Project 3
Sudarshana Jagadeeshi

Contents

Objective

We are given a set of coordinates that represent the physical location of the nodes and various constraints on the topology of the graph.

The problem at hand is to optimize using two separate optimization algorithms to yield a graph with minimum cost.

Approach

We first generate random coordinates by creating an array where each element is an array [x,y] within the range (0, BOXSIZE). We then generate a random graph as described in the "Generation of random input" section. Then, both Simulated Annealing and Tabu Search run on this graph, and report their results to be plotted and analyzed.

Algorithms

Generation of random input

We know that a fully connected graph will of course satisfy all the constraints. We could theoretically just return this, but it would be a lot of work for our algorithms to optimize.

This algorithm does the next best thing by first creating a path graph to ensure connectivity, then adding random edges until the degree and diameter constraints are satisfied.

Neighbors

Neighbors are those graphs where one of the edges is removed from the original. In the case that all graphs with one edge removed are invalid, then the neighbors are defined as all those graphs where one of the edges is added from the original. This is to keep the algorithms from stalling out when the number of edges becomes critical.

There are of course, other neighbors that could be generated. Each of the edges could be assigned a new destination. However this would add E * N new neighbors and would slow the algorithm considerably.

Simulated Annealing

First the neighbors are generated. From the neighbors, a random neighbor is selected. This random neighbor is used to calculate delta- the difference in cost between the chosen neighbor and the original graph.

If the delta is less than 0, that is, the neighbor is less costly than the original, the neighbor is set to be the point for the next iteration.

If not, the following formula is used to see if the worse graph will be picked anyway:

$$\text{Prob}_{accept} = \mathrm{e}^{-\frac{\Delta E}{kT}}$$

For our application, k is always 1. T is initialized to a user chosen value and changes every epoch according to the following formula:

$$T_n = \frac{a}{b + \log n}$$

In our case, a is a constant initialized to a user chosen value, n is the epoch value, and b is always 1.

Tabu Search

This is a greedy search. Instead of randomly choosing from the neighborhood, the best value is chosen. The only difference is that the neighborhood is filtered through a tabu list- which are the k most recently visited points. If the neighbor is already in the tabu list, it cannot be selected. This prevents the algorithm from getting stuck in a local optimum.

