Zahrasaffari/993613042

تسک اول : پارکینگ

در این تسک از الگوریتم بک ترکینگ استفاده شده است. در واقع در این رهیافت تحلیل مرتبه زمانی به طور دقیق مشخص نیست زیرا از نگاه کلی همان مرتبه زمانی نمایی بروت فورس را دارد اما در عمل بخاطر هرس شدن زمان کمتری مصرف می کند به همین دلیل در اجرای تمام تسک ها و توابع با ایمپورت کردن تابع تایم، زمان دقیق اجرا از لحظه گرفتن ورودی تا چاپ جواب را بر حسب میلی ثانیه محاسبه کرده و چاپ کردم (در این صورت دستیاران آموزشی راحت تر میتوانند مناسب بودن زمان اجرا را بسنجند.)به طور کلی برای این بخش دو سورس، یکی یونیت تست و دیگری تابع اصلی نوشته شد.

```
def promising(n, null, matrix):
    switch = False
    if null[0] - 2 >= 0 and matrix[null[0] - 1][null[1]] ==
matrix[null[0] - 2][null[1]]:
        switch = True
    if null[0] + 2 < n and matrix[null[0] + 1][null[1]] ==
matrix[null[0] + 2][null[1]]:
        switch = True
    if null[1] - 2 >= 0 and matrix[null[0]][null[1] - 1] ==
matrix[null[0]][null[1] - 2]:
        switch = True
    if null[1] + 2 < n and matrix[null[0]][null[1] + 1] ==
matrix[null[0]][null[1] + 2]:
        switch = True

    return switch
```

- چک میکند که در مراحل بعدی جایگزین هایی برای خانه خالی شده از پارکینگ وجود خواهد داشت به این صورت که عمودی و افقی بودن ماشین ها و نزدیک دیوار شدن آنها را در مراحل بعدی در نظر میگیرد و در غیر این صورت ادامه نمی دهد

```
if null[0] - 2 >= 0 and matrix[null[0] - 2][null[1]] not in order:
    if matrix[null[0] - 1][null[1]] == matrix[null[0] - 2][null[1]]:
        order.append(matrix[null[0] - 1][null[1]])
        matrix[null[0]][null[1]] = matrix[null[0] - 2][null[1]]
        null[0] -= 2
        matrix[null[0]][null[1]] = 0
        return parking(n, matrix, null, ask, order)
```

- اگر در مرحله بخشی امید بخش بودن آن ثابت شد، در این بخش در هر کدام از این چهار شرط(که حکم فرزندان این حالت را دارند) چک میشود که در بالا یا پایین هردو عدد متوالی برابر است یاخیر(که همان افقی و عمودی بودن ماشین های مجاور را نشان میدهد) ازاین طریق

حالت های مختلف پر شدن این خانه را(یعنی ازچپ راست بالا یا پایین)در نظر میگیرد و ادامه میدهد.

```
➢  if null == ask:
       print(order)
➢  /////////////////
➢  if not parking(n, matrix, null, ask, order):
        print("Not possible to empty this part of the parking lot")
➢
```

- و در نهایت زمانی که خانه خالی با خانه ای که درخواست تخلیه روی آن داده شده بود برابر شد، ترتیب جابه جایی ماشین ها که در آرایه اوردر ذخیره شده است را چاپ می کند و یا عدم وجود جواب تایید می شود.

تسک دوم : مسیریابی

قسمت اول رفتن به مقصد:

در این بخش از الگوریتم دایجسترا برای یافتن کوتاه ترین مسیر کمک گرفته شد و مرتبه زمانی آن در حالت کلی n^n است. از طرفی با در نظر گرفتن این خاسته از سوال که برای به یک مقصد معین، در صورت وجود بیشتر از یک مسیر بهینه، همه آنها باید چاپ شود، در هنگام برابری دو مقدار بهینه، به شرطی که به انتها (مکان مقصد) رسیده باشیم آنرا به تابع سولوشن میفرستن که از دست نرود و این روند موجب به دست آوردن تمام مینیمم مسیرهای برابر و بهینه به آن مقصد می شود. برای این بخش هم به طور جداگانه با ایمپورت کردن تابع تایم، زمان دقیق اجرا از لحظه گرفتن ورودی تا چاپ جواب را بر حسب میلی ثانیه محاسبه کرده و چاپ کردم.
برای این بخش هم دو سورس، یکی یونیت تست و دیگری تابع اصلی نوشته شد.

```
➢  parent = [-1] * row
   dist[src] = 0
   queue = []

   for i in range(row):
       queue.append(i)

   while queue:

       u = minDistance(dist, queue)

       queue.remove(u)
```

