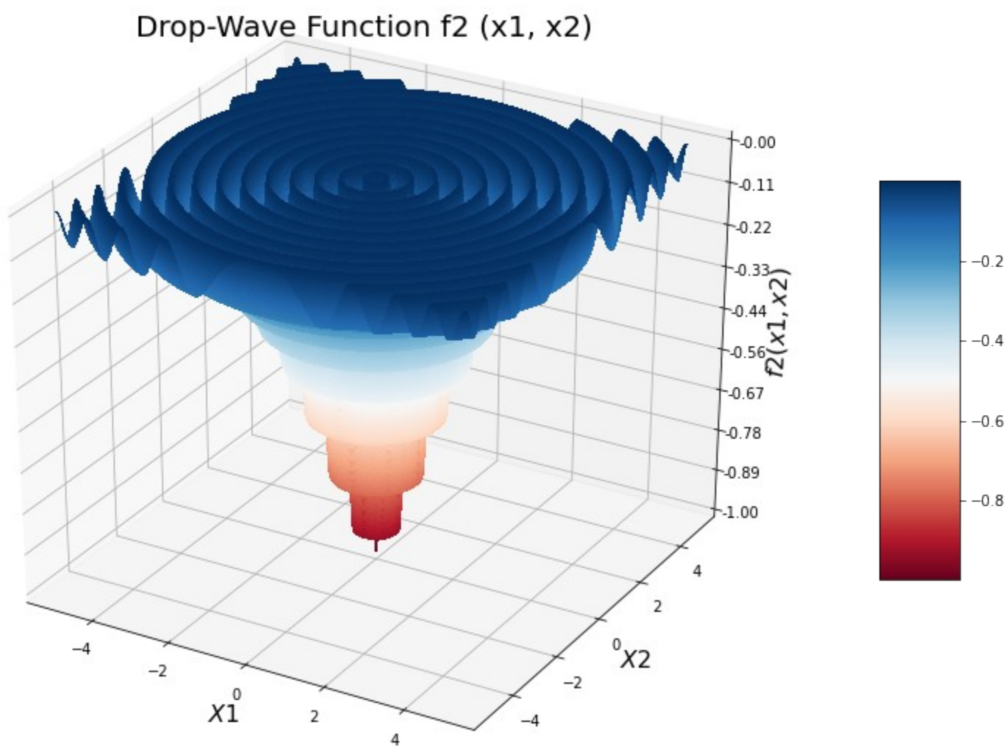
```
import numpy as np
import scipy
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter
import math
import pylab
# from google.colab import files

def f1(x):
    return math.pow(np.linalg.norm(x), 2)

def f2(x):
    norm_x = np.linalg.norm(x)
    return -(1 + math.cos(12 * norm_x))/(0.5 * math.pow(norm_x, 2) +
2)

def f3(x):
    return math.pow(np.linalg.norm(x), 2)
```

Question 1.1: Plot in 3D the drop-wave function f_2 defined above, over the domain $(x_1, x_2) \in [-5, 5] \times [-5, 5]$ (the vertical axis should show the value of $f_2(x_1, x_2)$ over this domain)

```
# Plot the drop-wave function f2
x_step = 0.01
x1_range = np.arange(-5, 5.01, x_step)
x2_range = np.arange(-5, 5.01, x_step)

X1_grid, X2_grid = pylab.meshgrid(x1_range, x2_range) # grid of point
assert(X1_grid.shape == X2_grid.shape)
print("X1-X2 Grid Shape: ", X1_grid.shape)
# print(X1_grid)
# print(X2_grid)
m, n = X1_grid.shape
F_X1_X2 = np.zeros((m,n)) # evaluation of the function on the grid

for i in range(m):
    for j in range(n):
        x = np.array([[X1_grid[i,j]], [X2_grid[i,j]]])
        F_X1_X2[i,j] = f2(x)

# print(F_X1_X2, F_X1_X2.shape)
fig = plt.figure(figsize=(14,10))
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X1_grid, X2_grid, F_X1_X2, rstride=1,
cstride=1, cmap=pylab.cm.RdBu, linewidth=0, antialiased=False)

ax.zaxis.set_major_locator(LinearLocator(10))
ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
```

```
fig.colorbar(surf, shrink=0.5, aspect=5)

plt.title("Drop-Wave Function f2 (x1, x2)")
ax.set_xlabel('$X1$', fontsize=16)
ax.set_ylabel('$X2$', fontsize=16)
ax.set_zlabel(r'$f2 (x1, x2)$', fontsize=16)
plt.grid(linestyle='--')
# plt.legend()
plt.show()
```

```
X1-X2 Grid Shape:  (1001, 1001)
```




```
# f1 = lambda x: math.pow(np.linalg.norm(x),2)
# f2 = lambda x: -(1 + math.cos(12 * np.linalg.norm(x)))/(0.5 *
pow(np.linalg.norm(x),2) + 2)
# f3 = lambda x: math.pow(np.linalg.norm(x),2)

parameters = {'figure.figsize':(10,6), 'axes.labelsize': 20,
'axes.titlesize': 20}
plt.rcParams.update(parameters)

def compute_df1_dx(x):
    n = x.shape[0]
    df1_dx = np.zeros((n,1))
    for i in range(n):
```

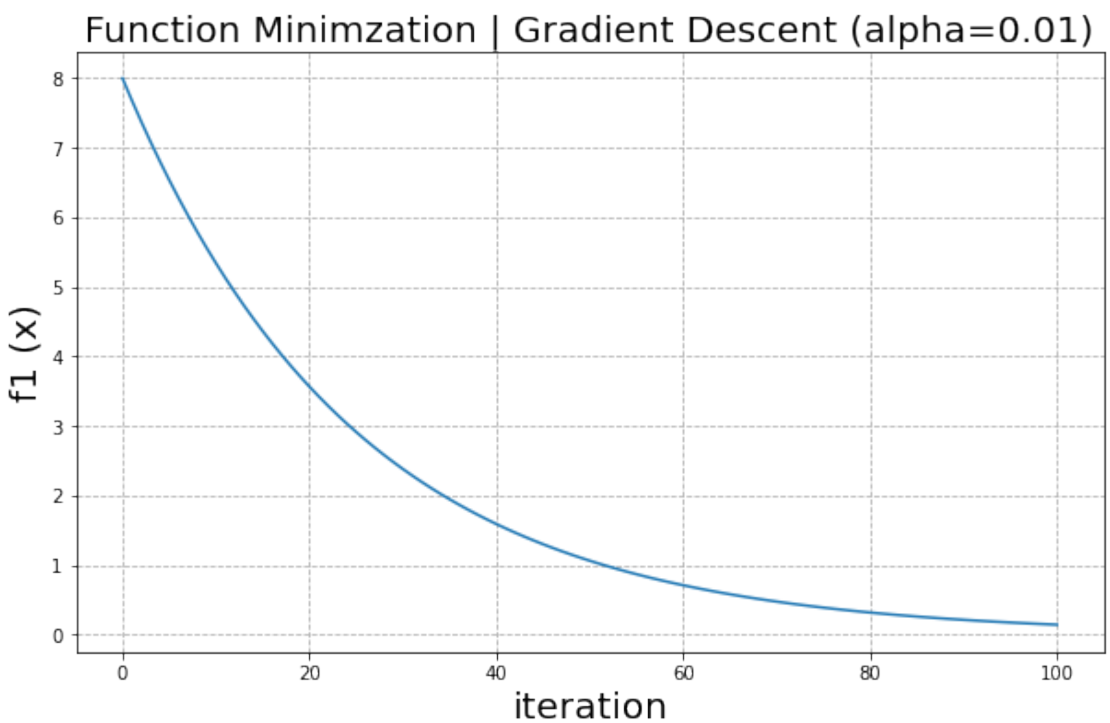
```
        df1_dx[i,0] = 2*x[i,0]

    return df1_dx

def compute_df2_dx(x):
    n = x.shape[0]
    df2_dx = np.zeros((n,1))

    term1 = np.linalg.norm(x)
    term2 = math.sin(12*term1)
    term3 = 0.5*math.pow(term1,2) + 2

    df2_dx[0,0] = ((12*x[0,0]) / term1)*(term2/term3) - (x[0,0]*f2(x))
/ term3
    df2_dx[1,0] = ((12*x[1,0]) / term1)*(term2/term3) - (x[1,0]*f2(x))
/ term3
    return df2_dx

def compute_df3_dx(x):
    n = x.shape[0]
    df3_dx = np.zeros((n,1))

    for i in range(n):
        df3_dx[i,0] = 2*x[i,0]

    return df3_dx
```

Question 1.2: Perform GD with step size $\alpha = 0.01$ on the three functions. Plot how the function value changes with respect to the number of iterations (x-axis: number of iterations, y-axis: function value; same for all the following plots as well).

```
# Gradient Descent with Fixed Step size
def grad_descent(x, f, compute_df_dx, max_itr=100, alpha=1e-2):
    n = x.shape[0]
    f_val = np.zeros((max_itr+1,1))
    x_val = np.zeros((max_itr+1,n))

    f_val[0,0] = f(x)
    x_val[0,:] = np.reshape(x, (n))

    for i in range(max_itr):
        p = -compute_df_dx(x)
        x = x + alpha*p

        x_val[i+1,:] = np.reshape(x, (n))
        f_val[i+1,0] = f(x)

    return x_val, f_val
```

```
max_itr = 100
alpha = 1e-2
x = np.ones((2,1))*2
x_val, f1_val = grad_descent(x, f1, compute_df1_dx, max_itr, alpha)

print("Minimum f1(x) =", round(f1_val[max_itr,0], 5), "achieved at x
=", x_val[max_itr,:])

fig = plt.figure()
plt.plot(range(max_itr+1), f1_val)
plt.title('Function Minimzation | Gradient Descent (alpha=0.01)')
plt.xlabel('iteration')
plt.ylabel('f1 (x)')
plt.grid(linestyle = '--')
plt.show()
# plt.savefig("GD_2_1_loss.png")
```

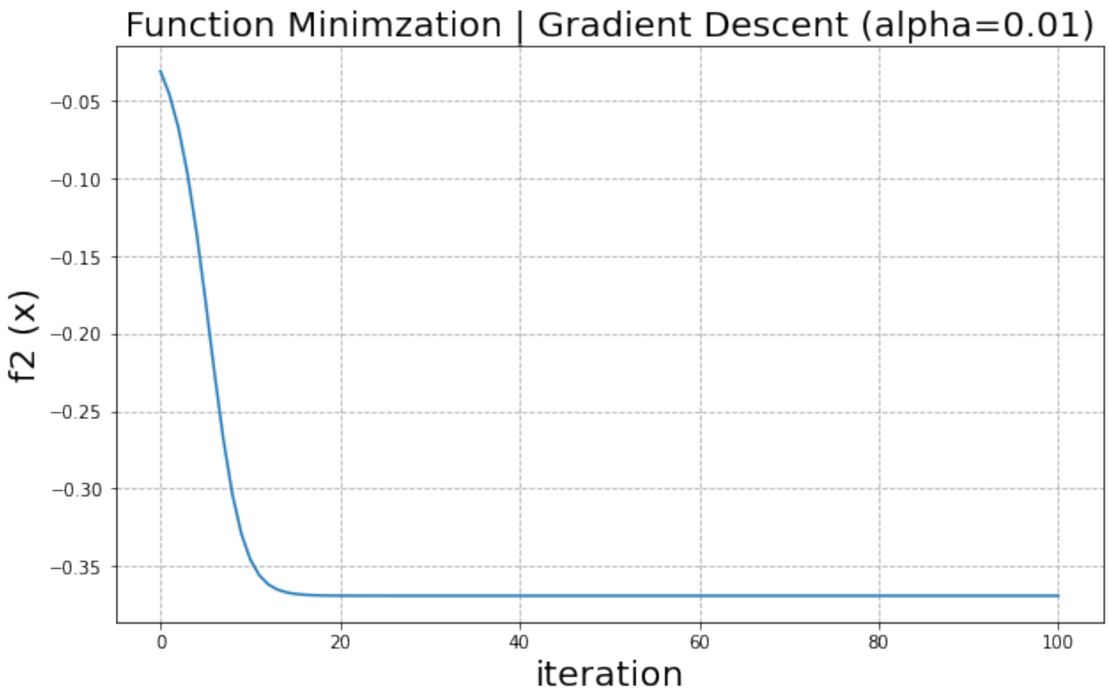
```
Minimum f1(x) = 0.1407 achieved at x = [0.26523911 0.26523911]
```




```
max_itr = 100
alpha = 1e-2
x = np.ones((2,1))*2
x_val, f2_val = grad_descent(x, f2, compute_df2_dx, max_itr, alpha)

print("Minimum f2(x) =", round(f2_val[max_itr,0], 5), " achieved at x
=", x_val[max_itr,:])

fig = plt.figure()
```

```
plt.plot(range(max_itr+1), f2_val)
plt.title('Function Minimzation | Gradient Descent (alpha=0.01)')
plt.xlabel('iteration')
plt.ylabel('f2 (x)')
plt.grid(linestyle = '--')
plt.show()
# plt.savefig("GD_2_1_loss.png")
```

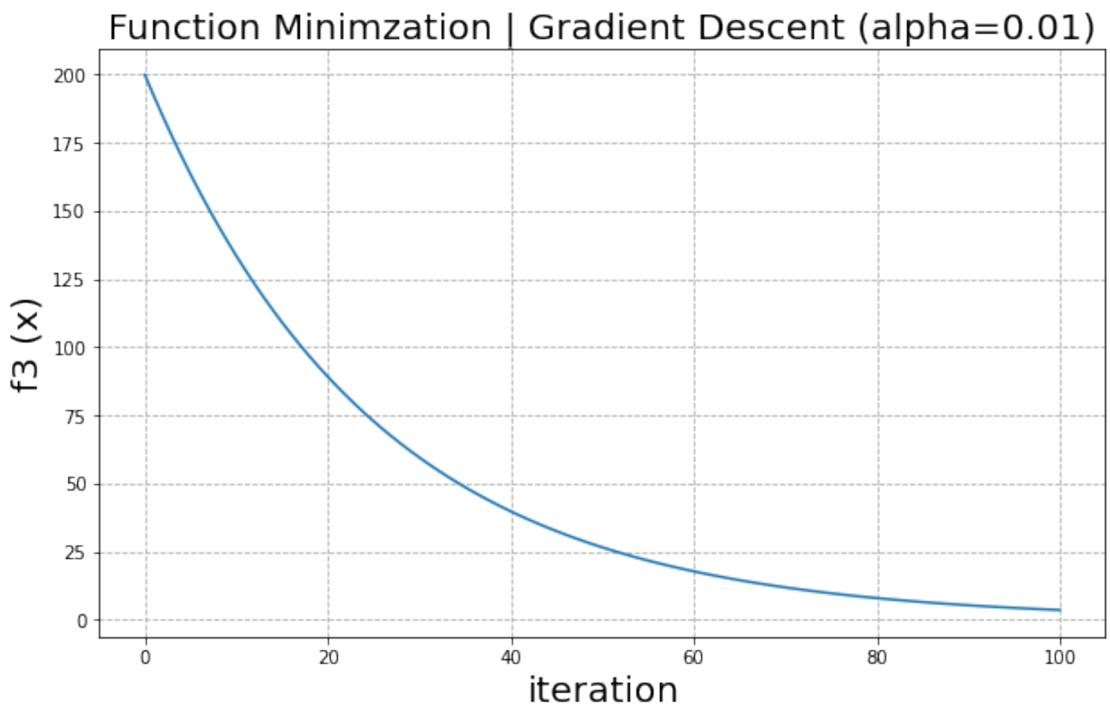
```
Minimum f2(x) = -0.36913  achieved at x = [1.84646292 1.84646292]
```




```
max_itr = 100
alpha = 1e-2
x = np.ones((50,1))*2
x_val, f3_val = grad_descent(x, f3, compute_df3_dx, max_itr, alpha)

print("Minimum f3(x) =", round(f3_val[max_itr,0], 5), " achieved at x
=", x_val[max_itr,:])

fig = plt.figure()
plt.plot(range(max_itr+1), f3_val)
plt.title('Function Minimzation | Gradient Descent (alpha=0.01)')
plt.xlabel('iteration')
plt.ylabel('f3 (x)')
plt.grid(linestyle = '--')
plt.show()
# plt.savefig("GD_2_1_loss.png")
```

```
Minimum f3(x) = 3.51759  achieved at x = [0.26523911 0.26523911
0.26523911 0.26523911 0.26523911 0.26523911
```

```
 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911
 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911
 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911
 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911
 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911
 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911
 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911 0.26523911
 0.26523911 0.26523911]
```

