CS454 AI Based Software Engineering

Coursework #1: Stochastic Optimization

20150751 최상진

1. Introduction

One of the classical NP-hard problems, the Traveling Salesman Problem (TSP) aims to find the shortest hamiltonian cycle in a graph. There might be some algorithms to find the optimal solution for small instances, but since NP-hard problems cannot be solved in polynomial time, none of them are feasible for large instance TSPs. Based on rl11849, a sample instance of cities from the TSPLIB library, our goal was to find the shortest hamiltonian cycle for 11849 cities. There are many optimization methods used to solve the TSP. After reviewing the course material and reading several articles related to solving TSP, I experimented with three optimization methods (Genetic Algorithm, Hill Climbing, and Simulated Annealing) and added some variations to those methods to optimize further. Genetic Algorithm did not went well, but the other two optimization methods scored one of the best scores in our COINSE Leaderboard. I would like to explain Simulated Annealing which is a stochastic optimization algorithm and analyze the results based on different constraints.

2. Stochastic Optimization Algorithm

A meta-heuristic algorithm has three key components: Representation, Operators, and Fitness.[1] Representation focuses on how the problem is represented and formulated. Operators decide how the next state will be different from the present state. Fitness evaluates the optimization level of the current status. Stochastic Optimization algorithm is basically the same as a meta-heuristic algorithm but it utilizes random variables. Simulated Annealing algorithm utilizes random variables in the Operator. I would start by introducing Hill Climbing algorithm which is the basis of the Simulated Annealing algorithm, and further

[1] cs454-slide02.pdf 25p

explain the Simulated Annealing algorithm with the variations that I made to optimize further.

Hill Climbing algorithm is the most basic optimization algorithm that aims to reduce the difference between the fitness of the current state and the fitness of the local optimal solution. The operator decides which neighboring state to move on based on the fitness evaluation result of the current state and the fitness evaluation result of the neighboring states. After Comparing the fitness evaluation results of all the neighboring states, the operator chooses the best neighboring state which has the highest fitness evaluation result. The algorithm ends when the fitness evaluation results of all of the neighboring states are lower than that of the current state. The Hill Climbing method optimizes the solution in a fast speed compared to other optimization algorithms, but it has a high probability that the current state gets stuck in a local optimum.

Simulated Annealing is a stochastic optimization algorithm for approximating the global optimum. It is basically the same as the Hill Climbing algorithm, but the Simulated Annealing algorithm improves the disadvantages of the Hill Climbing algorithm by using an additional temperature constraint as described in the pseudo code below. The algorithm initially starts with a high temperature. As the algorithm proceeds, the temperature variable decreases at a certain rate. Until the temperature becomes lower than a certain end temperature, the operation and the fitness evaluation process continues.

```
temperature = temperature_start
While temperature > temperature_end:
     Update_optimal_solution()
     temperature *= (1 - cooling_rate)
```

When the temperature is high, the solution is unstable and can make random moves.[2] Thus the operator has a higher chance to pick a state that has a lower fitness evaluation result than the current fitness evaluation result in a high temperature. By focusing on

[2] cs454-slide04.pdf 19p

exploring rather than exploiting, this technique prevents the search from being stuck in a local optimum.

I further optimized the algorithms with several variations to the algorithm. Instead of choosing random neighbors for the next step, I looked up all existing pairs one by one for each step and evaluated the fitness. Instead of swapping two cities in a path, I reversed the path between the two cities. After all the pairs are looked up, I rotated the path so that the path could make more random variations that could lead to a better optimum in the long run. The below Pseudo code describes the variations that I made to the algorithm. I will further explain in detail in the real code.

```
for i in range(number_of_all_cities):
    for j in range(i + 1, number_of_all_cities):
        if (get_Acceptance_Prob(i, j) > random(0, 1)):
            reverse(i, j)
rotate_path()
```

3. The Submission Code: Simulated Annealing method

The code takes the following parameters as input : -f (maximum fitness evaluation), -t (starting temperature), -r (temperature cool rate), -h (help)

