ncaa_region_optimizer

August 5, 2021

1 Genetic Algorithms for Region Partitioning

We will be using some Python modules installed by pip rather than Anaconda, so I must adjust the import path.

[ ]:
```
import sys
sys.path.insert( 1, '/usr/local/lib/python3.7/site-packages' )
```

Import other packages.

[2]:
```
import pandas as pd
import numpy as np
from geopy.distance import geodesic # this works only if you've done conda␣
↪install geopy -c conda-forge
import random
import statistics
from plotly_for_usa_points import usa_map # this works only if you've done␣
↪conda install plotly
# also one needs all the extensions mentioned here:
# https://github.com/plotly/plotly.py#jupyterlab-support-python-35
from tqdm.notebook import tqdm
from ga_for_partitions import optimize_partition
import math
from matplotlib import pyplot as plt
```

[3]:
```
%matplotlib inline
```

1.1 Import the data

[4]:
```
data_filename = 'wrestling-schools-data.csv'
df = pd.read_csv( data_filename )
len( df )
```

[4]: 106

[5]:
```
df.head()
```

```
[5]:    UniqueID           College/University Name                  Street  \
     0         1                    Adrian College          10 S Madison St
     1         2  Alfred State College  (add 2018)  10 Upper College Drive
     2         3                      Alma College        614 W Superior St
     3         4                          Augsburg      2211 Riverside Ave
     4         5                    Augustana (IL)             639 38th St

              City State   Latitude  Longitude  Power-1  Power-2  NCAA Asgt  \
     0      Adrian    MI  41.899337 -84.044547   2.4514    2.927        3.0
     1      Alfred    NY  42.254334 -77.789646   0.0000    0.000        NaN
     2        Alma    MI  43.380011 -84.655654   5.1091    5.941        3.0
     3 Minneapolis    MN  44.963541 -93.267835   9.6340    8.890        2.0
     4 Rock Island    IL  41.470591 -90.583733   0.0000    0.301        1.0

        ND Asgt  ND Asgt2  ND Asgt3  Purdue Asgt1  Purdue Asgt2  WSU Asgt1  \
     0      2.0       1.0       5.0           4.0           2.0        2.0
     1      NaN       NaN       NaN           5.0           7.0        4.0
     2      6.0       1.0       5.0           4.0           2.0        5.0
     3      6.0       5.0       3.0           5.0           4.0        6.0
     4      2.0       4.0       3.0           3.0           2.0        5.0

        WSU Asgt2
     0        6.0
     1        4.0
     2        6.0
     3        5.0
     4        5.0
```

1.1.1 Drop schools we don't want in this analysis

Some schools were dropped for various domain-specific reasons. See paper.

```
[6]: df = df.drop( [ 31, 61, 85 ] )
     num_schools = len( df )
     num_schools
```

```
[6]: 103
```

1.1.2 Make it easy to fetch desired rows/columns

```
[7]: def school ( key ):
         if type( key ) == int or type( key ) == np.int64:
             column = 'UniqueID'
         else:
             column = 'College/University Name'
         return df[df[column] == key].iloc[0]
```

```
( SCH_ID, SCH_NAME, SCH_ADDR, SCH_CITY, SCH_STATE, SCH_LAT, SCH_LNG, SCH_POW1,␣
↪SCH_POW2,
  SCH_NCAA, SCH_ND1, SCH_ND2, SCH_ND3, SCH_PUR1, SCH_PUR2, SCH_WSU1, SCH_WSU2 )␣
↪= \
    list( df.columns.values )
def all_ids ():
    return list( df['UniqueID'] )
def index_to_id ( index ):
    return df['UniqueID'].iloc[index]
# print( school( 2 )[SCH_NAME] )
# print( school( 'Augsburg' )[SCH_ID] )
# print( school( 50 )[SCH_LAT], get_school( 50 )[SCH_LNG] )
```

1.2 Map distance tools

Define measure for computing distance on the (curved) surface of the earth.