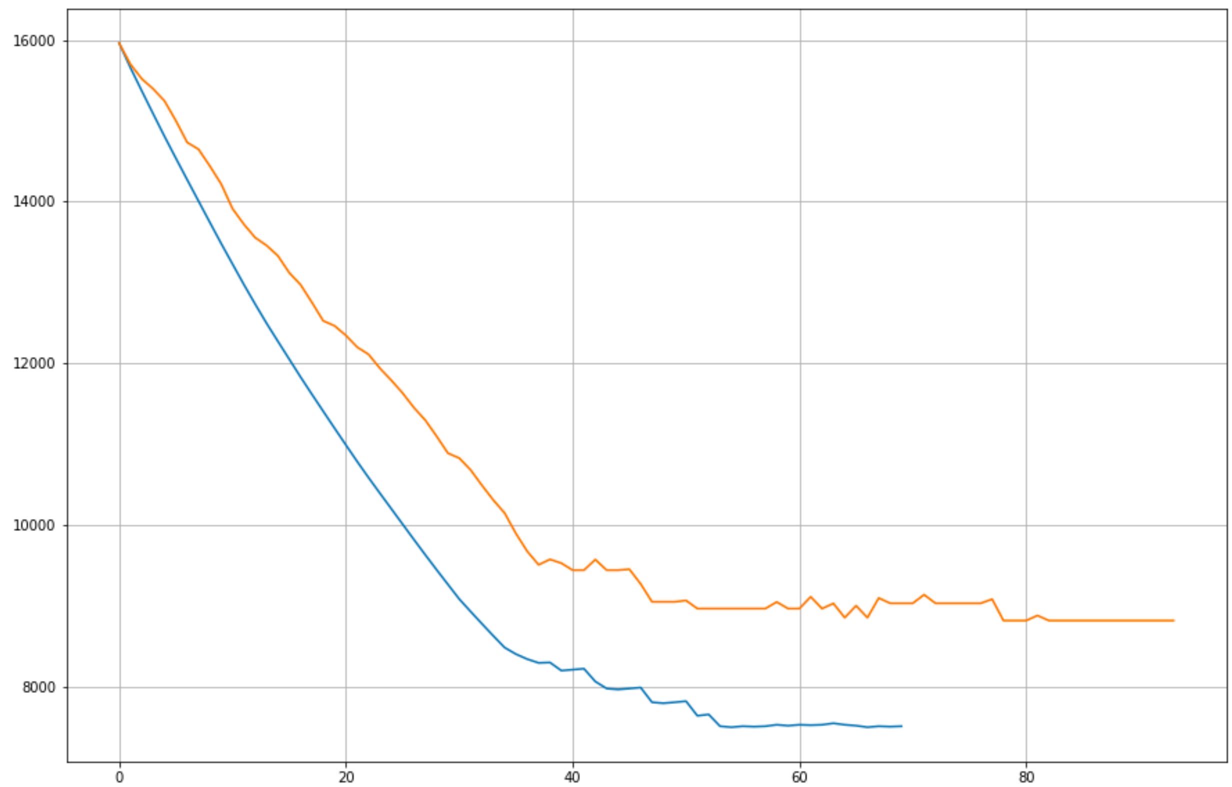
Stopping Criteria

For both algorithms, if no significant (> 0.5) improvement is found for a certain number of iterations, the algorithm terminates and returns its best graph.

Results

Figures

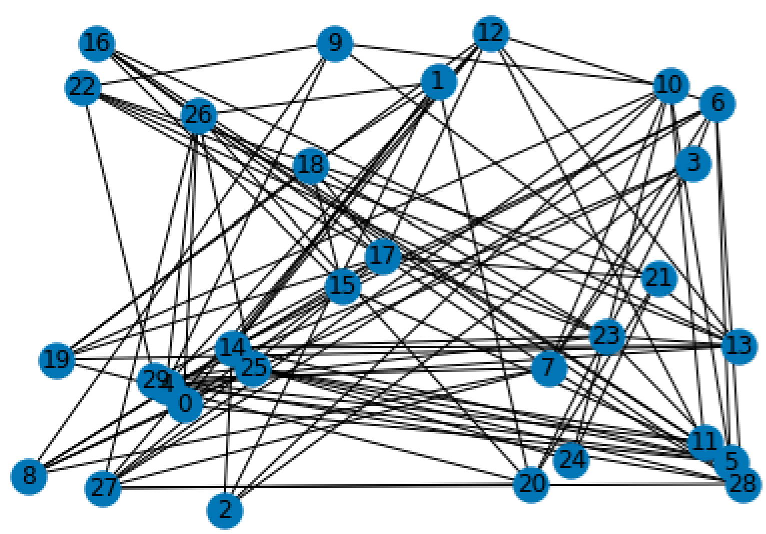
Fig 1.1 Cost for each algorithm w/ N=30



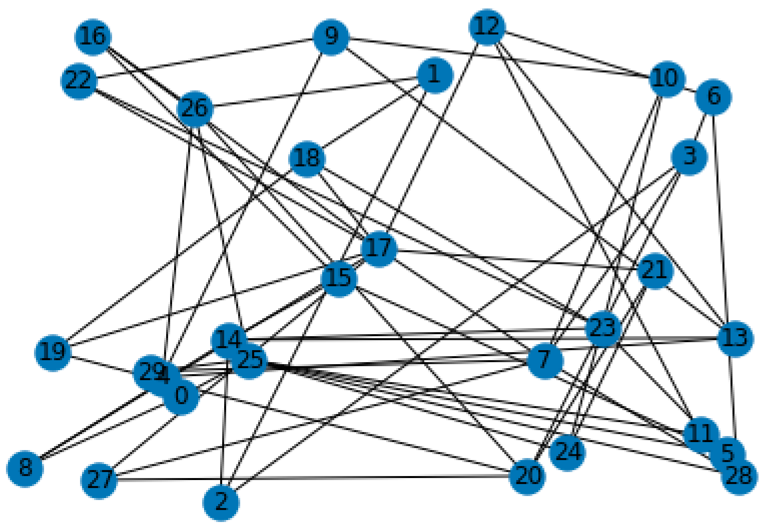
Simulated annealing is in orange.

