ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΠΙΔΟΣΗ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

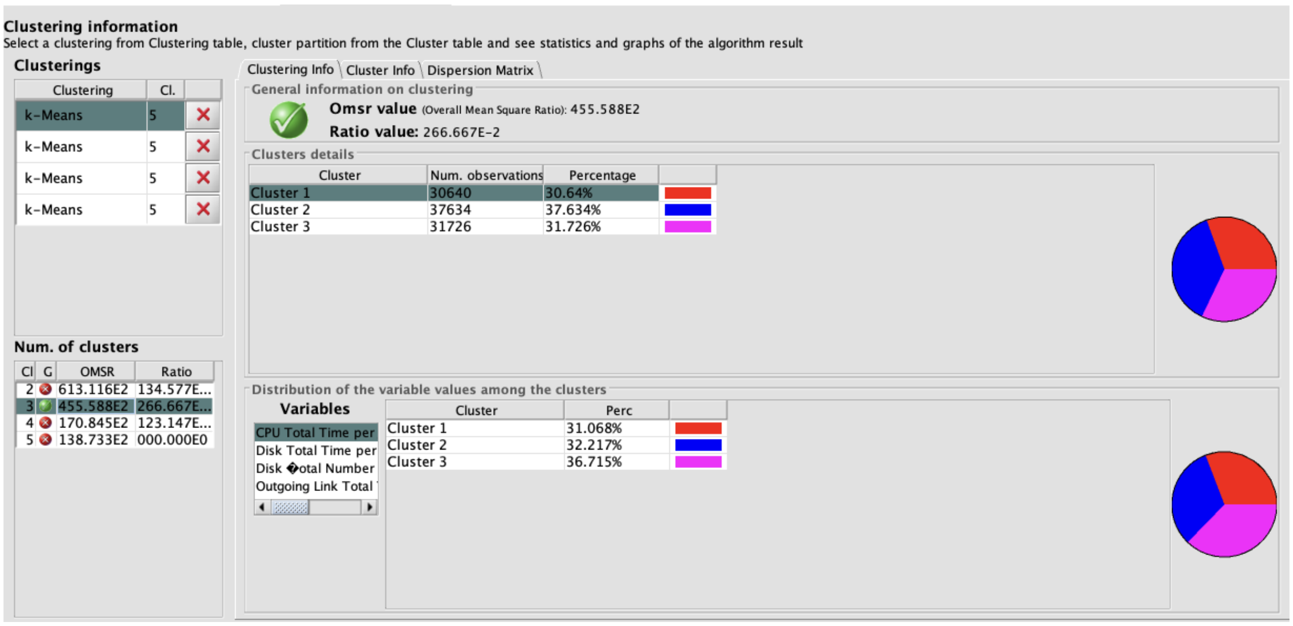
Ιούνιος 2024

Ιωάννης Γιαννούκος 031 18918
Αργυρώ Τσίπη 031 19950

Εργασία 2

Α. Χαρακτηρισμός φορτίου

Έχοντας πραγματοποιήσει 5 φορές την διαδικασία, επιλέγουμε την ομαδοποίηση των τριών clusters που έχει το μεγαλύτερο ratio.



Clustering information
Select a clustering from Clustering table, cluster partition from the Cluster table and see statistics and graphs of the algorithm result

Clusterings

Clustering	Cl.	
k-Means	5	✗
k-Means	5	✗
k-Means	5	✗
k-Means	5	✗

Num. of clusters

Cl	G	OMSR	Ratio
2	✗	613.116E2	134.577E...
3	●	455.588E2	266.667E...
4	✗	170.845E2	123.147E...
5	✗	138.733E2	000.000E0

Clustering Info | Cluster Info | Dispersion Matrix

General information on clustering
Omsr value (Overall Mean Square Ratio): 455.588E2
Ratio value: 266.667E-2

Clusters details

Cluster	Num. observations	Percentage	
Cluster 1	30640	30.64%	🟥
Cluster 2	37634	37.634%	🟦
Cluster 3	31726	31.726%	🟪

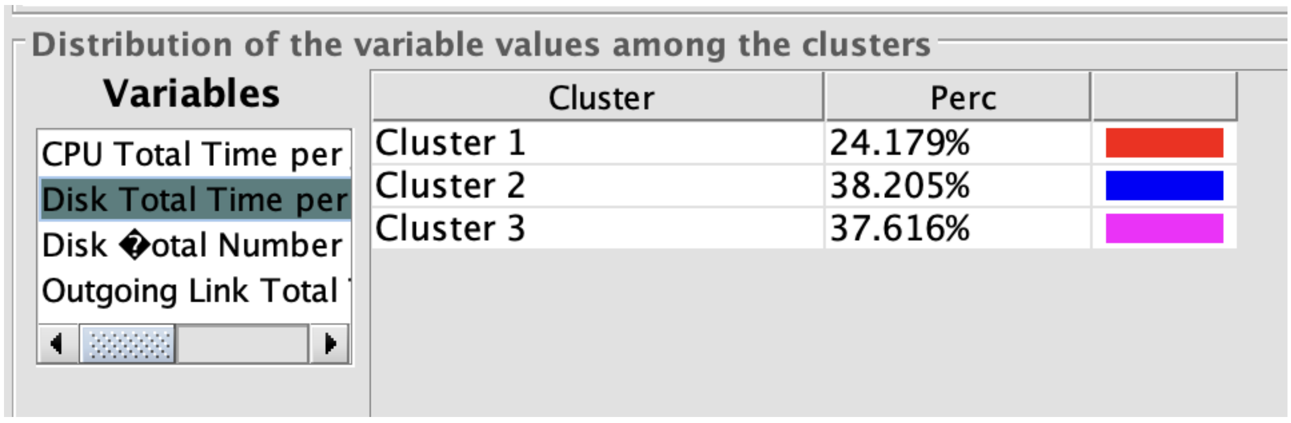
Distribution of the variable values among the clusters

