

۱ نقشه خودسازمان دهنده

یک کلاس به اسم SOM برای پیاده سازی نقشه خودسازمان دهنده تعریف می کنیم. این کلاس به صورت یک شبکه عصبی تعریف می شود که $input_size$ ورودی و $output_size$ خروجی دارد و وزن های $w_{i,j}$ برای $1 \leq i \leq output_size$ و $1 \leq j \leq input_size$ روی آن تعریف شده اند.

۱.۱ محاسبه

خروجی i -ام این شبکه، بر اساس ورودی های $x_1, \dots, x_{input_size}$ و وزن های شبکه با تابع زیر محاسبه می شود.

$$y_i = \sqrt{\sum_{j=1}^{input_size} (x_j - w_{i,j})^2}.$$

که همان فاصله اقلیدسی بین دو بردار x و w_i می باشد.
از آنجا که فاصله اقلیدسی را در بخش های دیگری از کد نیز استفاده می کنیم، یک تابع کمکی برای محاسبه آن تعریف می کنیم.

```
def euclidean_distance(a, b):  
    return sum([(a[i] - b[i]) ** 2 for i in range(len(a))]) ** .5
```

حال می توانیم متد `compute` را برای این کلاس تعریف کنیم.

```
def compute(self, input):  
    return [  
        euclidean_distance(input, self.w[i])  
        for i in range(self.output_size)  
    ]
```

۲.۱ خودسازمان دادن

برای یادگیری در نقشه خودسازمان دهنده به این صورت عمل می کنیم که بعد از محاسبه خروجی ها روی ورودی ها، اگر کوچک ترین خروجی y_m باشد، بردارهای وزنی که فاصله اقلیدسی آن ها از بردار وزن w_m (بردار وزن های کوچک ترین خروجی که کلاستر انتخاب شده تعبیر می شود) در یک همسایگی کوچک قرار بگیرد را، کمی به بردار ورودی نزدیک می کنیم. این کار از طریق فرمول زیر ممکن می شود.

$$\Delta w_{i,j} = \begin{cases} \alpha(x_j - w_{i,j}) & \text{ed}(w_m, w_i) \leq \nu \\ 0 & \text{otherwise.} \end{cases}$$

که در آن ed همان فاصله اقلیدسی دو بردار است. توجه کنیم که در این فرمول از دو پارامتر α و ν استفاده کردیم. می‌توانیم α را درجه یادگیری و ν را آستانه همسایگی بنامیم که پارامترهایی مختص شبکه می‌باشند. حال می‌توانیم متد `self_optimize` را برای این کلاس تعریف کنیم.

```
def self_optimize(self, input, m):
    for i in range(self.output_size):
        if euclidean_distance(self.w[m], self.w[i]) <= self.nu:
            for j in range(self.input_size):
                self.w[i][j] += self.alpha * (input[j] - self.w[i][j])
```

۳.۱ مقداردهی اولیه

نهایتاً متد `__init__` را برای راه‌اندازی یک instance از این کلاس تعریف می‌کنیم.

```
def __init__(self, random_range, input_size, output_size, alpha, nu):
    self.input_size = input_size
    self.output_size = output_size
    self.alpha = alpha
    self.nu = nu
    self.w = [
        [randrange(*random_range[i]) for i in range(input_size)]
        for j in range(output_size)
    ]
```

که در آن وزن‌ها را در ابتدا به صورت تصادفی در بازه داده شده انتخاب می‌کنیم.

۲ خواندن مسئله و ترسیم آن

حال که شبکه مورد نظر را پیاده‌سازی کردیم. می‌توانیم مسئله را از فایل ورودی بخوانیم و آن را ترسیم کنیم. برای نمایش شهرها و جاده‌ها، کلاسی به اسم `Visualizer` پیاده‌سازی کردیم که کد کامل آن در فایل سورس `main.py` موجود است. یک صفحه 4×3 برای نمایش ۱۲ نقشه تعریف می‌کنیم.