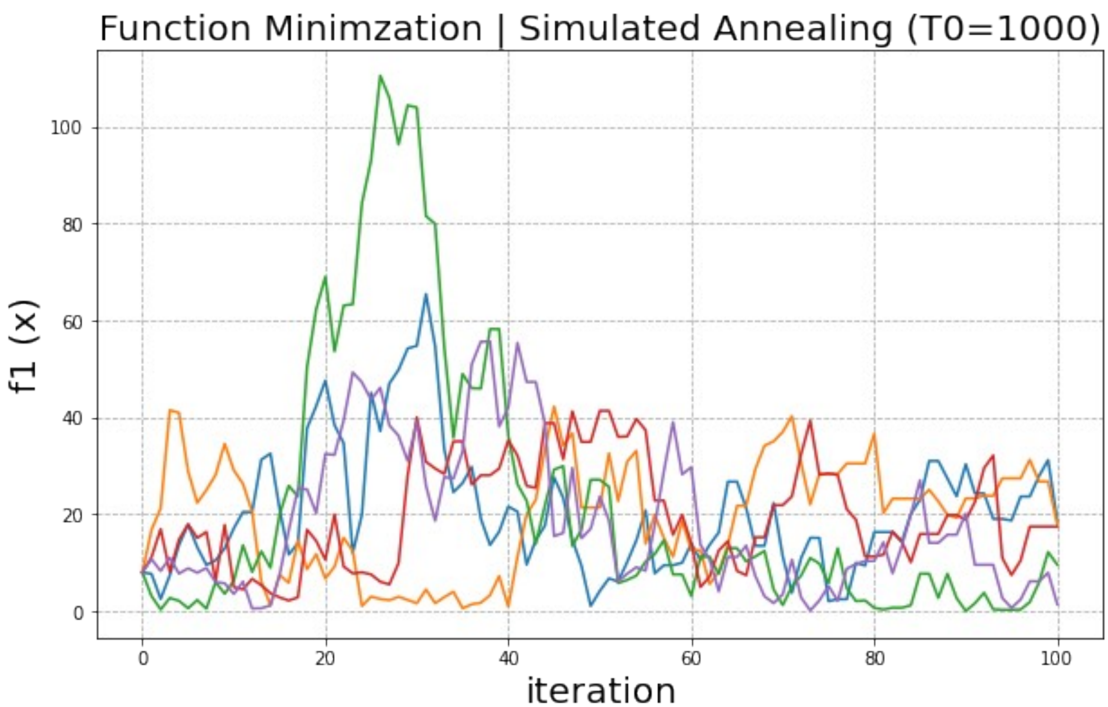


Question 1.3: Perform SA with two different initial temperatures T = 1000 and T = 10, for each function. Plot how the function values changes over iterations. Overall 3 functions and 2 different temperatures, so 6 graphs in total. Because the algorithm is stochastic, for each function and each temperature, plot 5 different runs (i.e. 5 different random seeds of your choice) in the same graph. That is, start at the same initial point, and since each step will be stochastic, you should plot 5 different trajectories for each function (i.e., 5 sequences of points in each of the 6 plots requested, use a different color for each sequence). This requirement is the same for the following two algorithms as well.

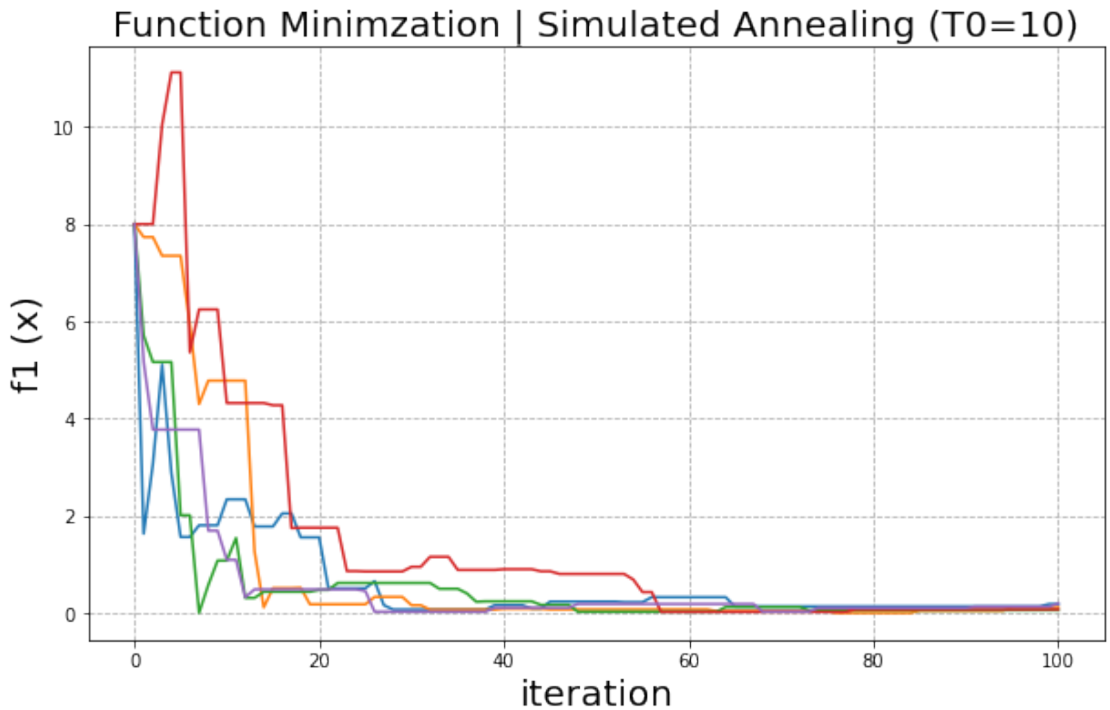
```python
# Simulated Annealing Function
def simulated_annealing(x, f, max_itr=100, init_T=1000, seed_val=1):
    # np.random.seed(seed_val)
    n = x.shape[0]
    f_val = np.zeros((max_itr+1,1))
    x_val = np.zeros((max_itr+1,n))

    f_val[0,0] = f(x)
    x_val[0,:] = np.reshape(x, (n))
```

```
    dx_mean = np.zeros((n))
    dx_cov = np.eye(n)

    for i in range(max_itr):
        dx = np.random.multivariate_normal(dx_mean,
dx_cov).reshape((n,1))
        new_x = x + dx
        delta_fx = f(new_x) - f(x)

        if (delta_fx >= 0):
            T = init_T/(i+1)
            prob_accept = math.exp(-delta_fx/T)
            if prob_accept >= 0.5:
                x = new_x
        else:
            x = new_x

        x_val[i+1,:] = np.reshape(x, (n))
        f_val[i+1,0] = f(x)

    return x_val, f_val

# SA for f1
init_temp_lst = [1000, 10]

for init_temp in init_temp_lst:
    fig = plt.figure()
    seed_val = [1,2,3,4,5]
    num_runs = 5

    for run in range(num_runs):
        max_itr = 100
        x = np.ones((2,1))*2
        x_val, f1_val = simulated_annealing(x, f1, max_itr, init_temp,
seed_val[run])

        print("Minimum f1(x) =", round(f1_val[max_itr,0], 5),
"achieved at x =", x_val[max_itr,:], " for T =", init_temp)
        plt.plot(range(max_itr+1), f1_val)

    plt.title('Function Minimzation | Simulated Annealing
(T0='+str(init_temp)+')')
    plt.xlabel('iteration')
    plt.ylabel('f1 (x)')
    plt.grid(linestyle = '--')
    # plt.legend()
    plt.show()
    # plt.savefig("GD_2_1_loss.png")
```

```
Minimum f1(x) = 18.74983 achieved at x = [-2.64076457  3.43164598]
for T = 1000
Minimum f1(x) = 17.90776 achieved at x = [ 3.96257354 -1.48518226]
for T = 1000
Minimum f1(x) = 9.60935 achieved at x = [2.9478939  0.95878866]  for T
= 1000
Minimum f1(x) = 17.45158 achieved at x = [1.95056259 3.69416912]  for
T = 1000
Minimum f1(x) = 1.44403 achieved at x = [-0.95261952  0.73249318]  for
T = 1000
```

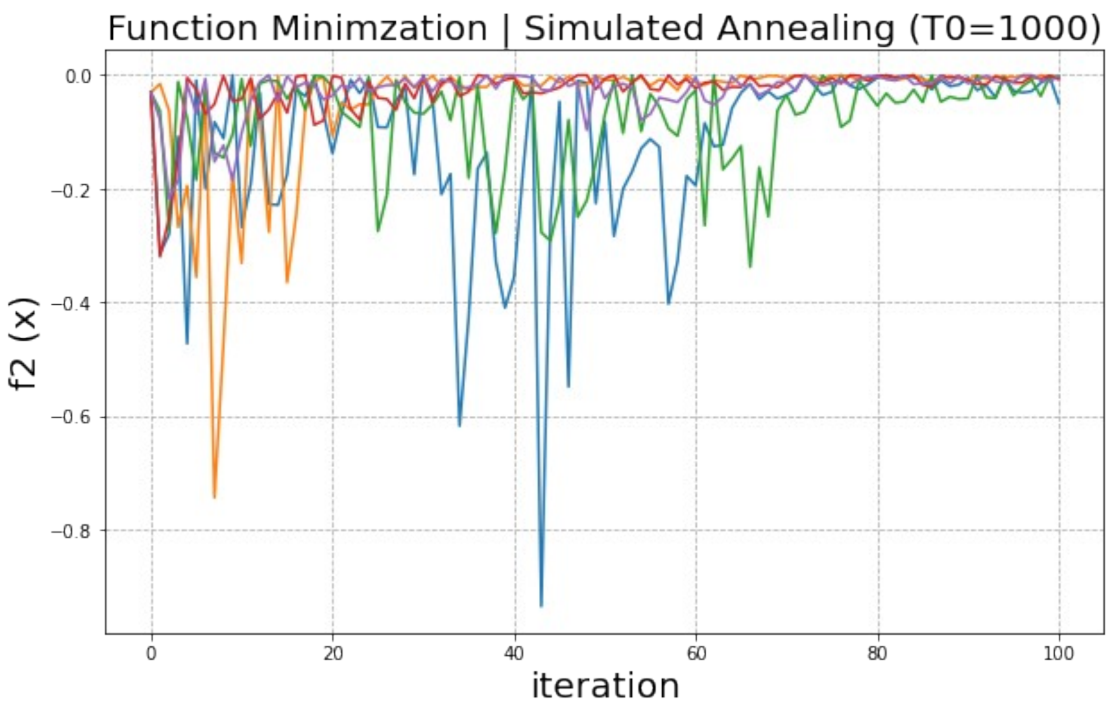



```
Minimum f1(x) = 0.18469 achieved at x = [-0.09180763  0.41983702]  for
T = 10
Minimum f1(x) = 0.11411 achieved at x = [0.31153548 0.13058416]  for T
= 10
Minimum f1(x) = 0.05948 achieved at x = [ 0.1950825  -0.14637554]  for
T = 10
Minimum f1(x) = 0.0755 achieved at x = [0.15897298 0.22411945]  for T
= 10
Minimum f1(x) = 0.19615 achieved at x = [-0.332542    0.29251959]  for
T = 10
```

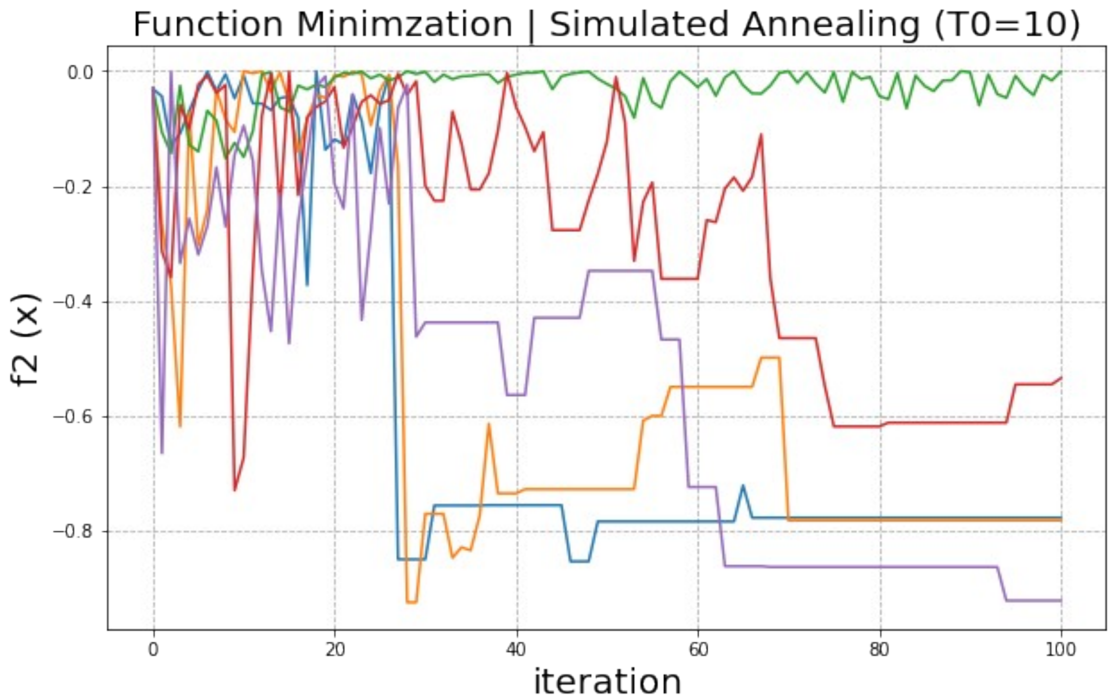



```
# SA for f2
init_temp_lst = [1000, 10]

for init_temp in init_temp_lst:
    fig = plt.figure()
    seed_val = [1,2,3,4,5]
    num_runs = 5

    for run in range(num_runs):
        max_itr = 100
        x = np.ones((2,1))*2

        x_val, f2_val = simulated_annealing(x, f2, max_itr, init_temp,
seed_val[run])

        print("Minimum f2(x) = ", round(f2_val[max_itr,0], 5), "
achieved at x = ", x_val[max_itr,:], " for T=", init_temp)
        plt.plot(range(max_itr+1), f2_val)

    plt.title('Function Minimzation | Simulated Annealing
(T0='+str(init_temp)+')')
    plt.xlabel('iteration')
    plt.ylabel('f2 (x)')
    plt.grid(linestyle = '--')
    # plt.legend()
    plt.show()
    # plt.savefig("GD_2_1_loss.png")
```

```
Minimum f2(x) =  -0.04938  achieved at x =  [6.99397285 4.51615881]
for T= 1000
Minimum f2(x) =  -0.00305  achieved at x =  [-21.28094935  -
6.1173565 ]  for T= 1000
Minimum f2(x) =  -0.00346  achieved at x =  [7.05484784 4.09531046]
for T= 1000
Minimum f2(x) =  -0.00779  achieved at x =  [ 11.51446785 -
13.71930505]  for T= 1000
Minimum f2(x) =  -0.00183  achieved at x =  [ 7.81645624 17.51747634]
for T= 1000
```