Variables
CPU Total Time per
Disk Total Time per
Disk �otal Number
Outgoing Link Total

Cluster	Perc	
Cluster 1	31.068%	🟥
Cluster 2	32.217%	🟦
Cluster 3	36.715%	🟪

Στην περίπτωση αυτή, το ratio είναι ίσο με 266.66E.

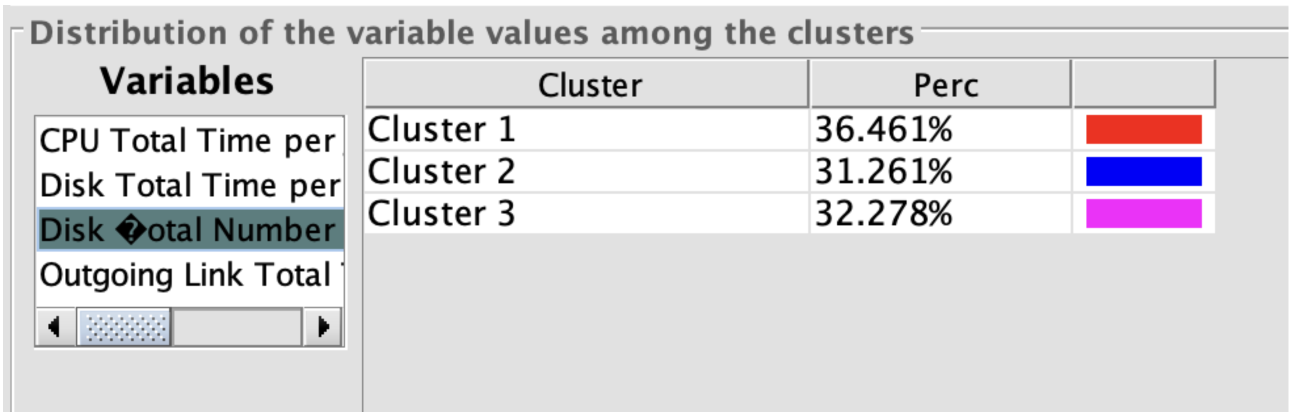
Έχουμε, λοιπόν, τις ακόλουθες τιμές:



Distribution of the variable values among the clusters

Variables
CPU Total Time per
Disk Total Time per
Disk �otal Number
Outgoing Link Total

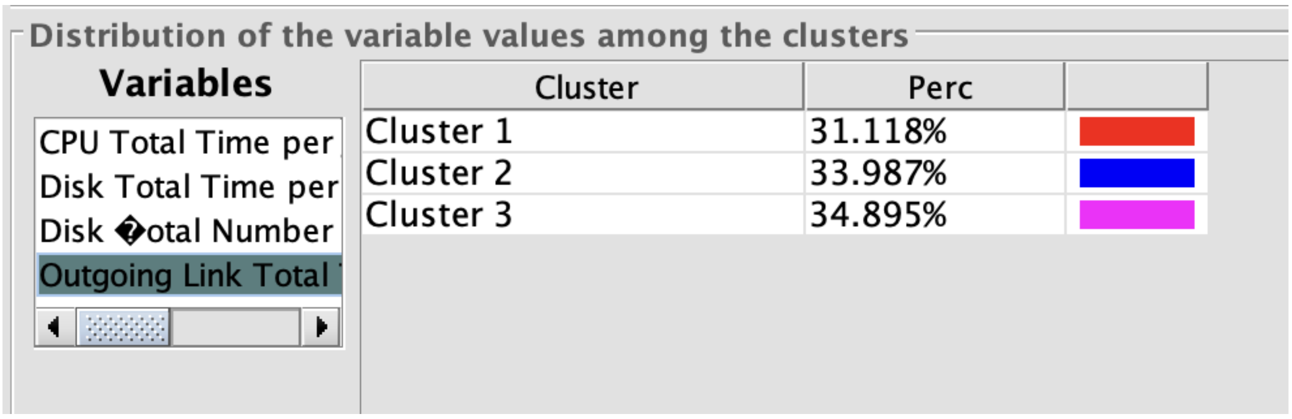
Cluster	Perc	
Cluster 1	24.179%	🟥
Cluster 2	38.205%	🟦
Cluster 3	37.616%	🟪



