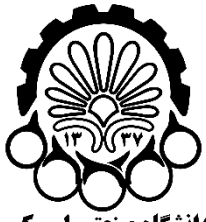
به نام خداوند بخشنده و مهربان



درس هوش مصنوعی

گزارش پروژه

پیاده‌سازی جستجوی محلی برای کمینه‌سازی مسائل مختلف با نمونه مسائل تصادفی

استاد درس:

دکتر قطعی

نام دانشجو:

امیررضا رادجو

۹۷۱۳۰۱۸

بهار ۱۴۰۰

در این گزارش به مقایسه دو الگوریتم تپه نوردی و ذوب فلزات میپردازیم و از مزایا و معایب هرکدام نام برده و آنها را در عمل با یکدیگر مقایسه میکنیم تا دریابیم در چه زمانی استفاده از کدام یک از این الگوریتم ها سود بیشتری به ما خواهد رساند

ابتدا لازم است با هر یک از الگوریتم ها به طور کلی آشنا شویم.

الگوریتم اول: الگوریتم جستجوی تپه نوردی

یک وضعیت دلخواه را به عنوان وضعیت فعلی در نظر میگیرد و سپس از میان همسایه های وضعیت فعلی بهترین همسایه را انتخاب مینماید

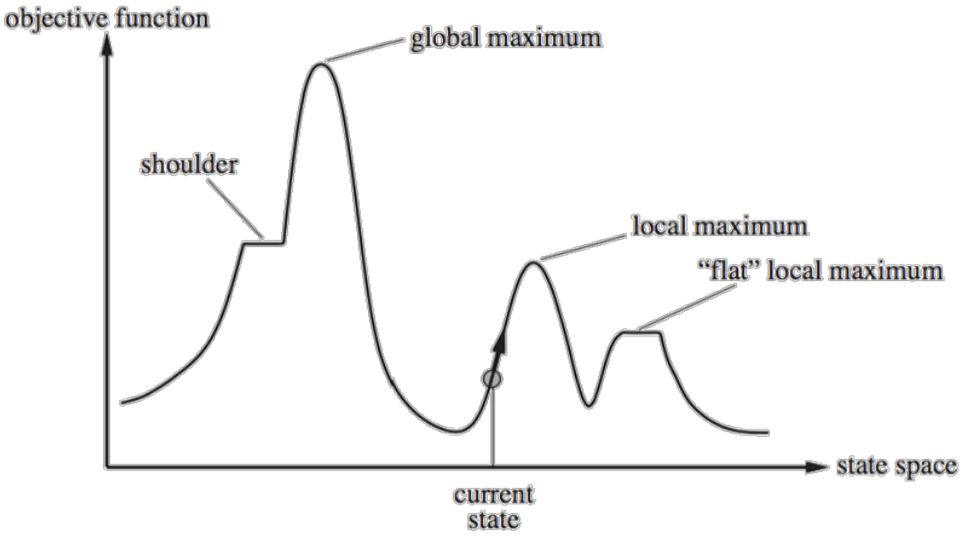
- مدام در جهت بهبودی اوضاع حرکت میکند
- زمانی متوقف میشود که هیچ همسایه بهتری موجود نباشد
- به جای ثبت حالت مسیر تنها کافیست گره مد نظر ثبت شود

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

از مشکلات جستجوی تپه نوردی میتوان به گیر کردن در ماکسیمم های محلی اشاره کرد.