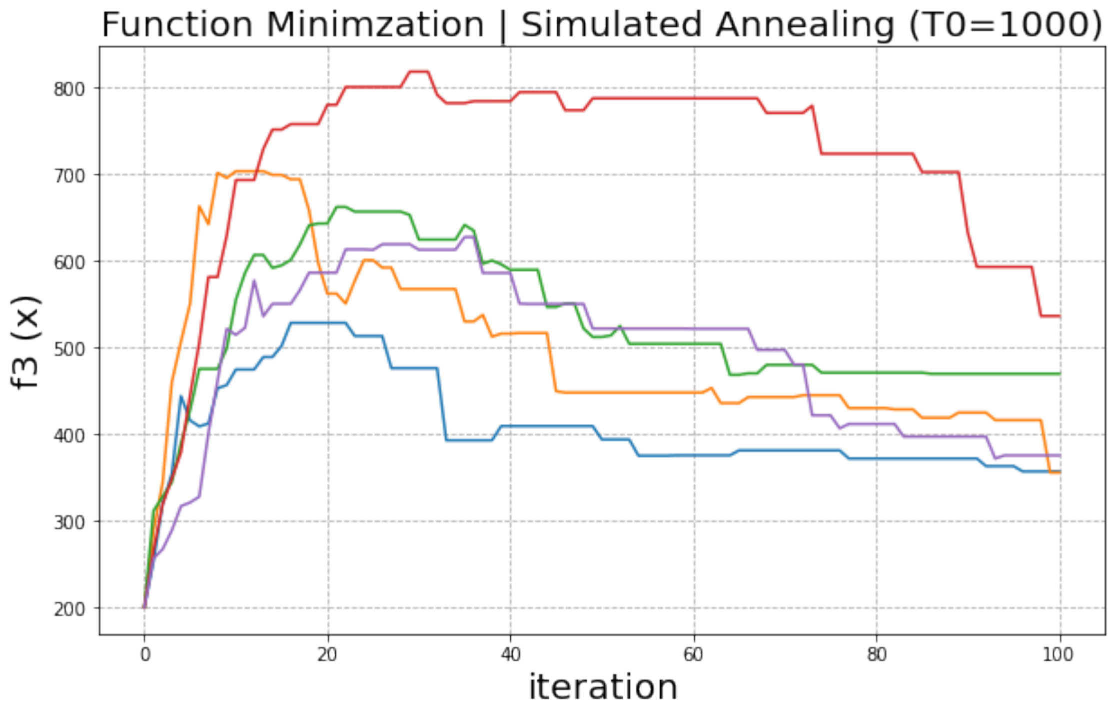

```
Minimum f2(x) =  -0.7782  achieved at x =  [ 0.43543221 -0.96408892]
for T= 10
Minimum f2(x) =  -0.7829  achieved at x =  [-1.03018421  0.05177686]
for T= 10
Minimum f2(x) =  -0.00075  achieved at x =  [ 2.39963122 -9.89976287]
for T= 10
Minimum f2(x) =  -0.53519  achieved at x =  [0.48110134 0.4212081 ]
for T= 10
Minimum f2(x) =  -0.92265  achieved at x =  [-0.0671658   0.49562803]
for T= 10
```




```
# SA for f3
init_temp_lst = [1000, 10]

for init_temp in init_temp_lst:
    fig = plt.figure()
    seed_val = [1,2,3,4,5]
    num_runs = 5

    for run in range(num_runs):
        max_itr = 100
        x = np.ones((50,1))*2

        x_val, f3_val = simulated_annealing(x, f3, max_itr, init_temp, 
seed_val[run])

        print("Minimum f3(x) = ", round(f3_val[max_itr,0], 5), " 
achieved at x = ", x_val[max_itr,:])
        plt.plot(range(max_itr+1), f3_val)

    plt.title('Function Minimzation | Simulated Annealing 
(T0='+str(init_temp)+')')
    plt.xlabel('iteration')
    plt.ylabel('f3 (x)')
    plt.grid(linestyle = '--')
    # plt.legend()
    plt.show()
    # plt.savefig("GD_2_1_loss.png")
```

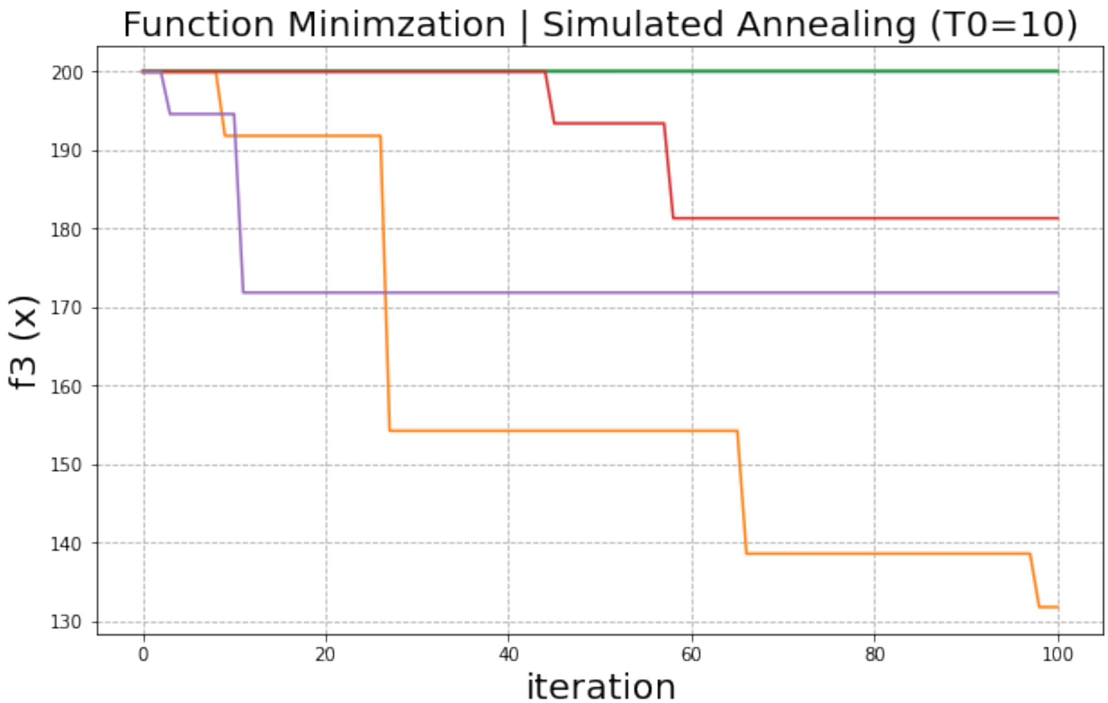
```
Minimum f3(x) =  356.66973  achieved at x =  [-0.98295245 -3.43299076
-4.52568177  0.10861091 -3.76052004  3.47604528
 -1.32451709 -3.07943833 -1.27482712 -3.07959755  0.70344594
2.80591002
  2.00893862 -0.99524251  1.80664292 -3.10465526  2.20470423 -
3.71521286
 -3.54271841  0.92125323  3.96127466 -0.23121954 -0.78778903 -
0.37568652
  1.68956407 -1.68454979  0.76659693  0.84325436 -2.40629965 -
1.40730364
  4.39509582  1.53053898  2.36680601  2.76374017  1.96537147
0.15768428
 -0.0135786   3.28221425  6.72512312 -1.96568259  4.42829315 -
3.67502716
  1.08017751 -2.70283874  2.9500736   4.3542191   0.04222725 -
0.7404826
  2.21078778 -3.44758503]
Minimum f3(x) =  355.33478  achieved at x =  [ 0.6643464  -1.25335965
-2.84989685  1.75345705  2.38635062  4.45425547
 -1.05676172  0.92062146  0.16880908  3.61648965 -1.61675908
0.47643814
  5.49971882 -3.68727763 -0.11995389 -3.33343077 -0.99370497
5.95840654
  5.01962087 -0.70592526  0.07851107  1.59771741  1.85737713
2.28374098
  0.06494723  0.20988069 -2.30038991 -0.20484984 -1.58777532 -
3.03744539
 -1.33432675 -1.43483284  0.6871805   0.66635575 -0.17101369
1.86686033
  7.56406512  1.31187804 -0.24944984 -3.65169585  3.84151404
3.96677616
  1.43887099  2.53384885 -2.56841625 -0.46652803 -2.67997447
0.72444961
  4.35723625  0.2784463 ]
Minimum f3(x) =  469.42746  achieved at x =  [-4.48032649  2.31540365
-2.28574979  0.63962504  2.49487084 -3.57195749
  5.6220503  -2.65914663  8.24948394  2.39897958  0.58496249
1.52748971
  2.52844842 -1.36342822  4.18407507 -2.86992231 -1.19431952
3.18509275
 -0.54396964  0.00932868 -3.0028211   0.26533181  7.15593489 -0.915465
 -4.36852556 -0.78625745 -3.32742302 -3.00096959  2.05147193 -
0.07758731
  2.0685584   2.47779228  1.60119627  1.40733895 -0.4865658
0.01827081
  3.21414504  1.26504606  1.9954286   0.34661836  1.80733436 -
4.71169784
  0.31852669  1.73117717 -2.92325134  2.12027119 -0.98075189
0.82520861
 -1.67529568 -8.71409653]
```

```
Minimum f3(x) =  536.23358  achieved at x =  [-3.14315029 -0.51506411
3.14011677 -0.05001047  2.84347249  8.76650074
  2.95235742 -0.4673108   5.07964646  6.09063299 -1.1457268
2.07921043
 -1.20753762  3.08927393  0.19862139 -4.90333972  0.86159752 -
1.52986925
 -3.21442806 -0.61346388 -1.67745809  4.57142447  5.49065897
4.13015794
  2.44353685 -0.72329221 -1.04711453 -0.43354823  0.33373562
3.15132955
 -0.17121116  1.62173259  1.79683208 -0.68526186 -4.97689923
1.47031937
 -4.26552154  7.72690461  6.05399936 -2.01277845 -0.2770818
3.10137242
 -0.47782142 -3.14063982  2.35998992 -0.60040149 -0.56938095
1.07824865
  6.0854246   0.28107662]
Minimum f3(x) =  375.15969  achieved at x =  [-3.49139687 -1.4550385
-0.77391454  2.55063427  1.83193837 -3.88013804
 -0.88411055  0.85385172  0.15094399  1.2568856   5.5019494  -
1.2061727
  3.12294934  2.10365096  3.32340276 -0.22756884 -1.4795765
0.47307438
  0.96693856 -0.43025831  1.88127884  0.14377064  1.73217473
6.53613469
 -0.09264139  7.26559136  4.25559294 -0.50202059  1.80643965 -
1.13102419
  2.96487421  2.41923121  6.99057451  0.52061663 -0.77307786 -
2.31057507
 -3.83420103  0.57384347 -1.06192521  2.03877551  0.12028292 -
2.54209885
  1.63253696 -0.77207562 -2.59229259  2.33850116 -2.86571916
0.30126604
 -0.16245807  5.13966871]
```




```
Minimum f3(x) =  199.99936  achieved at x =  [1.38190298 2.45105914
1.65731257 2.44894239 1.96343682 1.01233219
 0.86341176 2.5395425  2.18734357 1.62229689 2.5009853  1.14817123
 2.03120185 3.18153259 3.85795532 1.09289703 2.39556336 1.57478324
 2.40206458 1.85689849 3.68078791 1.0749938  1.70053595 3.92481288
 1.17519537 1.30557648 1.11542118 1.09939317 1.70134292 0.54174573
 1.50859506 0.93932784 1.93087225 1.37829765 1.92235853 3.0525717
 2.08373019 1.84294509 1.73279572 2.8307135  0.56937404 1.24207155
 2.17799445 1.00873847 1.1987885  1.09058894 3.3097515  1.15970345
 1.52297574 1.16513173]
Minimum f3(x) =  131.79148  achieved at x =  [-0.21533854 -0.09240056
-1.05704635  1.92685012  0.26049592  0.71699287
  1.74645219  0.1654867  -0.89675077  2.82839102  0.27234449 -
0.72254786
  0.83685072  2.71993375  2.13257086  0.16729279  2.00556279
0.7632107
  1.83458258  2.34270763  0.34789879 -1.65240359  2.60358216 -
0.14049433
  1.88990606 -0.42805909 -0.63132551  2.68278947  1.91444196
0.71097749
  1.41494636  0.32477872  1.19353665  1.09840696  1.0958392
2.92751715
  4.57468744  1.27806021  1.90926192  1.6935378   1.6421629  -
0.79735927
 -0.502701   -0.53002004  1.47932282  0.45961795  1.04870651
2.91993706
  1.11580337  1.3496736 ]
Minimum f3(x) =  200.0  achieved at x =  [2. 2. 2. 2. 2. 2. 2. 2. 2.
```

```
2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.
2.
 2. 2.]
Minimum f3(x) =  181.32742  achieved at x =  [ 1.89867158  1.39255114
0.11446885  1.32120298  2.22691965  1.53159613
  0.6926864   1.5878774   2.33926593  0.51297626  1.68880153 -
0.39514628
  0.4290913   2.34761948  2.77000061  2.61253852  1.9000076
1.54190477
  2.99169377  0.2119074   0.03783606  1.6127867   2.34434713
4.39091297
  1.95001499  0.22108502  2.9041895   2.0217408   1.78623157 -
0.06479108
  3.78485241  1.26731547  1.6171638   2.40720722 -0.25842904
1.37439734
  0.37906042  3.29703593  0.93291869 -0.11510591  0.4536743
1.34396667
 -1.62335096  2.69148837  2.6959538  -0.569712    0.07104187
2.03526674
  2.63774162  2.01699404]
Minimum f3(x) =  171.85414  achieved at x =  [ 1.40207101  1.81054707
1.04672851  1.1649782   1.85605163  0.68884359
 -0.54314173  1.40986819  2.44682536  0.62781071  0.99816046
1.33659786
  1.850293   -0.06147082  2.72043919  3.36921363  0.70113671
3.06912129
 -1.26281789  1.59330273  1.93282784  1.05271124  0.30075328
0.10064187
  0.5692493   1.05288677  0.01243998  0.03621264  3.21076811
3.26569614
  0.64362733 -1.56135669  2.52268518  3.03275234  1.44650535 -
0.1971049
  0.04434745  1.85769747  2.92541047  0.55734506  1.30663335
2.54151236
  2.93798393  0.65810361  2.18215412  2.95181758  0.1009879
2.15158235
  3.42804681  2.29144623]
```

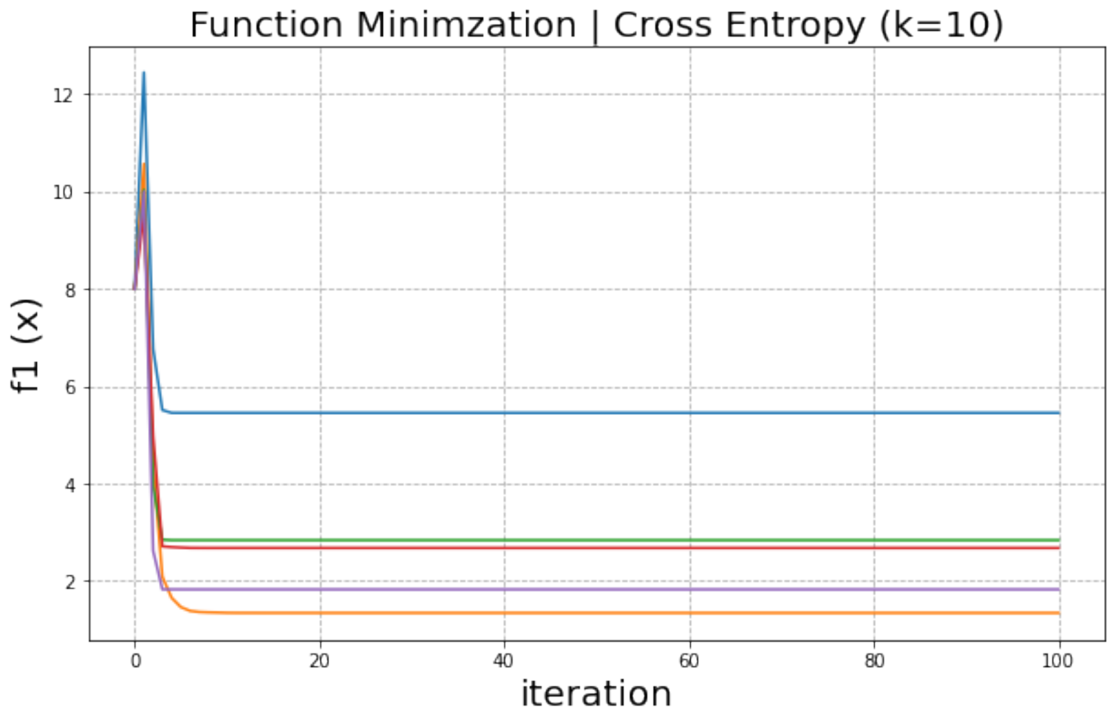


Question 1.4: Perform CE with two sample sizes, k = 10 and k = 50, for each function. Perform 5 runs for each function and each sample size. Plot the average of the function values of all samples in each iteration for each function. Overall 6 plots (each plot should show 5 random trajectories).