Fig 1.2 Graphs for each algorithm with n=30

ORIGINALCOST 15959.453578024226
Graph with 30 nodes and 100 edges



```
TABUSEARCHCOST 7496.650758372878
Graph with 30 nodes and 62 edges
```




```
SIMULATEANNEALINGCOST 8815.845971000197
Graph with 30 nodes and 61 edges
```

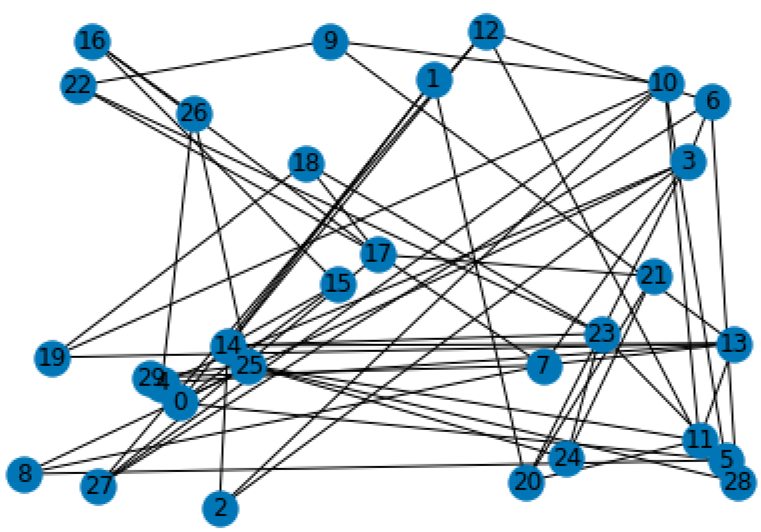


Fig 2.1 Cost for each algorithm w/ N=20

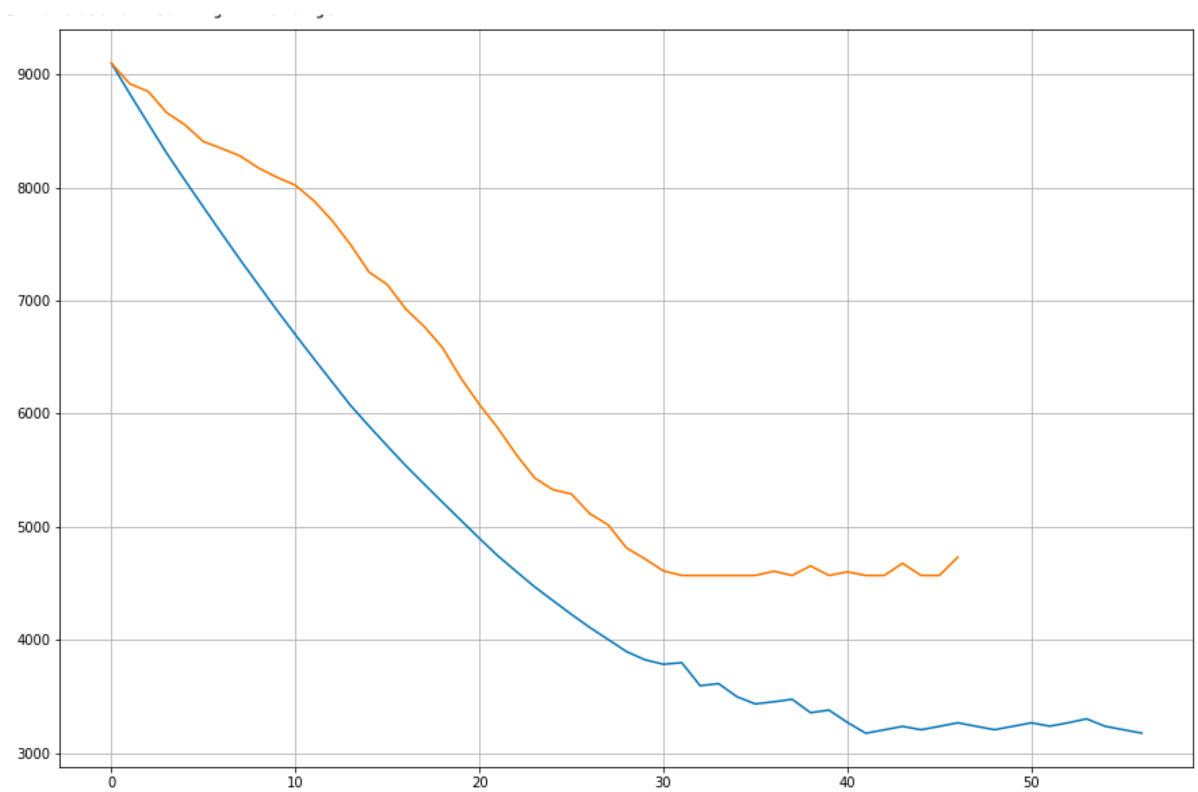
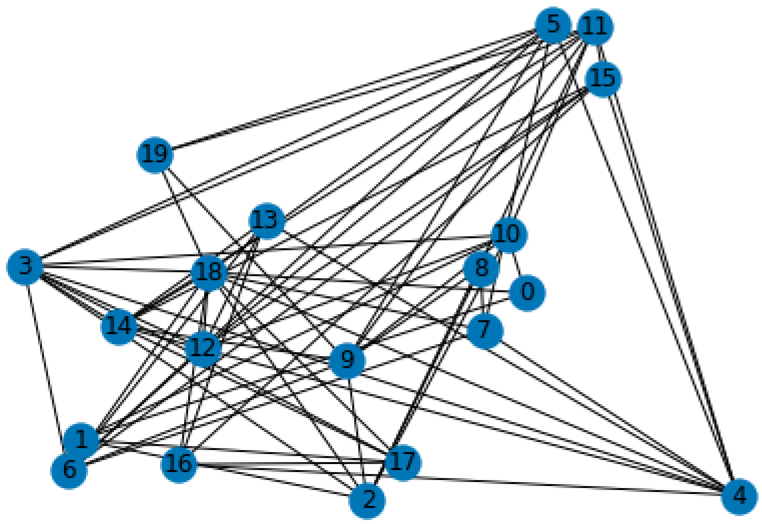
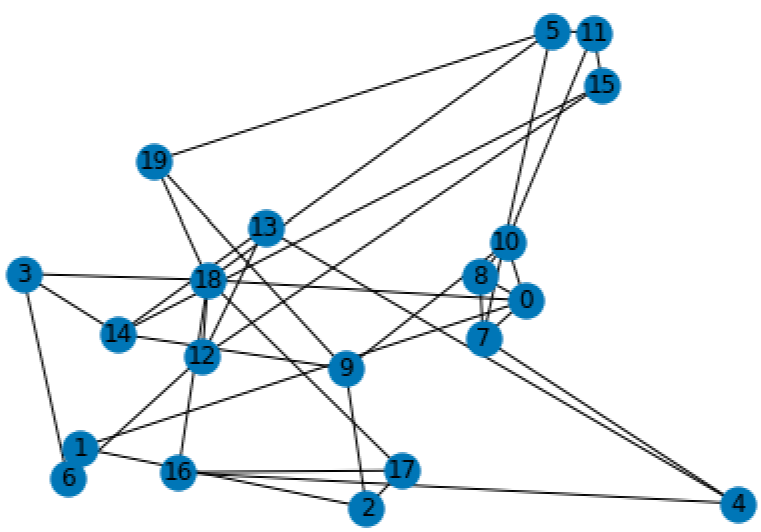