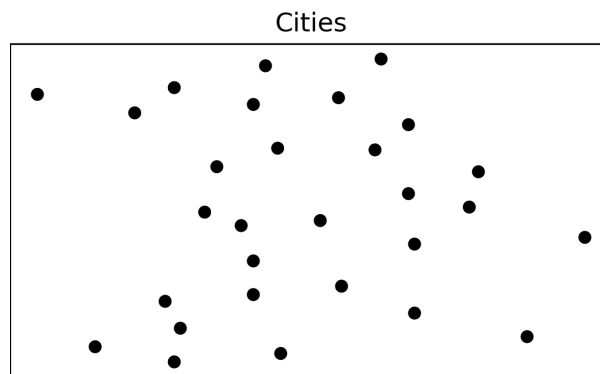
```
visualizer = Visualizer(3, 4)
```

شهر را از روی فایل ورودی می‌خوانیم.

```
with open(FILE + '.tsp') as f:
    cities = [{'position': tuple(map(float, l.split()[1:3])), 'cluster':
    None} for l in f.readlines()]
```

و نهایتاً نقشه خام شهرها را ترسیم می‌کنیم.

```
visualizer.add(cities, title='Cities')
```



۳ خوشه‌بندی

برای خوشه‌بندی به کمک کلاس SOM که تعریف کردیم، ابتدا یک instance از آن ایجاد می‌کنیم.

```
DIMENSION = 2
CLUSTERS_COUNT = 5
ALPHA = .4
NU = 9
RANDOM_RANGE = [
    [min(map(lambda x: x['position'][i], cities)), max(map(lambda x: x['position'][i], cities)) + 1]
    for i in range(DIMENSION)
]
som = SOM(RANDOM_RANGE, DIMENSION, CLUSTERS_COUNT, ALPHA, NU)
```

ثابت DIMENSION که تعداد ورودی‌های شبکه را مشخص می‌کند نشان‌دهنده این است که شهرها در فضای دوبعدی قرار دارند و لذا شامل دو مختصات می‌باشند. باقی پارامترها و ثابت‌ها به صورت تجربی انتخاب شده‌اند. و بازه‌ای که وزن‌ها را در ابتدا به صورت تصادفی از آن بازه انتخاب می‌کنیم همان بازه‌ای است که مختصات شهرها در آن قرار دارد.

حال در یک حلقه به خوشه‌بندی شهرها می‌پردازیم. هر بار تمام شهرها را خوشه‌بندی می‌کنیم و نقشه خوشه‌بندی شده را نصب می‌کنیم. سپس شهرها و خوشه‌های انتخابی را به som می‌دهیم تا از آن‌ها یاد بگیرد و خودش را سازمان‌دهی کند. در نهایت درجه یادگیری و آستانه همسایگی را اندکی کم می‌کنیم چرا که در ابتدا وزن‌ها تصادفی انتخاب شده‌اند و یادگیری باید زیاد باشد اما به مرور زمان باید کاهش یابد.

```
iteration = 0
while True:
    iteration += 1
```

```

# Cluster each city
for city in cities:
    som_output = som.compute(city['position'])
    city['cluster'] = som_output.index(min(som_output))

# Visualize the clustered cities
visualizer.add(cities, cluster_centers=som.w, title=f'Iteration {
iteration}')

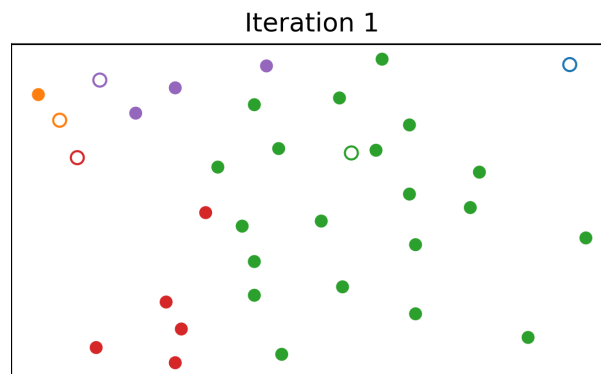
# Learn from each city
for city in cities:
    som.self_optimize(city['position'], city['cluster'])

# Lower the degree of learning (alpha)
# and the threshold of neighborhood (nu)
som.alpha -= .01
som.nu -= 1

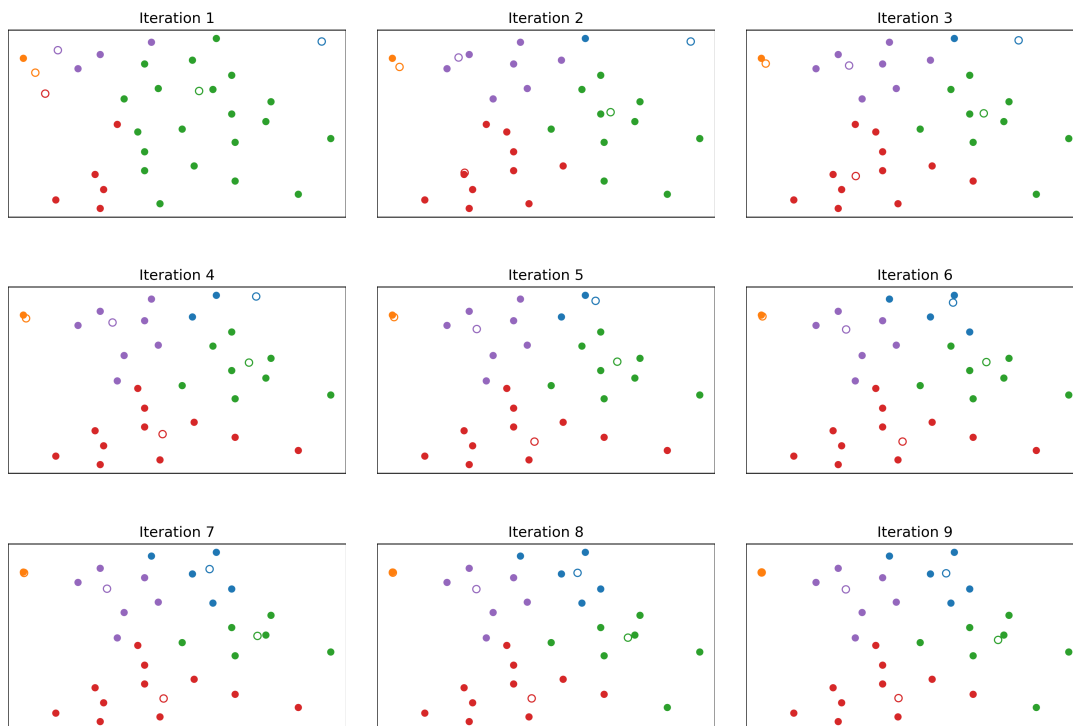
# Exit Condition
if som.alpha <= 0 or som.nu <= 0:
    break

