SUMMARY

This project presents a novel distributed algorithm called Heuristic Meeting-based Patrolling (HMP) for multi-robot patrolling in environments with limited communication and potential robot failures. The algorithm enhances a reactive strategy by introducing periodic synchronization meetings among robots at shared locations to coordinate tasks, detect failures, and reassign responsibilities.

The environment is partitioned so that each robot is assigned a subregion to patrol. Meetings occur at strategically chosen points, allowing robots to share status and reorganize patrolling duties if a robot becomes unresponsive. A critical feature of HMP is fault tolerance: when a robot fails, others detect the failure via missed meetings and adapt by taking over its area. The algorithm includes mechanisms for rescheduling meetings, load balancing (annealing), and prioritizing tasks to maintain efficiency even after faults.

Multiple HMP variants were implemented and tested using the Multi-Agent Exploration & Patrolling Simulator in simulated maps (e.g., building and cave layouts), and their performance was compared against existing strategies like Conscientious Reactive, Heuristic Conscientious Reactive, Single Cycle, and Expected Reactive. Metrics included average and worst idleness—how long areas go unvisited. Results showed HMP performs competitively under normal conditions and robustly adapts under failure scenarios. However, limitations were found in meeting time scheduling, which in some cases caused cascading failures.

The study concludes by suggesting future improvements in adaptive scheduling and dynamic partitioning to enhance HMP's robustness and performance further.

Distributed Multi-Robot Partition-based Patrolling with Fault Tolerance

Puvikaran Santhirasegaram (psanth20)* · Mads Beyer Mogensen (mmogen20)* · Henrik van Peet (hvanpe20)*
*E-mail: @student.aau.dk
Department of Computer Science, Aalborg University, 9220 Aalborg Øst, DK

Abstract—
Patrolling tasks in multi-robot systems are essential for applications such as surveillance and monitoring, where minimizing the time between visits to any given location is critical. This project presents Heuristic Meeting-based Patrolling (HMP), a novel distributed and fault-tolerant algorithm designed to efficiently patrol partitioned environments with limited communication. Building on the Heuristic Conscientious Reactive (HCR) strategy and incorporating periodic synchronization meetings, HMP enables decentralized coordination and dynamic fault recovery through minimal communication. Robots exchange information at shared meeting points, enabling detection of failures and redistribution of responsibilities. We evaluate HMP and its simplified variants in various simulated environments using the Multi-Agent Exploration and Patrolling Simulator (MAEPS). The results demonstrate that HMP offers strong performance under both normal and fault conditions, comparative to state-of-the-art patrolling strategies in terms of idleness metrics. However, we also identify limitations in meeting scheduling under certain fault conditions, which can cause cascading failures. Based on these findings, we discuss potential improvements for future work, including enhanced meeting scheduling and adaptive partitioning strategies.
Theme: P10 – Specialisation in Distributed Systems
Project Period: 01-02-2025 – 13-06-2025
Project Group: cs-25-ds-10-17
Page number: 10
Date of completion: June 13, 2025

Index Terms—Multi-robot systems, Patrolling, MAEPS, Partitioning.

1 Introduction

Patrolling with multi-robot systems is a critical task in applications like surveillance and monitoring, where minimizing the time any location goes unvisited is key. Many existing algorithms assume ideal conditions—such as full communication and no failures—which limits their practical use.

In this paper, we propose Heuristic Meeting-based Patrolling (HMP), a distributed and fault-tolerant patrolling algorithm based on partitioning the environment and coordinating robots through scheduled meetings. Building on the Heuristic Conscientious Reactive [1] (HCR) strategy and drawing inspiration from the synchronization system developed in [8]. Our approach enables robots to patrol local areas efficiently while meeting periodically to exchange information and detect failures. The required communication range is minimal, due to the robots meeting together at meeting points to communicate. We introduce variants of the algorithm with different fault-handling strategies and evaluate them in simulation across various map types and failure scenarios.

2 Related work

Several studies have investigated the use of a multi-robot system for patrolling an area [25, 15]. Patrolling algorithms can be broadly classified into the following categories: cyclic strategies, distributed coordination, centralized coordination, stigmergy-based methods, and learning-based approaches. This section reviews related work following the same order.

In [10], cyclic path generation in combination with event handling is explored. The paper claims its algorithms to be robust and specifies what to do in case of robot failure, however, it does not discuss how to detect robot failure.

Cyclic strategies are also used in [23], where two patrolling strategies are proposed that incorporate vertex weighting. Both strategies utilize a cyclic path combined with a stop-go mechanism; however, neither is capable of handling robot failures.

SingleCycle [2] (SC) is a cycle algorithm, which calculates the TSP-path for the vertices and follows this cycle.

Caraballo et al. [5] investigate how multi-robot systems can achieve fault-tolerant terrain patrol despite limited communication range. The study focused on creating cyclic area partitions and synchronizing the robots based on the timing of their closest encounters. It also explored the impact of various ”shifting” strategies, where robots take over adjacent partitions when needed.

Random Reactive [18] (RR) randomly walks the graph from vertex to vertex. The Conscientious Reactive [18] (CR) algorithm prioritizes visiting the neighboring vertex with the highest ”idleness” (time since last visit). HCR is an extension of CR that also takes the distance into account

in addition to the idleness when deciding which vertex to visit next.

Portugal et al. [27] proposed two distributed multi-robot patrol algorithms based on a Bayesian framework: the Greedy Bayesian Strategy (GBS) and the State Exchange Bayesian Strategy (SEBS). GBS directs each robot to independently maximize its local utility—defined as the ratio of instantaneous idleness to the estimated travel time—without requiring communication with other robots. In contrast, SEBS enhances decision-making by allowing robots to exchange their intended movements with nearby peers, thereby accounting for the presence and actions of other robots. This coordination typically results in superior performance compared to GBS and earlier distributed strategies, primarily by reducing inter-robot conflicts. However, SEBS's effectiveness can degrade when the number of robots varies dynamically, as it relies on parameters that are sensitive to the robot population size. Building on the concept of declaring intention, Yan and Zhang [32] developed a similar approach called Expected Reactive [32] (ER) that minimizes dependency on such parameters, but with equal performance to SEBS.

Patrolling strategies that are centralized or require global communication between agents clashes with real-world limitations and are therefore not considered in this paper [2, 6].

Another direction of patrolling algorithms stores information in the map, referred to as stigmergy, markers, or flags. This requires robots to be equipped with some kind of information storage device and capable of deploying those on the map. We will not focus on these types of algorithms, such as Reactive with Flags from [1], and other stigmergy patrolling algorithms, [33, 4, 12, 13, 9].

A number patrolling algorithms have also been developed which make use of reinforcement learning, but the robots may not have the onboard compute to support running reinforcement learning or neural network-based policies, which is why these are not covered in this paper [21, 22, 24, 26, 29].

Extensive research has been conducted on distributed reactive algorithms, particularly focusing on reducing robot interference through intention sharing, assuming a limited communication range. Several studies have also explored the development of cyclic patrolling strategies. In the context of cyclic pathing, the integration of synchronized information systems into multi-robot patrolling has been explored. However, the use of reactive strategies incorporating synchronization points for information sharing—similar to a synchronized information system—has not yet been explored.

3 Problem Formulation

In this work, we consider a team of $k > 0$ identical robots, each capable of sensing, communication, and movement. The following combined assumptions apply to both the robots' capabilities and the characteristics of the environment to be patrolled.

To maintain realism, we assume that robots have a limited communication range with reliable communication. Additionally, robot failures are considered possible, as this reflects real-world conditions — a system unable to tolerate such failures would be inherently fragile.

For sensing, we assume that full coverage of the environment can be achieved by simultaneously placing robots at a set of n vertices within the configuration space. That is, if there are k available robots, and $n = k$, each robot is positioned at one of the n vertices, and their combined sensor footprints would entirely cover the environment. Each of these vertices is essential for complete coverage. Since $n > k$, at least one robot must visit multiple viewpoints over time to ensure continuous monitoring of the entire environment. We assume a limited sensing (visibility) range, which is factored into the generation of vertices used to cover the environment during patrol.

The area to be patrolled is modeled as an undirected, connected graph $G = (V, E)$ where V represents the set of vertices to be visited by the robots and $E \subseteq V \times V$ represents the set of edges between the vertices. Each edge is weighted by the time it takes to travel between its nodes[1]:

$$c_{i,j} \text{ is the cost of edge } (i, j) \tag{1}$$

This graph serves as the topological map for the patrolling task and is assumed to be known in advance for the robots.

For efficient patrolling of the environment, the vertices are divided into a set of assignments denoted $\mathcal{A} \subseteq 2^V$ to distribute the workload among the robots. Hence, an assignment a_i is a subset of V. All vertices must be a part of at least one assignment:

$$V = \bigcup_{a_i \in \mathcal{A}} a_i \tag{2}$$

Ideally, assignments consist of neighboring vertices. Each robot has local knowledge of the other robots' assignments. This is modeled as the following mapping:

$$\alpha : \mathcal{A} \to R \tag{3}$$

Initially, the mapping is bijective, a robot has only one assignment and vice-versa. Once a robot becomes faulty, and hence unable to patrol, its assignment is reassigned to another active robot. As a result, the mapping becomes non-injective (a single robot may be responsible for multiple assignments), but it must remain surjective to ensure that all assignments, thus all vertices, continue to be patrolled.

In this work, the evaluation metric used to compare the patrolling algorithms is the instantaneous graph idleness and the worst graph idleness. These evaluation metrics are some of the most used in the field of studying patrolling algorithms [28, 15, 19]

The instantaneous vertex idleness (or simply idleness) of a vertex v at time t is defined:

$$i_v(t) = t - t_{last} \tag{4}$$

where t_{last} is the last time v was visited. In other words, the instantaneous graph idleness is the duration for which vertex v has remained unvisited. Given the instantaneous vertex idleness, the instantaneous graph idleness at tick t is given as:

$$I_G(t) = \frac{\sum_{v \in V} i_v(t)}{|V|} \tag{5}$$

1. $c_{i,i} = 0$, and $c_{i,j} = c_{j,i}$ as the graph is undirected.

Symbol	Explanation
$G = (V, E)$	The patrolling graph
v	The set of vertices
$E \subseteq V \times V$	The set of edges
$c_{i,j}$	Travel time between vertex i and j
R	Set of robots
$\mathcal{A}$	Set of assignments
$\alpha : \mathcal{A} \to R$	Mapping of assignment to robot
$\mathcal{A}_r$	Set of assignments robot r patrols
t_{now}	Current time
t_{last}	Last time vertex v was visited
$i_v(t)$	Instantanous idleness of vertex v at time t
$I_G(t)$	Instantanous idleness of graph G at time t
$\overline{I_G}$	Average idleness of graph G over period T
IW_G	Worst idleness of graph G in period T
$M \subseteq V$	Set of meeting points
m_{next}	The next meeting time of meeting m
$m_{nextnext}$	The next meeting time of meeting m after m_{next}
$R_m \subseteq R$	Set of robots that should attend meeting m
D_a	Max travel time between 2 vertices in assignment a
Δt_a^{min}	Min time between two meetings in assignment a
Δt_{Global}^{min}	Min time between two meetings in any assignment
G_M	Meeting graph
$c(m)$	The color of meeting m
T_{cycle}	Meeting cycle
pt_r	Robot r's meeting time proposal

TABLE 1: Symbol Reference

That is, the average idleness across all vertices at that moment. The average idleness of graph G over period T is computed as the mean of the instantaneous graph idleness values across the period T, defined as:

$$\overline{I_G} = \frac{\sum_{t=0}^{T} I_G(t)}{|T|} \tag{6}$$

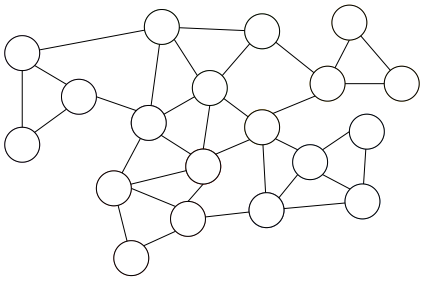
The worst idleness of graph G, IW_G, over period T represents the maximum instantaneous vertex idleness observed during period T.

$$IW_G = \max_{v \in V,\ 0 \le t \le T} i_v(t) \tag{7}$$

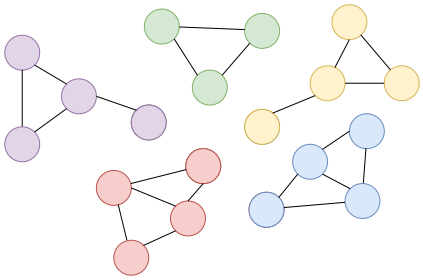
This paper aims to develop a multi-robot patrolling strategy which minimizes the average idleness and the worst idleness in this problem setting.

4 HMP Patrolling

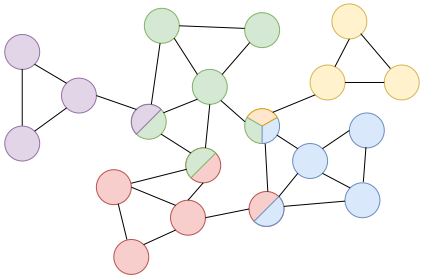
In this section we propose a new algorithm inspired by [8]. The fundamental idea is to create a synchronized system like [8], but instead of covering the area through cyclic paths, we modify it by adding meetings and constraining its next vertex method by only walking freely, if it does not immediately have to head to a meeting. The following subsections describe the different parts of the algorithm in more depth.



(a) Original graph: All vertices are connected to each other, but for readability only some edges are shown.



(b) Partitioned graph

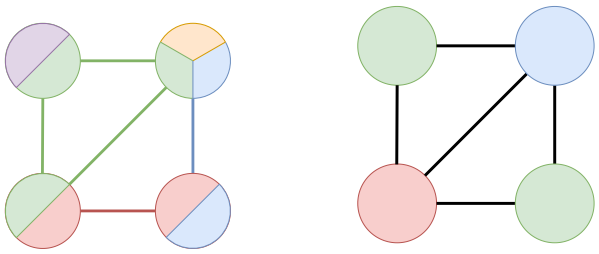


(c) Partitions are connected together with shared vertices (meeting points).

Fig. 1: Partitioning and Meeting Point Generation. These graphs are for illustration purposes. Real results might differ.

4.1 Meeting

Given that each robot is patrolling a distinct area of the environment, if a robot becomes faulty, its assigned area will be unpatrolled. Furthermore, due to the limited communication range, robots can not detect fault or coordinate with other active robots over long distance. To strengthen the connection between robots, certain $v \in V$ are designated as meeting point between several robots. These meeting points M serve as a spot to share information amongst the robots attending the meeting and detect faulty robots, as they would not be able to attend. A each meeting m the next two meeting times are agreed upon. For a meeting m, the next two meeting times are denoted as m_{next} and $m_{nextnext}$. Hence, m_{next} is the next meeting time for meeting m, and $m_{nextnext}$ is the next meeting time for meeting m after m_{next}. Additionally, $R_m \subseteq R$ denotes the set of robots that should attend the meeting m.



(a) Meeting graph (b) Colored meeting graph

Fig. 2: Graph transformations showing the simplified meeting structure (left) and original colored partitioning (right)

4.2 Initial Step

Before the patrolling can begin, each robot must know the set of assignments $\mathcal{A}$ and the mapping α in order to identify its area to patrol. Additionally, the robots must know when and where the initial meetings are held. This involves three steps.

4.2.1 Partition graph

To create $\mathcal{A}$, G is partitioned into $P = \{p_1, p_2, \cdots, p_{[R]}\}$, where each partition $p_i \subseteq V$. This means that the assignment a_i corresponds to p_i. Any partition algorithm can be used for this part. In this work, we chose the partitioning algorithm that was implemented by the other master group †. The partitioning algorithm is the K-way partitioning algorithm that uses spectral bisection algorithm, both defined in [16].

In this case, the K would be $|R|$. An example of this step is represented in the transition from Figure 1a to an graph Figure 1b that is partition.

4.2.2 Meeting point generation

A meeting m is held between robot r_i with a_i and r_j with a_j at a vertex selected from the intersection of their respective assignments a_i and a_j. That is, if the sets of vertices in a_i and a_j have a non-empty intersection, one of these shared vertices is chosen as the meeting point. In cases $a_i \cap a_j = \emptyset$, a meeting can still be arranged if the assignments are adjacent. Specifically, a meeting is possible if there exists an edge between any vertex assigned to robot i and any vertex assigned to robot j in the reduced graph G_{RNN}. The reduced graph $G_{RNN} = (V, E_{RNN})$ shares the same vertices v as G, but contains a subset of the edges E from G. E_{RNN} is constructed using the reverse nearest neighbors (RNN) method. That is, an undirected edge $(v_i, v_j) \in E_{RNN}$ exists if v_i is the nearest neighbor of v_j, or vice versa. Hence, given that $v_u \in a_i$ and $v_r \in a_j$, if there is exists an edge $(v_u, v_r) \in E_{RNN}$, either v_u or v_r becomes the meeting point between a_i and a_j. The decision for selecting a meeting point among v_u or v_r is based on a_i and a_j. If $|a_i] \geq |a_j|$, then v_u is selected as the meeting point. Furthermore, v_u is added to a_j. This strategy gives priority to the larger assignment, further reducing the load on that assignment. Figure 1c illustrates the result of the meeting point generation from derived Figure 1b.