[8]:
```
def school_latlng ( school ):
    return ( school[SCH_LAT], school[SCH_LNG] )
```

Now pre-compute the distance between any two pair of schools and cache it in a matrix, because we'll be asking these distance questions a million times below, and this cache will speed it up a lot.

[9]:
```
school_locations = [ school_latlng( school( index_to_id( i ) ) )
                     for i in range( num_schools ) ]
distance_matrix = [ [ geodesic( school_locations[i], school_locations[j] ).
↪miles \
                      for i in range( num_schools ) ] \
                    for j in range( num_schools ) ]
def distance_lookup ( school_index1, school_index2 ):
    return distance_matrix[school_index1][school_index2]
```

1.3 Utilities for partitions

[10]:
```
num_parts_in_partition = 6
def indices_for_part_in_partition ( part_index, partition ):
    return [ i for i in range( len( partition ) ) if partition[i] == part_index␣
↪]
def schools_in_part_in_partition ( part_index, partition ):
    return [ school( index_to_id( i ) )
             for i in indices_for_part_in_partition( part_index, partition ) ]
def size_of_part_in_partition ( part_index, partition ):
    return partition.count( part_index )
def random_partition ():
```

```
    return [ random.randint( 0, num_parts_in_partition - 1 ) for i in range(␣
↪num_schools ) ]
```

[11]:
```
def print_partition ( partition ):
    for part_index in range( num_parts_in_partition ):
        schools = schools_in_part_in_partition( part_index, partition )
        powers = [ school[SCH_POW2] for school in schools ]
        print( 'Region {:1d}, {:2d} schools, mean power {:7.5f} (stdev {:7.5f}):
↪'.format(
            part_index + 1, size_of_part_in_partition( part_index, partition ),
            statistics.mean( powers ), statistics.stdev( powers ) ) )
        print( '----------------------------------------------------------' )
        centroid = (
            statistics.mean( [ school[SCH_LAT] for school in schools ] ),
            statistics.mean( [ school[SCH_LNG] for school in schools ] ),
        )
        print( '    Centroid: {:7.3f} lat, {:7.3f} lon'.format(
            centroid[0], centroid[1] ) )
        latlngs = [ school_latlng( school ) for school in schools ]
        print( '    Mean distance to centroid: {:8.3f} miles'.format(
            statistics.mean( [ great_circle( centroid, latlng ).miles for␣
↪latlng in latlngs ] )
        ) )
        for s in schools:
            print( '        {:30.30s} {:30.30s} {:>7.1f} miles'.format(
                s[SCH_NAME],
                '{}, {}, {}'.format( s[SCH_ADDR], s[SCH_CITY], s[SCH_STATE] ),
                great_circle( school_latlng( s ), centroid ).miles
            ) )
        print()
# print_partition( random_partition() )
```

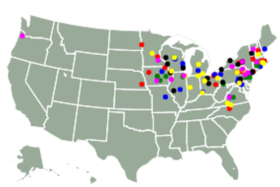
1.4 Test map-drawing tools

[12]:
```
usa_map( { 'black' : [ school_latlng( school( id ) ) for id in all_ids() ] },␣
↪'All Schools' )
```

All Schools

```
[13]: def partition_map ( schools_partition, title = 'Partition of All Schools' ):
    colors = [ 'red', 'blue', 'green', 'black', 'yellow', 'magenta', 'cyan', 'white', 'gray', 'orange' ]
    def points_in_part ( part_index ):
        return [ school_latlng( s )
                 for s in schools_in_part_in_partition( part_index, schools_partition ) ]
    return usa_map( {
        colors[i] : points_in_part( i ) for i in range( max( schools_partition ) + 1 )
    }, title )
partition_map( random_partition(), 'Plotting a random partition as an example' )
```

Plotting a random partition as an example



We will also want to be able to alter a partition so that its parts are numbered from west to easy, so that we can easily name them, as follows.

