Greedy Best-First Search Algorithm: Line-by-Line Explanation

-------------------------------------------------

import sys

import heapq

- Importing necessary libraries:

- `sys`: Used for handling system-level operations like maximum integer values.

- `heapq`: Implements the priority queue (min-heap) used for efficient node selection based on the smallest heuristic value.

-------------------------------------------------

class Graph:

def \_\_init\_\_(self, vertices, heuristics):

self.V = vertices

self.graph = {}

self.heuristics = heuristics # Heuristic values for each node

- Defining the `Graph` class:

- `vertices`: Number of nodes in the graph.

- `graph`: A dictionary to store the graph structure (adjacency list).

- `heuristics`: A dictionary that holds the heuristic values for each node.

-------------------------------------------------

def add\_edge(self, u, v, weight):

if u not in self.graph:

self.graph[u] = []

if v not in self.graph:

self.graph[v] = []

self.graph[u].append((v, weight))

self.graph[v].append((u, weight))

- The `add\_edge` method is used to add an edge between two nodes `u` and `v` with a specified weight:

- It checks if each node exists in the `graph` dictionary, and if not, creates a new entry.

- It then adds the weight of the edge to the adjacency list for both nodes `u` and `v`, making the graph undirected.

-------------------------------------------------

def greedy\_best\_first\_search(self, start, goal):

# Priority queue to select the node with the smallest heuristic value

open\_list = []

heapq.heappush(open\_list, (self.heuristics[start], start))

- Here, we initialize the priority queue (`open\_list`) and push the start node with its heuristic value.

- `heapq.heappush` ensures that the node with the smallest heuristic value is always at the front of the queue.

-------------------------------------------------

# Set to track visited nodes

visited = set()

- A set `visited` is created to track the nodes that have been visited during the search.

-------------------------------------------------

# Parent dictionary to reconstruct the path

parent = {start: None}

- The `parent` dictionary tracks the parent of each node, which helps in reconstructing the path from the start node to the goal node after reaching the goal.

-------------------------------------------------

while open\_list:

current\_h, current\_node = heapq.heappop(open\_list)

print(f"Visiting node {current\_node}, heuristic value: {self.heuristics[current\_node]}")

- The algorithm enters a `while` loop that runs as long as there are nodes in the priority queue (`open\_list`).

- It pops the node with the smallest heuristic value using `heapq.heappop`.

- The `current\_h` is the heuristic value of the node, and `current\_node` is the node itself.

- It prints the current node and its heuristic value.

-------------------------------------------------

if current\_node == goal:

print("Goal reached!")

# Reconstruct the path

path = []

total\_heuristic\_value = 0 # Initialize total heuristic value

while current\_node is not None:

path.append(current\_node)

total\_heuristic\_value += self.heuristics[current\_node] # Add the heuristic value of each node

current\_node = parent[current\_node]

path.reverse()

print("Path:", path)

print("Total heuristic value of the path:", total\_heuristic\_value)

return path

- If the current node is the goal node:

- Prints "Goal reached!"

- Reconstructs the path from the start node to the goal by following the parent nodes in reverse order.

- It also calculates the total heuristic value of the path by summing the heuristic values of the nodes along the path.

- The reconstructed path is printed, and the function returns the path.

-------------------------------------------------

visited.add(current\_node)

- Marks the current node as visited by adding it to the `visited` set.

-------------------------------------------------

for neighbor, \_ in self.graph.get(current\_node, []):

if neighbor not in visited:

parent[neighbor] = current\_node

heapq.heappush(open\_list, (self.heuristics[neighbor], neighbor))

- Iterates over the neighbors of the current node.

- If the neighbor has not been visited:

- Sets the current node as the parent of the neighbor.

- Pushes the neighbor into the priority queue (`open\_list`) with its heuristic value.

-------------------------------------------------

print("Goal not reachable.")

return None

- If the goal node is not reachable (i.e., the priority queue is empty), it prints "Goal not reachable" and returns `None`.

-------------------------------------------------

# Create the graph

print("Lokesh Dhoble 22131")

print("Estimated cost from current node to goal ")

graph = {}

n = int(input("Enter the number of edges: "))

for i in range(n):

edge = input("Enter the edge (source destination weight): ").split()

source, destination, weight = edge[0], edge[1], int(edge[2])

# Add the edge to the graph (undirected graph)

if source not in graph:

graph[source] = {}

if destination not in graph:

graph[destination] = {}

graph[source][destination] = weight

graph[destination][source] = weight

- The user is prompted to input the number of edges and the edges themselves.