4.2.3 Meeting time generation

The first initial meeting times m_{next} and $m_{nextnext}$ for $m \in M$ need to be assigned. A key consideration is to ensure that the meeting times are synchronized to prevent conflicts - no robot should attend multiple meetings simultaneously at different vertices. Furthermore, the interval between consecutive meetings must be carefully decided. If the interval is too short, the robot may not have sufficient time to patrol its assigned area effectively while also attending meetings. Conversely, if the interval is too long, the detection of faulty robots may be delayed, leading to inefficiencies in the global patrolling performance. Furthermore, the interval also depends on the robots' assignment. An assignment covering a larger area typically requires longer intervals between meetings to adequately patrol the area. On the other hand, if a robot is involved in many meetings, shorter intervals may be preferable. In such cases, the robot naturally patrols its area while moving between various meeting points, and by the time the robot returns to the same meeting point, it should have sufficiently covered its assigned area.

Hence, we have defined the minimum meeting interval Δt_i^{min} for a_i to be:

$$\Delta t_i^{min} = \left\lceil \frac{|a_i|}{|M_i|} \right\rceil \cdot D_i \tag{8}$$

This ensures that there is time to visit all vertices in a assignment between two consecutive meetings for m. M_{a_i} is the set of meetings in assignment a_i. D_i denotes the maximum shortest-path time between any two distinct vertices within a_i, and is given as follows:

$$D_i = \max_{\substack{u,v \in a_i \\ u \neq v}} c_{i,j} \tag{9}$$

Given the minimum meeting intervals, the globally meeting intervals Δt_{Global}^{min} is given as:

$$\Delta t_{Global}^{min} = \max_{i \in I} \Delta t_i^{min} \tag{10}$$

Given Δt_{Global}^{min}, we need to schedule such that no robot is attending two meetings simultaneously at different vertices. This scheduling constraint can be modeled as a vertex coloring problem, more specifically, finding the chromatic number of the meeting graph $G_M = (M, E_M)$ [31]. Each node in G_M represents $m \in M$. An undirected edge $(m_i, m_j) \in E_M$ exists if there is at least one robot that must attend both meetings m_i and m_j. Figure 2a illustrates the meeting graph derived from Figure 1c

By applying a vertex coloring algorithm to G_M, each $m \in M$ is assigned a color $c(m) \in \mathbb{N}$ such that adjacent nodes receive different colors. In this work, we have chosen the Welsh-Powell algorithm [17]. Each color index corresponds to a distinct time in the meeting cycle. A meeting cycle T_{cycle} is defined as the duration between two consecutive attendances at the same meeting point. By mapping the color indices to meeting times, m_{next} for m is given by:

$$m_{next} = \Delta t_{Global}^{min} \cdot c(m) \tag{11}$$

Figure 2b shows the resulting colored graph of the meeting graph in Figure 2a. Hence, the meeting time $m_{nextnext}$ for meeting m is:

$$m_{nextnext} = m_{next} + T_{cycle} \tag{12}$$

where

$$T_{cycle} = \max_{m \in M} m_{next} \tag{13}$$

That is, the highest m_{next} among the meetings defines the meeting cycle.

4.3 Patrolling mechanism

Given that the robot knows its assignment and the meeting it must attend, Algorithm 1 outlines the robot's core behavior.

While the robot is not participating in a meeting, it patrols its assigned area. To determine the vertex v to patrol in lines 2 and 9 in Algorithm 1, we apply the logic from HCR. Unlike pre-computed routes, this approach simplifies the annealing of vertices, as it eliminates the need to recompute routes each time the robot's assignments change. Additionally, since the interval between consecutive meetings at m may vary, relying on pre-computed routes requires recalculations after each meeting.

At the beginning of Algorithm 1 and after the robot has attended a meeting, the next meeting m is determined. As shown on lines 5-7 in Algorithm 1, the algorithm checks whether the robot can visit v before proceeding to m. If that is not possible, it should go straight to the m, otherwise it misses the meeting m. If possible, the robot moves to v, and when reaching v, it marks v as being visited, and the execution of Algorithm 1 continues.

Algorithm 1 Main Patrolling algorithm

```
1:  r ← this robot
2:  m ← NEXTMEETING()
3:  v ← NEXTSUGGESTEDVERTEX()
4:  loop
5:      v_t ← ESTIMATETRAVELTIME(r, v)
6:      m_t ← ESTIMATETRAVELTIME(v, m)
7:      if t_now + v_t + m_t < m_next then
8:          MOVETO(v)
9:          v ← NEXTSUGGESTEDVERTEX()
10:     else
11:         MOVETO(m)
12:         ATTENDMEETING(r, m)
13:         m ← NEXTMEETING()
14:     end if
15: end loop
```

4.4 Finding the Next Meeting

This section describes how `NextMeeting` works.

When deciding which meeting point to visit next, all meeting points are put into 1 of 3 priorities, with priority 1 being the highest and 3 the lowest:

Priority 1 m_{next} has passed[2], but $m_{nextnext}$ has not. This is top priority as this allows the robot to report that it is still alive and patrolling its assignments.

Priority 2 m_{next} has not passed and thus $m_{nextnext}$ has also not passed.

2. To determine whether or not the time has passed, travel time is also taking into account. That means if we cannot get there in time we say that it has passed.

Algorithm 2 AttendMeeting(r, m)

```
1:  R_expected ← R_m \ r
2:  BROADCASTATTENDING(r, m)
3:  R_attending ← RECEIVEATTENDING(r, m)
4:  while R_expected ≠ R_attending ∧ t_now < m_next do
5:      Wait for 1 tick
6:      BROADCASTATTENDING(r, m)
7:      R_attending ← RECEIVEATTENDING(r, m)
8:  end while
9:  NEGOTIATENEXTMEETING()
10: if R_expected ≠ R_attending then
11:     HANDLEMISSINGROBOTS(R_expected \ R_attending)
12: end if
13: DOANNEALING()
```

Priority 3 Both m_{next} and $m_{nextnext}$ has passed. This is the lowest priority as the robot's obligations with these meeting points cannot be met.

The next meeting point is determined by these priorities. The meeting point in each priority is chosen based on the minimum of the revelant time for that priority (either m_{next} or $m_{nextnext}$.) However, a lower-pritority meeting point is chosen when there is time to visit it before visiting a higher priority meeting point.

4.5 Agreeing on Next Meeting Times

This describes how `NegotiateNextMeeting` works.

Whenever robots meet at a meeting point, they have to decide the $m_{nextnext}$ meeting time. Each robot r proposes a time (pt_r) for the meeting point, which is the earliest time the robot can attend the meeting, taking all of its other meetings into consideration. They use the following equation, where M_r is the set of meeting points robot r has, and A_r is the assignments robot r patrols:

$$pt_r = \max_{m \in M_r} m_{nextnext} + \sum_{i \in A_r} \Delta t_i^{min} \tag{14}$$

R_m is the set of robots that attend meeting m. The next next meeting time is decided by choosing the maximum proposed meeting time:

$$m_{next} = m_{nextnext} \tag{15}$$

$$m_{nextnext} = \max_{r \in R_m} pt_r \tag{16}$$

4.5.1 Renegotiation

Sometimes both m_{next} and $m_{nextnext}$ need to be renegotiated. This happens when one or multiple robots participating in the meeting are taking over an assignment, or the robot is visiting a priority 3 meeting.

When a robot is taking over an assignment, it will have to tell its neighbors that the next meeting time must be rescheduled as it has to visit the assignment it is taking over, and it must have the time for this. It will also tell the robots at every meeting that the $m_{nextnext}$ meeting time is not good and must be renegotiated. Fault detection and handling will be described in the following section.

When a robot is visiting a priority 3 meeting it assumes that no other robot will be there at the meeting, as no usable meeting time information is available.

In both of these cases we need to figure out a new m_{next} and $m_{nextnext}$. This works the same as when doing a normal agreement, but instead of setting $m_{nextnext}$, m_{next} is set instead.

$$m_{next} = \max_{r \in R_m} pt_r \tag{17}$$

Setting $m_{nextnext}$ to $m_{next} + 10$ ensures a quick negotiation of a real meeting time. However, this approach is somewhat ad hoc and may warrant a more robust solution. This will be expanded on in section 7.

4.6 Fault-detection and handling

Whenever a robot does not show up the meeting point, another robot will begin to take over its assignment. Here the robot with the lowest id that is not currently taking over another robot's assignments is selected. If no such robot can be found, the missing robot is ignored. It will most likely be taken care of next meeting time.

When a robot begins to take over another robot's assignments, it first goes to all its current meeting points in its assignments and forces them to renegotate the meeting times, so there is time to visit the assignments that the robot is taking over. This is explained in subsubsection 4.5.1.

When hitting the meeting point that originally had the missing robot, it checks if another robot with a lower robot id has already taken over the assignments. If this is the case, the robot abandons taking over the assignments, and continues as normal. Otherwise, it will visit the new meeting points in the overtaking assignments.

4.7 Meeting Early

The process can be optimized by holding the meeting if every robot is there early. When a robot is at a meeting point early, it continuously broadcasts a message, saying it is at the meeting point and at what time it expects the other robots to be there at. If all robots arrive before the agreed-upon meeting time, the meeting is held early. Otherwise, the meeting is held at the planned meeting time.

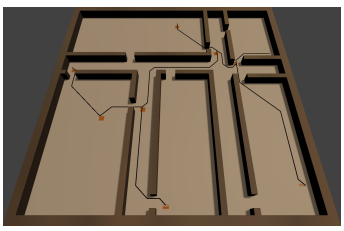
4.8 Annealing

It is inefficient when a robot has many assignments compared to its peers. This will increase the worst idleness.

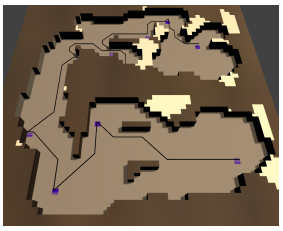
To mitigate this, when the robots meet, they compare how many assignments they have. If a robot has 2 or more assignments than another, it gives a assignment to that robot. This spreads out the assignments which will decrease the worst idleness. This uses the same mechanism as fault-detection and handling in how assignments are taken over.

4.9 Other HMP Variants

We have also developed three variations of the HMP algorithm, drawing inspiration from the strategies presented in [8] and [5]. These variants are HMP Random Takeover (HMP-RT), HMP Quasi-random Takeover (HMP-QT), and HMP Immediate Takeover (HMP-IT), all of which do not coordinate with their own neighbors before taking over an adjacent assignment, and always hold meetings at a fixed interval (Δt^{min}_{Global}). In HMP-RT, during a meeting, regardless of whether any robot is missing, each robot randomly selects one of the assignments associated with the meeting point to patrol, which may include the assignment it is currently in. In HMP-QT, if a robot is missing during a meeting, there is a 50% chance that the other robot will either take over the missing robot's assignment or stay in its current assignment. Finally, HMP-IT behaves as its name suggests: it immediately takes over a neighboring assignment if the corresponding robot is missing. These three variants are studied to evaluate the performance of more simplistic versions of HMP. Due to their fixed meeting interval, no priorities are required subsection 4.4, neither is `NegotiateNextMeeting` subsection 4.5, and robots switch between assignments based on a decision-strategy (random, quasi or immediate) instead of using annealing subsection 4.8.



(a) Screenshot of a 50x50 building map.



(b) Screenshot of a 50x50 cave map.

Fig. 3: Map types available in MAEPS.

In addition to the takeover variants a new variant with a wholly different approach was developed. This variant called HMP Single Meeting Point (HMP-SMP), modifies the meeting point placement algorithm, and how to determine when to visit the meeting. It works by only placing a single meeting point at the vertex which is the closest to all assignments. The meeting is then held at a fixed interval (Δt^{min}_{Global}). When a robot dies, its assignments are taken over by the robot with the least amount of assignments. As the fixed meeting interval does not change, the meeting point might be visited multiple times before all vertices in a robots assignment is visited.

5 Experiments

Here, we present the results of experiments carried out using the Multi-Agent Exploration & Patrolling Simulator (MAEPS), which has been extended to have all the features required for this paper, the implementation details can be found in Appendix A. These experiments evaluate the expected performance of the algorithms proposed in this study described in section 4. For comparison the experiments are also run with the following algorithms from other papers as a benchmark: RR, CR, HCR, ER, SC. Additionally, Partitioned Heuristic Conscientious Reactive (PHCR) was implemented as it act as the baseline for the proposed algorithms, described in section 4.

We use two types of maps, building maps and cave maps, as illustrated in Figure 3.

5.1 Communication

Communication between robots is line-of-sight. If a robot cannot directly see another robot, they cannot communicate.

However, if robot a can see robot b and b can see robot c, but c cannot see a, then a and c can still communicate. This is because the robots function as relays.

5.2 Repeated Executions and Metric Values

For each simulation configuration we have run 100 simulations using the seeds 1 to 100 for the map generation. For each of the metrics (average idleness and worst idleness), we computed their values for every execution. We then averaged these values across all seeds to represent the performance of the given simulation configuration.

5.3 Analysis Procedure

In the following subsections, we analyze the results from different perspectives.

Firstly, in subsubsection 5.4.1, experiments are conduct with different map sizes and robot count. This provides information about the ratio between the map size and the robot count, which is used to determine the standard parameters for the subsequent experiments. The objective is to identify an optimal number of robots per map size that provides a fair baseline for all compared algorithms. This ensures no algorithm is more favored. The optimal number of robots is evaluated qualitatively, where diminishing returns on worst idleness is too great.

Secondly, in subsubsection 5.4.2, we analyze the performance of the algorithms using the standard parameters showing their performance for each tick in a no faulty condition.

Lastly, in section subsubsection 5.4.3, the fault tolerance of the algorithm is assessed. This is done by killing robots at specific times. We have chosen 4 different scenarios in which different amount of robots are killed. The first scenario tests killing a single robot. The second scenario kills two robots with breathing room between faults. Third scenario kills two robots right after each other, testing how the algorithm handles multiple simultaneous robot failures. Last scenario kills three robots with breathing room. The exact parameters are as follows:

1) Kills a robot at tick 75000
2) Kills a robot at tick 75000, and 150000
3) Kills a robot at tick 150000 and 150150
4) Kills a robot at tick 75000, 150000, and 225000

5.4 Experiment Results

5.4.1 Experiment 1: Standard Parameters

As explained in Appendix C the performance of the simulator is highly dependent on map size. Therefore, choosing a map size that allows, in reasonable time, to run the experiments on a wide range of seeds, is paramount. Here we chosen a map size of 150 as bigger map sizes become computationally infeasible with the combinational complexity that our experiments have. For the robot count, we assessed that 8 robots marked the spot where diminishing returns became significant based on Figure 14 and Figure 15. Therefore, the standard parameters were determined to be a map size of 150x150 and a robot count of 8.

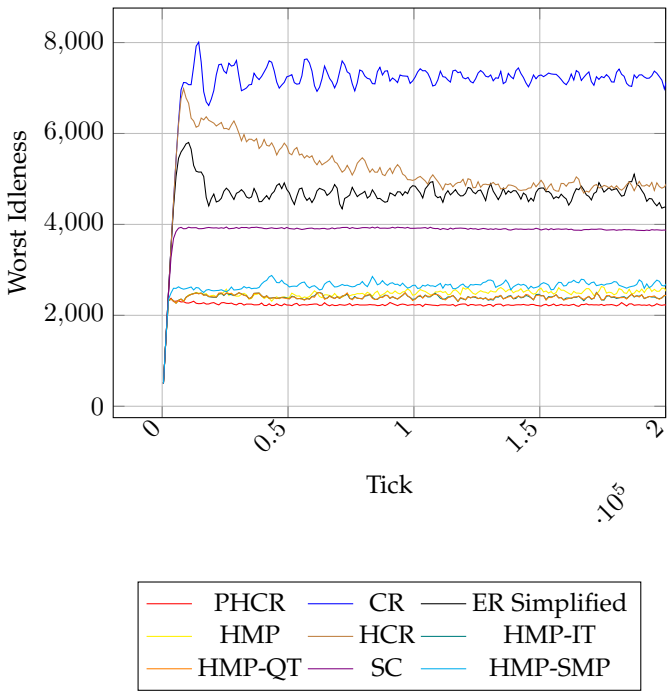


Fig. 4: Experiment 2: Worst Idleness over time on the Building Map

5.4.2 Experiment 2: Performance Comparison In No Fault Settings

Using the standard parameters found in experiment 1, this experiment evaluates the performance of the algorithms in a non-faulty setting. This experiment runs with 50 seeds for both the BuildingMap and CaveMap. Figure 4 shows the worst idleness of each algorithm on the building map. Some algorithms are excluded from the figure, such as Random reactive, due to a significantly higher worst idleness. Furthermore, Figure 4 displays results only up to 200,000 ticks, as all algorithms have reached a stable state by that point, and the worst idleness values no longer change significantly. This exclusion provides a clearer comparison view. A simplified version with the cave map is not shown here, as it is very similar to the building map. The full results of experiment 2 are provided in Figure 16, 17, 18 and 19 in Appendix D.

