

Uniprocessor - Scheduler Simulator

Siddharth SAHAY
Jordi UGARTE

Prof. Joël GOOSSENS, Yannick MOLINGHEN

November 2024



Contents

1	Introduction	1
2	Methodology	2
2.1	Implementation Overview	2
2.2	Project Structure	3
3	Experimental Setup	4
3.1	Usage	4
3.2	Return Codes	4
3.3	Dataset Structure	5
3.4	Evaluation Metrics	5
3.5	Parallel Processing	5
4	Experiment Results	6
4.1	Comparative Algorithm Performance	6
4.1.1	80% utilisation	6
4.1.2	10-tasks	8
4.2	Individual Algorithm Performance	10
4.2.1	Deadline Monotonic	10
4.2.2	Earliest-Deadline First	12
4.2.3	Round-Robin	13
5	Additional Implementations	15
5.1	Rate Monotonic	15
5.2	Audsley	15
5.3	Comparative Performance	15
5.3.1	10 Tasks	15
5.3.2	80% Utilisation	17
6	Summary	18

Abstract

This project involves the development of a simulator that runs the following scheduling algorithms: Deadline Monotonic, Earliest-Deadline First and Round-Robin. The simulator receives a command that specifies the type of algorithm and a specific task set in a CSV format containing the respective attributes of offset (O), computation time (C), deadline (D), and period (T). The simulator checks various metrics about the taskset and outputs whether the task set is schedulable, not schedulable (or additionally, if it took too much time to be executed). The return codes are described in later chapters.

There is also a provision to run multiple tasksets by passing an entire folder in the command. In this case, the simulator plots a pie chart that shows the amount of schedulable, not-schedulable and infeasible task sets. It then also compares the performance of the specified algorithm vs. an optimal one (EDF).

This report dives into the analysis we have performed regarding these scheduling algorithms, and how they perform against each other in two settings.

Scheduling algorithms, as we know, are strategies used to allocate resources to tasks to optimize certain objectives like efficiency, speed, or fairness. For this project, we look at these tasks through the lens of real-time operating systems, where these tasks are thought of as having a hard deadline, i.e. missing a deadline for a task could be catastrophic, if not leading to system failure or severe performance degradation. Real-world applications of such scheduling algorithms are countless - they are essential in areas like networking, manufacturing, and of course operating systems, and many more. The scheduling algorithms we study in this project are the following:

- Deadline Monotonic (DM)
- Earliest-Deadline First (EDF)
- Round-Robin (RR)

The idea of determining feasibility is simple: we need to certify real-time applications and machines so that we can know in advance that the systems will not fail in real-life situations. We must therefore study these algorithms, determine optimality and figure out applications ahead of time.

A very quick brief of the algorithms we will consider:-

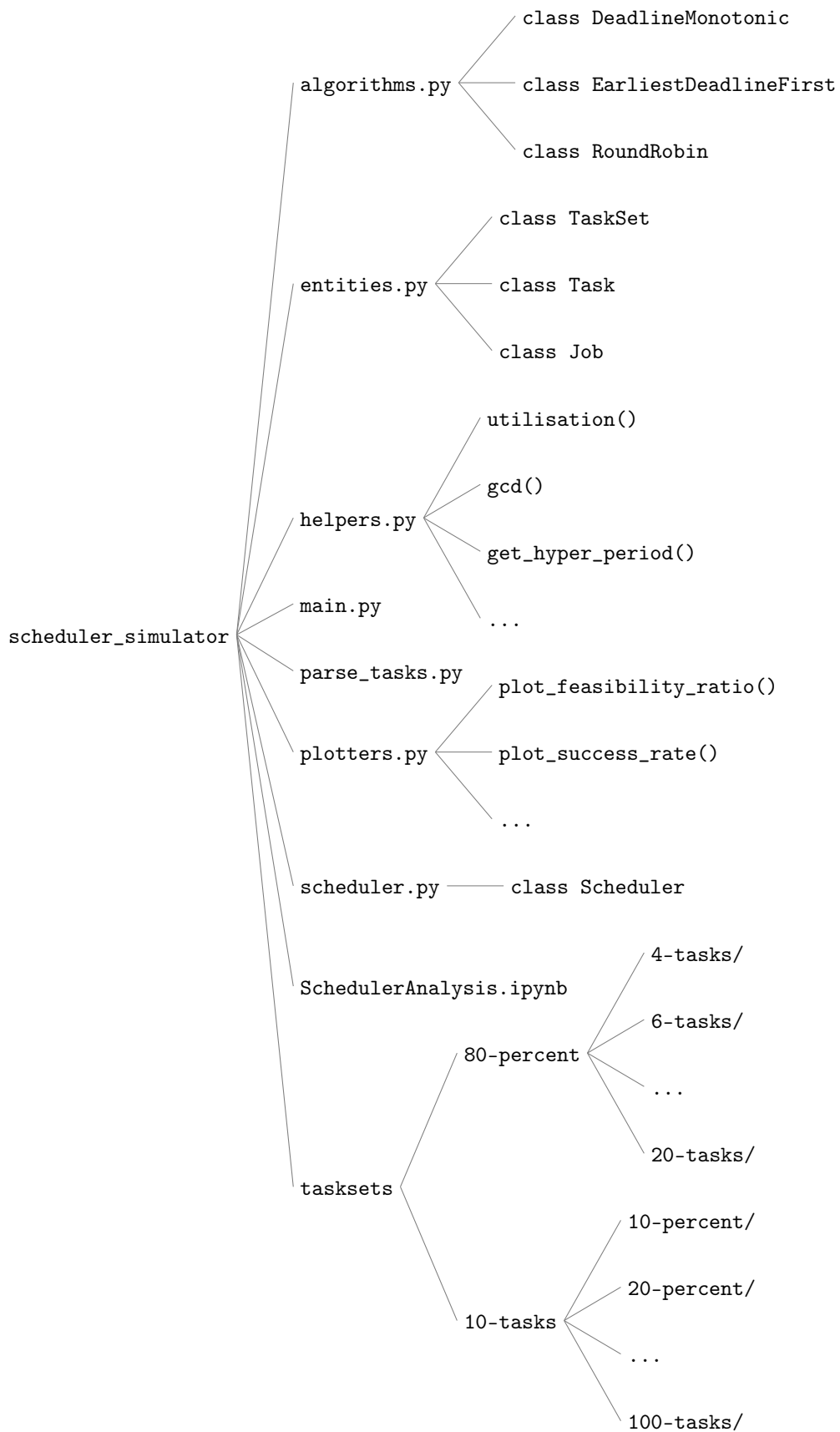
- **Deadline Monotonic (DM):** A fixed-priority algorithm where tasks are assigned priorities based on their deadlines—tasks with shorter deadlines receive higher priority. DM is widely used in real-time systems where tasks have hard deadlines, as it can guarantee schedulability under specific conditions, especially for periodic tasks with fixed constraints. But it is not an optimal algorithm, as we will see.
- **Earliest-Deadline First (EDF):** A dynamic-priority algorithm that prioritizes tasks with the nearest deadline. EDF is considered optimal for single-processor real-time scheduling as it maximizes the likelihood of all tasks meeting their deadlines, given feasible task sets (i.e. utilisation ≤ 1)
- **Round-Robin (RR):** A time-sharing, cyclic scheduling algorithm that allocates equal time slices to each task in a rotating order. Though it is not typically used for strict real-time constraints, RR can *sometimes* offer fairness and simplicity, making it useful in scenarios where responsiveness and workload distribution are key. In real-time operating systems though, it is usually never the right option, as we will establish in later chapters.