Fig 2.2 Graphs for each algorithm with n=20

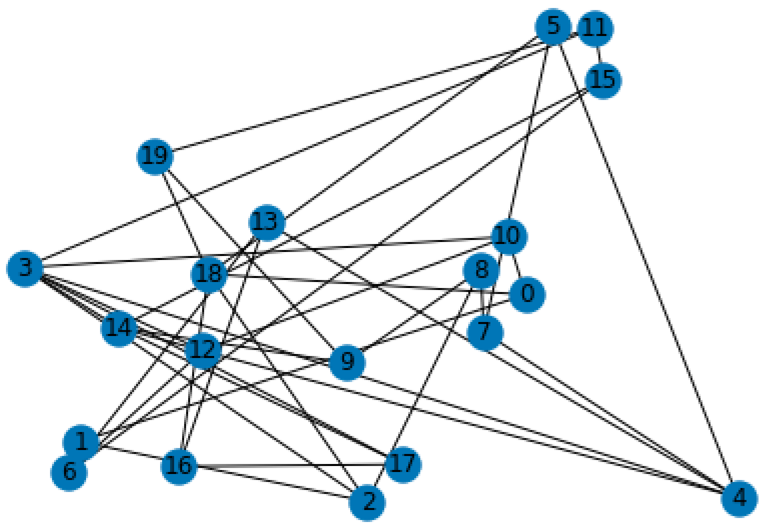
ORIGINALCOST 9100.391952118978
Graph with 20 nodes and 67 edges



TABUSEARCHCOST 3174.5146359755745
Graph with 20 nodes and 36 edges



```
SIMULATEANNEALINGCOST 4570.349078682738
Graph with 20 nodes and 36 edges
```



Analysis

The number of edges in the graph a big deal- it is harder to find improvements and restructure a graph with few remaining edges. We can see this as both graphs begin “struggling” at around the same point. But though both graphs have the same number of edges, simulated annealing always performs worse than tabu-search by approximately 15-30%.The gap gets closer as the number of nodes increase.

My theory it is because SA fails to remove the costliest edges first, instead choosing randomly. It is then forced to rely on them to satisfy constraints. To stop relying on these longer edges, large changes are required that cannot be reversed by simply moving locally. It would require a much larger neighborhood, and a much hotter system.

Tabu search, being greedy, is shown to converge must faster. It also abuses the cheap nodes (the ones closest to other nodes) to connect the graph better. In the Fig 2.2, node 18 is in an excellent position. The degree of this node is 7 in tabu search. Simulated annealing, on the other hand, assigns a degree of 7 to node 3, which is not very centralized.

Since both algorithms have about the same running time (the only difference is that tabu search must search the tabu list), we can say that tabu search is the better algorithm for this problem, which greatly rewards a greedy strategy.

Appendix

Annotated Source Code

A link to the code can be found here:
https://colab.research.google.com/drive/1uQ9ZIeTQ6EZzQcrJxqYPvL96ebcgICCf?usp=sharing

```python

import networkx as nx
import matplotlib.pyplot as plt
import random
import math
import copy
import sys
import numpy as np

N= 20 #number of nodes
BOXSIZE= 250 #size of the box


a= 500
b= 1 #for simulated annealing
T= 1 #for simulated annealing

#@title gen_coordinates

def gen_coordinates():
  coordinates= [] #coordinates is a Nx2 array
  while (1):
    randomx= random.randint(0, BOXSIZE+1)
    randomy= random.randint(0, BOXSIZE+1)

    if (len(coordinates) == N):
      print("Finished generating points")
      break;
    if ([randomx, randomy] in coordinates):
      print("Spot taken, trying again")
    else:
      coordinates.append([randomx, randomy])

  print(coordinates)

  return coordinates

#@title gen_neighbor
def gen_neighbor(G, coordinates, tabulist, tabumode):
  nhood=[]
  addhood=[]

  #adding removal neighbors
  for e in G.edges():
    tempG= nx.Graph()
    tempG.add_edges_from(G.edges)
    tempG.remove_edge(e[0], e[1])

    nhood.append(tempG)

