WSN simulator

A Wireless Sensor Network simulator in Python and C++ (via SWIG).

It basically simulates the communication among nodes and communication with the base station. It has a energy model that helps estimates the network lifetime. It has some pre-defined scenarios (including clustering techniques):

* Direct Communication (from nodes directly to the base station);
* MTE (M. Ettus. System Capacity, Latency, and Power Consumption in Multihop-routed SS-CDMA Wireless Networks. In Radio and Wireless Conference (RAWCON 98), pages 55–58, Aug. 1998)
* LEACH (W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, Energy-efficient communication protocols for wireless sensor networks, In Proceedings of the 33rd Annual Hawaii International Conference on System Sciences (HICSS), Hawaii, USA, January 2000.)
* FCM (D. C. Hoang, R. Kumar and S. K. Panda, "Fuzzy C-Means clustering protocol for Wireless Sensor Networks," 2010 IEEE International Symposium on Industrial Electronics, Bari, 2010, pp. 3477-3482.)

It also implements a modified version of PSO (Particle Swarm Optimization) in order to schedule sleeping slots to every node at every communication round. This implementation is based on (C. Yu, W. Guo and G. Chen, "Energy-balanced Sleep Scheduling Based on Particle Swarm Optimization in Wireless Sensor Network," 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum, Shanghai, 2012, pp. 1249-1255.), but contains improvements, specially concerning the learning of better solutions.

# Running it

1 - Choose your settings in the configuration file (config.py)

2 - python setup.py build\_ext --inplace

3 - python run.py

# Requirements

All non-trivial requirements (the ones you cannot get via pip install) are inside this repository. Trivial requirements include:

* *matplotlib* for plotting graphics
* *numpy* for array manipulation and operations
* *logging* for improved printing (warnings, errors, running information)
* *inspect* (for debugging)
* *multiprocessing* (disabled) for reducing simulation time

# Code structure

Code is structured as follows:

* /python contains all python modules and classes
  + */python*/network/ contains classes that model the energy source, the network, each node and the aggregation models
    - energy\_source.py models battery and “plugged in” sources.
    - network.py models the network, which is basically a list of nodes.
    - node.py represent every node in the network and contains an energy source.
    - aggregation\_model.py offers different aggregations models (from zero cost to total cost).
  + */python*/routing/ contains classes that model all routing protocols: fcm, mte, leach and direct communication. It also contains code for running Dijkstra’s path (used in MTE) and a file named prioridict.py that is also used for MTE. The file routing\_protocol.py contains a class that is extended by all routing protocols and that define a common interface for these protocols (this way the network code that uses these protocols can be generic and not different for each protocol).
    - dijkstra.py implements Dijkstra’s path.
    - direct\_communication.py implements DC.
    - fcm.py implements FCM protocol.
    - mte.py implements MTE protocol.
    - peach.py implements LEACH protocol.
    - prioridict.py is used by dijkstra.py to describe the cost graph.
    - routing\_protocol.py is an interface to all routing protocols.
  + */python*/sleep\_scheduler/ contains a single class that represents an interface to sleep scheduling protocols. It wrapps the algorithms implemented in C++.
    - sleep\_scheduler.py is the wrapper that holds c++ sleep scheduling objects.
  + */python*/utils contains the files grid.py, region.py and regions\_converter.py that are used for modeling and calculating the coverage and overlapping areas.
    - Grid.py represents the grid over the field.
    - region.py represents a single region with its owners.
    - regions\_converter.py reads the network and convert it to regions.
    - tracer.py stores the traces that are plotted or saved.
    - utils.py contains all plot functions plus other utility functions.
* /cc contains all c++ classes and swig interfaces.
  + Ecca.h, ecca.cc, ecca.i: ECCA header, implementation and swing interface.
  + modified\_pso.h, modified\_pso.cc, modified\_pso.i: Modified PSO header, implementation and swing interface.
  + pso.h, pso.cc, pso.i: PSO header, implementation and swing interface.
  + genetic\_algorithm.h, genetic\_algorithm.cc, genetic\_algorithm.i: Genetic algorithm header, implementation and swing interface.
  + Regions.h, regions.cc: stores the regions
  + individual.h, individual.cc: represent and individual in the genetic algorithm
  + optimizer.h, optimizer.cc, optimizer.i: represent an optimizer. PSO, ECCA, Genetic algorithm and Modified PSO inherit from this class.
  + types.h and custom\_types declares some custom types.
* /scripts contain a single python script to plot energy graphs
* /results will be generated when run.py is called and simulation results will be stored there

config.py has no executable code and contains all configurations. It also contains a list called scenarios that stores all scenarios that are going to be simulated. It can also contain executable code to change parameters during the simulation without requiring any user interference. This file is documented and contains examples of use.

Run.py has two functions: run\_scenarios that run all scenarios specified at config.py and run\_parameter\_sweep that perform and example of code that runs the same scenarios but with different paramenters.

Each file is commented and it describes its behavior.

# Listings of all source code

Below, all sources are listed. The sequence ‘--’ indicates the beginning of a new file.

## from directory ./python:

### -- file: + ./python/\_\_init\_\_.py

Empty file.

### -- file: + ./python/network/node.py

import config as cf

import numpy as np

from python.network.energy\_source import \*

from python.utils.utils import \*

class Node(object):

def \_\_init\_\_(self, id, parent = None):

self.pos\_x = np.random.uniform(0, cf.AREA\_WIDTH)

self.pos\_y = np.random.uniform(0, cf.AREA\_LENGTH)

if id == cf.BSID:

self.energy\_source = PluggedIn(self)

else:

self.energy\_source = Battery(self)

self.id = id

self.network\_handler = parent

self.reactivate()

def reactivate(self):

"""Reactivate nodes for next simulation."""

self.alive = 1

self.tx\_queue\_size = 0

self.\_next\_hop = cf.BSID

self.distance\_to\_endpoint = 0

self.amount\_sensed = 0

self.amount\_transmitted = 0

self.amount\_received = 0

self.membership = cf.BSID

# aggregation function determines the cost of forwarding messages

# (in number of bits)

self.aggregation\_function = lambda x: 0

self.time\_of\_death = cf.INFINITY

self.\_is\_sleeping = 0

self.sleep\_prob = 0.0

# for coverage purposes

self.neighbors = []

self.nb\_neighbors = -1

self.exclusive\_radius = 0

@property

def next\_hop(self):

return self.\_next\_hop

@next\_hop.setter

def next\_hop(self, value):

self.\_next\_hop = value

distance = calculate\_distance(self, self.network\_handler[value])

self.distance\_to\_endpoint = distance

@property

def is\_sleeping(self):

if self.is\_head():

self.\_is\_sleeping = 0

return self.\_is\_sleeping

@is\_sleeping.setter

def is\_sleeping(self, value):

"""Cluster heads cannot be put to sleep."""

self.\_is\_sleeping = value if not self.is\_head() else 0

def \_only\_active\_nodes(func):

"""This is a decorator. It wraps all energy consuming methods to

ensure that only active nodes execute this method. Also it automa-

tically calls the battery.

"""

def wrapper(self, \*args, \*\*kwargs):

if self.alive and not self.is\_sleeping:

func(self, \*args, \*\*kwargs)

return 1

else:

return 0

return wrapper

@\_only\_active\_nodes

def update\_sleep\_prob(self):

"""Update the sleep probability. This method supposes that the

endpoint is the cluster head

"""

# cluster heads should not go to sleep. Neither dead nodes.

if self.next\_hop == cf.BSID:

self.sleep\_prob = 0.0

else:

self.sleep\_prob = 0.5

return

@\_only\_active\_nodes

def update\_sleep\_prob2(self, nb\_neighbors):

"""Updates the sleep probability according to paper X."""

if self.next\_hop == cf.BSID:

self.sleep\_prob = 0.0

if nb\_neighbors == 0:

term1 = 0

else:

term1 = PSO\_E\*(nb\_neighbors-1)/nb\_neighbors

if self.distance\_to\_endpoint == 0:

term2 = 0

else:

term2 = PSO\_F\*(self.distance\_to\_endpoint-1)/self.distance\_to\_endpoint

self.sleep\_prob = term1 + term2

def is\_head(self):

if self.next\_hop == cf.BSID and self.id != cf.BSID and self.alive:

return 1

return 0

def is\_ordinary(self):

return 1 if self.next\_hop != cf.BSID and self.id != cf.BSID else 0

@\_only\_active\_nodes

def \_aggregate(self, msg\_length):

logging.debug("node %d aggregating." % (self.id))

# number of bits to be sent increase while forwarding messages

aggregation\_cost = self.aggregation\_function(msg\_length)

self.tx\_queue\_size += aggregation\_cost

# energy model for aggregation

energy = cf.E\_DA \* aggregation\_cost

self.energy\_source.consume(energy)

@\_only\_active\_nodes

def transmit(self, msg\_length=None, destination=None):

logging.debug("node %d transmitting." % (self.id))

if not msg\_length:

msg\_length = self.tx\_queue\_size

msg\_length += cf.HEADER\_LENGTH

if not destination:

destination = self.network\_handler[self.next\_hop]

distance = self.distance\_to\_endpoint

else:

distance = calculate\_distance(self, destination)

# transmitter energy model

energy = cf.E\_ELEC

if distance > cf.THRESHOLD\_DIST:

energy += cf.E\_MP \* (distance\*\*4)

else:

energy += cf.E\_FS \* (distance\*\*2)

energy \*= msg\_length

# automatically call other endpoint receive

destination.receive(msg\_length)

# after the message is sent, queue is emptied

self.tx\_queue\_size = 0

self.amount\_transmitted += msg\_length

self.energy\_source.consume(energy)

@\_only\_active\_nodes

def receive(self, msg\_length):

logging.debug("node %d receiving." % (self.id))

self.\_aggregate(msg\_length - cf.HEADER\_LENGTH)

self.amount\_received += msg\_length

# energy model for receiver

energy = cf.E\_ELEC \* msg\_length

self.energy\_source.consume(energy)

@\_only\_active\_nodes

def sense(self):

self.tx\_queue\_size = cf.MSG\_LENGTH

self.amount\_sensed += cf.MSG\_LENGTH

def battery\_depletion(self):

self.alive = 0

self.sleep\_prob = 0.0

self.time\_of\_death = self.network\_handler.round

self.network\_handler.deaths\_this\_round += 1

### -- file: + ./python/network/\_\_init\_\_.py

Empty file.