2.1 Implementation Overview

Python is our choice of programming language for this project. The code is structured in the following manner:-

- There are classes describing the task sets we will consider; the main TaskSet class contains a set of Tasks. Tasks are the model of a task containing attributes such as offset (O), computation Time (C), deadline (D) and period (T). Each task will in turn spin instances of Jobs, which are described by the remaining computation time and the release time.
- There is a main Scheduler class, which is the parent class, almost abstract in its implementation. It is meant to be extended by each algorithm to override default behaviour and functions, e.g., getting top priority, getting the simulation interval, etc. The basic simulation method which is the core algorithm within all schedulers is implemented in this class.
- Each algorithm class inherits from the Scheduler class and overrides basic functionality. These child classes help dictate various properties of each algorithm. How each algorithm determines feasibility is defined here in the child classes.
- There are also helper functions that help determine the feasibility of a task set and include some common functions to categorize it, such as: utilisation formulas, GCD (greatest-common divisor), LCM (least-common multiple), hyper-period, feasibility interval, delta time step.
- We also have methods to plot the results we obtain from the simulator.
- The whole project can be tested entirely by running a Python notebook called SchedulerAnalysis, which contains all the executions to run the folder tasksets. Once the notebook is fully run, it will plot the success rate of each folder. The folders of tasks sets are described later.

2.2 Project Structure



3.1 Usage

To run the code, use the following command:

```
python3 main.py [rm|dm|audsley|edf|rr] <task_set_location> -v -f
```

- **Algorithm:** Choose from `dm`, `edf`, `rr` (or even `rm` and `audsley`).
- **Task Set Location:**
 - If a **single task set file** is specified, the code evaluates just that file.
 - If a **directory** is specified, it evaluates all task sets within the folder (recursively) in parallel.
- **Options:**
 - `--verbose` or `-v`: Provides detailed output including the whole schedule, if a simulation is done, and a text output instead of just the return code.
 - `--force_simulation` or `-f`: Forces full simulation, even if utilization is below a set threshold. This was mainly used for debugging purposes and testing.

3.2 Return Codes

The program returns specific codes to indicate the outcome of each task set evaluation:

- **0:** The task set is schedulable, and the execution was simulated.
- **1:** The task set is schedulable, and a shortcut was used instead of full simulation.
- **2:** The task set is not schedulable, and the execution was simulated.
- **3:** The task set is not schedulable, and a shortcut was used instead of full simulation.
- **4:** Time-out, this was added on top of the existing return codes to account for the many task sets that had too long a feasibility interval to do a simulation.

These return codes provide insight into both the schedulability of each task set and whether the evaluation required a full simulation. We also introduced an additional return code of 4 to represent task sets that time out! These "undecided" task sets also form a small portion of the task set data we were provided, as we will see in later chapters.

3.3 Dataset Structure

The dataset is organized into two main types of folders: 80-percent and 10-tasks, each grouping task sets based on specific parameters.

- **80-percent:** This folder contains task sets with an average utilization of approximately 80%, with varying numbers of tasks. Here, our objective is to determine how the number of tasks affects the schedulability of each algorithm.
- **10-tasks:** This folder includes task sets where each set has exactly 10 tasks, but utilization varies. Our goal here is to assess how changing utilization impacts the schedulability and success rate of each scheduling algorithm.

We have used the second dataset that was provided, where most tasks have a worst-case execution-time of 1 but have the advantage of offering shorter hyper periods than the original dataset. This allows faster analysis of these algorithms.

In the next chapter, we will present the results of our analysis of these folders by calculating the ratio of feasible task sets and the success rate of each algorithm for each folder. This helps us understand the performance and limitations of the algorithms in different scenarios.

3.4 Evaluation Metrics

The key metrics used to assess algorithm performance are:

- **Schedulability:** Whether a task set meets its deadlines under each algorithm.
- **Success Rate:** Calculated as the ratio of schedulable task sets to total task sets for each algorithm.

If we run the command mentioned before on a particular folder, we plot statistics related to the different return codes, and also compare the performance of the algorithm with an optimal scheduling algorithm (which we consider as EDF).

3.5 Parallel Processing

We have used Python's multiprocessing module to implement parallel processing of task sets, in the case a folder containing multiple files is provided as input. The constant **NUMBER_OF_PARALLEL_PROCESSES** in *main.py* is set to 8, to use 8 cores but can easily be changed to suit a different processor and environment.

This is done to be able to efficiently process the entire folders of 80-percent/ and 10-tasks/ that each contain close to 5000 task sets.

From our task set data, we can compare our three algorithms (DM, EDF, RR) side by side in two distinct scenarios. The first scenario involves fixing the utilisation at 80 percent and varying the number of tasks. This allows us to observe how each algorithm handles different levels of task concurrency under a high-utilisation environment. By examining performance as the number of tasks increases, we can evaluate each algorithm's ability to maintain schedulability as the system becomes more loaded, which is particularly useful for applications where the task count may fluctuate but the overall workload is consistently high. Here, we consider 80% to be high in this regard, since the Liu and Layland formula already tells us that as the number of tasks tends to infinity, the utilisation of fixed-priority task sets is around 69%, and 80% is well above that limit.

The second scenario involves fixing the number of tasks at 10 and varying the utilisation. This setup enables us to assess the algorithms' performance as the intensity of the workload changes. As utilisation increases, tasks consume a larger proportion of available processing time, which can push the system "closer to its limits". This scenario is valuable for understanding each algorithm's adaptability to increasing demands, particularly in systems where the number of tasks is stable but task execution requirements may vary, such as in applications with changing workloads.