Distribution of the variable values among the clusters

Variables
CPU Total Time per
Disk Total Time per
Disk �otal Number
Outgoing Link Total

Cluster	Perc	
Cluster 1	36.461%	🟥
Cluster 2	31.261%	🟦
Cluster 3	32.278%	🟪

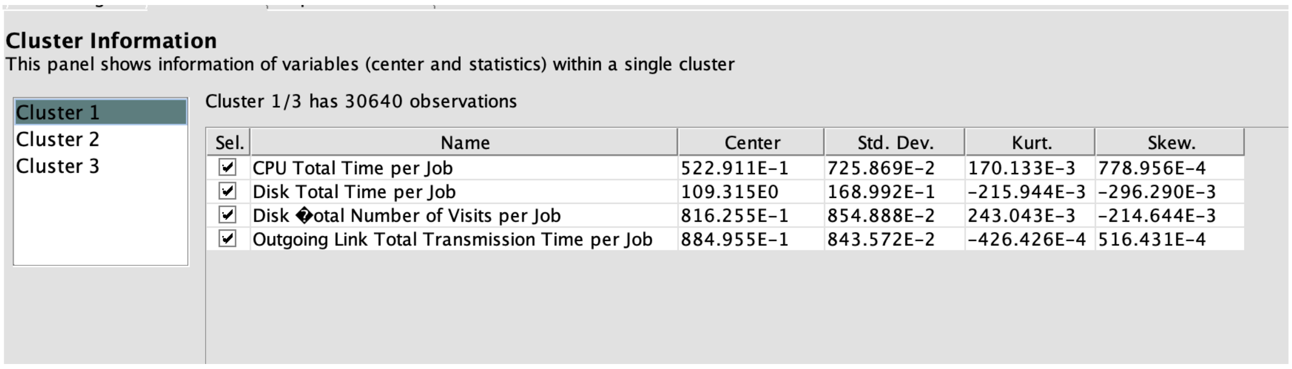


Distribution of the variable values among the clusters

Variables

- CPU Total Time per
- Disk Total Time per
- Disk �otal Number
- Outgoing Link Total

Cluster	Perc	
Cluster 1	31.118%	🟥
Cluster 2	33.987%	🟦
Cluster 3	34.895%	🟪

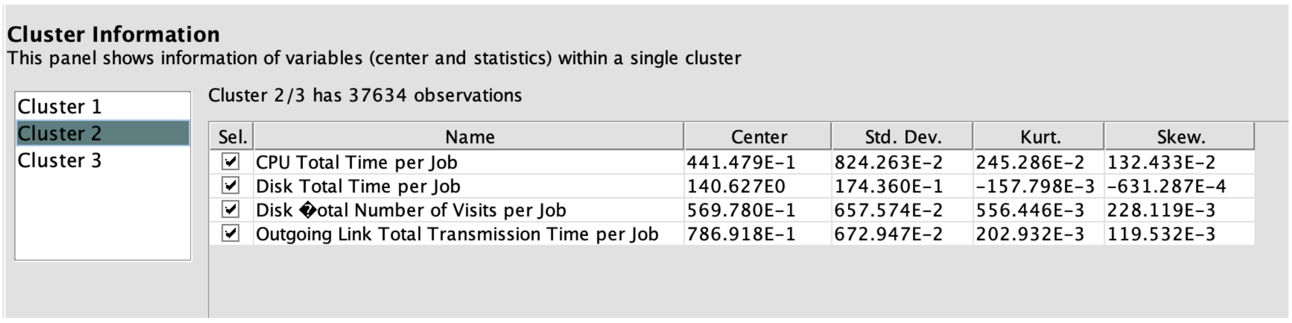


Cluster Information
This panel shows information of variables (center and statistics) within a single cluster

- Cluster 1
- Cluster 2
- Cluster 3

Cluster 1/3 has 30640 observations

Sel.	Name	Center	Std. Dev.	Kurt.	Skew.
✓	CPU Total Time per Job	522.911E-1	725.869E-2	170.133E-3	778.956E-4
✓	Disk Total Time per Job	109.315E0	168.992E-1	-215.944E-3	-296.290E-3
✓	Disk �otal Number of Visits per Job	816.255E-1	854.888E-2	243.043E-3	-214.644E-3
✓	Outgoing Link Total Transmission Time per Job	884.955E-1	843.572E-2	-426.426E-4	516.431E-4

