بهترین پرواز

پروژه اول درس مبانی و کاربردهای هوش مصنوعی

استاد درس:

دکتر حسین کارشناس

اعضای گروه:

یسنا طالبی – 4003613045

ایلیا شعبان پورفولادی – 4003613041

Best Flight Table of Contents

Creating the Graph of Flights 3
- Dijkstra Graph 3
- A* Graph 3

Dijkstra Algorithm 4
- Calling the function : 4
- Initial Data Structures : 4
- Initialization of Graph Vertices : 5
- Initialization of the Children of Source Airport : 5
- Finding the Nearest Unexplored Airport : 6
- Updating the Tag of the Children of Chosen Node : 6
- Returning the Path : 7

A* Algorithm 8
- Heuristic Function : 8
- Using Heuristic Function in A* : 9

Dijkstra Algorithm vs. A* Algorithm 10
- Difference in the Search Environment : 10
- Difference in Execution Time : 12

Refrences 13

Used Libraries 13

هدف در این سؤال پیاده‌سازی برنامه‌ای برای پیدا کردن بهترین مسیر پروازی از نقطه مبدأ به نقطه مقصد است.

Creating the Graph of Flights

در این پروژه، ما برای ساخت گراف از ساختمان داده‌ی dictionary استفاده کردیم.

Dijkstra Graph :

در این دیکشنری هر فرودگاه مبدا یک key است و فرودگاه هایی که از طریق فرودگاه مبدا به آن‌ها پرواز مستقیم وجود دارد و فاصله‌ی آن‌ها، به صورت یک دیکشنری، value در نظر گرفته شده‌اند.

```
Dijkstra_graph = {
   source_airport1 : {
      destination_airport1 : edge_weight,
      destination_airport2 : edge_weight,
      destination_airport3 : edge_weight,
      ...
   }
    source_airport2 : {
      destination_airport1 : edge_weight,
      destination_airport2 : edge_weight,
      destination_airport3 : edge_weight,
      ...
   }
   ...
}
```

A* Graph :

در گراف A* علاوه بر فاصله‌ی هر فرودگاه مقصد از فرودگاه مبدا آن، فاصله‌ی مستقیم فرودگاه‌های مقصد از فرودگاه هدف هم به وسیله‌ی تابع heuristic ، بدست آمده و در این گراف ذخیره می‌شود.

```
A*_graph = {
   source_airport1 : {
      destination_airport1 : [edge_weight, heuristic]
      destination_airport2 : [edge_weight, heuristic]
      destination_airport3 : [edge_weight, heuristic]
      ...
   }
    source_airport2 : {
      destination_airport1 : [edge_weight, heuristic]
      destination_airport2 : [edge_weight, heuristic]
      destination_airport3 : [edge_weight, heuristic]
      ...
   }
   ...
}
```

Dijkstra Algorithm

الگوریتم دایجسترا به عنوان یک الگوریتم مسیریابی کورکورانه، با درنظر گرفتن یک نود مبدا در یک گراف، فاصله‌ی کوتاه‌ترین مسیر از مبدا به تمام نودهای دیگر را بدست می‌آورد.

این الگوریتم با بررسی کردن تمام مسیرهای موجود، تضمین می‌کند که مسیر بدست آمده مسیر بهینه است.

Calling the function :

```
dijkstra_shortest_path = dijkstra(dijkstra_graph, src_airport, dest_airport)
```

تابع دایجسترا گراف پروازها، فرودگاه مبدا و فرودگاه هدف را به عنوان ورودی دریافت میکند و در خروجی کوتاه ترین مسیر را بصورت لیستی از فرودگاه‌های مبدا تا مقصد برمی‌گرداند.

Initial Data Structures :

```
dist = {}
prev = {}
explored = []
unexplored = {}
```

dist : دیکشنری از فرودگاه‌ها (vertices) و تگ‌های آن‌ها.

prev : دیکشنری از فرودگاه‌ها و والد آن‌ها.

explored : لیستی از فرودگاه‌های visit شده.

unexplored : دیکشنری از فرودگاه‌هایی که تگ آن‌ها مقدار دهی شده و می‌توانیم مسیر را از آن‌ها ادامه دهیم.