Comparing the algorithms in these two scenarios provides insight into their strengths and weaknesses under different types of load conditions. It allows us to identify which algorithms are better suited for systems with fluctuating task counts versus those with fluctuating utilisation, helping inform algorithm choice for real-world applications with similar constraints. Though, in our case, as we will see, there is a clear winner!

4.1 Comparative Algorithm Performance

4.1.1 80% utilisation

4.1.1.1 Varying number of tasks

Let's first look at the ratio of tasks that are feasible to establish the baseline in Figure 4.1.

This graph shows the feasibility ratio of task sets for EDF scheduling at a fixed 80% utilisation while varying the number of tasks. The high feasibility ratio at lower task counts indicates EDF's effectiveness with smaller task sets. However, as the number of tasks increases, the feasibility ratio drops sharply, especially beyond 12 tasks. This provides us with a benchmark to now calculate the success rate of each of the other algorithms and discuss how well they performed.

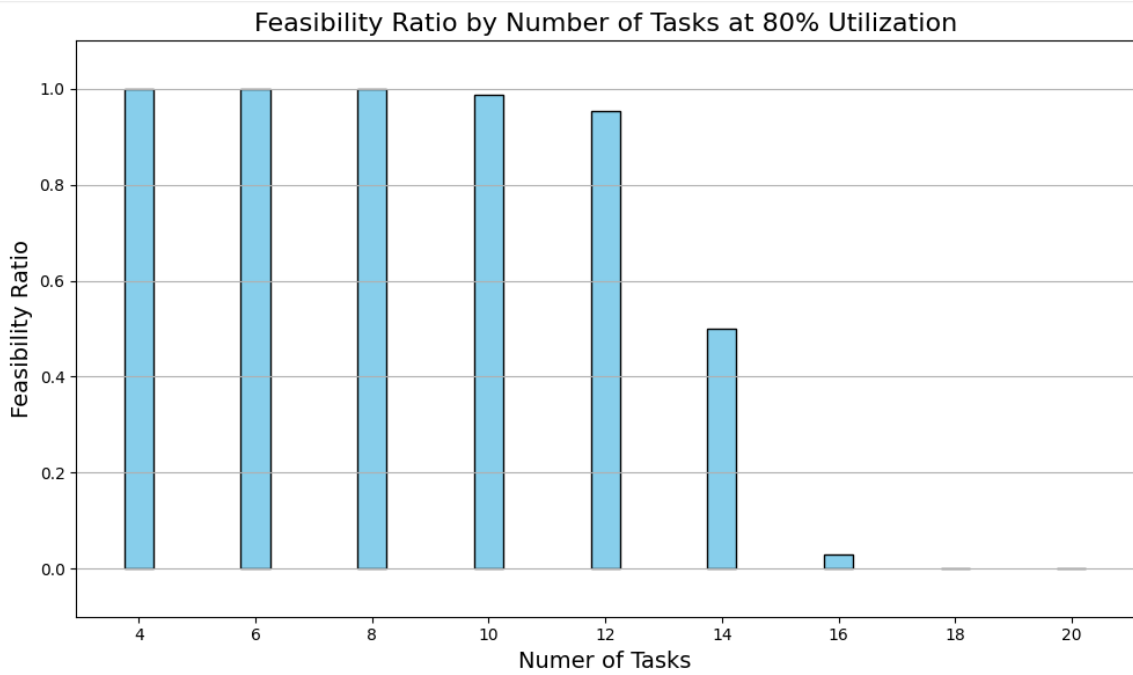


Figure 4.1: Feasibility Ratio - 80% Utilisation, Varying Number of Tasks

Info Note

This decline is mainly due to the dataset being randomly generated. Although intended for 80% utilisation, some task sets exceed 100% utilisation, making them infeasible even with an optimal scheduler.

EDF is of course an optimal scheduling algorithm, which means it can successfully schedule any feasible task set that has a utilisation below 100%. So, the observed decline in the feasibility ratio is mainly a "byproduct" of the dataset's random generation process. This randomness sometimes created infeasible task sets, which reduces the overall feasibility ratio at higher task counts. But it does not reveal any weakness in the EDF algorithm itself.

4.1.1.2 Success Rate of DM, EDF, RR

This graph shows the success rate of our three algorithms in question at 80% utilisation while varying the number of tasks executed starting with 4 tasks and going up to 20 tasks.

Deadline Monotonic here has a success rate of about 60% when the number of tasks is relatively low, at 4. As this number goes up, we see a sharp decline in success rate. This is likely owing to the fact that DM is a static priority assignment algorithm and as the number of tasks goes up, it fails to handle the tighter time windows to execute tasks without deadline misses. Small delays quickly cascade through the system and we see a lot of deadline misses - even as early as when the system has just 12 tasks to handle with 80% utilisation.

Round Robin has the lowest success rate among the three algorithms, as expected, beginning near zero even with the lowest task counts and remaining at or near zero as the number of tasks increases. This poor performance reflects Round Robin's non-optimal approach for real-time scheduling tasks with strict deadline requirements.

Since RR doesn't prioritize tasks based on deadlines, it fails to maintain feasibility effectively, especially under high utilization.

