Pathfinding Algorithm Challenge

Challenge Preamble

In this lab, you will explore the application of **Dijkstra's Algorithm** for pathfinding in a grid-based environment. You will implement the algorithm, visualize the result, and analyze its performance for finding the shortest path in a 2D maze. This lab tests your ability to implement classic graph algorithms in Python and generate visual output to verify correctness.

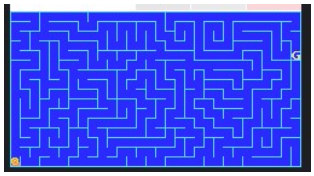
Part 1 – Theoretical Foundations

In this section, we will cover the foundational theory needed to understand the Dijkstra pathfinding algorithm. We will explore key concepts in computer vision, graph theory, data structures, and plotting. Understanding these will help you implement and visualize the algorithm effectively.

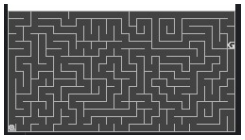
Part 1.1 – OpenCV Computer Vision Theory

In the context of this problem, OpenCV is used to process an image of a maze, identifying start, goal, and path locations using image matching techniques. Let's go over some key methods in OpenCV:

`imread()` Method:

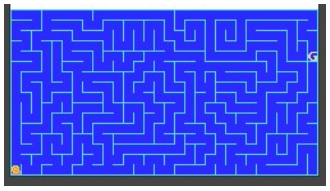
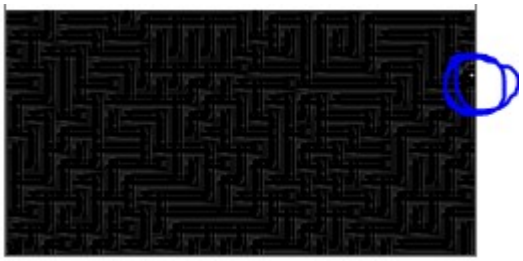
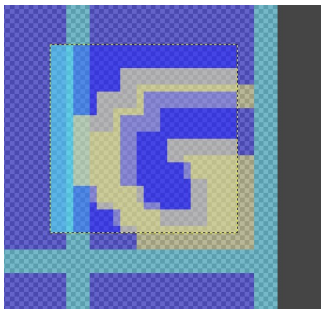
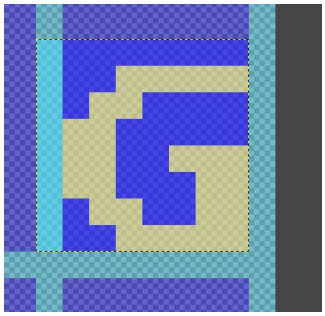
- The **`imread`**() function in OpenCV reads an image from a file into a matrix (array) format.
- Syntax: **`cv2.imread(filename, flags)`**
    - **`filename`**: The name of the file to be loaded.
    - **`flags`**: Specifies the way the image should be read (e.g., color, grayscale).
- Example:
    - ```
      img = cv2.imread("maze.png", cv2.IMREAD_COLOR)
      ```
    - This reads the image as a colored image.
    - 
      ```

`cvtColor`() Method:

- Converts an image from one color space to another, which is essential for operations like template matching that work on grayscale images.
- Syntax: **`cv2.cvtColor(src, code)`**
    - **`src`**: The source image.
    - **`code`**: The type of conversion (e.g., **`cv2.COLOR_BGR2GRAY`** converts from BGR (blue, green, red colors) to grayscale).
- Example:
    - ```
      gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
      ```
    - 

`match_template`() Method:

- This function is used to find a template image within a larger image by sliding the template across the main image and calculating a match score at each position.
- Syntax: **`cv2.matchTemplate(image, template, method)`**
    - **`image`**: The source image where you are searching for a match.
    - **`template`**: The template image to match.
    - **`method`**: The comparison method (e.g., **`cv2.TM_CCOEFF_NORMED`** for normalized correlation).
- Example:
    - ```
      result = cv2.matchTemplate(gray_img, template, cv2.TM_CCOEFF_NORMED)
      ```
    - This returns a matrix where each element is the match score for that position.

