بانام معمار هستی

نام درس :

مبانی و هوش مصنوعی

نام استاد:

دکتر کارشناس

پروژه:

بهترین پرواز: الگوریتم دایکسترا و *A

پدیدآورندگان:

علی آقاخانی

حمید مهرانفر

شماره دانشجویی:

۴۰۰۳۶۶۳۰۰۲

۴۰۰۳۶۱۳۰۵۸

دانشگاه اصفهان

بهترین پرواز : الگوریتم دایکسترا

منابع :

https://bradfieldcs.com/algos/graphs/dijkstras-algorithm/

https://www.geeksforgeeks.org/priority-queue-set-1-introduction/

https://docs.python.org/3/library/heapq.html

https://www.tutorialspoint.com/python_data_structure/python_heaps.htm

https://chat.openai.com/

https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-using-priority_queue-stl/

https://www.geeksforgeeks.org/how-to-check-the-execution-time-of-python-script/

ماژول گراف :

در این فایل کلاس های مورد نیاز گراف پیاده سازی شده است . گراف توسط روشadjacency map پیاده سازی شده است . در واقع هر node توسط یک dictionary که کلید های ان node های همسایه و مقدار ان کلید ها edge متصل کننده است ، مقدار دهی می شود و همه ی node ها در یک لیست ذخیره می شوند. مسافت ، قیمت پرواز و زمان پرواز در کلاس edge ذخیره می شود و بقیه ی اطلاعات در کلاس node ذخیره می شود . همچنین توسط تابع checkValues بررسی می شود که اطلاعات node ها تکراری نباشد .تابع getVertex ، یک رشته از نام فرودگاه میگیرد و node ان را بر میگرداند.

ماژول دایکسترا:

این فایل مربوط به پیاده سازی الگوریتم دایکسترا است . تابع find_shortest_path در واقع کوتاه ترین مسیر را بااستفاده از الگوریتم دایکسترا بر می گرداند .

توضیح توابع :

الف)سازنده

```
def __init__(self, graph):
    self.graph = graph
    self.distances = {}
    self.previous_airports = {}
```

در کد روبه رو ابتدا در سازنده کلاس سه متغیر را تعریف میکنیم. Distance یک دیکشنری است که برای دنبال کردن کوتاه ترین فاصله از فرودگاه مبداء تا سایر فرودگاه در گراف استفاده می شود. previous_airports نیز تمام فرودگاه های قبلی که موردنظر ما هستند را در خود نگه داری می کنند.

ب) تابع find_shortest_path :

```
# Initialize distances
self.distances = {node.data.airport: float('inf') for node in self.graph.vertices}
self.distances[start] = 0
```

در قسمت اول این تابع ما distance اینگونه تعریف می کنیم که برای هر نود در گراف مقدار فاصله اش را بی نهایت مقدار دهی شود . در خط بعدی ما فاصله خود شروع کننده از خودش را صفر مقدار دهی میکنیم.

```
    start_node = self.graph.get_vertex(start)
    priority_queue = [(0, start_node)]
    self.previous_airports = {}
```

در این قسمت از کد ما نود شروع کننده را از گراف پیدا می کنیم و در star_node قرار می دهیم. در خط بعدی ما priority_queue داریم که در واقع یک تاپل است از که دو مقدار را در خود نگه می دارد : ۱. فرودگاه ۲. فاصله کنونی . در خط بعد نیر این صف با فرودگاه مبداء مقدار دهی اولیه می شود .

```
while priority_queue:
    current_distance, current_airport = heapq.heappop(priority_queue)
```

در این قسمت در هر تکرار حلقه، فرودگاهی را با کمترین فاصله از priority_queue پاپ می کند . این با استفاده از heappop از ماژول heapq به دست می آید، که تضمین می کند که فرودگاه با کمترین فاصله شناخته شده ابتدا برداشته می شود(اولین بار خود مبداء برداشته می شود).

```
if current_airport.data.airport == end:
    path = []
    while current_airport:
        path.insert(0, current_airport)
        current_airport = self.previous_airports.get(current_airport)
    return path
```

در این قسمت چک می کنیم اگر فرودگاه کنونی با فرودگاه مقصد یکی بود آنگاه یعنی ما کوتاه مسیر را پیدا کرده ایم . سپس به صورت بازگشتی از فرودگاه کنونی (مقصد) شروع کرده و به به لیست path اضافه می کنیم .