Ex) python3 tsp_solver.py rl11849.tsp -f 100000 -t 30 -r 0.01

The code is written in Python in an object oriented style. There are 3 classes: `City`, `CitiesList`, and `Path`. I will first explain the classes that I have implemented.

```python
class City:
    def __init__(self, x, y, num):
        self.x = x
        self.y = y
        self.num = num

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def getNum(self):
        return self.num

    def calculateDistance(self, city):
        return math.sqrt((self.getX() - city.getX())**2 + (self.getY() - city.getY())**2)
```

`City` is a class that contains the data of each city. It stores the x coordinate, y coordinate, and unique number of a city. `getX()` method returns the x coordinate of a city. `getY()` method returns the y coordinate of a city. `getNum()` method returns the unique number of a city. `calculateDistance(city)` method returns the distance of the two cities.

```
class CitiesList:
    def __init__(self):
        self.citiesList = []

    def addCity(self, city):
        self.citiesList.append(city)

    def getList(self):
        return self.citiesList

    def getCity(self, idx):
        return self.citiesList[idx]
```

`CitiesList` is a class that stores `City` class instances in a list. `addCity(city)` appends a `City` class instance in the list. `getList()` returns the current list. `getCity(idx)` returns the `City` class instance that is stored in citiesList[idx].

```
class Path:
    def __init__(self, citieslist, dimension, fit, temp, rate):
        self.citieslist = citieslist
        self.dimension = dimension
        self.path = []
        self.fit = fit
        self.temp = temp
        self.rate = rate
```

`Path` is a class that contains the path (permutation) of the cities in a list. It stores a `citiesList` class instance, the total number of cities named dimension, a list that stores the permutation of cities, maximum fitness evaluation named fit, temperature variable named temp, and the cooling rate variable named rate. There are lots of methods in this class, so I divided the methods in to several screenshots.

```
def addCity(self, city):
    self.path.append(city)

def getCity(self, idx):
    return self.path[idx]

def generateRandomPath(self):
    for i in range(0, self.dimension):
        self.path.append(self.citieslist.getCity(i))
    random.shuffle(self.citieslist.getList())

def getTotalDistance(self):
    distance = 0
    for i in range(0, self.dimension-1):
        distance += self.path[i].calculateDistance(self.path[i+1])
    distance += self.path[self.dimension-1].calculateDistance(self.path[0])
    return distance

def rotate(self, n):
    copy = self.path[:]
    self.path = copy[n:] + copy[:n]
```

`addCity(city)` appends the city to the path list. `getCity(idx)` returns the city that is stored in self.path[idx]. `generateRandomPath()` generates a random permutation of cities. `getTotalDistance()` returns the total distance of the path by using the method `calculateDistance` defined in the `City` class instance. `rotate(n)` rotates the path by first making a copy of the list and splitting in 2 and concatenating the 2 lists.

```
def getAcceptanceProb(self, city1, city2):
    self.fit -= 1
    beforeDistance = self.path[city1 - 1].calculateDistance(self.path[city1]) + self.path[city2].calculateDistance(self.path[city2+1])
    afterDistance = self.path[city1 - 1].calculateDistance(self.path[city2]) + self.path[city1].calculateDistance(self.path[city2+1])
    if (beforeDistance > afterDistance):
        return 1
    else:
        prob = math.exp((beforeDistance - afterDistance)/self.temp)
        return prob

def reverse(self, city1, city2):
    reversedPath = self.path[:]
    reversedPath[city1 : city2+1] = reversed(self.path[city1 : city2+1])
    self.path = reversedPath
```

`getAcceptanceProb(city1, city2)` is one of the most important methods in my code. This method evaluates the fitness and returns a probability that this operation (reversing the path between the two cities) will be chosen by the operator. First, it decreases the fitness evaluation number by 1. If `self.fit == 0`, then the total process stops and returns the result of the shortest path that is found until the current time. The `getAcceptanceProb` method then calculates the distance before the reverse operation and the distance after the reverse operation. Instead of calculating the total distance of the path, it only calculates the paths between two cities that have been changed by the reverse operation. If `afterDistance` is smaller than `beforeDistance`, the method returns 1. This is because when the resulting path of the reverse operation is better than before, the operator will always choose this path.