- If you have this image:  and you want to compare it with this template:  … He will take the template, slide it across the image and perform mathematical comparison on each pixel. The result gives this matrix (image):  . Everything seems to be a dark grey version of the path EXCEPT where both images match perfectly, as noted with a very white pixel in the blue circle.
- To explain this a bit more clearly, imagine this comparison. When we apply a transparency filter, we can see how it visually doesn't match:
-  no match -> output pixel will "gray" fit.
-  perfect match -> output pixel will be "white".

`minMaxLoc`() Method:

- This function finds the minimum and maximum values in a matrix and the corresponding locations of those values.

- Syntax: **`cv2.minMaxLoc(src)`**
    - src: The source array (result from **`match_template`**()).
- Example:
    - ```
      min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)
      ```
    - **`max_loc`** gives the location where the template best matches the source image.
    - This function conveniently traverses the previous result matrix (image), and returns a tuple of 4 values
        - **`min_val`**: in this context, probably close to 0, where there is no match at all;
        - **`max_val`**: close to white, for a perfect match;
        - **`min_loc`**: tuple for the location (X, Y) of the **min_val**;
        - **max_loc**: tuple for the location (X, Y) of the **max_val**.
      ```

Part 1.2 – Graph Theory

Dijkstra's algorithm operates on a graph, where each tile in the maze can be thought of as a node, and connections between adjacent tiles represent edges between nodes. Let's explore key graph-related data structures:

Linked List Data Structure:

- A **linked list** is a sequence of nodes where each node contains data and a pointer (or reference) to the next node.
- In this problem, a node could be a **Tile** object that stores references to neighboring tiles (north, south, east, west).

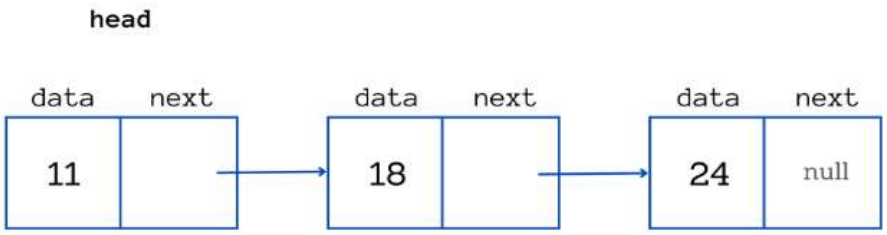


Figure 1 ref. https://www.freecodecamp.org/news/how-linked-lists-work/

- Example:

```
class Node:
    """A class representing a node in a singly linked list."""
    def __init__(self, data):
        self.data = data  # Store the data in the node
        self.next = None  # Pointer to the next node in the list

    def add_next(self, next_node):
        """Set the next node for the current node."""
        self.next = next_node
```

Tree Data Structure:

- A **tree** is a hierarchical data structure consisting of nodes, where each node may have child nodes. In the context of pathfinding, the search for the shortest path can be visualized as expanding a tree, where each node leads to its neighboring nodes. In this problem, each tile can have up to four connections (north, south, east, and west).

