Network Optimization Course Project.
Yana Garipova
January 2021

Project Objectives:

- Solving Maximum Lifetime Scheduling Problem using the Linear Programming Garg-Könemann algorithm.

- Optimizing the Garg-Könemann algorithm to use for larger sensor networks (when the number of total coverages is large).

To begin, let us summarize the the Garg-Könemann algorithm:

-Initialize vector x (total active time for every sensor) and set equal to 0.
-Initialize vector y (active time for a sensor in a coverage) and set equal to a specific parameter beta.
-List all the total coverages as constraints: in terms of y-variables and coefficients 1 of a sensor is used in a certain coverage and 0 otherwise.
- Until all of the constraints are satisfied:
    -Search for the "most problematic" constraint among all coverages (*)
    -Update y and corresponding x.

The step (*) search for the "most problematic" constraint can be problematic when we have a lot of coverages to consider and may take a long time to solve. However, there is something we can do to improve this.

The code for the original GK Framework was distributed to all students.

Google Colab:
https://drive.google.com/file/d/15FpKTnXeL9-ywARyhxw9UD8K2zZw3H44/view?usp=sharing

Propose a method to optimize the search for the “most problematic” constraint in Garg-Könemann Framework (GK).

To optimize the Garg-Könemann framework, instead of considering predefined target coverages one-by-one, we can form a new coverage by using Greedy Algorithm as in Target Coverage Problem. The new constraint-searching function would be called “getApproximatelyBestTargetCoverage()”

The Target Coverage method utilizes:
- A monotone submodular function f, which can be defined as the number of targets a sensor can monitor;
- The sensor cost value, defined as the time used, given with the weightList.

The algorithm would continue to calculate the ratio of f to cost for every sensor and add the sensor which gives the maximum value until the chosen sensor subset output provides a total coverage.

This method is NP-hard and doesn’t guarantee to output the optimal solution. However, its runtime is proportional to the number of sensors and not to the number of total-coverage combinations of sensors as stated in the original GK framework. This approach would make a significant difference when working with large sensor networks.

To implement this approach, the following code was written:

```
#returns how many new targets would be added by a sensor
def f_delta_targets_monitored(coverage, coveredTarget, sensor):
    monitored = set()
    count = 0
    for i in coverage:
        monitored = monitored.union(coveredTarget[i])
    for i in sensor:
        if i not in monitored:
           count+=1
    return count



#form coverage from input: targets[sensor] set coveredTarget,
#all targets targetSet, weightList[sensor] as cost
def getApproximatelyBestTargetCoverage(coveredTarget, targetSet, weightList):
    return_set = []  #start with empty set S, treated as a list
    minWeight = 0
    #same as while f(S) < f(V)
    while isCover(coveredTarget, targetSet, return_set)!= True:
        max = 0
        for sensor in coveredTarget: #sensor is a set
            potential_value = f_delta_targets_monitored(return_set,
coveredTarget, sensor) / weightList[coveredTarget.index(sensor)]
            if potential_value > max:
                max = potential_value
                max_sensor = sensor
        max_sensor_index = coveredTarget.index(max_sensor) #index in the coveredTarget list
        return_set.append(max_sensor_index)
        minWeight += weightList[max_sensor_index]

    return return_set, minWeight
```

Result Comparison

Given:

```
coveredTarget = [set([1,2]),set([2,3]),set([1,3])]
targetSet = set([1,2,3])
```

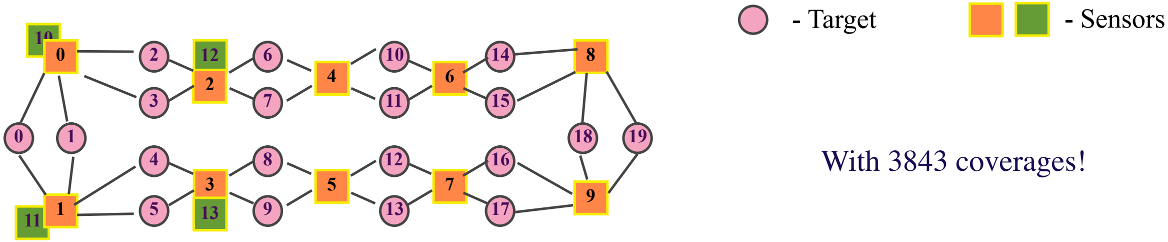
schedule, weightList output with getBestTargetCoverage() function :

```
({(0, 1): 0.4969050068359628,
  (0, 2): 0.4968152479835719,
  (1, 2): 0.4968152479835719},
 [0.503486315545794, 0.503486315545794, 0.49850130252058816])
```

schedule, weightList output with getApproximatelyBestTargetCoverage() function :

```
({(0, 1): 0.4969050068359628,
  (1, 2): 0.4968152479835719,
  (2, 0): 0.4968152479835719},
 [0.503486315545794, 0.503486315545794, 0.49850130252058816])
```

For the small sensor network the outputs are very similar and both functions returns fast.
Consider a larger sensor network like this:



With 3843 coverages!

```
coveredTarget = [set([0,1,2,3]),set([0,1,4,5]),set([2,3,6,7]),
set([4,5,8,9]),set([6,7,10,11]),set([8,9,12,13]),set([10,11,14,15]),
set([12,13,16,17]),set([14,15,18,19]), set([16, 17, 18, 19]),
set([0,1,2,3]),set([0,1,4,5]),set([2,3,6,7]), set([4,5,8,9])]

targetSet = set([0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19])
```

getBestTargetCoverage() outputs this schedule after 109 seconds:

```
{(0, 3, 4, 7, 8): 0.8356,
(1, 2, 5, 6, 9): 0.8359,
(5, 6, 9, 11, 12): 0.1029,
(4, 7, 8, 10, 13): 0.1029,
(3, 4, 7, 8, 10): 0.03592,
(2, 5, 6, 9, 11): 0.03592,
(1, 5, 6, 9, 12): 0.0217,
(0, 4, 7, 8, 13): 0.0217}
```

getApproximatelyBestTargetCoverage() outputs this schedule after 6 seconds :

```
{(0, 3, 4, 7, 8): 0.9946,
(1, 2, 5, 6, 9): 0.9946}
```

We can see that applying Target Coverage problem to recreate the most problematic constraint can greatly speed up the Garg-Könemann algorithm while providing a fairly good solution.
In the future it would be interesting to test these algorithms on a larger and more complicated sensor network.