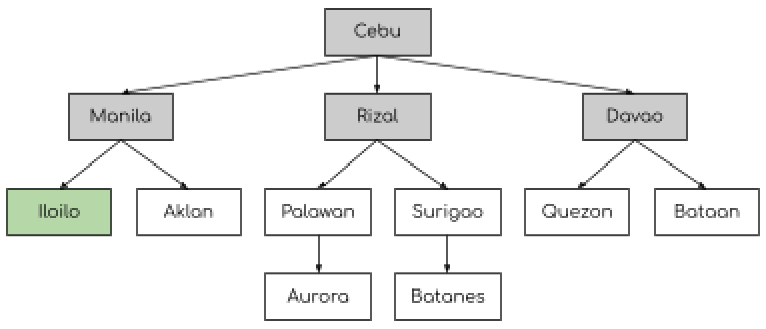
We go to the "next level" and travel to Manila.



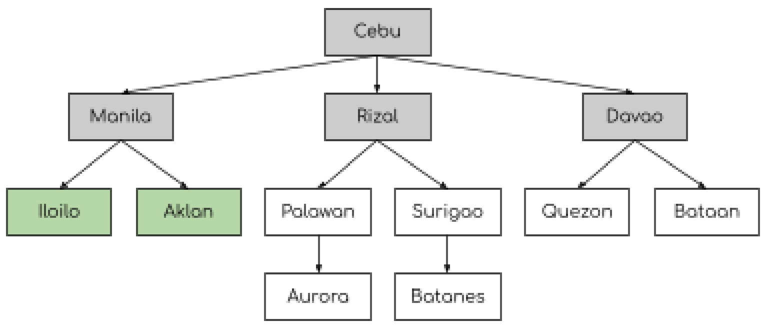
We travel to Rizal.



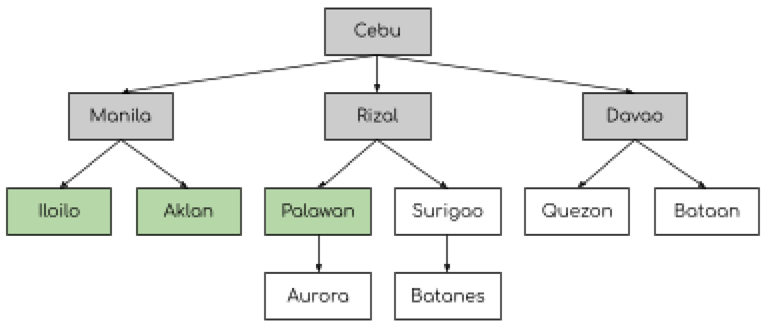
We travel to Davao.



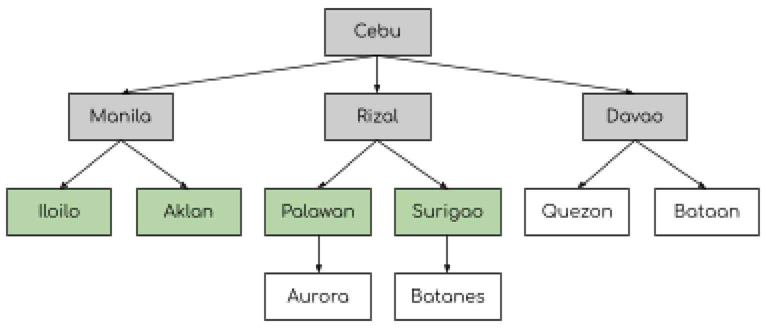
We go to the “next level” and travel to Iloilo.



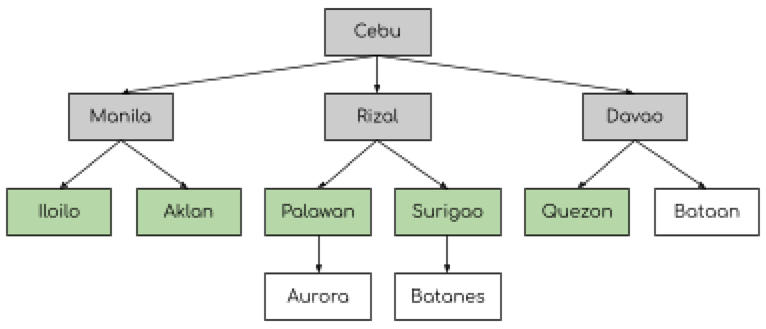
We travel to Aklan.