Schools labeled with this index:	Fall into the region with this name:
0	West
1	Central
2	Midwest
3	Mideast

Schools labeled with this index:	Fall into the region with this name:
4	East
5	Northeast

[14]:
```
def resequence_partition ( partition ):
    part_indices = range( max( partition ) + 1 )
    def average_longitude ( part_index ):
        lngs = [ school_latlng( s )[1] for s in \
                 schools_in_part_in_partition( part_index, partition ) ]
        return sum( lngs ) / len( lngs ) if len( lngs ) > 0 else 0
    result = [ ]
    new_indices = sorted( part_indices, key=average_longitude )
    permutation = dict( zip( new_indices, part_indices ) )
    convert = lambda i: permutation[i] if i in permutation else len(␣
↪permutation )
    return [ convert( partition[i] ) for i in range( len( partition ) ) ]
region_names = [ 'West', 'Central', 'Midwest', 'Mideast', 'East', 'Northeast' ]
def region_name ( index ):
    if 0 <= index < len( region_names ):
        return region_names[index]
    return 'Other'
# resequence_partition( random_partition() )
```

1.5 Components of the Objective Function

First, we will want to experiment with the range of the various components of the objective function, to see how we should rescale them to match each other.

[15]:
```
def how_to_standardize ( func, num_tries=2500 ):
    data = np.array( [ func( random_partition() ) for i in tqdm( range(␣
↪num_tries ) ) ] )
    return data.mean() - 3*data.std(), data.mean() + 3*data.std()
```

1.5.1 Component 1: Variance of size of parts in the partition

[16]:
```
def part_size_variance ( partition ):
    return statistics.variance( (
        size_of_part_in_partition( part, partition )
        for part in range( num_parts_in_partition )
    ) )
# obj_fn_A1, obj_fn_B1 = how_to_standardize( part_size_variance )
bad_region_sizes = [
    np.round( (i+1)/(num_parts_in_partition*(num_parts_in_partition+1)/
↪2)*num_schools ) \
```

```
    for i in range(num_parts_in_partition)
]
obj_fn_A1, obj_fn_B1 = 0, statistics.variance( bad_region_sizes )
# plus we want size variance to be bad, so we reverse A and B:
def obj_fn_component_1 ( partition ):
    return np.clip( ( part_size_variance( partition ) - obj_fn_B1 ) / (␣
↪obj_fn_A1 - obj_fn_B1 ), 0, 1 )
obj_fn_A1, obj_fn_B1, bad_region_sizes
```

[16]: (0, 82.66666666666667, [5.0, 10.0, 15.0, 20.0, 25.0, 29.0])

1.5.2 Component 2: Total distance between schools in each part of the partition

[17]:
```
# from itertools import combinations
def total_distance_in_one_part ( part_index, partition ):
    # # Formerly, we used total travel distance among all pairs of schools,␣
↪like so:
    # indices = indices_for_part_in_partition( part_index, partition )
    # return sum( ( distance_lookup( i, j )
    #               for i, j in combinations( indices, 2 ) ) )
    # To be consistent with other clustering techniques, we now use total␣
↪distance to centroid:
    indices = indices_for_part_in_partition( part_index, partition )
    centroid = ( df.iloc[indices][SCH_LAT].mean(), df.iloc[indices][SCH_LNG].
↪mean() )
    return sum( [ geodesic( school_locations[i], centroid ).miles for i in␣
↪indices ] )
def total_distance_of_all_parts ( partition ):
    return sum( ( total_distance_in_one_part( part, partition )
                  for part in range( num_parts_in_partition ) ) )
# obj_fn_A2, obj_fn_B2 = how_to_standardize( total_distance_of_all_parts )
num_schools_in_a_part = int(num_schools/num_parts_in_partition)
obj_fn_A2, obj_fn_B2 = 0, num_parts_in_partition*(num_schools_in_a_part*500) #␣
↪a 500mi radius region would be very bad
# plus we want travel distance to be bad, so we reverse A and B:
def obj_fn_component_2 ( partition ):
    return np.clip( ( total_distance_of_all_parts( partition ) - obj_fn_B2 ) /␣
↪( obj_fn_A2 - obj_fn_B2 ), 0, 1 )
obj_fn_A2, obj_fn_B2
```

[17]: (0, 51000)