- For each edge, the source and destination nodes along with the weight are added to the graph.

-------------------------------------------------

# Get the heuristic values from the user

heuristics = {}

for node in graph.keys():

heuristics[node] = int(input(f"Enter the heuristic value for node {node}: "))

- This loop asks the user to enter heuristic values for each node in the graph.

-------------------------------------------------

# Get the source and goal nodes

source = input("Enter the source node: ")

goal = input("Enter the goal node: ")

- The user is asked to input the source and goal nodes for the search.

-------------------------------------------------

# Create the Graph object

g = Graph(len(graph), heuristics)

- A `Graph` object is created using the number of nodes and the heuristic values.

-------------------------------------------------

# Add edges to the graph

for node in graph:

for neighbor in graph[node]:

g.add\_edge(node, neighbor, graph[node][neighbor])

- The edges from the user input are added to the `Graph` object using the `add\_edge` method.

-------------------------------------------------

# Perform Greedy Best-First Search

g.greedy\_best\_first\_search(source, goal)

- The `greedy\_best\_first\_search` method is called with the source and goal nodes to find the shortest path.

-------------------------------------------------

Possible Questions & Answers:

1️⃣ \*\*What is Greedy Best-First Search?\*\*

- It is an algorithm that finds the shortest path from the start node to the goal node by selecting the node with the smallest heuristic value (an estimate of the cost to reach the goal).

2️⃣ \*\*How does the algorithm select the next node to explore?\*\*

- It selects the node with the smallest heuristic value from the open list (priority queue).

3️⃣ \*\*What is the heuristic value?\*\*

- The heuristic value is an estimate of the cost to reach the goal node from the current node. It is used to prioritize which nodes should be explored next.

4️⃣ \*\*What is a priority queue and why is it used?\*\*

- A priority queue is a data structure that stores nodes along with their priorities (heuristic values). It ensures that the node with the smallest heuristic value is processed next, allowing the algorithm to follow the most promising path.

5️⃣ \*\*Why is the parent dictionary needed?\*\*

- The parent dictionary helps in reconstructing the path from the source node to the goal node once the goal is found.

6️⃣ \*\*Can this algorithm find the optimal solution?\*\*

- No, Greedy Best-First Search does not guarantee an optimal solution because it only focuses on the heuristic value and not the actual path cost.

7️⃣ \*\*When is Greedy Best-First Search useful?\*\*

- It is useful when we have a good heuristic estimate and want to quickly reach the goal, even if it’s not the optimal path.

8️⃣ \*\*What’s the difference between this algorithm and A\*?\*\*

- A\* uses both the actual path cost and the heuristic to decide which node to explore, whereas Greedy Best-First Search only uses the heuristic.

9️⃣ \*\*What are some applications of Greedy Best-First Search?\*\*

- It is used in AI search problems, robotics for pathfinding, and game AI for decision-making.

Job Scheduling Algorithm with Heuristic Values: Line-by-Line Explanation

-------------------------------------------------

profit = []

jobs = []

deadline = []

heuristics = []

- Initializing four empty lists to store the following:

- `profit`: The profit associated with each job.

- `jobs`: The names of the jobs.

- `deadline`: The deadline of each job.

- `heuristics`: The heuristic value for each job (used to prioritize scheduling).

-------------------------------------------------

print("Lokesh Dhoble 22131")

n = int(input("Enter the number of jobs: "))

- Displaying the student name and prompting the user to enter the number of jobs.

-------------------------------------------------

for i in range(n):

p = int(input("Enter the profit of job {}: ".format(i + 1)))

profit.append(p)

j = input("Enter the name of job {}: ".format(i + 1))

jobs.append(j)

d = int(input("Enter the deadline of job {}: ".format(i + 1)))

deadline.append(d)

h = int(input("Enter the heuristic value for job {}: ".format(i + 1)))

heuristics.append(h)

- A `for` loop to input the details of each job:

- The profit for the job is added to the `profit` list.

- The job name is added to the `jobs` list.

- The deadline for the job is added to the `deadline` list.

- The heuristic value is added to the `heuristics` list.

-------------------------------------------------

profitNJobs = list(zip(profit, jobs, deadline, heuristics))

profitNJobs = sorted(profitNJobs, key=lambda x: x[0], reverse=True)

- Combining all the job-related information (profit, job name, deadline, heuristic) into a single list called `profitNJobs`.

- Sorting this list in descending order based on the profit (`x[0]` represents the profit), so jobs with higher profits come first.

