Course: Math 535 - Mathematical Methods in Data Science (MMiDS) - Spring 2023

Author: Rick Moe, University of Wisconsin-Madison

Spectral Cut Optimization

Here we will see a minor optimization that can be made to the implementation of the spectral cut algorithm which reduces the time complexity from $O(n^3)$ to $O(n^2)$.

Initial Algorithm

Below you will find the implementation of the spectral cut algorithm presented in class and on the homework.

In [ ]:
```python
# Python 3
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt
import networkx as nx
import time
```

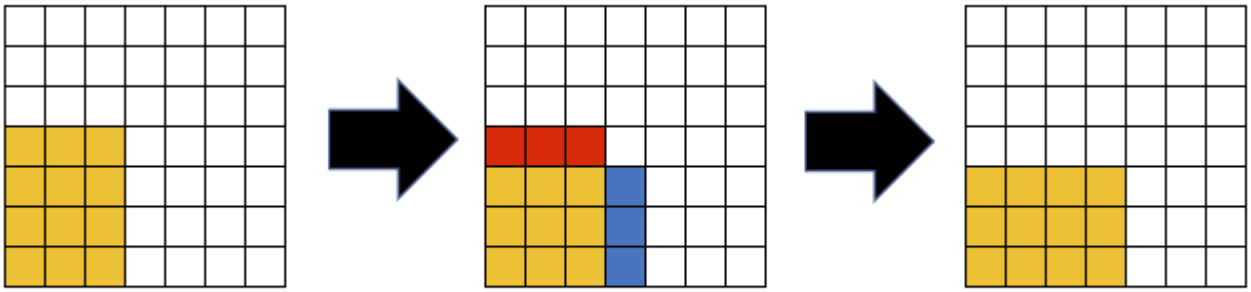
In [ ]:
```python
def cut_ratio(A, order, k):
    n = A.shape[0] # number of vertices
    edge_boundary = 0 # initialize size of edge boundary
    for i in range(k): # for all vertices before cut
        for j in range(k,n): # for all vertices after cut
            edge_boundary += A[order[i],order[j]] # add one if {i,j} in E
    denominator = np.minimum(k, n-k)
    return edge_boundary/denominator

def spectral_cut(A):
    n = A.shape[0] # number of vertices

    # laplacian
    degrees = A.sum(axis=1)
    D = np.diag(degrees)
    L = D - A

    # spectral decomposition
    w, v = LA.eigh(L)
    order = np.argsort(v[:,np.argsort(w)[1]]) # index of entries in increasing order

    # cut ratios
    phi = np.zeros(n-1) # initialize cut ratios
    for k in range(1,n):
        phi[k-1] = cut_ratio(A, order, k)
    imin = np.argmin(phi) # find best cut ratio
    return order[0:imin+1], order[imin+1:n]
```

The time complexity of the 'cut_ratio' function above is $O(nk)$ because of the need to sum all elements of the reordered matrix from rows k to n and columns 1 to $k-1$ for a total of $(k-1)(n-k) \in O(nk)$. The 'spectral_cut' function calls 'cut_ratio' n times, varying k from 1 to n, making the time complexity $O(n^3)$. Although we will not be changing the core algorithm, we can speed up the algorithm considerably by keeping track of information from previous calls to 'cut_ratio'.

Explanation of Optimization

As it stands, 'cut_ratio' does not utilize any information from previous iterations. Thus, to compute the edge boundary, it must compute the whole sub-matrix sum at each iteration. However, we can see that some elements of this sum must be added in many different iterations (i.e. the bottom left corner is added at each iteration). Furthermore, we can see that the edge boundary computed at each iteration sums a lot of the same elements as the last iteration. Thus, instead of initializing the edge boundary to zero and adding each element, we could consider initializing it to the last edge boundary and then compute the new sum.

Say that we had the sum of a matrix from rows k to n and columns 1 to $k-1$. We could then compute the sum of the matrix from rows $k+1$ to n and columns 1 to k by subtracting the elements from the k-th row from indices 1 to $k-1$ then adding the elements of the k-th column from indices $k+1$ to n. This would then give us the submatrix sum needed in the $k+1$-th iteration. Furthermore, we are only subtracting $k-1$ elements from a row and adding $n-k$ elements from a column at each iteration, meaning the submatrix sum can be computed in linear time.



