دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گزارش ۳ درس هوش مصنوعی

پیاده‌سازی جستجوی محلی برای کمینه سازی مسائل مختلف با نمونه مسائل تصادفی

به قلم:
امیر بابامحمودی

استاد
دکتر مهدی قطعی

فروردین ۱۴۰۰

مقدمه:

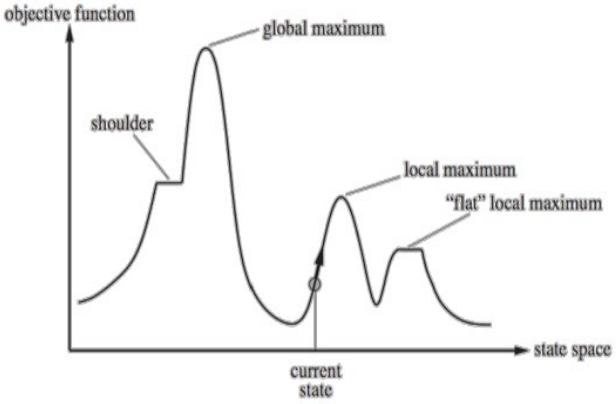
در دنیای امروزه مسائلی وجود دارند که در آن ها ما به دنبال یک راه حل کامل و ترسیم مسیر کامل از چگونگی رسیدن از یک نقطه اولیه به نقطه ی هدف نیستیم. بلکه تنها جواب نهایی و حالت هدف میباشد که برای ما مهم است. یک مثال بارز برای آن مسئله ی ملکه های شطرنج (n queens) میباشد که در آن تنها دنبال یک چینش برای ملکه ها هستیم که هیچ یک از دو به دوی آن ها با هم برخوردی نداشته باشند.
روش ها و الگوریتم هایی تحت این نوع مسائل وجود دارند که به آن ها جستجوی محلی(local search) گفته میشود که خود انواع مختلفی دارند که در این گزارش به تحلیل و مقایسه دو تا از پر استفاده ترین این الگوریتم ها یعنی جستجوی تپه نوردی(Hill-climbing search) و شبیه سازی ذوب فلزات (simulated annealing) میپردازیم.
در الگوریتم های جستجوی محلی یک (یا چند) حالت رو به عنوان استیت شروعی در نظر گرفته و تنها به حالت های همسایه ی آن حرکت میکنیم . در واقع یک حالت فعلی را به عنوان جوابی از مساله پذیرفته و سعی در بهبود آن داریم.

مزایا:

۱ ـ از آنجایی که در این روش نیازی به ترسیم مسیر کامل برای رسیدن به جواب نداریم و تنها استیت نهایی رو میخواهیم استفاده از حافظه در این روش بسیار کم میباشد. ($O(c)$ که c یک عدد ثابت است).
۲ ـ این الگوریتم به ما کمک میکند که در مسائلی با فضای حالت بسیار بزرگ(بی نهایت) در زمان کم و با استفاده از حافظه کم به جواب معقولی برسیم.
۳ ـ قابل استفاده در مسائل بهینه سازی محض.
۴ ـ یافت بهترین حالت با توجه به شرایطی که برای تابع هدف میگذاریم.

بررسی دورنمای فضای حالت:

دورنما شامل استیت و ارتفاع(یا مقدار تابع هزینه در آن استیت و یا مقدار تابع هدف) میباشد.



اگر ارتفاع متناظر با هزینه باشد که هدف یافتن کمترین عمق بوده و اگر ارتفاع متناظر با تابع هدف باشد هدف یافت بلندترین قله میباشد

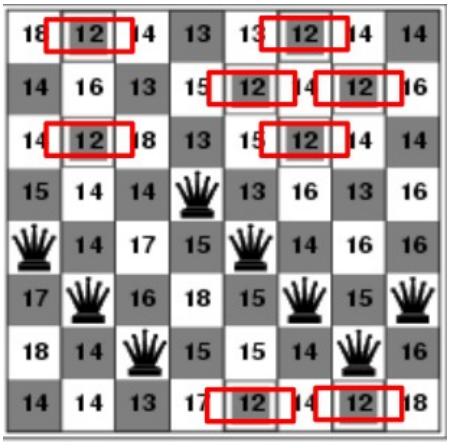
شکل 1)

جستجوی تپه نوردی(Hill-climbing search):

در این الگوریتم یک استیت را به عنوان استیت شروع در نظر گرفته و با جستجو در میان همسایه های آن بهترین همسایه ی آن را انتخاب میکنیم.