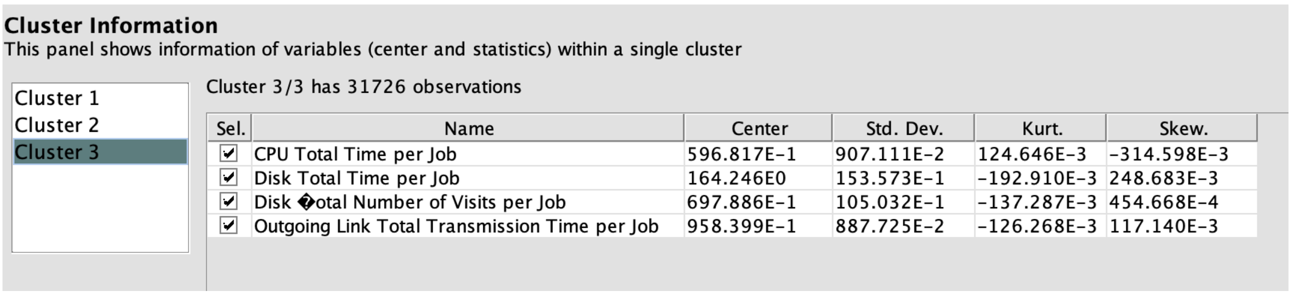


Cluster Information
This panel shows information of variables (center and statistics) within a single cluster

- Cluster 1
- Cluster 2
- Cluster 3

Cluster 2/3 has 37634 observations

Sel.	Name	Center	Std. Dev.	Kurt.	Skew.
✓	CPU Total Time per Job	441.479E-1	824.263E-2	245.286E-2	132.433E-2
✓	Disk Total Time per Job	140.627E0	174.360E-1	-157.798E-3	-631.287E-4
✓	Disk �otal Number of Visits per Job	569.780E-1	657.574E-2	556.446E-3	228.119E-3
✓	Outgoing Link Total Transmission Time per Job	786.918E-1	672.947E-2	202.932E-3	119.532E-3



Cluster Information
This panel shows information of variables (center and statistics) within a single cluster

- Cluster 1
- Cluster 2
- Cluster 3

Cluster 3/3 has 31726 observations

Sel.	Name	Center	Std. Dev.	Kurt.	Skew.
✓	CPU Total Time per Job	596.817E-1	907.111E-2	124.646E-3	-314.598E-3
✓	Disk Total Time per Job	164.246E0	153.573E-1	-192.910E-3	248.683E-3
✓	Disk �otal Number of Visits per Job	697.886E-1	105.032E-1	-137.287E-3	454.668E-4
✓	Outgoing Link Total Transmission Time per Job	958.399E-1	887.725E-2	-126.268E-3	117.140E-3

B. Ανάλυση επίδοσης

Παρακάτω παραθέτουμε τον Python κώδικα της προσομοίωσης.

```python
### SIMULATION MAIN ALGORITHM

import numpy as np
from scipy.stats import erlang
from collections import deque
import matplotlib.pyplot as plt

stations: tuple[str] = ("CPU", "DISK", "OUT")
categories: tuple[str] = ("A", "B", "C")
station_index: dict[str, int] = {}
category_index: dict[str, int] = {}
for i, station in enumerate(stations):
    station_index[station] = i
for j, category in enumerate(categories):
    category_index[category] = j

sec_of_msec = lambda t: t * 0.001

D_CPU_A: float = sec_of_msec(52.2911)
D_DISK_A: float = sec_of_msec(109.315)
v_DISK_A: float = 81.6255
D_OUT_A: float = sec_of_msec(8.84955)

D_CPU_B: float = sec_of_msec(4.41479)
D_DISK_B: float = sec_of_msec(140.627)
v_DISK_B: float = 56.978
D_OUT_B: float = sec_of_msec(78.6918)

D_CPU_C: float = sec_of_msec(59.6817)
D_DISK_C: float = sec_of_msec(164.246)
v_DISK_C: float = 69.7886
D_OUT_C: float = sec_of_msec(95.8399)

v_CPU_A: float = v_DISK_A + 1
v_CPU_B: float = v_DISK_B + 1
v_CPU_C: float = v_DISK_C + 1
v_OUT_A = v_OUT_B = v_OUT_C = 1

arrival_rate: float = 256776 / 45100  # λ: total arrivals per total time
frame
average_arrival_time: float = 1 / arrival_rate

Dij = (
    (D_CPU_A, D_CPU_B, D_CPU_C),
    (D_DISK_A, D_DISK_B, D_DISK_C),
    (D_OUT_A, D_OUT_B, D_OUT_C),
)
vij = (
    (v_CPU_A, v_CPU_B, v_CPU_C),
    (v_DISK_A, v_DISK_B, v_DISK_C),
    (v_OUT_A, v_OUT_B, v_OUT_C),
)
Sij = ([], [], [])
for i in range(len(stations)):
    for j in range(len(categories)):
        Sij[i].append(Dij[i][j] / vij[i][j])

# Result variables
# X_k: list of numbers that correspond to the average number of each
category
# X_k_per_cycle: tuple of list of numbers that correspond to the average
number of a category of each cycle
lamda_j = [None, None, None]
lamda_j_per_cycle = ([None], [None], [None])
R_j = [None, None, None]
R_j_per_cycle = ([None], [None], [None])
U_i = [None, None, None]
U_i_per_cycle = ([None], [None], [None])


def Erlang_4_service_time(category: str) -> float:
    # mean = k / λ
    k = 4
    lamda = k / Sij[station_index["CPU"]][category_index[category]]
    return erlang.rvs(a=k, scale=1 / lamda)


def Exponential_service_time(station: str, category: str) -> float:
    # mean = 1 / λ
    mean = Sij[station_index[station]][category_index[category]]
    return np.random.exponential(scale=mean)


U = lambda: float(np.random.uniform())


def job_category() -> str:
    """
    This function returns a category based on the number of
    arrivals of said category per total arrivals.

    Returns:
        str: A string indicating the category of the job that arrived.
    """
    p_A = 0.30640
    p_B = 0.37634
    # p_C = 0.31726
    category_prob = U()
    if category_prob <= p_A:
        return "A"
    elif category_prob <= p_A + p_B:
        return "B"
    else:
        return "C"


# plot_list has 3 lists, one for each station. Every time an event happens,
the number of jobs
# using the station, either waiting in its queue or not, is appended on the
station's list.
plot_list: tuple[list[int]] = ([], [], [])
```