The experiment shows that in non-faulty conditions, that the HMP algorithm competes with and even beats the reactive algorithms. PHCR beats everyone as expected as it stays within its partition.

5.4.3 Experiment 3: Fault Tolerance

In this experiment, we evaluate how the algorithms perform in faulty conditions. Like the previous experiment, some algorithms has been omitted because their worst idleness made the figures unreadable. The full results of this experiment can be seen in figures 20 through 35. The HMP algorithm does seem to cope with faults, returning to baseline after a considerate amount of time. This is especially evident in Figure 5, where the algorithm cannot recover in time before another fault, but does seem to be going down after the 3rd fault. The worst idleness does spike considerably, but on the cave map (Figure 6) it does beat most if not all

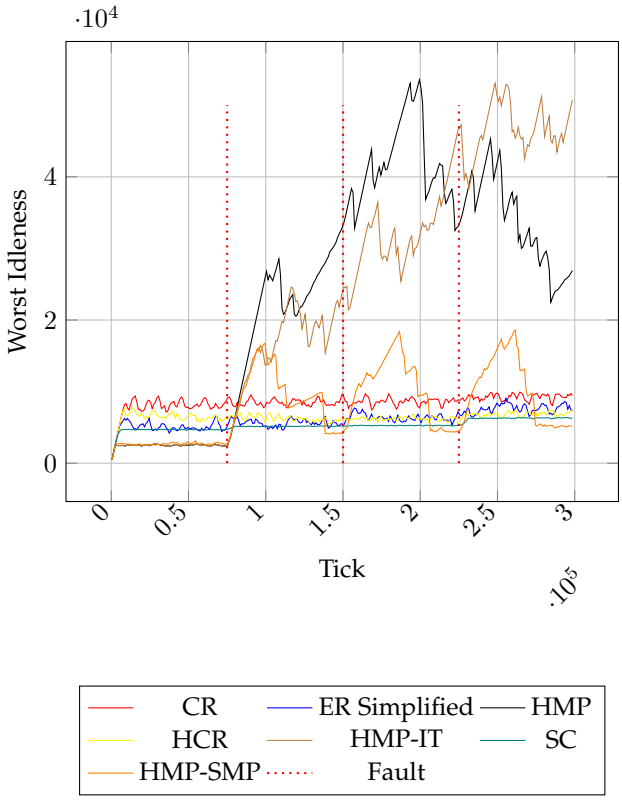


Fig. 5: Experiment 3: Worst Idleness over time on the Building Map on with fault at tick 75000, 150000, and 225000

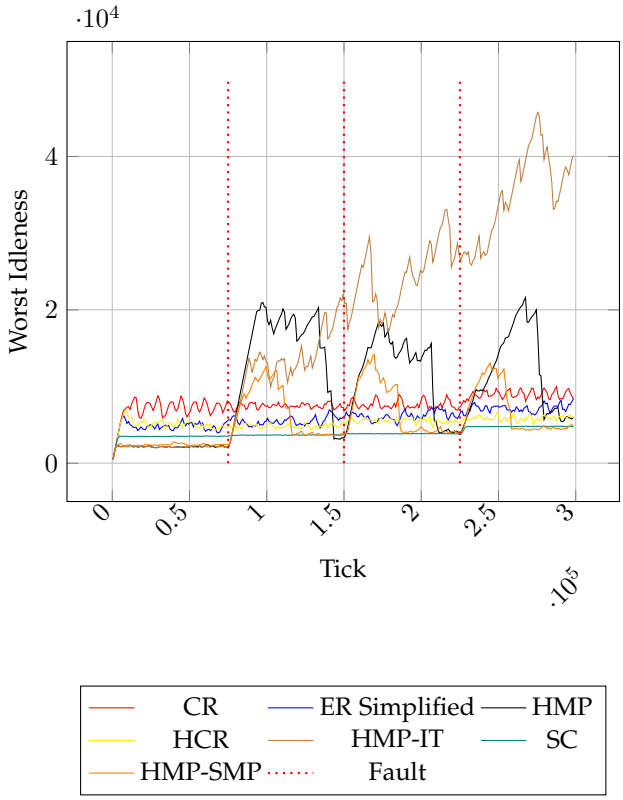


Fig. 6: Experiment 3: Worst Idleness over time on the Cave Map on with fault at tick 75000, 150000, and 225000

algorithms, when it has fully recovered from the fault. The HMP-IT performed the best of the "takeover"-variants, but still performed worse than all the other algorithms except RR.

6 Discussion

HMP does not work as expected. This is due to how they schedule meeting times in the case where they need both a new m_{next} and $m_{nextnext}$, which places both meeting times next to each other. The problem with this is that if a fault is detected when meeting at that meeting point, there is no time to go to the neighbors and tell them that it is going to take longer. This introduces a cascading failure where robots constantly are not showing up to the meetings they are supposed to show up at. These failures then exacerbates the problem as when the robots tell their neighbors they are going to take longer they have to agree on both a new m_{next} and $m_{nextnext}$, which would schedule them next to each other. For any robot its meetings must all be visited once before they can be visited again. That is if a robot has meeting points x, and y the following sequence is illegal: $xyyx$. A legal sequence would be $xyxy$.

The "takeover"-variants, which are simplified versions of HMP, performed poorly in faulty conditions, a lot worse than HMP and all other algorithms except RR. This proves the effectiveness of the priorities of meetings, `NegotiateNextMeeting` method and annealing described in section 4.

7 Conclusion & Future Work

We have developed a novel distributed, and fault-tolerant patrolling algorithm. The algorithm does effectively patrol in faulty-conditions, but it does need a considerable recovery time before the worst idleness goes back down to pre-fault idleness. The algorithms performs comparably to existing state-of-the-art distributed reactive and cyclic algorithms. However, there is still room for improvement.

As discussed in section 6 HMP does not work as expected due to it mangling the meeting order, by scheduling m_{next} and $m_{nextnext}$ right next to each other. To fix this, the algorithm must be able to schedule the meetings in such a way that the sequence always remains legal, or that it eventually will become legal.

HMP-SMP could be improved by recalculating partitions whenever a dead robot is detected. This would ensure that a robot does not have way more work than any other, which would improve the worst idleness. This also fixes the issue where the meeting point might be visited multiple times before a robot has visited all vertices in its assignment.

Acknowledgments

We would like to express our sincere gratitude to Michele Albano, Giovanni Bacci and Timothy Robert Merritt for supervising this project and providing invaluable insights and expertise that were essential to achieving the objectives of this paper. We are also thankful for the continuous feedback and support received throughout the course of the project.

We would like to thank the masters group consisting of Casper Nyvang Sørensen, Christian Ziegler Sejersen and

Jakob Meyer Olsen, for their contributions to the MAEPS project, some of which we have incorporated into our paper. To clarify the distinction between our work and theirs, we have marked any code originally developed by them with an †.

AI tools were utilized during the development of this work. Grammarly[14] was employed for proofreading, while ChatGPT[20] and GitHub Copilot[11] assisted with code writing and debugging.

The complete source code for this project is available on GitHub: https://github.com/cs-24-sw-9-xx/MAEPS

REFERENCES

[1] A Almeida et al. "Combining idleness and distance to design heuristic agents for the patrolling task". In: *II Brazilian Workshop in Games and Digital Entertainment*. 2003, pp. 33–40.

[2] Alessandro Almeida et al. "Recent Advances on Multi-agent Patrolling". In: *Advances in Artificial Intelligence – SBIA 2004*. Ed. by Ana L. C. Bazzan and Sofiane Labidi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 474–483. ISBN: 978-3-540-28645-5.

[3] *Burst Compiler Manual — Compilation*. Version Number: 1.8. Unity Technologies. URL: https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/compilation.html.

[4] Giorgio Cannata and Antonio Sgorbissa. "A minimalist algorithm for multirobot continuous coverage". In: *IEEE Transactions on Robotics* 27.2 (2011). Publisher: IEEE, pp. 297–312.

[5] Luis E. Caraballo et al. "Stochastic strategies for patrolling a terrain with a synchronized multi-robot system". In: *European Journal of Operational Research* 301.3 (2022), pp. 1099–1116. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2021.11.049. URL: https://www.sciencedirect.com/science/article/pii/S0377221721010006.

[6] Y. Chevaleyre. "Theoretical analysis of the multi-agent patrolling problem". In: *Proceedings. IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2004. (IAT 2004).* Sept. 2004, pp. 302–308. DOI: 10.1109/IAT.2004.1342959. URL: https://ieeexplore.ieee.org/document/1342959/?arnumber=1342959 (visited on 10/10/2024).

[7] Nicos Christofides. "Worst-case analysis of a new heuristic for the travelling salesman problem". In: *Operations Research Forum*. Vol. 3. Issue: 1. Springer, 2022, p. 20.

[8] Jose-Miguel Díaz-Báñez et al. "A General Framework for Synchronizing a Team of Robots Under Communication Constraints". In: *IEEE Transactions on Robotics* 33.3 (June 2017). Conference Name: IEEE Transactions on Robotics, pp. 748–755. ISSN: 1941-0468. DOI: 10.1109/TRO.2017.2676123. URL: https://ieeexplore.ieee.org/document/7888572/?arnumber=7888572 (visited on 10/15/2024).

[9] Shigeo Doi. "Proposal and evaluation of a pheromone-based algorithm for the patrolling problem in dynamic environments". In: *2013 IEEE Symposium on Swarm Intelligence (SIS)*. IEEE, 2013, pp. 48–55.

[10] Yehuda Elmaliach, Noa Agmon, and Gal A Kaminka. "Multi-robot area patrol under frequency constraints". In: *Annals of Mathematics and Artificial Intelligence* 57 (2009). Publisher: Springer, pp. 293–320. ISSN: 1012-2443.

[11] Github Inc. *GitHub Copilot*. en. 2025. URL: https://github.com/features/copilot (visited on 01/16/2025).

[12] Arnaud Glad et al. "Self-Organization of Patrolling-Ant Algorithms". In: *2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. 2009, pp. 61–70. DOI: 10.1109/SASO.2009.39.

[13] Arnaud Glad et al. "Theoretical study of ant-based algorithms for multi-agent patrolling". In: *ECAI 2008*. IOS press, 2008, pp. 626–630.

[14] Grammarly Inc. *Grammarly*. en-US. URL: https://www.grammarly.com/ (visited on 01/16/2025).

[15] Li Huang et al. "A survey of multi-robot regular and adversarial patrolling". In: *IEEE/CAA Journal of Automatica Sinica* 6.4 (July 2019), pp. 894–903. ISSN: 2329-9274. DOI: 10.1109/JAS.2019.1911537.

[16] Pavla Kabelıkova. "Graph Partitioning Using Spectral Methods". en. PhD thesis. Ostrava, Czech Republic: Technical University of Ostrava, May 2006.

[17] Velin Kralev and Radoslava Kraleva. "A comparative analysis between two heuristic algorithms for the graph vertex coloring problem". eng. In: *International journal of electrical and computer engineering (Malacca, Malacca)* 13.3 (2023), pp. 2981–2989. ISSN: 2088-8708.

[18] Aydano Machado et al. "Multi-agent movement coordination in patrolling". In: *Proceedings of the 3rd International Conference on Computer and Game*. 2002, pp. 155–170.

[19] Aydano Machado et al. "Multi-agent Patrolling: An Empirical Analysis of Alternative Architectures". In: *Multi-Agent-Based Simulation II*. Ed. by Jaime Simão Sichman, Françcois Bousquet, and Paul Davidsson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 155–170. ISBN: 978-3-540-36483-2.

[20] OpenAI. *ChatGPT*. en-US. URL: https://openai.com/index/chatgpt/ (visited on 01/16/2025).

[21] Mehdi Othmani-Guibourg, Amal El Fallah-Seghrouchni, and Jean-Loup Farges. "LSTM Path-Maker: a new LSTM-based strategy for the multi-agent patrolling". eng. In: *Proceedings of the 52nd Hawaii International Conference on System Sciences*, pp. 616–625.

[22] Mehdi Othmani-Guibourg, Amal El Fallah-Seghrouchni, and Jean-Loup Farges. "Path Generation with LSTM Recurrent Neural Networks in the context of the Multi-agent Patrolling". In: *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2018, pp. 430–437.

[23] F. Pasqualetti, J. W. Durham, and F. Bullo. "Cooperative Patrolling via Weighted Tours: Performance Analysis and Distributed Algorithms". eng. In: *IEEE trans-*

actions on robotics 28.5 (2012). Place: PISCATAWAY Publisher: IEEE, pp. 1181–1188. ISSN: 1552-3098.

[24] David Portugal, Micael S Couceiro, and Rui P Rocha. "Applying Bayesian learning to multi-robot patrol". In: *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. IEEE, 2013, pp. 1–6.

[25] David Portugal, Rui Rocha, and Luis M. Camarinha-Matos. "A Survey on Multi-robot Patrolling Algorithms". eng. In: *IFIP Advances in Information and Communication Technology*. Vol. 349. IFIP Advances in Information and Communication Technology. ISSN: 1868-4238. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 139–146. ISBN: 3-642-19169-X.

[26] David Portugal and Rui P Rocha. "Cooperative multi-robot patrol with Bayesian learning". In: *Autonomous Robots* 40.5 (2016). Publisher: Springer, pp. 929–953.

[27] David Portugal and Rui P. Rocha. "Distributed multi-robot patrol: A scalable and fault-tolerant framework". In: *Robotics and Autonomous Systems* 61.12 (Dec. 2013), pp. 1572–1587. ISSN: 0921-8890. DOI: 10.1016/j.robot.2013.06.011. URL: https://www.sciencedirect.com/science/article/pii/S0921889013001206.

[28] David Portugal and Rui P. Rocha. "Multi-robot patrolling algorithms: examining performance and scalability". eng. In: *Advanced robotics* 27.5 (2013). Publisher: Routledge, pp. 325–336. ISSN: 0169-1864.

[29] Hugo Santana et al. "Multi-agent patrolling with reinforcement learning". In: *Autonomous Agents and Multiagent Systems, International Joint Conference on*. Vol. 4. IEEE Computer Society, 2004, pp. 1122–1129.

[30] *Strato - High-Performance Computing at Aalborg University*. URL: https://hpc.aau.dk/strato/ (visited on 06/06/2025).

[31] Arthur T. White. "Chapter 8 - Map-Coloring Problems". In: *Graphs of Groups on Surfaces*. Ed. by Arthur T. White. Vol. 188. North-Holland Mathematics Studies. ISSN: 0304-0208. North-Holland, 2001, pp. 89–106. DOI: https://doi.org/10.1016/S0304-0208(01)80009-8. URL: https://www.sciencedirect.com/science/article/pii/S0304020801800098.

[32] Chuanbo Yan and Tao Zhang. "Multi-robot patrol: A distributed algorithm based on expected idleness". English. In: *International Journal of Advanced Robotic Systems* 13.6 (Dec. 2016). _eprint: https://doi.org/10.1177/1729881416663666, p. 1729881416663666. ISSN: 17298806. DOI: 10.1177/1729881416663666. URL: https://doi.org/10.1177/1729881416663666.

[33] Vladimir Yanovski, Israel A. Wagner, and Alfred M. Bruckstein. "A distributed ant algorithm for efficiently patrolling a network". eng. In: *Algorithmica* 37.3 (2003). Place: NEW YORK Publisher: Springer Nature, pp. 165–186. ISSN: 0178-4617.

APPENDIX A
IMPLEMENTATION

A.1 Implement new Line-of-sight Algorithm

A new line-of-sight algorithm was implemented for use in the waypoint (vertex) generation process. It has been optimized to utilize a bitmap and is designed to run as a Burst job using the Burst Compiler [3], which compiles directly to optimized machine code. This new algorithm also enables multithreading. Additionally, the algorithm has been extended to support a maximum visibility distance.

