Start the program

Go into the root directory of the package, and use python -m wsnsims.conductor.driver

New Files added:

The new files are present in *wsnsims/loaf*.

- *loaf_sim.py* contains all functions for the first and second phase, as well as the main function called by the simulation driver.
- loaf_runner.py contains all function for the metrics calculations (energy, delay, buffer size)
- movement.py is copied from the MINDS algorithm.
- energy.py is copied from the MINDS algorithm, with one modification:
    - During the second phase, the rendezvous points are added as segments, but should not be used for calculating the data volume, as they are only points for the tour path of the MDC, no real nodes.
    - So they are removed from the cluster data volume computation, see `cluster_data_volume()`

The file *wsnsims/conductor/driver*, which is launched at start, has also been modified to include the calculation of LOAF. A function `run_loaf()` has been written.

First Phase:

The first phase follows the algorithm in the paper:

- the virtual center is computed using the function `compute_center()`. Then, the closest real segment is computed using `get_closest()`. Set then the attribute `center` of the simulation as this segment.
- A for loop is then used on each segment to add them to a list of segments in a range of size R around the center.
- Using this list, a cluster is created (the central cluster, `original_cluster`) with the center eG as relay node.
- All segments not present in the central cluster are added to `remaining_segments`
- For each segment in `remaining_segments`, a cluster is created, with the only segment being the segment and the center. The clusters are added to `noncentral_clusters`.
- N(0)_cluster is named `number_remaining_segments` and set as the number of segments in `remaining_segments`, and N(0)cluster is named `index_n_cluster` and set as the same value
- The value of k is then taken from the MDC count in the environment variables: `k = self.env.mdc_count`.
- Start the do-while:
    - get the index of C_x and C_y that minimize the energy using `_minimize_energy()`:

        - Loop on all C_x and C_y, x different from y.
        - Calculate the energy sum:
            - Loop on all C_i, to get the sum of energy for all C_i except C_x and C_y
            - To get the energy, use the models previously made, see “loaf/energy” and “loaf/movement”.
            - To do so, a LOAFRunner has to be created. Following the implementation of the other algorithms, the runners are used at the end to calculate the metrics (energy, buffer, …). We use it here to get the energy consumption.
            - Because we merge C_x and C_y for the computation, a temporary LOAF object has to be created, by copying the current object. The merged cluster of C_x and C_y is added to the copy, and C_x and C_y are removed.
            - The model for the energy computation is then applied on the copy.
        - If the energy is less than the previous minimum, store the current minimum `energy_minimum`, x and y as `index_chosen_cx` and `index_chosen_cy`.
        - Return the x and y with minimum energy
    - Retrieve the C_x and C_y from `noncentral_clusters` as `cluster_x` and `cluster_y`. Then get the union of C_y in C_x, and put the result at the C_x place in `noncentral_clusters`.
    - Remove C_x from `noncentral_clusters`.
    - Increase r, reduce N_cluster.
    - Copy the current list of clusters `noncentral_clusters`, and add it to the current simulation attribute `clusters`.
    - Continue the loop as long as r < N_cluster – (k-1), which means: `round_num < (number_remaining_segments - (k – 1))`.
- Copy the `noncentral_clusters` in the clusters of the simulation.
- Add the central cluster `original_cluster`.
- For each cluster, change its ID from 0 to the number of clusters minus one (for display purpose)

Second Phase:

The central cluster is the last one in the list of clusters: `self.clusters[-1]`.

- The central cluster is set as `c_k`. The variable epsilon is set as the communication range. The value is read from the environment.
- The tour paths (TR(r)_i), the rendezvous points (P(r)_i), the energy of the MDCs (E(r)_i) and the centers of mass (CoM(r)_i) are store respectively in `tour_paths`, `rendezvous_points`, `mdc_energies` and `centers_of_mass`. These are dictionary, that have the round number `round_num` as key, and all the values for this round in a list as value. To be able to do so in Python, the list are initialized as list of elements `None` with the size being the number of clusters.
- `list_values` will be used later on to initialize the list for further rounds. At first, only the round 0 is initialized. The later one are done lazily.
- A runner is created to get the energy.
- For each clusters except the central one:
    - Calculate the tour path of the existing segments
    - Calculate the energy consumption
    - Calculate the center of mass
    - Use this center of mass to get the rendezvous point between the center and the center of mass. The rendezvous point is at a distance R from the center.
- Calculate the center of mass and the energy for the central cluster.
- Calculate the average energy using `np.average()` on the list of energies.