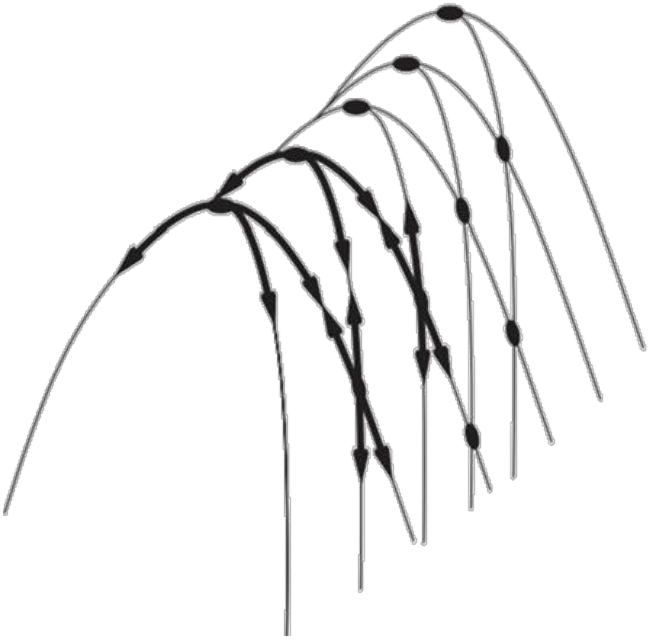
احتمال رخ دادن این حالت زمانی بیشتر است که نقطه تصادفی انتخابی در نزدیکی بیشینه های محلی قرار داشته باشد در این صورت الگوریتم به بیشینه محلی مورد نظر میرسد و آن را به عنوان جواب نهایی انتخاب میکند بنابرین میتوان بیان کرد که الگوریتم جستجوی تپه نوردی به نقطه تصادفی انتخابی وابسته است و نقاط تصادفی متفاوت امکان دادن جواب های متنوع را دارند بنابرین در یک مساله خاص با الگوریتم جستجوی تپه نوردی حالیتی به مانند تابع ثابت به وجود نخواهد آمد زیرا تابع جواب به نقطه تصادفی انتخابی وابسته است.

یکی دیگر از مواردی که برای الگوریتم جستجوی تپه نوردی مشکل ایجاد میکند فلات ها میباشد. فلات ها شامل بیشینه های محلی ثابت یا شانه ها میباشند که دارای مقداری ثابت در قسمت هایی از تابع هستند که باعث گیر کردن الگوریتم در این نقاط میشوند.



یکی از بزرگترین مشکلات برای تمامی الگوریتم های حریصانه دماغه ها میباشند.

دماغه ها در واقع تعدادی بیشینه محلی پشت سر هم هستند که گذشتن از آن ها برای الگوریتم های حریصانه کاری بسیار دشوار است و پیاده سازی های خاص و منحصر به فردی برای مواجهه با آنها نیاز است.



الگوریتم بعدی: الگوریتم ذوب فلزات

در این الگوریتم تلاش شده تا شانسی به وجود بیاید تا الگوریتم بتواند از ماکسیمم های مرکزی فرار کند. این الگوریتم در واقع ترکیبی از دو الگوریتم قدم زدن تصادفی و الگوریتم جستجوی تپه نوردی میباشد

در صورتی که الگوریتم قدم زدن تصادفی را در تعداد بسیار بالا انجام دهیم تقریبا شانس پیمایش تمامی نود ها برابر ۱ خواهد بود بنابرین این الگوریتم یک الگوریتم کامل حساب میشود. ولی در عین حال کارامدی پایینی دارد زیرا زمان بسیار زیادی نیاز است تا شانس تمامی نود ها به ۱ میل کند. در سوی دیگر الگوریتم جستجوی تپه نوردی الگوریتم کارامدی به شمار میرود با این تفاوت که تمامی نود ها شانس پیمایش شدن را ندارند و احتمال گیر کردن در ماکسیمم های نسبی به نسبت بسیار زیاد است

الگوریتم ذوب فلزات ترکیبی از این دو الگوریتم است که شانس رهایی از نقاط ماکسیمم نسبی را افزایش میدهد. روند الگوریتم به شکل زیر است

یک حالت را به عنوان حالت فعلی در نظر بگیر

هر بار یکی از همسایه ها را به شکل تصادفی انتخاب کن

اگر حالت انتخاب شده بهتر بود آن را جایچزین و در غیر این صورت به احتمال مشخصی آن را جایگزین کن.

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(*problem*.INITIAL-STATE)
  for $t = 1$ **to** ∞ **do**
    $T \leftarrow schedule(t)$
    if $T = 0$ **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow next.\text{VALUE} - current.\text{VALUE}$
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

حال برای مقایسه دو الگوریتم موجود آن ها را پیاده سازی کرده ایم و هربار با دیتاهایی خاص آن ها را مورد ارزیابی قرار میدهیم تا تفاوت آن ها را در عمل نیز مشاهده کنیم.

