A\* Algorithm

Changes:

1. import library to make finding the smallest node faster.  
2. Improved the heuristic values to make the search smarter.  
3. Given a condition to make sure the start and goal nodes exist.  
4. Cleaned up the code formatting so it looks easy to read.  
5. The code now runs without errors and gives the correct path output.

**New code**

import heapq  
  
class Graph:  
 def \_\_init\_\_(self, adjacency\_list):  
 self.adjacency\_list = adjacency\_list  
  
 def get\_neighbors(self, v):  
 return self.adjacency\_list.get(v, [])  
  
 def all\_nodes(self):  
 nodes = set(self.adjacency\_list.keys())  
 for edges in self.adjacency\_list.values():  
 for nbr, \_ in edges:  
 nodes.add(nbr)  
 return nodes  
  
 def h(self, n):  
 H = {'A': 7, 'B': 6, 'C': 2, 'D': 0}  
 return H.get(n, 0)  
  
 def a\_star\_algorithm(self, start, goal):  
 if start not in self.all\_nodes() or goal not in self.all\_nodes():  
 print(f"Invalid start or goal node: {start}, {goal}")  
 return None  
  
 open\_heap = []  
 heapq.heappush(open\_heap, (0 + self.h(start), start))  
 g = {start: 0}  
 parents = {start: None}  
 closed\_set = set()  
  
 while open\_heap:  
 f, current = heapq.heappop(open\_heap)  
  
 if current in closed\_set:  
 continue  
  
 if current == goal:  
 path = []  
 while current:  
 path.append(current)  
 current = parents[current]  
 path.reverse()  
 print("Path found:", path)  
 return path  
  
 closed\_set.add(current)  
  
 for neighbor, weight in self.get\_neighbors(current):  
 if neighbor in closed\_set:  
 continue  
  
 tentative\_g = g[current] + weight  
  
 if neighbor not in g or tentative\_g < g[neighbor]:  
 g[neighbor] = tentative\_g  
 heapq.heappush(open\_heap, (tentative\_g + self.h(neighbor), neighbor))  
 parents[neighbor] = current  
  
 print("Path does not exist!")  
 return None  
  
  
adjacency\_list = {  
 'A': [('B', 1), ('C', 3), ('D', 7)],  
 'B': [('D', 5)],  
 'C': [('D', 12)]  
}  
  
graph = Graph(adjacency\_list)  
graph.a\_star\_algorithm('A', 'D')

**old code**

from collections import deque

class Graph:

def \_\_init\_\_(self, adjacency\_list):

self.adjacency\_list = adjacency\_list

def get\_neighbors(self, v):

return self.adjacency\_list[v]

def h(self, n):

H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

return H[n]

def a\_star\_algorithm(self, start\_node, stop\_node):

open\_list = set([start\_node])

closed\_list = set([])

g = {}

g[start\_node] = 0

parents = {}

parents[start\_node] = start\_node

while len(open\_list) > 0:

n = None

for v in open\_list:

if n == None or g[v] + self.h(v) < g[n] + self.h(n):

n = v

if n == None:

print('Path does not exist!')

return None

if n == stop\_node:

reconst\_path = []

while parents[n] != n:

reconst\_path.append(n)

n = parents[n]

reconst\_path.append(start\_node)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

for (m, weight) in self.get\_neighbors(n):

if m not in open\_list and m not in closed\_list:

open\_list.add(m)

parents[m] = n

g[m] = g[n] + weight

else:

if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n

if m in closed\_list:

closed\_list.remove(m)

open\_list.add(m)

open\_list.remove(n)

closed\_list.add(n)

print('Path does not exist!')

return None

adjacency\_list = {

'A': [('B', 1), ('C', 3), ('D', 7)],

'B': [('D', 5)],

'C': [('D', 12)]

}

graph1 = Graph(adjacency\_list)

graph1.a\_star\_algorithm('A', 'D')