Homework 2

February 6, 2018

0.1 IE6511 Homework 2

Done by: Aloisius Stephen and Yang Xiaozhou

```
In [1]: import numpy as np
        import pandas as pd
        import matplotlib
        import matplotlib.pyplot as plt
        %matplotlib inline

        np.random.seed(100)

        font = {'family' : 'sans-serif',
                'weight' : 'normal',
                'size'   : 12}

        matplotlib.rc('font', **font)
```

0.2 1. Homework on Genetic Algorithm

1. A binary string of length 4

2. Parent one before crossover: 0010, after crossover: 0010 Parent two before crossover: 0011, after crossover: 0011 Parent three before crossover: 0011, after crossover: 1011 Parent four before crossover: 1010, after crossover: 0010

3. Pair one children: 0010 and 0011 Pair two children: 1011 and 0010

4. After mutation, Pair one children: 0000 and 0001 Pair two children: 1001 and 0000

5. x and f(x) values of children: children one, x = 0, f(x) = 0 children two, x = 1, f(x) = 1 children three, x = 9, f(x) = 6561 children four, x = 0, f(x) = 0

   Total f(x) = 6561+ 1 = 6562

   Probability of being selected: children one, p = 0 children two, p = 1/6562 children three, p = 6561/6562 children four, p = 0

6. Binary strings of parents: Parent one: 0110 0010 1001 Parent two: 0001 0010 0011

   Children binary strings, x value (Crossover point 3): Children one: 0111 0010 0001, x = (7,2,9) Children two: 0000 0010 1011, x = (0,2,3)

1 Simulated Annealing

1.1 2. SA Parameter Selection when cost function range = (MaxCost and MinCost) are known

a)

$$
\begin{aligned}
avg\Delta cost &= 0.25(MaxCost - MinCost) \quad (1) \\
&= 25 \quad (2)
\end{aligned}
$$

$$
\begin{aligned}
T_0 &= -\frac{avg\Delta cost}{\ln P1} \quad (3) \\
& \quad (4)
\end{aligned}
$$

```
In [2]: P_1 = 0.4
        T_0 = -25/ np.log(P_1)
        print("T_0: %.4f" %T_0)
```

```
T_0: 27.2839
```

b)

$$
\begin{aligned}
T_0 &= -\frac{0.25(MaxCost - MinCost)}{\ln P1} \quad (5) \\
& \quad (6)
\end{aligned}
$$

c)

$$
\begin{aligned}
T_{final} &= -\frac{0.25(MaxCost - MinCost)}{\ln P2} \quad (7) \\
& \quad (8)
\end{aligned}
$$

d)

$$
\begin{aligned}
\alpha &= \left\{ -\frac{0.25(MaxCost - MinCost)}{T_0 \ln P2} \right\}^{\frac{1}{G}} \quad (9) \\
&= \left\{ -\frac{25}{100 \ln 0.001} \right\}^{\frac{1}{200}} \quad (10)
\end{aligned}
$$

```
In [3]: alpha = np.power(-25/(100*np.log(0.001)), 1/200)
        print("Alpha: %.4f" %alpha)
```

```
Alpha: 0.9835
```

e)

$$\alpha = \left\{ -\frac{0.25(MaxCost - MinCost)}{T_0 \ln P2} \right\}^{\frac{1}{G \div M}} \tag{11}$$

$$= \left\{ -\frac{25}{100 \ln 0.001} \right\}^{\frac{1}{200 \div 10}} \tag{12}$$

$$= \left\{ -\frac{25}{100 \ln 0.001} \right\}^{\frac{1}{20}} \tag{13}$$

```
In [4]: alpha = np.power(-25/(100*np.log(0.001)), 1/20)
        print("Alpha: %.4f" %alpha)

Alpha: 0.8471
```

1.2 3. SA Parameter Selection when you have computed AP cost values (no coding necessary)

```
In [5]: S_0 = 1
        Avg_Delta_Cost = 1/5 * np.sum([x-40 for x in [60, 50, 65, 75, 45]])
        P_1 = 0.9
        T_0 = -Avg_Delta_Cost/ np.log(P_1)
        print("T_0: %.4f" %T_0)

T_0: 180.3332
```

```
In [6]: Avg_Delta_Cost

Out[6]: 19.0
```