A.2 Implement Patrolling Algorithms

Multiple patrolling algorithms have been implemented to be used for comparison with the algorithms we developed. The algorithms implemented from other papers are: HCR, PHCR, SC, ER. Two versions of SC have been implemented, one based on an exact TSP solver and another using christofides, [7].

A.3 Implement HMPPatrolling and its variants

The different HMP-based algorithms described in section 4 have been implemented. The implementation can be found at https://github.com/cs-24-sw-9-xx/MAEPS/tree/main/Assets/Scripts/Algorithms/Patrolling/HMPPatrollingAlgorithms

A.4 Make PatrollingAlgorithm Component based

The `PatrollingAlgorithm` class has been refactored to a component-based architecture, allowing for easily extending the base algorithm using components. One such component is the `CollisionRecoveryComponent`, which as the name implies recovers the robot from a collision state. The component structure has been refactored to allow components to consist of other components. This was not just a code refactoring, but also an architecture refactoring. Additionally, using C# generators, wait conditions, such as wait for n logic ticks, or wait for a specific robot state, can be used to wait for the condition before continuing execution. This greatly reduces or completely eliminates the need for building bespoke state-machines. Implementing this was also a prerequisite for allowing the component-based architecture.

A.5 User Interface

The user interface has gained new features and a few old features have been removed. The current version of the visualization modes can be seen in Figure 7.

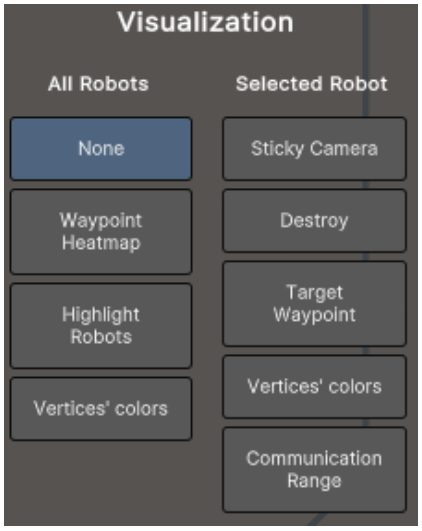


Fig. 7: Screenshot of the visualization mode button panel in MAEPS.

Two buttons named “Vertices’ colors” have been added. The one on the “All Robots” side highlights which partitions a vertex is assigned to, additionally robots are colored corresponding to which partitions they belong to. This was implemented to assist in debugging the partitioning and patrolling algorithms that use partitioning and proved very useful, a screenshot demonstrating this feature can be seen in Figure 8.

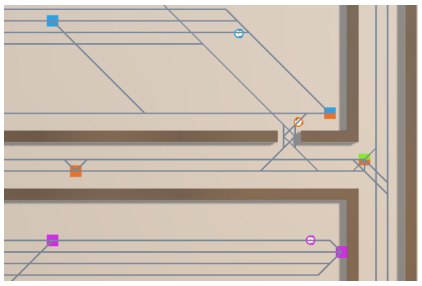


Fig. 8: Screenshot of the all robot vertices' colors visualization mode in MAEPS. Vertices with multiple colors belong to multiple partitions.

Three buttons have been added on the "Selected Robot" side. The "Destroy" button destroys a robot, simulating robot failure. The "Communication Range" highlights a robots communication range, as seen in Figure 9.

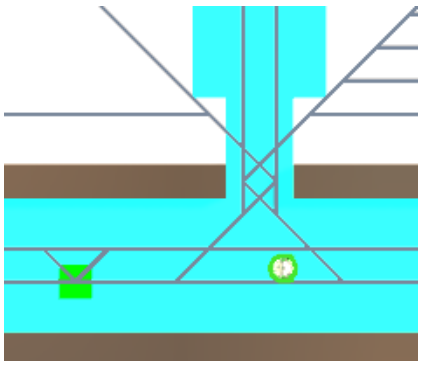


Fig. 9: Screenshot of the single robot communication range visualization mode in MAEPS. Here LOS is used as communication range algorithm.

The robot coverage visualization mode has been removed, since it doesn't make much sense for patrolling, since the vertices are guaranteed to have LOS of the entire map. So, the waypoint heatmap is the patrolling equivalent of the coverage heatmap.

Due to the logging of csv data for experiments and the addition of the data-processor, the gui tool to build a graph during runtime has become obsolete, and has been removed since it heavily impacted the runtime performance.

A.6 Add VirtualStigmergy Component

A `VirtualStigmergy` component has been constructed for key-value storage and sharing. It is possible to specify a marker type which enables differentiating stigmergies with the same key and value types. The `VirtualStigmergy` only accept types that are unable to smuggle information. If the virtual stigmergy was able to pass references to for instance other robots, then they could share information through state directly without going through the communication manager and virtual stigmergy.

A.7 Performance Optimizations and Refactoring

A lot of refactoring and runtime optimization has been done since last semester. The experiments would not have been practically possible without these optimizations.

- The project's version of unity has been updated to 6000.0.37f1.
- Map configs have been refactored to have a base class. Previously, `CaveMapConfig` and `BuildingMapConfig` had nothing in common, making them difficult to work with regarding making experiments.
- The tests have been sped up, as they were taking a lot of time (multiple minutes) to complete.
- We enabled il2cpp for builds to make optimized builds for running the experiments.
- We parallelized the optimized builds and the test steps in the GitHub Actions.
- We refactored the `SimulationMapBuilder`, such that a map builder could easily be substituted, which allowed us to make a special map builder which could construct a map from a string or a text file, which proved very useful for speeding up tests.
- GitHub Actions have been set up to make a dedicated server build and upload this as an artifact. In the code all GUI-related code was marked, such that the project can be compiled without the GUI-components, which is required to make a headless build.

- The code stripping level was set to maximum. This required us to refactor any code that used reflection, since reflection is incompatible with this level.
- The waypoint generator has been changed to penalize waypoints close to walls, as these could sometimes cause collisions and unusual robot behavior. A cache for the waypoint generation has also been made.
- Where possible, MAEPS now uses bitmaps instead of `Bool[,]`, which use less memory and are faster.
- Meshes were used in map generation and visualization modes. These meshes have to be disposed properly in order to free this memory, otherwise there is a memory leak. Destroying the Unity objects that owned these meshes did not properly dispose of them. This memory leak caused a lot of trouble during development and especially during the running of experiments until we finally found and fixed this leak.
- We fixed the `Line2D` intersection which is used in `RayTracingMap`. Previously, it didn't work with 90-degree angles, and the previous group added small floating-point values as a workaround.
- Where possible memory allocation on the heap has been avoided or minimized. For instance, `ArrayPool<T>` is used where arrays are frequently created and destroyed. This resulted in significantly less memory pressure on the garbage collector.
- The calculation of the `Adjacency Matrix` has been optimized by using symmetry, reducing the time complexity from N^2 to $0.5 \cdot N^2$, with N being the number of robots.
- The calculation of the `Distance Matrix`, which consists of the distances between all vertex pairs, has been optimized by running breadth-first search in parallel using multithreading.
- Benchmarks have been added for both the building and the cave map generators, which has been used to verify optimizations. No benchmark results for the map generations are included in this paper, but note that a benchmark for the start-up time of a scenario can be found in Appendix C which map generation is part of.
- A new waypoint connector has been added called `AllConnectedWaypointsConnector`, which as the name implies, connects all vertices to all other vertices, making it a complete graph.

A.8 A* Improvements

A custom A* implementation has been made, since it is faster than converting the map to the type required by the library. The A* algorithm is slightly penalized for tiles close to walls, since robots can sometimes collide with walls. Also, direction changes are slightly penalized, resulting in fewer turns, since the robots come to a full stop just to turn slightly. This avoids "zigzag" paths.

A.9 Work for Running Experiments

Our experiments were run on the AAU Cloud [30], and a guide has been created explaining to how to run an experiment on a virtual machine and how to run it on AAU Strato. The guide can be found here: Guide_VirtualMachine.md.

Since Unity API methods cannot be used by different threads, we created a workaround for parallelization by spawning multiple instances of the simulator using a bash script. The run script takes multiple command-line arguments, such as which experiment to run and the range of instances to run, allowing an experiment to be split across different servers.

A script has been created for downloading an optimized build generated by GitHub Actions and stored as a GitHub artifact. This script allows specifying the workflow ID from which to download the build. This enables downloading the optimized build from branches other than main.

After encountering bugs that caused simulations to get stuck and therefore never finish, blocking any remaining simulations from being run, a `MaxLogicTicks` parameter was added to the simulator. This parameter makes the `SimulationManager` abort any scenario that takes too long and continue with the next scenario. Additionally, the run script pipes the output from each instance into a combined `output.log` file for debugging. Each log entry contains the current time and instance ID. After completing the experiment, the script greps the output.log to check for scenarios that did not finish before reaching `MaxLogicTicks`, as well as any exceptions.

A.10 Data-Logging

Extensive logging has been set up. The existing csv file writer used a library, but this has been removed and replaced with the custom csv writer, since the library used reflection was cannot be used in combination with code stripping. The experiment data is structured into "data/experiment/scenario-name/data-here", where the raw data for each vertex is put into a subfolder since there are typically many of them. The csv data logged for a patrolling scenario is for each tick the:

- ReceivedMessageCount
- SentMessageCount
- GraphIdleness
- WorstGraphIdleness
- TotalDistanceTraveled
- CompletedCycles, a cycle being the minimum amount of times any vertex has been visited.
- AverageGraphIdleness
- NumberOfRobots, the current amount of robots, this changes if a robot is destroyed through the gui-button or fault-injection.

A.11 Data-Processor

A `data-processor` has been created for handling the data of the experiments. This involves combining data across parameters, such as calculating the average result across multiple seeds. The `data-processor` also support the plotting of data as boxplots and line graph.

APPENDIX B
PROJECT MANAGEMENT

Time Management

Our previous semester's time plan, as can be seen in Figure 10, accurately estimated the literature review, problem setting, and initial implementation phases. However, as can be seen in Figure 11 it significantly underestimated the time required for collaborative improvements, primarily due to more features being needed by both groups and extensive time spent optimizing the performance of MAEPS, which can be read about in Appendix A.

Testing and evaluation also consumed far more time than we had planned. A significant portion of the necessary functionality for this phase was shared between both groups, and has therefore also been marked as collaborative improvements. This included, for instance, experiment data logging, the experiment runner (with scripts and command-line argument capabilities for experiment selection), the data processor (for aggregating across seeds and plotting graphs), and a guide for executing experiments on AAU starto, [30].

The algorithm design phase likewise exceeded its allocated time. This was partly due to the difficulty in transforming our abstract idea into a concrete computational problem. Additionally, our initial concept proved unable to handle transient failures. After brainstorming ways to make our idea fully fault tolerant, we made variations of the base algorithm using these new ideas for fault handling strategies, as described in section 4, which is the final reason why this step spanned such a long period. To streamline the process, we opted to implement the algorithm concurrently with its design, a strategy that effectively helped us discern what worked and what didn't.

Apart from initial literature notes and the early conceptualization of our idea (and its later transformation into a computational problem), very little thesis writing occurred until the project's later stages. This delay was a direct result of the aforementioned stages taking longer than anticipated, coupled with difficulties encountered during experiment execution—specifically, issues with runtime performance and a few bugs. These challenges effectively pushed back the collaborative improvement and test/evaluation phases, consequently delaying the thesis writing.

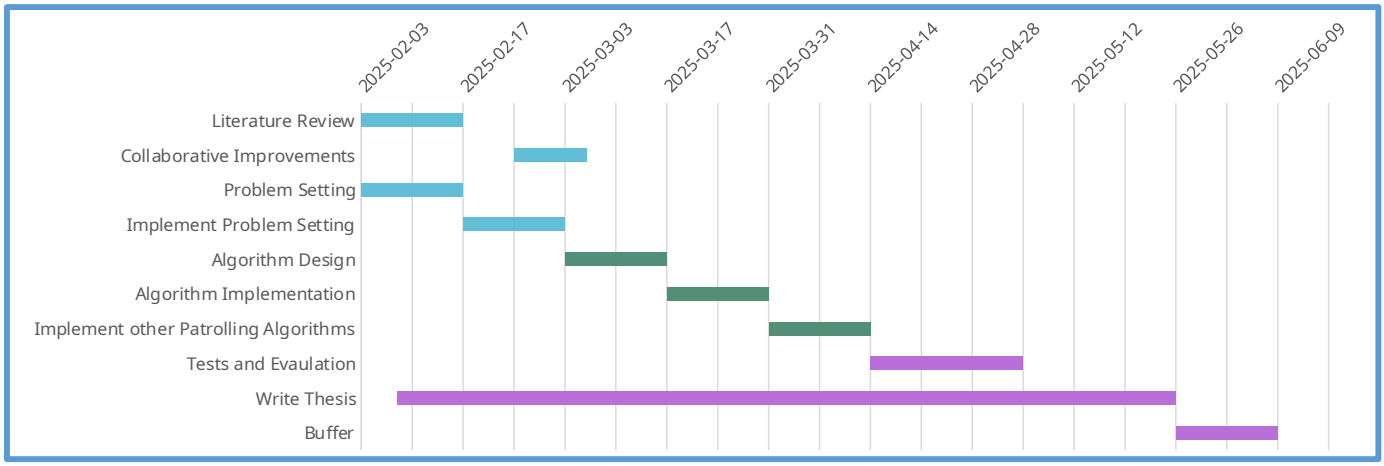


Fig. 10: Gantt Diagram for the Master Thesis A. (plan)

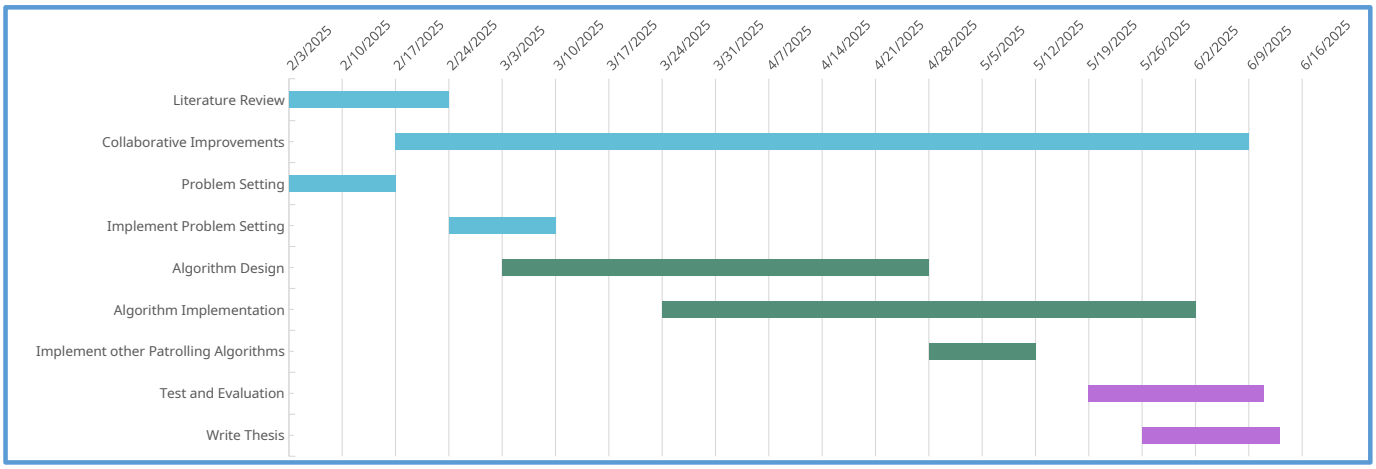


Fig. 11: Gantt Diagram for the Master Thesis A. (actual time spent)

Task Management Framework

We adopted a Kanban board as our task management framework to provide a clear visual representation of our project's workflow, aiding in task management, progress tracking and helping us see what the other team members are currently working on. As illustrated in Figure 12, our board is divided into the following columns:

- Backlog: This column captures all tasks that have been identified but not yet scheduled for implementation.
- To Do: Tasks that have been selected for implementation but not yet started are moved to this column. This serves as the queue of work to be done in the current week.
- In Progress: Tasks that are currently being worked on are placed in this column. It provides visibility into what team members are actively engaged in.
- Done: Tasks that have been completed and reviewed successfully are moved to this column.