The essence of the Simulated Annealing algorithm occurs when the `beforeDistance` is smaller than `afterDistance`. When this case happens, the Hill Climbing algorithm will discard this operation and continue on to find another path. But by discarding these cases will lead to a high probability of getting stuck in a local optimum. In the case of Simulated Annealing algorithm, the operator decides whether or not to move on to this path based on the below probability. The `getAcceptanceProb` method returns this probability or 1 based on the comparison result of the before and after distance.

```
prob = math.exp(beforeDistance - afterDistance)/self.temp)
```

`reverse(city1, city2)` method reverses the path between two cities. This is how the path is changed from the current state to the next state.

`simulatedAnnealing()` method is the same as the pseudo code explained before. It checks whether or not `self.fit` equals 0 and returns when the condition is met. After all pairs are considered by the `getAcceptanceProb` method, the path is rotated by 5000 by the `rotate` method. `organize` method makes the path to a 1-based permutation.

```python
def stimulatedAnnealing(self):
    for i in range(1, dimension - 2):
        for j in range(i, dimension - 1):
            if (self.getAcceptanceProb(i , j) > random.random()):
                self.reverse(i , j)
            if (self.fit == 0):
                return
    self.rotate(5000)
    self.temp *= (1 - self.rate)

def organize(self):
    for i in range(dimension):
        if (self.path[i].getNum() == 1):
            start = i
            break
    copy = self.path[:]
    self.path = copy[start:] + copy[:start]
```

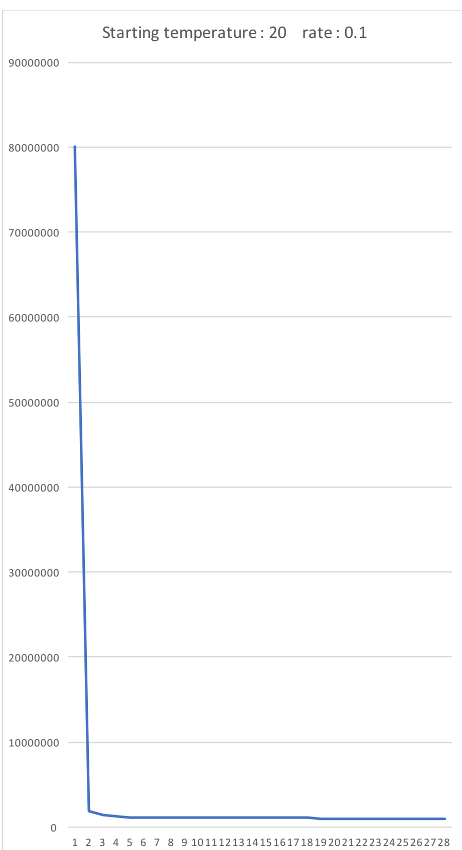
4. Computational Results

Experiment 1 : Starting temperature 10, rate 0.05

Total time: Approximately 7 hrs

Final Distance : 1035898

Each loop took approximately 15 ~ 20minutes.




```
1th distance : 1849066.3627424086
temp : 9.5
2th distance : 1300599.248730273
temp : 9.025
3th distance : 1154730.698687435
temp : 8.57375
4th distance : 1097164.9997389298
temp : 8.1450625
5th distance : 1073064.4832088565
temp : 7.737809374999999
6th distance : 1059829.0548638224
temp : 7.3509189062499996
7th distance : 1053722.5077710547
temp : 6.9833729609374995
8th distance : 1048884.8550632868
temp : 6.634204312890624
9th distance : 1044472.9134447357
temp : 6.302494097246092
10th distance : 1042518.0279642447
temp : 5.987369392383788
11th distance : 1040944.0848649429
temp : 5.688000922764598
12th distance : 1040020.2012997103
temp : 5.403600876626368
13th distance : 1039536.1078045139
temp : 5.133420832795049
14th distance : 1039202.9914972077
temp : 4.876749791155296
15th distance : 1038594.5777489026
temp : 4.632912301597531
16th distance : 1037787.3811621824
temp : 4.401266686517654
17th distance : 1036818.7324526681
temp : 4.181203352191771
18th distance : 1036342.2338357519
temp : 3.972143184582182
19th distance : 1036290.404445064
temp : 3.7735360253530725
20th distance : 1036315.7226753114
temp : 3.584859224085419
21th distance : 1036158.2044298967
temp : 3.4056162628811477
22th distance : 1036172.6961616682
temp : 3.2353354497370903
23th distance : 1036072.6255409928
temp : 3.0735686772502357
24th distance : 1035982.1847513977
temp : 2.919890243387724
25th distance : 1035985.8780115254
temp : 2.7738957312183374
26th distance : 1035908.3484415621
temp : 2.6352009446574205
27th distance : 1035916.4775322352
temp : 2.5034408974245492
28th distance : 1035898.3410874157
```