Here is an examle of a iteration of this process to transform the sum at iteration k into the sum at iteration k+1

This reduction in time complexity of 'cut_ratio' from $O(nk)$ to $O(n)$ leads to a reduction in the runtime of 'spectral_cut' from $O(n^3)$ to $O(n^2)$.

The code is nearly identical, although we now track the last edge boundary in both functions and we change the process to compute the new edge boundary so that it uses the above optimization. We can also note that setting the initial edge boundary to 0 means that the first calculation will simply add zero to the first column (aside from the first element) and subtract nothing since there are no elements from indices 1 to $k-1$ if $k=1$.

In [ ]:
```
def cut_ratio_opt(A, order, k, last_eb=0):
    n = A.shape[0] # number of vertices
    edge_boundary = last_eb # initialize size of edge boundary
    # Subtractive
    for j in range(0, k-1):
        edge_boundary -= A[order[k-1], order[j]]
    # Additive
    for i in range(k, n):
        edge_boundary += A[order[i], order[k-1]]
    denominator = np.minimum(k, n-k)
    return edge_boundary/denominator, edge_boundary

def spectral_cut_opt(A):
    n = A.shape[0] # number of vertices

    # laplacian
    degrees = A.sum(axis=1)
    D = np.diag(degrees)
    L = D - A

    # spectral decomposition
    w, v = LA.eigh(L)
    order = np.argsort(v[:,np.argsort(w)[1]]) # index of entries in increasing order

    # cut ratios
    last_eb = 0
    phi = np.zeros(n-1) # initialize cut ratios
    for k in range(1,n):
        phi[k-1], last_eb = cut_ratio_opt(A, order, k, last_eb)
    imin = np.argmin(phi) # find best cut ratio
    return order[0:imin+1], order[imin+1:n]
```

Testing on Homework Graph

Loading the "COVID_PPI.txt" file from the homework and running both algorithms yields impressive results.

In [ ]:
```
# Initialize graph
G = nx.Graph()

# Load COVID_PPI.txt edgelist
path = "COVID_PPI.txt"
edgelist = list(np.genfromtxt(path, delimiter=",", dtype=str))

for e in edgelist:
    G.add_edge(e[0], e[1])

G0 = G.subgraph(sorted(nx.connected_components(G), key=len, reverse=True)[0])
```

In [ ]:
```
A = nx.adjacency_matrix(G0).toarray()
print(f'n = {len(A)}')
```

```
n = 1536
```

In [ ]:
```python
# Tests the runtime of the passed function
def test_runtime(function, *args, iterations=10):
  t_start = time.time()
  output = ()
  for _ in range(iterations):
    output = function(*args)
  avg_time = (time.time() - t_start) / iterations
  return *output, avg_time

# Checks two output arrays for equality
def check_output(out1, out2, printMatching=True):
  if len(out1) != len(out2):
    print(f'{len(out1)} != {len(out2)}')
  else:
    diff = [(out1[i], out2[i]) for i in range(len(out1)) if out1[i] != out2[i]]
    if len(diff) > 0:
      print(diff)
    elif printMatching:
      print("Matching")
```

In [ ]:
```python
s, sc, avg_time = test_runtime(spectral_cut, A, iterations=3)
print(f'Initial Runtime: \t{avg_time} s')
s_opt, sc_opt, avg_time_opt = test_runtime(spectral_cut_opt, A, iterations=30)
print(f'Optimized Runtime: \t{avg_time_opt} s')
check_output(s, s_opt)
check_output(sc, sc_opt)
```

```
Initial Runtime:        217.7725942929586 s
Optimized Runtime:      1.727420194943746 s
Matching
Matching
```

Clearly, the optimized algorithm performs much better than the initial algorithm on this data set. We also see that both functions return the exact same result suggesting the optimization does not affect the correctness of the algorithm.

Further Testing

We can construct a few more functions to easily compare the runtimes of these two algorithms on randomized graphs of various sizes. To stay consistent with the intended usage of this algorithm, we will generate simple, undirected graphs meaning the adjacency matrix will be symmetric. We also need to ensure that the graph is connected, which we will accomplish by connecting each node i to node $i + 1$ for the purposes of this problem.