```

```python
  #adding addition neighbors
  for s in range(N):
    for d in range(N):

      if (s == d): #no self loops
        continue;
      else:
        if ([s,d] not in G.edges):
          tempG= nx.Graph()
          tempG.add_edges_from(G.edges)
          tempG.add_edge(s, d)

          addhood.append(tempG)


  while (1):
    if (len(nhood) == 0): #all the graphs have been popped off
      print("NO VALID REMOVAL GRAPHS")

      if(not tabumode): #i.e. called by SA
        idx= random.choice(range(len(addhood)))
      else: #else get the min cost add neighbor
        costs= []
        for i in addhood:
          costs.append(calc_cost(i, coordinates))

        mincost= min(costs)
        idx= costs.index(mincost)
      selected= addhood[idx]
      break; #add-graphs are not checked for validity, they are always valid


    if(not tabumode):
      idx= random.choice(range(len(nhood))) #choose a random graph
    else: #choose the best graph (greedy)
      costs= []
      #print("NHOOD LENGTH", len(nhood))
      for i in nhood:
        costs.append(calc_cost(i, coordinates))

      mincost= min(costs)
      idx= costs.index(mincost)
    selected= nhood[idx]

    selectedcost= calc_cost(selected, coordinates)

    if (not valid(selected)):
        nhood.pop(idx)
    else: #it is a valid graph
```

```
      if(not tabumode):
        break;
      if( tabumode and (selectedcost in tabulist)):
        nhood.pop(idx)
      elif(tabumode and (selectedcost not in tabulist)):
        break;

  return selected

def isdegree3(G):
  for e in G.degree:
    if e[1] < 3:
      #print(e[0], "is not degree 3, it is: " , e[1])
      return False
  return True

def valid(graph):
  if (not nx.is_connected(graph)):
    #print("IN VALID: NOT CONNECTED!")
    return False
  if (nx.algorithms.distance_measures.diameter(graph) >= 4):
    #print("IN VALID: NOT DIAMETER <= 4")
    return False
  if ( not isdegree3(graph)):
    #print("IN VALID: NOT DEGREE >= 3")
    return False
  else:
    return True

def calc_cost(G, coordinates):
  total = 0
  for e in G.edges:
    source= e[0]
    dest= e[1]

    try:
      temp= [coordinates[dest][0]-coordinates[source][0], coordinates[dest][1]-coordinates[source][1]]
      total += np.linalg.norm(temp)
    except IndexError:
      print("DUMP",dest, source)
  return total

#@title gen_random_graph
def gen_random_graph(coordinates):
  G = nx.Graph()
  for i in range(N):
    #pos is used for plotting only, not for the algorithm
    G.add_node(i, pos= (coordinates[i][0], coordinates[i][1]))