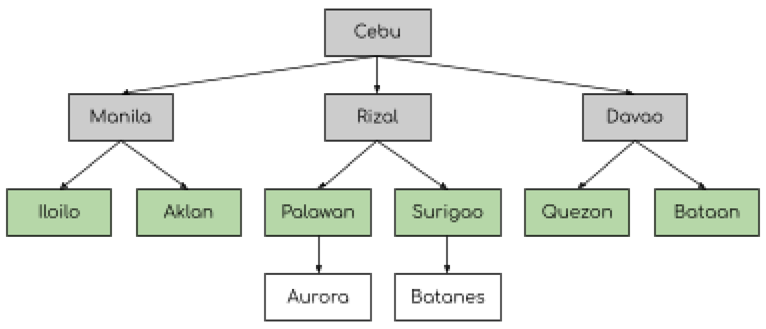
We travel to Palawan.



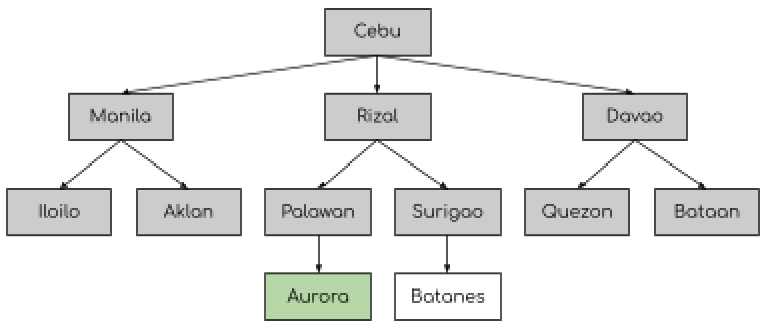
We travel to Surigao.



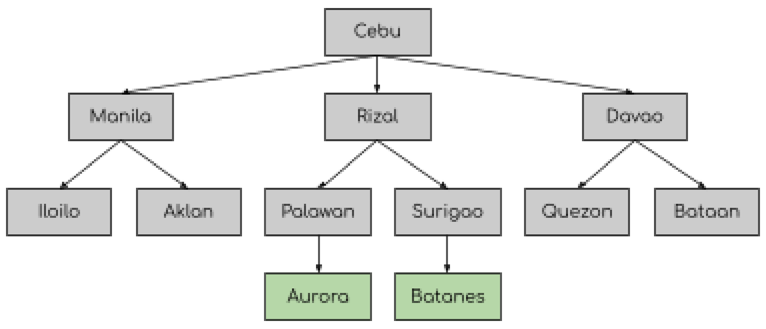
We travel to Quezon.



We travel to Bataan.



We go to the "next level" and travel to Aurora.



We finally reach Batanes!

💡 Looking Forward

When you start coding depth-first search and breadth-first search formally, you will realize their connection to two data structures that were introduced during our earlier AIEP sessions on computational thinking:

- Depth-first search makes use of stacks (last-in, first-out).
- Breadth-first search makes use of queues (first-in, first-out).

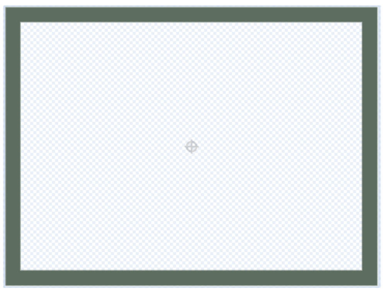
Walkthrough Automatic Maze Generation[3]

In this walkthrough, we will be exploring how recursive backtracking — the fusion of last session's topic (recursion) and this session's lesson (backtracking) — can be employed to automatically generate mazes in Scratch.

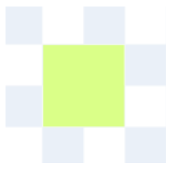
Here is a video of the final output.

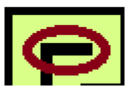
A. Setting up the Backdrop and Sprites

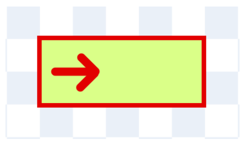
1. Convert the backdrop to a bitmap, and set its color to black.
2. Delete all the existing sprites.
3. Create a sprite named Bounds to mark the boundary of the stage.



4. Create another sprite named Maze. This sprite will have several costumes, the first of which is a costumed named Tile, which represents a single tile in our maze. Make sure that:
   - This tile is located at the center of the drawing area.
   - This tile is a square with a side of 8 pixels (it spans 2 boxes in the drawing area).