### -- file: + ./python/network/aggregation\_model.py

"""Aggregation cost functions. Determine the cost of cluster heads for-

warding messages.

"""

def zero\_cost\_aggregation(msg\_length):

return 0

def total\_cost\_aggregation(msg\_length):

return msg\_length

def linear\_cost\_aggregation(factor):

"""Defines a family of functions."""

return lambda x: int(x\*factor)

def log\_cost\_aggregation(msg\_length):

return int(math.log(msg\_length))

### -- file: + ./python/network/energy\_source.py

import config as cf

import logging

class EnergySource(object):

def \_\_init\_\_(self, parent):

self.energy = cf.INITIAL\_ENERGY

self.node = parent

def recharge(self):

self.energy = cf.INITIAL\_ENERGY

class Battery(EnergySource):

def consume(self, energy):

if self.energy >= energy:

self.energy -= energy

else:

logging.info("node %d: battery is depleted." % (self.node.id))

self.energy = 0

self.node.battery\_depletion()

class PluggedIn(EnergySource):

def consume(self, energy):

pass

### -- file: + ./python/network/network.py

import config as cf

import logging

from python.network.node import \*

from python.utils.grid import \*

import matplotlib.pyplot as plt

from python.utils.utils import \*

from python.utils.tracer import \*

from python.sleep\_scheduling.sleep\_scheduler import \*

from multiprocessing.dummy import Pool as ThreadPool

class Network(list):

"""This class stores a list with all network nodes plus the base sta-

tion. Its methods ensure the network behavior.

"""

def \_\_init\_\_(self, init\_nodes=None):

logging.debug('Instantiating nodes...')

if init\_nodes:

self.extend(init\_nodes)

else:

nodes = [Node(i, self) for i in range(0, cf.NB\_NODES)]

self.extend(nodes)

# last node in nodes is the base station

base\_station = Node(cf.BSID, self)

base\_station.pos\_x = cf.BS\_POS\_X

base\_station.pos\_y = cf.BS\_POS\_Y

self.append(base\_station)

self.\_dict = {}

for node in self:

self.\_dict[node.id] = node

self.perform\_two\_level\_comm = 1

self.round = 0

self.centroids = []

self.routing\_protocol = None

self.sleep\_scheduler\_class = None

self.initial\_energy = self.get\_remaining\_energy()

self.first\_depletion = 0

self.per30\_depletion = 0

self.energy\_spent = []

def reset(self):

"""Set nodes to initial state so the same placement of nodes can be

used by different techniques.

"""

for node in self:

node.energy\_source.recharge()

node.reactivate()

# allows for updates of BS position between simulations

self[-1].pos\_x = cf.BS\_POS\_X

self[-1].pos\_y = cf.BS\_POS\_Y

self.round = 0

self.centroids = []

self.energy\_spent = []

self.routing\_protocol = None

self.sleep\_scheduler\_class = None

self.first\_depletion = 0

self.per30\_depletion = 0

self.perform\_two\_level\_comm = 1

def simulate(self):

tracer = Tracer()

self.routing\_protocol.pre\_communication(self)

all\_alive = 1

percent70\_alive = 1

self.deaths\_this\_round = 0

if self.sleep\_scheduler\_class:

self.\_sleep\_scheduler = SleepScheduler(self, self.sleep\_scheduler\_class)

for round\_nb in range(0, cf.MAX\_ROUNDS):

self.round = round\_nb

print\_args = (round\_nb, self.get\_remaining\_energy())

print("round %d: total remaining energy: %f" % print\_args)

nb\_alive\_nodes = self.count\_alive\_nodes()

if nb\_alive\_nodes == 0:

break

tracer['alive\_nodes'][2].append(nb\_alive\_nodes)

if cf.TRACE\_ENERGY:

tracer['energies'][2].append(self.get\_remaining\_energy())

if self.sleep\_scheduler\_class:

log = self.\_sleep\_scheduler.schedule()

for key, value in log.iteritems():

tracer[key][2].append(value)

self.routing\_protocol.setup\_phase(self, round\_nb)

# check if someone died

if self.deaths\_this\_round != 0:

if all\_alive == 1:

all\_alive = 0

self.first\_depletion = round\_nb

if float(nb\_alive\_nodes)/float(cf.NB\_NODES) < 0.7 and \

percent70\_alive == 1:

percent70\_alive = 0

self.per30\_depletion = round\_nb

# clears dead counter

self.deaths\_this\_round = 0

self.routing\_protocol.broadcast(self)

self.\_run\_round(round\_nb)

tracer['first\_depletion'][2].append(self.first\_depletion)

tracer['30per\_depletion'][2].append(self.per30\_depletion)

return tracer

def \_run\_round(self, round):

"""Run one round. Every node captures using its sensor. Then this

information is forwarded through the intermediary nodes to the base

station.

"""

before\_energy = self.get\_remaining\_energy()

for i in range(0, cf.MAX\_TX\_PER\_ROUND):

self.\_sensing\_phase()

self.\_communication\_phase()

after\_energy = self.get\_remaining\_energy()

self.energy\_spent.append(before\_energy - after\_energy)

def \_sensing\_phase(self):

"""Every alive node captures information using its sensor."""

for node in self.get\_alive\_nodes():

node.sense()

def \_communication\_phase(self):

"""Each node transmits respecting its hierarchy: leaves start the

communication, then cluster heads forward the messages, until all

messages reach the base station. This method works for any hierar-

chy (even for LEACH).

"""

#ordinary\_nodes = self.get\_ordinary\_nodes()

#heads = self.get\_ch\_nodes()

#msg = str("%d ordinary nodes, %d heads." % (len(ordinary\_nodes), len(heads)))

#logging.debug("Hierarchical communication: %s" % (msg))

alive\_nodes = self.get\_alive\_nodes()

if self.perform\_two\_level\_comm == 1:

self.\_two\_level\_comm(alive\_nodes)

else:

self.\_recursive\_comm(alive\_nodes)

def \_recursive\_comm(self, alive\_nodes):

"""Hierarchical communication using recursivity. This method suppo-

ses that there is no cycle in the network (network is a tree).

Otherwise, expect infinite loop.

"""

next\_alive\_nodes = alive\_nodes[:]

for node in alive\_nodes:

#check if other nodes must send info to this node

depends\_on\_other\_node = 0

for other\_node in alive\_nodes:

#if other\_node == node:

# continue

if other\_node.next\_hop == node.id:

depends\_on\_other\_node = 1

break

if not depends\_on\_other\_node:

node.transmit()

next\_alive\_nodes = [n for n in next\_alive\_nodes if n != node]

if len(next\_alive\_nodes) == 0:

return

else:

self.\_recursive\_comm(next\_alive\_nodes)

def \_two\_level\_comm(self, alive\_nodes):

"""This method performs communication supposing that there are only

ordinary nodes and cluster heads, this method is less generic than

its recursive version, but it is faster.

"""

# heads wait for all ordinary nodes, then transmit to BS

for node in self.get\_ordinary\_nodes():

node.transmit()

for node in self.get\_heads():

node.transmit()

def get\_alive\_nodes(self):

"""Return nodes that have positive remaining energy."""

return [node for node in self[0:-1] if node.alive]

def get\_active\_nodes(self):

"""Return nodes that have positive remaining energy and that are

awake."""

is\_active = lambda x: x.alive and not x.is\_sleeping

return [node for node in self[0:-1] if is\_active(node)]

def get\_ordinary\_nodes(self):

return [node for node in self if node.is\_ordinary() and node.alive]

def get\_heads(self, only\_alives=1):

input\_set = self.get\_alive\_nodes() if only\_alives else self

return [node for node in input\_set if node.is\_head()]

def get\_sensor\_nodes(self):

"""Return all nodes except base station."""

return [node for node in self[0:-1]]

def get\_average\_energy(self):

return np.average(self.energy\_spent)

def someone\_alive(self):

"""Finds if there is at least one node alive. It excludes the base station,

which is supposed to be always alive."""

for node in self[0:-1]:

if node.alive == 1:

return 1

return 0

def count\_alive\_nodes(self):

return sum(x.alive for x in self[:-1])

def get\_BS(self):

# intention: make code clearer for non-Python readers

return self[-1]

def get\_node(self, id):

"""By default, we assume that the id is equal to the node's posi-

tion in the list, but that may not be always the case.