لینک الگوریتم های پیاده سازی شده:

https://github.com/amirradjou/HillTSP

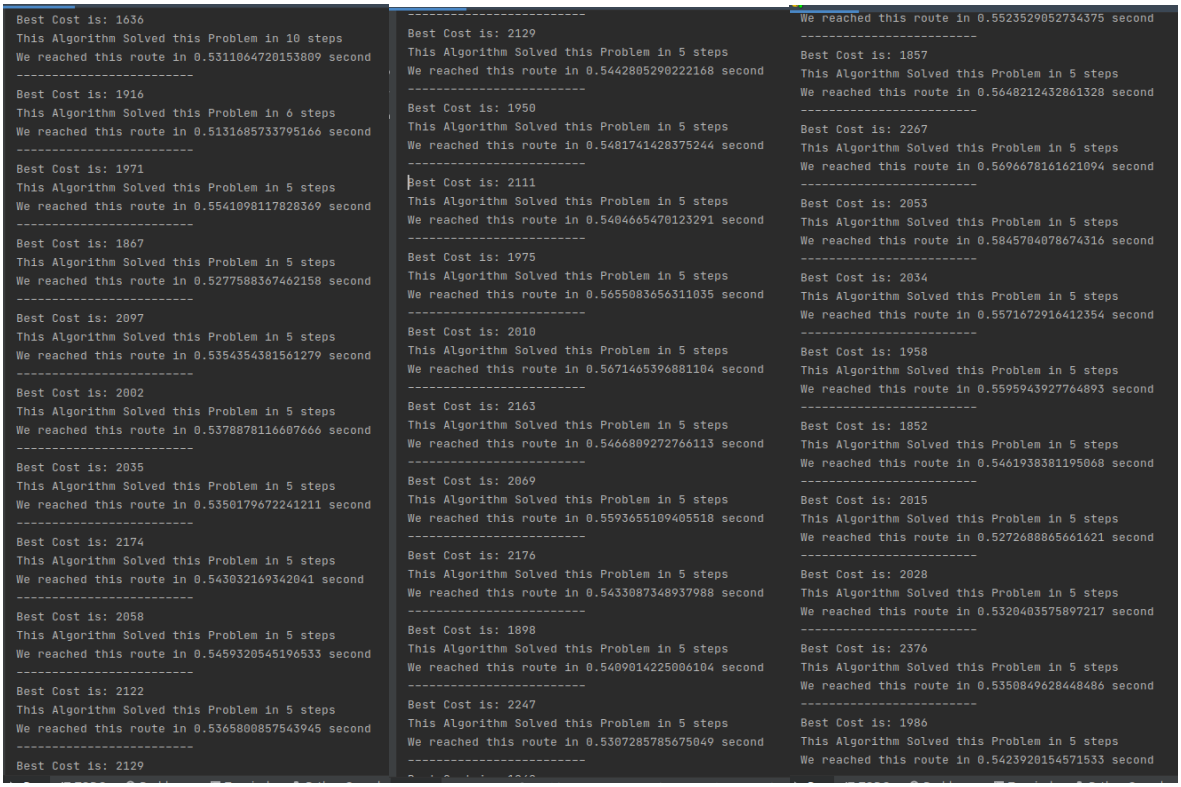
https://github.com/amirradjou/SimulatedAnnealingTSP

بخشی از کدها:

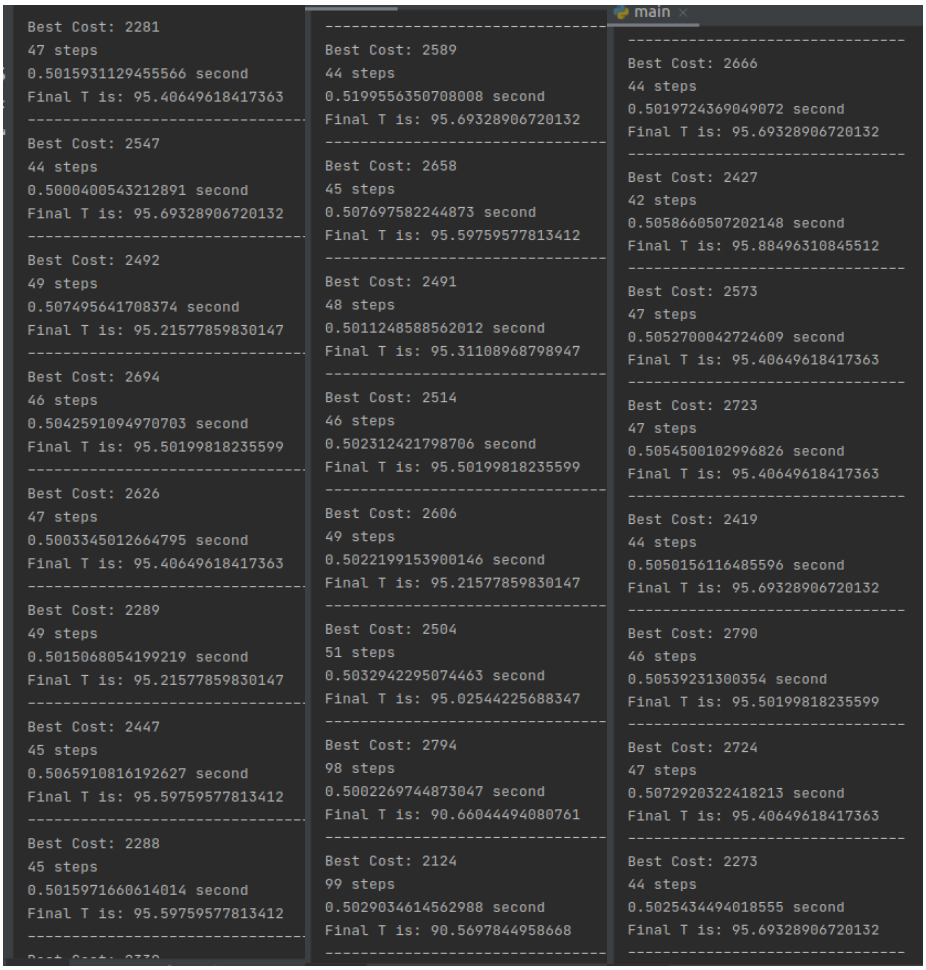
```
            new_neighbor = swapPositions(route, 1, j).copy()
            my_neighbors.append(new_neighbor)

    return my_neighbors


n = int(input("Please Insert the number of cities: "))

cities = generate_random_cities(n)
# for i in range(0, n):
#     print(cities[i])

random_route = generate_random_route(n)
# print(random_route)

cost = calculate_cost(random_route)
# print(cost)

# neighbors = generate_neighbors(random_route, n)
# print(neighbors)

best_route = random_route
steps = 0
tic = time.time()
while True:
    cost = calculate_cost(best_route)
    neighbors = generate_neighbors(best_route, n)
    best_cost = cost
    for route in neighbors:
        new_route_cost = calculate_cost(route)
        if new_route_cost < best_cost:
            best_route = route.copy()
            best_cost = new_route_cost
    steps += 1
    if best_cost == cost:
        break

toc = time.time()
sec = toc - tic
print("Best Route is: " + str(best_route))
print("Best Cost is: " + str(best_cost))
print(f"This Algorithm Solved this Problem in {steps} steps")
print(f"We reached this route in {sec} second")
```

```
def calculate_cost(route):
    cost = 0
    for i in range(0, n):
        city_A = route[i - 1]
        city_B = route[i]
        cost = cost + cities[city_A][city_B]
    return cost


def generate_random_route(number_of_cities):
    l = list(range(number_of_cities))
    random.shuffle(l)
    return l


def generate_random_cities(number_of_cities):
    cities = [[0 for i in range(number_of_cities)] for j
    for i in range(0, number_of_cities):
        for j in range(0, number_of_cities):
            if i > j:
                random_number = random.randint(1, 50)
                cities[i][j] = random_number
                cities[j][i] = random_number
    return cities


def generate_neighbors(route, n):
    my_neighbors = []
    for i in range(0, n - 1):
        for j in range(i + 1, n):
            new_neighbor = swapPositions(route, i, j).co
            my_neighbors.append(new_neighbor)

    return my_neighbors


n = int(input("Please Insert the number of cities: "))

cities = generate_random_cities(n)
# for i in range(0, n):
#     print(cities[i])

random_route = generate_random_route(n)
# print(random_route)
```