-------------------------------------------------

slot = [0] \* (n + 1)

total\_profit = 0

ans = ['null'] \* (n + 1)

- Initializing:

- `slot`: A list that tracks the available time slots (initialized to 0).

- `total\_profit`: A variable to accumulate the total profit from scheduled jobs.

- `ans`: A list initialized with `'null'` values to store the scheduled jobs in their respective slots.

-------------------------------------------------

for i in range(n):

job = profitNJobs[i]

for j in range(job[2], 0, -1):

if slot[j] == 0:

ans[j] = job[1]

total\_profit += job[0]

slot[j] = 1

break

- A `for` loop to schedule jobs:

- For each job in the sorted list `profitNJobs`, the algorithm attempts to place the job in the latest available slot before its deadline.

- If a slot is available (i.e., `slot[j] == 0`), it assigns the job to that slot, updates the total profit, and marks the slot as occupied (`slot[j] = 1`).

- The `break` statement ensures that the job is placed in only one available slot.

-------------------------------------------------

print("Jobs scheduled with Heuristic Values:")

for i in range(1, n + 1):

if ans[i] != 'null':

job\_index = jobs.index(ans[i])

print(f"Job: {ans[i]} | Profit: {profit[job\_index]} | Heuristic: {heuristics[job\_index]}")

- After scheduling, the program prints the list of jobs that have been scheduled, along with their profit and heuristic value.

- It checks each slot, and if a job is scheduled in that slot (`ans[i] != 'null'`), it fetches the corresponding job's profit and heuristic value.

-------------------------------------------------

# Output the total profit

print("Total profit:", total\_profit)

- Finally, the total profit from all scheduled jobs is printed.

Job Scheduling Algorithm with Heuristic Values: Line-by-Line Explanation

-------------------------------------------------

1️⃣ \*\*What is Job Scheduling?\*\*

- Job Scheduling is the process of assigning jobs to time slots in order to maximize profit. Each job has a profit, deadline, and heuristic value, and the goal is to schedule jobs in such a way that they complete before their deadlines while maximizing the total profit.

-------------------------------------------------

2️⃣ \*\*What is the significance of heuristic values in this algorithm?\*\*

- Heuristic values are used to prioritize jobs during scheduling. Higher heuristic values are associated with jobs that have a higher priority, and they influence the order in which jobs are considered for scheduling.

-------------------------------------------------

3️⃣ \*\*How does the algorithm handle scheduling?\*\*

- The algorithm first sorts the jobs in descending order of profit. Then, for each job, it tries to find an available time slot before the job's deadline. If a slot is found, the job is scheduled, and the total profit is updated.

-------------------------------------------------

4️⃣ \*\*Why are jobs sorted based on profit?\*\*

- Sorting jobs by profit ensures that the most profitable jobs are considered first. This approach aims to maximize the total profit by scheduling the higher-profit jobs as early as possible.

-------------------------------------------------

5️⃣ \*\*What happens if a job cannot be scheduled within its deadline?\*\*

- If a job cannot be scheduled (i.e., if no available time slot exists before its deadline), it is skipped, and no profit is added for that job.

-------------------------------------------------

6️⃣ \*\*How are the jobs stored and managed during the scheduling process?\*\*

- Jobs, their profits, deadlines, and heuristic values are stored in separate lists. These lists are then combined into a single list of tuples called `profitNJobs`. This combined list is sorted based on the profit to prioritize high-profit jobs.

-------------------------------------------------

7️⃣ \*\*How is the total profit calculated?\*\*

- The total profit is accumulated by adding the profit of each scheduled job. After all jobs have been considered, the total profit is printed.

-------------------------------------------------

8️⃣ \*\*What is the purpose of the `slot` array?\*\*

- The `slot` array keeps track of the availability of time slots. Initially, all slots are empty (`0`). When a job is scheduled in a time slot, that slot is marked as occupied (`1`).

-------------------------------------------------

9️⃣ \*\*What would happen if multiple jobs have the same profit?\*\*

- In case of multiple jobs having the same profit, the heuristic value comes into play, and the algorithm schedules the job with the higher heuristic value first, assuming both jobs have the same priority in terms of profit.

-------------------------------------------------

🔟 \*\*Is the algorithm guaranteed to schedule all jobs?\*\*

- No, the algorithm is not guaranteed to schedule all jobs. It can only schedule jobs that have available time slots before their deadlines. Jobs that cannot be scheduled in this manner are skipped.

-------------------------------------------------