با توجه به شکل ۱ یعنی سعی بر این داریم که مدام به نوک تپه حرکت کنیم(فرض این است که ارتفاع مقدار تابع هدف است)

به مثال مساله ی n-وزیر توجه کنید :

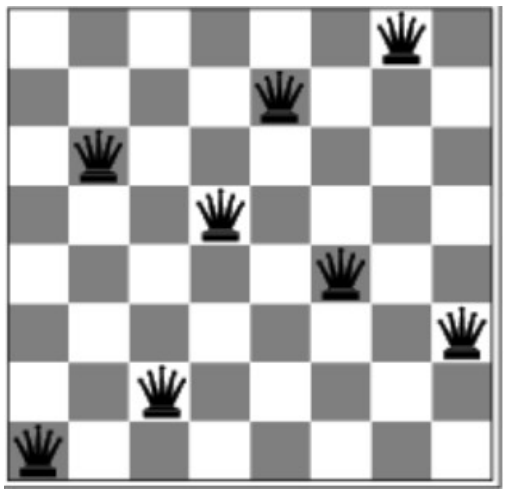


شکل2)یک استیت شروع
با h = 17

همانگونه که میبینید با حرکت دادن هر ملکه بصورت عمودی در ستون خود بهترین هزینه ای که میتوان برای همسایه ای در نظر گرفت h = 12 میباشد

پس از ۵ مرحله انجام جستجوی تپه ای :

همانگونه که میبینید h = 1 بوده که یعنی با شروع کردن با استیت نشان داده شده در شکل ۲ مسئله با روش جسجتجوی تپه ای موفقیت آمیز حل نشد.



شکل3)پس از ۵ بار یافتن
بهترین همسایه

یکی از مشکلات رایجی که در این روش وجود دارد همانگونه که دیدیم گیر کردن در یک بیشینه ی محلی میباشد.

از دیگر مشکلات این الگوریتم میتوان به فلات ها یعنی استیت هایی که همسایه ی بهتر از خود برای حرکت نداشته و در آن استیت گیر میکنند

شبیه سازی ذوب فلزات (simulated annealing):

در این روش بجای اینکه از استیت شروع به بهترین همسایه برویم بطور رندوم یک همسایه را انتخاب کرده اگر همسایه انتخاب شده بهتر از استیت فعلی بود به آن میرویم و اگر نه با احتمال $e^{\Delta E/T}$ به آن استیت میرویم.(ΔE اختلاف هزینه ی بین دو استیت بوده و T دما میباشد)
دلیل اینکه احتمالی میگذاریم که به استیتی با هزینه بیشتر برویم این است که ممکنه است با اینکار بتوان از بیشینه های محلی گریخت.

حال به شرح و تحلیل خروجی های کد n-وزیر میپردازیم :

اندازه ی مسئله	تعداد دفعاتی که در ۲۰ تکرار h = 0 شد(جواب بهینه)	میانگین تعداد بار هایی که به همسایه ی بعدی رفتیم	میانگین زمان رسیدن به جواب نهایی(ms)	میانگین h های نهایی(هزینه نهایی)
4	6	1.6	0	0.75
9	1	3.2	4.75	1.65
15	0	5.95	46.5	2

جدول1) ثبت نتایج پس از ۲۰ تکرار از الگوریتم جستجوی تپه ای

اندازه ی مسئله	تعداد دفعاتی که در ۲۰ تکرار h = 0 شد(جواب بهینه)	میانگین تعداد بار هایی که به همسایه ی بعدی رفتیم	میانگین زمان رسیدن به جواب نهایی(ms)	میانگین h های نهایی(هزینه نهایی)
4	20	144.9	1.8	0
9	20	14692.6	630.75	0
15	20	14994	1145	0

تحلیل : همانگونه که میشد پیش بینی هم کرد در الگوریتم جستجوی تپه ای تعداد بارهای زیادی در بیشینیه محلی افتاده و بخاطر همین به بهینه ترین نتیجه نرسیدیم که در الگوریتم ذوب فلزات میبینیم بخاطر پرش هایی که گاها به استیت هایی پر هزینه تر داریم از بیشینه های محلی فرار میتوانیم کنیم. ولی از نظر زمانی نیز میبینیم که الگوریتم جستجوی تپه ای در زمان بسیار کمتری به جواب بهینه ی خود میرسد.