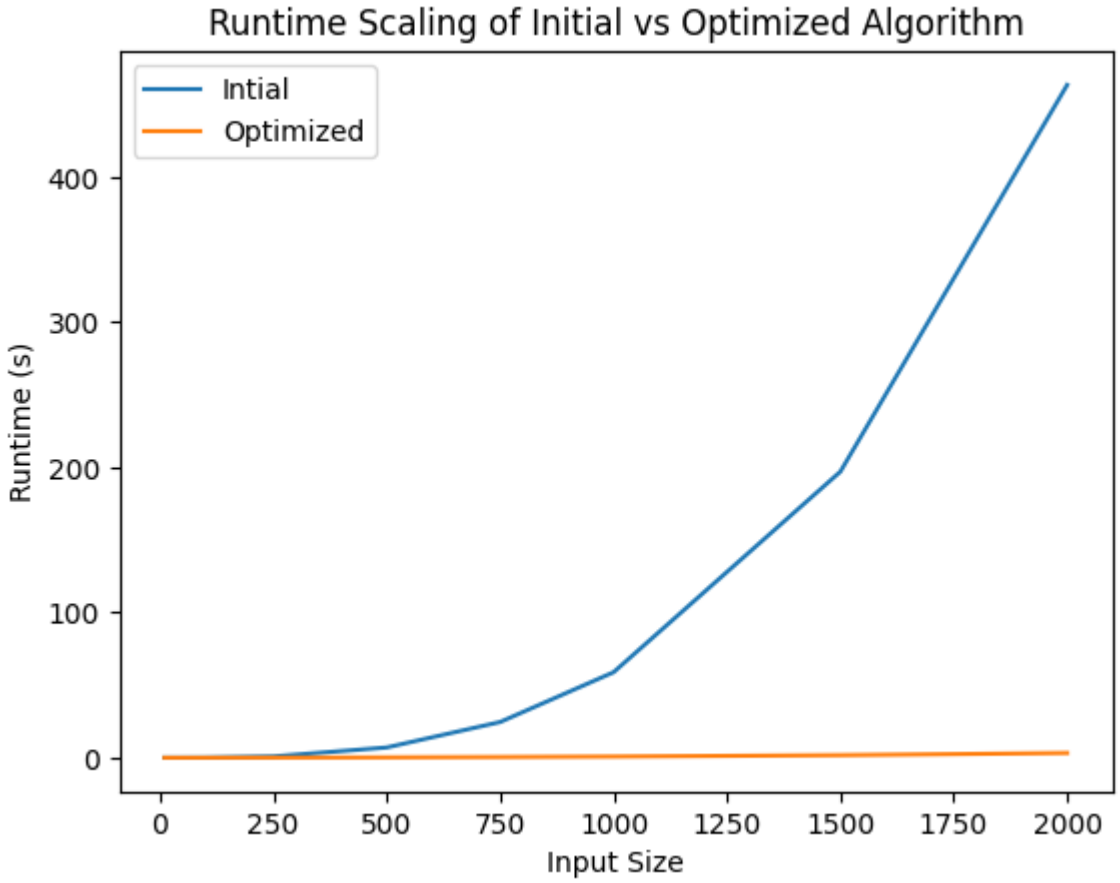
In [ ]:
```python
# Generates a connected, symmetric adjacency matrix
def gen_matrix(size, edge_chance=.5):
  A = np.zeros((size, size))
  for i in range(size - 1):
    for j in range(i + 1, size):
      # Connect each i to i + 1 to ensure connectedness
      if i + 1 == j:
        A[i][j] = 1
        A[j][i] = 1
      # Randomly determine if edge should be added
      elif np.random.rand() <= edge_chance:
        A[i][j] = 1
        A[j][i] = 1
  return A

# Runs both functions for a given number of iterations and computes the runtimes
def compare_runtimes(size, iterations=10, edgeChance=.5):
  A = gen_matrix(size, edgeChance)
  s, sc, avg_time = test_runtime(spectral_cut, A, iterations=iterations)
  print(f'Initial Runtime   (n={size}):\t{avg_time} s')
  s_opt, sc_opt, avg_time_opt = test_runtime(spectral_cut_opt, A, iterations=iterations)
  print(f'Optimized Runtime (n={size}):\t{avg_time_opt} s')
  check_output(s, s_opt, printMatching=False)
  check_output(sc, sc_opt, printMatching=False)

  return avg_time, avg_time_opt
```

We can then use these methods to test the runtimes of both functions on a variety of different matrix sizes to compare how the algorithms scale with the size of the input.