5. Duplicate the Tile costume, and name the new costume Corridor. This represents two connected tiles. Extend the tile so that it is now 20 pixels long (it spans 5 boxes in the drawing area).
   - Since we are just extending the tile, the center of the corridor should NOT coincide with the center of the drawing area.
   - The reason for the tile spanning 5 boxes instead of just 4 (even if it corresponds to two connected tiles) is to provide an allowance for the "borders" of the maze (encircled in the figure on the right).
   
   - Add an outline to the corridor and draw an arrow at its start. *These will serve as guides to indicate the corridor's direction while we are programming.*

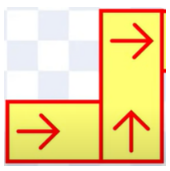


[3] Adapted from https://www.youtube.com/watch?v=22Dpi5e9uz8&t=886s

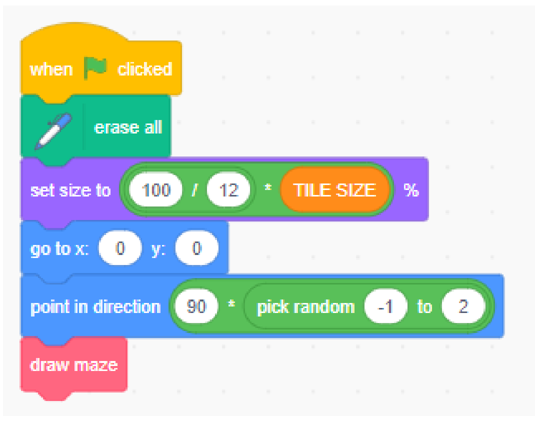
B. Drawing the Maze

1. Create a variable named `TILE SIZE` and set its value to 20 (feel free to experiment with this value).

2. Set the initial configuration of the first corridor in the maze:
   - Set the size of the sprite to `100 / 12 * TIME SIZE %`.
     - To understand this scaling, observe that one straight path (meaning, before a turn happens) is 12 pixels long (it spans 3 boxes in the drawing area).

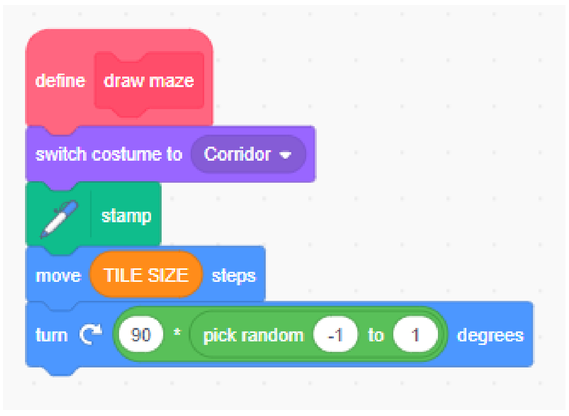


   - Position the sprite at the origin.
   - Randomize the direction of the sprite.
     - The possible directions are –90, 0, 90, and 180 degrees, corresponding to the four cardinal directions.
   - Call the My Block `draw maze`, which we will code in the next step.

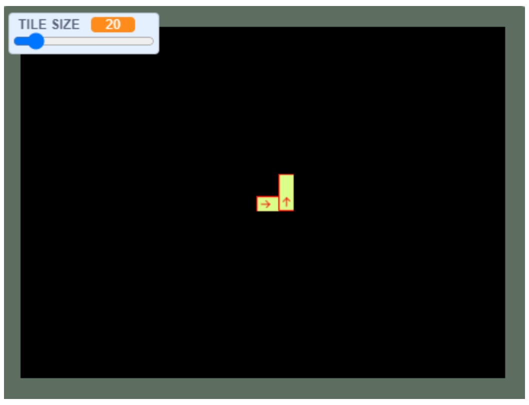


```
when [flag] clicked
erase all
set size to (100 / 12 * TILE SIZE) %
go to x: 0 y: 0
point in direction (90 * pick random -1 to 2)
draw maze
```

3. Create a My Block named `draw maze`.
   - Switch the costume to Corridor since a maze is essentially just a collection of connected corridors.
   - The `stamp` block creates a bitmap image of a corridor and stamps it on the stage, particularly on the last 8 pixels (two boxes) of the previous corridor.
   - If the direction of a corridor is upward, the corridor connected to it can simply extend the maze upward or shift the direction either to the left or to the right. However, it cannot go downward, as that would contradict the shape of a maze.
     - Generalizing this idea, the direction of the connected corridor is obtained by turning –90, 0, or 90 degrees. We should <u>NOT</u> turn 180 degrees.