1.5.3 Component 3: Variance of mean powers of each part in partition

[18]:
```
def mean_power_of_part ( part_index, partition ):
    indices = indices_for_part_in_partition( part_index, partition )
    powers = df.iloc[indices][SCH_POW2]
    return powers.mean() if len( powers ) > 0 else 0
def part_power_variance ( partition ):
    return statistics.variance( (
        mean_power_of_part( part, partition )
        for part in range( num_parts_in_partition )
    ) )
# obj_fn_A3, obj_fn_B3 = how_to_standardize( part_power_variance )
obj_fn_A3, obj_fn_B3 = 0, part_power_variance( list( np.ceil(df[SCH_POW2].
↪rank(pct=True)*num_parts_in_partition )-1 ) )
# plus we want power variance to be bad, so we need a -1 multiplier:
def obj_fn_component_3 ( partition ):
    return np.clip( ( part_power_variance( partition ) - obj_fn_B3 ) / (␣
↪obj_fn_A3 - obj_fn_B3 ), 0, 1 )
obj_fn_A3, obj_fn_B3
```

[18]: (0, 9.249948009137002)

1.5.4 Objective function: geometric mean of 3 components

[19]:
```
def objective_function ( partition ):
    return ( obj_fn_component_1( partition ) \
           * obj_fn_component_2( partition ) \
           * obj_fn_component_3( partition ) ) ** (1/3)
```

1.6 Solving the problem with Genetic Algorithms

[20]:
```
num_generations = 10000
def progress_bar ( name="Progress", size=num_generations ):
    bar = tqdm( range( size ), desc=name )
    def step ( *args ):
        bar.update( 1 )
        bar.display()
    return step
best, fitness_curve = optimize_partition(
    objective_function = objective_function,
    initial_pool = [ random_partition() for i in range( 10 ) ],
    size_of_partition = num_parts_in_partition,
    prob_mutate = 0.1,
    num_generations = num_generations,
    progress_callback = progress_bar()
```

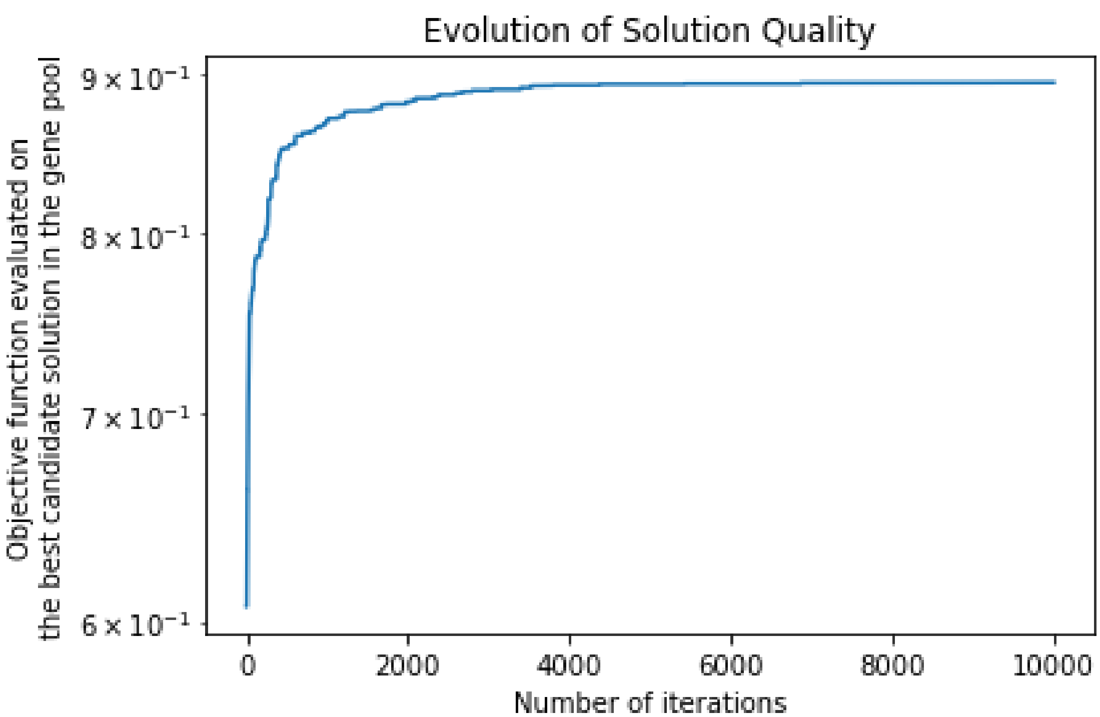
```
)
```

```
Progress:   0%|          | 0/10000 [00:00<?, ?it/s]

After 10000 generations: max score =   0.8942   100% done, 07:15/07:15 (00:00)
```