- در این بخش یک ارایه برای استخراج بازگشتی مسیر و در نتیجه چاپ کل مسیر، یک ارایه (dist) برای ذخیره سازی مینیمم فاصله هر مکان و یک صف برای کل مکان ها در نظر گرفته شده است که در هر مرحله نزدیک ترین را انتخاب و پاپ میکند.

```
if graph[u][i] and i in queue:
    if dist[u] + graph[u][i] <= dist[i]:
        if dist[u] + graph[u][i] == dist[i] and i == destination:
            printSolution(destination, dist, parent)
        dist[i] = dist[u] + graph[u][i]
        parent[i] = u
```

- در خط چهارم از این بخش از کد ، در صورت بخورد به دو مسیر بهینه با یک مبدا و مقصد و طول مشترک، تابع پرینت سولوشن را صدا میزنیم تا هر چند مسیر متفاوتی که وجود داشته باشد بتوانیم همه آنها با مراحل کامل چاپ کنیم.

قسمت دوم گذر در شهر:

در این بخش برای پیدا کردن بهترین راهی که بتوان طی کرد تا تعدادی مکان را دیدن کنیم و دوباره باز گردیم از الگوریتم TSP رهیافت بک ترکینگ و مرتبه زمانی به طور کلی(n!) استفاده شد و برای این بخش مانند همه توابع دیگر به دلیل اینکه تفاوت زمان اجرا در عمل از محاسبات به شدت کمتراست، زمان دقیق اجرا از لحظه گرفتن ورودی تا چاپ جواب را بر حسب میلی ثانیه محاسبه شده و چاپ شد و همچنین دو سورس، یکی برای یونیت تست و دیگری برای تابع اصلی پیاده سازی شد.

```
tsp(graph, v, source, n, 1, 0, [])

if len(answer) > 0:
    result = answer[0]
else:
    result = "It's impossible"
    print(result)
    return result
```

- در این بخش باصدا زدن تابع، مراحل جابه جایی بین مکان ها، در ارایه گلوبال answer ذخیره میشود، بنابراین طبیعتا اگر نال باشد ،یعنی مکان ها و جاده ها همبند نباشند، الگوریتم مقدار "ناممکن" را بر می گرداند.

```
print("Expected Answer:", s)
start_time = time.time()

if main(n, matrix, 0) == s:
    print("\nTEST ACCEPTED!")

else:
    print("TEST FAILED")

print("Algorithm execution time:", (time.time() - start_time) * 1000,
"ms")
```

- این قسمت بخشی از یونتیت تست مربوطه است که مقدار مورد انتظار و درست را در متغیر s ذخیره کرده و در چک شده که ایا خروجی تابع نوشته شده نیز با ورودی های فوق، با این مقدار برابر است یا خیر، که مذابق آن مقدارACCEPTED یا FAILED برمیگرداند.(محاسبه تایم زمان اجرا نیز در اینجا مشخص شده است)

تسک سوم: بازسازی جاده ها

در این تسک برای بازسازی جاده‌ها از الگوریتم پریم با مرتبه زمانی n^2 استفاده کردم زیرا انتظار می‌رفت تعداد مسیر های بین مکان ها زیاد باشد. همچنین زمان دقیق اجرا از لحظه گرفتن ورودی تا چاپ جواب را بر حسب میلی ثانیه محاسبه کردم و چاپ کردم و همچنین دو سورس، یکی برای یونیت تست و دیگری برای تابع اصلی پیاده سازی شد که برای مثال در تابع یونیت تست مربوطه یک الگوریتم جدید برای ساخت درخت تست ابدا شد به این صورت که ابتدا با تعداد n-1 یال که وزن همه آنها یک عدد رندوم در یک بازه معین است درخت ساخته میشود و سپس تعدادی یال اضافه به صورت رندوم با وزن های بیشتر از آن بازه قبلی، به درخت اضافه میشود. اکنون میدانیم الگوریتم ما باید مجموع طول آن یال های اولیه را بازگرداند تا صحت عملکرد آن معلوم شود.

```
for item in tree:
    x = np.random.randint(1, n * n)
    s += x
    matrix[ item[0] - 1 ][ item[1] - 1 ] = x
    matrix[ item[1] - 1 ][ item[0] - 1 ] = x
```

- در این بخش درخت ذکر شده (با طول جاده های رندوم بین۱ تا n^2 که این مقدار کاملا فرضی است)پر میشود.

```
for i in range(n):
    for j in range(n):
        if i != j and matrix[i][j] == 0 and np.random.randint(0,2):
            matrix[i][j] = np.random.randint(n * n + 1, n * n * n)
            matrix[j][i] = matrix[i][j]
```

- و در این بخش همان گونه که گفته شد تعدادی جاده اضافی با با طول بیشتر از آن مقدار فرضی به درخت اضافه میشود.
- Auxiliary link:

/https://www.geeksforgeeks.org/greedy-algorithms
/https://www.geeksforgeeks.org/backtracking-algorithms