ابتدا برای ۱۰۰ شهر و با محدودین زمانی نیم ثانیه الگوریتم جستجوی تپه نوردی را انجام میدهیم:



حال با همین شرایط الگوریتم ذوب فلزات را پیاده سازی میکنیم:



مشاهده میکنیم تعداد مراحل انجام شده در الگوریتم ذوب فلزات چندین مرتبه بیشتر از الگوریتم تپه نوردی در زمان و تعداد شهر یکسان میباشد ولی در عین حال مسیر یافته شده توسط الگوریتم تپه نوردی با فاصله تقریبا کمی بهتر میباشد اما تفاوت چندانی دیده نمیشود.

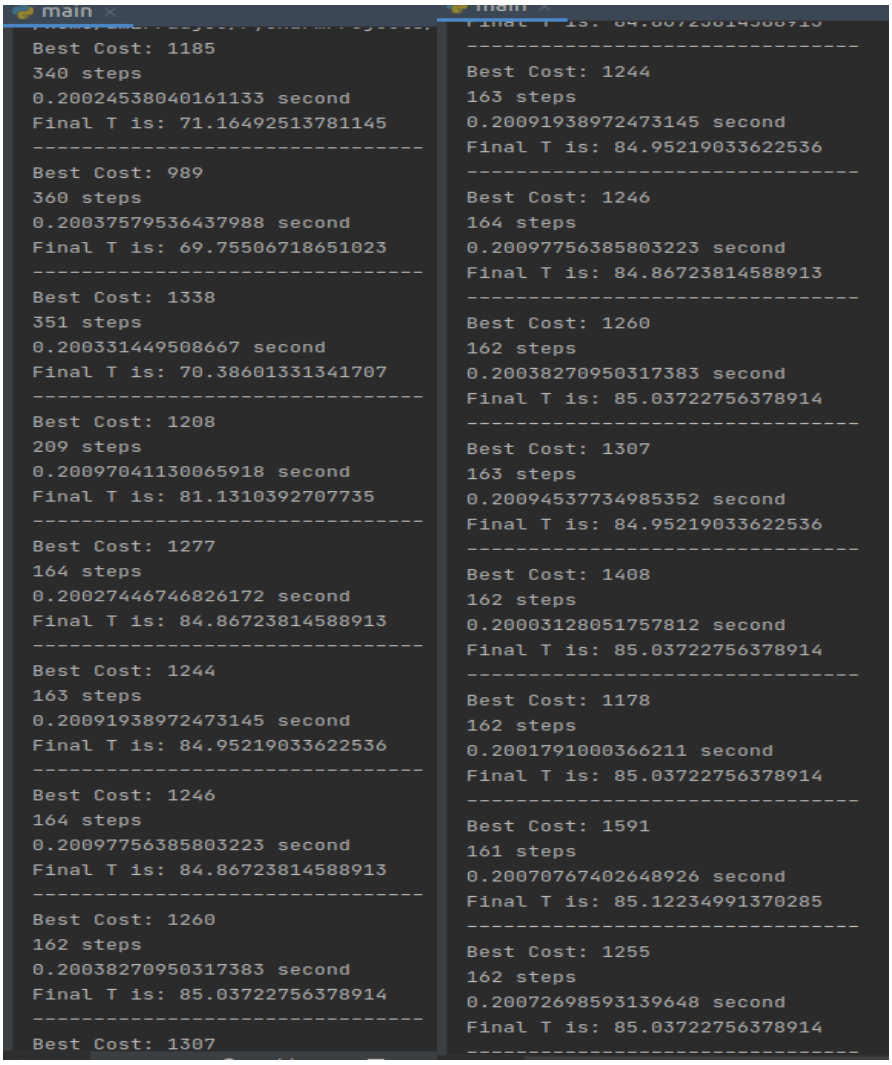
حال این الگوریتم ها را در زمان کمتر و با تعداد شهر کمتر تست میکنیم. تعداد شهر ها را به ۵۰ و مدت زمان را به ۲ دهم ثانیه کاهش میدهیم.