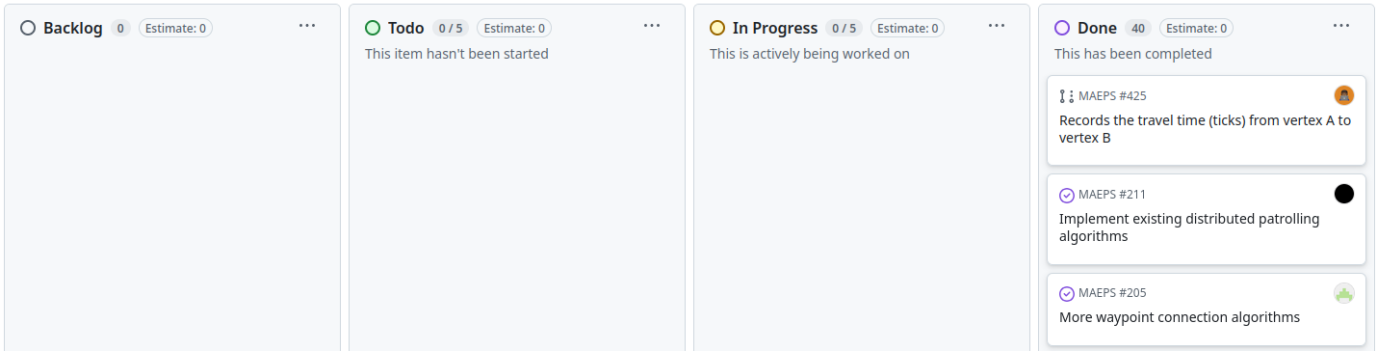


Fig. 12: Picture of the Kanban board.

APPENDIX C
MAEPS STARTUP BENCHMARK

An experiment was conducted to explore the performance of the startup process of the MAEPS simulator for a patrolling scenario. The startup mainly consists of:

- Generating the tile-map.
- Generating waypoint/vertices.
- Optionally, partitioning.
- Connecting waypoint/vertices.

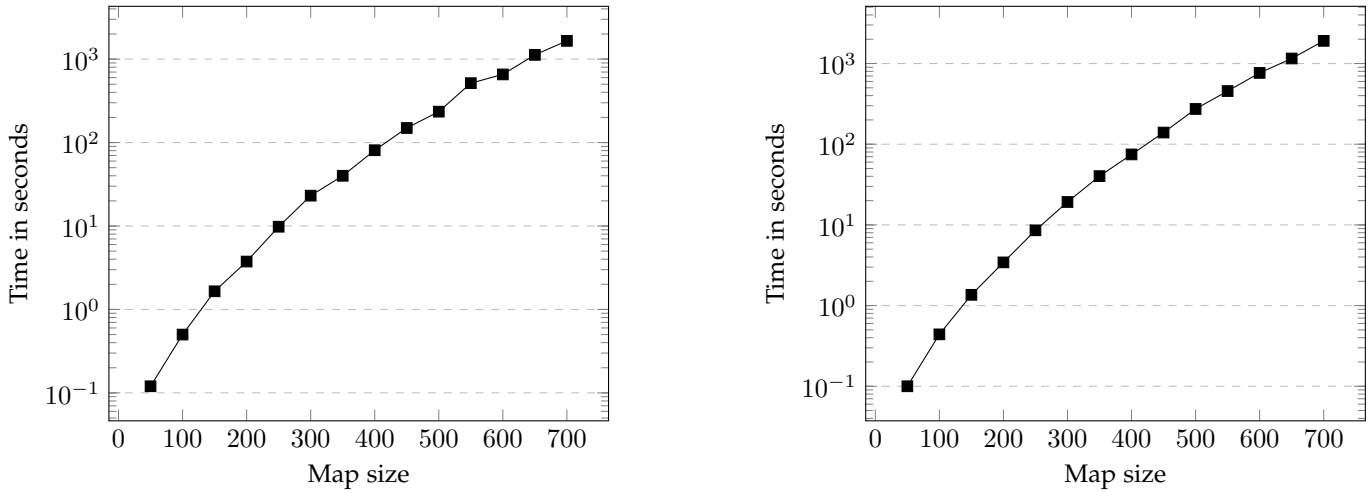
For these results, the `ReverseNearestNeighborGenerator` was used, which does not do partitioning. The experiment was carried out using this version of MAEPS: https://github.com/cs-24-sw-9-xx/MAEPS/commit/9d6cd8662e665cf5cb6c4eb1161b210572200953. It was run on a 16 vCPU machine on AAU strato [30].

Map size	Time in seconds
50	0.12
100	0.5
150	1.65
200	3.75
250	9.8
300	23.1
350	39.99
400	80.9
450	149.5
500	234.6
550	515.1
600	655.9
650	1124.8
700	1655.8

TABLE 2: Start-up time for a building map scenario.

Map size	Time in seconds
50	0.1
100	0.44
150	1.36
200	3.43
250	8.57
300	19.23
350	40.22
400	74.75
450	139.5
500	273.4
550	455.8
600	765.2
650	1154.7
700	1910.2

TABLE 3: Start-up time for a cave map scenario.



(a) Building map

(b) Cave map

Fig. 13: The start-up time of a scenario for the two map types.