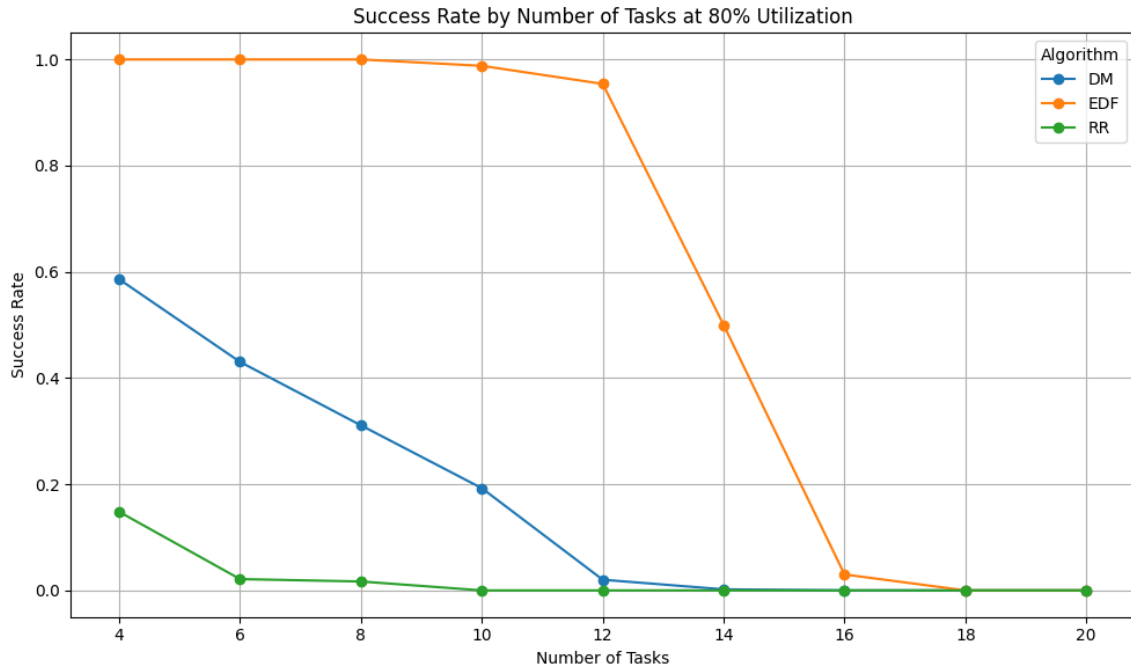


Figure 4.2: Success Rate - 80% Utilisation, Varying Number of Tasks

4.1.2 10-tasks

4.1.2.1 Varying utilisation

The feasibility ratio for EDF with varying utilisation and a fixed number of tasks can be seen in Figure 4.3. This plot provides a benchmark for comparing DM and RR against the optimal scheduler for our dataset.

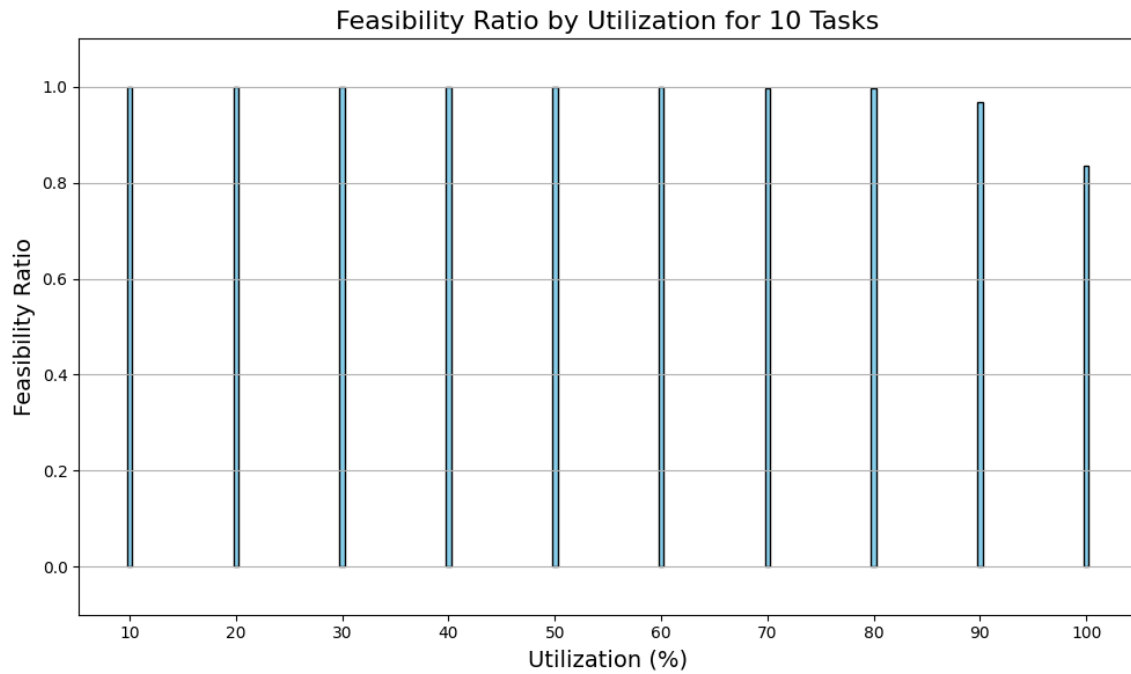


Figure 4.3: Feasibility Ratio - 10-tasks, Varying Utilisation

Info Note

This decline in feasibility is again because of the element of randomness in the dataset during generation. Near the 90% mark we see a decrease in feasibility even from EDF, but this can be attributed to utilisation being arbitrarily above 100% and not due to EDF being unable to schedule the task sets.

4.1.2.2 Success Rate of DM, EDF, RR

The success rate of the algorithms in question with a fixed number of 10 tasks execution and a varying utilisation is shown on Figure 4.4.

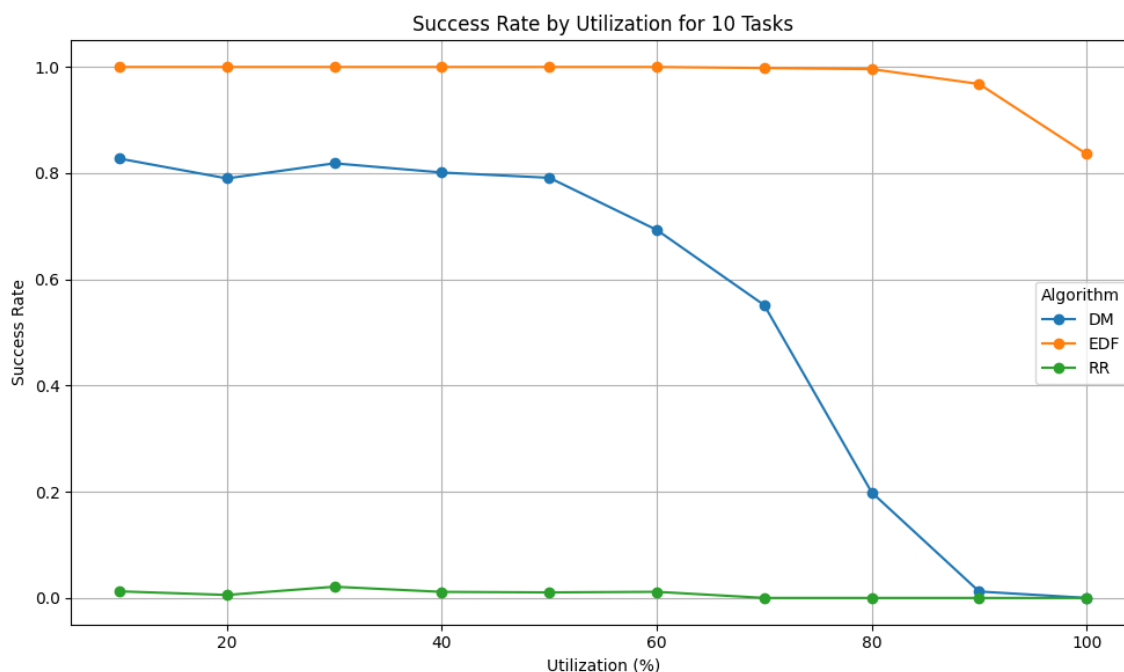


Figure 4.4: Success Rate - 10-tasks, Varying Utilisation

This trend is similar to the one we saw in the previous section with a varying number of tasks. Performance for DM declines rapidly after the 70% utilisation mark, with number of tasks kept constant. RR is unaffected by utilisation as success rate is near-0 throughout.

