Metric k-Center Problem
Solving with GA & WoC Techniques

John Murray

Computer Science

Speed School of Engineering

jmmurr02@louisville.edu

1. Introduction

The Metric K-Center problem is a combinatorial optimization problem centered in graph theory. It is defined formally as:

> *Given a complete, undirected graph* G *which is equal to* (V,E) *where* V *is the set of all vertices and* E *is the set of all edges and where distance is defined as* $d(v_i,v_j) \epsilon N$ *that satisfies the triangle inequality[1] , find a subset* $S \subseteq V$ *where* $|S| = k$ *while minimizing* $d(v,s)_{s \epsilon S. v \epsilon V}$ *and* k*.*

An Example

A company in the United States is expanding their business into Western Europe. As such, they need to plan their distribution centers such that they are able to come up with the most-optimal configuration using the minimal amount of facilities but minimizing distance to all of the major cities.

In this case, each city would be a vertex in our graph and the distance between each city the edge. The distribution centers would be the set S and the number of distribution centers k.

2. Approach

The solution was written in JRuby (Ruby interpreter for the JVM) with a graphical “results viewer” written in CRuby and Javascript. The project is divided into two logical components, the “runner” and the “viewer” both of which are described below.

The Runner

The runner is the program responsible for running the algorithm to produce the results. This portion of the project was written in JRuby to take advantage of the *real* thread-model supported by the JRuby interpreter and is not inhibited by the global interpreter lock (like CRuby). This was important for optimizing the algorithm to compute solutions in parallel. Being able to compute in parallel is very useful when searching for the most optimal k value (further explanation below).

The Viewer

The viewer is the program responsible for visually representing the intermediary and final results for each run. The flow for rendering results is generally as follows:

- The *runner* is ran over a set of test data
- A run-UID is generated
- On each iteration, the solution is logged in JSON notation into the local DB with the UID
- A final solution is logged in JSON notation into the local DB with the UID
- The *viewer* server is started
- The latest run is requested from the DB
- Each iteration result, along with the final result, are rendered via Javascript to a web-page
- The client can view results or load results from a previous run for viewing

The reason for not creating one integrated Solver/UI was mainly due to the issues with currently existing UI frameworks in Ruby. Not being built with desktop applications in mind, Ruby does not have great library support for GUIs. However, it does have fantastic facilities for putting together minimal web-servers which influenced the resulting browser-based UI.

The Algorithm

Before discussing the algorithm it will be important to note a few things. In the description of the problem given above you will note that we are trying to minimize k while also minimizing cost. However, in the definition of our problem, there is no weight given for k. Thus, the following will hold for $k < |V|$ and an optimal solution-set S:

$$cost(G)_{|S| = k+1} \leq cost(G)_{|S|=k}$$

So, we could either solve for a particular k, we could give k a weight, or we could solve for a range of k. In my algorithm, I’ve chosen to solve for a range of k hoping that we will yield more interesting results as to what a good value of k will be if k were to be given an arbitrary weight.

It is also worth noting that this problem is similar to finding a Dominating Set (which is also an NP-complete problem). However, since our graph in the Metric k-Center problem is complete, we have a dominating set when the following is true: $S = \{v_i\}, v_i \epsilon V$. This is also true for any number of elements from V in set S. So, while similar, we cannot use the known algorithms for finding a dominating set for our graph unless we can reduce the graph to be non-complete.

So, as mentioned earlier, JRuby was used to compute the results in a concurrent fashion. Since we are computing multiple values of k there is no need to run those computations in a non-concurrent fashion (just to save time). The main portion of code that is responsible for computing each k value is roughly the following:

```
# load files
filename    = get_filename
file_handle = open_file(filename, 'r')

# load data
nodes   = Utils.parse(file_handle.read)
k_range = ((nodes.length * 0.2).floor)...((nodes.length * 0.8).ceil)

# find solution(s)
k_range.peach(CONNECT_POOL_SIZE) do |k|

  # initialize run in DB
  db_result = init_db_run(k)

  # initialize the population
  population = Population.new(nodes.dup, {
    :run_id => db_result.run_id,
    :k => k,
    :size => GA_POPULATION_SIZE
  })

  # Do the evolutions and WoC's (the estimation)
  GA_EVOLUTIONS.times do
    population.evolve
    experts = population.experts
    consensus = WOC.consensus_of(experts, k: k)
    population << consensus
    population.record!
  end

  # Collect the final metrics
  best_solution = population.experts.first
  db_result.cost = best_solution.cost
  db_result.save
```

```
    # Close the thread-specific DB connection
    ActiveRecord::Base.clear_active_connections!
  end
```