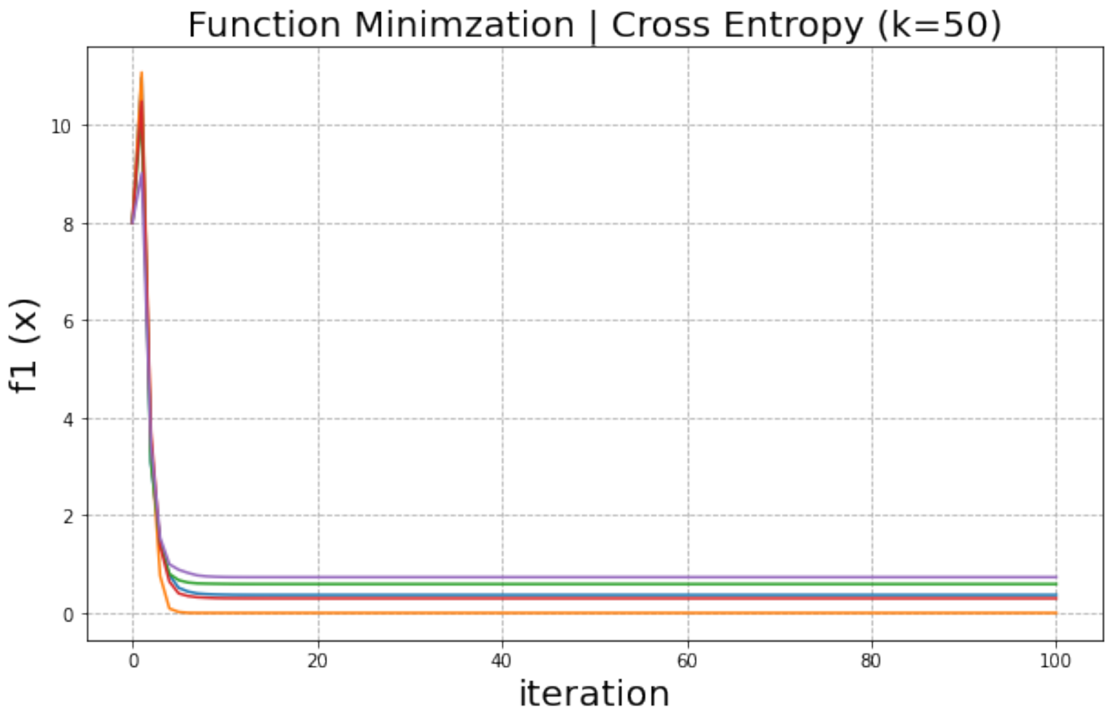
```
# Cross-Entropy Method
def cross_entropy(x, f, max_itr=100, sample_size=10, elite_frac=0.2):
    n = x.shape[0]
    f_val = np.zeros((max_itr+1,1))
    x_val = np.zeros((max_itr+1,n))

    f_val[0,0] = f(x)
    x_val[0,:] = x.reshape(n)

    elite_mean = np.zeros((n,1)) + x
    elite_mean = elite_mean.reshape(n)
    elite_cov = np.eye(n)
    elite_sample_size = int(elite_frac * sample_size)

    for i in range(max_itr):
        x_samples = np.random.multivariate_normal(elite_mean,
elite_cov, size=(sample_size))

        f_val_sample = np.zeros(sample_size)
        for j in range(sample_size):
            f_val_sample[j] = f(np.reshape(x_samples[j,:], (n,1)))

        sort_index = np.argsort(f_val_sample, axis=0)
```

```
        elite_samples = x_samples[sort_index[0:elite_sample_size], :]

        elite_mean = np.mean(elite_samples, axis=0)
        elite_cov = np.cov(elite_samples, rowvar=False)

        # f_val[i+1,0] = f(elite_mean.reshape(n,1))
        f_val[i+1,0] = np.mean(f_val_sample)
        x_val[i+1,:] = elite_mean

    return x_val, f_val

# Cross-Entropy for f1(x)
sample_size_lst = [10, 50]

for sample_size in sample_size_lst:
    fig = plt.figure()
    num_runs = 5
    for run in range(num_runs):
        max_itr = 100
        x = np.ones((2,1))*2
        x_val, f1_val = cross_entropy(x, f1, max_itr, sample_size)

        print("Minimum f1(x) =", round(f1_val[max_itr,0], 5), "
achieved at x =", x_val[max_itr,:], " for sample K =", sample_size)
        plt.plot(range(max_itr+1), f1_val)

    plt.title('Function Minimzation | Cross Entropy
(k='+str(sample_size)+')')
    plt.xlabel('iteration')
    plt.ylabel('f1 (x)')
    plt.grid(linestyle = '--')
    # plt.legend()
    plt.show()
```

```
Minimum f1(x) = 5.45131  achieved at x = [1.35521965 1.90123378]  for
sample K = 10
Minimum f1(x) = 1.34232  achieved at x = [0.89588586 0.73464804]  for
sample K = 10
Minimum f1(x) = 2.83752  achieved at x = [1.43938653 0.87503403]  for
sample K = 10
Minimum f1(x) = 2.67646  achieved at x = [1.28751706 1.00933873]  for
sample K = 10
Minimum f1(x) = 1.82611  achieved at x = [1.19747349 0.62622999]  for
sample K = 10
```



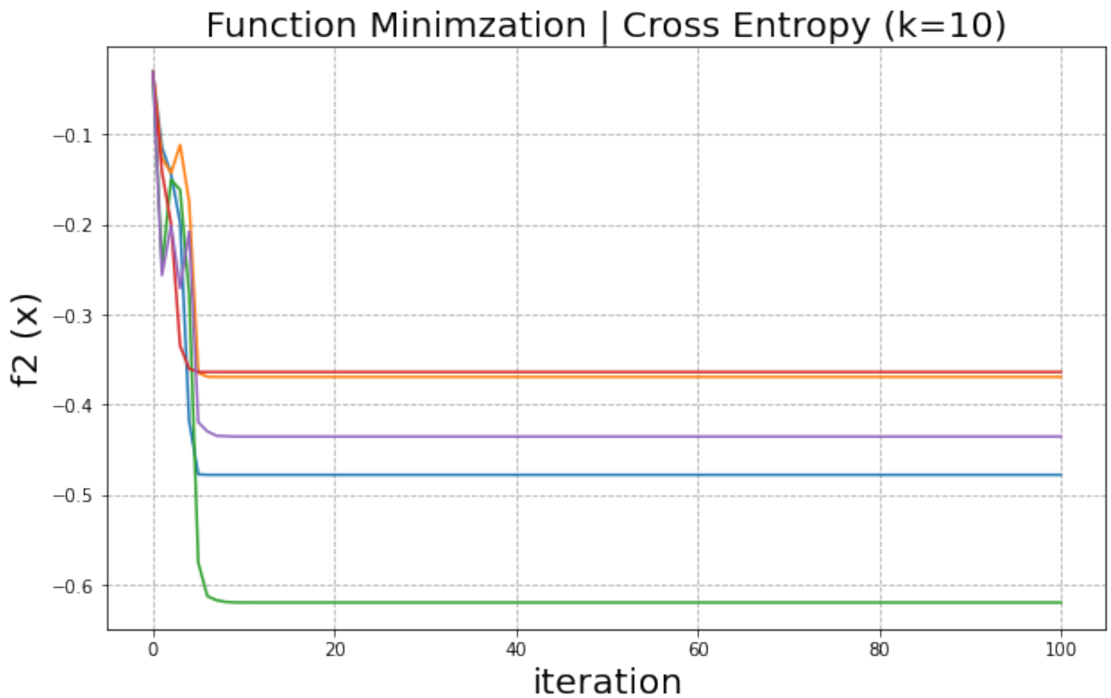

```
Minimum f1(x) = 0.36943  achieved at x = [0.3726456  0.48017723]  for
sample K = 50
Minimum f1(x) = 0.0  achieved at x = [-4.29401339e-32 -1.88180761e-33]
for sample K = 50
Minimum f1(x) = 0.59226  achieved at x = [0.41109569 0.65058402]  for
sample K = 50
Minimum f1(x) = 0.29865  achieved at x = [0.1984778  0.50917071]  for
sample K = 50
Minimum f1(x) = 0.73335  achieved at x = [0.65748811 0.54868698]  for
sample K = 50
```



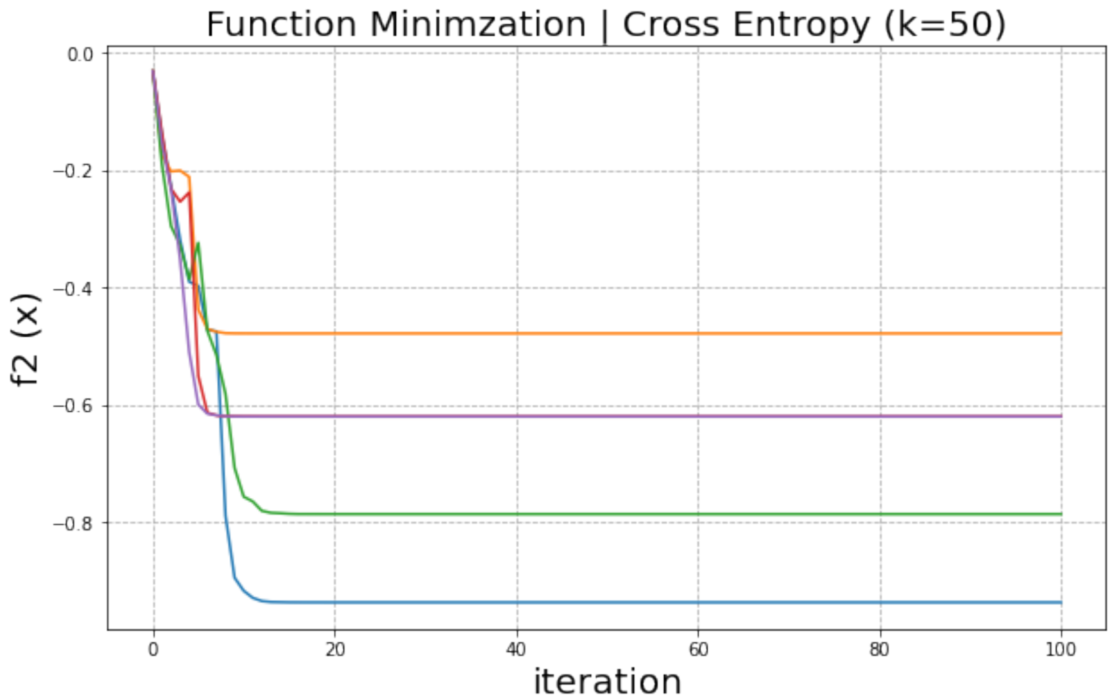

```
# Cross-Entropy for f2(x)
sample_size_lst = [10, 50]

for sample_size in sample_size_lst:
    fig = plt.figure()
    num_runs = 5
    for run in range(num_runs):
        max_itr = 100
        x = np.ones((2,1))*2
        x_val, f2_val = cross_entropy(x, f2, max_itr, sample_size)

        print("Minimum f2(x) =", round(f2_val[max_itr,0], 5), "
achieved at x =", x_val[max_itr,:], " for sample K =", sample_size)
        plt.plot(range(max_itr+1), f2_val)

    plt.title('Function Minimzation | Cross Entropy
(k='+str(sample_size)+')')
    plt.xlabel('iteration')
    plt.ylabel('f2 (x)')
    plt.grid(linestyle = '--')
    # plt.legend()
    plt.show()
```

```
Minimum f2(x) = -0.47778  achieved at x = [0.66209408 1.97996539]  for
sample K = 10
Minimum f2(x) = -0.36913  achieved at x = [2.37658883 1.0819374 ]  for
sample K = 10
Minimum f2(x) = -0.6195  achieved at x = [-1.54687495 -0.23122102]
```

for sample K = 10
Minimum f2(x) = -0.36367 achieved at x = [2.07928434 1.5459393] for sample K = 10
Minimum f2(x) = -0.43533 achieved at x = [-0.04234957 2.13766222] for sample K = 10



Minimum f2(x) = -0.93625 achieved at x = [0.43250623 0.28907195] for sample K = 50
Minimum f2(x) = -0.47778 achieved at x = [1.40423923 1.54454061] for sample K = 50
Minimum f2(x) = -0.78575 achieved at x = [0.78661526 0.68262696] for sample K = 50
Minimum f2(x) = -0.6195 achieved at x = [0.51508901 1.47681003] for sample K = 50
Minimum f2(x) = -0.6195 achieved at x = [0.82468692 1.32897556] for sample K = 50

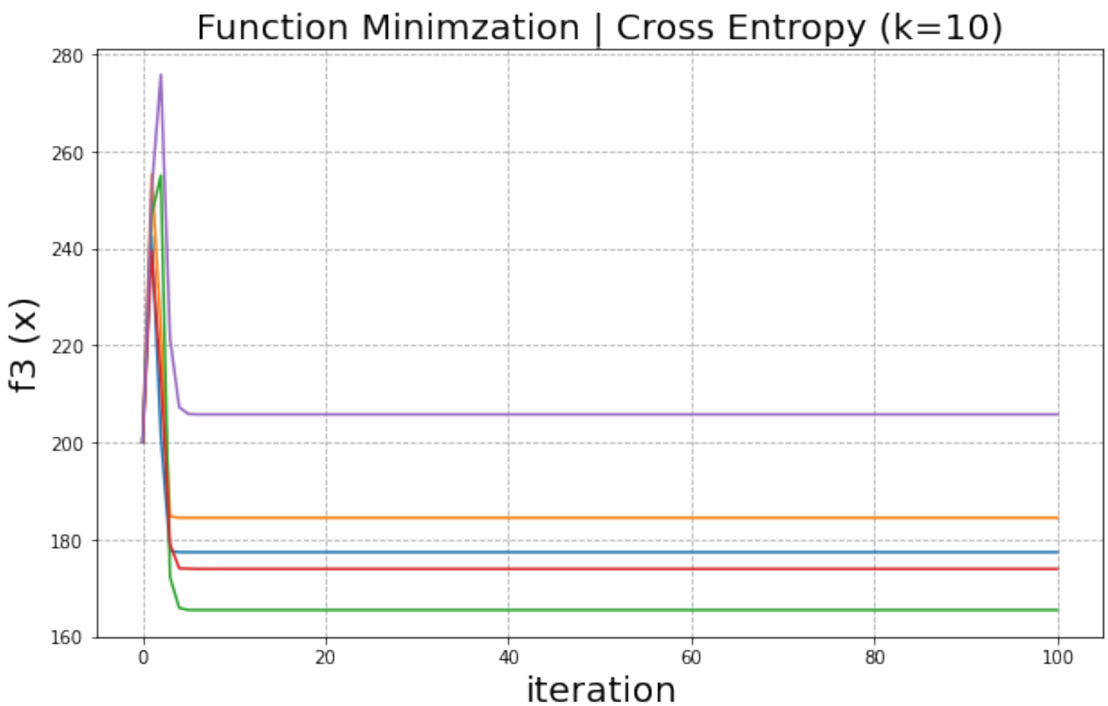



```
# Cross-Entropy for f3(x)
sample_size_lst = [10, 50]

for sample_size in sample_size_lst:
    fig = plt.figure()
    num_runs = 5
    for run in range(num_runs):
        max_itr = 100
        x = np.ones((50,1))*2
        x_val, f3_val = cross_entropy(x, f3, max_itr, sample_size)

        print("Minimum f3(x) =", round(f3_val[max_itr,0], 5), "
achieved at x =", x_val[max_itr,:])
        plt.plot(range(max_itr+1), f3_val)

    plt.title('Function Minimzation | Cross Entropy
(k='+str(sample_size)+')')
    plt.xlabel('iteration')
    plt.ylabel('f3 (x)')
    plt.grid(linestyle = '--')
    # plt.legend()
    plt.show()
```

```
Minimum f3(x) = 177.41878  achieved at x = [0.65731265 1.33449839
1.40330609 1.59204474 2.05571709 1.99928727
 1.94574693 2.22665074 0.88430474 2.55878073 2.11062558 2.28767047
 1.38629453 1.40837195 0.14357411 1.06746918 2.10442969 1.5502938
 1.85529806 0.95002385 1.55865435 1.19187347 1.40079941 1.85384649
```