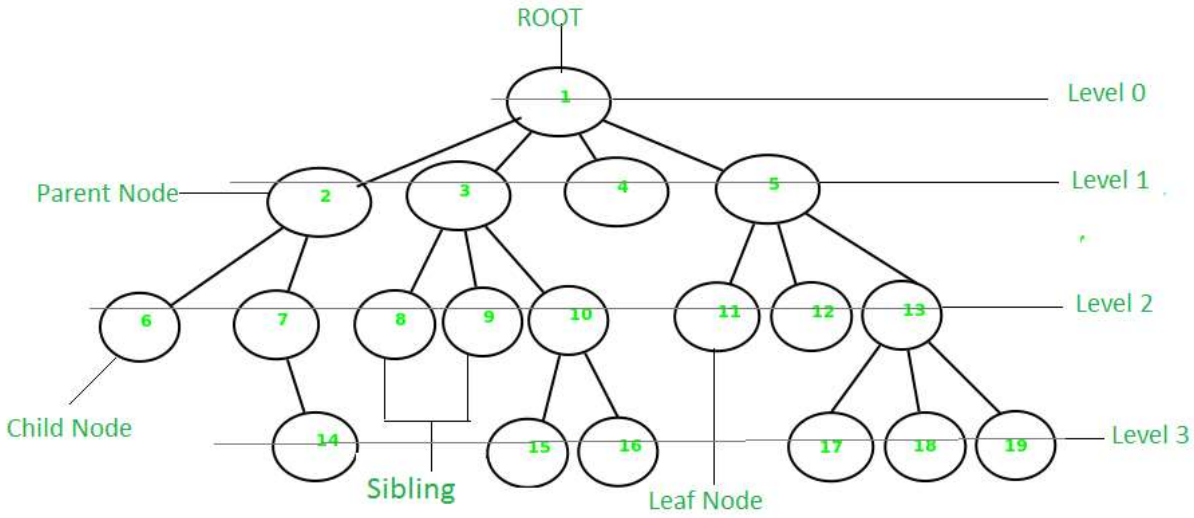
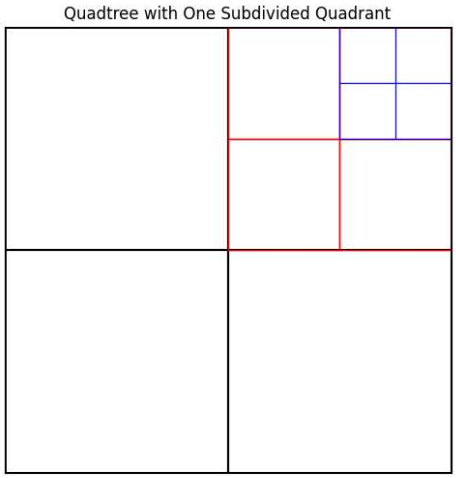


Figure 2 ref.: https://www.geeksforgeeks.org/what-is-data-structure-types-classifications-and-applications/

- There are many other types and applications of Trees data structures in video games. The following are examples (**FYI only**, not required for this challenge):
- **BSP Tree (Binary Space Partitioning Tree)**: A **BSP tree** is used to recursively subdivide a scene into convex sets. It is highly useful in rendering scenes where the order of drawing is important (such as with transparent objects or when using ray tracing). BSP trees were used in classic first-person shooters like *Doom* and *Quake* for **visibility sorting**.

- **Quadtree: Spatial Partitioning in 2D:** A **quadtree** is a tree data structure used to partition a 2D space by recursively subdividing it into four quadrants. Each internal node in the quadtree has four children (hence the name "quad"). The primary advantage of using a quadtree is that it allows for efficient spatial queries, such as collision detection, area-of-interest searches, and visibility culling in 2D spaces. **Use Case in Video Games:** Consider a 2D game with many moving objects (e.g., bullets, enemies, power-ups) in a large world. Without optimization, checking every object for potential collisions with every other object would be inefficient. By dividing the space into quadrants, a quadtree allows the game engine to narrow down collision checks to objects that are in the same quadrant or adjacent ones.



- **Octree: Spatial Partitioning in 3D:** An **octree** is the 3D counterpart of a **quadtree**. It partitions 3D space into eight smaller regions (octants), making it ideal for handling 3D environments. Each node in an octree has eight children, corresponding to the eight subregions of a cubic space.

Part 1.3 – Pathfinding Algorithm

Dijkstra's algorithm is a classic shortest-path algorithm used to find the minimum distance between nodes in a graph. It works by progressively exploring the shortest known distances from a source node, updating neighbors if a shorter path is found, until the goal node is reached, or all nodes are processed.