"""

return self.\_dict[id]

def notify\_position(self):

"""Every node transmit its position directly to the base station."""

for node in self.get\_alive\_nodes():

node.transmit(msg\_length=cf.MSG\_LENGTH, destination=self.get\_BS())

def broadcast\_next\_hop(self):

"""Base station informs nodes about their next hop."""

base\_station = self.get\_BS()

for node in self.get\_alive\_nodes():

base\_station.transmit(msg\_length=cf.MSG\_LENGTH, destination=node)

def get\_nodes\_by\_membership(self, membership, only\_alives=1):

"""Returns all nodes that belong to this membership/cluster."""

input\_set = self.get\_alive\_nodes() if only\_alives else self

condition = lambda node: node.membership == membership and node.id != cf.BSID

return [node for node in input\_set if condition(node)]

def get\_remaining\_energy(self, ignore\_nodes=None):

"""Returns the sum of the remaining energies at all nodes."""

set = self.get\_alive\_nodes()

if len(set) == 0:

return 0

if ignore\_nodes:

set = [node for node in set if node not in ignore\_nodes]

transform = lambda x: x.energy\_source.energy

energies = [transform(x) for x in set]

return sum(x for x in energies)

def set\_aggregation\_function(self, function):

"""Sets the function that determines the cost of aggregation."""

for node in self:

node.aggregation\_function = function

def split\_in\_clusters(self, nb\_clusters=cf.NB\_CLUSTERS):

"""Split this nodes object into other nodes objects that contain only

information about a single cluster."""

clusters = []

for cluster\_idx in range(0, nb\_clusters):

nodes = self.get\_nodes\_by\_membership(cluster\_idx)

cluster = Network(init\_nodes=nodes)

cluster.append(self.get\_BS())

clusters.append(cluster)

return clusters

def \_calculate\_nb\_neighbors(self, target\_node):

"""Calculate the number of neighbors given the sensor coverage

radius.

"""

# if number of neighbors was calculated at least once

# skips calculating the distance

if target\_node.nb\_neighbors != -1:

# only check if there are dead nodes

all\_neighbors = target\_node.neighbors

nb\_dead\_neighbors = sum(1 for x in all\_neighbors if not x.alive)

target\_node.neighbors[:] = [x for x in all\_neighbors if x.alive]

return target\_node.nb\_neighbors - nb\_dead\_neighbors

nb\_neighbors = 0

shortest\_distance = cf.COVERAGE\_RADIUS\*2

for node in self.get\_alive\_nodes():

if node == target\_node:

continue

distance = calculate\_distance(target\_node, node)

if distance <= cf.COVERAGE\_RADIUS:

nb\_neighbors += 1

target\_node.neighbors.append(node)

if distance < shortest\_distance:

shortest\_distance = distance

if shortest\_distance != cf.INFINITY:

exclusive\_radius = shortest\_distance - cf.COVERAGE\_RADIUS

if exclusive\_radius < 0:

exclusive\_radius = 0.0

node.nb\_neighbors = nb\_neighbors

node.exclusive\_radius = exclusive\_radius

def update\_neighbors(self):

for node in self.get\_alive\_nodes():

self.\_calculate\_nb\_neighbors(node)

self.update\_sleep\_prob()

def update\_sleep\_prob(self):

for node in self.get\_alive\_nodes():

node.update\_sleep\_prob()

### -- file: + ./python/utils/tracer.py

import config as cf

"""Utility class used to store local traces."""

class Tracer(dict):

def \_\_init\_\_(self):

rounds\_label = 'Rounds'

# every tuple has a y-axis label, x-axis label, list with values,

# boolean that indicates if it is plotable and if is printable

# lifetime/energy-related log

self['alive\_nodes'] = ('Number of alive nodes', rounds\_label, [], 1, 0)

if cf.TRACE\_ENERGY:

self['energies'] = ('Energy (J)' , rounds\_label, [], 1, 0)

self['first\_depletion'] = ('First depletion' , rounds\_label, [], 0, 0)

self['30per\_depletion'] = ('30 percent depletion' , rounds\_label, [], 0, 0)

# coverage-related log

self['coverage'] = ('Coverate rate' , rounds\_label, [], 0, 1)

self['overlapping'] = ('Overlapping rate' , rounds\_label, [], 0, 1)

self['nb\_sleeping'] = ('% of sleeping nodes' , rounds\_label, [], 0, 1)

# learning-related log

self['initial\_fitness'] = ('Initial learning' , rounds\_label, [], 0, 1)

self['final\_fitness'] = ('Final learning' , rounds\_label, [], 0, 1)

self['term1\_initial'] = ('term1 learning' , rounds\_label, [], 0, 1)

self['term2\_initial'] = ('term2 learning' , rounds\_label, [], 0, 1)

self['term1\_final'] = ('term1 final' , rounds\_label, [], 0, 1)

self['term2\_final'] = ('term2 final' , rounds\_label, [], 0, 1)

### -- file: + ./python/utils/region.py

class Region(object):

"""A region represents all regions that have the same owners (are co-

vered by the same nodes. Therefore it may represent disjoint regions

as a single region.

"""

def \_\_init\_\_(self, area, owners=set()):

# total area of the region

self.area = area

# nodes that cover this region

self.owners = owners

def \_\_str\_\_(self):

to\_print = ""

for owner in self.owners:

to\_print += " " + str(owner)

return to\_print + " " + str(self.area)

### -- file: + ./python/utils/test\_plot.py

import matplotlib.pyplot as plt

from matplotlib.colors import BoundaryNorm

from matplotlib.ticker import MaxNLocator

import numpy as np

# make these smaller to increase the resolution

dx, dy = 0.05, 0.05

# generate 2 2d grids for the x & y bounds

y, x = np.mgrid[slice(1, 5 + dy, dy),

slice(1, 5 + dx, dx)]

print(x)

print(y)

z = np.sin(x)\*\*10 + np.cos(10 + y\*x) \* np.cos(x)

print(z)

# x and y are bounds, so z should be the value \*inside\* those bounds.

# Therefore, remove the last value from the z array.

z = z[:-1, :-1]

levels = MaxNLocator(nbins=15).tick\_values(z.min(), z.max())

# pick the desired colormap, sensible levels, and define a normalization

# instance which takes data values and translates those into levels.

cmap = plt.get\_cmap('PiYG')

norm = BoundaryNorm(levels, ncolors=cmap.N, clip=True)

fig, ax1 = plt.subplots(nrows=1)

# contours are \*point\* based plots, so convert our bound into point

# centers

cf = ax1.contourf(x[:-1, :-1] + dx/2.,

y[:-1, :-1] + dy/2., z, levels=levels,

cmap=cmap)

fig.colorbar(cf, ax=ax1)

ax1.set\_title('contourf with levels')

# adjust spacing between subplots so `ax1` title and `ax0` tick labels

# don't overlap

fig.tight\_layout()

plt.show()

### -- file: + ./python/utils/regions\_converter.py

import logging

import numpy as np

import config as cf

from python.utils.utils import \*

from python.utils.region import \*

"""This module classes are used to calculate the network coverage area

and the network overlapping area. They are optimized in order to speed

up simulation time and therefore coding simplicity is sometimes compro-

mised.

"""

class RegionsConverter(list):

"""Helps to convert a grid to regions."""

\_area\_single\_pixel = cf.GRID\_PRECISION\*\*2

\_exclude\_area\_if = 0.005 # exclude regions with less area than this

def \_\_init\_\_(self, grid):

logging.info('Creating Regions instance.')

self.extend(grid.\_exclusive\_regions)

self.\_grid2regions(grid.\_pixels)

#self.\_remove\_small\_regions()

self.\_extract\_exclusive\_regions()

logging.info(self)

def \_grid2regions(self, pixels):

"""Convert a grid to regions."""

logging.info('converting grid to regions.')

for x, line in pixels.iteritems():

for y, pixel in line.iteritems():

owners = set(pixels[x][y])

region = self.\_get\_region(owners)

if region:

# increase area

region.area += self.\_area\_single\_pixel

else:

# create region

new\_region = Region(self.\_area\_single\_pixel, owners)

self.append(new\_region)

def \_extract\_exclusive\_regions(self):

"""Separate regions that overlap from regions that have a single

owner. This aims to improve performance.

"""

logging.info('extracting exclusive regions.')

self.\_exclusive\_regions = {}

del\_idx = []

for idx, region in enumerate(self):

if len(region.owners) == 1:

owner = list(region.owners)[0]

self.\_exclusive\_regions[owner] = region.area

del\_idx.append(idx)

for idx in del\_idx[::-1]:

del self[idx]

def \_remove\_small\_regions(self):

"""Removing small regions improves performance."""

logging.info('removing small regions.')

total\_coverage = self.\_get\_total\_coverage()

#print("total coverage %f" %(total\_coverage))

del\_idx = []

for idx, region in enumerate(self):

percentage = region.area/total\_coverage

if percentage < self.\_exclude\_area\_if:

del\_idx.append(idx)

for idx in del\_idx[::-1]:

del self[idx]

def \_get\_region(self, owners):

"""Return region if owners match otherwise return 0.

Args:

owners (list): List of node's ids

"""

for region in self:

if owners == region.owners:

return region

return 0

def \_\_str\_\_(self):

"""Print all regions."""

sum = 0.0

regions\_str = ''

for owner, area in self.\_exclusive\_regions.iteritems():

regions\_str += "%s, %f \n" %(str(owner), area)

sum += area

for region in self:

regions\_str += "%s, %f \n" %(str(region.owners), region.area)

sum += region.area

regions\_str += "total area: %f\n" %(sum)

return regions\_str

def \_get\_total\_coverage(self):

"""Sums the areas from every region."""

coverage = 0.0

if hasattr(self, '\_exclusive\_regions'):

for area in self.\_exclusive\_regions:

coverage += area

for region in self:

coverage += region.area

return coverage

def \_check(self, exclusive, overlapping):

for owner, area in exclusive.iteritems():

assert area >= 0.0, "Negative region found!"

for region in self:

assert region.area >= 0.0, "Negative region found!"

def convert(self):

overlapping\_regions = []

for region in self:

overlapping\_regions.append((list(region.owners), region.area))

self.\_check(self.\_exclusive\_regions, overlapping\_regions)

return self.\_exclusive\_regions, overlapping\_regions

### -- file: + ./python/utils/\_\_init\_\_.py

Empty file.