You can see that the main algorithm is rather simple looking (thanks in part to Ruby's syntax) and we are simply performing the following actions:

1. Opening and parsing an input file
2. Specifying the range of k-values, in this case 20-80% of $|V|$ since we know that for low values of k we get a non-optimal solution and for high-values of k we get a solution that is impractical since we are trying to minimize k.
3. In parallel for each value of k with thread-pool of size `CONNECT_POOL_SIZE`
    a. Initialize a DB-entry for the specific run with value k
    b. Initialize our population with size `GA_POPULATION_SIZE`
    c. Perform `GA_EVOLUTIONS` evolutions inserting the consensus solution back into the population after each evolution
    d. Saving the results to the DB and closing the thread-specific DB-connection

The next important thing to discuss is the method for creating the initial population, cross-breeding two solutions, and the mutation algorithm. The initial population is generated randomly and is quickly conveyed by the following method:

```
  def init_people
    @opts[:size].times do
      nodes_s = @nodes.sample(@opts[:k])
      nodes_v = @nodes - nodes_s
      self << Individual.new(nodes_v, nodes_s)
    end
  end
```

You can see that I am creating a population of size `@opts[:size]` and randomly building S of size k from the set of nodes. Fairly simple way to start. I could also use a greedy heuristic to create my initial solutions, but that could yield too similar of results that would produce lower yield results for the GA.

The more interesting algorithm is how the child of two parent-solutions is generated. The algorithm uses the common values between the two parents in S and then uses a greedy heuristic to determine the rest of the values for S if $|S|$ in the child is less than k. The algorithm is represented by a single method:

```
  def breed (people)
    p1, p2 = people
```

```
    intersect = p1.nodes_s & p2.nodes_s

    while intersect.length < @opts[:k]
      max_dist = 0
      furthest_node = nil

      intersect.each do |s|
        @nodes.each do |v|
          if v != s && v.distance_to(s) > max_dist
            max_dist = v.distance_to(s)
            furthest_node = v
          end
        end
      end

      intersect << furthest_node
    end

    [mutate(Individual.new(@nodes - intersect, intersect))]
  end
```

You can see above that we are taking the intersection (the common values) and augmenting those with values collected from our greedy heuristic while $|S|<k$. The greedy heuristic chooses the node in V that is furthest away from any value in S and then adds to to S.

The mutation algorithm takes a member of the population and randomly swaps values in S with values in V. It is represented very obviously in the following function:

```
  def mutate (person)
    if rand > (1 - MUTATION_RATE)
      (person.nodes_s.length * 0.2).ceil.times do
        s = person.nodes_s.delete(person.nodes_s.sample)
        v = person.nodes_v.delete(person.nodes_v.sample)

        person.nodes_v << s
        person.nodes_s << v
      end
    end
    person
  end
```

The only thing worth noting is that if we mutate the person, based on random chance, we are changing approximately 20% of the nodes within S.

3. Results

Input Data

The input data consisted of 30, 75, and 100 data-points in a coordinate plane bounded on the x and y-axis by 0 and 1,000. The results were generated by a tool that was written to aid with the project. It generates random data-points within the bounds specified of a size determined by the user-input to the command-line interface that the tool uses. The main code used for generation is simply as follows:

```
def main
  opts = parse_opts

  points = []

  opts[:number].times do |n|
    points << {
      x: rand * SCALE,
      y: rand * SCALE,
    }
  end

  File.open(opts[:filename], 'w') { |f| f.puts points.to_json }
end
```