This algorithm uses a **priority queue** (using the **heapq** module) to always expand the least costly node next, ensuring that once the algorithm processes a node, it knows the shortest path to that node. Let's review a bit the theory of a **Queue** and a **priority Queue**.

Queue Data Structure:

A **Queue** is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. This means that the first element added to the queue is the first one to be removed. **Queues** are widely used in programming for managing tasks or processes where the order of operations matters.

Key Operations of a Queue:

1. **Enqueue**:
    - Adding an element to the **back** of the queue.
    - Example: If you enqueue A, B, and C into the queue, they will be ordered as [A, B, C], with A being at the front.
2. **Dequeue**:
    - Removing and returning the element from the **front** of the queue.
    - Following the FIFO principle, the first element inserted (at the front) is the first to be removed.
    - Example: After dequeuing the queue [A, B, C], the result will be [B, C] with A removed.

Common Uses of a Queue:

- **Task Scheduling**: Queues are used in operating systems and applications to manage tasks in order. For example, print jobs or process execution.
- **Breadth-First Search (BFS)**: In graph traversal, a queue is used to explore nodes level by level.
- **Buffer Management**: Queues are used in handling streaming data, where data needs to be processed in the order it arrives (e.g., audio or video streaming).

Queue Example in Python:

In Python, you can use the built-in collections.deque to implement a queue efficiently:

python Copy code

```python
from collections import deque

# Initialize a queue
queue = deque()

# Enqueue elements
queue.append("A")
queue.append("B")
queue.append("C")

print("Queue after enqueuing A, B, C:", list(queue))

# Dequeue an element
queue.popleft()
print("Queue after dequeuing:", list(queue))

# Peek at the front element
print("Front of the queue:", queue[0])

# Check if the queue is empty
print("Is the queue empty?", len(queue) == 0)
```

Output:

less Copy code

```
Queue after enqueuing A, B, C: ['A', 'B', 'C']
Queue after dequeuing: ['B', 'C']
Front of the queue: B
Is the queue empty? False
```

Priority Queue Data Structure

A **Priority Queue** is an advanced version of a standard queue in which each element has a priority associated with it. Elements are dequeued based on their priority rather than their order of insertion, meaning the element with the highest priority (or lowest value, depending on implementation) is dequeued first, regardless of when it was added. This is different from a **FIFO queue**, which always processes elements in the exact order they were added.

In Python, priority queues can be implemented using the **heapq** module, which uses a **binary heap** data structure to maintain the order efficiently.

Key Operations of a Priority Queue:

1. **Enqueue (Insertion)**:
    - An element is added to the priority queue along with its priority.
    - In a min-heap (the default behavior in heapq), elements with the **lowest priority value** are dequeued first.
2. **Dequeue (Remove)**:
    - The element with the **highest priority** (or lowest value) is removed from the queue.
    - This differs from a regular queue where the first element added is removed first (FIFO).

Difference Between a Queue and a Priority Queue:

- In a regular **queue** (FIFO), the **order** of element insertion determines when an element is dequeued.
- In a **priority queue**, the **priority value** determines which element is dequeued first, regardless of when it was added.

*Priority Queue Example in Python (Using **heapq**):*

Python's **heapq** module provides an efficient way to implement a priority queue with a binary heap. Here's an example:

```
task_list = []
heapq.heappush( *args: task_list, (2, "Task A", "Data for A"))
heapq.heappush( *args: task_list, (1, "Task B", "Data for B"))
heapq.heappush( *args: task_list, (3, "Task C", "Data for C"))
heapq.heappush( *args: task_list, (4, "Task D", "Data for D"))
heapq.heappush( *args: task_list, (0, "Task E", "Data for E"))
print(f"\n tasks (tuples): {task_list}")

highest_priority_task = heapq.heappop(task_list)
print(f"\n next task to do (highest_priority_task)): {highest_priority_task}")
```

```
tasks (tuples): [(0, 'Task E', 'Data for E'), (1, 'Task B', 'Data for B'), (3, 'Task C', 'Data for C'), (4, 'Task D', 'Data for D'), (2, 'Task A', 'Data for A')]

next task to do (highest_priority_task)): (0, 'Task E', 'Data for E')
```