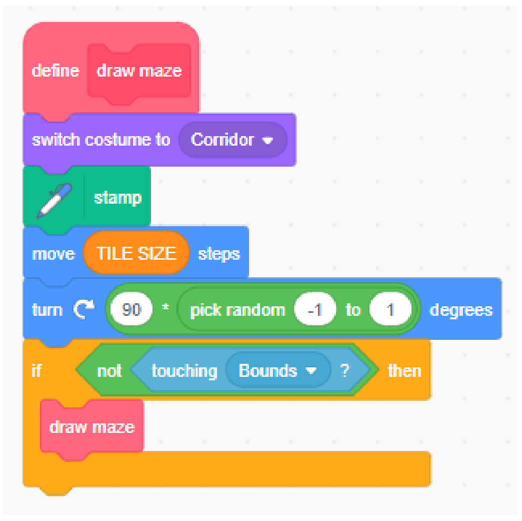
```
define draw maze
switch costume to Corridor
stamp
move (TILE SIZE) steps
turn ↻ (90 * pick random -1 to 1) degrees
```



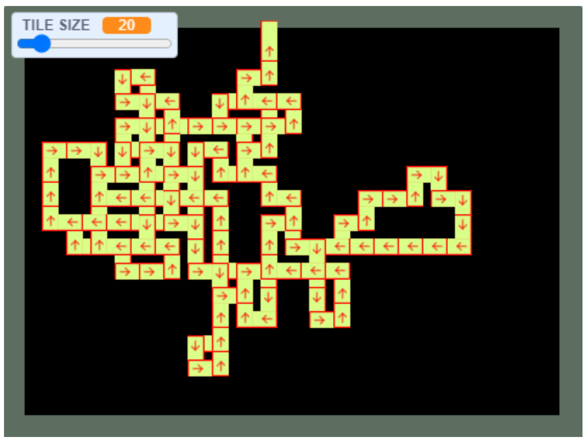
TILE SIZE 20

Asian MathSci League, Inc (AMSLI)
Partner: National Olympiad in Informatics Philippines (NOI.PH)
Email address: amsliphil@yahoo.com / ask@noi.ph
Contact Nos: +632-9254526 +63906-1186067

4. Using recursion, we keep on drawing connected corridors. The base case (stopping condition) is the moment we touch the sprite Bounds.

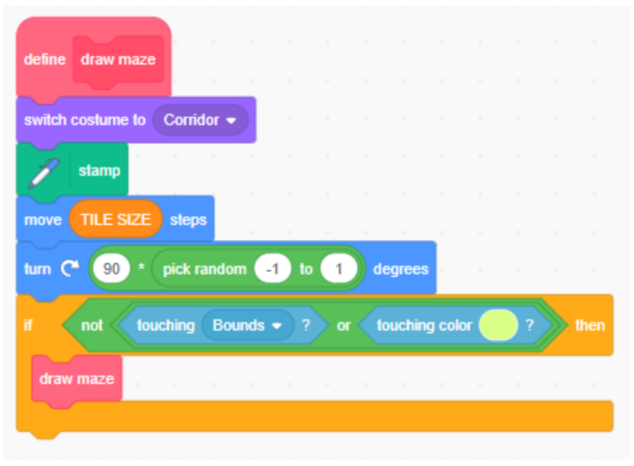


```
define draw maze
switch costume to Corridor
stamp
move TILE SIZE steps
turn ↻ 90 * pick random -1 to 1 degrees
if not touching Bounds ? then
    draw maze
```

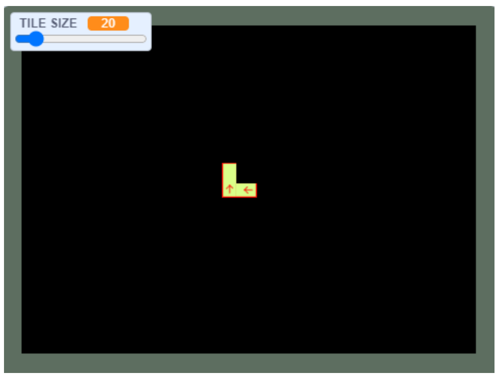


Observe that we get a snake-like figure that does stop when the boundary of the stage is hit. However, the corridors are overlapping each other.

5. The intuitive workaround would be to modify the base case so that stopping is also triggered when we touch another corridor (as sensed via the tile color).