1.3 4. SA Implementation

```
In [7]: # cost function
        def cost(S):

            cost = np.power(10,9)-(625-np.power(S[0]-25, 2))*(1600-np.power(S[1]-10, 2))*np.si

            return cost


        # neighbor function
        def neighbor(S):

            neighbor = S
            pos = np.random.randint(0, 2) # randomly pick one of the two decision variables
            nei_value = S[pos]
            while nei_value == S[pos]:
```

```
        # randomly generate a neighbor value
        nei_value = np.random.randint(max(S[pos]-25, 0), min(S[pos]+25, 127)+1)
        pass
    neighbor[pos] = nei_value # form the neighbor

    return neighbor


# simulated annealing algorithm
def SA(S_initial, T_initial, alpha, beta, M, Max_time):

    solution = np.zeros([Max_time,3])

    T = T_initial
    CurS = S_initial
    BestS = CurS
    CurCost = cost(CurS)
    BestCost = CurCost
    time = 0

    while time < Max_time:

        for i in range(0, M):
            NewS = neighbor(CurS)
            NewCost = cost(NewS)
            diff_cost = NewCost - CurCost

            if diff_cost < 0:
                CurS = NewS
                CurCost = NewCost
                if NewCost < BestCost:
                    BestS = NewS
                    BestCost = NewCost
            elif np.random.random() < np.exp(-diff_cost/T):
                CurS = NewS
                CurCost = NewCost

            solution[time+i]=time+i+1, CurCost, BestCost

        time = time + M
        T = alpha*T
        M = beta*M

    solution = pd.DataFrame(solution, columns=['Iteration_Number', 'Current_Cost', 'Bes

    return (solution, BestS)
```

1.4 5. Running SA:

1.4.1 a)

```
In [21]: #np.random.seed(100)

         beta = 1
         G = 1000
         M = 1
         Max_time = 1100
         P_1 = 0.9
         P_2 = 0.05
         AP = 20


         def SAparameter(beta, G, M, Max_time, P_1, P_2, AP):

             # generate S1
             start_s = [np.random.randint(0,128), np.random.randint(0,128)]
             AP = AP - 1

             # evaluate neighbors of S1
             s_neighbor = pd.DataFrame([[0,0]], columns = ["s1", "s2"], index = range(AP))
             cost_ap = pd.DataFrame([[0]], columns = ["cost"], index = range(AP))

             # include start_s itself
             s_neighbor.iloc[0] = start_s
             cost_ap.iloc[0] = cost(start_s)

             for i in range(1, AP):
                 s_neighbor.iloc[i] = neighbor(start_s)
                 cost_ap.iloc[i] = cost(s_neighbor.iloc[i])

             neighbor_cost = s_neighbor.join(cost_ap)
             avg_delta_cost = np.sum((neighbor_cost.cost-min(neighbor_cost.cost)))/(AP-1)

             # base on AP parameter search, decide on the starting S
             S_0 = neighbor_cost.sort_values('cost').head(1).values.ravel()[:2]

             # calculate algorithm parameters
             T_0 = -avg_delta_cost/np.log(P_1)
             T_2 = -avg_delta_cost/np.log(P_2)
             alpha = np.power(np.log(P_1)/np.log(P_2), 1/G)

             param = [T_0,T_2,alpha, avg_delta_cost]

             return param

In [22]: param = SAparameter(beta, G, M, Max_time, P_1, P_2, AP)
```

```
print('T_0: %.5f' %param[0])
print('T_2: %.5f' %param[1])
print('alpha: %.5f' %param[2])
print('Avg_Delta_Cost: %.5f' %param[3])
```