الگوریتم تپه نوردی:

```
Best Cost is: 320
This Algorithm Solved this Problem in 25 steps
We reached this route in 0.1546180248260498 secon
-------------------------
Best Cost is: 372
This Algorithm Solved this Problem in 21 steps
We reached this route in 0.1264963150024414 secon
-------------------------
Best Cost is: 437
This Algorithm Solved this Problem in 19 steps
We reached this route in 0.11559700965881348 seco
-------------------------
Best Cost is: 331
This Algorithm Solved this Problem in 20 steps
We reached this route in 0.12046408653259277 seco
-------------------------
Best Cost is: 369
This Algorithm Solved this Problem in 26 steps
We reached this route in 0.1568922996520996 secon
-------------------------
Best Cost is: 289
This Algorithm Solved this Problem in 30 steps
We reached this route in 0.1821727752685547 secon
-------------------------
Best Cost is: 328
This Algorithm Solved this Problem in 22 steps
We reached this route in 0.15849041938781738 seco
-------------------------
Best Cost is: 345
This Algorithm Solved this Problem in 27 steps
We reached this route in 0.16392135620117188 seco
-------------------------
Best Cost is: 326
This Algorithm Solved this Problem in 24 steps
We reached this route in 0.21031641960144043 seco
-------------------------
Best Cost is: 573
This Algorithm Solved this Problem in 16 steps
We reached this route in 0.21312713623046875 seco
-------------------------
```

```
Best Cost is: 506
This Algorithm Solved this Problem in 16 steps
We reached this route in 0.21324968338012695 second
-------------------------
Best Cost is: 454
This Algorithm Solved this Problem in 15 steps
We reached this route in 0.20244836807250977 second
-------------------------
Best Cost is: 436
This Algorithm Solved this Problem in 15 steps
We reached this route in 0.20058345794677734 second
-------------------------
Best Cost is: 441
This Algorithm Solved this Problem in 15 steps
We reached this route in 0.2039649486541748 second
-------------------------
Best Cost is: 446
This Algorithm Solved this Problem in 14 steps
We reached this route in 0.20929360389709473 second
-------------------------
Best Cost is: 459
This Algorithm Solved this Problem in 15 steps
We reached this route in 0.21041226387023926 second
-------------------------
Best Cost is: 407
This Algorithm Solved this Problem in 16 steps
We reached this route in 0.21251821517944336 second
-------------------------
Best Cost is: 422
This Algorithm Solved this Problem in 15 steps
We reached this route in 0.20260262489318848 second
-------------------------
Best Cost is: 483
This Algorithm Solved this Problem in 15 steps
We reached this route in 0.20811128616333008 second
-------------------------
Best Cost is: 380
This Algorithm Solved this Problem in 15 steps
We reached this route in 0.2008826732635498 second
-------------------------
```

الگوریتم ذوب فلزات:



مشاهده میشود که در این حالت تفاوت در بهترین مسیر انتخابی بسیار قابل توجه خواهد شد بنابرین میتوان نتیجه گرفت که الگوریتم تپه نوردی با اینکه نود های کمتری را پیمایش میکند اما در زمان و تعداد شهر کمتر بهتر از الگوریتم ذوب فلزات عمل میکند

این در حالیست که در صورت عدم وجود محدودیت زمانی الگوریتم ذوب فلزات میتواند مسیر های بهتری را پیدا کند زیرا شانس عبور آن از ماکسیمم های نسبی بیشتر است.

References:

https://towardsdatascience.com/how-to-implement-the-hill-climbing-algorithm-in-python-1c65c29469de

جزوه هوش مصنوعی دکترفاطمه موسوی