### -- file: + ./python/utils/utils.py

import matplotlib.pyplot as plt

import math

import pandas as pd

import numpy as np

import os

import time

from numpy import linspace, meshgrid

from matplotlib.mlab import griddata

import config as cf

from python.network.network import \*

plt.rcParams.update({'font.size': 14})

def calculate\_nb\_clusters(avg\_distance\_to\_BS):

"""Calculate the optimal number of clusters for FCM."""

term1 = math.sqrt(cf.NB\_NODES)/(math.sqrt(2\*math.pi))

term2 = cf.THRESHOLD\_DIST

term3 = cf.AREA\_WIDTH/(avg\_distance\_to\_BS\*\*2)

return int(term1\*term2\*term3)

def calculate\_distance(node1, node2):

"""Calculate the Euclidean distance between two nodes."""

x1 = node1.pos\_x

y1 = node1.pos\_y

x2 = node2.pos\_x

y2 = node2.pos\_y

return calculate\_distance\_point(x1, y1, x2, y2)

def calculate\_distance\_point(x1, y1, x2, y2):

"""Calculate the Euclidean distance between two points."""

return math.sqrt((x1 - x2)\*\*2 + (y1 - y2)\*\*2)

def print\_positions(nodes):

# check positions

for node in nodes:

print("%d %d" %(node.pos\_x, node.pos\_y))

def plot\_curves(curves):

"""Generic plotter of curves."""

assert len(curves) <= 7, "More plots (%d) than colors." %len(curves)

colors = ['b-', 'r-', 'k-', 'y-', 'g-', 'c-', 'm-']

color\_idx = 0

for scenario, curve in curves.iteritems():

X = range(0, len(curve))

plt.plot(X, curve, colors[color\_idx], label=scenario)

color\_idx += 1

plt.show()

def save2csv\_raw(traces):

to\_csv = []

dir\_path = cf.RESULTS\_PATH + time.strftime("%Y-%m-%d\_%H:%M:%S") + '/'

os.makedirs(dir\_path)

for scenario\_name, tracer in traces.iteritems():

for i, val in enumerate(tracer['coverage'][2]):

tmp = {'cov' : val,

'sleep' : tracer['nb\_sleeping'][2][i]}

to\_csv.append(tmp)

df = pd.DataFrame(to\_csv)

df.to\_csv(dir\_path + scenario\_name + '-cov\_vs\_sleeping.csv')

def print\_coverage\_info(traces):

for scenario\_name, tracer in traces.iteritems():

args = (scenario\_name, tracer['first\_depletion'][2][0])

print("%s: first depletion at %d" % args)

args = (scenario\_name, tracer['30per\_depletion'][2][0])

print("%s: 30 percent depletion at %d" % args)

for trace\_name, trace in tracer.iteritems():

if not trace[4]:

continue

values = np.array(trace[2])

mean = np.nanmean(values)

stdev = np.nanstd(values)

args = (scenario\_name, trace\_name, mean, stdev)

print("%s: %s avg (std): %f (%f)" % args)

def save2csv(traces):

to\_csv = []

for scenario\_name, tracer in traces.iteritems():

tmp = {'scenario\_name': scenario\_name,

'first\_depletion': tracer['first\_depletion'][2][0],

'30per\_depletion': tracer['30per\_depletion'][2][0]}

for trace\_name, trace in tracer.iteritems():

if not trace[4]:

continue

values = np.array(trace[2])

mean = np.nanmean(values)

stdev = np.nanstd(values)

tmp[trace\_name+ ' (mean)'] = mean

tmp[trace\_name+ ' (stdev)'] = stdev

to\_csv.append(tmp)

df = pd.DataFrame(to\_csv)

dir\_path = cf.RESULTS\_PATH + time.strftime("%Y-%m-%d\_%H:%M:%S") + '/'

os.makedirs(dir\_path)

df.to\_csv(dir\_path + 'results\_summary.csv')

def plot\_traces(traces):

first\_tracer = traces.itervalues().next()

nb\_columns = len([1 for k, v in first\_tracer.iteritems() if v[3]])

fig, ax = plt.subplots(nrows=1, ncols=nb\_columns)

colors = ['b', 'r', 'k', 'y', 'g', 'c', 'm']

line\_style = ['-', '--', '-.', ':']

color\_idx = 0

line\_idx = 0

for scenario, tracer in traces.iteritems():

subplot\_idx = 1

for trace\_name, trace in tracer.iteritems():

if not trace[3]:

continue

ax = plt.subplot(1, nb\_columns, subplot\_idx)

#ax.set\_title(trace\_name)

X = range(0, len(trace[2]))

color\_n\_line = colors[color\_idx] + line\_style[line\_idx]

plt.plot(X, trace[2], color\_n\_line, label=scenario)

plt.xlabel(trace[1])

plt.ylabel(trace[0])

plt.legend(fontsize=11)

subplot\_idx += 1

color\_idx = (color\_idx+1)%len(colors)

line\_idx = (line\_idx+1)%len(line\_style)

plt.xlim(xmin=0)

plt.ylim(ymin=0)

plt.grid(b=True, which='major', color='0.6', linestyle='--')

plt.show()

def plot\_nodes\_plane(nodes):

X\_ch = [node.pos\_x for node in nodes if node.is\_head()]

Y\_ch = [node.pos\_y for node in nodes if node.is\_head()]

X\_or = [node.pos\_x for node in nodes if node.is\_ordinary()]

Y\_or = [node.pos\_y for node in nodes if node.is\_ordinary()]

X\_de = [node.pos\_x for node in nodes if not node.alive]

Y\_de = [node.pos\_y for node in nodes if not node.alive]

plt.scatter(X\_ch, Y\_ch, color='b')

plt.scatter(X\_or, Y\_or, color='r')

plt.scatter(X\_de, Y\_de, color='k')

plt.show()

def plot\_clusters(network):

colors = ['b', 'k', 'y', 'g', 'm', 'c']

# print clusters

plt.figure()

for cluster\_id in range(0, cf.NB\_CLUSTERS):

cluster = network.get\_nodes\_by\_membership(cluster\_id, only\_alives=0)

X = [node.pos\_x for node in cluster if not node.is\_head()]

Y = [node.pos\_y for node in cluster if not node.is\_head()]

color\_ref = float(cluster\_id)/cf.NB\_CLUSTERS\*0.6

plt.scatter(X, Y, color=colors[cluster\_id%len(colors)])

x\_border = [0.0 for y in range(0, int(cf.AREA\_LENGTH))]

y\_border = [y for y in range(0, int(cf.AREA\_LENGTH))]

x\_border.extend([cf.AREA\_WIDTH for y in range(0, int(cf.AREA\_LENGTH))])

y\_border.extend([y for y in range(0, int(cf.AREA\_LENGTH))])

x\_border.extend([x for x in range(0, int(cf.AREA\_WIDTH))])

y\_border.extend([0.0 for x in range(0, int(cf.AREA\_WIDTH))])

x\_border.extend([x for x in range(0, int(cf.AREA\_WIDTH))])

y\_border.extend([cf.AREA\_LENGTH for x in range(0, int(cf.AREA\_WIDTH))])

z\_border = [0 for x in range(0, int(2\*cf.AREA\_LENGTH + 2\*cf.AREA\_WIDTH))]

for cluster\_id in range(0, cf.NB\_CLUSTERS):

X = [node.pos\_x for node in network[0:-1]]

Y = [node.pos\_y for node in network[0:-1]]

Z = [1 if node.membership==cluster\_id else 0 for node in network[0:-1]]

X, Y, Z = grid(X, Y, Z)

plt.contour(X, Y, Z, 1, colors='0.6')

# print centroids

#heads = network.get\_heads(only\_alives=0)

heads = [x for x in network.centroids]

X = [node.pos\_x for node in heads]

Y = [node.pos\_y for node in heads]

plt.scatter(X, Y, color='r', marker='^', s=80)

# print BS

X = [network.get\_BS().pos\_x]

Y = [network.get\_BS().pos\_y]

plt.scatter(X, Y, color='r', marker='x', s=80)

plt.xlim(xmin=0)

plt.ylim(ymin=0)

plt.xlim(xmax=cf.AREA\_WIDTH)

plt.ylim(ymax=cf.AREA\_LENGTH)

plt.show()

def plot\_time\_of\_death(network):

"""Plot time of death as a colormap."""

x = [node.pos\_x for node in network[0:-1]]

y = [node.pos\_y for node in network[0:-1]]

z = [node.time\_of\_death for node in network[0:-1]]

X, Y, Z = grid(x, y, z)

c = plt.contourf(X, Y, Z)

cbar = plt.colorbar(c)

cbar.ax.set\_ylabel('number of rounds until full depletion')

# print centroids

#heads = network.get\_heads(only\_alives=0)

heads = [x for x in network.centroids]

X = [node.pos\_x for node in heads]

Y = [node.pos\_y for node in heads]

plt.scatter(X, Y, color='r', marker='^', s=80)

# print BS

X = [network.get\_BS().pos\_x]

Y = [network.get\_BS().pos\_y]

plt.scatter(X, Y, color='r', marker='x', s=80)

# plot nodes

for cluster\_id in range(0, cf.NB\_CLUSTERS):

cluster = network.get\_nodes\_by\_membership(cluster\_id, only\_alives=0)

X = [node.pos\_x for node in cluster if not node.is\_head()]

Y = [node.pos\_y for node in cluster if not node.is\_head()]

color\_ref = float(cluster\_id)/cf.NB\_CLUSTERS\*0.6

plt.scatter(X, Y, color='0.6')

plt.xlim(xmin=0)

plt.ylim(ymin=0)

plt.xlim(xmax=cf.AREA\_WIDTH)

plt.ylim(ymax=cf.AREA\_LENGTH)

plt.show()

def log\_curves(curves):

"""Write results."""