Experiment 2 : Starting temperature : 20 rate : 0.01

Total time: Approximately 10 hrs

Final distance: 999367

Each loop took 15 ~ 20 min

```
1th distance : 1894110.3013615874
temp : 19.8
2th distance : 1355037.7000401514
temp : 19.602
3th distance : 1196743.886074526
temp : 19.40598
4th distance : 1136149.3006447998
temp : 19.211920199999998
5th distance : 1105610.8789709888
temp : 19.019800997999997
6th distance : 1087764.8442100252
temp : 18.829602988019996
7th distance : 1076961.7012467333
temp : 18.641306958139797
8th distance : 1067106.1070061282
temp : 18.4548938885584
9th distance : 1060488.0942864993
temp : 18.270344949672815
10th distance : 1052937.6642679947
temp : 18.087641500176087
11th distance : 1049590.0404261732
temp : 17.906765085174325
12th distance : 1042823.6062563354
temp : 17.727697434322582
13th distance : 1039770.5513398111
temp : 17.550420459979357
14th distance : 1039359.3854983794
temp : 17.374916255379564
15th distance : 1034794.5653633497
temp : 17.201167092825767
16th distance : 1032774.7478625859
temp : 17.02915542189751
17th distance : 1029701.8416075737
temp : 16.858863867678537
18th distance : 1026627.3598174006
temp : 16.690275229001752
19th distance : 1026146.3883130077
temp : 16.523372476711735
20th distance : 1023996.2894004517
temp : 16.35813875194462
21th distance : 1021730.8792862063
temp : 16.194557364425172
22th distance : 1018538.0028222202
temp : 16.03261179078092
23th distance : 1017196.5120463464
temp : 15.87228567287311
24th distance : 1013851.7252607589
temp : 15.713562816144378
25th distance : 1014431.1225976136
temp : 15.556427187982935
26th distance : 1011092.333093063
temp : 15.400862916103106
27th distance : 1011846.2739712074
temp : 15.246854286942074
28th distance : 1010731.0527925988
temp : 15.094385744072653
29th distance : 1009130.7960571061
temp : 14.943441886631927
30th distance : 1008996.0063238429
temp : 14.794007467765608
31th distance : 1008627.8706489493
temp : 14.646067393087952
32th distance : 1007571.4017503399
temp : 14.499606719157072
33th distance : 1007227.0751630063
temp : 14.354610651965501
34th distance : 1005539.4288074
temp : 14.211064545445845
35th distance : 1004326.6142621419
temp : 14.068953899991387
36th distance : 1004173.4170781727
temp : 13.928264360991474
37th distance : 1004250.3248216867
temp : 13.788981717381558
38th distance : 1004295.7252079658
temp : 13.651091900207742
39th distance : 1001255.4385253672
temp : 13.514580981205665
40th distance : 1001938.0657652322
temp : 13.379435171393608
41th distance : 1001603.1342750468
temp : 13.245640819679672
42th distance : 1000286.7539957683
temp : 13.113184411482875
43th distance : 1000610.2065850905
temp : 12.982052567368045
44th distance : 1000481.3807681155
temp : 12.852232041694364
45th distance : 999236.2296202516
temp : 12.723709721277421
46th distance : 999367.5992787684
temp : 12.596472624064647
```