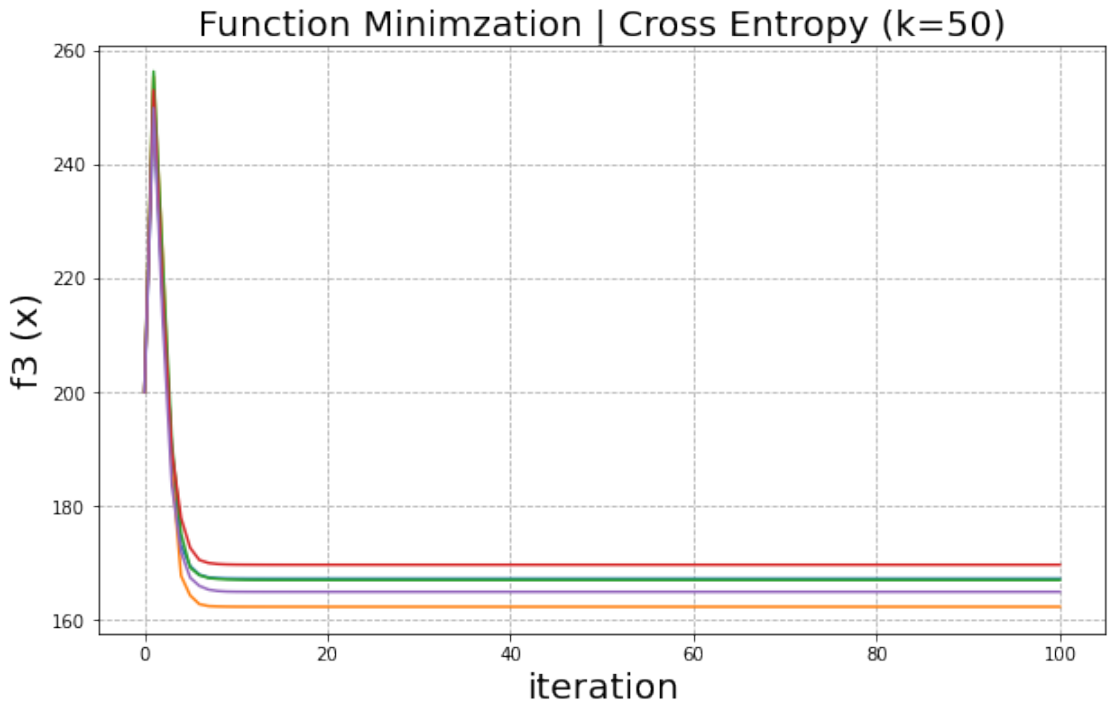
```
 2.68835374 1.82676173 1.08460009 2.55767387 2.55128796 2.48185649
 1.40010547 2.00663591 1.5136292  2.32728583 0.83185115 1.28821296
 0.89202651 1.72888695 1.50978723 2.04084071 1.37610911 2.69249967
 2.46416066 1.40007859 1.93736282 3.70629402 2.47779058 2.28062921
 2.20803035 1.76598208]
Minimum f3(x) = 184.50663  achieved at x = [ 1.15614235  1.40629207
1.37741408  0.80231881  1.50516479  2.94435442
  1.03201195  1.92147207  1.34739311  1.32810787  2.5533453
1.64874943
  2.04776961  1.48894899  1.73590708  2.25371964  1.58792765
2.13135712
  1.35165144  1.68094695  1.09150787  2.114235    0.76154196
1.99858621
  2.96541717  1.34888904 -0.04399091  2.86398376  2.40417555
1.67155857
  1.63288987  1.63342573  2.49451589  1.15500721  1.95214017
1.47847407
  2.56011063  2.70879101  1.20483804  1.04084831  3.0349001
1.24524883
  2.33550751  2.47895524  2.62126163  2.26554372  0.60438993
1.86365489
  2.43919226  2.61298094]
Minimum f3(x) = 165.49239  achieved at x = [ 2.55899658  0.60946144
1.88237885  1.81071205  2.19672863  1.48309219
  1.41649549  2.62005546  1.60402418  1.90396446  1.5900092
2.8656585
  2.96811491  0.77553425  1.5203393   0.61598803  1.16736738
1.99506984
  3.06606253  1.43453602  1.94815135  0.70264599  1.31436712
1.50489671
  1.82306319  2.87120768  1.69123453  2.42867332  2.04208607
1.01649189
  2.64520661  0.70705001  1.85596108  1.31039797  1.48272988
1.33885313
  1.48084255  1.36912785  2.35154559  0.45521084  0.96689838
1.96175742
  1.48034407  2.06894293 -0.26169026  1.43572355  2.40126264
2.61939372
  0.77481759  1.77048614]
Minimum f3(x) = 173.95974  achieved at x = [2.03635988 2.10145383
1.80480453 1.05759347 1.94923546 2.71400155
 0.2537265  0.61162253 0.53434944 0.43060628 2.64543115 2.33844502
 2.37507542 1.73830377 2.70918901 2.68557833 0.92317942 1.50087176
 1.56917457 2.52418284 1.82333659 2.13376626 0.66056391 0.39713062
 0.82825249 2.0845565  1.79290782 1.6494678  2.92756225 0.34387307
 2.55852698 1.50953471 2.6418519  1.34200799 2.60754239 2.22826944
 1.51042398 1.05250553 3.14906916 2.28404429 1.86613884 1.25558059
 0.49498076 1.53008976 0.32322428 2.39646626 2.06696997 2.35666781
 1.2110071  0.72216238]
Minimum f3(x) = 205.81927  achieved at x = [ 1.61548857  2.03924175
```

```
2.3888218   2.0991592   1.75285526  2.46612711
  3.79322262  1.65097691  1.57094164  1.11136846  2.50560433
1.83439352
  1.36655395  2.58185984  2.62884049  1.55284168  1.95052587
1.97970845
  2.02916326  1.76049862  1.99144226  2.63107168  1.21695694
2.50430347
  2.3075088   2.73114647  2.10127572  0.80995586  1.35384445
2.71695701
  0.83721784  2.36230863  1.52405668  3.142525    0.90743414
1.38905479
  2.21671788  1.89056247  2.28734828  1.03351532  1.80700018
1.6407201
  1.99766119 -0.30605024  1.73076787  1.38321663  2.90728155
1.95340688
  1.79563062  2.08636012]
```




```
Minimum f3(x) = 167.25786  achieved at x = [2.1734146  1.78417892
1.60350888 2.62714471 1.56238379 0.69708094
 1.39209693 1.68245042 1.62788922 0.95250815 1.44809306 1.87183768
 2.38286444 1.2824333  2.00692979 1.6832918  1.95374378 1.99239786
 1.13719286 1.44509768 2.26790033 1.73639814 1.3251136  1.43759851
 1.53234257 2.67291654 2.98738525 1.20360121 1.74021718 2.0369728
 2.12495512 2.11742221 1.18340519 1.8270383  1.93335359 1.53499652
 2.03059325 1.98458381 1.19209617 1.54210962 1.39979417 2.215417
 1.64641344 1.96732913 1.63853918 2.16446201 1.89496123 2.35988467
 1.79110296 1.94132439]
Minimum f3(x) = 162.33038  achieved at x = [1.98932245 0.72555091
```

```
1.65414654 0.90100233 1.85740269 0.65470607
 1.42188283 1.30287564 2.38329086 1.26187722 2.0017334  1.65883145
 2.02497443 0.75606293 1.3965972  2.15764366 1.99570734 1.02392771
 1.86557226 2.36836136 1.68604174 1.12676235 1.95496296 2.99300981
 2.27001657 1.3741507  1.494396   1.73176397 1.3078968  2.36650585
 1.18685939 1.1594144  1.88143246 1.47535756 2.62910813 2.87520994
 2.21443601 1.95339606 1.79088788 2.35233965 1.28832211 2.10985615
 1.42895719 1.73801067 1.35637201 1.59127453 2.12815721 1.3623027
 1.85817462 2.07410573]
Minimum f3(x) = 167.02078  achieved at x = [1.90288089 1.63553018
2.08250405 1.03431809 1.94043895 1.36435469
 1.87133622 1.92685918 1.83297734 1.32381868 1.75255292 1.5676945
 2.56594962 1.73269714 2.51222342 1.27386695 1.71217677 1.69501787
 2.03707468 1.92618737 0.96068303 1.53691837 1.44374688 2.46462123
 2.07100433 2.28862327 0.99341371 2.72899044 1.74433016 2.14131949
 2.22748944 3.10413625 1.33787913 1.01959603 1.4938031  1.17923471
 2.16112908 0.94455505 2.24818892 1.39318644 1.71739457 2.21137031
 1.50781825 1.48746081 2.05581657 0.95333914 1.8799182  1.75378977
 2.31737565 0.75484273]
Minimum f3(x) = 169.69833  achieved at x = [1.85548872 1.43821466
1.94851301 2.37424293 2.27913008 1.98394283
 0.91714728 1.65377405 2.21598492 1.43702684 1.384577   1.43067077
 2.52963124 2.4223389  2.04073315 1.55740266 1.47797544 1.00630972
 2.24824842 1.43262418 1.71587126 0.80507314 2.20878556 2.29803518
 1.51959575 2.51565875 1.89735633 1.0105143  1.66702361 1.52574143
 2.33730797 1.92532563 1.05738194 1.60190786 2.24988929 1.54870147
 1.58829875 1.35800111 1.64133029 1.8198938  2.25860224 2.1565696
 2.10912456 2.23885079 1.72114178 1.98616944 2.36982923 1.41493167
 1.95134072 1.26299923]
Minimum f3(x) = 164.95082  achieved at x = [1.58125146 1.86775631
1.37468477 2.0629966  2.35175968 2.350798
 2.0656696  1.92849404 1.93531075 1.14032383 1.26381078 1.40957562
 1.9753348  2.30330088 1.88079186 1.44064113 2.54500735 2.32589371
 1.18388043 1.27606862 1.41705812 2.41690221 1.7763086  1.87052079
 1.49599588 1.92902536 2.19576073 0.57477298 1.68129051 1.64262071
 2.56848256 1.59340612 0.72904866 0.35932312 0.55751256 1.05597577
 2.07346377 1.5662746  1.27532367 1.97686578 2.40898351 1.86106872
 2.15411215 2.00476699 2.08733774 1.29577731 2.21602451 2.31637029
 1.79160886 1.81915508]
```



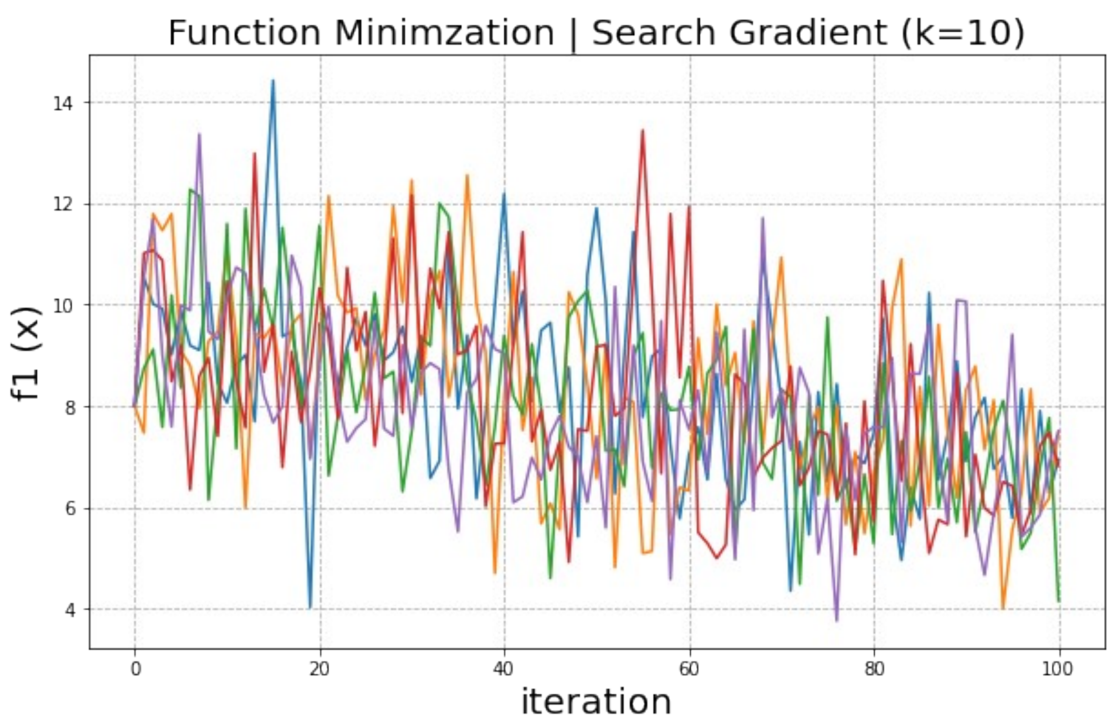
Question 1.5: Perform SG with two sample sizes, k = 10 and k = 50, for each function. Perform 5 runs for each function and each sample size. Plot the average of the function values of all in each iteration for each function. Overall 6 plots (each plot should show 5 random trajectories). Note: If you see the gradient update diverging (leading to very large values), you can use the normalized gradient $\frac{\nabla f(x)}{\|\nabla f(x)\|_2}$ in each update instead of just $\nabla f(x)$ (so that after multiplying with the learning rate, each update step is guaranteed to be small in magnitude), but you don't have to do this and you can just show the divergence behavior as well.

```
# Search Gradient
def search_gradient(x, f, max_itr=100, alpha=1e-2, sample_size=10):
    n = x.shape[0]
    f_val = np.zeros((max_itr+1,1))
    x_val = np.zeros((max_itr+1,n))

    f_val[0,0] = f(x)
    x_val[0,:] = x.reshape(n)

    x_mean = np.zeros((n,1)) + x
    x_cov = np.eye(n)

    i = 0
    while i < max_itr:
        x_samples = np.random.multivariate_normal(x_mean.reshape(n),
x_cov, size=(sample_size))
```

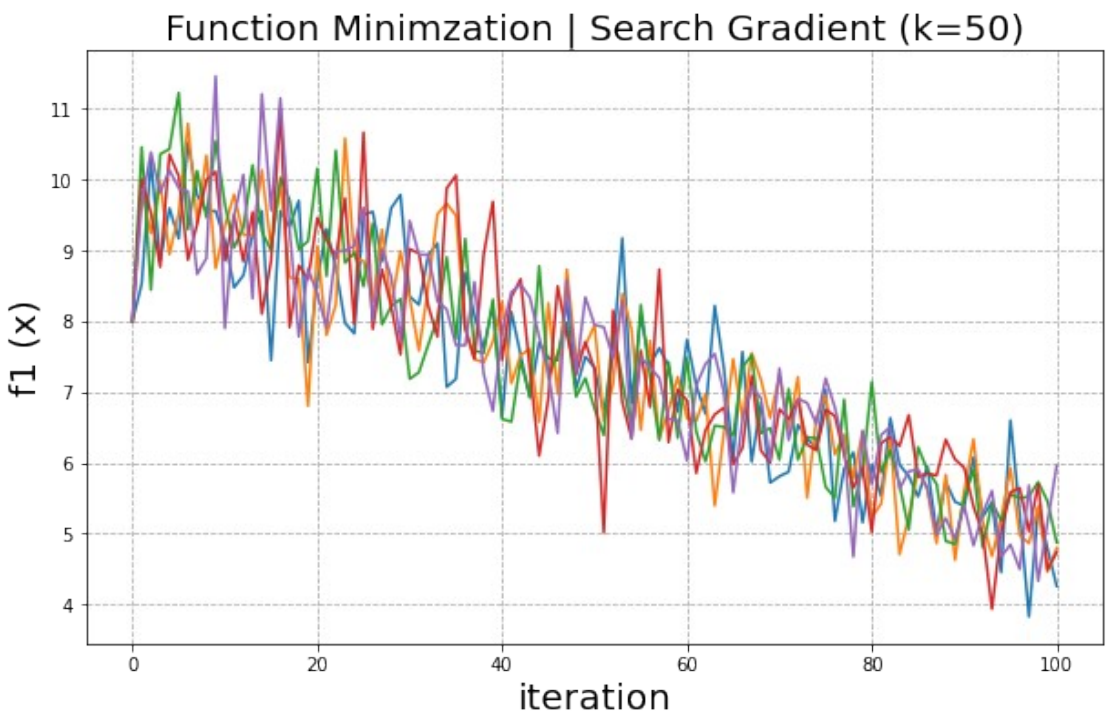
```
        f_val_sample = np.zeros(sample_size)

        x_cov_inv = np.linalg.inv(x_cov)

        sum_grad_mean = np.zeros((n,1))
        sum_grad_cov = np.zeros((n,n))

        for j in range(sample_size):
            x_sample = np.reshape(x_samples[j,:], (n,1))
            f_val_sample[j] = f(x_sample)

            term0 = x_sample - x_mean
            grad_mean_log_prob = np.matmul(x_cov_inv, term0)

            term1 = np.matmul(term0, term0.T)
            grad_cov_log_prob = -0.5*x_cov_inv +
0.5*np.matmul(np.matmul(x_cov_inv, term1), x_cov_inv)

            sum_grad_mean = sum_grad_mean + grad_mean_log_prob *
f_val_sample[j]
            sum_grad_cov = sum_grad_cov + grad_cov_log_prob *
f_val_sample[j]

        avg_grad_mean = sum_grad_mean / sample_size
        avg_grad_cov = sum_grad_cov / sample_size

        # x_mean = x_mean - alpha * avg_grad_mean
        # x_cov = x_cov - alpha * avg_grad_cov

        x_mean = x_mean - alpha * avg_grad_mean /
np.linalg.norm(avg_grad_mean)
        x_cov = x_cov - alpha * avg_grad_cov /
np.linalg.norm(avg_grad_cov)

        f_val[i+1,0] = np.mean(f_val_sample)
        x_val[i+1,:] = x_mean.reshape(n)

        i = i+1

    return x_val, f_val

# Search Gradient for f1(x)
sample_size_lst = [10, 50]

for sample_size in sample_size_lst:
    fig = plt.figure()
    num_runs = 5
    for run in range(num_runs):
        max_itr = 100
```

```
        alpha = 1e-2
        x = np.ones((2,1))*2
        x_val, f1_val = search_gradient(x, f1, max_itr, alpha,
sample_size)

        print("Minimum f1(x) =", round(f1_val[max_itr,0], 5), "
achieved at x =", x_val[max_itr,:], "for sample K = ", sample_size)
        plt.plot(range(max_itr+1), f1_val)

    plt.title('Function Minimzation | Search Gradient
(k='+str(sample_size)+')')
    plt.xlabel('iteration')
    plt.ylabel('f1 (x)')
    plt.grid(linestyle = '--')
    # plt.legend()
    plt.show()
```

```
Minimum f1(x) = 6.94584  achieved at x = [1.48344689 1.48256931] for
sample K =  10
Minimum f1(x) = 7.48176  achieved at x = [1.48766152 1.48835024] for
sample K =  10
Minimum f1(x) = 4.15254  achieved at x = [1.45343188 1.52843913] for
sample K =  10
Minimum f1(x) = 6.82201  achieved at x = [1.49230278 1.48675488] for
sample K =  10
Minimum f1(x) = 7.50071  achieved at x = [1.55201341 1.43176069] for
sample K =  10
```