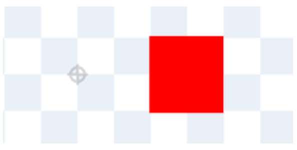
```
define draw maze
switch costume to Corridor
stamp
move TILE SIZE steps
turn ↻ 90 * pick random -1 to 1 degrees
if not touching Bounds ? or touching color ? then
    draw maze
```



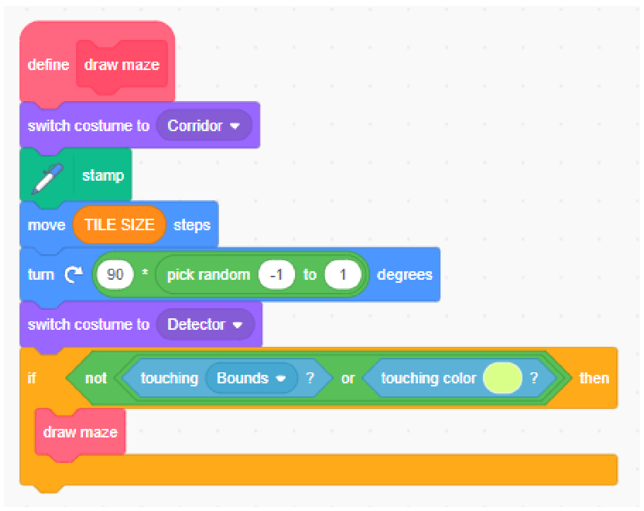
However, recall that a corridor is stamped on top of the previous corridor. Hence, stamping itself already causes the sprite to touch the previous corridor, thereby creating only one pair of connected corridors.

C. Adding a "Detector" Costume

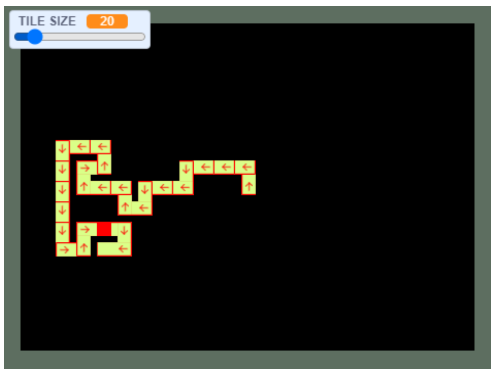
1. An ingenious solution is to create a detector to "sense" in advance if corridors will overlap. Duplicate the Corridor costume, and name the new costume Detector. Shrink the corridor to its last 8 pixels (2 boxes) and fill it with a different color for distinction. Make sure that the detector is NOT centered on the drawing area.



2. Incorporate the detector into the `draw maze` My Block by switching to this costume before performing the recursive procedure.

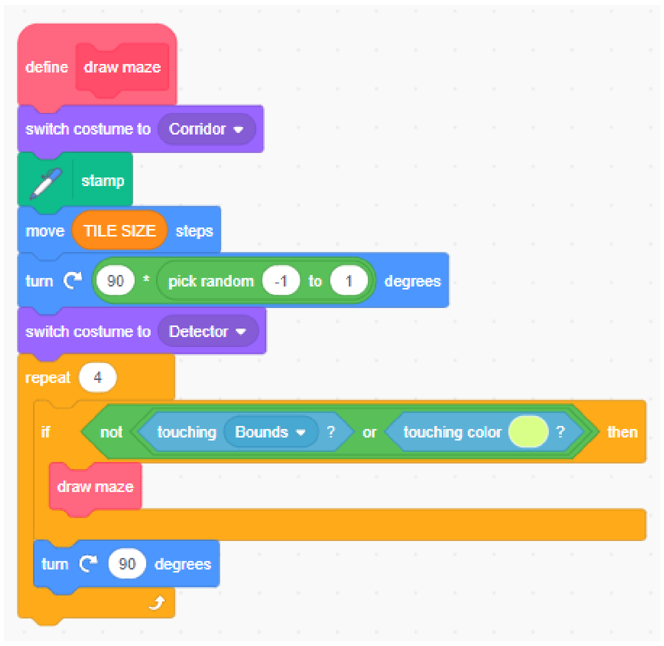


```
define draw maze
switch costume to Corridor ▾
stamp
move TILE SIZE steps
turn ↻ 90 * pick random -1 to 1 degrees
switch costume to Detector ▾
if not touching Bounds ▾ ? or touching color ? then
    draw maze
```

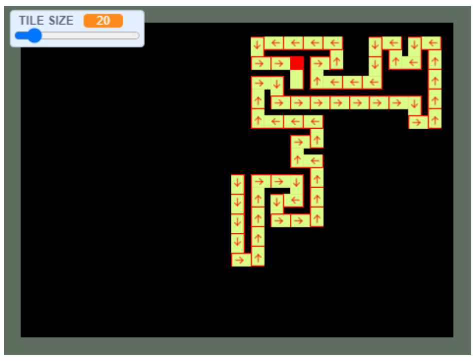


As intended, the generation stops when the corridors are about to overlap — but it simply stops and does not try another direction.