Initialization of Graph Vertices :

```
for source_airport in graph.keys():
    dist[source_airport] = float('inf')

for destination_list in graph.values():
    for destination in destination_list.keys():
        if destination not in dist.keys():
            dist[destination] = float('inf')

dist[src_airport] = 0
explored.append(src_airport)
```

در این دو حلقه، تگ تمام نودهای گراف (فرودگاه‌های مبدا و فرودگاه‌های مقصد) را توسط تابع float(‘inf’) ، $+\infty$ و تگ فرودگاه مبدا را مساوی صفر قرار می‌دهیم.

همچنین فرودگاه مبدا را به لیست explored اضافه می‌کنیم.

Initialization of the Children of Source Airport :

```
edges = graph[src_airport]
for destination in edges.keys():
    dist[destination] = edges[destination]
    unexplored[destination] = edges[destination]
    prev[destination] = src_airport
```

در این حلقه، تگ تمام فرزندان فرودگاه مبدا آپدیت شده و مقدار آن مساوی فاصله‌ی آن‌ها تا فرودگاه مبدا درنظر گرفته شده است. این نودها همچنین به لیست unexplored اضافه شده‌اند تا بتوانیم مسیر را توسط آن‌ها ادامه دهیم. در آخر فرودگاه مبدا به عنوان والد این نودها در دیکشنری prev قرار گرفته است.

Finding the Nearest Unexplored Airport :

```
while dest_airport not in explored:

    if len(unexplored) == 0:
        return None

    v_distance = min(unexplored.values())
    v = list(unexplored.keys())
        [list(unexplored.values()).index(v_distance)]

    if v not in graph.keys():
        unexplored.pop(v)
        continue

    explored.append(v)
    unexplored.pop(v)
```

در اینجا حلقه‌ی while تا وقتی که فرودگاه هدف در لیست explored قرار نگرفته (پیدا نشده) ادامه خواهد داشت و اگر فرودگاه دیگری برای بررسی کردن نداشتیم (دیکشنری unexplored خالی بود) به این معناست که هدف پیدا نشده و تابع دایجسترا None برمی‌گرداند.

درغیر این صورت، از میان نودهای موجود در دیکشنری unexplored ، فاصله‌ی نزدیک‌ترین نود تا نود مبدا را توسط تابع min() انتخاب می‌شود و سپس نام فرودگاه را نیز توسط فاصله‌ی بدست آمده، از دیکشنری unexplored استخراج می‌کنیم.

اگر از فرودگاه انتخاب شده به سمت فرودگاه دیگری پروازی وجود نداشت، آن را از دیکشنری unexplored خارج و با تکرار حلقه نود دیگری انتخاب می‌کنیم و در غیر این صورت فرودگاه انتخاب شده را به لیست explored اضافه می‌کنیم و از دیکشنری unexplored حذف می‌کنیم.

Updating the Tag of the Children of Chosen Node :

```
edges = graph[v]
for w in edges.keys():
    if w not in explored:
        if dist[v] + edges[w] < dist[w]:
            dist[w] = dist[v] + edges[w]
            unexplored[w] = dist[w]
            prev[w] = v
```

با بدست آوردن نودهای متصل(w) به نود انتخاب شده (v) (اگر w قبلا visit نشده بود) بررسی می‌کنیم که جمع تگ v و فاصله‌ی v از w (وزن یال بین v و w) از تگ فعلی w کوچک تر است یا نه. اگر شرط برقرار نبود تگ w تغییری نمی‌کند و اگر برقرار بود، تگ w مساوی با جمع تگ v و فاصله‌ی v از w قرار می‌گیرد و این نود به دیکشنری unexplored اضافه می‌شود (یا مقدار آن در unexplored آپدیت می‌شود). نود v همچنین به عنوان والد نود w ، در دیکشنری prev اضافه می‌شود.

Returning the Path :

در نهایت وقتی فرودگاه هدف پیدا شد، از تابع از حلقه‌ی while خارج می‌شود.