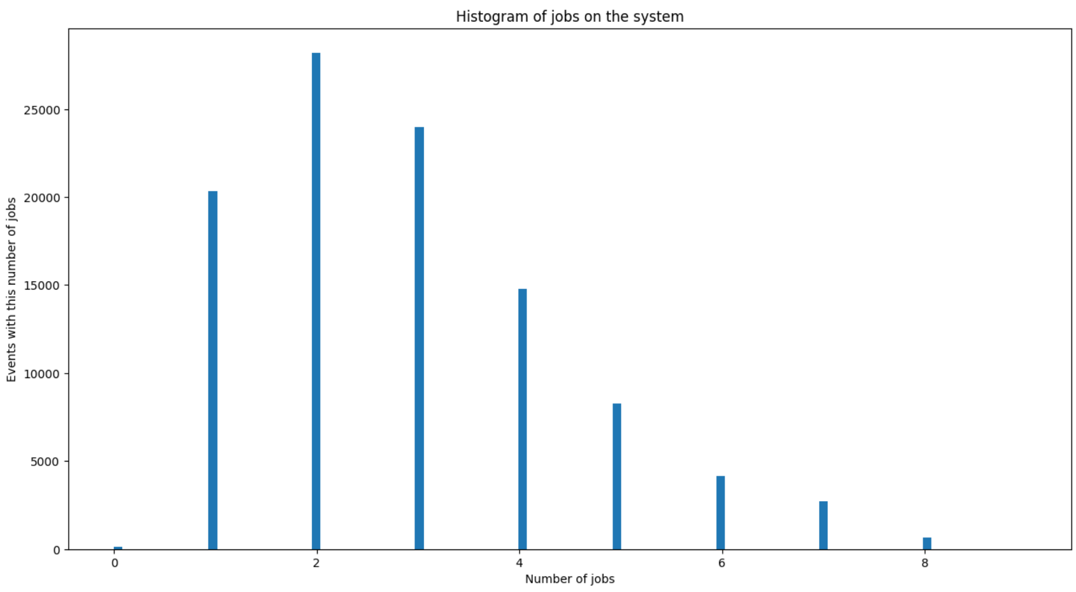
```python
def status(event: str) -> None:
    """
    This function is used to updated the number of jobs in the system
    every time a new event happens.

    Args:
        event (str): Dummy arguments that indicates the current event.
    """
    for i, station in enumerate(stations):
        if station == "CPU":
            plot_list[i].append(len(STATION_queue[i]))
        else:
            plot_list[i].append(0 if empty_STATION[i] else
len(STATION_queue[i]) + 1)
    return


STATION_queue = (deque(), deque(), deque())

STATION_current_job: list[int] = [None, None, None]
STATION_current_category: list[str] = [None, None, None]
STATION_start_time: list[float] = [None, None, None]
STATION_remaining_time: list[float] = [None, None, None]

STATION_EMPTY_TIME: list[float] = [0, 0, 0]  # re-initialized for each
cycle
STATION_LAST_EMPTY: list[float] = [0, 0, 0]
empty_STATION: list[bool] = [True, True, True]

STATION_empty_time_per_cycle = ([None], [None], [None])


def set_current_counters(
    station: str, job: int, category: str, start_time: float,
remaining_time: float
) -> None:
    """
    This function updates the variables of a station that correspond to the
current job,
    current category, start time and remaining time.

    Args:
        station (str): The station of which the corresponding variables
will change.
        job (int): Id of the job that is currently served by the station.
If the station
            if the CPU, which uses a Processor sharing policy, this is the
id of the job
            that requires the shortest time.
        category (str): The category of the job.
        start_time (float): The time that the station will start processing
the job.
        remaining_time (float): The required service time by the job.
    """
    i = station_index[station]
    STATION_current_job[i] = job
    STATION_current_category[i] = category
    STATION_start_time[i] = start_time
    STATION_remaining_time[i] = remaining_time
    return


def next_event(arrival_time: float) -> str:
    """
    This function calculates the event that will happen sooner.

    Args:
        arrival_time (float): The time that the next arrival will happen.

    Returns:
        str: Returns either the station that will finish serving a job or
"arrival" if a
            job will arrive in the system before a station finishes serving
a job.
    """
    valid_sum = lambda x, y: x + y if x != None and y != None else None
    CPU_finish_time = valid_sum(STATION_start_time[0],
STATION_remaining_time[0])
    DISK_finish_time = valid_sum(STATION_start_time[1],
STATION_remaining_time[1])
    OUT_finish_time = valid_sum(STATION_start_time[2],
STATION_remaining_time[2])

    lst = [arrival_time, CPU_finish_time, DISK_finish_time,
OUT_finish_time]

    min_value = float("inf")
    min_position = None

    for index, value in enumerate(lst):
        if value is not None and value < min_value:
            min_value = value
            min_position = index

    return ["arrival", "CPU", "DISK", "OUT"][min_position]


# arrival_time(job_id: int) -> arrival_time_of_job: float
arrival_time_of_job = lambda job_id: job_id * average_arrival_time
# departure_time_of_job: dict[job_id, departure_time_of_job]
departure_time_of_job: dict[int, float] = {}
# disk_visits: dict[job_id, current_number_of_visits]
disk_visits: dict[int, float] = {}

clock: int = 0
job_id: int = 1
curr_jobs: int = 0  # re-initialized for each cycle (automatically)


def decrease_CPU_remaining_time(time_passed: float) -> None:
    """
    This function decreases the remaining time of all the jobs in CPU's
queue by the given
    time.

    Args:
        time_passed (float): The time that the remaining times of the jobs
will be reduced by.
