***Documentation of the Code: A Pathfinding (Simplified)*\***

**1. Objective**

This program is a simple version of the *A (A-Star) algorithm*\*, which is used to find the **best (shortest) path** between two locations.  
It finds the least-cost route from the **Start** to the **Goal** using both actual distance and an estimated distance (heuristic).

**2. Step-by-Step Code Explanation**

**Import Statement**

from collections import deque

This line imports deque (a double-ended queue) from Python’s collections module — though it is not used in this code.

**Class Definition**

class MapPath:

Defines a class named **MapPath** that contains all the logic for routes and pathfinding.

**Constructor (Initialization Method)**

def \_init\_(self, routes):

self.routes = routes

⚠️ There is a **typing mistake** here — it should be \_\_init\_\_ (with **double underscores**).  
Otherwise, the class won’t initialize correctly.

✅ Correct version:

def \_\_init\_\_(self, routes):

self.routes = routes

This constructor stores the route map (connections between cities or locations) inside the class.

**Neighbors Function**

def neighbors(self, place):

return self.routes[place]

This function returns all the **neighboring locations** connected to a given place.  
Example:  
neighbors('Start') → [('Town', 3), ('Forest', 5), ('Village', 8)]

**Heuristic Function (Estimate)**

def estimate(self, city):

estimates = {

'Start': 1,

'Town': 2,

'Forest': 2,

'Village': 1,

'Goal': 0

}

return estimates[city]

This function provides an **estimated cost (heuristic)** from a given city to the goal.  
It helps the algorithm guess which direction is closer to the goal.

* Goal has 0 (because it’s the destination).
* Others have approximate estimated values.

**Main Function — Finding the Path**

def find\_path(self, start, end):

This is the **main function** that runs the A\* pathfinding algorithm.

**Variables Used:**

open\_set = set([start])

closed\_set = set([])

cost = {start: 0}

parent = {start: start}

* **open\_set:** Cities that are still available to explore.
* **closed\_set:** Cities that are already visited.
* **cost:** Total distance (or cost) from the start to each city.
* **parent:** Stores from which city we reached the current one (used to build the final path).

**Selecting the Best Node**

for node in open\_set:

if current is None or cost[node] + self.estimate(node) < cost[current] + self.estimate(current):

current = node

This loop chooses the **node with the lowest (cost + estimate)** value —  
meaning the city that seems closest to the goal considering both the actual cost and the estimated distance.

**Checking if Goal is Reached**

if current == end:

If the current city is the goal, the algorithm reconstructs the path:

path = []

while parent[current] != current:

path.append(current)

current = parent[current]

path.append(start)

path.reverse()

print("Best path:", path)

It traces back the path using the parent dictionary, reverses it, and prints the **final best path**.

**Exploring Neighboring Cities**

for (next\_city, distance) in self.neighbors(current):

For each connected city:

* If it’s not already explored, add it to the open set.
* If a cheaper route is found, update the cost and parent.

This ensures the algorithm always keeps the lowest-cost route.

**Updating the Sets**

open\_set.remove(current)

closed\_set.add(current)

After exploring the current city, move it from the open set to the closed set.

**If No Path is Found**

print("No route found.")

return None

If the goal cannot be reached, this message is displayed.

**3. Route Map (Data)**

paths = {

'Start': [('Town', 3), ('Forest', 5), ('Village', 8)],

'Town': [('Village', 4)],

'Forest': [('Goal', 6)],

'Village': [('Goal', 2)]

}

This dictionary defines the **connections** between places and their distances.  
Example:

* Start → Town (3)
* Start → Forest (5)
* Start → Village (8)
* Village → Goal (2)

**4. Creating an Object**

map\_game = MapPath(paths)

This creates an object of the MapPath class and passes the routes to it.

**5. Running the Algorithm**

map\_game.find\_path('Start', 'Goal')

This line starts the pathfinding process to find the shortest route from “Start” to “Goal”.

**6. Example Output**

Best path: ['Start', 'Village', 'Goal']

This means the shortest route is from **Start → Village → Goal**.

**🧩 Summary Table**

| **Step** | **Description** |
| --- | --- |
| 1 | Class MapPath handles routes and pathfinding. |
| 2 | estimate() provides a heuristic guess for the remaining distance. |
| 3 | find\_path() performs the A\* search algorithm. |
| 4 | open\_set holds unvisited cities; closed\_set holds visited ones. |
| 5 | cost tracks total distance from start to each city. |
| 6 | When Goal is reached, the shortest path is printed. |
| 7 | If no path exists, “No route found” is displayed. |