APPENDIX D
EXPERIMENT DATA AND PLOTS

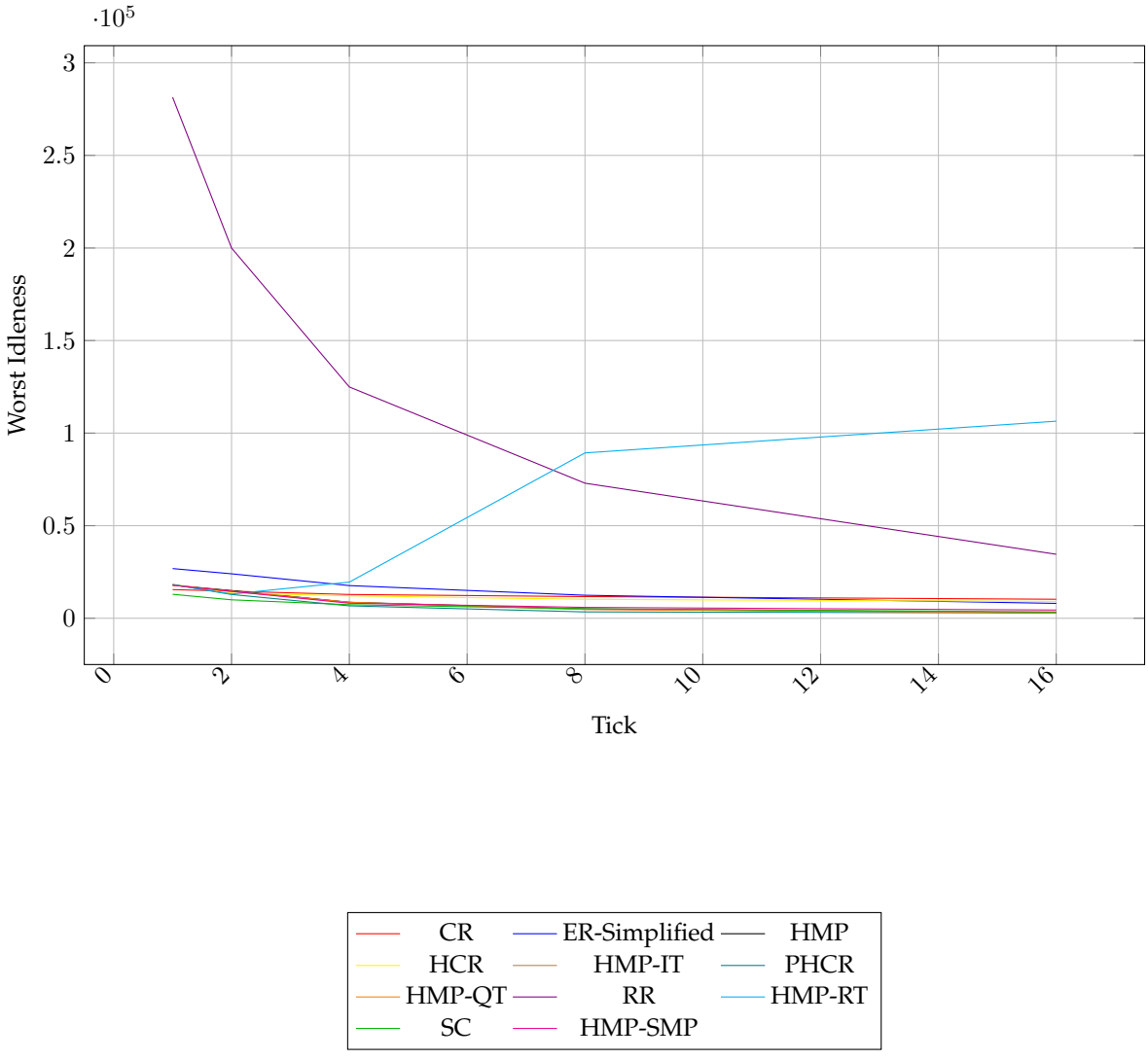


Fig. 14: Experiment 1: Building Map - Map size 150

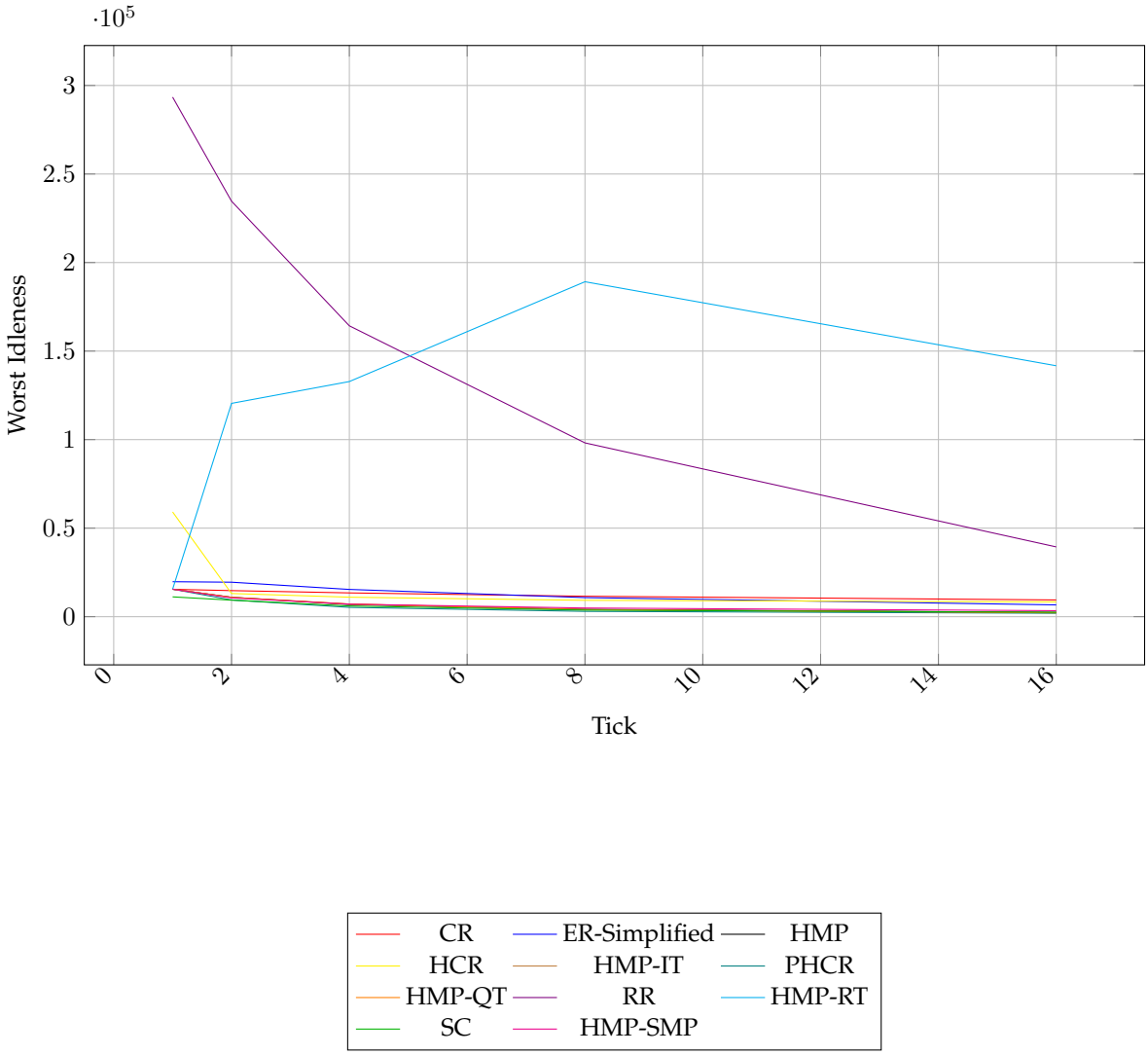


Fig. 15: Experiment 1: Cave Map - Map size 150

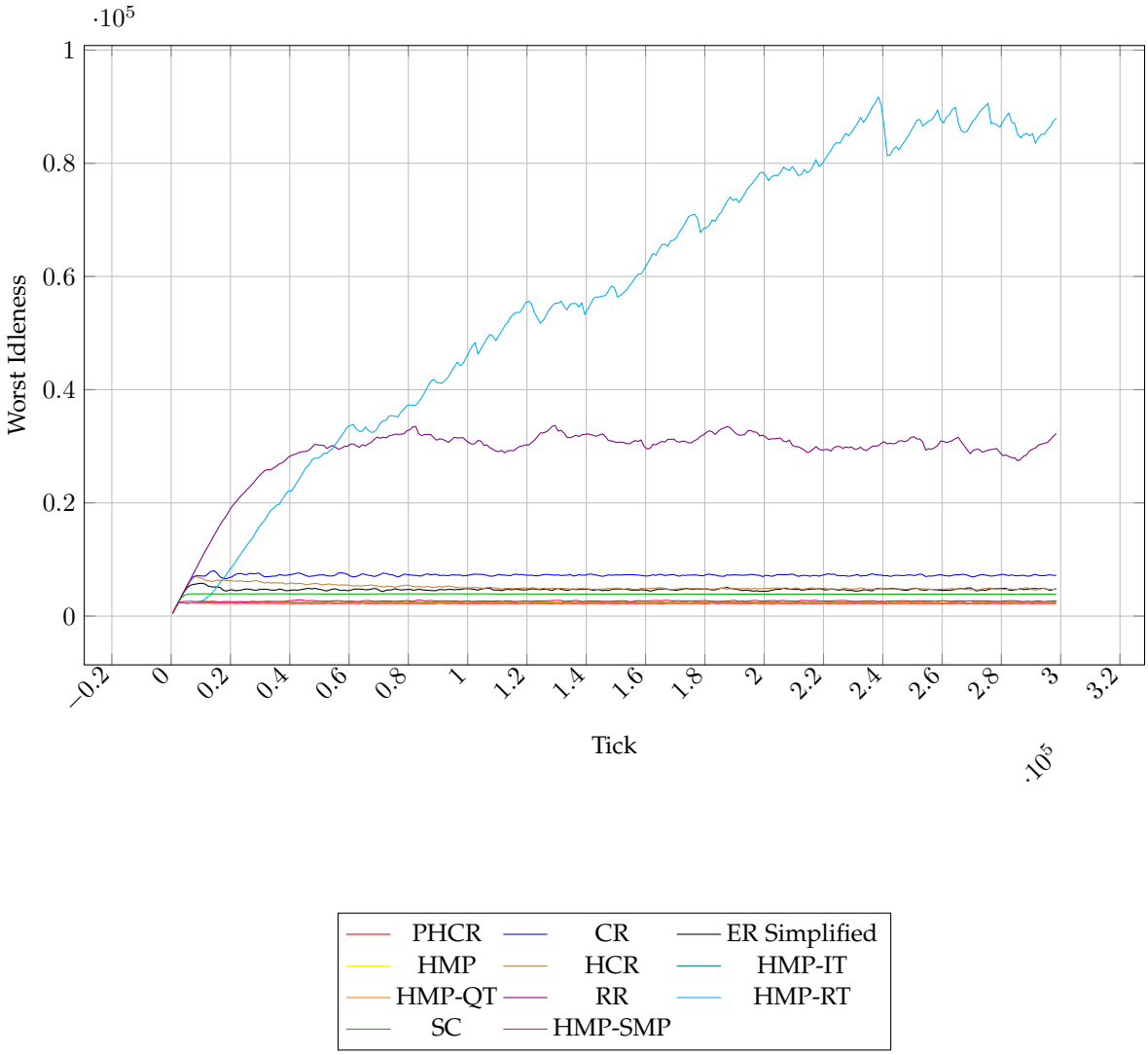


Fig. 16: Experiment 2: Worst Idleness over time on the Building Map

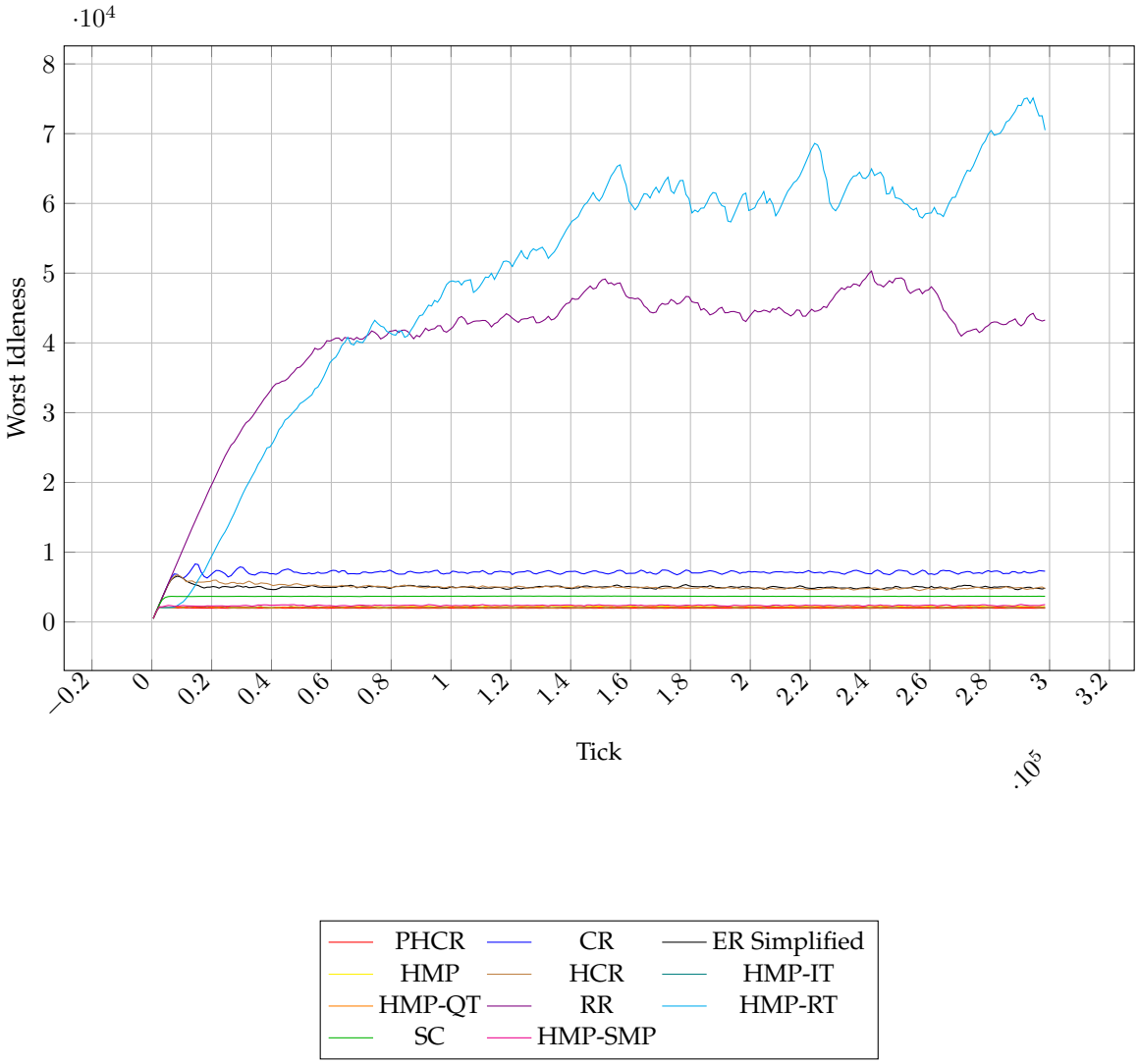


Fig. 17: Experiment 2: Worst Idleness over time on the Cave Map

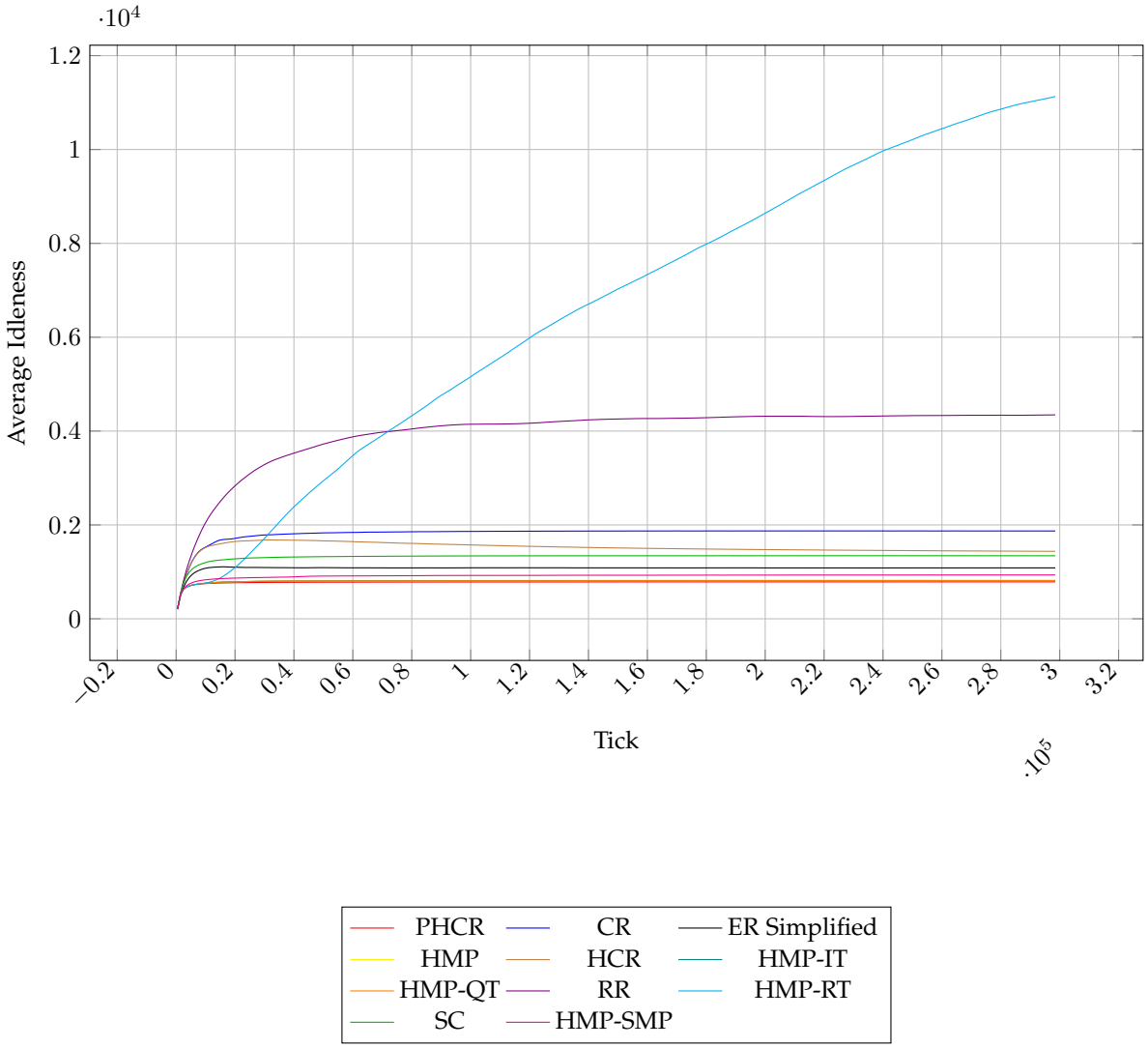


Fig. 18: Experiment 2: Average Idleness over time on the Building Map

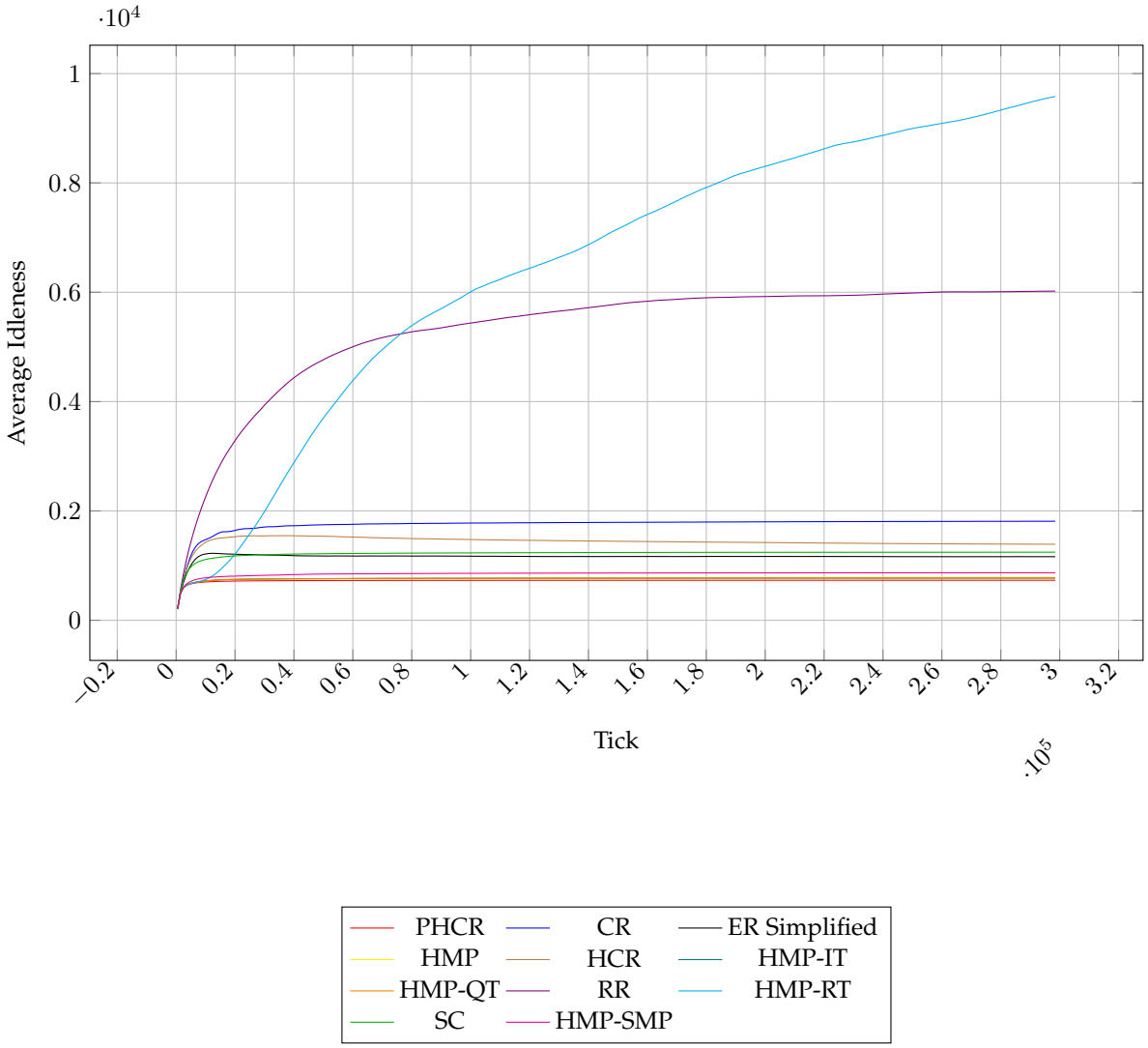


Fig. 19: Experiment 2: Average Idleness over time on the Cave Map

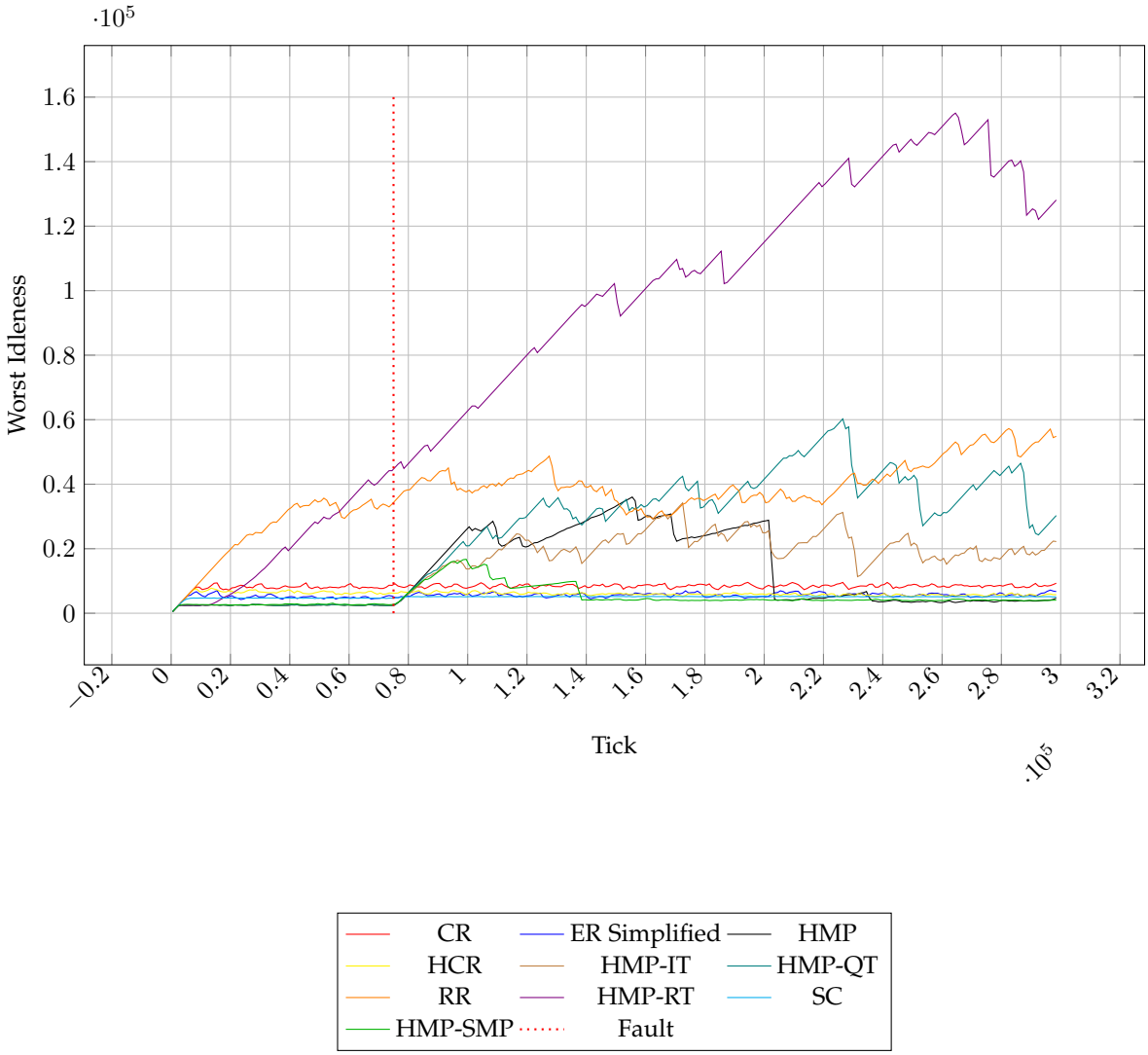


Fig. 20: Experiment 3: Worst Idleness over time on the Building Map on with fault at tick 75000