[21]:
```
# print_partition( best )
```

[22]:
```
plt.plot( range( len( fitness_curve ) ), fitness_curve )
plt.xlabel( 'Number of iterations' )
plt.ylabel( 'Objective function evaluated on\nthe best candidate solution in␣
↪the gene pool' )
plt.yscale( 'log' )
plt.title( 'Evolution of Solution Quality' )
plt.savefig( 'evolution.pdf' )
plt.show()
```



1.7 Viewing All Solutions

[23]:
```
from itertools import combinations
def robustness_check ( partition ):
    num_alternatives = 0
    num_better = 0
    best_obj_fun = objective_function( partition )
```

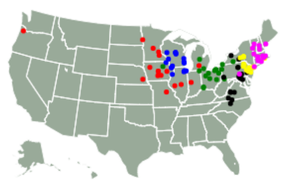
```
    for i, j in tqdm( list( combinations( range( len( partition ) ), 2 ) ) ):
        if partition[i] != partition[j]:
            num_alternatives += 1
            partition[i], partition[j] = partition[j], partition[i] # try this␣
↪swap
            value_obj_fun = objective_function( partition )
            if value_obj_fun > best_obj_fun:
                num_better += 1
                best_obj_fun = value_obj_fun
            partition[i], partition[j] = partition[j], partition[i] # unswap,␣
↪back to original
    print( f"{num_better} of {num_alternatives} neighboring partitions had␣
↪better objective functions." )
    print( f"Best objective function value among neighboring partitions:␣
↪{best_obj_fun}" )
def partition_report ( name ):
    partition = list( ( df[name] - df[name].min() ).fillna( -1 ).astype( int ) )
    print( 'Region'.ljust( 20 ), 'N'.rjust( 6 ), 'Power'.rjust( 7 ), 'Distance'.
↪rjust( 10 ) )
    for i in sorted( list( pd.Series( partition ).unique() ) ):
        if i == -1:
            continue
        s = size_of_part_in_partition( i, partition )
        print( region_name( i ).ljust( 20 ),
               str( s ).rjust( 6 ),
               str( round( mean_power_of_part( i, partition ), 2 ) ).rjust( 7 ),
               str( round( total_distance_in_one_part( i, partition ) / s, 2 )␣
↪).rjust( 10 ) )
    print( 'Components'.ljust( 20 ),
           str( round( obj_fn_component_1( partition ), 2 ) ).rjust( 6 ),
           str( round( obj_fn_component_2( partition ), 2 ) ).rjust( 7 ),
           str( round( obj_fn_component_3( partition ), 2 ) ).rjust( 10 ) )
    print( 'Objective function'.ljust( 20 ), str( objective_function( partition␣
↪) ).rjust( 25 ) )
#     robustness_check( partition )
    partition_map( resequence_partition( partition ), name+' Six-Part␣
↪Partition' )
```

[24]:
```
GA = 'GA Best'
df[GA] = best
partition_report( GA )
```

```
Region                    N   Power   Distance
West                     17    2.39     102.28
Central                  18    2.41     137.23
Midwest                  17    1.89      75.26
Mideast                  17    1.86      84.39
```

East	17	2.43	276.51
Northeast	17	1.77	143.36
Components	1.0	0.72	0.99
Objective function		0.8942055764708868	