There is quite an interesting correlation here with the Liu and Layland bound, which says that an average with a large number of tasks, is around 69%. So, if utilisation is kept below this bound, DM is likely to meet deadlines, on average! And we can clearly see this in the graph above.

RR proves again to be unsuitable for use in real-time scheduling scenarios.

The performance of EDF is also noteworthy - with utilisation kept below 100%, it should theoretically be able to schedule all tasksets, but as explained in the "Info Note" above, due to randomness creeping into the task set generation we have a percentage of tasks in the 90% and 100% utilisation categories that spill over 100% utilisation and therefore are infeasible. But barring this, EDF's performance is far superior to DM (and, of course, RR).

4.2 Individual Algorithm Performance

Lastly, we will analyze the performance of each algorithm individually across the dataset, focusing on the 80% utilisation and 10-tasks categories.

In the graphs below we also see a portion attributed to "Timed Out". This is added specifically to be able to compare all the algorithms at a 1:1 ratio, instead of comparing different algorithms where a different number of tasks had a decided outcome (vs. tasks that were "undecided").

4.2.1 Deadline Monotonic

4.2.1.1 10 Tasks

- **Feasible Task Sets:** 53.4% of task sets were feasible, with 41.3% quickly identified using shortcuts and 12.1% requiring full simulation.
- **Not Schedulable Task Sets:** 44.2% were unschedulable, with 42.2% confirmed through simulation and only 2% using shortcuts.
- **Timed Out:** 2.4% of task sets timed out, due to the simulation interval being too large.

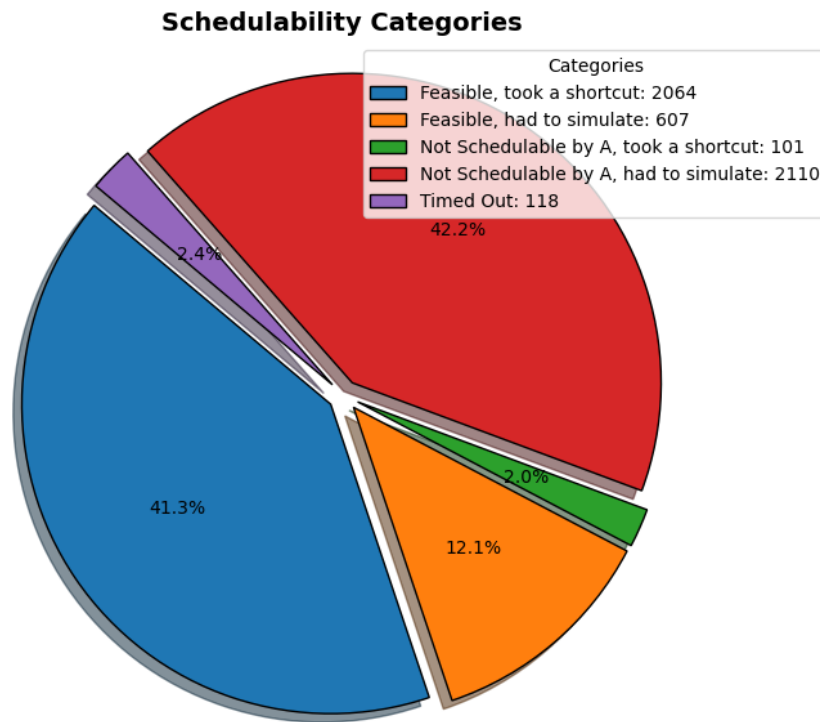


Figure 4.5: Feasibility of DM in 10-tasks/

With 10 tasks, DM could quickly determine schedulability for a substantial portion (41.3%), though it still needed full simulation for a large percentage of cases, especially those concluded to be infeasible. This suggests that as the task set grows, DM's efficiency somewhat drops, requiring more detailed analysis to assess feasibility accurately.

Use of busy period for DM

The feasibility interval (for DM, and RM) is given by:-

$$L = O_{\max} + \text{lcm}(T_1, T_2, \dots, T_n)$$

But in many cases, due to the randomness in the generation of our task sets, the LCM of the periods would be comfortably in the millions (even in the billions), which makes it impractical to simulate the schedule for each task set, given there were roughly 10,000 task sets in all to compute. In these cases, we have resorted to using the busy period where relevant.

$$W_i^{(k+1)} = \sum_{j=1}^i \left\lceil \frac{W_i^{(k)}}{T_j} \right\rceil C_j$$

The busy period is shorter than the feasibility interval because it only includes the continuous interval where tasks are "back-to-back" without idle slots. In Deadline Monotonic, it's sufficient to check feasibility within this busy period because it represents the worst-case scenario for scheduling demand. If all deadlines are met within this interval, they will also be met in subsequent cycles, as the task execution pattern repeats. This makes the busy period a more efficient and practical interval for determining schedulability compared to the longer feasibility interval.