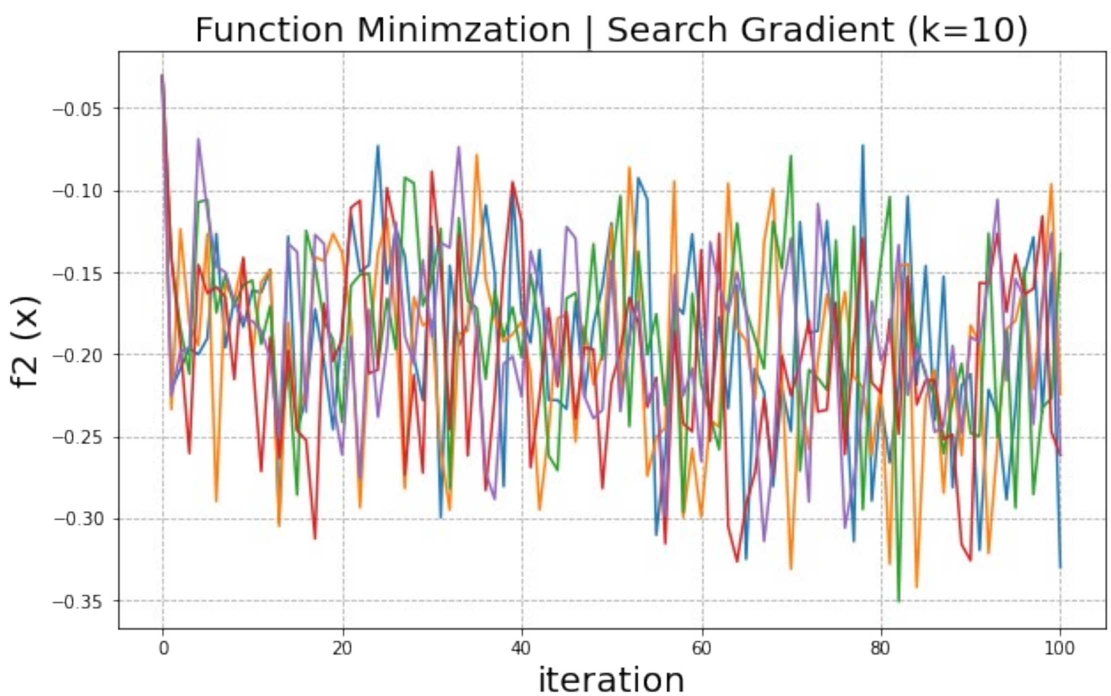
```
Minimum f1(x) = 4.26101  achieved at x = [1.33666092 1.3174997 ] for
sample K =  50
Minimum f1(x) = 4.79825  achieved at x = [1.31890165 1.3462616 ] for
sample K =  50
Minimum f1(x) = 4.87649  achieved at x = [1.30881772 1.34054485] for
sample K =  50
Minimum f1(x) = 4.73295  achieved at x = [1.35397537 1.34825937] for
sample K =  50
Minimum f1(x) = 5.95958  achieved at x = [1.3386613  1.32273145] for
sample K =  50
```



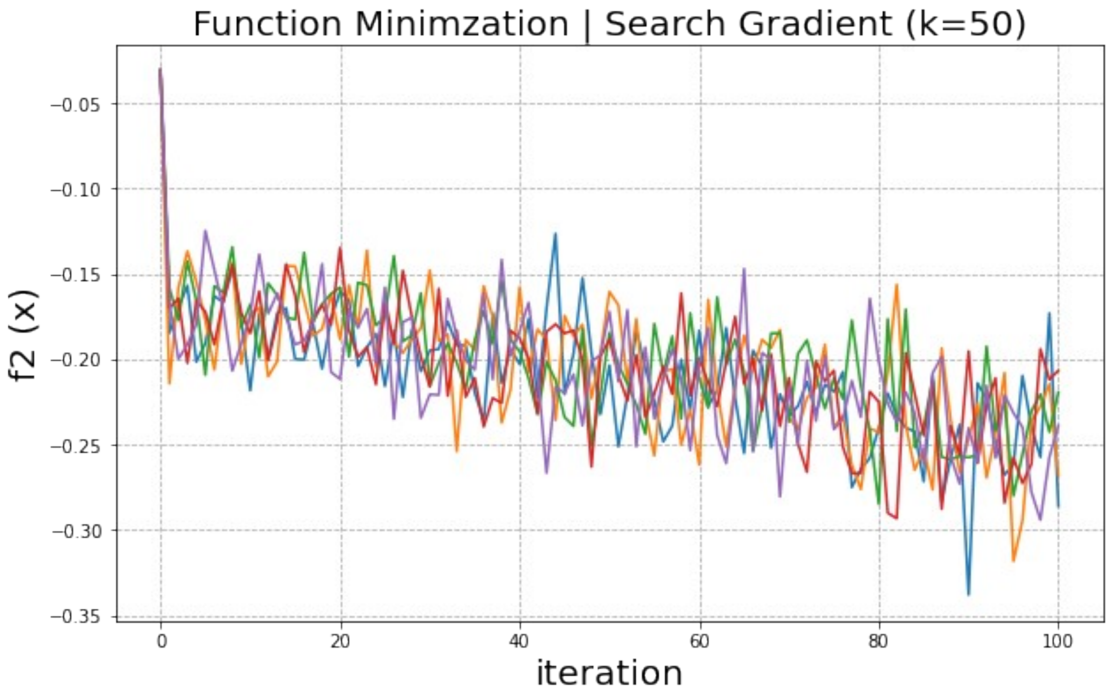

```
# Search Gradient for f2(x)
sample_size_lst = [10, 50]

for sample_size in sample_size_lst:
    fig = plt.figure()
    num_runs = 5
    for run in range(num_runs):
        max_itr = 100
        alpha = 1e-2
        x = np.ones((2,1))*2
        x_val, f2_val = search_gradient(x, f2, max_itr, alpha,
sample_size)

        print("Minimum f2(x) =", round(f2_val[max_itr,0], 5), "
achieved at x =", x_val[max_itr,:], "for sample K = ", sample_size)
        plt.plot(range(max_itr+1), f2_val)
```

```
    plt.title('Function Minimzation | Search Gradient 
(k='+str(sample_size)+')')
    plt.xlabel('iteration')
    plt.ylabel('f2 (x)')
    plt.grid(linestyle = '--')
    # plt.legend()
    plt.show()
```

```
Minimum f2(x) = -0.3299  achieved at x = [1.71837125 1.57490994] for 
sample K =  10
Minimum f2(x) = -0.22437  achieved at x = [1.60419381 1.60172484] for 
sample K =  10
Minimum f2(x) = -0.13864  achieved at x = [1.70156977 1.64593567] for 
sample K =  10
Minimum f2(x) = -0.26169  achieved at x = [1.66843345 1.67275641] for 
sample K =  10
Minimum f2(x) = -0.26055  achieved at x = [1.68174541 1.71504184] for 
sample K =  10
```



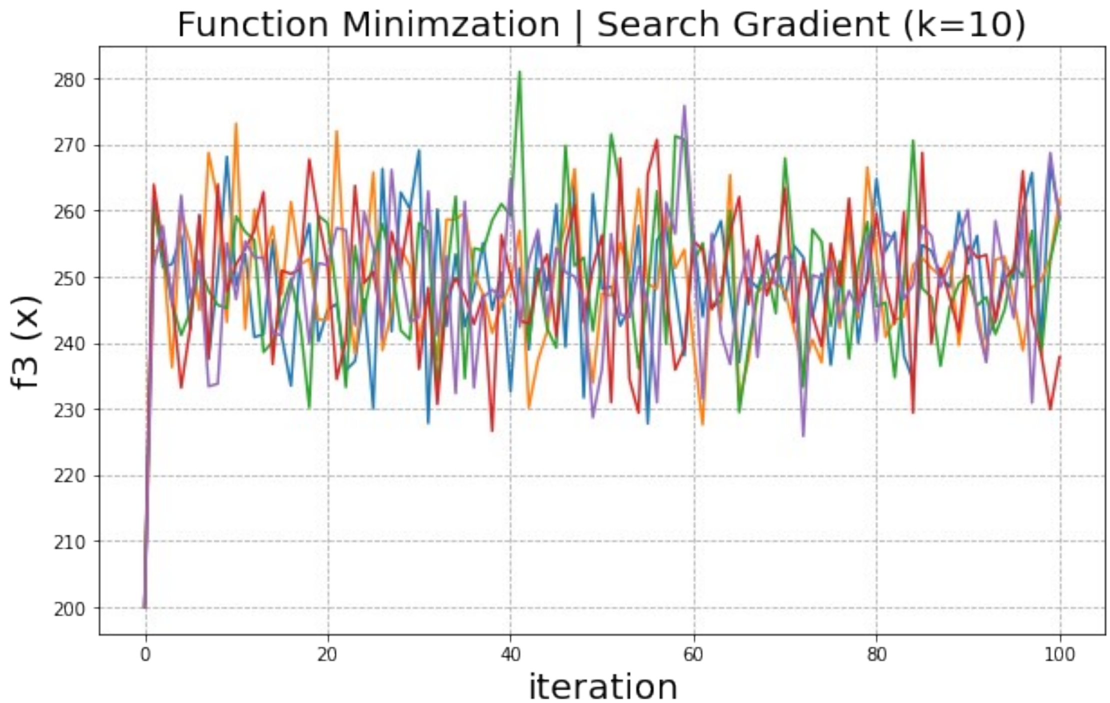

```
Minimum f2(x) = -0.28602  achieved at x = [1.37202753 1.36544579] for 
sample K =  50
Minimum f2(x) = -0.26834  achieved at x = [1.44743191 1.38699145] for 
sample K =  50
Minimum f2(x) = -0.2193  achieved at x = [1.43512879 1.38078511] for 
sample K =  50
Minimum f2(x) = -0.20658  achieved at x = [1.3734341  1.41802188] for 
sample K =  50
Minimum f2(x) = -0.2389  achieved at x = [1.37035065 1.43914808] for 
sample K =  50
```




```
# Search Gradient for f3(x)
sample_size_lst = [10, 50]

for sample_size in sample_size_lst:
    fig = plt.figure()
    num_runs = 5
    for run in range(num_runs):
        max_itr = 100
        alpha = 1e-2
        x = np.ones((50,1))*2
        x_val, f3_val = search_gradient(x, f3, max_itr, alpha,
sample_size)

        print("Minimum f3(x) =", round(f3_val[max_itr,0], 5), "
achieved at x =", x_val[max_itr,:])
        plt.plot(range(max_itr+1), f3_val)

    plt.title('Function Minimzation | Search Gradient
(k='+str(sample_size)+')')
    plt.xlabel('iteration')
    plt.ylabel('f3 (x)')
    plt.grid(linestyle = '--')
    # plt.legend()
    plt.show()
```