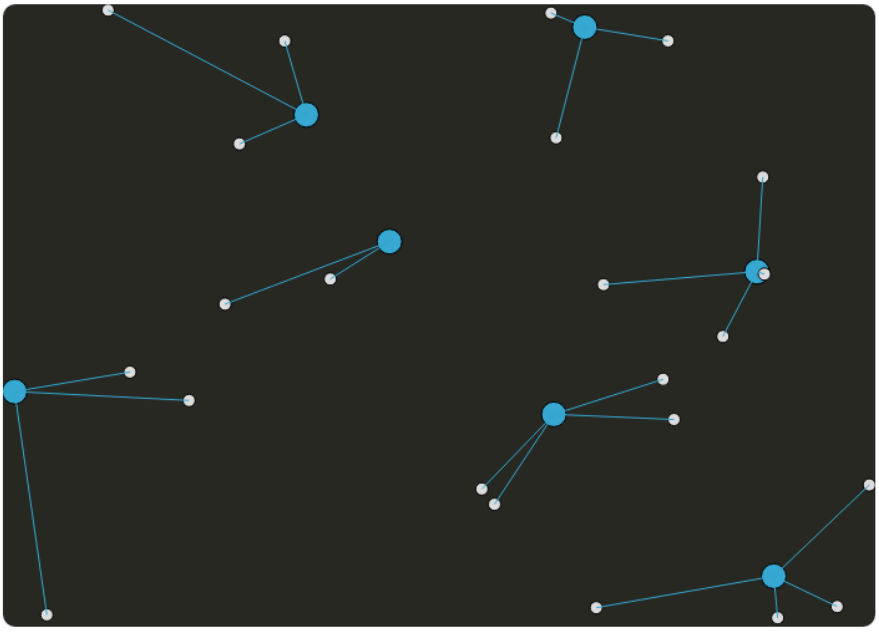
The bulk of the code is in parsing the command line options and displaying any help necessary. You can see that we are storing the results in JSON format since Ruby has a great JSON lib that makes [de]serialization a simple task.

Visualization

As mentioned prior, the project was split into two components. The runner, which has been thoroughly explained by this point, and the UI. The UI however, is better discussed while looking at images rather than code. Below are the results for mid-range k values for sets of 30, 75, and 100.