In [ ]:
```python
test_sizes = [10, 25, 50, 100, 250, 500, 750, 1000, 1500, 2000]
initial_times, opt_times = [], []
for size in test_sizes:
  # Scale iterations between 5 and 30 based on the cube of size
  iterations = max(5, min(30, int(1e9 / size ** 3)))
  initial_time, opt_time = compare_runtimes(size, iterations)
  initial_times.append(initial_time)
  opt_times.append(opt_time)
```

```
Initial Runtime   (n=10):       0.000364534060160319 s
Optimized Runtime (n=10):       0.00029919942220052085 s
Initial Runtime   (n=25):       0.0020952622095743814 s
Optimized Runtime (n=25):       0.0009965101877848308 s
Initial Runtime   (n=50):       0.010813347498575846 s
Optimized Runtime (n=50):       0.003168177604675293 s
Initial Runtime   (n=100):      0.06498509248097738 s
Optimized Runtime (n=100):      0.015164494514465332 s
Initial Runtime   (n=250):      0.8561938762664795 s
Optimized Runtime (n=250):      0.05401879946390788 s
Initial Runtime   (n=500):      7.031945377588272 s
Optimized Runtime (n=500):      0.19424688816070557 s
Initial Runtime   (n=750):      24.56968445777893 s
Optimized Runtime (n=750):      0.4281795024871826 s
Initial Runtime   (n=1000):     58.87911434173584 s
Optimized Runtime (n=1000):     0.7732016563415527 s
Initial Runtime   (n=1500):     197.0830846309662 s
Optimized Runtime (n=1500):     1.7257779598236085 s
Initial Runtime   (n=2000):     463.54153833389285 s
Optimized Runtime (n=2000):     3.220724630355835 s
```

In [ ]:
```
plt.plot(test_sizes, initial_times, label="Intial")
plt.plot(test_sizes, opt_times, label="Optimized")
plt.title("Runtime Scaling of Initial vs Optimized Algorithm")
plt.xlabel("Input Size")
plt.ylabel("Runtime (s)")
plt.legend(loc="upper left")
plt.show()
```



Although the algorithms perform nearly identically for small input sizes, the optimized algorithm clearly outperforms the initial algorithm for larger input sizes. This makes sense considering we improved the time complexity from $O(n^3)$ to $O(n^2)$. Besides the obvious increased speed when running the same test cases as we ran on the previous algorithm, this huge performance boost allows us to run this algorithm on much larger input sizes than we could with the inital algorithm given a reasonable amount of time to run.

Predicting Runtime With Least Squares

We can now use the calculated runtimes for given input sizes to predict the amount of time needed to run each algorithm on a given input size. We can do this using least squares. We can also add squared terms, cubic terms, and terms of other degrees to allow for the approximation of quadratic functions, cubic functions, and so on. If our theoretical calculations were correct, we should see a drastic improvement in the error of the prediction until the cubic term for the initial algorithm (as it is $O(n^3)$) but only until the quadratic term for the optimized algorithm (as it is $O(n^2)$). We can use algorithms provided in class to solve the linear least squares problem.