Dijkstra’s algorithm

Dijkstra’s Algorithm, developed by Dutch computer scientist **Edsger W. Dijkstra** in 1956 and published in 1959, is one of the foundational algorithms in computer science for finding the shortest path in a weighted graph. Initially conceived to solve routing problems, such as determining the shortest route between two cities, this algorithm has become widely applicable in networking, mapping, and pathfinding in games. The core idea behind **Dijkstra's** Algorithm is to explore paths in order of increasing distance from a starting node, keeping track of the shortest path to each node along the way. By systematically choosing the shortest known distance at each step, **Dijkstra’s** Algorithm ensures that once a node’s shortest path is found, it cannot be improved upon. The algorithm’s efficiency and reliability have made it a fundamental tool for solving a variety of real-world shortest-path problems.

Key points of **Dijkstra's** algorithm:

- Works with positive edge weights.
- Greedily explores the node with the smallest known distance.
- Does not require a heuristic function, making it different from A*.

In this challenge, **Dijkstra's** algorithm will be applied to a grid of tiles. The distance between each neighboring tile is uniform (a distance of 1), which simplifies the problem.

Input: A grid representing the maze, with start and goal positions.
Output: The shortest path from the start to the goal.

The grid is represented as a collection of Tile. Each Tile has the following properties:

- Position (x, y)
- Neighbors (North, East, South, West)
- Distance from the start tile (initialized to infinity)

We use a **priority queue** to store the tiles to be explored, always processing the tile with the smallest distance. As we process each tile, we update its neighbors and add them to the queue if we find a shorter path.

Steps:

1. Initialize all tiles with infinite distance except the start tile, which is set to 0.
2. Use a priority queue to explore tiles, expanding the tile with the smallest known distance.
3. For each tile, update the distance of its neighbors if a shorter path is found.
4. When the goal tile is reached, reconstruct the path by tracing back through the neighbors.

Part 1.4 – Matplotlib: Creating a Plot (bonus)

To visualize the maze and the computed path, we use **Matplotlib**, a powerful library for creating plots in Python.

Subplots and Axes:

- **`subplots()`** is used to create a figure and a set of subplots (axes) for drawing the grid and path.
- Syntax: **`fig, ax = plt.subplots()`**
- Example:
- ```
  fig, ax = plt.subplots()
  ```

Drawing Shapes with **`patches.Rectangle()`**:

- In the context of the maze, each tile can be represented as a rectangle.
- **`patches.Rectangle((x, y), width, height)`** draws a rectangle at position (x, y) with a given width and height.
- Example:
- ```
  rect = patches.Rectangle((i, j), 1, 1, fill=False)
  ax.add_patch(rect)
  ```
- This will create an empty rectangle frame of 1 pixel wide, 1 pixel height at location **`i`** for the x-coordinate and **`j`** for the y-coordinate.

Annotating with Arrows:

- Arrows can be used to indicate connections between tiles. The **`annotate()`** method allows you to draw an arrow between two points on the plot.
- Syntax: **`ax.annotate('', xy=(x_end, y_end), xytext=(x_start, y_start), arrowprops=dict(...))`**
- ```
  arrow_params = dict(facecolor='black', shrink=0.05, headwidth=5, headlength=5)
  ax.annotate( text: '', xy=(10, 10), xytext=(5, 5), arrowprops=arrow_params)
  ```
  ```

- This will create a black arrow at 5,5 to 10, 10 with the mentioned parameters.

Displaying the Plot:

- Finally, use **`plt.show()`** to display the completed plot.

Part 2 – Code Walkthrough

Due to the extreme complexity of this project, we will divide it into parts, mostly organized by similarly than part 1.

Part 2.1 – OpenCV Computer Vision implementation

In this part of the challenge, we'll break down the functions required for the **computer vision** aspect of the maze-solving problem. The task involves reading an image of a maze, identifying important elements like **walls**, **start positions**, and the **goal**, and then converting the image data into a graph of tiles that Dijkstra's algorithm can use. This process relies heavily on **OpenCV**, a powerful library for image processing in Python.

find_goal(img_maze)

This function is responsible for identifying the **goal tile** in the maze image. The goal is identified by matching it against a set of goal templates.

Input: maze image a 241x129 pixel image of the maze 30x16 tiles of 8x8 pixels (+1 pixel).

Output: a tuple corresponding to the position of the goal in tile idx (e.g. (5, 20))

Key Steps:

Load Goal Templates:

The function loads four goal templates (**goal00.png**, **goal01.png**, etc.). These images represent different visual representations of the goal (one with 1 wall west, 1 wall north, 2 wall, no walls).

Match Templates with the Maze Image:

The function uses **cv2.matchTemplate()** to scan the entire maze image and find the best match for each goal template. It stores the location and match score for each template.

Find the Best Match:

The function uses cv2.minMaxLoc() to locate the highest match score for each template and store the corresponding location of the goal.

Return the Goal Position:

The function returns the coordinates of the start tile in terms of the grid by dividing the pixel coordinates by 8.

find_start(img_maze)

Similar to the **`find_goal()`** function**, `find_start()`** identifies the **start tile** in the maze image by matching it against a set of predefined **start** templates. (e.g. **start00.png**, **start01.png**, etc.).

Input: maze image a 241x129 pixel image of the maze 30x16 tiles of 8x8 pixels (+1 pixel).

Output: a tuple corresponding to the position of the goal in tile idx (e.g. (5, 20))

build_graph(img_gray, width, height)

This function is responsible for transforming the grayscale image of the maze into a navigable graph of tiles. It identifies different wall types from a set of predefined wall templates and constructs a 2D array (graph) of Tiles. We will cover the graph of Tiles in the next section (Part 2.2), but for now we will just construct the basic of wall detection and print the result at the screen.

Input:

- **img_gray**: Grayscale maze image a 241x129 pixel image of the maze 30x16 tiles of 8x8 pixels (+1 pixel).
- **width**: hardcoded value of 241
- **height**: hardcoded value of 129

Output: The graph contained in a **list** of **list** of **Tile**.

Key Steps:

Calculate the size of the grid:

The maze image is divided into 8x8 pixel tiles. The dimensions of the grid (number of tiles) are calculated by dividing the image width and height by 8.

Map Initialization:

We first need to create a 2D array (**tilemap**) where each element corresponds to a tile in the maze. The value in this array indicates the type of tile (whether it matches one of the wall templates or is a valid path). We can initialize this array with -1.

Load Wall Templates:

The function loads four wall templates (**wall00.png**, **wall01.png**, etc.), each representing different possible wall patterns. These templates will be compared to the maze tiles to identify walls.

Compare tiles with Wall Templates:

For each tile in the maze image, the function extracts an 8x8 pixel section and compares it to the wall templates using **`cv2.matchTemplate()`**. This function computes a similarity score between the tile and each wall template.

Identify the Most Likely Wall:

The cv2.matchTemplate() function returns a similarity score for each wall template. The template with the highest score is chosen as the best match for the tile. As we loop through all tiles in X and Y, we save the correct tile in our **`tilemap`** array.

Print the most likely wall template

Temporarily, we will build a 2D array (list of list) of int that will contains:

0: Wall west and wall north, match template **wall00.png**

1: Wall west, match template **wall01.png**

2: Wall north, match template **wall02.png**

3: No Wall, match template **wall03.png**

Return:

Temporarily before returning, we will print the whole graph in a way that will help us debug whether our algorithm worked:

Example output:

```
{0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}
{1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3}
{1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3}
…
{1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3}
{1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3}
```

Part 2.2 – Tilemap and graph implementation

In this section, we'll describe how the **Tiles** class and the remaining part of the **`build_graph()`** function work together to represent the maze as a navigable graph. This graph is made up of **tiles** that represent the sections of the maze, with each tile connected to its neighboring tiles. The maze is processed from an image, and its structure is translated into a graph that algorithms like Dijkstra's can use to find the shortest path.

The **Tile** Class

The Tiles class serves as the building block for the maze's graph structure. Each tile represents a square section of the maze, and it can be connected to its neighboring tiles in the north, south, east, and west directions.

*Key Features of the **Tile** Class:*

Neighbors:

Each tile keeps track of its neighbors (north, south, east, and west) using the attributes **`tile_north`**, **`tile_south`**, **`tile_east`**, and **`tile_west`**.

These are initially set to **`None`**, but they can later be linked to adjacent tiles during graph construction.

Position:

Each tile stores its grid position on the maze using the position attribute, which is a tuple of its (x, y) coordinates. This helps to uniquely identify each tile.

Distance for Pathfinding:

The distance attribute is initialized to infinity (float('inf')). This will later be used by Dijkstra's algorithm to track the shortest known distance from the start tile to each tile.

Special Attributes:

`is_start`: Marks the tile as the start point in the maze.

`is_goal`: Marks the tile as the goal point in the maze.

Neighbor Management:

The class includes methods (`set_north()`, `set_south()`, etc.) to assign neighboring tiles. This allows the tiles to be linked together based on the maze's structure.

Comparison method for Priority Queue:

The **`__lt__()`** method (less than) is implemented to allow tiles to be compared based on their distance. This is crucial for Dijkstra's algorithm, which will use a priority queue to select the next tile to process based on the shortest known distance.

Unique ID:

The **`get_id()`** method returns the tile's position as a tuple, allowing easy identification of the tile when necessary.

`build_graph(img_gray, width, height)` (revisited)

For this part of the tutorial, we will modify the **`build_graph()`** function to add proper creation of the graph and tilemap. We can remove the code that print the values, and add the following at the end.

Map Initialization:

We first need to add another 2D array that will now from the array of integers values. Having the Tile object, we will now create a 2D array of this object. Using list comprehension, you can create and initialize this 2D array with a fresh Tile object into a variable called **tilemap_obj.**

Tile Linking:

Looping through each recorded value in **tilemap**, you need to check for wall on each side of the tile. This part then checks the walls between neighboring tiles and, if there is no wall between two tiles, it links them together using the **`set_east()`**, **`set_west()`**, **`set_north()`**, and **`set_south()`** methods from the **Tiles** class.

Return the Graph:

Finally, the function returns the 2D list **tilemap_obj**, which contains all the **Tiles** objects, each linked to its valid neighbors. This structure will be used in the pathfinding part of the algorithm.

Part 2.3 – Pathfinding Algorithm: Dijkstra's Algorithm in Action

In this section, we will walk through the implementation of **Dijkstra's Algorithm** for finding the shortest path from the start tile to the goal tile in a maze, represented as a graph of tiles. This explanation builds upon the theoretical foundation provided in **Part 1.4** about **Dijkstra's Algorithm** and the use of the **priority queue** (covered in **Part 1.3** on the heap queue data structure).

pathfinding(timemap_obj, pos_start, pos_goal)

Input:

- **timemap_obj**: a 2D array (list of list) of 30x16 array of object **Tile.**
- **width**: hardcoded value of 241.
- **height**: hardcoded value of 129.

Output: The graph contained in a **list** of **list** of **Tile**.

Starting Point and Goal Point Initialization:

We begin by identifying the **start tile** and **goal tile** in the map_tiles 2D array based on the pos_start and pos_goal coordinates. These positions were likely found using the **`find_start()`** and **`find_goal()`** functions from **Part 2.1**.