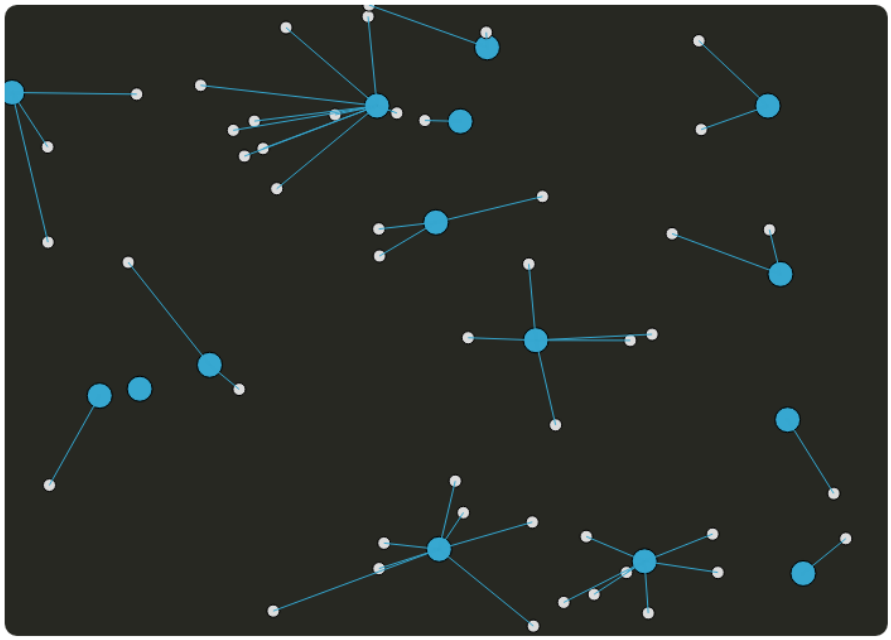
30-nodes

[2012-11-18 02:42:46 -0500] k: 7 generation: 2 Render Result

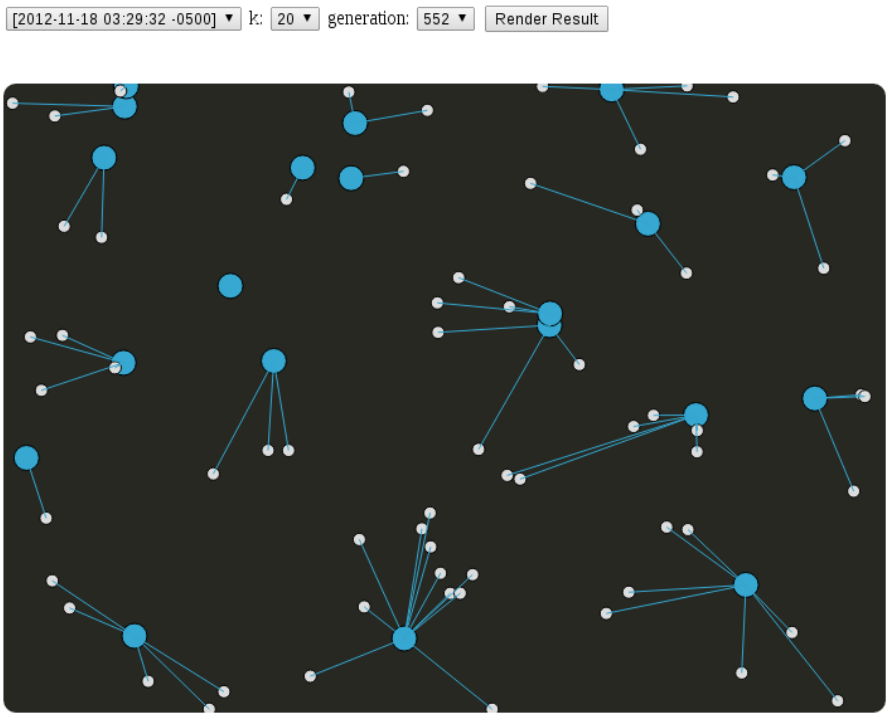


75-nodes

[2012-11-18 02:58:00 -0500] k: 15 generation: 1501 Render Result



100-nodes

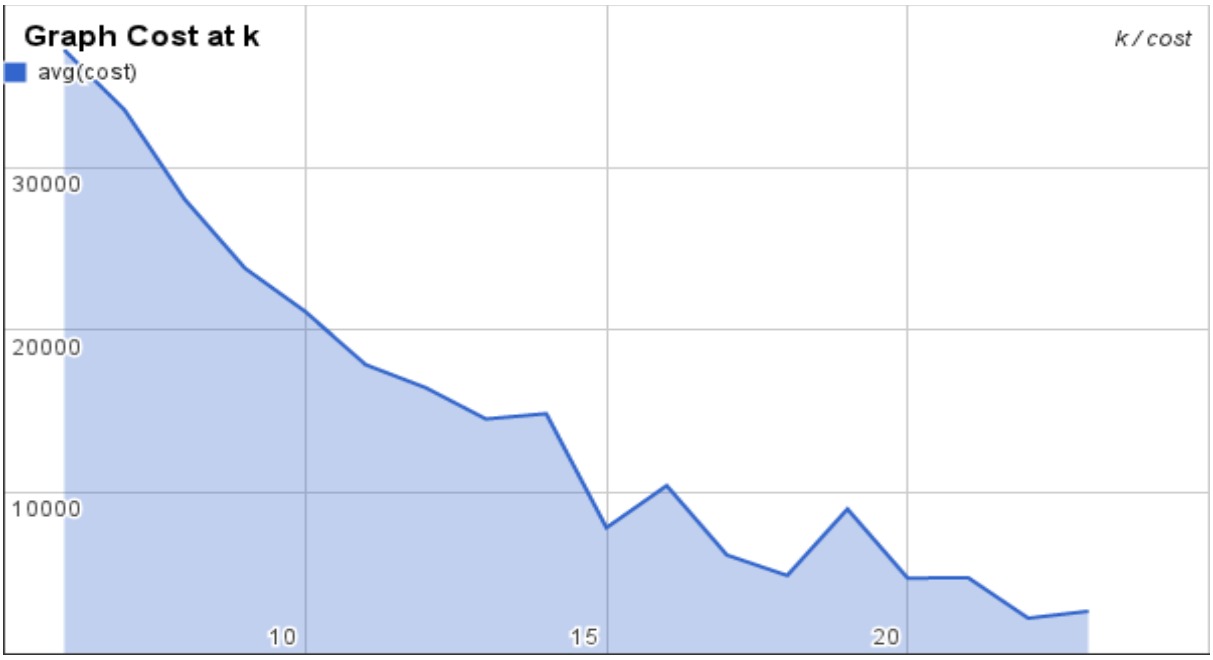


One thing to note is since our algorithm is solely focused on minimizing the distance from any node in V to the closest node in S, we are only showing those distances. You'll notice that all nodes in S are represented by large, blue circles while the small, white circles represent nodes in V.