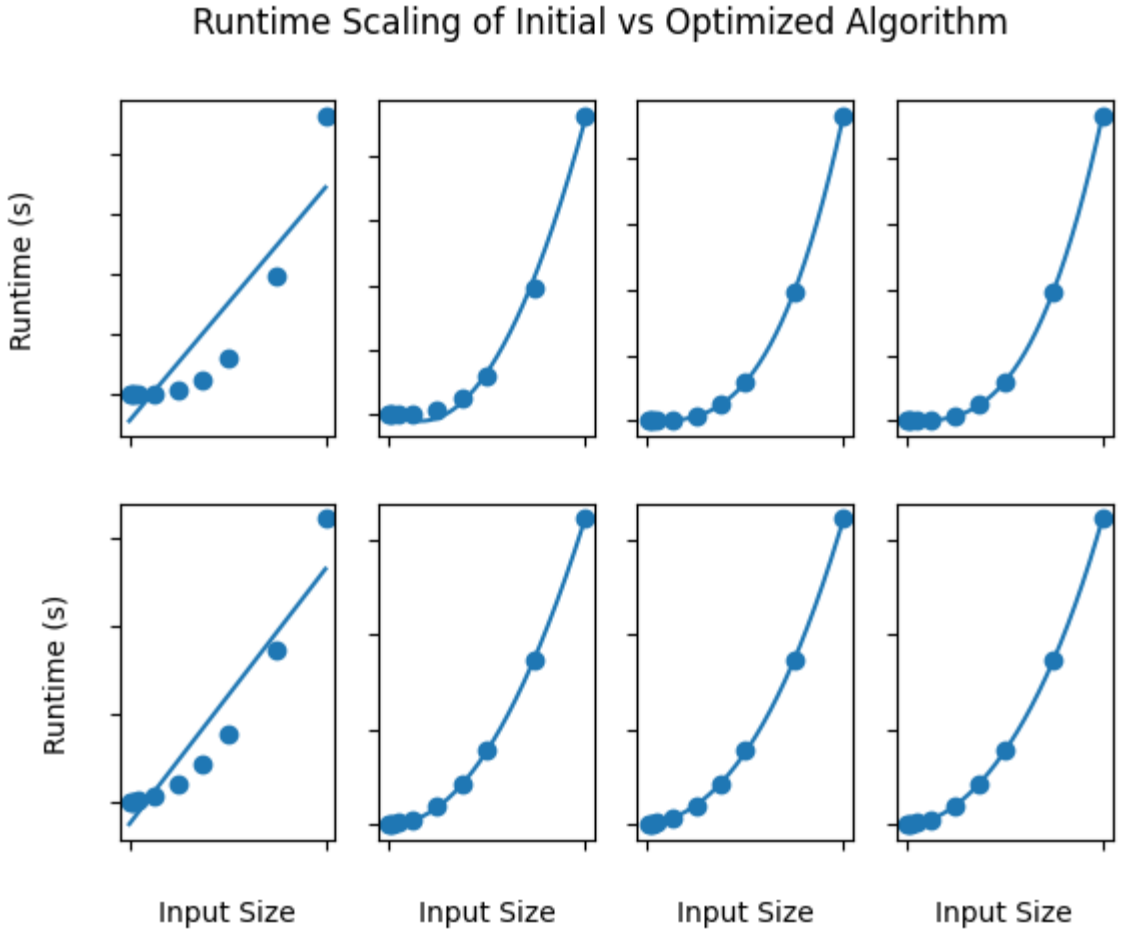
In [ ]:
```
def mmids_backsubs(U,b):
    m = b.shape[0]
    x = np.zeros(m)
    for i in reversed(range(m)):
        x[i] = (b[i] - np.dot(U[i,i+1:m],x[i+1:m]))/U[i,i]
    return x

def mmids_forwardsubs(L,b):
    m = b.shape[0]
    x = np.zeros(m)
    for i in range(m):
        x[i] = (b[i] - np.dot(L[i,0:i],x[0:i]))/L[i,i]
    return x

def mmids_cholesky(B):
    n = B.shape[0] # number of rows
    L = np.zeros((n, n)) # initiallization of L
    for j in range(n):
        L[j,0:j] = mmids_forwardsubs(L[0:j,0:j],B[j,0:j])
        L[j,j] = np.sqrt(B[j,j] - LA.norm(L[j,0:j])**2)
    return L

def ols_by_chol(A, y):
    L = mmids_cholesky(A.T @ A)
    z = mmids_forwardsubs(L, A.T @ y)
    return mmids_backsubs(L.T, z)
```

We can then construct a function to construct the matrices needed for least squares given an allowed degree for the prediction. This allows us to easily vary the degree given to the function.