4.2.1.2 80% Utilisation

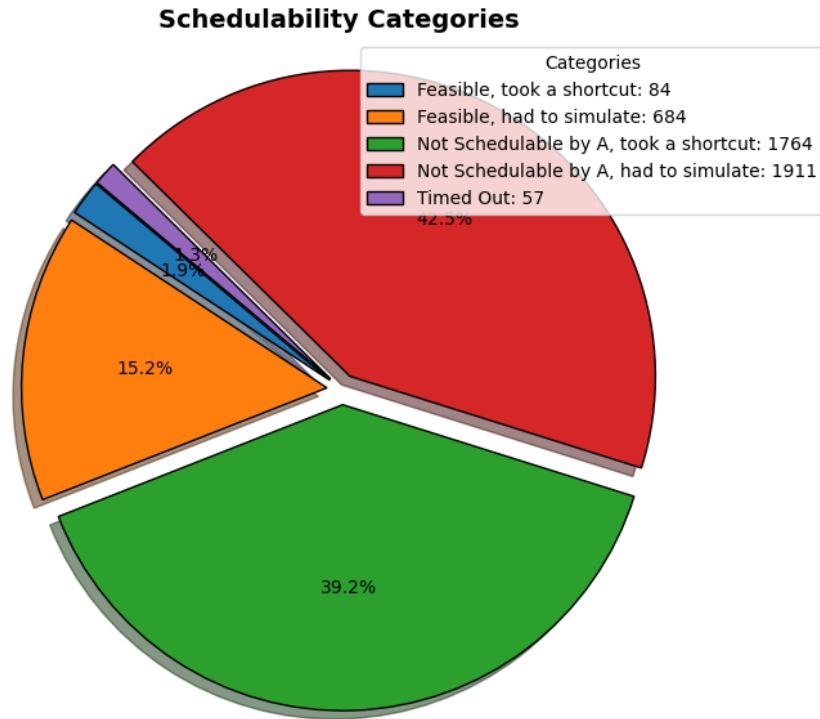


Figure 4.6: Feasibility of DM in 80-percent/

- **Feasible Task Sets:** Only 17.1% of task sets were feasible, with very few (1.9%) identified using shortcuts and the majority (15.2%) requiring full simulation.

- **Not Schedulable Task Sets:** 81.7% were unschedulable, with 39.2% quickly identified as infeasible using short-cuts, while 42.5% required simulation.
- **Timed Out:** 1.3% of task sets timed out.

At 80% utilization, DM even struggled more with feasibility, with only 17.1% of task sets marked as feasible. The high utilization pushed many sets beyond its limits, with DM only able to quickly identify infeasibility for 39.2% (quite low). Relying on simulation for checking feasibility reflects the problems DM faces in high-utilization scenarios, where it's harder to meet deadlines due to limited system capacity.

4.2.2 Earliest-Deadline First

EDF, on the other hand, was completely decidable when determining feasibility.

4.2.2.1 10 Tasks

- **Feasible Task Sets:** The vast majority of task sets (98%) were deemed feasible without needing full simulation, reflecting EDF's high efficiency and effectiveness in handling a 10-task workload.
- **Not Schedulable by A, took a shortcut:** A very small portion (2%) of task sets were quickly identified as unschedulable. These had a sporadic utilisation above 100%, as discussed earlier.
- **No Simulations Required:** EDF did not need to run any full simulations, we can tell purely by the utilisation if a task is schedulable by EDF, since it is optimal.

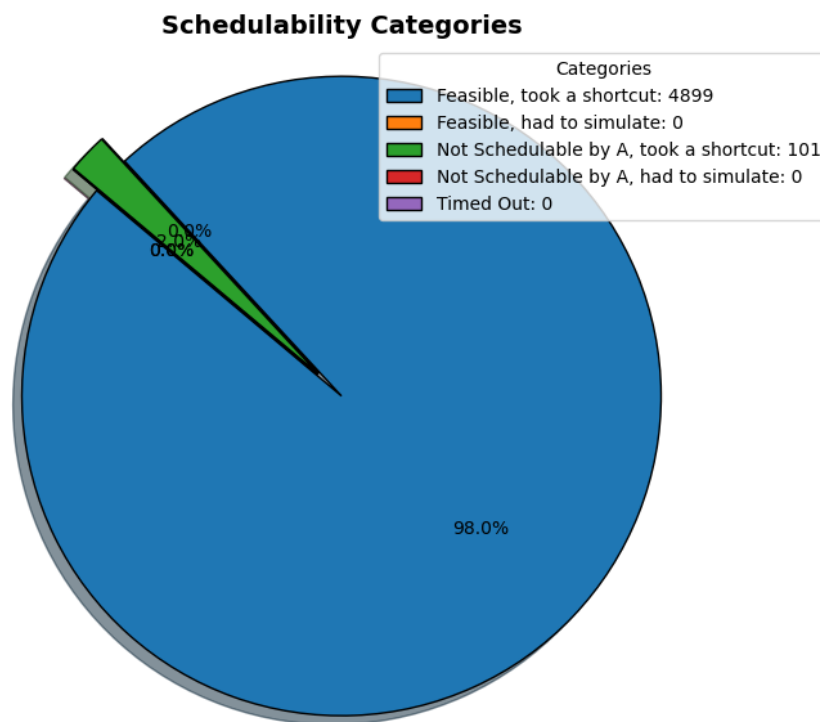


Figure 4.7: Feasibility of EDF in 10-tasks/

4.2.2.2 80% Utilisation

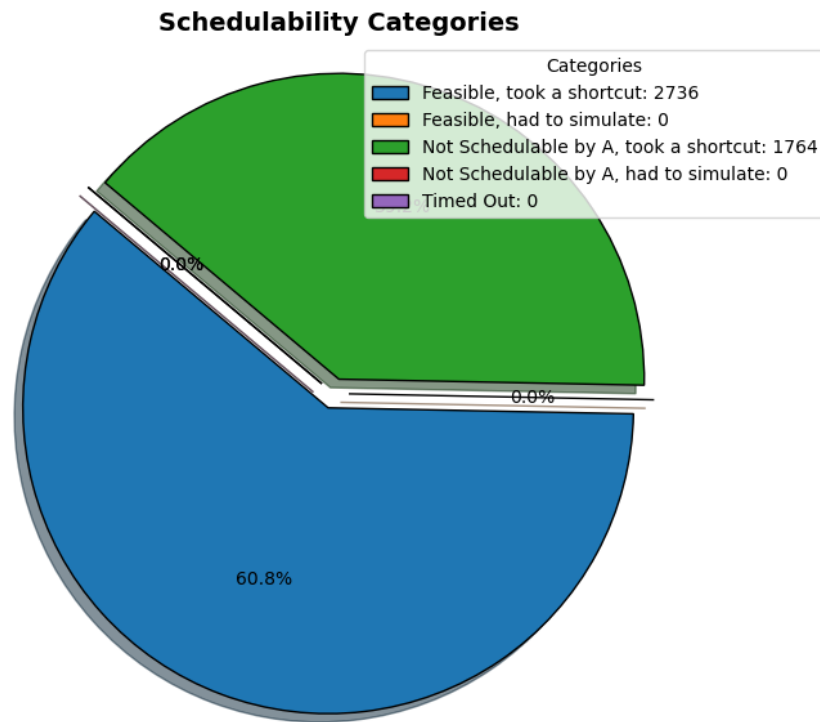


Figure 4.8: Feasibility of EDF in 80-percent/

- **Feasible Task Sets:** 60.8% were feasible without simulation, reflecting higher complexity at 80% utilization.
- **Not Schedulable by A, took a shortcut:** 39.2% were quickly identified as unschedulable. This can be explained because some task sets had a sporadic utilisation above 100%, as discussed earlier, it is not a comment on the optimality of EDF.
- **No Simulations Required:** Again, no simulations are ever required for EDF since we can tell purely by the utilisation.