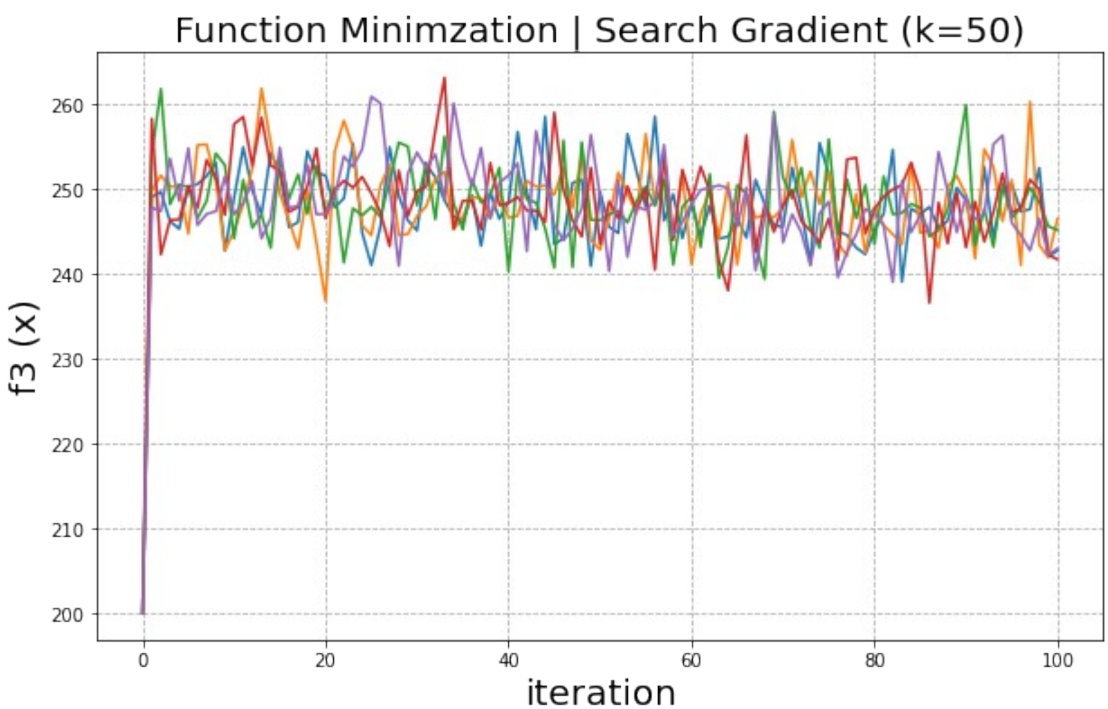
```
Minimum f3(x) = 259.20551  achieved at x = [1.99950174 1.98097569
2.01366563 2.00528528 2.01358678 2.01694479
 1.99331778 1.97406474 1.99698797 2.01483631 1.99447338 2.00080464
 1.98066    1.99580145 1.98659856 1.97772305 2.00735347 1.98196328
```

```
 1.99122544 1.97711868 1.99275782 1.97194613 2.00448674 1.97420492
 1.9872727  2.00182175 2.01254717 1.98301133 1.99854829 1.99813451
 2.00248956 1.99496205 1.98877314 1.99070522 1.98097995 2.02336367
 2.00860241 1.98321315 1.98767981 1.97874684 1.99829187 1.94833243
 1.9726905  1.98463684 2.0098373  2.00958261 1.99066422 2.0002155
 1.98874447 1.99017329]
Minimum f3(x) = 261.65771  achieved at x = [2.01806225 2.00835307
1.97049603 2.00204639 1.99497025 1.98836377
 1.98536605 1.9834054  1.98171479 1.97595565 1.98325893 1.9920094
 1.97906803 2.00716064 1.98013194 1.99272906 2.02170444 2.00242018
 1.98347848 1.9936883  2.02640707 1.98764785 1.99945107 1.98962571
 1.96471981 2.00884158 1.98086184 1.983111   1.99469815 2.00788601
 2.01470537 2.0041889  2.00264907 2.01787329 1.99787556 1.9898554
 1.97117645 2.01001429 1.98310167 1.98792732 2.00007483 1.98447644
 1.97811504 1.98993188 1.98453522 1.99042988 1.99025857 1.98336382
 1.9951765  1.97650666]
Minimum f3(x) = 258.94925  achieved at x = [2.00418687 1.98669529
1.97505552 1.99614799 1.96655304 1.98268475
 1.98709874 1.98956134 1.99605864 2.0082983  1.98385765 2.00231475
 2.00153032 1.96705534 1.99758804 2.00334409 1.9882772  1.99117882
 1.98047009 1.97200081 1.95659679 1.99000901 1.99201739 2.00082797
 1.9692355  1.98599691 2.01397294 1.99822449 1.99361543 1.98866925
 1.99251567 1.99787121 2.01218748 2.01648077 2.00894688 1.96211876
 1.99398885 1.98813494 2.01102731 2.00724769 2.00740479 1.98908762
 2.01162003 1.99046849 1.99362857 2.01947522 1.96546114 1.99763842
 1.9973375  2.00045452]
Minimum f3(x) = 237.81452  achieved at x = [2.00763095 2.017734
2.01565682 1.99573959 1.96638438 1.99394687
 1.98846157 1.94877255 1.99992897 1.98280063 1.9839583  1.98184396
 1.99522196 2.00609886 2.02595134 1.99942263 1.98246476 1.99125732
 1.9846396  2.00833004 1.98712069 1.98708151 1.98768962 2.01467376
 2.015406   2.00360826 1.99444559 2.0022847  1.97445626 1.99317992
 1.98074302 1.98807785 2.00452715 2.01538808 2.00500526 1.99475648
 1.97826322 1.97951504 1.97594576 1.99307799 2.00907904 1.98195116
 1.9968929  1.98840021 2.00637721 1.98469909 1.99383511 2.0075852
 1.97363745 2.03943692]
Minimum f3(x) = 258.77337  achieved at x = [2.00476193 2.00794813
2.00665017 1.9746425  2.00060216 1.96309189
 1.98722012 2.00248058 2.01817198 2.00279909 2.00775952 1.98979244
 1.98666171 1.9861519  1.98309771 1.99524278 2.01453142 1.98039407
 1.98593836 1.99063248 2.02165211 2.0003229  1.9982847  2.01177174
 1.9813581  1.97308203 2.00936768 1.98913415 2.00218557 1.98677691
 1.99291798 2.02071724 1.98969029 1.99713532 1.99788339 1.976361
 2.00784615 1.98392924 2.00472168 2.00273643 1.97558815 1.97647332
 1.97741158 1.97287516 1.9897923  1.99966767 1.99125293 1.98895835
 1.99984215 1.99467522]
```




```
Minimum f3(x) = 242.733  achieved at x = [1.99652663 1.9917651
1.98355186 1.99371003 1.99003276 1.97022334
 1.99619319 1.97785205 1.97773479 1.99917238 2.00422    2.00964696
 1.99286033 1.97165351 1.97238777 1.98088988 2.01322003 1.97397618
 1.99641408 1.97792992 1.99180849 1.97916313 1.97994293 1.96853359
 2.01016434 1.97643304 2.006377   1.98024949 1.99228229 2.00792702
 1.96863248 1.99394603 1.98570451 1.98471488 1.98695565 1.98235748
 1.97895914 2.00329865 2.00509134 1.97669394 2.00156292 1.98815779
 1.98084779 1.97473043 1.97944216 1.98633819 2.00471112 1.97838663
 1.98320968 1.97775158]
Minimum f3(x) = 246.46605  achieved at x = [1.97602288 2.00074517
1.98735664 1.98283891 1.98966927 1.97560167
 2.00440438 1.97166067 1.99254989 1.98070689 1.96378036 1.98457696
 1.98974432 1.98329363 1.94533432 1.99846982 2.01622555 1.99064625
 1.99343377 1.99664998 1.98389072 1.9934411  1.99526044 1.99610102
 1.95470293 1.98510956 1.99630195 1.99962872 1.96878487 1.96210208
 1.96977349 1.97687281 1.98410222 1.99279858 1.97897406 1.97466988
 1.99553016 1.96345292 1.96510914 1.98650239 1.99626219 1.98078765
 2.00670074 1.98869212 1.99934085 2.0003545  1.98214214 1.97997414
 2.02683479 1.98372281]
Minimum f3(x) = 245.1545  achieved at x = [1.99163432 1.96680777
1.9768877  1.97010156 1.96287484 1.97568318
 1.96084965 1.99913618 1.99455718 1.97816074 1.98474166 2.00022899
 1.96830795 2.00617221 2.00985692 1.98707213 2.00311769 1.97456629
 1.98540888 1.97449095 1.97334531 1.98126917 1.98555812 2.00374103
 1.99574152 1.98706647 1.98115527 1.94979903 1.97622027 1.98797957
 1.99937986 2.00153462 1.97955957 1.96999135 2.00077353 1.99694052
 1.96023631 1.9837894  1.99355201 1.963231   1.98142415 1.96356238
```

```
 1.98133112 2.02232435 1.96912503 1.99131292 1.95136518 1.97092156
 1.98795613 2.00920158]
Minimum f3(x) = 241.69159  achieved at x = [1.97941087 1.96389774
1.97691219 1.97757411 1.96365377 1.965073
 1.96718485 1.99560734 1.97955914 1.96922336 1.98031583 1.96263533
 1.99689011 1.9484872  1.96475847 1.99602365 1.98871174 1.99252205
 1.96232316 1.98102012 1.97089626 1.97095364 1.97558478 1.98351193
 1.98425108 1.98030377 1.98062092 1.9625135  1.96446294 1.98951932
 1.9719991  1.98482955 1.98017681 1.98696937 1.99090565 1.9751156
 1.9933118  1.98665155 1.9875397  1.97521578 2.00384721 1.96952712
 1.98419942 1.98983402 1.96100558 2.0173857  1.96763524 1.98532455
 1.97663349 1.98828542]
Minimum f3(x) = 242.97562  achieved at x = [1.98749371 2.00533337
1.99938743 1.95962251 1.97917364 1.96041394
 1.97627179 1.9917269  2.00503313 1.97787845 2.00239933 2.00608798
 1.96970678 1.9825984  1.98674498 1.97846234 1.98983276 1.99895951
 1.98572719 1.98024584 1.99893862 1.9664765  1.97228508 1.98675292
 1.9825635  1.97705284 1.97105329 1.99456404 1.98230401 1.976226
 1.98707343 1.97440165 1.98158945 1.9823634  1.98183436 1.98118334
 1.98525138 1.97817047 2.00691002 1.98529187 1.98467442 1.98899167
 1.97452937 1.96796384 1.97922625 1.96714029 1.95043926 1.98246267
 1.98686876 1.96925373]
```



1 Question 1

Question 1.6: Based on the performance of these algorithms with different parameters on different types of functions, summarize some intuition about how to choose among these algorithms and the parameters. For instance, when the function dimension is large, is it better to sample or find gradient? What are the different trends of the algorithms for different sample sizes? Do some algorithms give more stable behavior than others? There is no standard answer here, just reflect a bit on how you should choose from the different strategies.

Answer: Functions $f_1(x_1, x_2)$ and $f_3(x)$ are both convex functions of similar nature but in different dimensional space (N = 2 and N = 50 resp.). Therefore, as evident from the gradient descent minimization plots of these two functions, they follow similar behaviour. For example, the gradient descent minimization for f_1 and f_3 follow the same curve over 100 iterations except for the function values due to extra dimensions in f_3.

In simulated annealing, a random Δx is being sampled from a normal distribution $\mathcal{N}(0, I)$ at each iteration, and based on function values at $x + \Delta x$ and acceptance probability, the value of x is updated to the new value $x + \Delta x$. From the plots of simulated annealing, we can infer that $T = 10$ performs better compared to $T = 1000$ (probabilistically) in most cases. This can be understood in terms of acceptance probability.

$$P(\text{accept}|f(x + \Delta x) > f(x)) = \exp\left(-\frac{f(x + \Delta x) - f(x)}{T}\right) \tag{1}$$

For the same value of $(f(x + \Delta x) - f(x))$, higher values of T leads to greater probability of accepting higher function values at $x + \Delta x$ compared to lower values of T. In simulated annealing, acceptance of higher function values can be seen as exploration. This can lead to divergence and larger number of iterations to converge to the minimum. Therefore, for T=10, the SA algorithm performs better compared to for T=1000 (in probability sense).

Cross-Entropy method takes K random samples around the mean of a distribution (multivariate gaussian in our case) and finds elite samples from the K samples where the function values are lower compared to non-elite samples. Using the elite samples, it updates the parameters of sampling distribution using maximum likelihood. Intuitively, large number of samples should provide more information about the function and therefore, better successive parameter updates. From the cross-entropy plots, $K = 50$ performs better compared to $K = 10$ for the three functions.

Search Gradient method is similar to cross-entropy method except instead of updating the distribution parameters based on function values at elite samples, the sampling distribution parameters are updated using gradient descent. In our case, the mean and covariance parameters of multivariate gaussian distribution are updated using gradient descent updates. Again, following the same reasoning as cross-entropy methods, larger samples should provide more information regarding the gradient direction (which is computed using maximum likelihood) and thus better successive updates in the long run. From the plots, K=50 performs better than K=10 for f_1 and f_2.

For a 50-dim convex function $f_3(x)$ after 100 iterations
Minimum using GD = 3.517
Minimum using SA (T=10) = 181 (median of 5 runs)
Minimum using CE (K=50) = 167 (median of 5 runs)
Minimum using SG (K=50) = 242 (median of 5 runs)

From this we can infer that in case of convex function in high-dimensional case, it is better to use gradient methods than random sampling.

For a 2-dim drop-wave function $f_2(x)$ after 100 iterations
Minimum using GD = -0.369
Minimum using SA (T=10) = -0.778 (median of 5 runs)
Minimum using CE (K=50) = -0.6915 (median of 5 runs)
Minimum using SG (K=50) = -0.23 (median of 5 runs)

For non-convex functions, GD may lead to only local minima but sampling based methods can lead to global minima. For ex, here SA is performing better than the rest.

Stability: From the plots, GD and CE based methods are much more stable compared to SA and SG based methods.

Question 2: Proof of Dijkstra Algorithm Separation Property

Proof: Consider a graph G with N nodes and positive cost edges. Let the start node be S_{00} and its adjacent neighboring nodes be denoted by $\mathcal{N}(S_{00}) = \{S_{11}, S_{12}, ..., S_{1n_1}\}$. The objective of Dijkstra algorithm is to find the shortest path of each node in G from the start node S_{00}. Initially, all the nodes are assigned very high distance values (say $d = \infty$) except for the start node S_{00}, for which $d = 0$.

We initialize an empty set for shortest-path tree set (aka explored set) $S_E(0)$, an empty frontier set $S_F(0)$, and an unexplored set $S_U(0)$ consisting of nodes in G not present in either $S_E(0)$ or $S_F(0)$. Initially, all nodes are contained in $S_U(0)$. The argument 0 here represents iteration number.

Iteration - 1
$S_E(1) = \{S_{00}\}$ (since it is the only node with shortest distance $d = 0$ in the unexplored set $S_U(0)$)
$S_F(1) = \mathcal{N}(S_{00}) = \{S_{11}, S_{12}, ..., S_{1n_1}\}$ (the distances of adjacent nodes of S_{00} are updated in the first iteration).
$S_U(1) =$ all the nodes in G not present in $S_E(1)$ or $S_F(1)$.
For iteration=1, any acyclic path from the only state in explored set $S_E(1) = \{S_{00}\}$ to any state in unexplored set $S_U(1)$ has to pass through some state in frontier set $S_F(1)$ since there is only a single node in $S_E(1)$ and all its neighboring nodes are present in $S_F(1)$ which connect it to the unexplored nodes $S_U(1)$ in G. Thus, the separation property holds after the end of iteration-1 update.

Iteration - 2
First, a node with minimum distance from start node S_{00} is selected from the frontier set $S_F(1)$ (say S_{1j_1}) and added to explored set $S_E(1)$.
$S_E(2) = \{S_{00}, S_{1j_1}\}$
Let the adjacent nodes of S_{1j_1} be the set of nodes $\mathcal{N}(S_{1j_1}) = \{S_{21}, S_{22}, ..., S_{2n_2}\}$. (NOTE: some or all of the nodes in $\mathcal{N}(S_{1j_1})$ can be also present in $\mathcal{N}(S_{00})$).
Now, the distances of nodes in $\mathcal{N}(S_{1j_1}) - S_E(2)$ are updated based on whether their previous assigned distance is greater than the distance of S_{1j_1} + cost of corresponding edges. After the end of iteration-2 update,
$S_E(2) = \{S_{00}, S_{1j_1}\}$
$S_F(2) = \mathcal{N}(S_{00}) \bigcup \mathcal{N}(S_{1j_1}) - S_E(2)$
$S_U(2) =$ all the nodes in G not present in $S_E(2)$ or $S_F(2)$
Any neighboring node of S_{00} and S_{1j_1} is present in the union of explored set and frontier set $S_E(2) \bigcup S_F(2)$. Therefore, any acyclic path from any node in explored set $S_E(2)$ to any node in unexplored set $S_U(2)$ has to pass through some node in the frontier set $S_F(2)$. Separation Property holds for iteration-2.

Iteration - K
Let us assume that the separation property holds after the end of iteration K.
Let $S_E(K) = \{S_{00}, S_{1j_1}, S_{2j_2}, ..., S_{(K-1)j_{K-1}}\}$

$S_F(K) = \{\mathcal{N}(S_{00}) \bigcup \mathcal{N}(S_{1j_1}) \bigcup \mathcal{N}(S_{2j_2}) \bigcup ... \bigcup \mathcal{N}(S_{(K-1)j_{K-1}})\} - S_E(K)$ (excluding nodes in $S_E(K)$)
$S_U(K)$ = all nodes in G not present in either of $S_E(K)$ or $S_F(K)$

Iteration - (K+1)
The node in $S_F(K)$ with minimum assigned distance from start node S_{00} (say S_{Kj_K}) is found and added to the explored set $S_E(K)$.
$S_E(K+1) = \{S_{00}, S_{1j_1}, S_{2j_2}, ..., S_{(K-1)j_{K-1}}, S_{Kj_K}\}$
For all adjacent nodes of $S_{Kj_K} \in S_F(K) \bigcup S_U(K)$, their distances from start node S_{00} can be updated. After the update at the end of iteration (K+1),
$S_F(K+1) = S_F(K) \bigcup \mathcal{N}(S_{Kj_K}) - S_E(K+1)$
Due to our assumption in iteration-K, the only path from any node in $S_E(K)$ to any node in unexplored set $S_U(K)$ is through some node in frontier set $S_F(K)$. This implies that all adjacent nodes of $S_E(K)$ are present in $S_E(K) \bigcup S_F(K)$ only.

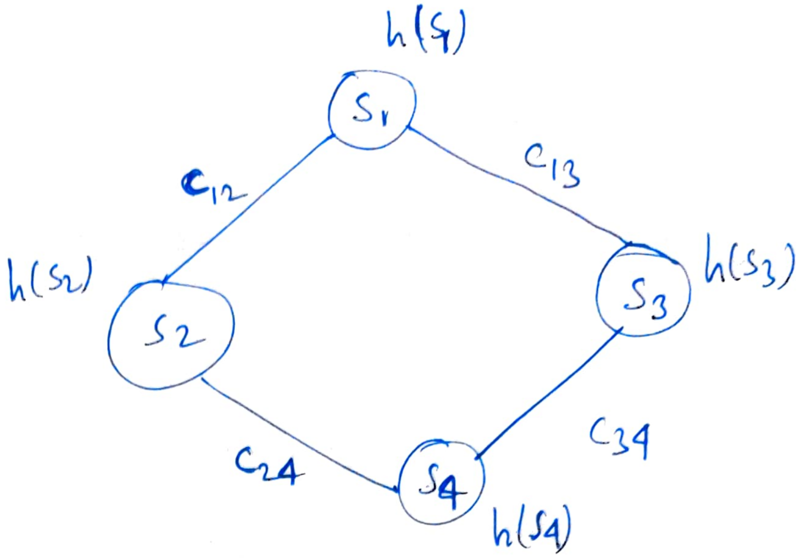
All the adjacent nodes of $S_E(K+1)$ are present in the set $S_E(K) \bigcup S_F(K) \bigcup \mathcal{N}(S_{Kj_K})$ which is equal to the set $S_E(K+1) \bigcup S_F(K+1)$.