In [ ]:
```
# Runs ols_by_chol using a given degree of approximation
def ols_with_degree(x, y, degree):
    A = []
    x_temp = np.ones(len(x))
    for _ in range(degree + 1):
        A.append(x_temp)
        x_temp = x_temp * x
    A = np.array(A).T
    beta = ols_by_chol(A, y)
    error = LA.norm(y - A @ beta) ** 2
    return beta, error

# Another function given in class, calculates a polynomial function
# output given an input and coefficients
def eval_polynomial(coefficients, x):
    # p = list of coefficients; p[0] = constant term
    return sum((a*x**i for i,a in enumerate(coefficients)))
```

In [ ]:
```
x = np.linspace(0, 2000, num=1000)
max_degree = 4

for deg in range(1, max_degree + 1):
  beta, error = ols_with_degree(test_sizes, initial_times, deg)
  print(f'Initial Error (deg = {deg}):\t{error}')
  y = np.zeros(len(x))
  for b in reversed(beta):
    y *= x
    y += b
  plt.subplot(2, max_degree, deg)
  plt.scatter(test_sizes, initial_times)
  plt.plot(x, y)
  plt.tick_params(labelcolor='none')
  if deg == 1:
    plt.ylabel("Runtime (s)")

for deg in range(1, max_degree + 1):
  beta, error = ols_with_degree(test_sizes, opt_times, deg)
  print(f'Optimized Error (deg = {deg}):\t{error}')
  y = np.zeros(len(x))
  for b in reversed(beta):
    y *= x
    y += b
  plt.subplot(2, max_degree, max_degree + deg)
  plt.scatter(test_sizes, opt_times)
  plt.plot(x, y)
  plt.tick_params(labelcolor='none')
  plt.xlabel("Input Size")
  if deg == 1:
    plt.ylabel("Runtime (s)")
plt.suptitle("Runtime Scaling of Initial vs Optimized Algorithm")
plt.show()
```

```
Initial Error (deg = 1):        38210.03366904726
Initial Error (deg = 2):        909.8244919259462
Initial Error (deg = 3):        0.12090101503566872
Initial Error (deg = 4):        0.07909492618762068
Optimized Error (deg = 1):      0.9458283739597976
Optimized Error (deg = 2):      0.00338621744324894
Optimized Error (deg = 3):      0.0005148249441531077
Optimized Error (deg = 4):      0.0001961551655924438
```



The error drops quickly for the first few degrees, but begins to slow down beyond degree 3 for the initial algorithm and after degree 2 for the optimized algorithm. Since the runtime will never decrease from adding a degree (the new coefficient could be zero leading to the same minimizing coefficients in the previous degree), this supports the notion that the initial algorithm has a runtime of $O(n^3)$ while the optimized algorithm has a runtime of $O(n^2)$, despite the noise in the data. We can now use the third-degree approximation of the inital runtimes and the second-degree approximation of the optimized runtimes to predict, with some degree of accuracy, how long it would take to run the algorithms on arbitrarily long input sizes.

In [ ]:
```
beta_initial, _ = ols_with_degree(test_sizes, initial_times, 3)
beta_opt, _ = ols_with_degree(test_sizes, opt_times, 2)

input_sizes = [10 ** pow for pow in range(3, 10)]
for size in input_sizes:
    pred_intial_time = eval_polynomial(beta_initial, size)
    pred_opt_time = eval_polynomial(beta_opt, size)
    print(f'Predicted Initial Runtime   (n = {size}):  \t{pred_intial_time} s')
    print(f'Predicted Optimized Runtime (n = {size}):  \t{pred_opt_time} s')
```

```
Predicted Initial Runtime   (n = 1000):         58.806790300328196 s
Predicted Optimized Runtime (n = 1000):         0.764593815838885 s
Predicted Initial Runtime   (n = 10000):        56262.68261225736 s
Predicted Optimized Runtime (n = 10000):        83.22824971952733 s
Predicted Initial Runtime   (n = 100000):       55756388.00357056 s
Predicted Optimized Runtime (n = 100000):       8400.831525777443 s
Predicted Initial Runtime   (n = 1000000):      55703114774.94571 s
Predicted Optimized Runtime (n = 1000000):      840873.5359884598 s
Predicted Initial Runtime   (n = 10000000):     55697760889476.305 s
Predicted Optimized Runtime (n = 10000000):     84095267.7508256 s
Predicted Initial Runtime   (n = 100000000):    5.569722523517837e+16 s
Predicted Optimized Runtime (n = 100000000):    8409605926.920228 s
Predicted Initial Runtime   (n = 1000000000):   5.569717166709094e+19 s
Predicted Optimized Runtime (n = 1000000000):   840961384220.7174 s
```