```

برای مثال در تکرار اول خوشه‌بندی به صورت زیر خواهد بود.



در تصویر بالا، دایره‌های توپر نماینده شهرها و دایره‌های توخالی نماینده مرکز خوشه‌ها می‌باشند که بر اساس بردار وزن هر خروجی به دست می‌آیند. همه تکرارها را در زیر می‌بینیم.



در این مراحل، جابه‌جایی مراکز خوشه‌ها برای رسیدن به حالت بهتر قابل مشاهده می‌باشد.

۴ مسیریابی درون خوشه‌ای

حال که شهرها را خوشه‌بندی کردیم. می‌توانیم یک مسیر خوب درون هر خوشه پیدا کنیم و سپس خوشه‌ها را به یکدیگر متصل کنیم تا پاسخ خوبی برای مسئله فروشنده دوره‌گرد به دست آید. برای این منظور لازم است تا به لیست شهرهای هر خوشه دسترسی داشته باشیم پس ابتدا این لیست را برای هر خوشه تعریف می‌کنیم. (در این قسمت خوشه‌های خالی را در صورت وجود حذف می‌کنیم.)

```
clusters = list(filter(lambda x: x, [
    {'cities': [city for city in cities if city['cluster'] == i]}
    for i in range(CLUSTERS_COUNT)
]))
```

- سپس در یک حلقه برای هر یک از خوشه‌ها مراحل زیر را انجام می‌دهیم.
- اگر تعداد شهرهای خوشه کمتر از ۷ بود، تمام حالت‌های ممکن مسیر حداکثر $7! = ۷۲۰$ می‌باشد. تمام این حالات را بررسی می‌کنیم و بهترین مسیر را انتخاب می‌کنیم.
- در غیر این صورت، یکی از مسیرها را به صورت تصادفی در نظر می‌گیریم. (این بخش را می‌توان با الگوریتم‌های بهینه‌سازی دیگر ترکیب کرد تا پاسخ بهتری به دست آید.)
- مسیر به‌دست‌آمده و هزینه آن برای این خوشه را ذخیره می‌کنیم. همچنین دو سر انتهایی این مسیر را ذخیره می‌کنیم تا بعداً هنگامی که می‌خواهیم دو خوشه را به یکدیگر متصل کنیم از آن استفاده کنیم.