    """
    for index in range(len(STATION_queue[station_index["CPU"]])):
        STATION_queue[station_index["CPU"]][index][2] -= time_passed
    return


def add_job_to_station(station: str, job_id: int, job_category: str) ->
None:
    """
    This function adds a job to a station's queue.

    Args:
        station (str): The station-destination of the job.
        job_id (int): The job id of the job that will be added.
        job_category (str): The category of the job that will be added.
    """
    global clock
    i = station_index[station]
    service_time = float(
        Erlang_4_service_time(job_category)
        if station == "CPU"
        else Exponential_service_time(station, job_category)
    )

    def add_job_to_CPU_queue(
        job_id: int, job_category: str, service_time: float
    ) -> None:
        """
        This function adds a job to the correct position of the CPU's
queue. All the jobs
        in the CPU's queue are sorted in increasing order of remaining
service time.

        Args:
            job_id (int): The job's id that will be inserted into the
queue.
            job_category (str): The job's category that will be inserted
into the queue.
            service_time (flaot): The service time needed by the job.
        """
        insert_position = len(STATION_queue[station_index["CPU"]])
        for index, value in enumerate(STATION_queue[station_index["CPU"]]):
            if value[2] > service_time:
                insert_position = index
                break
        STATION_queue[station_index["CPU"]].insert(
            insert_position, [job_id, job_category, service_time]
        )
        return

    match station:
        case "CPU":
            if STATION_queue[i]:
                decrease_CPU_remaining_time(clock - STATION_start_time[i])
            add_job_to_CPU_queue(job_id, job_category, service_time)
            job, category, remaining_time = STATION_queue[i][0]
            set_current_counters(station, job, category, clock,
remaining_time)
        case _:
            if empty_STATION[i]:
                set_current_counters(station, job_id, job_category, clock,
service_time)
            else:
                STATION_queue[i].append([job_id, job_category,
service_time])

    if empty_STATION[i]:
        STATION_EMPTY_TIME[i] += clock - STATION_LAST_EMPTY[i]
        STATION_LAST_EMPTY[i] = clock
        empty_STATION[i] = False

    return


def load_job_from_queue(station: str) -> None:
    """
    This function takes the next job in a station's queue and puts it in
position for
    calculation. This station can only be a station that uses FIFO policy.

    Args:
        station (str): The station on which the load will happen.
    """
    global clock
    i = station_index[station]
    if not STATION_queue[i]:
        set_current_counters(station, None, None, None, None)
        empty_STATION[i] = True
        STATION_LAST_EMPTY[i] = clock
    else:
        job, category, remaining_time = STATION_queue[i].popleft()
        set_current_counters(station, job, category, clock, remaining_time)

    return


previous_regen_point = 0
jobs_per_category: tuple[list[int]] = ([], [], [])  # re-initialized for
each cycle

cycle_index: int = 0
cycles_length: list[int] = [None]
Qj_cycle: tuple[list[int]] = ([None], [None], [None])
# Qt: list that contains the number of jobs in the system each time an
event happens (clears after each cycles ends)
Qt: list[int] = []  # re-initialized for each cycle
# yi: interval of Qt divided by i-th cycle's length
yi: list[float] = [None]
check_condition = lambda ratio: ratio < 0.1

# average1() computes the average value of a list disregarding its 1st
element
average1 = lambda lst: sum(lst[1:]) / (len(lst) - 1) if len(lst) > 1 else
None
```