$\mathcal{N}(S_E(K+1)) \in S_E(K+1) \bigcup S_F(K+1) \implies \mathcal{N}(S_E(K+1)) \in S_E(K) \bigcup S_F(K+1) \bigcup \{S_{Kj_K}\}$ and $\mathcal{N}(S_{Kj_K}) \in S_F(K+1)$
This means, any acyclic path from the explored set $S_E(K+1)$ to the unexplored set $S_U(K+1)$ has to pass through the frontier set $S_F(K+1)$ at some point. Thus, the separation property holds for iteration-(K+1) assuming it holds for iteration-K.

We assumed that the separation property holds for iteration-K and proved it to be true for iteration-(K+1). Therefore, by mathematical induction, separation property holds for any iteration.

2. Classical Search on Graphs

Question 3:



$S_1 \rightarrow$ start node

$S_4 \rightarrow$ goal node

$S_2, S_3 \rightarrow$ intermediate nodes.

$h(S) \rightarrow$ heuristic function

$h(S_4) = 0$ (Property of $h(S)$ for goal node)

C_{ij} are the edge weights or cost.

For consistency following ~~equations~~ inequalities need to be satisfied.

1. $h(S_1) \leq h(S_2) + c_{12}$
2. $h(S_1) \leq h(S_3) + c_{13}$
3. $h(S_2) \leq h(S_4) + c_{24} = c_{24}$
4. $h(S_3) \leq h(S_4) + c_{34} = c_{34}$.

Let $c_{12} = 2.2$
$c_{13} = 1.5$
$c_{24} = 1$
$c_{34} = 1.4$

$$\left.\begin{aligned} h(S_1) &= 1 \\ h(S_2) &= 0.3 \\ h(S_3) &= 0.5 \\ h(S_4) &= 0 \end{aligned}\right\} \text{Consistent}$$

Total cost from S_1 to S_4 along $S_1 \rightarrow S_2 \rightarrow S_4 = h(S_1) + c_{12} + h(S_2) + c_{24}$
$= 4.5$

Total cost from S_1 to S_4 along $S_1 \rightarrow S_3 \rightarrow S_4 = h(S_1) + c_{13} + h(S_3) + c_{34}$
$= 4.4$

Thus, $S_1 \rightarrow S_3 \rightarrow S_4$ is shortest path compared to $S_1 \rightarrow S_2 \rightarrow S_4$

Now, Let $h'(S) = 5h(S)$

$h'(S_1) = 5$ $\quad h'(S_2) = 1.5$ $\quad h'(S_3) = 2.5$ $\quad h'(S_4) = 0$

→ Edge weights are unchanged.

Now Consistency equations:

$$
\left.
\begin{array}{ll}
① & h'(S_1) \le h'(S_2) + C_{12} \\
② & h'(S_1) \le h'(S_3) + C_{13} \\
③ & h'(S_2) \le C_{24}
\end{array}
\right\} \text{Not satisfied.} \Downarrow \text{Inconsistent.}
$$

④ $h'(S_3) \le C_{34}$

Total cost from S_1 to S_4 along $S_1 \to S_2 \to S_4$ $= h'(S_1) + C_{12} + h'(S_2) + C_{24}$

$= 9.7$

Total cost from S_1 to S_4 along $S_1 \to S_3 \to S_4$ $= h'(S_1) + C_{13} + h'(S_3) + C_{34}$

Shortest path: $S_1 \to S_2 \to S_4$ $\qquad = 10.4$

Thus, multiplying $h(S)$ by 5 makes the graph inconsistent and misguides the search from $S_1 \to S_3 \to S_4$ to $S_1 \to S_2 \to S_4$.

1 Question 4

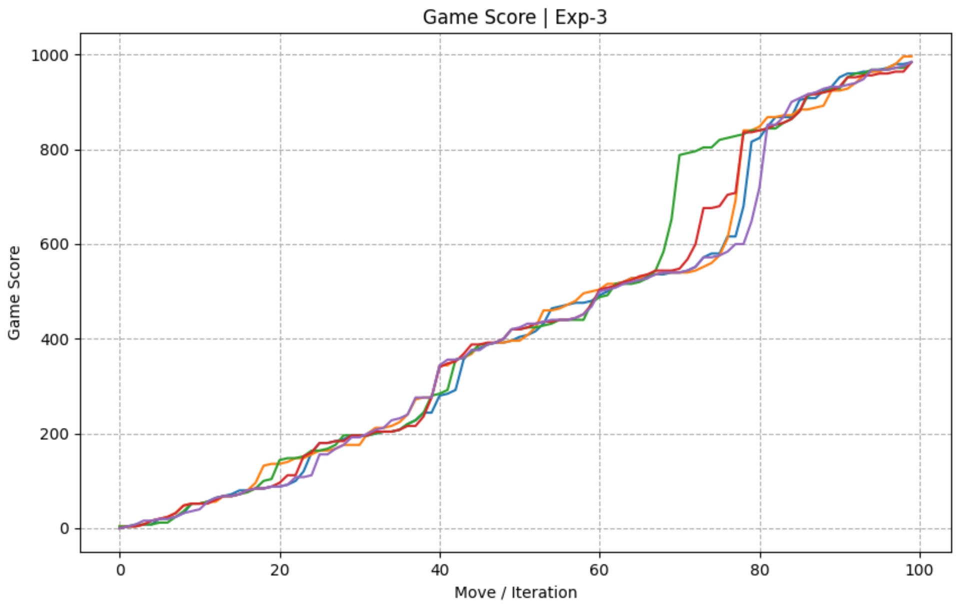


Figure 1: Exp-3 Performance; Game Score for 100 iterations and 5 different algorithm runs

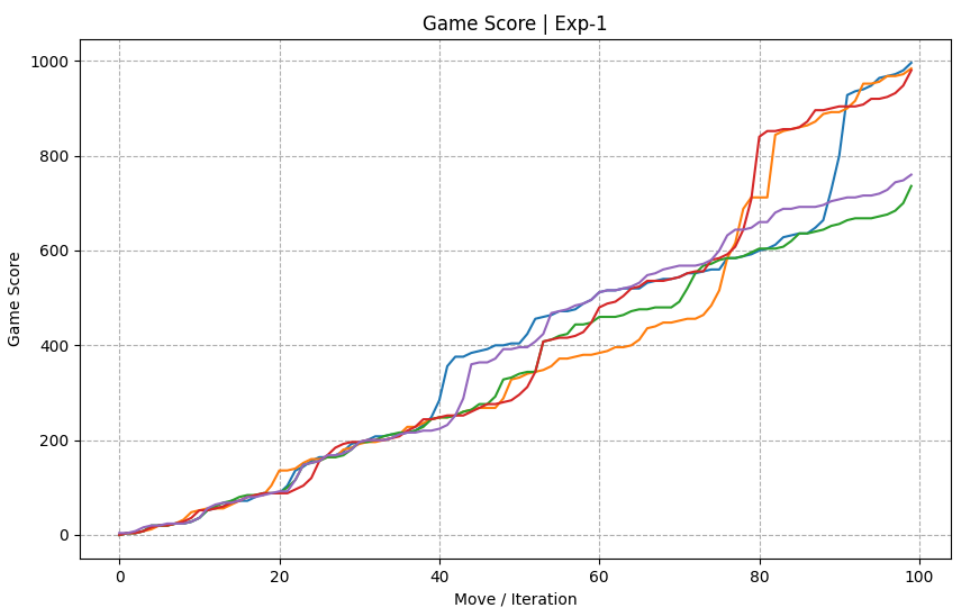


Figure 2: Exp-1 Performance; Game Score for 100 iterations and 5 different algorithm runs

2 Question 5

Design a different evaluation function for Exp-3 to perform better than Exp-3 with the previous evaluation function that simply uses the original game score. You can use any information from the game state, such as highest tile, pattern of the existing tiles, etc. Design your plots to show that the new algorithm (i.e. Exp-3 with the evaluation function you designed) is better than the original Exp-3.

Solution: For designing custom heuristic function, I created a "custom_expectimax" function inside "ai.py". This function has the same logic for the MAX_PLAYER and the CHANCE_PLAYER as the original "expectimax" function. The difference is in the heuristic score returned in case of leaf / terminal nodes. In original "expectimax" function, the returned score is the actual game score. A heuristic score should take into account how good any particular position (tile matrix) is to continue playing from, whereas the game score returned by the game engine only signifies the points collected in the game. It does not tells us how good a particular tile matrix is for the AI player. I have tried to use a weighted combination of several heuristics to lead the optimization problem towards better tile position and thus higher game scores. The following heuristics are useful:

1. Maximize the number of open / empty tiles
2. Maximize the value of the highest tile in the 4x4 tile matrix
3. Maximize the number of potential merges i.e, increasing the chance of adjacent equal value tiles helps in increasing the score as well as freeing up open tiles
4. Maximize the game score provided by the game engine
5. Monotonicity: This heuristic ensures that the values of the tiles are all either increasing or decreasing along the horizontal and vertical directions.
6. Using snake-shaped 4x4 weight matrix that multiplies each cell in the tile matrix. This helps to ensure that highest valued tiles are in the corners which increases the chances of potential merges with lower valued tiles.

While performing experiments with different heuristic metric, I realized that it requires quite a lot of efforts to tune the weights of different metrics.

Number of Successful Passes ($\geq$ 20000) on Test-2 using Custom Heuristics Score = 2 (out of 10) (refer Figure 3)

References:

1. https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048
2. https://stackoverflow.com/questions/26762846/2048-heuristic-unexpected-results
3. Composition of Basic Heuristics for the game 2048

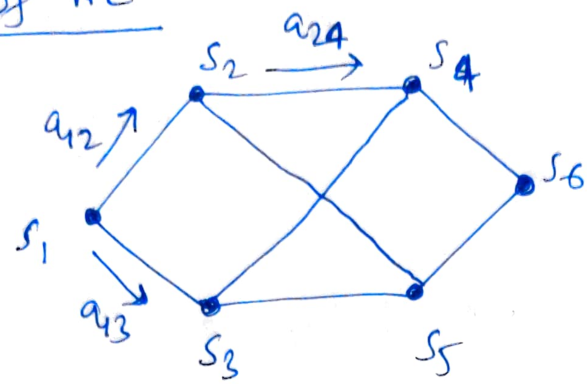
```
siriusA@Barkat-MacAir : ~/Desktop/UCSD/courses/Fall_2021/CSE257/assignments/assignment2/code/expectimax-main
$ python3 main.py -t 2
Note: each test may take a while to run.
Test 1/10:
	Score/Best Tile: 7372/512
	NOT SUFFICIENT (score less than 20000)
Test 2/10:
	Score/Best Tile: 7408/512
	NOT SUFFICIENT (score less than 20000)
Test 3/10:
	Score/Best Tile: 7616/512
	NOT SUFFICIENT (score less than 20000)
Test 4/10:
	Score/Best Tile: 23864/2048
	SUFFICIENT
Test 5/10:
	Score/Best Tile: 27276/2048
	SUFFICIENT
Test 6/10:
	Score/Best Tile: 14812/1024
	NOT SUFFICIENT (score less than 20000)
Test 7/10:
	Score/Best Tile: 11740/1024
	NOT SUFFICIENT (score less than 20000)
Test 8/10:
	Score/Best Tile: 6824/512
	NOT SUFFICIENT (score less than 20000)
Test 9/10:
	Score/Best Tile: 15740/1024
	NOT SUFFICIENT (score less than 20000)
Test 10/10:
	Score/Best Tile: 14768/1024
	NOT SUFFICIENT (score less than 20000)
FAILED (less than 4 passes)
```

Figure 3: Performance on Test-2 using custom heuristic score (2/10 successful pass)

```
siriusA@Barkat-MacAir : ~/Desktop/UCSD/courses/Fall_2021/CSE257/assignments/assignment2/code/expectimax-main
$ python3 main.py -t 2
Note: each test may take a while to run.
Test 1/10:
	Score/Best Tile: 12616/1024
	NOT SUFFICIENT (score less than 20000)
Test 2/10:
	Score/Best Tile: 7408/512
	NOT SUFFICIENT (score less than 20000)
Test 3/10:
	Score/Best Tile: 26060/2048
	SUFFICIENT
Test 4/10:
	Score/Best Tile: 14716/1024
	NOT SUFFICIENT (score less than 20000)
Test 5/10:
	Score/Best Tile: 7312/512
	NOT SUFFICIENT (score less than 20000)
Test 6/10:
	Score/Best Tile: 16424/1024
	NOT SUFFICIENT (score less than 20000)
Test 7/10:
	Score/Best Tile: 14812/1024
	NOT SUFFICIENT (score less than 20000)
Test 8/10:
	Score/Best Tile: 6824/512
	NOT SUFFICIENT (score less than 20000)
Test 9/10:
	Score/Best Tile: 15880/1024
	NOT SUFFICIENT (score less than 20000)
Test 10/10:
	Score/Best Tile: 14988/1024
	NOT SUFFICIENT (score less than 20000)
FAILED (less than 4 passes)
```

Figure 4: Performance on Test-2 using custom heuristic score (1/10 successful pass)

④ Basics of RL



→ Time spent in each state $= c(s_i)$ ← random variable.

Goal: To find good policy that minimize the expected total time from each state to arrival state s_6

$T_\pi(s_i)$ = Expected total time from state s_i to s_6 (airport).

→ Actions a_{ij} are deterministic, takes from s_i to s_j.

Question 6: Deterministic Policy

$\pi(s_1) = a_{12}$ $\quad \pi(s_2) = a_{24}$ $\quad \pi(s_4) = a_{46}$ $\quad \pi(s_3) = a_{35}$ $\quad \pi(s_5) = a_{56}$

$\pi(s_6) =$ None.

Overall Expected Time $T_\pi(s_1) = E_\pi\left[\text{Total time taken from } s_1 \text{ to } s_6 \text{ under policy } \pi\right]$

$$= E_\pi\left[c(s_1) + c(s_2) + c(s_4) + c(s_6)\right]$$

Because of the policy π, only a single route exists from s_1 to s_6 which is $s_1 \to s_2 \to s_4 \to s_6$. Each state s_i represents a segment. with time spent represented by random variable $c(s_i)$. Assuming the airport to be at the end of segment/state s_6, the total time taken would also include $\cancel{c(s_6)}$ $c(s_6)$.

$$T_\pi(s_1) = E_\pi[c(s_1)] + E_\pi[c(s_2)] + E_\pi[c(s_4)] + E_\pi[c(s_6)]$$

<u>Question 7:</u>

Minimal total travel time starting from $s_2 = T(s_2)$

" " " " " " $s_3 =$ $T(s_3)$

$T(s_1, a_{12}) \rightarrow$ total travel time starting at s_1 with action a_{12}

$$T(s_1, a_{12}) = \underbrace{\text{time taken } (s_1)}_{c(s_1)} + \underbrace{\text{time taken } (s_2 \rightarrow s_6)}_{T(s_2)}$$

$$= E[c(s_1)] + T(s_2)$$

$T(s_1, a_{13}) \rightarrow$ total travel time starting at s_1 with action a_{13}

$$T(s_1, a_{13}) = \underbrace{\text{time taken } (s_1)}_{c(s_1)} + \underbrace{\text{time taken } (s_3 \rightarrow s_6)}_{T(s_3)}$$

$$= E[c(s_1)] + T(s_3).$$

Optimal total travel time starting at s_1 $T(s_1)$

$$T(s_1) = \min(T(s_1, a_{12}), T(s_1, a_{13}))$$

minimum function.

Question 8: Given good estimates of $T(s_1)$

$$\text{(}s_1\text{)} \xrightarrow{a_{13}} s_3$$

$\hookrightarrow$ delay due to construction.

$$C(s_1)\Big|_{\text{today}} > E[C(s_1)]$$

Assumption: $T(s_3)$ does not changes.

~~Best~~ Let updated value of $T(s_1, a_{13})$ be $\hat{T}(s_1, a_{13})$

Time taken $(s_1 \to s_3 \to s_6) = T(s_3) + C(s_1)$

$$\hat{T}(s_1, a_{i3}) = T(s_1, a_{13}) + \alpha\left[T(s_3) + C(s_1) - \overset{T(s_1)}{\cancel{T(s_1, a_{13})}}\right]$$

$$= T(s_1, a_{13}) + \alpha\left[T(s_3) + C(s_1) - \overset{T(s_1)}{\cancel{T(s_3) - E[C(s_1)]}}\right]$$

$$\cancel{\hat{T}(s_1, a_{13}) = T(s_1, a_{13}) + \alpha\left[C(s_1) - E(C(s_1))\right]}$$

Due to the delay, the new updated will be larger than previous estimated value $T(s_1, a_{i3})$. ~~$\alpha \in (0$~~, $\boxed{\alpha > 0.}$