کد مسئله ی n-وزیر:

```
import random
import math

def firstState(n):
    queensLoc = []
    for i in range(n):
        row = random.randint(0 , n-1)
        queensLoc.append(row)
    return queensLoc

def findNeighbors(queensLoc):
    neighbors = []
    for i in range(len(queensLoc)):
        for j in range(len(queensLoc)):
            temp = queensLoc.copy()
            temp[i] = j
            neighbors.append(temp)
    return neighbors
```

تابع اول بصورت رندوم وزیر هارو جاگذاری میکند

این تابع برای یافتن همسایه های یک استیت است

```
def stateCost(queensLoc):
    stateCost = 0
    for curColumn in range(len(queensLoc)):
        curRow = queensLoc[curColumn]
        for nextColumn in range(curColumn+1 , len(queensLoc)):
            nextRow = queensLoc[nextColumn]
            if nextRow == curRow :
                stateCost += 1
                continue
            if abs(nextColumn - curColumn) == abs(nextRow - curRow):
                stateCost += 1
                continue
        return stateCost

def findBestNeighbor(neighbors):
    bestNeighborCost = stateCost(neighbors[0])
    bestNeighbor = neighbors[0]
    for i in range(len(neighbors[0])):
        tempCost = stateCost(neighbors[i])
        if tempCost < bestNeighborCost:
            bestNeighborCost = tempCost
            bestNeighbor = neighbors[i]
    return bestNeighborCost , bestNeighbor
```

تابعی برای یافت هزینه هر استیت ـ تابعی برای پیدا کردن بهترین همسایه

```
def hillAlgorithm(n):
    counter = 0
    startState = firstState(n)
    startStateCost = stateCost(startState)
    neighbors = findNeighbors(startState)
    bestNeighborCost , bestNeighbor = findBestNeighbor(neighbors)

    while bestNeighborCost < startStateCost:
        counter += 1
        startState = bestNeighbor
        startStateCost = bestNeighborCost
        neighbors = findNeighbors(startState)
        bestNeighborCost , bestNeighbor = findBestNeighbor(neighbors)
    return startStateCost , startState , counter
```

الگوریتم جستجوی تپه ای :

```python
def simulatedAnnealingAlgorithm(n  , temp = 100):
    startState = firstState(n)
    startStateCost = stateCost( startState)
    iteration = 0
    for t in scheduleFunc(100 , 1.01):
        if iteration >= 1000 or t < 1e-6 :
            return startStateCost , startState
        iteration += 1
        neighbors = findNeighbors(startState)
        randomNeighbor = neighbors[random.randint(0 , len(neighbors) - 1)]
        randomNeighborCost = stateCost(randomNeighbor)

        delta_E = randomNeighborCost - startStateCost
        if delta_E < 0:
            startState = randomNeighbor
            startStateCost = randomNeighborCost
            continue
        else :
            p = math.exp((-delta_E)/t)
            randomFloat = random.random()
            if randomFloat <= p:
                startState = randomNeighbor
                startStateCost = randomNeighborCost
            continue
```

الگوریتم شبیه سازی فلزات

مسئله ی فروشنده ی دوره گرد:

در این مساله یک نقشه وجود دارد که در آن شهر ها و فواصل بین هر کدام از آن ها رو میدانیم . دنبال این هستیم که با رفتن کمترین مسیر ممکن بتوانیم همه ی شهر ها را دیده و به شهری که اول از آن شروع کردیم برگردیم .

طبق روش جستجوی محلی یک مسیر رندوم را به عنوان شروع گرفته و سعی در بهبود همان مسیر داریم.

حال به تحلیل خروجی میپردازیم :

تعداد شهر ها	میانگین زمان رسیدن به جواب بهینه(ms)	میانگین تعداد گره های تولید شده	میانگین بارهایی که این الگوریتم بهتر بود در ۱۰ گردش ۲۰ تایی
5	0.01	1.5	0
10	0.05	4.35	0.6
20	0.19	8.6	0

جدول3) ثبت نتایج پس از ۲۰ تکرار از الگوریتم جستجوی تپه ای

تعداد شهرها	میانگین زمان رسیدن به جواب بهینه(ms)	میانگین تعداد گره های تولید شده	میانگین بارهایی که این الگوریتم بهتر بود در ۱۰ گردش ۲۰ تایی
5	0.01	208.25	0.1
10	0.05	109.1	0
20	20.	73.65	3.8

جدول4) ثبت نتایج پس از ۲۰ تکرار از الگوریتم شبیه سازی ذوب فلزات

تحلیل:همان گونه که از جداول قابل مشاهده میباشد از نظر زمانی هر دو الگوریتم تقریبا در زمانی مشابه به جواب بهینه خود میرسند. تعداد گره های تشکیل شده در الگوریتم جستجوی تپه ای بسیار کمتر است که البته چون حافظه در این روش استفاده زیادی ندارد برتری خاصی بحساب نمیاید. و اما میبینیم که هر چی شهر ها گسترده تر شده الگوریتم ذوب فلزات جواب بهینه تری به ما میدهد . اما با تعداد کم شهر میبینیم که تقریبا عملکرد دو الگوریتم یکسان است.