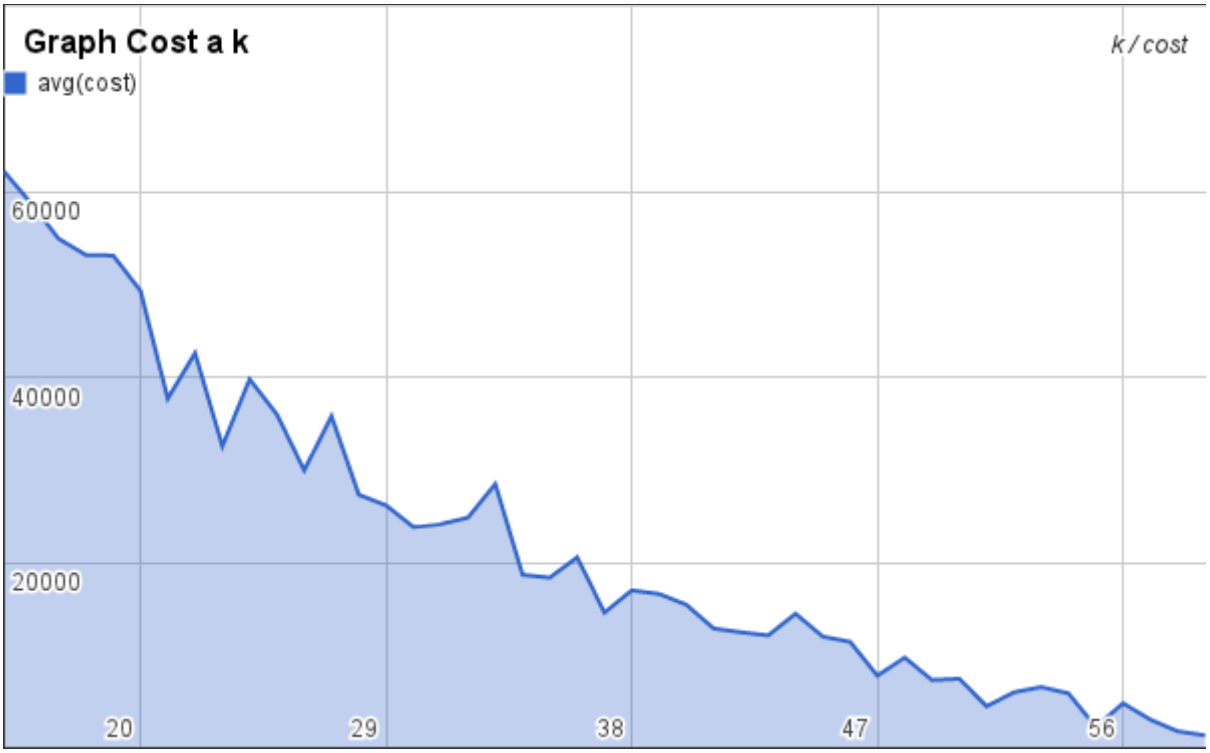
Inspecting K

The following graphs show the final cost for a graph for each value of K that was inspected.