Experiment 3 : Starting temperature : 10 rate : 0.1

Total time : Approximately 10 hrs

Final distance : 1048672

Each loop took 15 ~ 20 min

```
1th distance : 1840553.4991014898
temp : 9.0
2th distance : 1295705.2641327165
temp : 8.1
3th distance : 1148033.0651437652
temp : 7.29
4th distance : 1095475.2695026302
temp : 6.561
5th distance : 1075783.7777470276
temp : 5.9049000000000005
6th distance : 1063511.8557579748
temp : 5.3144100000000005
7th distance : 1058476.5813198278
temp : 4.7829690000000005
8th distance : 1055124.439865178
temp : 4.3046721
9th distance : 1051747.0106940106
temp : 3.8742048900000006
10th distance : 1050404.617845658
temp : 3.4867844010000004
11th distance : 1049631.9937988315
temp : 3.1381059609000004
12th distance : 1049381.394826074
temp : 2.82429536481
13th distance : 1049251.2435379762
temp : 2.541865828329
14th distance : 1049106.3790754625
temp : 2.2876792454961
15th distance : 1049103.12548344
temp : 2.05891132094649
16th distance : 1049062.6046179987
temp : 1.853020188851841
17th distance : 1049063.4413361237
temp : 1.6677181699666568
18th distance : 1048951.8003755426
temp : 1.5009463529699911
19th distance : 1048869.414681076
temp : 1.350851717672992
20th distance : 1048821.4090404857
temp : 1.2157665459056928
21th distance : 1048724.8670300182
temp : 1.0941898913151236
22th distance : 1048720.7753316988
temp : 0.9847709021836112
23th distance : 1048689.4573491877
temp : 0.88629381196525
24th distance : 1048687.4860428877
temp : 0.7976644307687251
25th distance : 1048682.1351593472
temp : 0.7178979876918525
26th distance : 1048681.4954342234
temp : 0.6461081889226673
27th distance : 1048676.0930991098
temp : 0.5814973700304006
28th distance : 1048676.6829151402
temp : 0.5233476330273605
29th distance : 1048677.8665999537
temp : 0.47101286972462447
30th distance : 1048680.189349805
temp : 0.423911582752162
31th distance : 1048676.1587069777
temp : 0.38152042447694584
32th distance : 1048674.009353234
temp : 0.34336838202925124
33th distance : 1048673.6575662247
temp : 0.30903154382632614
34th distance : 1048675.2768221044
temp : 0.27812838944369356
35th distance : 1048673.4317984313
temp : 0.2503155504993242
36th distance : 1048673.0214986857
temp : 0.2252839954493918
37th distance : 1048672.2047509644
temp : 0.2027555959044526
38th distance : 1048672.5209381052
temp : 0.18248003631400736
39th distance : 1048672.5209381108
temp : 0.16423203268260664
40th distance : 1048672.2047509619
temp : 0.14780882941434598
41th distance : 1048672.5372458808
temp : 0.13302794647291138
42th distance : 1048672.2047509628
temp : 0.11972515182562025
43th distance : 1048672.2047509644
temp : 0.10775263664305823
```

5. Conclusion

After experimenting several times with various starting temperatures and rate, the one that produced the shortest path was when the constraints were starting temperature 30, and rate 0.01. The distance of that optimized path is 981,074.52, but I forgot to take a screenshot for that experiment. In that run, I set up the ending temperature as 15. Thus, I believe that the code can produce better optimal solutions that is shorter that 981,074.52, if I have enough time to run and experiment the code. Hill Climbing algorithm was much faster than the Simulated Annealing algorithm and for some cases in the process of Simulated Annealing algorithm, the total distance got longer than the previous step. But as the Simulated Annealing algorithm continued, the search did not got stuck in a local optimum. It produced a better optimal solution than the Hill Climbing algorithm. The most important concept that I learned doing this coursework is “more variation leads to a more optimal solution”.