بعد با استفاده از previous_airports فرودگاه قبلی را به دست می آوریم . این کار را انقدر انجام می دهیم تا دیگر current_airport مقدارش غیر مجاز شود (None). و بعد لیست path را بر میگردانیم.

```
if self.distances[current_airport.data.airport] < current_distance:
    continue
```

این شرط بررسی می کند که آیا فاصله ذخیره شده تا current_airport کمتر از current_distance از صف اولویت است. اگر اینطور است، به این معنی است که یک مسیر کوتاهتر به فرودگاه کنونی قبلاً پیدا شده است، بنابراین نیازی به حساب کردن آن نیست. پس به بعدی می رویم.

```
for neighbor, value in current_airport.neighbors.items():
    total_distance = current_distance + value.distance
    if total_distance < self.distances[neighbor.data.airport]:
        self.distances[neighbor.data.airport] = total_distance
        self.previous_airports[neighbor] = current_airport
        heapq.heappush(priority_queue, (total_distance, neighbor))
```

این قسمت در واقع پیاده سازی الگوریتم است. با یک حلقه فور به همسایه های نود فعلی (فرودگاه فعلی) می رویم. total_distance کل مسیری برای رسیدن به همسایه اش را توسط همان فرودگاه فعلی حساب می کند. شرط if چک می کند که اگر این total_distance کمتر از فاصله فعلی که همان distance[] هست یا نه . اگر بود distance[neighbor] را با total_distanceکوتاه‌تر به‌روزرسانی می‌کند، previous_airports را به‌روزرسانی می‌کند تا نشان دهد که مسیر از ابتدا تا همسایه از فرودگاه فعلی می‌گذرد، و همسایه را با فاصله کلی به‌روزشده‌اش به صف اولویت پوش می کند.

بهترین پرواز : الگوریتم *A

ماژول گراف :

در این فایل کلاس های مورد نیاز گراف پیاده سازی شده است . گراف توسط روشadjacency map پیاده سازی شده است . در واقع هر node توسط یک dictionary که کلید های ان node های همسایه و مقدار ان کلید ها edge متصل کننده است ، مقدار دهی می شود و همه ی node ها در یک لیست ذخیره می شوند.مسافت ، قیمت پرواز و زمان پرواز در کلاس edge ذخیره می شود و بقیه ی اطلاعات در کلاس node ذخیره می شود . همچنین توسط تابع checkValues بررسی می شود که اطلاعات node ها تکراری نباشد .

تابع getVertex ، یک رشته از نام فرودگاه میگیرد و node ان را بر میگرداند.

ماژول a_star :

این فایل مربوط به پیاده سازی الگوریتم a star است . تابع gn ، g(n) مربوط به node ها را محاسبه می کند که مقدار ان برابر با مجموع مسافت و دو برابر قیمت پرواز بین دو فرودگاه است. تابع heuristic ، h(n) مربوط به node ها را محاسبه میکند که مقدار ان برابر با مسیر مستقیم بین دو فرودگاه است . این کار با استفاده از دو متد latitude و longitude انجام می شود . (برای محاسبه مسیر مستقیم از سایت geeksforgeeks استفاده شده است)

تابع a_star_search :

این تابع دو فرودگاه میگیرد و بهترین مسیر را براساس الگوریتم a_star ذخیره میکند . به این صورت که ابتدا node مربوط به این دو فرودگاه را ذخیره میکند . سپس لیست frontier برای بررسی node ها ، لیست visited برای جلوگیری از بررسی مجدد ، دیکشنری distance برای نگهداری مقادیر f(n) و دیکشنری answer برای نگهداری بهترین مسیر پرواز تعریف می کند.

ابتدا داخل frontier فقط مبدا قرار دارد و در اجرای اول انتخاب می شود . سپس مقدار f(n) ان در distance قرار میگیرد . سپس هزینه تمام همسایه های ان (که در واقع برابر است با مجموع هزینه مصرف شده تا ان لحظه با g(n)) حساب شده و داخل distance قرار میگیرد . همچنین همه ی node ها هم

داخل frontier ذخیره می شوند . در اخر هم node مبدا از لیست frontier حذف و به لیست visited اضافه میشود .

در اجرای بعدی ، برای هر node مجموع هزینه ی مصرف شده تا ان لحظه و h(n) ان node محاسبه می شود و کمترین ان انتخاب می شود . سپس مثل قبل ، همسایه های node انتخاب شده به داخل frontier وارد میشوند و هزینه ی ان ها هم داخل distance قرار میگیرد . در اخر هم node انتخاب شده از frontier حذف میشود و به visited اضافه می شود .

در بررسی همسایه های node انتخاب شده ، اگر هزینه دسترسی به یک node که قبلا از طریق یک مسیر دیگر بررسی شده است ، با استفاده از این مسیر کمتر شود ، مقدار هزینه ان اپدیت میشود، همچنین دوباره به frontier اضافه میشود که دوباره مورد بررسی قرار گیرد .

در هر اجرا ، همسایه های کمترین node انتخاب شده به دیکشنری answer اضافه می شود که value ان برابر است با کمترین node انتخاب شده . با این کار ، مسیر رسیدن به مقصد مشخص می شود .

ماژول time :

در این ماژول execution time برای الگوریتم ها محاسبه می شود . به این صورت که تایم شروع در متغیر start_time و تایم پایان در end_time ذخیره میشود . تفاوت این دو متغیر برابر با زمان طول کشیده بر حسب ثانیه است .