پیاده‌سازی این مراحل به صورت زیر خواهد شد.

```
for cluster in clusters:
    # If the cluster is small and it is possible, use brute-force and check
    # all permutations and find the best.
    if len(cluster) < 7:
        permutations = list(map(list, itertools.permutations(cluster['
cities'])))
        costs = [
            sum([euclidean_distance(p[j]['position'], p[j + 1]['position'])
for j in range(len(p) - 1)])
            for p in permutations
        ]
        best_cost = min(costs)
        best_permutation = permutations[costs.index(best_cost)]
    # Else, just use a random permutation. (This part can be better if we
    # use other optimization algorithms as a mix.)
    else:
        best_permutation = cluster['cities'].copy()
        shuffle(best_permutation)
        best_cost = sum([
            euclidean_distance(best_permutation[j]['position'],
best_permutation[j + 1]['position'])
            for j in range(len(best_permutation) - 1)
        ])

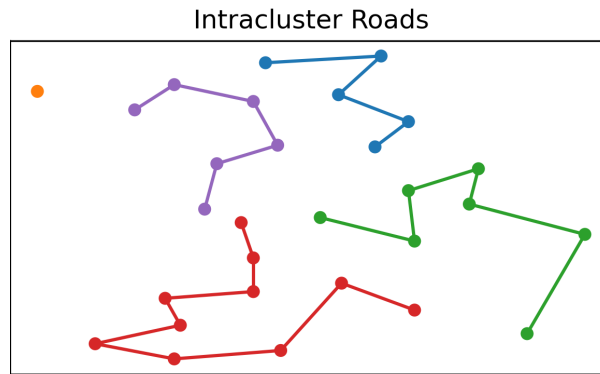
    # Add this path to roads array
    roads += [(best_permutation[j], best_permutation[j + 1]) for j in range
(len(best_permutation) - 1)]

    # Add two endpoints of the path to cluster data for connecting clusters
    cluster['ends'] = (best_permutation[0], best_permutation[-1])

    # Add the cost of this part
    cost += best_cost
```

نتیجه را ترسیم می‌کنیم

```
visualizer.add(cities, roads=roads, title=f'Intracluster Roads')
```



۵ مسیریابی بین خوشه‌ای

برای مسیریابی بین خوشه‌ای از آنجا که تنها ۵ خوشه داریم می‌توانیم تمام حالات را بررسی کنیم. برای ایجاد تمام حالات باید تمام جایگشت‌های خوشه‌ها را در نظر بگیریم اما چون روی یک دور قرار می‌گیرند جایگشت‌های دوری را یکی در نظر بگیریم همچنین جایگشتی مانند ۱, ۲, ۳, ۴, ۵ و ۵, ۴, ۳, ۲, ۱ نیز یکسان می‌باشند که آینه یکدیگر می‌باشند پس در کل $\frac{5!}{2} = 12$ جایگشت مختلف داریم. که به کمک توابع زیر این جایگشت‌ها را می‌سازیم.

```
def cmp(n):
    ''' Circular Mirrorable Permutations of numbers 1 to n '''
    if n == 3:
        return [[1, 2, 3]]

    ps = cmp(n - 1)
    r = []
    for p in ps:
        for i in range(n - 1):
            t = p.copy()
            t.insert(i, n)
            r.append(t)
    return r

def cm_permutations(l):
    n = len(l)
    return [[l[p[i] - 1] for i in range(n)] for p in cmp(n)]
```

اما یک نکته دیگر که باید به آن توجه کنیم این است که هر خوشه دو سر دارد که باید تصمیم بگیریم از کدام سر به خوشه بعدی متصل شود. اگر یک سر را صفر و سر دیگر را یک بنامیم برای هر خوشه دو حالت داریم که کدام سر آن به خوشه بعدی متصل می‌شود. اما سری که به خوشه قبلی متصل می‌شود به طور منحصربه‌فردی تعیین

می‌شود. (همان سری که به بعدی وصل نمی‌شود) پس به تعداد کدهای باینری به طول تعداد کلاسترها حالت مختلف برای وصل کردن داریم که برابر است با 2^5 و لذا کل حالات برابر $384 = 12 \times 2^5$ می‌باشد. برای ایجاد همه این حالات از کد زیر استفاده می‌کنیم.

```
options = [list(map(int, format(i, f'0{CLUSTERS_COUNT}b')) for i in range
(2 ** CLUSTERS_COUNT))]
```

حال می‌توانیم همه حالات را بررسی کنیم و بهترین حالت را انتخاب کنیم.