- Compute the standard deviation SD(0), store it in the list `sds` as first element (index 0 => round 0).
- Enter the do-while SD(rr) < SD(rr -1)
    - initialize the lists in the dictionary of the `list_values` for the current round.
    - Get the least energy to `e_least`.
    - Enter the do-while E(r) < E(r)_AVG
        - For each cluster except the central cluster, the new rendezvous point is computed:
            - if the cluster has an energy less than average, the new rendezvous point is the old one.
            - If the cluster has a higher energy than average:
                - The shifing `shifting_ratio` ratio (epsilon * E(r-1)_(ir−1) / E(r-1)_least) is computed easily.
                - The previous distance between the center of mass of the central cluster and the rendezvous point is stored in `dist`, using the `distance()` function from core.point.Vec2.
                - Using the `_calc_rdv_point()`, the new position of the rendezvous point is computed. It creates a new Segment with the location on the line between the central cluster's center of mass and the center of mass of the current cluster. The new location is at a distance `dist + shifting_ratio` from the central cluster's center of mass. The function uses trigonometry to calculate the new x and y coordinates:
                    - the angle of the line between the center of mass of the central cluster and the rendezvous point is computed and assigned to `angle`.
                    - Because `acos()` returns the same value for pi/2 and -pi/2, `asin` is also computed. If the angle returned is less than 0, the cosinus should be negative, thus is multiplied by -1.
                    - Then, using sinus and cosinus, the new coordinates are calculated
                    - A Segment instance is returned, with a new attribute: `fake_segment`, which is used to differentiate real Segments from simple rendezvous point. The value of this attribute is the cluster ID it belongs to.
                - The new rendezvous point is stored in the rendezvous point list.
                - Then remove the old occurrences of eG in the non-central clusters.
                - Using the function `_replace_rdv_point`, the older rendezvous point is also removed from the clusters:
                    - The ID of the cluster it blongs to is retrieved from the attribute `fake_segment`.
                    - Retrieve from the cluster the reference to the older rendezvous point, by matching the Segment with the same `fake_segment` attribute.
                    - If found, remove it from the cluster, the central cluster and the list of segments in the simulation.
                    - Add the new rendezvous point to the cluster, the central cluster and the list of segments in the simulation.
                    - If the rendezvous point was not in the list of segments already, add it.
        - Update the tour path in `tour_paths`.
        - For each Segments of all clusters (except the central one):
            - create the polygon `ck_path` formed by TR(r)_k as an `mp.Path`.
            - If the Segment location is in this polygon (checked using `contains_point`):
                - remove the segment from the cluster
                - add it to the central cluster.
        - Update the centers of mass and the tour paths
        - Calculate the total energy using a new runner.
        - For all clusters:

      - if no rendezvous point has been calculated for this round:
      - get the one from the previous round.
    - To emulate a do-while, have a condition at the end:
      - if the energy of the central cluster is less that the average energy, continue
      - otherwise, break out of the do-while.
  - Add the `number_tries` attribute, to prevent looping too many times on the next loop, set to 10 tries
  - Enter the do-while E(r) > E(r)_AVG
    - initialize the lists in the dictionary of the `list_values` for the current round.
    - Store the center of mass of the central cluster for the round `round_num` as the one from round `f`.
    - For each cluster except the central cluster, the new rendezvous point is computed:
      - if the cluster has an energy less than average, the new rendezvous point is the old one.
      - If the cluster has a lower energy than average:
        - The shifing `shifting_ratio` ratio (epsilon * E(r-1)_least / E(r-1)_(ir−1)) is computed easily.
        - The previous distance between the center of mass of the central cluster and the rendezvous point is stored in `dist`, using the `distance()` function from core.point.Vec2.
        - Using the `_calc_rdv_point()`, the new position of the rendezvous point is computed, this time with `dist - shifting_ratio`.
        - The new rendezvous point is stored in the rendezvous point list.
        - Then remove the old occurrences of eG in the non-central clusters.
        - Using the function `_replace_rdv_point`, the older rendezvous point is also removed from the clusters:
        - Update the tour path in `tour_paths` for the current cluster.
        - Update also the center of mass and energy
    - Update the center of mass of the central cluster after the new rendezvous point have been computed
    - For all clusters:
      - if no rendezvous point has been calculated for this round:
      - get the one from the previous round.
    - Because the elements returned by the second phase include C(r-1)_i, `cluster_r_minus_one` is created to store the list of clusters. A deep copy is performed to prevent modifications.
    - The new energies are calculated using a runner.
    - `number_tries` is decreased by one.
    - If `number_tries` is 0 or less, break out of the do-while
    - Otherwise, if the average energy is higher than the central cluster's energy, break out of the do-while
  - `rr` is increased, and the new standard deviation of the energy is computed using numpy's `std()`, and stored in `sds`.
  - If the new standard deviation `sds[rr]` is higher than the previous one `sds[rr - 1]`, break out of the loop.
- Return `cluster_r_minus_one` and the previous tour paths.

Results:

The final results are present in the *results* directory as CSV files, with the headers:

- 'max_delay', 'balance', 'lifetime', 'ave_energy', 'max_buffer'

These metrics are the same as the one from the LEEF algorithm paper: (“LEEF: Latency and energy efficient federation of disjoint wireless sensor segments”) are have been computed using the same functions and equations.