برای بدست آوردن مسیر بین فرودگاه مبدا و فرودگاه هدف، با استفاده از دیکشنری prev می‌توانیم والد هر نود را بدست آوریم. به این شکل از فرودگاه هدف شروع می‌کنیم و تا رسیدن به فرودگاه مبدا والد هر نود را به لیست path اضافه می‌کنیم. در نهایت با معکوس کردن path کوتاه‌ترین مسیر بدست آمده از فرودگاه مبدا به فرودگاه هدف را برمی‌گردانیم.

```
path = []
destination = dest_airport
path.append(destination)
while 1:
    path.append(prev[destination])
    destination = prev[destination]
    if src_airport in path:
        break
path.reverse()
return path
```

A* Algorithm

الگوریتم A* به عنوان یک الگوریتم مسیریابی آگاهانه، با در نظر گرفتن یک نود مبدا و یک نود هدف در یک گراف و بکارگیری یک تابع heuristic فاصله‌ی کوتاه ترین مسیر از مبدا به مقصد را بدست می‌آورد.

این الگوریتم با جهت دادن به جست و جو توسط تابع heuristic تضمین می‌کند که علاوه بر یافتن مسیر بهینه، هیچ الگوریتم دیگری نمی تواند با پیچیدگی زمانی کمتر (بررسی گره های کمتر قابل توجه) راه حل بهینه را پیدا کند.

Heuristic Function :

این تابع heuristic با توجه به طول و عرض جغرافیایی فرودگاه مبدا و فرودگاه هدف تخمینی از فاصله‌ی دو فرودگاه بدست می‌آورد.

با توجه به این که فاصله‌ی مستقیم دو فرودگاه همیشه کمتر یا مساوی با فاصله‌ی اصلی دو فرودگاه است پس این تابع heuristic یک تابع admissible و consistent است.

```
def distance(lat1, lat2, lon1, lon2):
    lon1 = radians(lon1)
    lon2 = radians(lon2)
    lat1 = radians(lat1)
    lat2 = radians(lat2)

    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat / 2) ** 2 + cos(lat1) *
        cos(lat2) * sin(dlon / 2) ** 2

    c = 2 * asin(sqrt(a))

    r = 6371

    return c * r
```

Using Heuristic Function in A* :

از آنجایی که تنها تفاوت الگوریتم A* با الگوریتم Dijkstra استفاده‌ی آن از تابع heuristic است، بنابراین نحوه‌ی اضافه شدن heuristic به الگوریتم را بررسی می‌کنیم.

```
edges = graph[src_airport]
for destination in edges.keys():
    dist[destination] = edges[destination][0]
    unexplored[destination] = edges[destination][0] + edges[destination][1]
    prev[destination] = src_airport
```

در اینجا برای مشخص کردن تگ فرودگاه‌های متصل به فرودگاه مبدا، تگ را مساوی با وزن یال آن‌ها قرار می‌دهیم اما در دیکشنری unexplored تگ هر فرودگاه را مساوی با جمع وزن یال و مقدار heuristic آن (فاصله‌ی مستقیم فرودگاه تا فرودگاه هدف) قرار می‌دهیم تا هربار که می‌خواهیم نزدیک‌ترین فرودگاه بعدی را انتخاب کنیم، اینکه این فرودگاه در جهت فرودگاه هدف است یا نه نیز تاثیرگذار باشد.

```
edges = graph[v]
for w in edges.keys():
    if w not in explored:
        if dist[v] + edges[w][0] < dist[w]:
            dist[w] = dist[v] + edges[w][0]
            unexplored[w] = dist[w] + edges[w][1]
            prev[w] = v
```