dir\_path = cf.RESULTS\_PATH + time.strftime("%Y-%m-%d\_%H:%M:%S") + '/'

os.makedirs(dir\_path)

# write alive\_nodes vs round number

df = pd.DataFrame.from\_dict(curves)

df.to\_csv(dir\_path + 'alive\_nodes.txt')

# write nodes position and time of death

def log\_coverages(pso\_wrapper):

dir\_path = cf.RESULTS\_PATH + time.strftime("%Y-%m-%d\_%H:%M:%S") + '/'

os.makedirs(dir\_path)

df = pd.DataFrame.from\_dict(pso\_wrapper.\_cov\_log)

df.to\_csv(dir\_path + 'cov\_log.txt')

def grid(x, y, z, resX=100, resY=100):

"Convert 3 column data to matplotlib grid"

xi = linspace(min(x), max(x), resX)

yi = linspace(min(y), max(y), resY)

Z = griddata(x, y, z, xi, yi, interp='linear')

X, Y = meshgrid(xi, yi)

return X, Y, Z

### -- file: + ./python/utils/cov\_plot.py

"""

========

Barchart

========

A bar plot with errorbars and height labels on individual bars

"""

import numpy as np

import matplotlib.pyplot as plt

plt.rcParams.update({'font.size': 14})

N = 3

cov\_means = (0.6509, 0.8648, 0.9445)

cov\_std = (0.0637, 0.0451, 0.0206)

ind = np.arange(N) # the x locations for the groups

width = 0.20 # the width of the bars

fig, ax = plt.subplots()

rects1 = ax.bar(ind + 0\*width, cov\_means, width, color='r', yerr=cov\_std)

over\_means = (0.4486, 0.3238, 0.3238)

over\_std = (0.2113, 0.0980, 0.0982)

rects2 = ax.bar(ind + 1\*width, over\_means, width, color='b', yerr=over\_std)

sleep\_means = (0.3572, 0.2949, 0.1801)

sleep\_std = (0.0661, 0.0467, 0.033)

rects3 = ax.bar(ind + 2\*width, sleep\_means, width, color='g', yerr=sleep\_std)

alive\_means = (1.09257, 1, 0.8695)

rects4 = ax.bar(ind + 3\*width, alive\_means, width, color='y')

# add some text for labels, title and axes ticks

ax.set\_ylabel('Rates')

ax.set\_xticks(ind + 1.5\*width)

ax.set\_xticklabels((r'$\alpha=1, \beta=0$', r'$\alpha=0.5, \beta=0.5$', r'$\alpha=0, \beta=1$'))

ax.legend((rects1[0], rects2[0], rects3[0], rects4[0]), ('Coverage rate', 'Overlapping rate', 'Sleeping rate', 'Lifetime proportion'), loc='upper center', bbox\_to\_anchor=(0.5, -0.1),

fancybox=True, shadow=False, ncol=2)

def autolabel(rects):

"""

Attach a text label above each bar displaying its height

"""

for rect in rects:

height = rect.get\_height()

ax.text(rect.get\_x() + rect.get\_width()/2., 1.05\*height,

'%d' % int(height),

ha='center', va='bottom')

#autolabel(rects1)

#autolabel(rects2)

#autolabel(rects3)

#autolabel(rects4)

plt.show()

### -- file: + ./python/utils/grid.py

import numpy as np

import logging

import config as cf

from python.utils.utils import \*

from python.utils.region import \*

def \_adjust2grid(pos):

"""Reallocate the x or y value to the grid.

Ex.: 5.454545 -> 5.45 (if GRID\_PRECISION == 0.01)

"""

return cf.GRID\_PRECISION\*int(pos/cf.GRID\_PRECISION)

"""This module classes are used to calculate the network coverage area

and the network overlapping area. They are optimized in order to speed

up simulation time and therefore coding simplicity is sometimes compro-

mised.

"""

class Grid(object):

"""This class is used to calculate the network coverage area and the

network overlapping area. This is done by using a pixel grid where

each node coverage area is 'painted' on the grid. Each pixel is an

infinitesimal point on the map. The area calculated by this method is

an approximation that depends on the size of the grid. Therefore the

accuracy is configurable.

"""

def \_\_init\_\_(self):

# pixels is a dictionary that follows the pattern:

# {pos\_x0: {pos\_y0: [node\_id0, node\_id1], pos\_y1: []}, pos\_x1:}

# only painted pixels are added to grid to save memory

self.\_pixels = {}

# for nodes that have no neighbors we just store the area

# it supposes that the number of neighbors attribute (Node) was

# already calculated

self.\_exclusive\_regions = []

def \_paint\_pixel(self, x, y, id):

"""Paint pixel if not painted yet, or add node id to painted node.

"""

if x not in self.\_pixels: # add line if it does not exist

self.\_pixels[x] = {}

if y in self.\_pixels[x]:

owners = self.\_pixels[x][y]

# pixel is already painted. Annotate node id to it.

owners.append(id)

logging.debug("overlapping pixel %s %s" %(x, y))

else:

# paint new pixel

self.\_pixels[x][y] = [id]

logging.debug("painting pixel %s %s" %(x, y))

def add\_node(self, node, coverage\_radius):

"""Paint the node on the grid. Assumes a circular radius. It is

optimized to skip exclusive regions (i.e. regions that are covered

by a single node.

"""

logging.info("adding node %d to grid" % (node.id))

# covers a rectangular area around the circle, but paints only area

# inside the radius

initial\_x = \_adjust2grid(node.pos\_x - coverage\_radius)

initial\_y = \_adjust2grid(node.pos\_y - coverage\_radius)

final\_x = \_adjust2grid(node.pos\_x + coverage\_radius)

final\_y = \_adjust2grid(node.pos\_y + coverage\_radius)

if initial\_x < 0.0:

initial\_x = 0.0

if initial\_y < 0.0:

initial\_y = 0.0

if final\_x > \_adjust2grid(cf.AREA\_WIDTH):

final\_x = \_adjust2grid(cf.AREA\_WIDTH)

if final\_y > \_adjust2grid(cf.AREA\_LENGTH):

final\_y = \_adjust2grid(cf.AREA\_LENGTH)

for pixel\_x in np.arange(initial\_x, final\_x, cf.GRID\_PRECISION):

for pixel\_y in np.arange(initial\_y, final\_y, cf.GRID\_PRECISION):

distance = calculate\_distance\_point(pixel\_x, pixel\_y,

node.pos\_x, node.pos\_y)

if distance < coverage\_radius:

self.\_paint\_pixel(str(\_adjust2grid(pixel\_x)),

str(\_adjust2grid(pixel\_y)),

node.id)

### -- file: + ./python/sleep\_scheduling/sleep\_scheduler.py

import numpy as np

import logging

from time import time

import config as cf

from python.utils.grid import \*

from python.utils.regions\_converter import \*

from python.utils.utils import \*

from cc.genetic\_algorithm import \*

from cc.pso import \*

from cc.modified\_pso import \*

from cc.ecca import \*

from multiprocessing.dummy import Pool as ThreadPool

"""Wraps the C++ instance that executes the PSO and also calculates

all coverage information.

"""

class SleepScheduler(object):

def \_\_init\_\_(self, cluster, optimizer\_class):

# need to update neighbors through this method, so grid can be

# generated faster

cluster.update\_neighbors()

self.\_cluster = cluster

grid = Grid()

for node in cluster.get\_sensor\_nodes():

grid.add\_node(node, cf.COVERAGE\_RADIUS)

regions\_converter = RegionsConverter(grid)

exclusive\_regions, overlapping\_regions = regions\_converter.convert()

config\_int = {'NB\_INDIVIDUALS': cf.NB\_INDIVIDUALS,

'MAX\_ITERATIONS': cf.MAX\_ITERATIONS}

config\_float = {'FITNESS\_ALPHA' : cf.FITNESS\_ALPHA,

'FITNESS\_BETA' : cf.FITNESS\_BETA,

'FITNESS\_GAMMA' : cf.FITNESS\_GAMMA,

'WMAX' : cf.WMAX,

'WMIN' : cf.WMIN}

configuration = (config\_int, config\_float)

ids = [node.id for node in cluster.get\_sensor\_nodes()]

self.\_optimizer = optimizer\_class(exclusive\_regions, overlapping\_regions,

ids, configuration)

def schedule(self):

"""Runs PSO to decide which nodes in the cluster will sleep. The cur-

rent cluster head should not be put to sleep, otherwise all informa-

tion for that node is lost.

"""

# when a single node (CH) is alive you must keep it awake

if (self.\_cluster.count\_alive\_nodes() <= 1):

return {}

membership = self.\_cluster[0].membership

logging.debug("running sleep scheduling for cluster %d" % (membership))

# no need to run sleep scheduling if all nodes are dead

# calculate sleep probability for each node

self.\_cluster.update\_sleep\_prob()

sensor\_nodes = self.\_cluster.get\_sensor\_nodes()

node\_ids = [node.id for node in sensor\_nodes]

energies = [node.energy\_source.energy for node in sensor\_nodes]

#head\_id = (self.\_cluster.get\_heads())[0].id

best\_configuration = self.\_optimizer.Run(energies)

best\_coverage = self.\_optimizer.GetBestCoverage()

best\_overlapping = self.\_optimizer.GetBestOverlapping()

learning\_trace = self.\_optimizer.GetLearningTrace()

term1\_trace = self.\_optimizer.GetTerm1Trace()

term2\_trace = self.\_optimizer.GetTerm2Trace()

#print("best cov: %f, best over: %f" %(best\_coverage, best\_overlapping))

#print("init: %f, final: %f" %(learning\_trace[0], learning\_trace[-1]))

#print(sum(ord(x) for x in best\_configuration))

#plot\_curves({'scenario': learning\_trace})

#logging.info('search finished.')