3. To this end, enclose the recursive procedure inside a loop that tries out all four cardinal directions.



```
define draw maze
switch costume to Corridor ▾
stamp
move TILE SIZE steps
turn ↻ 90 * pick random -1 to 1 degrees
switch costume to Detector ▾
repeat 4
    if not touching Bounds ▾ ? or touching color ? then
        draw maze
    turn ↻ 90 degrees
```



At this point, we are generating longer mazes, but we run into a major problem: eventually, the path traps itself in a dead-end and, despite moving in any of the cardinal directions, escape is nowhere in sight.

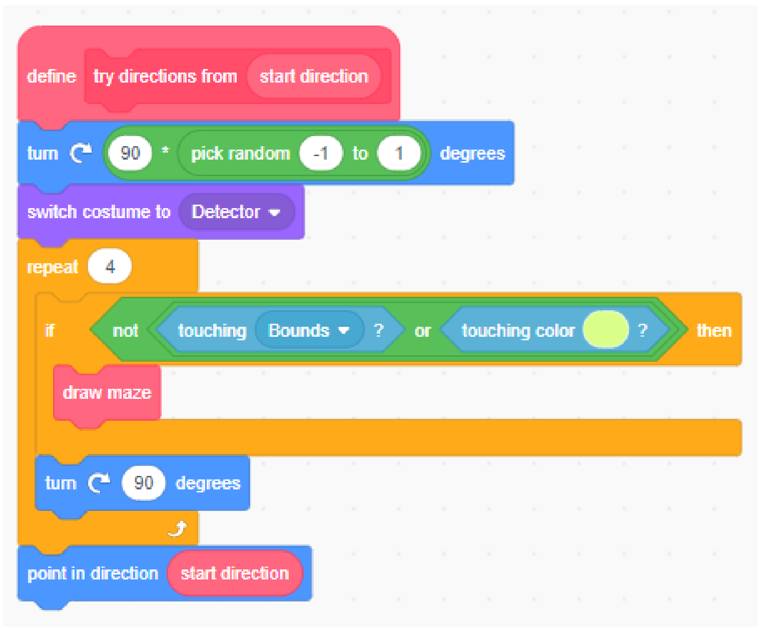
💭 Checkpoint

Does our maze generation procedure employ depth-first search or breadth-first search? What step is performed by graph search algorithms when a dead-end is encountered?

D. Recursive Backtracking

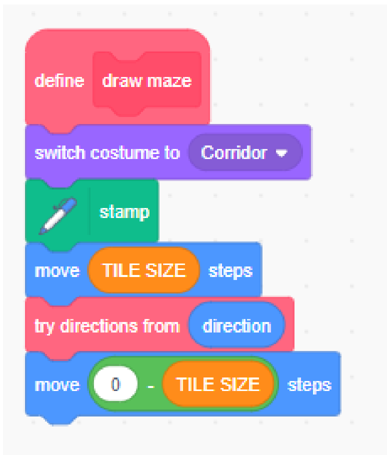
1. If your answers to the checkpoint are depth-first search and backtracking, then you are correct! One way to implement backtracking in Scratch is by creating a new My Block descriptively named `try directions from` and with one parameter: `start direction`.

2. The blocks inside the `draw maze` My Block (starting from the turning of the sprite) are transferred to this newly created My Block.
   - Note that the direction at the start of this My Block is the direction the corridor is facing *before* randomly trying out the four cardinal directions.
   - Therefore, at the end of this My Block, we have to return and face the original (starting) direction that the corridor is facing. This explains the need for the parameter `start direction`.