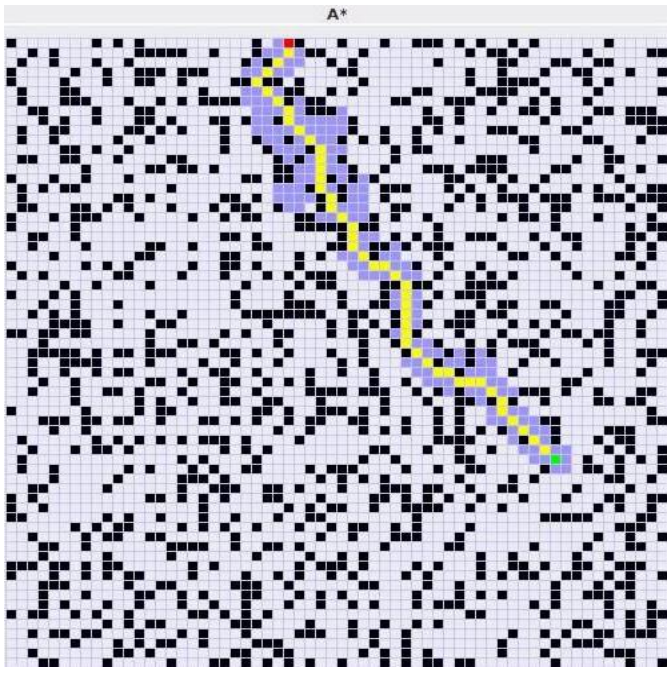
با بدست آوردن نودهای متصل(w) به نود انتخاب شده (v) (اگر w قبلا visit نشده بود) بررسی می‌کنیم که جمع تگ v و فاصله‌ی v از w (وزن یال بین v و w) از تگ فعلی w کوچک تر است یا نه. اگر شرط برقرار نبود تگ w تغییری نمی‌کند و اگر برقرار بود، تگ w مساوی با جمع تگ v و فاصله‌ی v از w قرار می‌گیرد و این نود به دیکشنری unexplored اضافه می‌شود (یا مقدار آن در unexplored آپدیت می‌شود). در اینجا مقداری که در دیکشنری unexplored قرار می‌گیرد تگ آن به علاوه‌ی مقدار heuristic آن است.

Dijkstra Algorithm vs. A* Algorithm

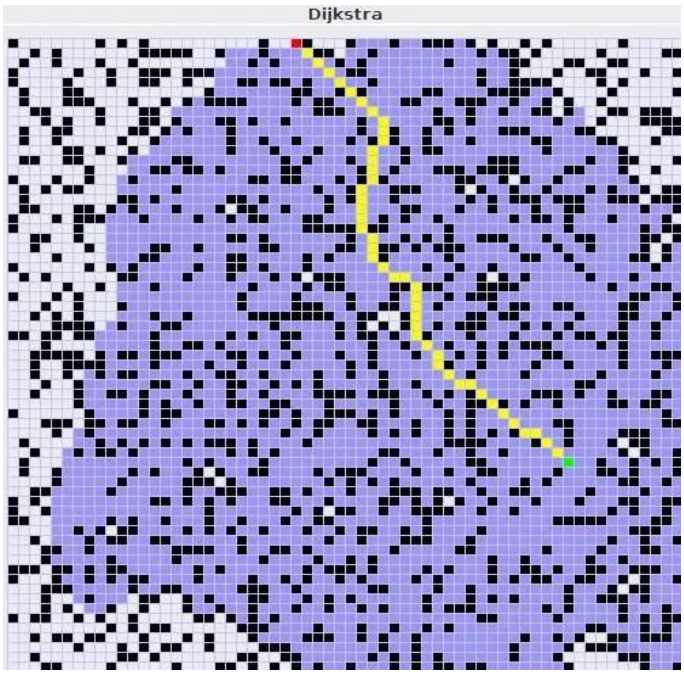
الگوریتم‌های Dijkstra و A* هر دو الگوریتم مسیریابی هستند که در علوم کامپیوتر استفاده می‌شوند. الگوریتم Dijkstra تضمین می‌کند که با بررسی تمام مسیرهای ممکن، مسیر بهینه را در یک گراف وزن‌دار پیدا کند. A* یک الگوریتم جستجوی آگاهانه است که از heuristic برای راهنمایی در جستجو استفاده می‌کند. انتخاب بین آنها بستگی به نیازهای خاص مسئله دارد. A* معمولاً زمانی ترجیح داده می‌شود که heuristic موجود و هدف بهینه‌سازی فرآیند جستجو است.