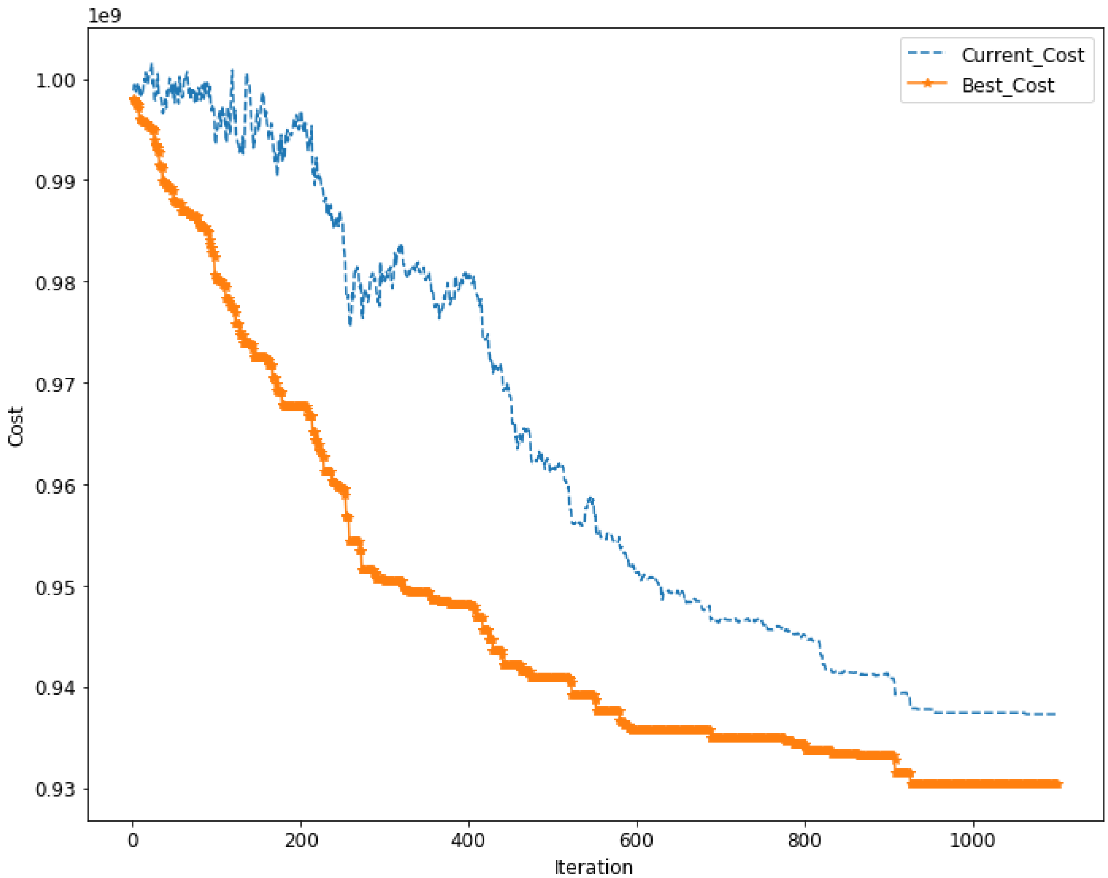
```
T_0: 29392957.69433
T_2: 1033756.32286
alpha: 0.99666
Avg_Delta_Cost: 3096857.17938
```

1.4.2 b)

In [23]:
```
T_0 = param[0]
alpha = param[2]

Z = [[np.random.randint(0,128), np.random.randint(0,128)] for i in range(30)]
```

In [24]:
```
import time
cpu_time = []

start_time = time.time()
sa_combine = SA(Z[0], T_0, alpha, beta, M, Max_time)[0]
cpu_time.append(time.time() - start_time)

for i in range(1,30):
    start_time = time.time()
    sa_combine = sa_combine.append(SA(Z[i], T_0, alpha, beta, M, Max_time)[0])
    cpu_time.append(time.time() - start_time)
```

In [25]:
```
#average of Current_Cost and Best_Cost plot
plt.figure(figsize=[10,8])
plt.plot(sa_combine.groupby('Iteration_Number').mean().Current_Cost, '--')
plt.plot(sa_combine.groupby('Iteration_Number').mean().Best_Cost, '*-')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.legend(['Current_Cost','Best_Cost'])
plt.tight_layout()
```




```
In [26]: print('Mean at 1000th iteration: \n')
         print(sa_combine.groupby('Iteration_Number').mean().loc[1000])
```

```
Mean at 1000th iteration:

Current_Cost    9.374573e+08
Best_Cost       9.304431e+08
Name: 1000.0, dtype: float64
```

```
In [27]: print('Std at 1000th iteration: \n')
         print(sa_combine.groupby('Iteration_Number').std().loc[1000])
```

```
Std at 1000th iteration:

Current_Cost    2.607572e+07
Best_Cost       1.987003e+07
Name: 1000.0, dtype: float64
```

```
In [28]: # Average CPU time
         print("Average CPU Time: %.5fs" %np.mean(cpu_time))
```

```
Average CPU Time: 0.03457s
```