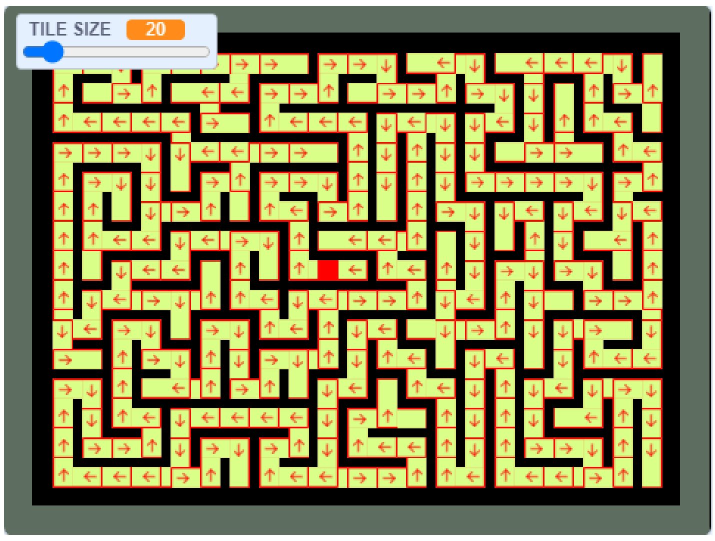


```
define try directions from (start direction)
turn ↻ ((90) * (pick random (-1) to (1))) degrees
switch costume to [Detector ▾]
repeat (4)
    if <not <<touching [Bounds ▾] ?> or <touching color [ ] ?>>> then
        draw maze
    end
    turn ↻ (90) degrees
end
point in direction (start direction)
```

3. Modify the `draw maze` My Block accordingly by calling `try directions from` and moving `0 - TILE SIZE` steps to complete the recursive backtracking.



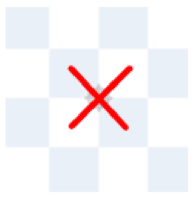
```
define draw maze
switch costume to [Corridor ▾]
stamp
move (TILE SIZE) steps
try directions from (direction)
move ((0) - (TILE SIZE)) steps
```



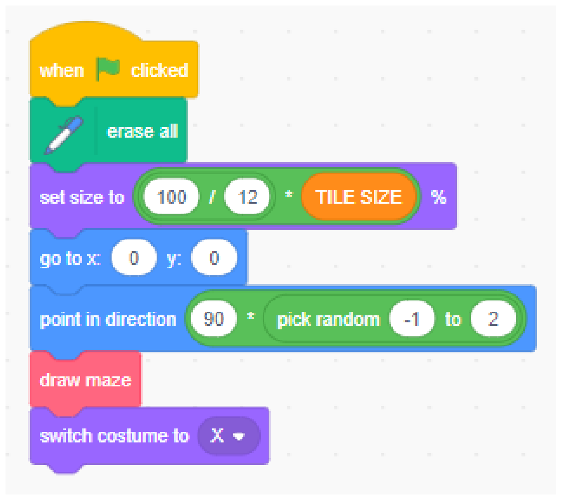
We now have a complete maze that fills the whole stage with corridors!

E. Finishing Touches

1. Remove the outline of the corridors, as well as the arrows.
2. Our maze is only missing one last thing: a marker for the goal or destination. Create a new costume for the Tile sprite, and name it X. The costume is literally an X; make sure that it fits inside one tile (which is 8 pixels or 2 boxes long and wide) and that it is positioned at the center of the drawing area.