Difference in the Search Environment :

همان طور که در تصاویر زیر مشاهده می‌کنید، محیط جست و جوی Dijkstra به مراتب بسیار بزرگ‌تر از محیط جست و جوی A* می‌باشد. به این معنی که الگوریتم A* با داشتن گراف، نود مبدا و نود هدف یکسان با الگوریتم Dijkstra ، نودهای بسیار کمتری را explore می‌کند تا به هدف برسد.

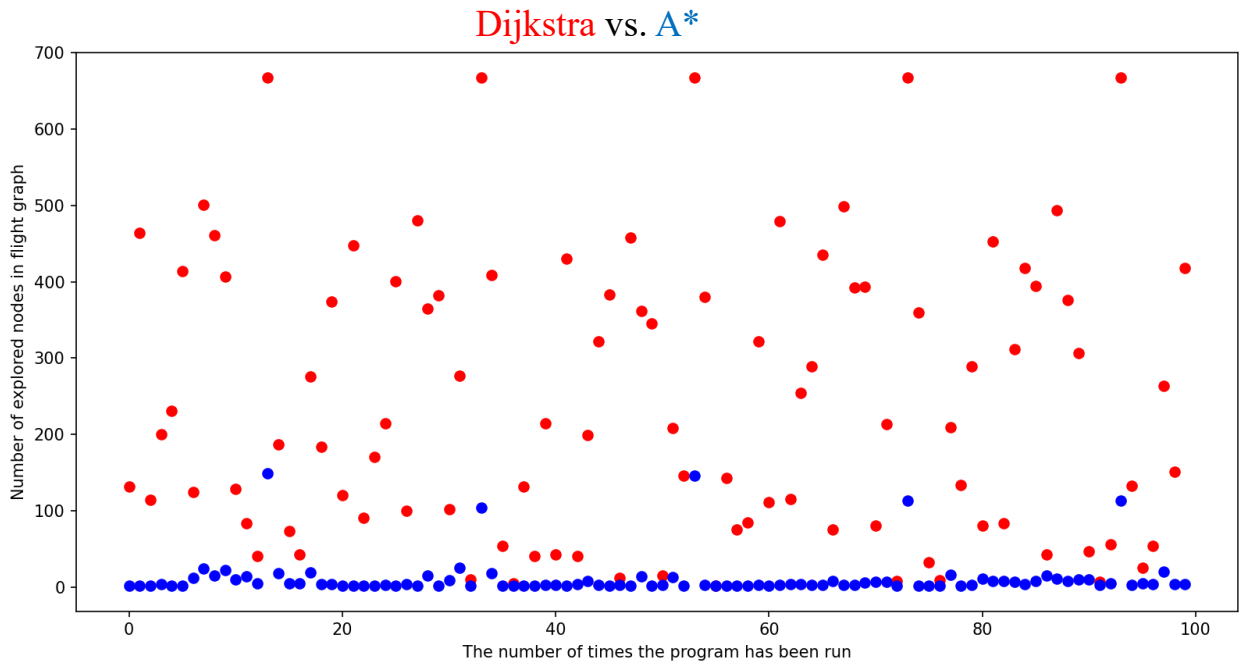


4.1. محیط جست‌وجوی الگوریتم A*



4.2. محیط جست‌وجوی الگوریتم Dijkstra

ما در کد خود نیز همین کار را تکرار کردیم و با دادن 100 فرودگاه مبدا و مقصد تصادفی به الگوریتم‌ها تعداد نودهای explore شده برای پیدا کردن هر مسیر را برای هر الگوریتم بدست آوردیم و با استفاده از کتابخانه‌ی matplotlib یک نمودار scatter برای این داده‌ها رسم کردیم.