#print(self.\_best\_configuration)

# actually put nodes to sleep

nb\_alive = len(self.\_cluster.get\_alive\_nodes())

nb\_sleeping = sum(ord(y) for x, y in zip(self.\_cluster, best\_configuration) if x.alive)

sleeping\_rate = float(nb\_sleeping)/float(nb\_alive)

#print("coverage %f, active rate %f" %(best\_coverage, 1-sleeping\_rate))

log = {}

log['coverage'] = best\_coverage

log['overlapping'] = best\_overlapping

log['nb\_sleeping'] = sleeping\_rate

log['initial\_fitness'] = learning\_trace[0]

log['final\_fitness'] = learning\_trace[-1]

log['term1\_initial'] = term1\_trace[0]

log['term1\_final'] = term1\_trace[-1]

log['term2\_initial'] = term2\_trace[0]

log['term2\_final'] = term2\_trace[-1]

#print("sleeping nodes %d out of %d" %(nb\_sleeping\_nodes, len(best\_configuration)))

#print([x.id for x in self.\_cluster if x.alive])

# set cluster's nodes to sleep accordingly to optimization algorithm

for idx, node in enumerate(sensor\_nodes):

node.is\_sleeping = ord(best\_configuration[idx])

return log

### -- file: + ./python/sleep\_scheduling/\_\_init\_\_.py

Empty file.

### -- file: + ./python/routing/mte.py

# -\*- coding: utf-8 -\*-

import logging, sys

from python.routing.dijkstra import \*

from python.utils.utils import \*

from python.network.node import \*

from python.network.network import Network

from python.routing.routing\_protocol import \*

import config as cf

class MTE(RoutingProtocol):

def \_find\_shortest\_path(self, network):

"""The base station decides the next-hop for every node using

Dijkstra's algorithm (shortest path). Then it broadcasts this infor-

mation to all network. This function builds a graph with weights/cost

related to each pair of network. The weights are not the Euclidean dis-

nces, but rather a funcion of distances. If the distance is greater

than THRESHOLD\_DIST d^4 i used, otherwise d^2 is used. This comes

from the energy model (see reference).

Reference:

M. Ettus. System Capacity, Latency, and Power Consumption in Multi-

hop-routed SS-CDMA Wireless Networks. In Radio and Wireless Confe-

rence (RAWCON 98), pages 55–58, Aug. 1998

"""

logging.info('MTE: setup phase')

# generate cost graph only for alive network (in dict form):

# origin\_id: {dest\_id1: cost1, dest\_id2: cost2, ...}, ...

alive\_nodes = network.get\_alive\_nodes()

alive\_nodes\_and\_BS = alive\_nodes + [network.get\_BS()]

G = {}

for node in alive\_nodes\_and\_BS:

G[node.id] = {}

for other in alive\_nodes\_and\_BS:

if other == node:

continue

distance = calculate\_distance(node, other)

cost = distance\*\*2 if distance < cf.THRESHOLD\_DIST else distance\*\*4

G[node.id][other.id] = cost

# calculate shortest path and set next\_hop accordingly

done = []

while len(alive\_nodes) != 0:

starting\_node = alive\_nodes[0]

shortest\_path = shortestPath(G, starting\_node.id, cf.BSID)

for i, id in enumerate(shortest\_path):

if id == cf.BSID or id in done:

break

network.get\_node(id).next\_hop = shortest\_path[i+1]

#network[id].next\_hop = shortest\_path[i+1]

alive\_nodes = [node for node in alive\_nodes if node.id != id]

done.append(id)

def \_setup\_phase(self, network):

"""Every node communicate its position to the base station. Then the

BS uses MTE to choose the routes and broadcasts this information to

the network. Finally, a round is executed.

"""

if network.deaths\_this\_round != 0:

self.\_find\_shortest\_path(network)

network.broadcast\_next\_hop()

def \_initial\_setup(self, network):

network.perform\_two\_level\_comm = 0

self.\_find\_shortest\_path(network)

network.broadcast\_next\_hop()

### -- file: + ./python/routing/fcm.py

import skfuzzy

import numpy as np

import logging, sys

from python.routing.mte import \*

from python.utils.utils import \*

from python.network.node import \*

from python.network.network import Network

from python.routing.routing\_protocol import \*

from python.sleep\_scheduling.sleep\_scheduler import \*

import config as cf

"""Every node communicate its position to the base station. Then the

BS uses FCM to define clusters and broadcast this information to the

network. Finally, a round is executed.

"""

class FCM(RoutingProtocol):

#def \_initial\_setup(self, network):

def \_setup\_phase(self, network):

"""The base station uses Fuzzy C-Means to clusterize the network. The

optimal number of clusters is calculated. Then FCM is used to select

the heads (centroids) for each cluster (only in the initial round).

Then each cluster head chooses a new cluster head for the next round.

Referece:

D. C. Hoang, R. Kumar and S. K. Panda, "Fuzzy C-Means clustering

protocol for Wireless Sensor Networks," 2010 IEEE International

Symposium on Industrial Electronics, Bari, 2010, pp. 3477-3482.

"""

logging.debug('FCM: setup phase')

sensor\_nodes = network.get\_sensor\_nodes()

# calculate the average distance to the BS

transform = lambda node: calculate\_distance(node, network.get\_BS())

distances\_to\_BS = [transform(node) for node in sensor\_nodes]

avg\_distance\_to\_BS = np.average(distances\_to\_BS)

nb\_clusters = calculate\_nb\_clusters(avg\_distance\_to\_BS)

# using a constant because calculating this value on-the-fly gives

# different result than the paper

nb\_clusters = cf.NB\_CLUSTERS

# format data to shape expected by skfuzzy API

data = [[node.pos\_x, node.pos\_y] for node in network[0:-1]]

data = np.array(data).transpose()

centroids, membership = skfuzzy.cluster.cmeans(data, nb\_clusters,

cf.FUZZY\_M, error=0.005,

maxiter=1000,

init=None)[0:2]

# assign node nearest to centroid as cluster head

heads = []

# also annotates centroids to network

network.centroids = []

for cluster\_id, centroid in enumerate(centroids):

tmp\_centroid = Node(0)

tmp\_centroid.pos\_x = centroid[0]

tmp\_centroid.pos\_y = centroid[1]

network.centroids.append(tmp\_centroid)

nearest\_node = None

shortest\_distance = cf.INFINITY

for node in network[0:-1]:

distance = calculate\_distance(node, tmp\_centroid)

if distance < shortest\_distance:

nearest\_node = node

shortest\_distance = distance

nearest\_node.next\_hop = cf.BSID

nearest\_node.membership = cluster\_id

heads.append(nearest\_node)

# assign ordinary network to cluster heads using fcm

for i, node in enumerate(network[0:-1]):

if node in heads: # node is already a cluster head

continue

cluster\_id = np.argmax(membership[:,i])

node.membership = cluster\_id

head = [x for x in heads if x.membership == cluster\_id][0]

node.next\_hop = head.id

self.head\_rotation(network)

#def \_setup\_phase(self, network):

def head\_rotation(self, network):

logging.debug('FCM: head rotation')

# head rotation

# current cluster heads choose next cluster head with the most

# residual energy and nearest to the cluster centroid

for cluster\_id in range(0, cf.NB\_CLUSTERS):

cluster = network.get\_nodes\_by\_membership(cluster\_id)

# check if there is someone alive in this cluster

if len(cluster) == 0:

continue

# someone is alive, find node with highest energy in the cluster

# to be the next cluster head

highest\_energy = cf.MINUS\_INFINITY

next\_head = None

for node in cluster:

if node.energy\_source.energy > highest\_energy:

highest\_energy = node.energy\_source.energy

next\_head = node

for node in cluster:

node.next\_hop = next\_head.id

next\_head.next\_hop = cf.BSID

# code temporary ommited

#def FCM\_MTE\_round(network, round\_nb, local\_traces=None, ret=None):

# """Every node communicate its position to the base station. Then the

# BS uses FCM to define clusters and broadcast this information to the

# network. Finally, a round is executed.

# """

# setup\_phase\_fcm(network, round\_nb)

# heads = Network(init\_network=network.get\_heads()+[network.get\_BS()])

# setup\_phase\_mte(heads)

# network.broadcast\_next\_hop()

# network.run\_round(round\_nb)

#def FCM\_PSO\_round(network, round\_nb, local\_traces=None, sleep\_schedulers=None):

# """Every node communicate its position to the base station. Then the

# BS uses FCM to define clusters and broadcast this information to the

# network. Finally, a round is executed.

# """

# setup\_phase\_fcm(network, round\_nb)

# if round\_nb == 0: # clusters do not change in FCM

# clusters = network.split\_in\_clusters()

# sleep\_schedulers = [SleepScheduler(cluster) for cluster in clusters]

#

# for sleep\_scheduler in sleep\_schedulers:

# sleep\_scheduler.schedule()

# network.run\_round(round\_nb)

#

# return sleep\_schedulers

### -- file: + ./python/routing/priodict.py

# Priority dictionary using binary heaps

# David Eppstein, UC Irvine, 8 Mar 2002

from \_\_future\_\_ import generators

class priorityDictionary(dict):

def \_\_init\_\_(self):

'''Initialize priorityDictionary by creating binary heap

of pairs (value,key). Note that changing or removing a dict entry will

not remove the old pair from the heap until it is found by smallest() or

until the heap is rebuilt.'''

self.\_\_heap = []

dict.\_\_init\_\_(self)

def smallest(self):

'''Find smallest item after removing deleted items from heap.'''