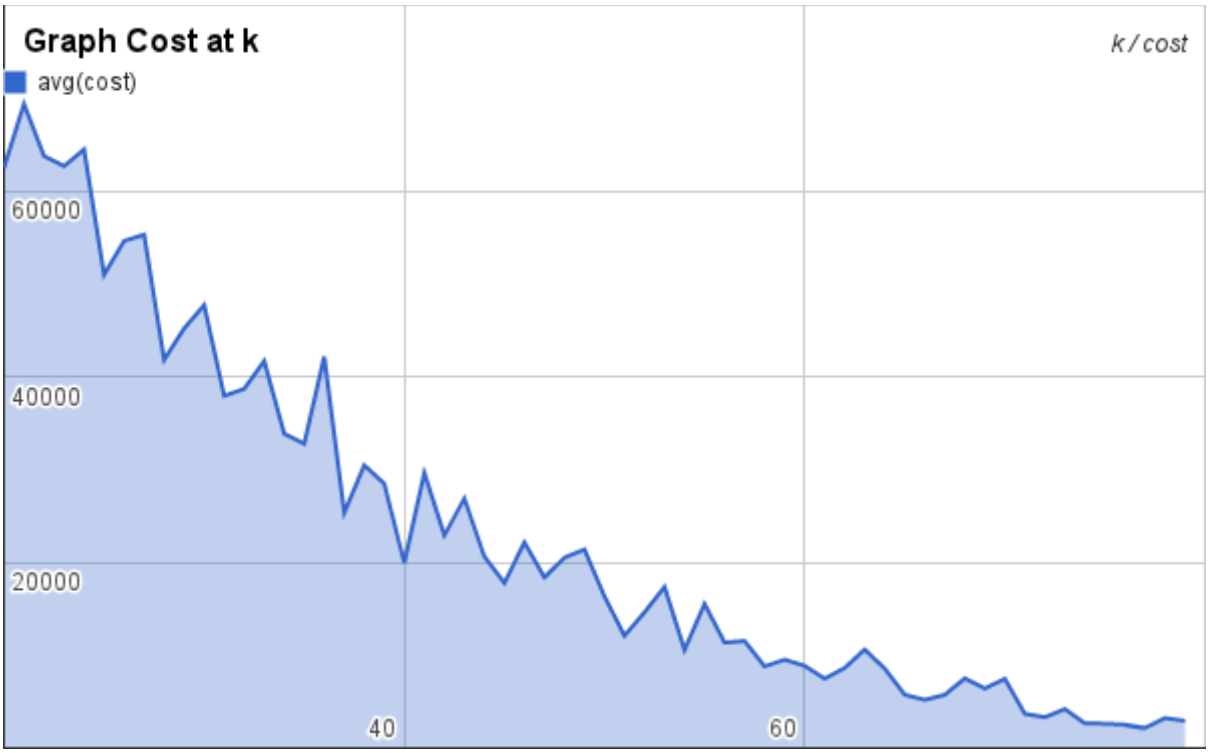
30-nodes



75-nodes



100-nodes



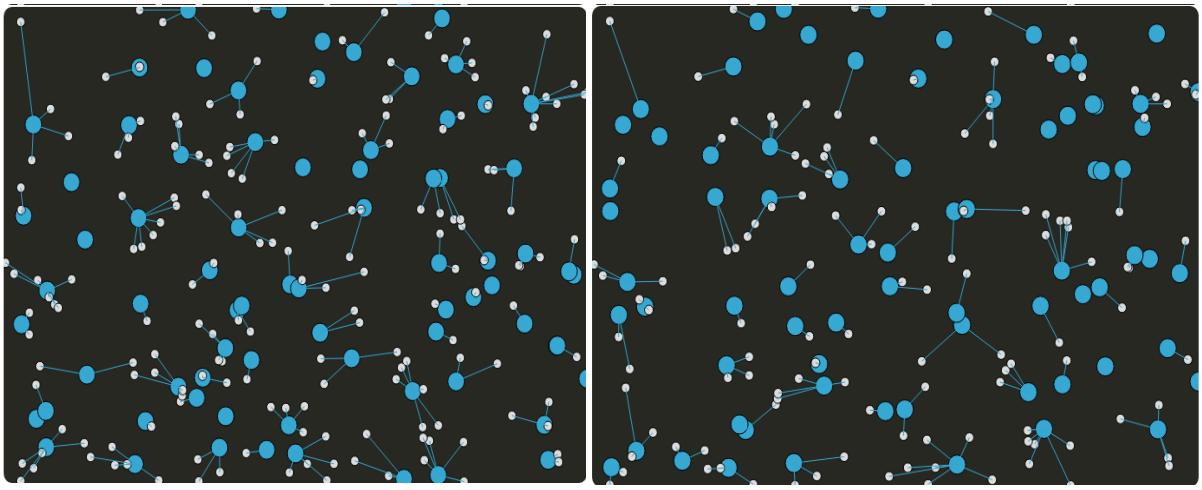
While we can see that we are achieving rather good results for all greater values of k (as discussed earlier), we can see that the most improvement happens around the 40% mark.That is, when 40% of all nodes in G are in the set S.

Generational Improvement

So we can see that greater values of k yields lower costs overall, however that is very expected. What is more interesting is how well the current algorithm is at improving cost over a random guess (which is what we're starting with). The following is a graph from a run with 250 nodes which was done with only one value of k, such that $k = 0.3 \times |S|$. This was done for only one value in k for time reasons as it is rather expensive to compute for many values of k (as the current algorithm operates).

You'll note that I am only showing the first 10 generations. The graph is so flat after the first generation, then you cannot see the slope, thus I have focused in on a particular range. The two solutions (for generation 1 and 2) visually look like the following:

Generation 1 **Generation 2**



While there is room for improvement in the second generation, the mutations were not enough to escape the local maximum. This could be overcome by running with a higher number of generations.

Further Study

If I were to continue this research I would experiment further with higher number of generations as well as a higher mutation rate to escape local maximums. Perhaps a more random child/ breeding strategy could also be used to generate more random results, thus increasing the chances of escaping any local maximums.

It would be valuable to study an alternative version of the problem where k is given explicit weights based on a deterministic parameter to the problem. This would aid greatly in finding an optimal solution.