GA Best Six-Part Partition

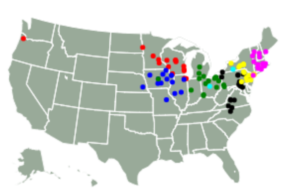


[25]:

```
partition_report( SCH_NCAA )
```

Region	N	Power	Distance
West	15	3.65	125.72
Central	17	2.34	250.28
Midwest	17	2.07	143.35
Mideast	18	2.24	141.95
East	17	1.74	82.86
Northeast	17	1.16	72.53
Components	0.99	0.73	0.93
Objective function		0.8740384020858701	

NCAA Asgt Six-Part Partition



[26]:

```
# Don't waste time computing this; ND's third result is better (below).
# partition_report( SCH_ND1 )
```

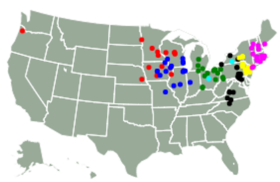
[27]:

```
# Don't waste time computing this; ND's third result is better (below).
# partition_report( SCH_ND2 )
```

[28]:
```
partition_report( SCH_ND3 )  # weighted optimization rectangles approach
```

```
Region                  N   Power   Distance
West                   17    2.52     144.36
Central                16    1.53      66.24
Midwest                16    3.12     264.48
Mideast                18    1.22      74.93
East                   16    1.58     109.68
Northeast              18    3.04     138.58
Components           0.99    0.74       0.93
Objective function          0.8773939127569947
```

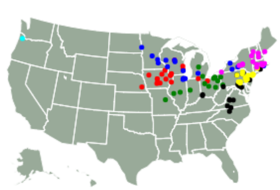
ND Asgt3 Six-Part Partition



[29]:
```
partition_report( SCH_PUR1 ) # balanced k-means approach
```

```
Region                  N   Power   Distance
West                   16    2.66       67.8
Central                18    1.01     110.32
Midwest                18    3.32     146.96
Mideast                15    2.21     176.04
East                   18    2.68     292.02
Northeast              17    1.01     181.03
Components           0.98    0.67        0.9
Objective function          0.8412137620282789
```

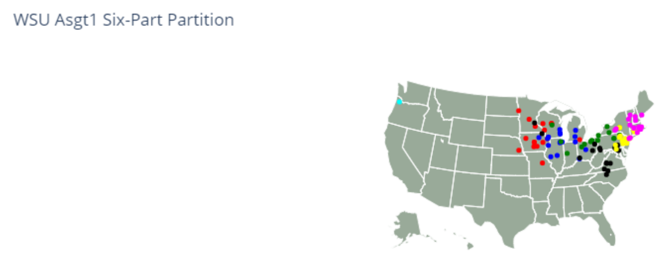
Purdue Asgt1 Six-Part Partition



[30]:
```
# Can't evaluate this one, because it's 8 regions, and the code herein assumes␣
↪6.
# partition_report( SCH_PUR2 )
```

[31]:
```
partition_report( SCH_WSU1 ) # weighted spatial clustering approach
```

Region	N	Power	Distance
West	17	2.14	76.04
Central	17	2.16	173.55
Midwest	18	2.01	94.59
Mideast	16	2.25	187.51
East	17	2.18	141.75
Northeast	17	2.16	225.0
Components	1.0	0.7	1.0
Objective function		0.887253059000749	



[32]:
```
# Can't evaluate this one, because it's 8 regions, and the code herein assumes␣
↪6.
# partition_report( SCH_WSU2 )
```

1.8 Exporting Partitions

The following function is useful for making spreadsheets that can easily beloaded into batchgeo.com for visualizing any region partition with high-quality graphics.

[33]:
```
def export_partition ( colname ):
    cols = ['College/University Name','Street','City␣
↪','State','Latitude','Longitude',colname]
    new_df = df[cols]
    cols[2] = 'City'
    cols[6] = 'Group'
    new_df.columns = cols
```

```
    new_df.to_csv( 'partition_'+colname+'.csv', index=False )
for col in df.columns[9:]:
    export_partition( col )
```

[ ]:

```

```