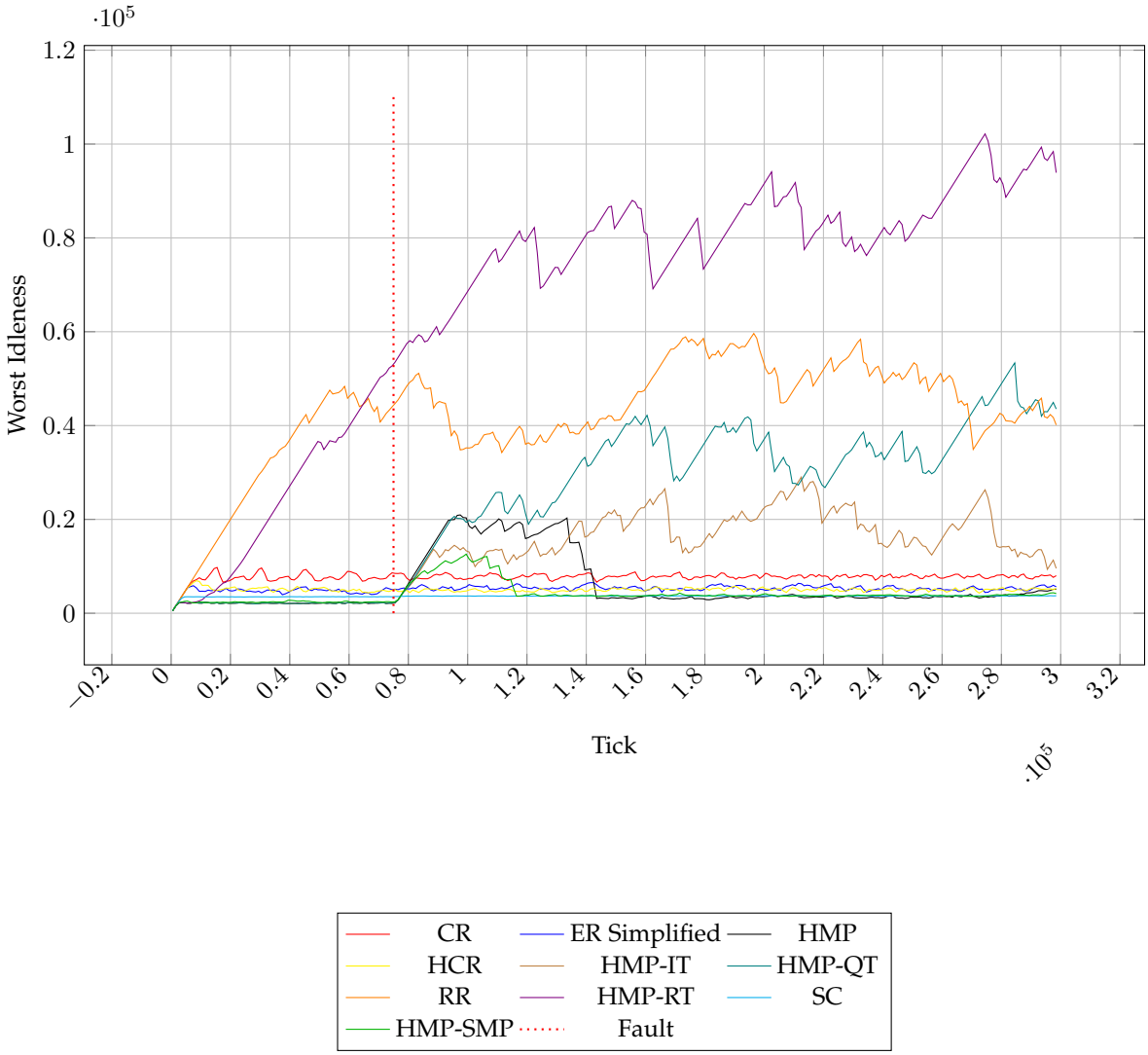


Fig. 21: Experiment 3: Worst Idleness over time on the Cave Map on with fault at tick 75000

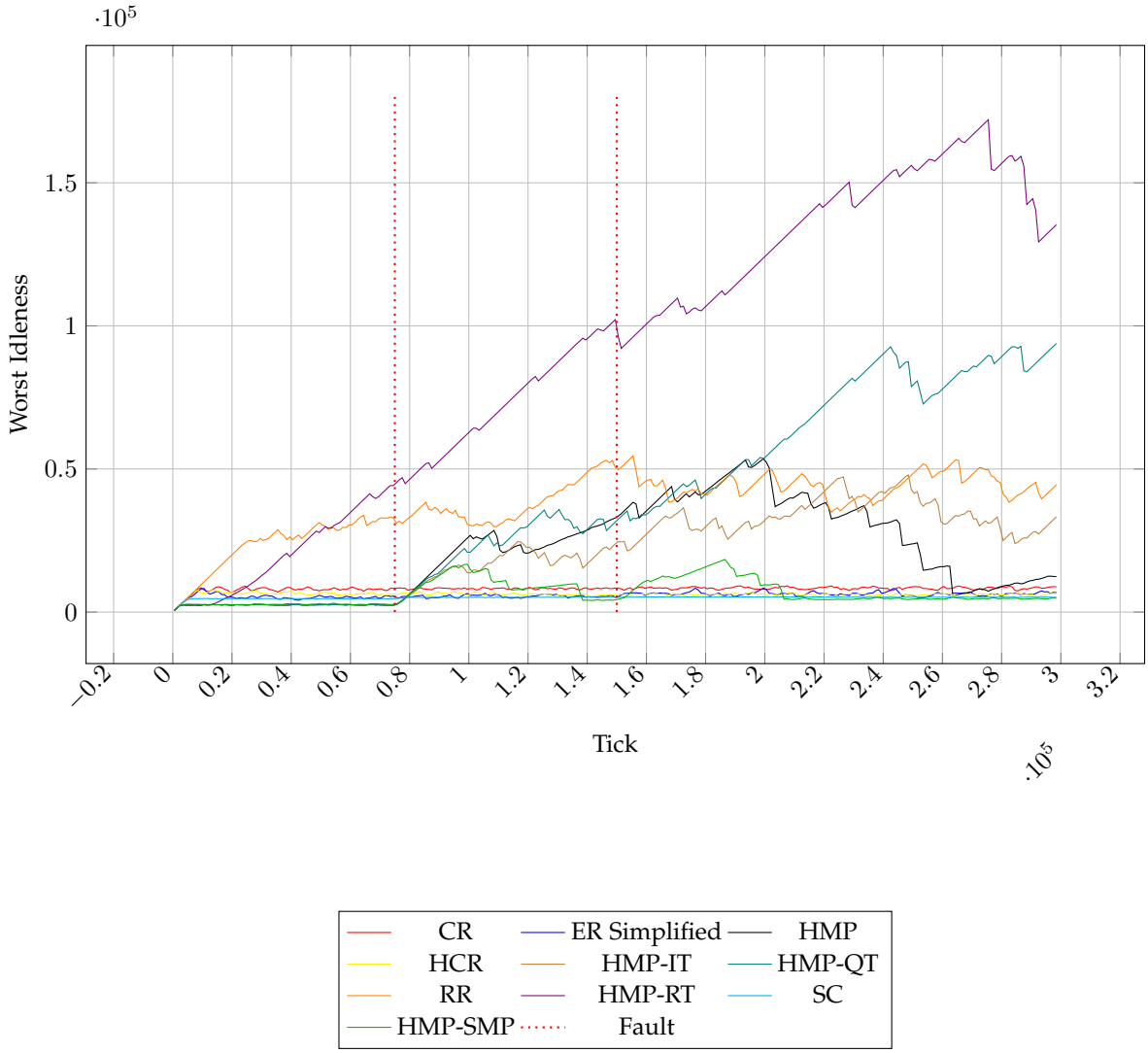


Fig. 22: Experiment 3: Worst Idleness over time on the Building Map on with fault at tick 75000,150000

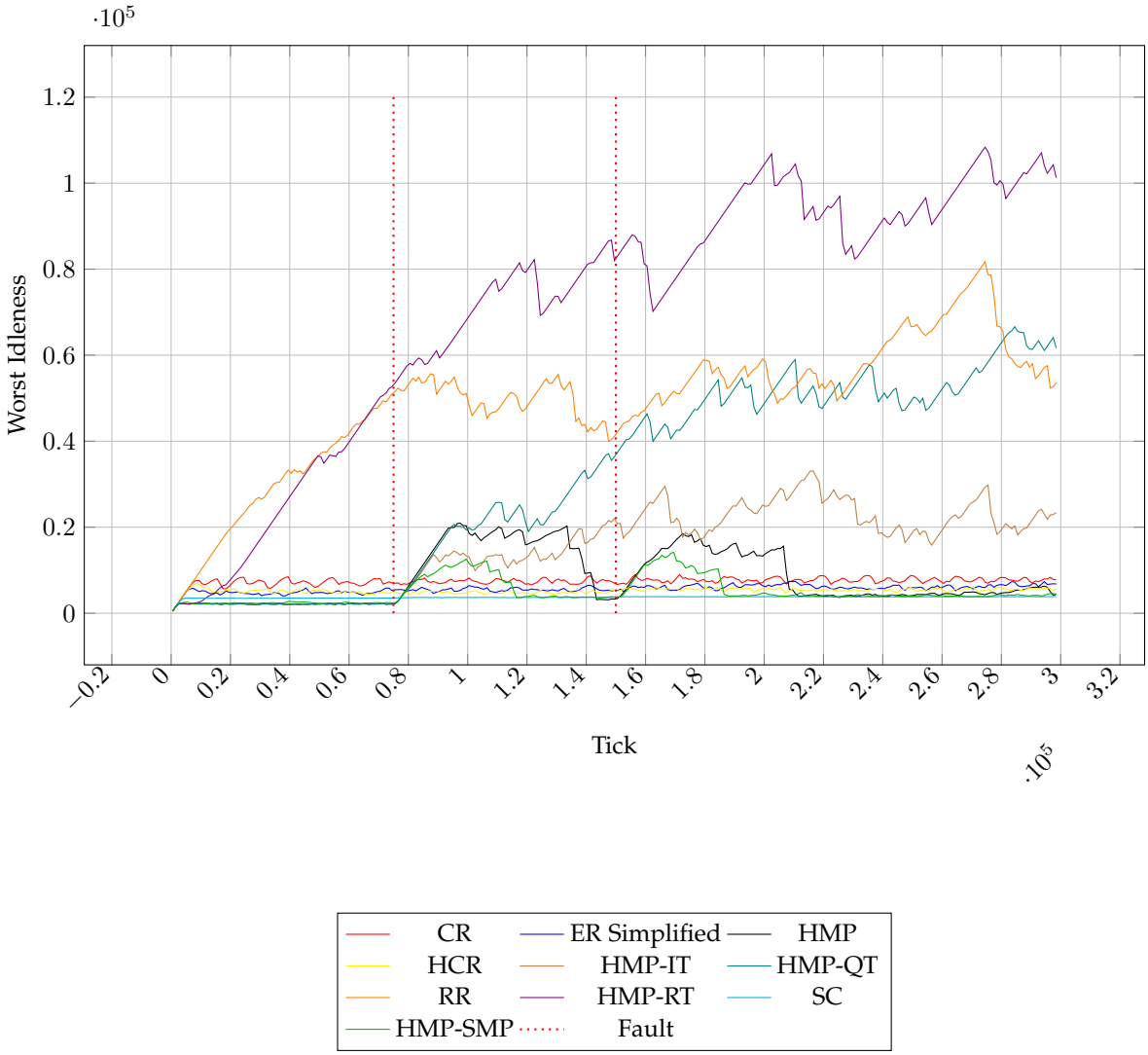


Fig. 23: Experiment 3: Worst Idleness over time on the Cave Map on with fault at tick 75000,150000

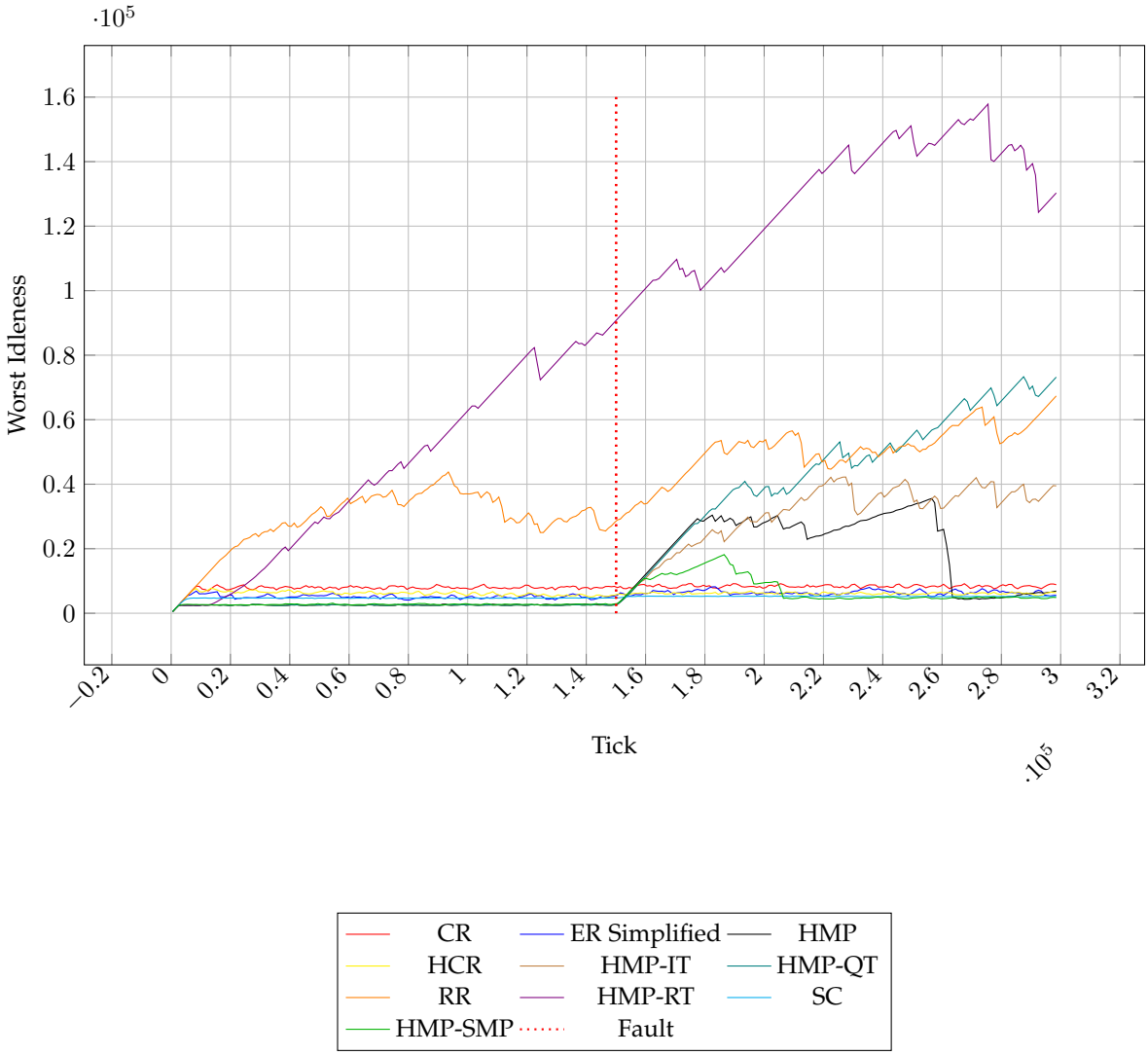


Fig. 24: Experiment 3: Worst Idleness over time on the Building Map on with fault at tick 150000,150150

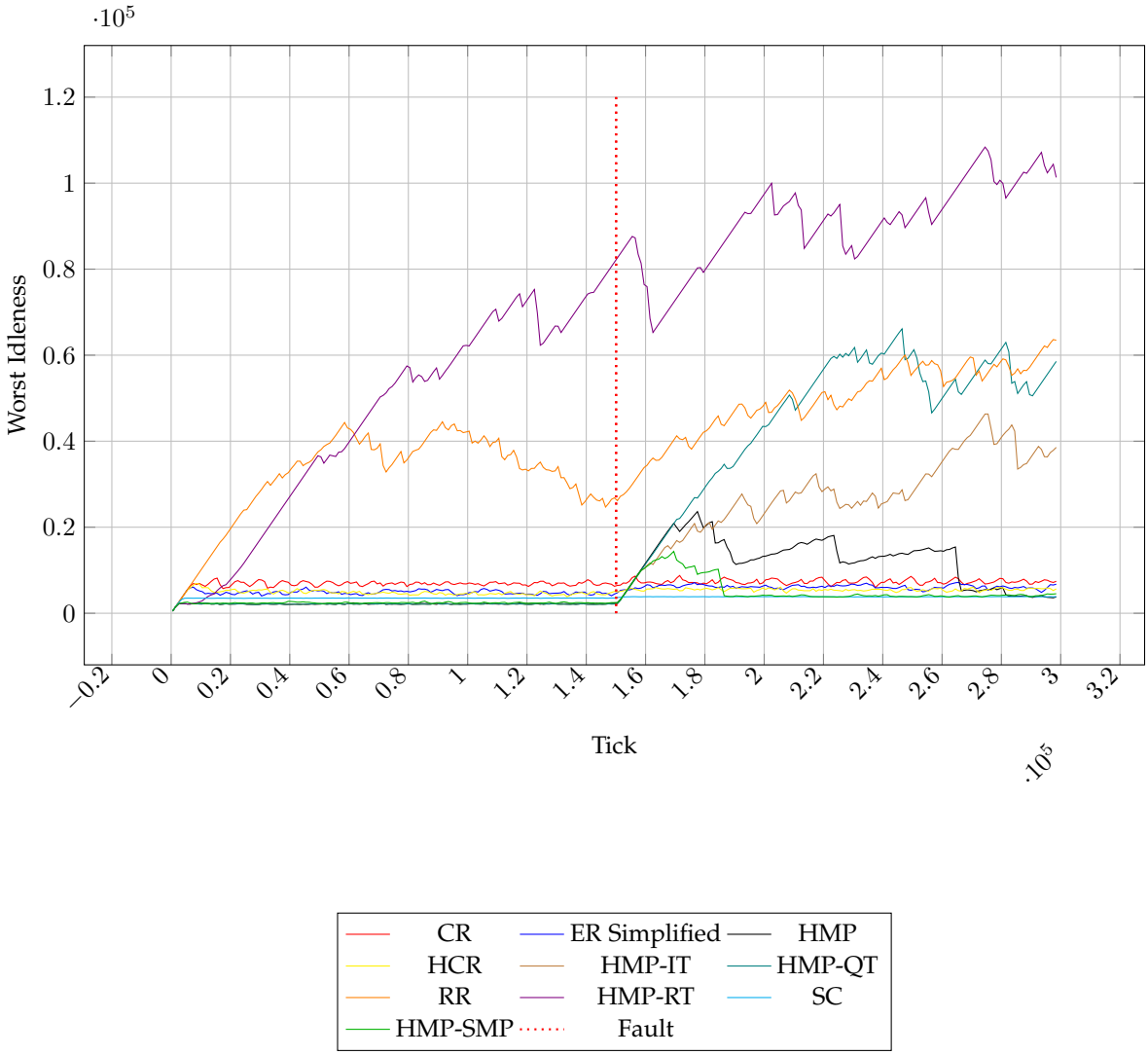


Fig. 25: Experiment 3: Worst Idleness over time on the Cave Map on with fault at tick 150000,150150