The **start tile** has its distance set to 0 because the distance from the start to itself is zero.

Priority Queue Setup:

A **priority queue** is used to manage the tiles that need to be processed. In Python, this is efficiently implemented using the **heapq module**, which maintains a **min-heap** (covered in **Part 1.3**) where the tile with the smallest distance is always processed next.

The **start_tile** is added to the priority queue with an initial distance of 0, making it the first tile to be processed.

Tracking Previous Tiles:

A dictionary called **previous_tiles** is created to store the tile that each tile was reached from. This is important for reconstructing the path after the goal is reached.

The start tile has no previous tile, so it is mapped to **None**.

Processing the Priority Queue:

The algorithm enters a loop that continues as long as there are tiles in the priority queue.

In each iteration, the tile with the smallest distance (the closest tile) is removed from the queue using **`heapq.heappop()`** (explained in **Part 1.3**).

The current tile and its distance are processed to explore its neighbors.

Goal Check:

If the current tile is the goal tile, the algorithm stops. There is no need to process further tiles once the goal has been reached because Dijkstra's algorithm guarantees that the shortest path has been found at this point.

Exploring Neighbors:

The algorithm then checks each neighbor of the current tile. The neighbors are stored in the neighbor list of the Tiles class, which was constructed during the **`build_graph()`** function in **Part 2.2**.

If a neighbor exists (i.e., it's not **None**), the algorithm calculates a potential new distance to that neighbor. Since each move is assumed to have a cost of 1, the new distance is the current tile's distance plus 1.

Updating Neighbor Distances:

If the newly calculated distance to a neighbor is shorter than the previously known distance, the algorithm updates the neighbor's distance.

The neighbor is then added to the priority queue so it can be processed later.

Additionally, the **`previous_tiles`** dictionary is updated to show that the current tile is the one that leads to the neighbor. This information will later be used to reconstruct the path.

Reconstructing the Path

Once the goal has been reached, the algorithm needs to reconstruct the path by backtracking from the goal tile to the start tile. This is done using the **`previous_tiles`** dictionary.

Starting from the **`goal_tile`**, the algorithm traces back through the **`previous_tiles`** dictionary to find the tile that led to it.

This process continues until the **`start_tile`** is reached, at which point None will be encountered in the **`previous_tiles`** dictionary (since the start has no predecessor).

The path is stored in reverse order (from goal to start), so the **`path.reverse()`** call is made to correct the order.

Final Path Return:

Finally, the constructed path is returned as a list of tiles, where the first element is the start tile and the last element is the goal tile. This path can then be used for visualization or further analysis.

Part 2.4 – Visualization of the Graph with matplotlib (bonus)

In this section, we'll walk through how Matplotlib is used to visualize the maze, including the start point, goal point, walls, and the computed path. Visualization is an important part of understanding how the algorithm navigates the maze and finds the shortest path. By creating a clear visual representation, we can better interpret the output of our maze-solving algorithm.

Introduction to Matplotlib

Matplotlib is a comprehensive library in Python for creating static, animated, and interactive visualizations. In the context of the maze challenge, Matplotlib is used to draw the grid representing the maze and plot the shortest path from the start to the goal.

- **Grids and Shapes**: We will use Rectangle patches to represent the tiles in the maze (either walls or walkable areas).
- **Annotations**: Text annotations will be used to highlight the start and goal points.
- **Path Plotting**: We will draw the shortest path from start to goal using a line.

The visualize_map(timemap_obj, width, height, path) Function

The **`visualize_map()`** function is responsible for rendering the entire maze, including walls, open paths, start, goal, and the computed path. Here's a detailed breakdown of the function.

Input:

- **timemap_obj**: a 2D array (list of list) of 30x16 array of object **Tile.**
- **width**: hardcoded value of 30.
- **height**: hardcoded value of 16.
- **path**: return from the **`build_graph()`** function

Output: None