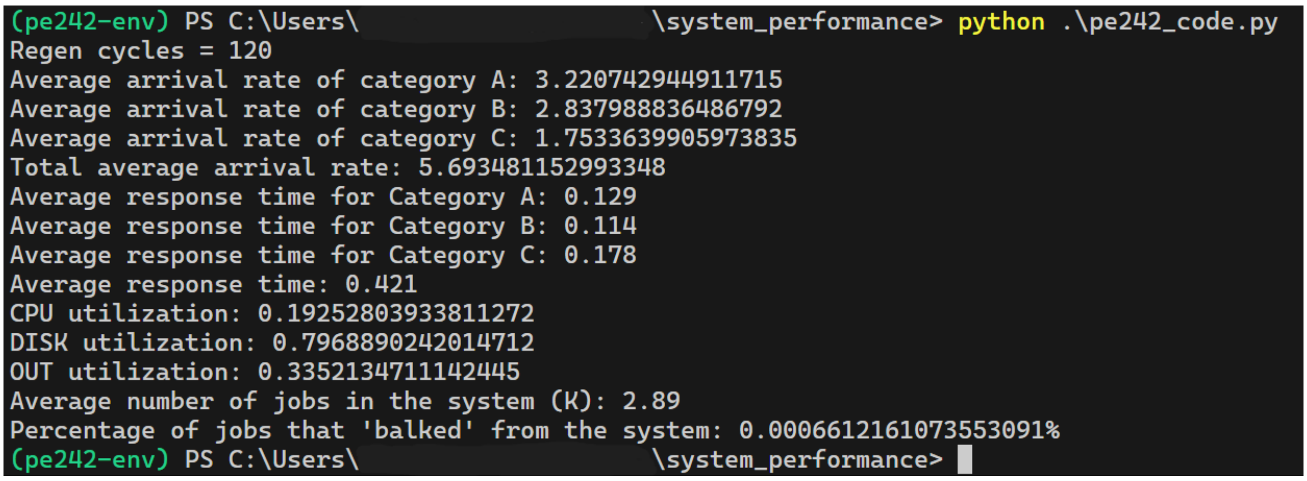
```python
average = lambda lst: sum(lst) / len(lst) if lst else None
sum1 = lambda lst: sum(lst[1:]) if lst else None

while cycle_index < 1000:
    is_system_in_initial_state = curr_jobs == 0
    if is_system_in_initial_state and cycle_index != 0:
        cycles_length.append(clock - previous_regen_point)
        yi.append(sum(Qt) / cycles_length[cycle_index])
        previous_regen_point = clock

        # Result statistics: lamda_j, R_j, U_i
        for j in range(len(categories)):
            Qj_cycle[j].append(len(jobs_per_category[j]))
            lamda_j_per_cycle[j].append(
                Qj_cycle[j][cycle_index] / cycles_length[cycle_index]
            )

            R_j_temp = []
            for job in jobs_per_category[j]:
                R_j_temp.append(departure_time_of_job[job] -
arrival_time_of_job(job))
            R_j_per_cycle[j].append(
                sum(R_j_temp) / len(R_j_temp) if jobs_per_category[j] else
0
            )

        for i in range(len(stations)):
            U_i_per_cycle[i].append(
                1 - STATION_EMPTY_TIME[i] / cycles_length[cycle_index]
            )
            STATION_empty_time_per_cycle[i].append(STATION_EMPTY_TIME[i])
            STATION_EMPTY_TIME[i] = 0

        if cycle_index % 20 == 0 and cycle_index != 0:
            y_bar = average1(yi)
            c_bar = average1(cycles_length)
            n = cycle_index

            sy_2 = 0
            sc_2 = 0
            syc = 0
            for index in range(1, cycle_index):
                sy_2 += (yi[index] - y_bar) ** 2
                sc_2 += (cycles_length[index] - c_bar) ** 2
                syc += (yi[index] - y_bar) * (cycles_length[index] - c_bar)

            sy_2 /= n - 1
            sc_2 /= n - 1
            syc /= n - 1

            R = y_bar / c_bar
            s_2 = sy_2 - 2 * R * syc + (R**2) * sc_2

            a = 0.05  # βαθμός εμπιστοσύνης 95% = 1 - α
            z1_a_2 = 1 - a / 2
            s = float(np.sqrt(s_2))
            confidence_interval = z1_a_2 * s / (c_bar * float(np.sqrt(n)))
            if check_condition(confidence_interval / R):
                break

        for j in range(len(categories)):
            jobs_per_category[j].clear()
        Qt.clear()
        cycle_index += 1

    if cycle_index == 0:
        cycle_index = 1

    time_of_arrival: float = arrival_time_of_job(job_id)
    event = next_event(time_of_arrival)

    if event == "arrival":
        clock = time_of_arrival
        disk_visits[job_id] = 0
        c: str = job_category()
        add_job_to_station("CPU", job_id, c)
        jobs_per_category[category_index[c]].append(job_id)
        curr_jobs += 1
        job_id += 1
    else:
        station = event
        i = station_index[station]
        clock = STATION_start_time[i] + STATION_remaining_time[i]
        match station:
            case "CPU":
                vij_DISK_category =
vij[station_index["DISK"]][category_index[c]]
                disk_visit_probability = vij_DISK_category /
(vij_DISK_category + 1)

                target_station = "DISK" if U() < disk_visit_probability
else "OUT"
                if target_station == "DISK":
                    disk_visits[STATION_current_job[i]] += 1
                add_job_to_station(
                    target_station, STATION_current_job[i],
STATION_current_category[i]
                )
                decrease_CPU_remaining_time(STATION_remaining_time[i])
                STATION_queue[i].popleft()
                if not STATION_queue[i]:
                    set_current_counters(station, None, None, None, None)
                    STATION_LAST_EMPTY[i] = clock
                    empty_STATION[i] = True
                else:
                    job, category, remaining_time = STATION_queue[i][0]
                    set_current_counters(station, job, category, clock,
remaining_time)