```

```
  #first create a path graph to ensure the graph is connected
  i=0
  j=1
  while (j < N):
    G.add_edge(i,j)
    i += 1
    j += 1

  for _ in range(500):
    #print("THIS ITERATION:", nx.algorithms.distance_measures.diameter(G), isdegree3(G) )
    if (valid(G)):
      #add a random edge
      print("Good, a random graph has been generated that meets criteria.")
      break;
    else:
      source= random.randint(0,N-1)
      dest= random.randint(0,N-1)

      while (dest == source): #no self loops
        dest= random.randint(0,N-1)

      G.add_edge(source, dest)

  return G

#@title tabu search
#maxreps- the maximum number of epochs w/o an improvement on bestvalue
#before the algorithm terminates
def tabu_search(G, coordinates, maxreps, maxtabulength):
  #TODO: Add convergence criteria and temperature function
  bestvalue= 9999999;
  tabulist=[]
  reps=maxreps;
  bestG=nx.Graph()
  epochcosts=[]

  for epoch in range(999): #just a value, the algorithm has diff stopping criteria
    Gcost= calc_cost(G, coordinates)
    epochcosts.append(Gcost)

    print("====epoch===== ", epoch, Gcost)
    print("reps", reps)
    if(reps == 0):
      print("MAX REPS REACHED")
      return {'best': bestG, 'costs': epochcosts}


    selected= gen_neighbor(G, coordinates, tabulist, True)

    tabulist.append(calc_cost(selected, coordinates))
```

```python
    tabulist.append(calc_cost(selected, coordinates))
    if(len(tabulist) > maxtabulength):
      tabulist.pop(0) #make sure tabu list stays <= maxtabulength

   #a nicer way to copy one graph to the other, copy.copy is expensive
    G= nx.classes.function.create_empty_copy(G)
    G.add_edges_from(selected.edges)

    #print("COMPARING:",calc_cost(G, coordinates), bestvalue)
    improvement= bestvalue -calc_cost(G, coordinates)
    #sometimes very miniscule improvements continue to be found,
    #causing the algorithm to never terminate, hence 0.5
    if (improvement > 0.5):
      print("IMPROVEMENT FOUND", improvement)
      bestG= nx.classes.function.create_empty_copy(G)
      bestG.add_edges_from(selected.edges)

      bestvalue= calc_cost(selected, coordinates)
      reps=maxreps;
      continue;
    else: #bestvalue was not changed
      reps -= 1

  return {'best': bestG, 'costs': epochcosts}

#@title simulated_annealing
def simulated_annealing(G, coordinates, maxreps):
  #TODO: Add convergence criteria and temperature function
  bestG= nx.Graph();
  reps=maxreps;
  costs=[]
  bestvalue=99999999

  for epoch in range(999):
    T= a/(1+ math.log(epoch+1) ) #cool the system, +1 to avoid domain error
    costs.append(calc_cost(G, coordinates))

    if(reps == 0):
      print("MAX REPS REACHED")
      return {'best': bestG, 'costs': costs}

    print("====epoch===== ", epoch, calc_cost(G, coordinates), bestvalue)

    Gcost= calc_cost(G, coordinates)
    selected= gen_neighbor(G, coordinates, [], False)
    delta= calc_cost(selected, coordinates)- Gcost

    #all similar to the other algorithm
    improvement= bestvalue- calc_cost(selected,coordinates)
```

```
        if(improvement > 0.5):
            print("IMPROVEMENT FOUND", improvement)
            bestvalue= calc_cost(selected, coordinates)

            bestG= nx.classes.function.create_empty_copy(G)
            bestG.add_edges_from(selected.edges)
            reps=maxreps
        else:
            reps = reps- 1
            print("REPS LEFT: ", reps)

        if (delta < 0):
            G= nx.classes.function.create_empty_copy(G)
            G.add_edges_from(selected.edges)
            continue;

        #calculate energy, always between 0 and 1
        prob= math.exp( ( -1 * delta) / T )
        print("DELTA", delta)
        print("PROB OF ACCEPTANCE", prob)
        num= random.random()
        if (num <= prob):
            print("TAKING A MISSTEP")
            G= nx.classes.function.create_empty_copy(G)
            G.add_edges_from(selected.edges)
        else:
            print("G not changed")

    return {'best': bestG, 'costs': costs}

#generate a random graph to feed into coordinates, G
coordinates= gen_coordinates()
G= gen_random_graph(coordinates)
print("INITIAL COST", calc_cost(G, coordinates))
costs=[]
costs2=[]

#RUN TABULIST
result= tabu_search(G, coordinates, 15, 10) #G, coordinates, maxreps, maxtabulength
solution= result['best']
costs= result['costs']

#RUN SIMULATED ANNEALING
result2= simulated_annealing(G, coordinates, 15)#G, coordinates, maxreps
solution2= result2['best']
costs2= result2['costs']

#makes sure the graphs are to scale
pos=nx.get_node_attributes(G,'pos')
```

```python

#plot the graphs
nx.draw(solution, pos, with_labels=True)
print("TABUSEARCHCOST", calc_cost(solution, coordinates))
print(solution)

nx.draw(solution2, pos, with_labels=True)
print("SIMULATEANNEALINGCOST", calc_cost(solution2, coordinates))
print(solution2)

nx.draw(G, pos, with_labels=True)
print("ORIGINALCOST", calc_cost(G, coordinates))
print(G)

#plot the double line graph
plt.figure(figsize=(15, 10))
plt.plot(costs)
plt.plot(costs2)
plt.grid()
print("Simulated annealing in orange")

#examine graph degrees
for i in range(N):
  print(solution.degree[i], end= " ")

print("\nSA:\n")
for i in range(N):
  print(solution2.degree[i], end= " ")
```

README

This program was written in python. Just type python3 <filename>.py

You may have to install the requisite libraries.

Sources

[1] “Tabu Search”, Class Handout

[2] “Simulated Annealing”, Class HAndout

[2] https://networkx.org/, NetworkX documentation

[3] https://matplotlib.org/, matplotlib documentation