4.2.3 Round-Robin

Round Robin was not feasible at all, compared to the other algorithms, having problems with almost all tasks.

This categorically proves that RR cannot handle deadline constraints effectively, as it processes tasks in a cyclic, time-sliced manner without considering task priorities or deadlines. With over 63% timed-out tasks and around 35% infeasible cases in both scenarios, RR's equal time-slicing fails to meet real-time deadlines, making it highly non-optimal for real-time applications.

4.2.3.1 10 Tasks

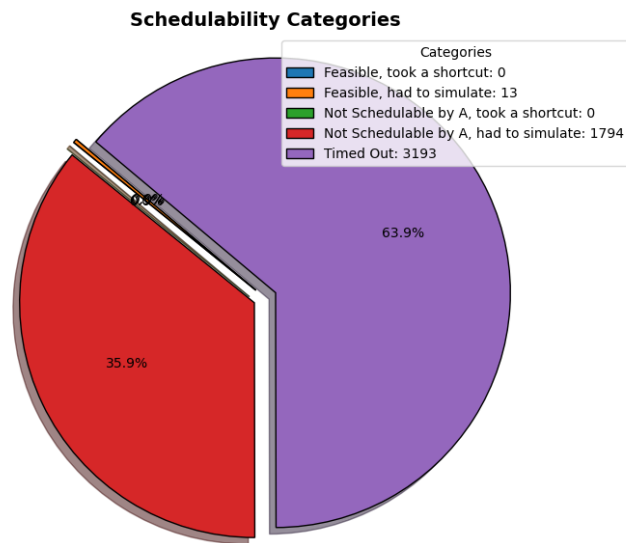


Figure 4.9: Feasibility of RR in 10-tasks/

4.2.3.2 80% Utilisation

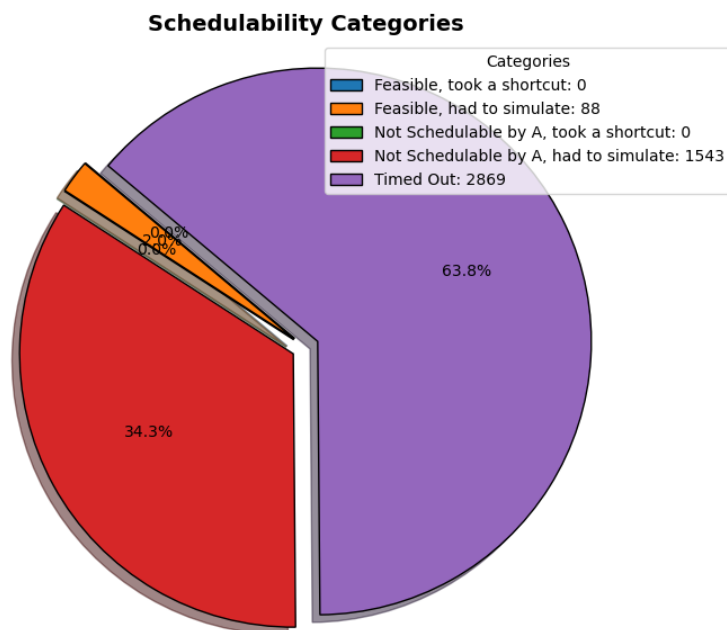


Figure 4.10: Feasibility of RR in 10-tasks/

Additional Implementations

Other scheduling algorithms, such as Rate Monotonic and Audsley, were implemented as well to experiment with different algorithmic dynamics. The *main.py* accepts as argument `rr` and `audsley` in addition to the others.

5.1 Rate Monotonic

Rate Monotonic (RM) is also a fixed-priority scheduling algorithm where tasks are assigned priorities based on their periods—tasks with shorter periods receive higher priority. RM is commonly used in real-time systems for periodic tasks, as it offers guaranteed schedulability under certain utilization bounds (the Liu and Layland bound!) However, RM is not optimal for all real-time scenarios, as it may fail to meet deadlines in cases where task periods are non-harmonic or utilization exceeds specific thresholds.

5.2 Audsley

Audsley is an optimal priority assignment algorithm for fixed-priority scheduling. Audsley's Algorithm systematically assigns priorities to tasks in a way that maximizes schedulability, ensuring that each task's deadline can be met if there exists a feasible fixed-priority assignment. Although it provides an optimal priority order for schedulable tasks, it is still limited by the fixed-priority approach and cannot adapt dynamically to changing task demands as a dynamic scheduler like EDF would.

5.3 Comparative Performance

Adding RM and Audsley into the mix here is very interesting for the comparisons with the algorithms we've already seen!

5.3.1 10 Tasks

RM has a moderate level of success at lower utilisations, remaining at about 60-70% success rate which then starts falling off after 70% utilisation (about the same time as DM. This is also likely due to the Liu and Layland bound that was discussed earlier. RM is also a static priority assignment scheduling algorithm, which implies that if task periods don't align harmonically there are likely to be deadline misses, especially as utilisation goes up.