            case "DISK":
                add_job_to_station(
                    "CPU", STATION_current_job[i],
STATION_current_category[i]
                )
                load_job_from_queue(station)
            case "OUT":
```

```python
                departure_time_of_job[STATION_current_job[i]] = clock
                curr_jobs -= 1
                load_job_from_queue(station)

    Qt.append(curr_jobs)
    status(event)


print(f"Regen cycles = {cycle_index}")
t = list(range(len(plot_list[0])))
CPU_jobs_per_cycle, DISK_jobs_per_cycle, OUT_jobs_per_cycle = plot_list
total_jobs_per_event = [
    cpu + disk + out
    for cpu, disk, out in zip(
        CPU_jobs_per_cycle, DISK_jobs_per_cycle, OUT_jobs_per_cycle
    )
]

# Histogram
plt.hist(total_jobs_per_event, bins="auto")
plt.title("Histogram of jobs on the system")
plt.xlabel("Number of jobs")
plt.ylabel("Events with this number of jobs")
plt.show()


### OUTPUT RESULTS
# Ζητούμενα: λ, λj, R, Rj, Ui, balking_percentage

for j, category in enumerate(categories):
    lamda_j[j] = sum1(lamda_j_per_cycle[j]) / cycle_index
    print(f"Average arrival rate of category {category}: {lamda_j[j]}")
print(f"Total average arrival rate: {1 / average_arrival_time}")

for j, category in enumerate(categories):
    R_j_per_cycle_without_zeros = [elem for elem in R_j_per_cycle[j] if
elem != 0]
    R_j[j] = round(
        float((average1(R_j_per_cycle_without_zeros) /
average1(Qj_cycle[j]))), 3
    )
    print(f"Average response time for Category {category}:
{float(np.round(R_j[j],3))}")
R = sum(R_j)
print(f"Average response time: {R}")

for i, station in enumerate(stations):
    # U_i[i] = round(float(1 - average1(U_i_per_cycle[i]) /
average1(cycles_length)), 3)
    U_i[i] = (clock - sum1(STATION_empty_time_per_cycle[i])) / clock
    print(f"{station} utilization: {U_i[i]}")


# times_of_N_jobs is a dictionary that maps the total number of jobs N in
the system
# with the number of events that happend while N number of jobs where in
the system
times_of_N_jobs: dict[int, int] = {}
```

```python
for total_jobs in total_jobs_per_event:
    if not total_jobs in times_of_N_jobs.keys():
        times_of_N_jobs[total_jobs] = 0
    times_of_N_jobs[total_jobs] += 1
total_num_of_events = len(total_jobs_per_event)

average_num_of_jobs = 0
for index, key in enumerate(times_of_N_jobs):
    average_num_of_jobs += key * times_of_N_jobs[key]
average_num_of_jobs /= total_num_of_events

print(
    f"Average number of jobs in the system (K):
{float(np.round(average_num_of_jobs, 3))}"
)

K = lambda: np.random.poisson(average_num_of_jobs)
entered_jobs = job_id - 1

balked_jobs: int = 0
theta = lambda: np.random.normal(loc=12, scale=3)

for event_index in range(total_num_of_events):
    if K() > theta():
        balked_jobs += 1

average_num_of_balked_jobs = balked_jobs / total_num_of_events

balking_percentage = float(
    average_num_of_balked_jobs / (entered_jobs +
average_num_of_balked_jobs) * 100
)
print(f"Percentage of jobs that 'balked' from the system:
{balking_percentage}%")
```

Εκτελώντας τον κώδικα αυτόν καταλήγουμε στα εξής αποτελέσματα:





```
(pe242-env) PS C:\Users\[redacted]\system_performance> python .\pe242_code.py
Regen cycles = 120
Average arrival rate of category A: 3.220742944911715
Average arrival rate of category B: 2.837988836486792
Average arrival rate of category C: 1.7533639905973835
Total average arrival rate: 5.693481152993348
Average response time for Category A: 0.129
Average response time for Category B: 0.114
Average response time for Category C: 0.178
Average response time: 0.421
CPU utilization: 0.19252803933811272
DISK utilization: 0.7968890242014712
OUT utilization: 0.33521347111424445
Average number of jobs in the system (K): 2.89
Percentage of jobs that 'balked' from the system: 0.0006612161073553091%
(pe242-env) PS C:\Users\[redacted]\system_performance>
```