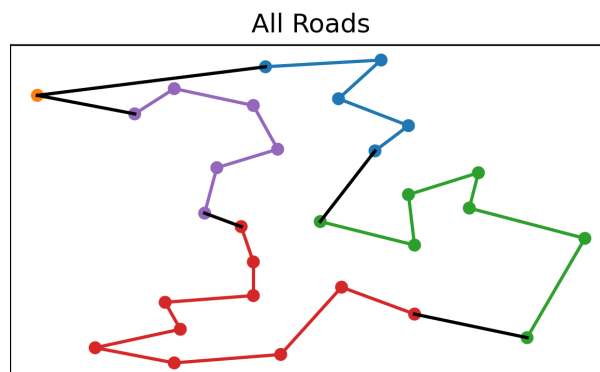
```
for p in permutations:
for option in options:
    po_cost = 0
    po_roads = []
    for j in range(len(p)):
        po_cost += euclidean_distance(p[j]['ends'][1 - option[j]]['position
'],
                                     p[j - 1]['ends'][option[j - 1]]['
position'])
        po_roads += [(p[j]['ends'][1 - option[j]], p[j - 1]['ends'][option[
j - 1]])]
    if intercluster_cost is None or po_cost < intercluster_cost:
        intercluster_cost = po_cost
        intercluster_roads = po_roads
```

نهایتاً جاده‌های بین‌خوشه‌ای را به لیست کل جاده‌ها اضافه می‌کنیم و هزینه آن را به کل هزینه‌ها و سپس نقشه جدید به همراه جاده‌های بین‌خوشه‌ای را رسم می‌کنیم.

```
roads += intercluster_roads
cost += intercluster_cost
```

```
visualizer.add(cities, roads=roads, title=f'All Roads')
```

که تصویر آن به صورت زیر خواهد شد.

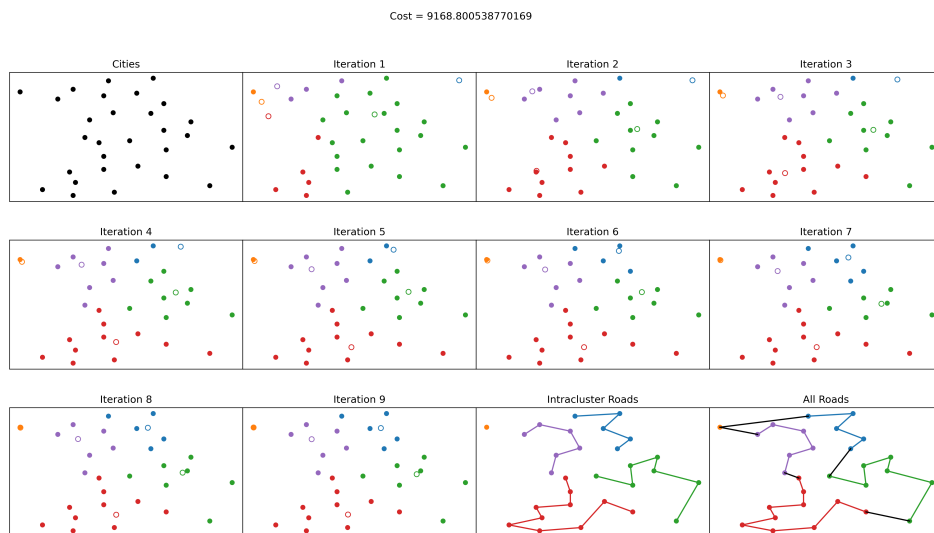


۶ پاسخ نهایی

در نهایت هزینه کل را چاپ می‌کنیم و تمام تصاویر را در کنار هم قرار می‌دهیم.

```
visualizer.fig.suptitle(f'Cost = {cost}')
```

```
visualizer.show()
```



همانطور که می‌بینیم هزینه کل ۹۱۶۸ به دست آمده و می‌دانیم بهترین پاسخ به این مسئله ۹۰۷۴ می‌باشد پس پاسخ قابل قبولی به دست آوردیم.