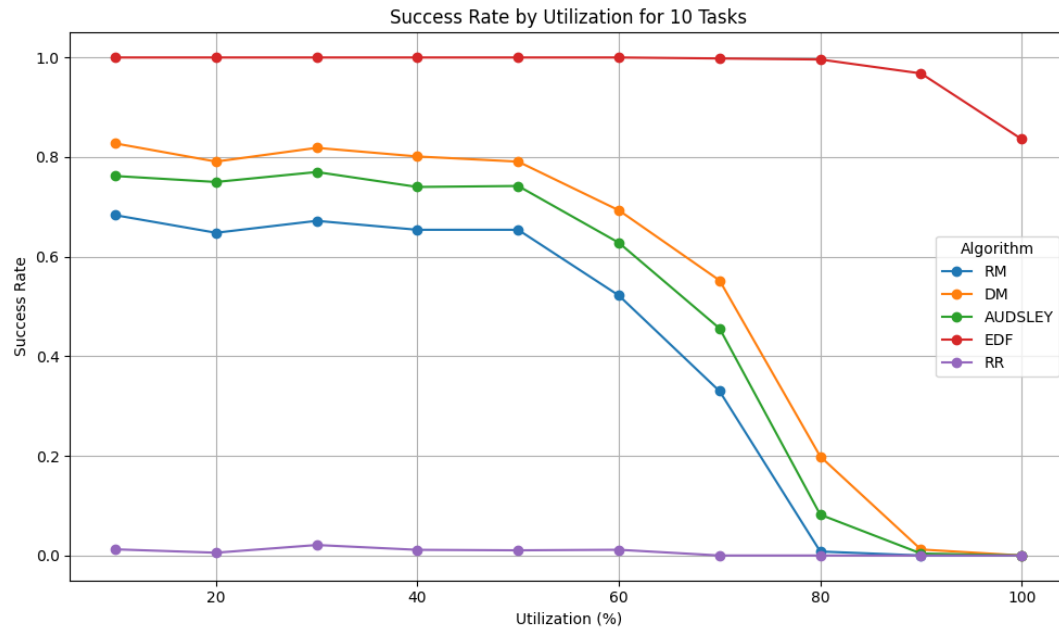


Figure 5.1: Success Rate of RM, Audsley in 10-tasks/

Audsley's algorithm performs similarly to DM, with a high success rate at lower utilization levels and a similar decline after around 60% utilization, despite theoretically being a more flexible priority assignment algorithm! Audsley's algorithm optimally assigns priorities to tasks to maximize schedulability but does so based on fixed priorities (like DM). This means that once priorities are set, they remain static, and Audsley doesn't dynamically adjust priorities like EDF. As utilization increases, Audsley's optimal priority assignment struggles with the load! This is quite a surprising discovery, since Audsley is supposed to be more optimal than DM due to how it assigns priority. Sadly, it is much more difficult to implement though - which might explain why it does not perform as well as DM.

5.3.2 80% Utilisation

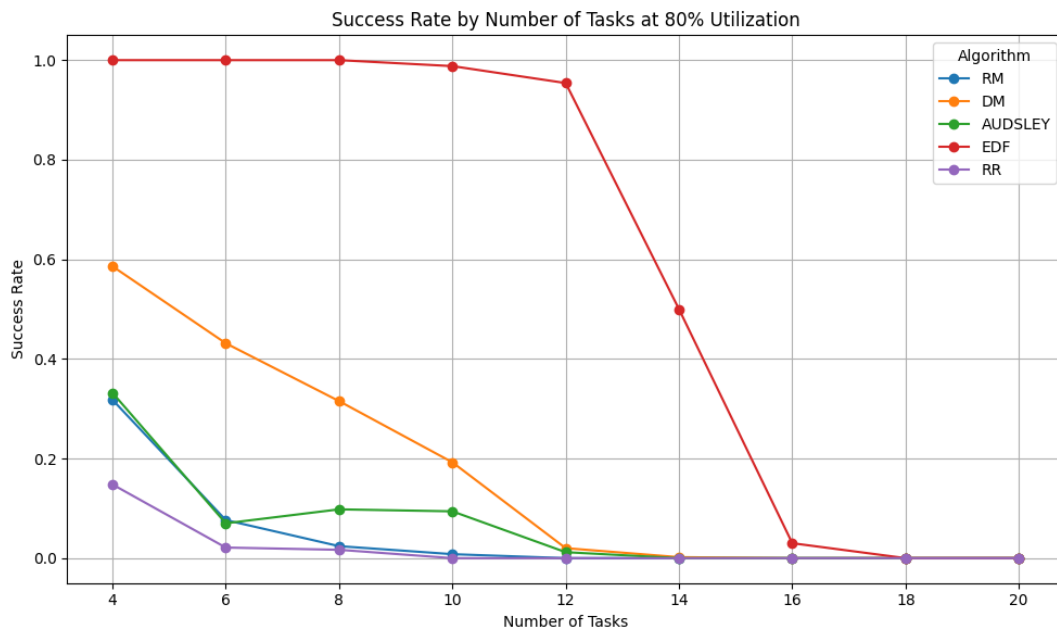


Figure 5.2: Success Rate of RM, Audsley in 80-percent/

The findings here are in line with what we observed in the previous section with a constant set of tasks and varying utilisation.

RM shows poor performance with increasing task count due to its strict period-based priorities.

Audsley again seems to struggle to feasibly schedule task sets that DM can. This can be explained due to it also being a fixed-priority algorithm (even though it has optimal assignment). With more tasks, the optimal priority assignment does not seem to be sufficient - its performance at the end is similar to DM.

This project explores and evaluates the performance of five scheduling algorithms—Deadline Monotonic (DM), Earliest-Deadline First (EDF), Round Robin (RR), Rate Monotonic (RM), and Audsley's Algorithm—through a scheduler simulator. Each algorithm represents a unique approach to task prioritization and scheduling, aiming to maintain task schedulability under different conditions.

Earliest-Deadline First (EDF) emerges as the clear winner and most optimal algorithm for feasible task sets, consistently meeting deadlines across scenarios by dynamically prioritizing tasks with the nearest deadlines. Its dynamic priority assignment matches up to the task of changing needs that even a large task-set has over the course of its execution.

Deadline Monotonic (DM), a fixed-priority algorithm, performs effectively at lower task counts but struggles as task loads increase. DM assigns priorities based on deadlines, giving higher priority to tasks with shorter deadlines, but its static nature makes it less adaptable under high-utilization conditions, leading to increased deadline misses as the number of tasks or utilisation goes up. Where the number of tasks go up - deadline creep is an issue, and where utilisation goes up - a fixed priority scheduler may even cause priority inversion.

Round Robin (RR), a cyclic scheduling approach that allocates equal time slices to each task, proves highly unsuitable for real-time systems with hard deadlines. With no consideration for task deadlines, RR shows infeasibility in maintaining deadlines across the board, regardless of the number of tasks and utilization.

Rate Monotonic (RM) prioritizes tasks based on their periods, with shorter periods receiving higher priority. While RM performs well under specific utilization limits, it lacks the flexibility of EDF and struggles with non-harmonic task periods, especially as utilization exceeds its theoretical bound. This results in a steep decline in success rate at higher task loads. It performs worse than DM in this regard.

Audsley's Algorithm, known for optimal priority assignment in fixed-priority scheduling, theoretically offers the most efficient task priority configuration. However, its fixed-priority nature limits its effectiveness at high utilizations, where it faces challenges similar to DM, despite optimal prioritization.

The simulator reveals EDF's robust performance, DM's moderate success with lower task counts/utilisations, and the utter shortcomings of RR for applications in real-time systems. RM and Audsley demonstrate interesting contrasts to the others but also highlight the limitations of fixed priority algorithms under fluctuating task demands!