if len(self) == 0:

raise IndexError, "smallest of empty priorityDictionary"

heap = self.\_\_heap

while heap[0][1] not in self or self[heap[0][1]] != heap[0][0]:

lastItem = heap.pop()

insertionPoint = 0

while 1:

smallChild = 2\*insertionPoint+1

if smallChild+1 < len(heap) and \

heap[smallChild] > heap[smallChild+1]:

smallChild += 1

if smallChild >= len(heap) or lastItem <= heap[smallChild]:

heap[insertionPoint] = lastItem

break

heap[insertionPoint] = heap[smallChild]

insertionPoint = smallChild

return heap[0][1]

def \_\_iter\_\_(self):

'''Create destructive sorted iterator of priorityDictionary.'''

def iterfn():

while len(self) > 0:

x = self.smallest()

yield x

del self[x]

return iterfn()

def \_\_setitem\_\_(self,key,val):

'''Change value stored in dictionary and add corresponding

pair to heap. Rebuilds the heap if the number of deleted items grows

too large, to avoid memory leakage.'''

dict.\_\_setitem\_\_(self,key,val)

heap = self.\_\_heap

if len(heap) > 2 \* len(self):

self.\_\_heap = [(v,k) for k,v in self.iteritems()]

self.\_\_heap.sort() # builtin sort likely faster than O(n) heapify

else:

newPair = (val,key)

insertionPoint = len(heap)

heap.append(None)

while insertionPoint > 0 and \

newPair < heap[(insertionPoint-1)//2]:

heap[insertionPoint] = heap[(insertionPoint-1)//2]

insertionPoint = (insertionPoint-1)//2

heap[insertionPoint] = newPair

def setdefault(self,key,val):

'''Reimplement setdefault to call our customized \_\_setitem\_\_.'''

if key not in self:

self[key] = val

return self[key]

### -- file: + ./python/routing/\_\_init\_\_.py

Empty file.

### -- file: + ./python/routing/leach.py

import logging, sys

import config as cf

from python.network.network import \*

from python.routing.routing\_protocol import \*

class LEACH(RoutingProtocol):

def setup\_phase(self, network, round\_nb=None):

"""The base station decides which nodes are cluster heads in this

round, depending on a probability. Then it broadcasts this information

to all network.

Reference:

W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, Energy-

efficient communication protocols for wireless sensor networks, In

Proceedings of the 33rd Annual Hawaii International Conference on

System Sciences (HICSS), Hawaii, USA, January 2000.

"""

logging.info('LEACH: setup phase.')

# decide which network are cluster heads

prob\_ch = float(cf.NB\_CLUSTERS)/float(cf.NB\_NODES)

heads = []

alive\_nodes = network.get\_alive\_nodes()

logging.info('LEACH: deciding which nodes are cluster heads.')

idx = 0

while len(heads) != cf.NB\_CLUSTERS:

node = alive\_nodes[idx]

u\_random = np.random.uniform(0, 1)

# node will be a cluster head

if u\_random < prob\_ch:

node.next\_hop = cf.BSID

heads.append(node)

idx = idx+1 if idx < len(alive\_nodes)-1 else 0

# ordinary network choose nearest cluster heads

logging.info('LEACH: ordinary nodes choose nearest nearest cluster head')

for node in alive\_nodes:

if node in heads: # node is cluster head

continue

nearest\_head = heads[0]

# find the nearest cluster head

for head in heads[1:]:

if calculate\_distance(node, nearest\_head) > calculate\_distance(node, head):

nearest\_head = head

node.next\_hop = nearest\_head.id

network.broadcast\_next\_hop()

### -- file: + ./python/routing/direct\_communication.py

import logging, sys

import config as cf

from python.routing.routing\_protocol import \*

class DC(RoutingProtocol):

def pre\_communication(self, network):

"""Setup all the point-to-point connections for the direct communica-

tion scenario. In this scenario, the setup is executed only once, and

all nodes send information directly to the base station.

"""

logging.info('Direct Communication: Setup phase')

for node in network:

node.next\_hop = cf.BSID

def broadcast(self, network):

pass

### -- file: + ./python/routing/routing\_protocol.py

import config as cf

"""This class defines the interface that should be used when defining a new

routing protocol.

"""

class RoutingProtocol(object):

def pre\_communication(self, network):

"""This method is called before round 0."""

if cf.NOTIFY\_POSITION:

network.notify\_position()

def setup\_phase(self, network, round\_nb=None):

"""This method is called before every round. It only redirects to

protected methods."""

if round\_nb == 0:

self.\_initial\_setup(network)

else:

self.\_setup\_phase(network)

def \_initial\_setup(self, network):

"""By default, this codes only calls \_setup\_phase."""

self.\_setup\_phase(network)

def \_setup\_phase(self, network):

"""Should set next hop and cluster heads for all clusters."""

pass

def broadcast(self, network):

network.broadcast\_next\_hop()

### -- file: + ./python/routing/dijkstra.py

# Dijkstra's algorithm for shortest paths

# David Eppstein, UC Irvine, 4 April 2002

# http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/117228

from python.routing.priodict import priorityDictionary

def Dijkstra(G,start,end=None):

"""

Find shortest paths from the start vertex to all vertices nearer than or equal to the end.

The input graph G is assumed to have the following representation:

A vertex can be any object that can be used as an index into a dictionary.

G is a dictionary, indexed by vertices. For any vertex v, G[v] is itself a dictionary,

indexed by the neighbors of v. For any edge v->w, G[v][w] is the length of the edge.

This is related to the representation in <http://www.python.org/doc/essays/graphs.html>

where Guido van Rossum suggests representing graphs as dictionaries mapping vertices

to lists of outgoing edges, however dictionaries of edges have many advantages over lists:

they can store extra information (here, the lengths), they support fast existence tests,

and they allow easy modification of the graph structure by edge insertion and removal.

Such modifications are not needed here but are important in many other graph algorithms.

Since dictionaries obey iterator protocol, a graph represented as described here could

be handed without modification to an algorithm expecting Guido's graph representation.

Of course, G and G[v] need not be actual Python dict objects, they can be any other

type of object that obeys dict protocol, for instance one could use a wrapper in which vertices

are URLs of web pages and a call to G[v] loads the web page and finds its outgoing links.

The output is a pair (D,P) where D[v] is the distance from start to v and P[v] is the

predecessor of v along the shortest path from s to v.

Dijkstra's algorithm is only guaranteed to work correctly when all edge lengths are positive.

This code does not verify this property for all edges (only the edges examined until the end

vertex is reached), but will correctly compute shortest paths even for some graphs with negative

edges, and will raise an exception if it discovers that a negative edge has caused it to make a mistake.

"""

D = {} # dictionary of final distances

P = {} # dictionary of predecessors

Q = priorityDictionary() # estimated distances of non-final vertices

Q[start] = 0

for v in Q:

D[v] = Q[v]

if v == end: break

for w in G[v]:

vwLength = D[v] + G[v][w]

if w in D:

if vwLength < D[w]:

raise ValueError, "Dijkstra: found better path to already-final vertex"

elif w not in Q or vwLength < Q[w]:

Q[w] = vwLength

P[w] = v

return (D,P)

def shortestPath(G,start,end):

"""

Find a single shortest path from the given start vertex to the given end vertex.

The input has the same conventions as Dijkstra().

The output is a list of the vertices in order along the shortest path.

"""

D,P = Dijkstra(G,start,end)

Path = []

while 1:

Path.append(end)

if end == start: break

end = P[end]

Path.reverse()

return Path

### from directory ./cc:

### -- file: + ./cc/optimizer.h

// The version of regions in c++ is used to improve performance. This

// code is supposed to be called by the Python script.

#ifndef OPTIMIZER\_H

#define OPTIMIZER\_H

#include <map>

#include <string>

#include <utility>

#include <vector>

#include <random>

#include "types.h"

#include "regions.h"

#include "individual.h"

// most of these definitions are used to improve readability

typedef unsigned int u\_int;

typedef std::vector<float> float\_v;

typedef std::vector<u\_int> u\_int\_v;

typedef std::pair<std::vector<u\_int>,

float> region\_t;

typedef std::vector<region\_t> regions\_t;

typedef std::map<u\_int, float> dict\_t;

// using char for the smallest addressable unit

typedef std::vector<char> individual\_t;

typedef std::pair<std::map<std::string, u\_int>,

std::map<std::string, float>> config\_t;

class Optimizer {

friend class Individual;

public:

Optimizer(dict\_t exclusive, regions\_t overlapping,

std::vector<u\_int> ids, config\_t config);

virtual ~Optimizer();

// returns a std::vector with the best configuration found (best particle),

// indicating, for each node, if it should sleep or not;

// the learning trace (trace of the best fitness value at each iteration);

// and a std::vector with the coverage and overlapping areas for the best

// configuration

individual\_t Run(float\_v energies);

// setters & getters

void SetAlpha(float value);

void SetBeta(float value);

void SetGamma(float value);

std::vector<float> GetLearningTrace();

std::vector<float> GetTerm1Trace();

std::vector<float> GetTerm2Trace();

float GetBestCoverage();

float GetBestOverlapping();

protected:

Regions \*regions\_;

// attributes

std::vector<u\_int> ids\_;

u\_int nb\_nodes\_;

u\_int nb\_individuals\_;

u\_int max\_iterations\_;

float wmax\_;

float wmin\_;

float fitness\_alpha\_;

float fitness\_beta\_;

float fitness\_gamma\_;

// session attributes (stored here for convenience)

// std::vector with all individuals

std::vector<Individual> population\_;

Individual best\_global\_;

std::vector<Individual> best\_locals\_;

// random related

std::default\_random\_engine generator\_;

float\_v energies\_;

float total\_energy\_;

std::vector<unsigned int> dead\_nodes\_;

unsigned int nb\_alive\_nodes\_;

// learning traces for the last run

std::vector<float> learning\_trace\_;

std::vector<float> term1\_trace\_;

std::vector<float> term2\_trace\_;

// methods

void PrintIndividual(individual\_t individual);

void CreatePopulation();

virtual void Optimize(const std::vector<u\_int> &can\_sleep);

void ClearLearningTraces();

void InitializeSessionData(const float\_v &energies);

void PushIntoLearningTraces(const fitness\_t &fitness);

// Returns a float indicating how fit a individual/particle is,

// and the coverage and overlapping areas for that particle.

virtual fitness\_t Fitness(Individual &individual) = 0;

};

#endif //OPTIMIZER\_H

### -- file: + ./cc/modified\_pso.h

// The version of regions in c++ is used to improve performance. This

// code is supposed to be called by the Python script.