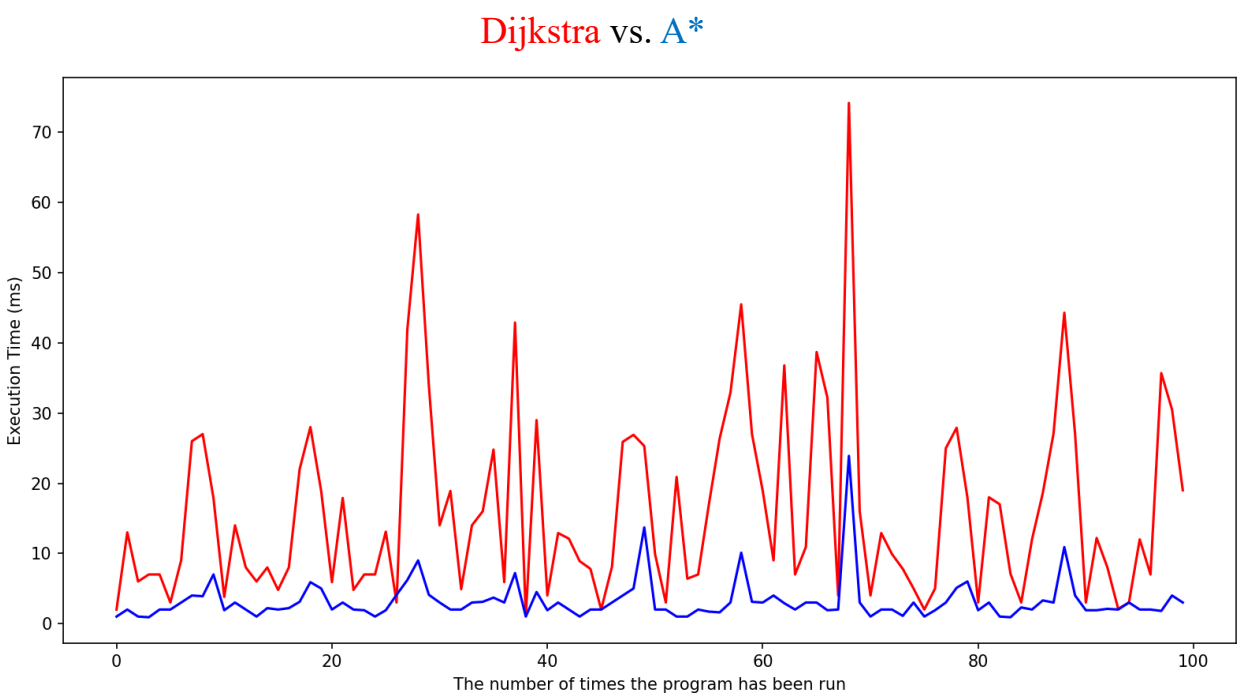


4.3. مقایسه الگوریتم‌های Dijkstra و A* از نظر تعداد نودهای explore شده برای پیدا کردن یک مسیر یکسان

همان طور که مشاهده می‌کنید تعداد نودهای explored در الگوریتم Dijkstra تا حدود 700 نود نیز می‌رسد در صورتی‌که در الگوریتم A* این عدد به مراتب کمتر است.

Difference in Execution Time :

برای مقایسه‌ی زمان اجرای دو الگوریتم، ما بار دیگر 100 فرودگاه مبدا و مقصد تصادفی به هردو الگوریتم دادیم و با استفاده از کتابخانه‌ی matplotlib یک نمودار plot برای این داده‌ها رسم کردیم.



4.4. مقایسه الگوریتم‌های Dijkstra و A* از نظر زمان اجرا برای پیدا کردن یک مسیر یکسان

همانطور که میبینید تضمین می‌کند که علاوه بر یافتن مسیر بهینه، هیچ الگوریتم دیگری نمی تواند با پیچیدگی زمانی کمتر (بررسی گره های کمتر قابل توجه) راه حل بهینه را پیدا کند.

Refrences

YouTube for Dijkstra (Spanning Tree Channel)

A* (A Star) Search Algorithm (Computerphile Channel)

Last Semester Design & Analysis of Algorithms Project (link)

tutorials Point

Geeks for Geeks

w3school

ChatGPT

Used Libraries

Pandas

Matplotlib

Time

Math

NumPy

Sys