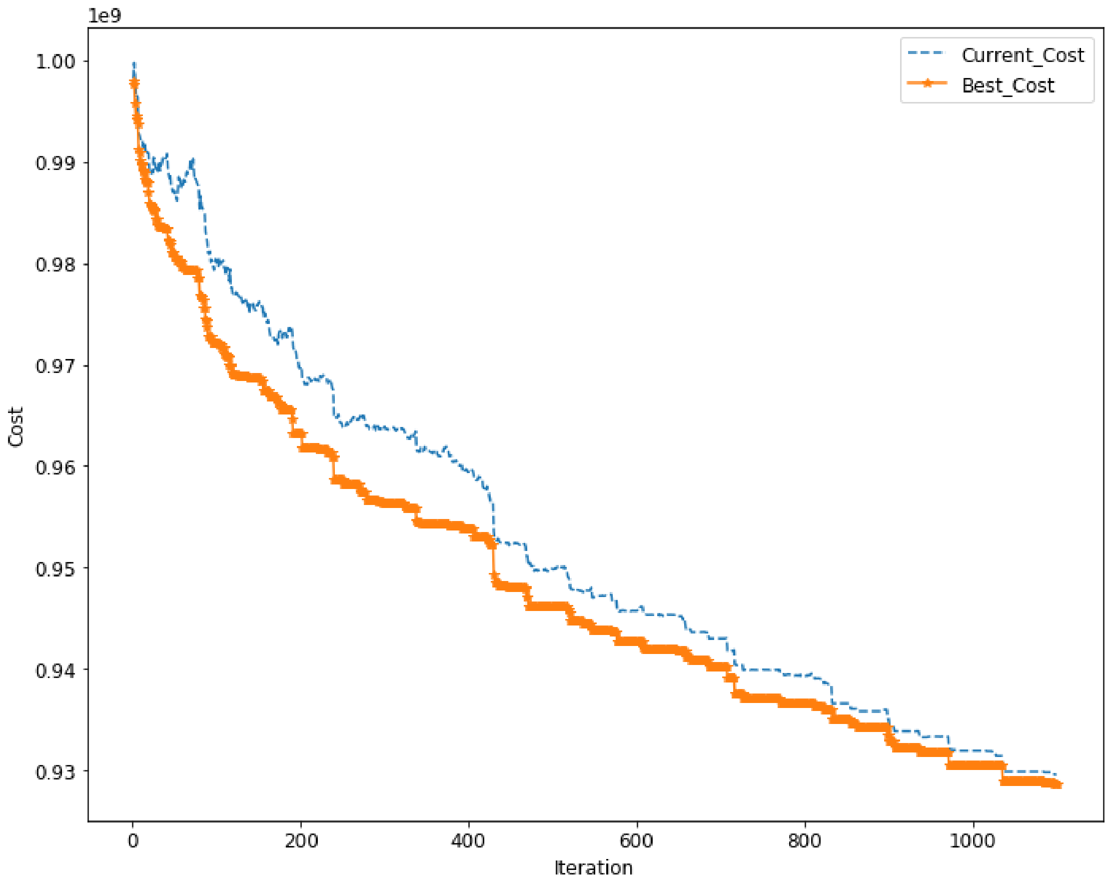
1.4.3 c)

```
In [29]: #np.random.seed(100)

         beta = 1
         G = 1000
         M = 1
         Max_time = 1100
         P_1 = 0.7
         P_2 = 0.05
         AP = 20

         # calculate algorithm parameters
         # param[3] is to use the same avg_delta_cost from part a
         T_0 = -param[3]/np.log(P_1)
         alpha = np.power(np.log(P_1)/np.log(P_2), 1/G)
         print('T_0: %.5f' %T_0)
         print('alpha: %.5f' %alpha)
```

```
T_0: 8682575.63928
alpha: 0.99787
```

```
In [30]: sa_combine_part_c = SA(Z[0], T_0, alpha, beta, M, Max_time)[0]

         for i in range(1,30):
             sa_combine_part_c = sa_combine_part_c.append(SA(Z[i], T_0, alpha, beta, M, Max_ti
```

```
In [31]: #average of Current_Cost and Best_Cost plot
         plt.figure(figsize=[10,8])
         plt.plot(sa_combine_part_c.groupby('Iteration_Number').mean().Current_Cost, '--')
         plt.plot(sa_combine_part_c.groupby('Iteration_Number').mean().Best_Cost, '*-')
         plt.xlabel('Iteration')
         plt.ylabel('Cost')
         plt.legend(['Current_Cost','Best_Cost'])
         plt.tight_layout()
```




```
In [32]: # Compare the mean at 1100th iteration for two P_1 values
         print("When P_1 = 0.9:    %.3f" %sa_combine.groupby('Iteration_Number').mean().loc[110
         print("When P_1 = 0.7:    %.3f" %sa_combine_part_c.groupby('Iteration_Number').mean()
```

```
When P_1 = 0.9:    930443110.363
When P_1 = 0.7:    928639254.100
```

1.4.4 d)

There aren't much improvement on Best Cost for from 1000th to 1100th iteration:

```
In [33]: print('When P1 = 0.9')
         print("1000th Iteration:    %.3f" %sa_combine.groupby('Iteration_Number').mean().loc[1
         print("1100th Iteration:    %.3f" %sa_combine.groupby('Iteration_Number').mean().loc[1
         print('When P1 = 0.7')
         print("1000th Iteration:    %.3f" %sa_combine_part_c.groupby('Iteration_Number').mean
         print("1100th Iteration:    %.3f" %sa_combine_part_c.groupby('Iteration_Number').mean
```

```
When P1 = 0.9
1000th Iteration:    930443110.363
```

```
1100th Iteration:    930443110.363
When P1 = 0.7
1000th Iteration:    930489923.274
1100th Iteration:    928639254.100
```