کد فروشنده دوره گرد:

```
def generateRandomSolotion(citiesMatrix):
    cities_name = [*range(0 , len(citiesMatrix) , 1)]
    answer = []
    for i in range(len(citiesMatrix)):
        rCity = cities_name[random.randint(0 , len(cities_name) - 1)]
        answer.append(rCity)
        cities_name.remove(rCity)
    return answer


def stateCost(citiesMatrix , answer):
    stateCost = 0
    for i in range(len(answer)):
        stateCost += citiesMatrix[answer[i-1]][answer[i]]
    return stateCost

def neighborStates(answer):
    neighbors = []
    for i in range(len(answer)):
        for j in range(len(answer)):
            copy = answer.copy()
            copy[i] , copy[j] = copy[j] , copy[i]
            neighbors.append(copy)
    return neighbors
```

تولید یک مسیر رندوم ـ پیدا کردن هزینه یک مسیر خاص ـ پیدا کردن همسایه های یک استیت

```
def findBestNeighbor(citiesMatrix, neighbors):
    bestNeighborCost = stateCost(citiesMatrix , neighbors[0])
    bestNeighbor = neighbors[0]
    for i in range(len(neighbors)):
        temp = stateCost(citiesMatrix , neighbors[i])
        if temp < bestNeighborCost:
            bestNeighborCost = temp
            bestNeighbor = neighbors[i]
    return bestNeighborCost , bestNeighbor
```

پیدا کردن بهترین همسایه از نظر هزینه

```python
def hillAlgorithm(citiesMatrix):
    counter = 0
    startState = generateRandomSolotion(citiesMatrix)
    startStateCost = stateCost(citiesMatrix , startState)
    neighbors = neighborStates(startState)
    bestNeighborCost , bestNeighbor = findBestNeighbor(citiesMatrix , neighbors)

    while bestNeighborCost < startStateCost:
        counter += 1
        startState = bestNeighbor
        startStateCost = bestNeighborCost
        neighbors = neighborStates(startState)
        bestNeighborCost , bestNeighbor = findBestNeighbor(citiesMatrix , neighbors)

    return startStateCost , startState , counter
```

الگوریتم جستجوی تپه ای

```python
def simulatedAnnealingAlgorithm(citiesMatrix , schedule = scheduleFunc() , temp = 100):
    counter = 0
    startState = generateRandomSolotion(citiesMatrix)
    startStateCost = stateCost(citiesMatrix , startState)
    iteration = 0
    for t in scheduleFunc(100 , 1.01):
        if iteration >= 1000 or t < 1e-6 :
            return startStateCost , startState , counter
        iteration += 1
        neighbors = neighborStates(startState)
        randomNeighbor = neighbors[random.randint(0 , len(neighbors) - 1)]
        randomNeighborCost = stateCost(citiesMatrix, randomNeighbor)

        delta_E = randomNeighborCost - startStateCost
        if delta_E < 0:
            startState = randomNeighbor
            startStateCost = randomNeighborCost
            counter += 1
            continue
        else :
            p = math.exp((-delta_E)/t)
            randomFloat = random.random()
            if randomFloat <= p:
                startState = randomNeighbor
                startStateCost = randomNeighborCost
                counter += 1
            continue
```

الگوریتم شبیه سازی ذوب فلز

منابع:

۱ـ در یافتن تابع هزینه در مسئله ی n - وزیر از یکی از دوستانم نیما حسینی با شماره دانشجویی ۹۷۱۳۰۱۴ مشورت گرفتم

۲ـ Artificial Intelligence A Modern Approach, Third Edition, Stuart J. Russell and . Peter Norvig

۳ـ https://towardsdatascience.com/how-to-implement-the-hill-climbing-algorithm-in-python-1c65c29469de

۴ـ https://en.wikipedia.org/wiki/Travelling_salesman_problem

۵ـ لینک گیتهاب جهت دریافت کامل کد:
https://github.com/amirbabamahmoudi/AI-projects/tree/main/localsearch