#ifndef MODIFIED\_PSO\_H

#define MODIFIED\_PSO\_H

#include <map>

#include <utility>

#include <vector>

#include <random>

#include "optimizer.h"

#include "individual.h"

class ModifiedPso: public Optimizer {

public:

ModifiedPso(dict\_t exclusive, regions\_t overlapping,

std::vector<u\_int> ids, config\_t config);

~ModifiedPso();

private:

// Mutate and Crossover functions modify the first argument since it is a

// reference. This is not the best practice but it is done for performance

// reasons.

void Mutate(Individual &individual, std::vector<u\_int> can\_sleep,

float mutation\_rate);

// Returns an individual that gets statistically half of its genes

// from individual1 and half from individual2

void Crossover(Individual &individual1, Individual &individual2);

void Optimize(const std::vector<u\_int> &can\_sleep);

fitness\_t Fitness(Individual &individual);

};

#endif //MODIFIED\_PSO\_H

### -- file: + ./cc/regions.h

// The version of regions in c++ is used to improve performance. This

// code is supposed to be called by the Python script.

#ifndef REGIONS\_H

#define REGIONS\_H

#include <map>

#include <utility>

#include <vector>

#include "types.h"

typedef unsigned int u\_int;

typedef std::pair<std::vector<u\_int>,

float> region\_t;

class Regions {

public:

Regions(std::map<u\_int, float> \_exclusive,

std::vector<region\_t> \_overlapping);

~Regions();

coverage\_info\_t GetCoverage(const std::vector<char> &individual,

const std::vector<float> &energies);

void InitSession(const std::vector<float> &energies);

private:

std::map<u\_int, float> exclusive\_;

std::vector<region\_t> overlapping\_;

float total\_coverage\_exclusive\_ = 0.0;

// session attributes

float total\_coverage\_ = 0.0;

float total\_overlapping\_ = 0.0;

};

#endif //REGIONS\_H

### -- file: + ./cc/pso.h

// The version of regions in c++ is used to improve performance. This

// code is supposed to be called by the Python script.

#ifndef PSO\_H

#define PSO\_H

#include <map>

#include <utility>

#include <vector>

#include <random>

#include "optimizer.h"

#include "individual.h"

class Pso: public Optimizer {

public:

Pso(dict\_t exclusive, regions\_t overlapping,

std::vector<u\_int> ids, config\_t config);

~Pso();

private:

// attributes

std::vector<float\_v> velocity\_;

void Optimize(const std::vector<u\_int> &can\_sleep);

fitness\_t Fitness(Individual &individual);

};

#endif //PSO\_H

### -- file: + ./cc/ecca.h

#ifndef ECCA\_H

#define ECCA\_H

#include <map>

#include <string>

#include <utility>

#include <vector>

#include "optimizer.h"

#include "individual.h"

class Ecca: public Optimizer {

public:

Ecca(dict\_t exclusive, regions\_t overlapping,

std::vector<unsigned int> ids, config\_t config);

virtual ~Ecca();

// returns a std::vector with the best configuration found (best particle),

// indicating, for each node, if it should sleep or not;

// the learning trace (trace of the best fitness value at each iteration);

// and a std::vector with the coverage and overlapping areas for the best

// configuration

individual\_t Run(std::vector<float> energies);

private:

//std::vector<Individual> CreatePopulation1();

std::vector<std::vector<Individual>>

FastNonDominatedSort(std::vector<Individual> &population);

bool Dominates(Individual &individual1,

Individual &individual2);

std::vector<Individual>

Reproduce(std::vector<Individual> &population, float crossover\_rate);

std::vector<Individual>

FindBestParents(std::vector<std::vector<Individual>> &fronts);

void CalculateCrowdingDistance(std::vector<Individual> &group);

void CrowdedSorting(std::vector<Individual> &group);

fitness\_t Fitness(Individual &individual);

};

#endif //ECCA\_H

### -- file: + ./cc/custom\_types.h

#ifndef CUSTOM\_TYPES\_H

#define CUSTOM\_TYPES\_H

// most of these definitions are used to improve readability

typedef unsigned int u\_int;

typedef std::vector<float> float\_v;

typedef std::vector<u\_int> u\_int\_v;

typedef std::pair<std::vector<u\_int>,

float> region\_t;

typedef std::vector<region\_t> regions\_t;

typedef std::map<u\_int, float> dict\_t;

// using char for the smallest addressable unit

typedef std::vector<char> individual\_t;

typedef std::pair<std::map<std::string, u\_int>,

std::map<std::string, float>> config\_t;

typedef struct {

float partial\_coverage;

float total\_coverage;

float partial\_overlapping;

float total\_overlapping;

float exclusive\_area;

} coverage\_info\_t;

typedef struct {

float fitness\_value;

coverage\_info\_t coverage\_info;

} fitness\_ret\_t;

#endif // CUSTOM\_TYPES\_H

### -- file: + ./cc/individual.h

#ifndef INDIVIDUAL\_H

#define INDIVIDUAL\_H

#include <vector>

#include <random>

#include "types.h"

class Optimizer;

class Individual {

public:

Individual();

Individual(unsigned int idx, Optimizer \*container\_handler);

// copy constructor

//Individual(const Individual &individual);

Individual(unsigned int idx, Individual &father,

Individual &mother, float crossover\_rate,

Optimizer \*container\_handler);

~Individual();

fitness\_t GetFitness() const;

void SetFitness(fitness\_t value);

std::vector<char> GetGenes();

void SetGenes(std::vector<char> value);

//static void SetNewRun();

//static std::vector<char> GetBestGenes();

//static fitness\_t GetBestFitness();

// used by NSGA-II to:

// indicate front membership

unsigned int rank\_;

unsigned int idx\_;

// value of the crowding distance

float crowd\_dist\_;

fitness\_t fitness\_;

// best fitness and genes in family lineage

//fitness\_t best\_fitness\_;

//std::vector<char> best\_genes;

// best fitness and genes in population's history

//static fitness\_t best\_global\_fitness\_;

//static std::vector<char> best\_global\_genes\_;

private:

std::vector<char> genes\_;

// All individuals share a handler to optimizer

static Optimizer \*optimizer\_;

static std::default\_random\_engine generator\_;

//fitness\_t UpdateFitness();

void SampleNewGenes();

};

#endif //INDIVIDUAL\_H

### -- file: + ./cc/my\_pso.h

// The version of regions in c++ is used to improve performance. This

// code is supposed to be called by the Python script.

#ifndef MY\_PSO\_H

#define MY\_PSO\_H

#include <map>

#include <utility>

#include <vector>

#include <random>

#include "optimizer.h"

using namespace std;

class MyPso: public Optimizer {

public:

MyPso(dict\_t exclusive, regions\_t overlapping,

vector<u\_int> ids, config\_t config);

~MyPso();

private:

// attributes

vector<float\_v> velocity\_;

// change individual's position randomly (random walk). The number of

// altered genes is proportional to acceleration

void Move(individual\_t &individual, vector<u\_int> can\_sleep,

float acceleration);

// individual1 copy parts of individual2 position depending on influence

// rate and how far it is from individual2 (the farer the more copies)

void Influence(const individual\_t &original\_individual,

const individual\_t &influencer,

individual\_t &new\_individual,

vector<u\_int> can\_sleep,

float influence\_rate);

void Optimize(float\_v energies, const vector<u\_int> &can\_sleep,

float total\_energy);

};

#endif //MY\_PSO\_H

### -- file: + ./cc/types.h

#ifndef TYPES\_H

#define TYPES\_H

typedef struct {

float partial\_coverage;

float total\_coverage;

float partial\_overlapping;

float total\_overlapping;

float exclusive\_area;

} coverage\_info\_t;

typedef struct {

float total;

float term1;

float term2;

coverage\_info\_t coverage\_info;

} fitness\_t;

typedef std::vector<fitness\_t> population\_fitness\_t;

#endif // TYPES\_H

### -- file: + ./cc/genetic\_algorithm.h

// The version of regions in c++ is used to improve performance. This

// code is supposed to be called by the Python script.

#ifndef GENETIC\_ALGORITHM\_H

#define GENETIC\_ALGORITHM\_H

#include <map>

#include <utility>

#include <vector>

#include <random>

#include "optimizer.h"

#include "individual.h"

using namespace std;

class GeneticAlgorithm: public Optimizer {

public:

GeneticAlgorithm(dict\_t exclusive, regions\_t overlapping,

vector<u\_int> ids, config\_t config);

~GeneticAlgorithm();

private:

// change individual's position randomly (random walk). The number of

// altered genes is proportional to mutation\_rate

void Mutate(Individual &individual, vector<u\_int> can\_sleep,

float mutation\_rate);

// individual1 copy parts of individual2 position depending on influence

// rate and how far it is from individual2 (the farer the more copies)

Individual Crossover(u\_int nb\_unfit,

const vector<u\_int> &can\_sleep);

void Optimize(const vector<u\_int> &can\_sleep);

// sort population according to fitness value

void SortFitness();

fitness\_t Fitness(Individual &individual);

};

#endif //GENETIC\_ALGORITHM\_H