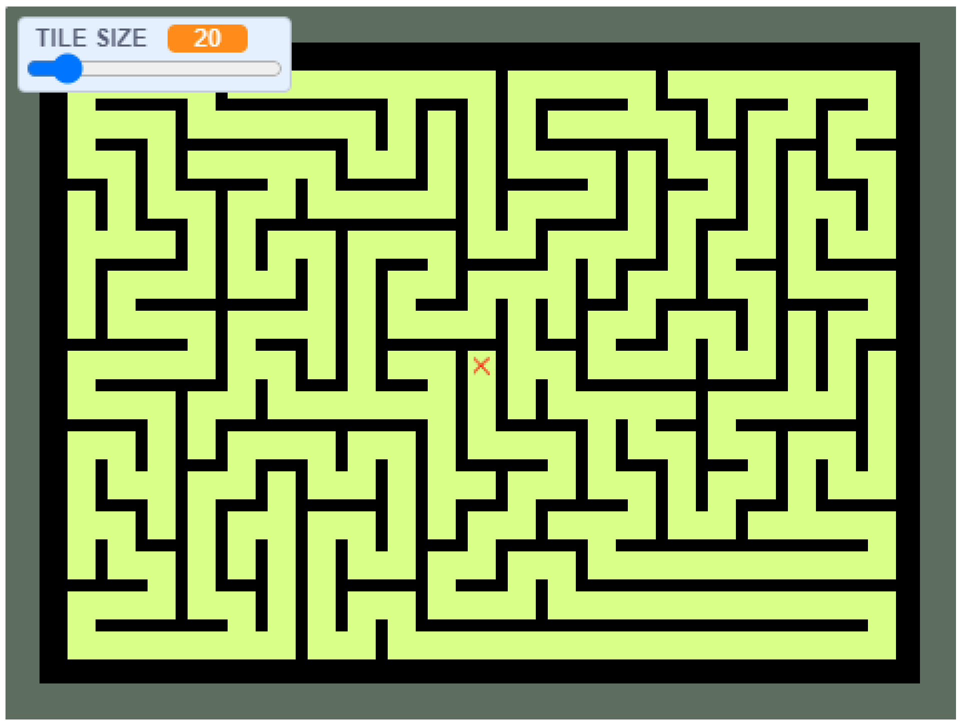


3. Switch to this costume after generating the maze.



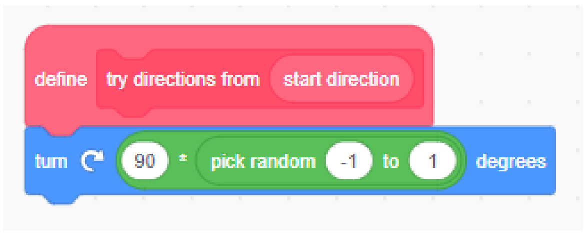
```
when [flag] clicked
erase all
set size to ((100 / 12) * (TILE SIZE)) %
go to x: (0) y: (0)
point in direction ((90) * (pick random (-1) to (2)))
draw maze
switch costume to (X v)
```

4. Enjoy generating mazes and tinkering with variables, such as `TILE SIZE`! Here is a video of the final output.



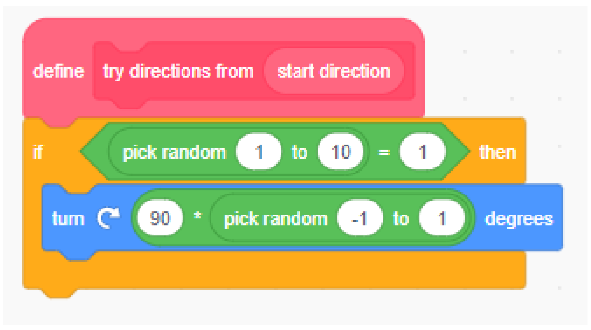
Enhancement Making the Maze Less Spiral

You might have noticed that the generated mazes are a bit too "spiral." This is the result of the direction of the corridor being randomized every time `try directions from` is called.



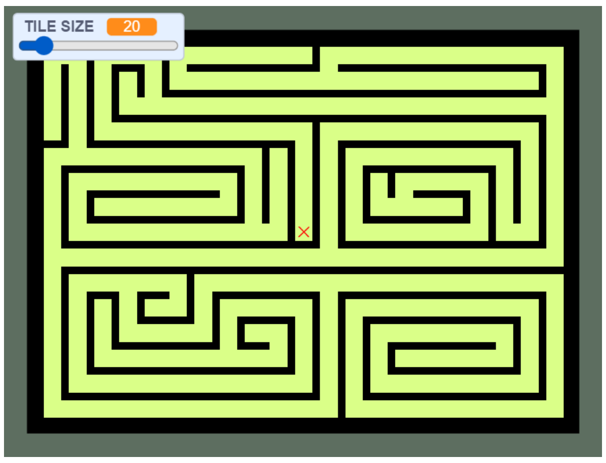
```
define try directions from (start direction)
turn ↻ ((90) * (pick random (-1) to (1))) degrees
```

In order to counteract this, we are going to introduce some bias against randomization. We first pick a random number from 1 to 10. The direction will be randomized only if this random number is equal to 1, reducing the chance of randomizing the direction to 10%.



```
define try directions from (start direction)
if <(pick random (1) to (10)) = (1)> then
    turn ↻ ((90) * (pick random (-1) to (1))) degrees
```

A generated maze after introducing this bias is shown below.



This time, the maze seems a bit too "straight." Can you find a way to increase the chance of randomizing the direction?

Challenge Adding a Maze Walker

Try to add a sprite that you can manipulate (for example, with the arrow keys) to travel the maze. In our next AIEP session, we are going to take this project a step further by learning how to *automatically* solve mazes — and dabbling in modern artificial intelligence!

`-- return 0; --`

Prepared by ***Mark Edward M. Gonzales***