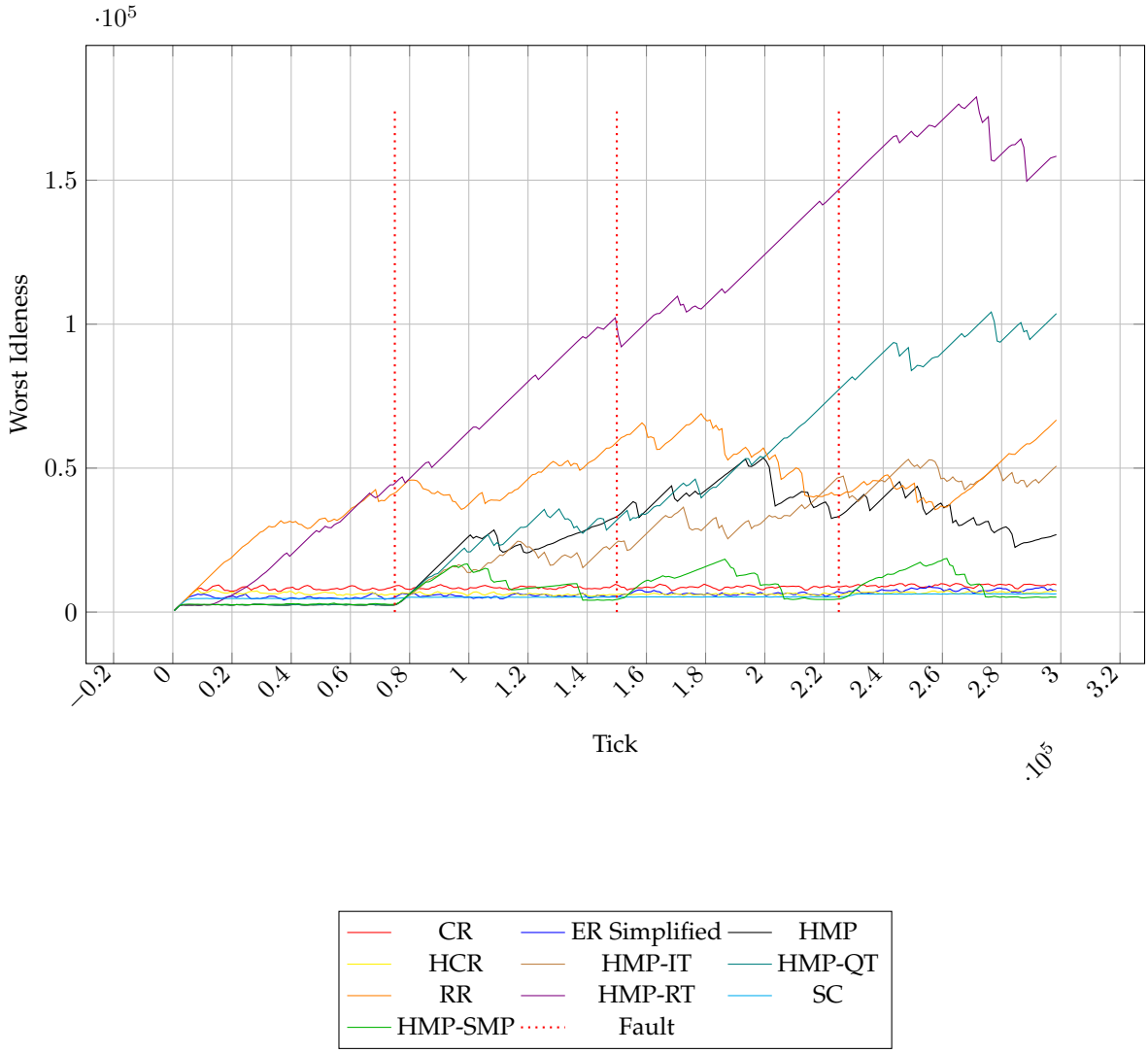


Fig. 26: Experiment 3: Worst Idleness over time on the Building Map on with fault at tick 75000,150000,225000

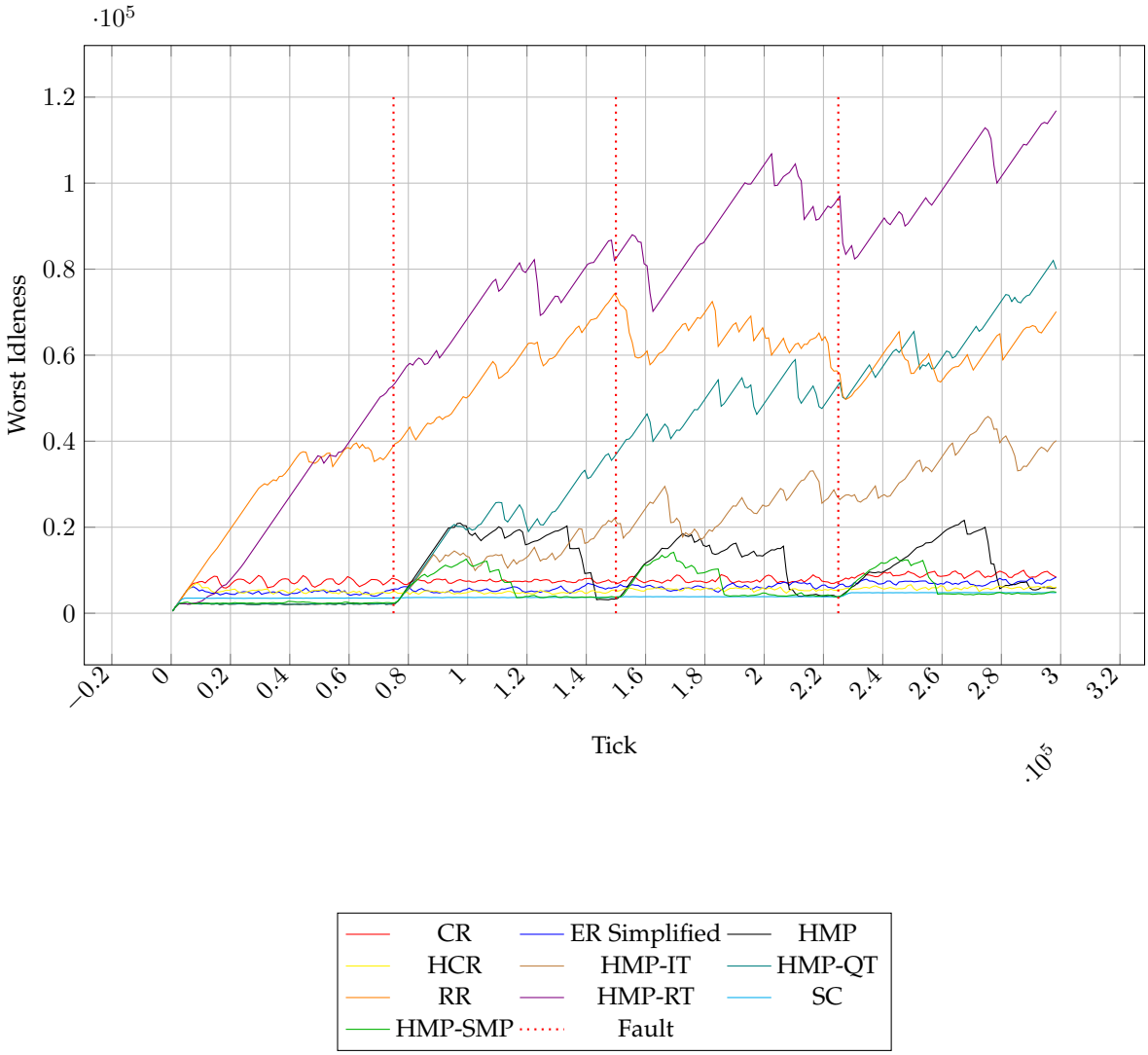


Fig. 27: Experiment 3: Worst Idleness over time on the Cave Map on with fault at tick 75000,150000,225000

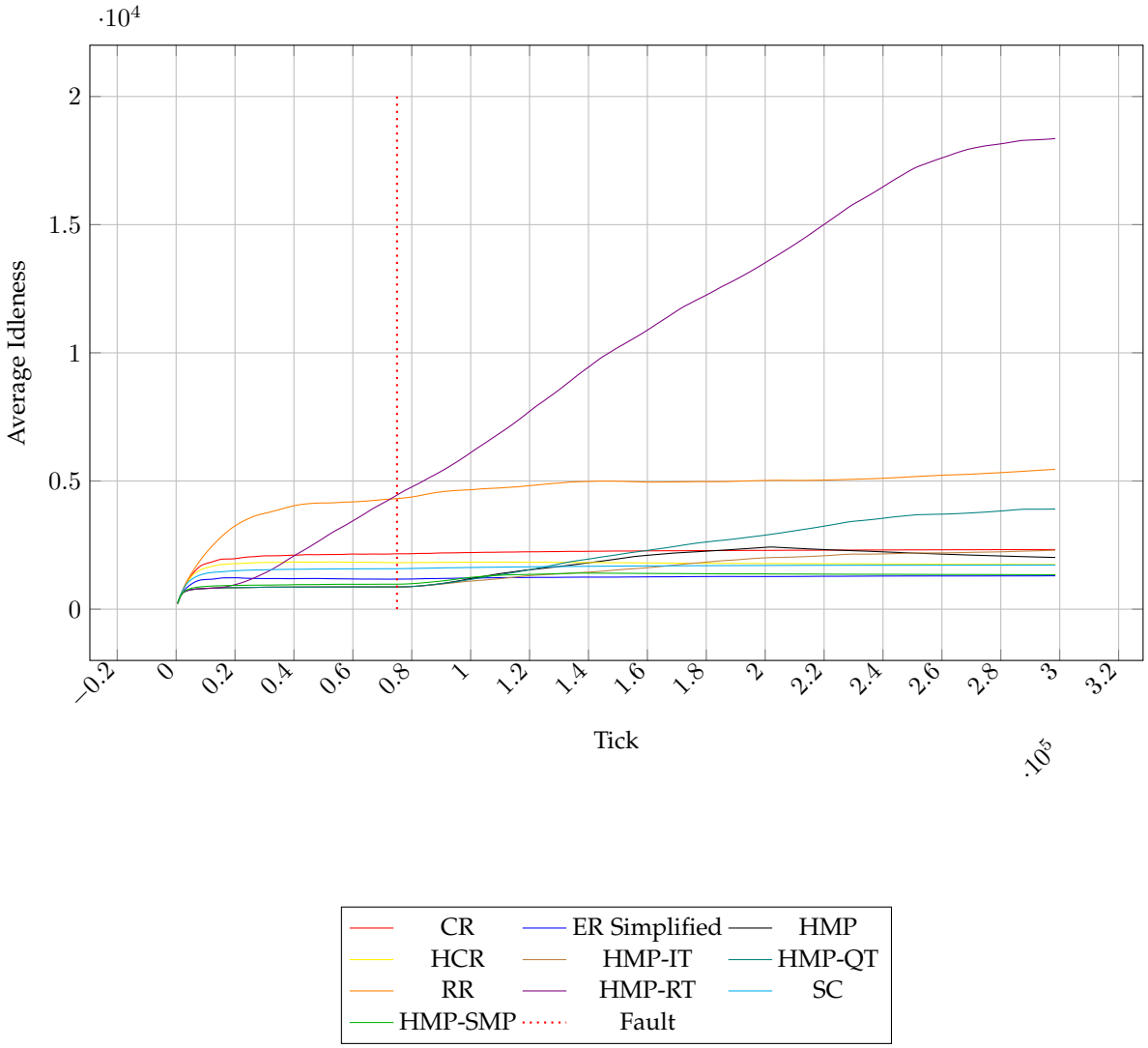


Fig. 28: Experiment 3: Average Idleness over time on the Building Map on with fault at tick 75000

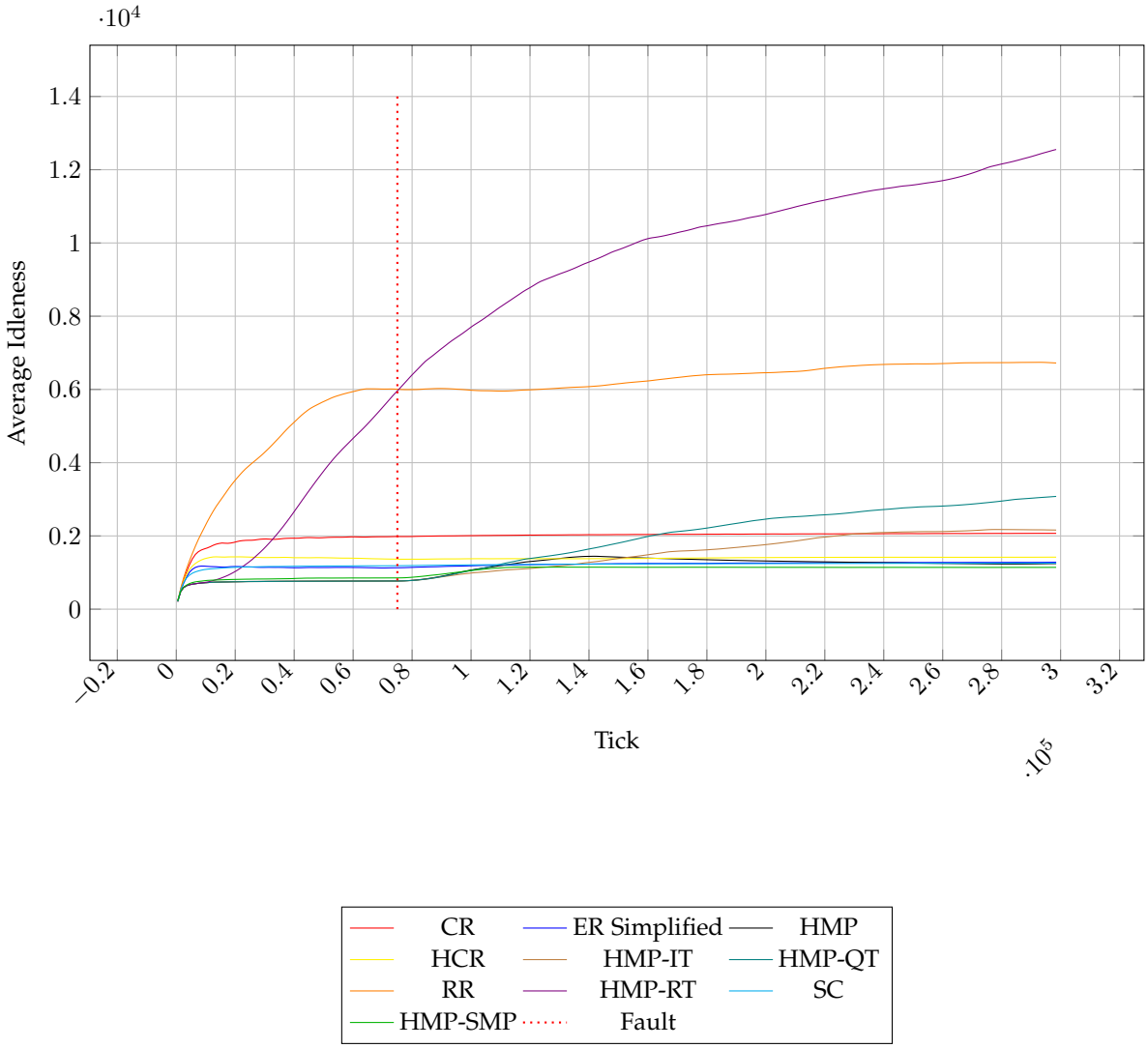


Fig. 29: Experiment 3: Average Idleness over time on the Cave Map on with fault at tick 75000

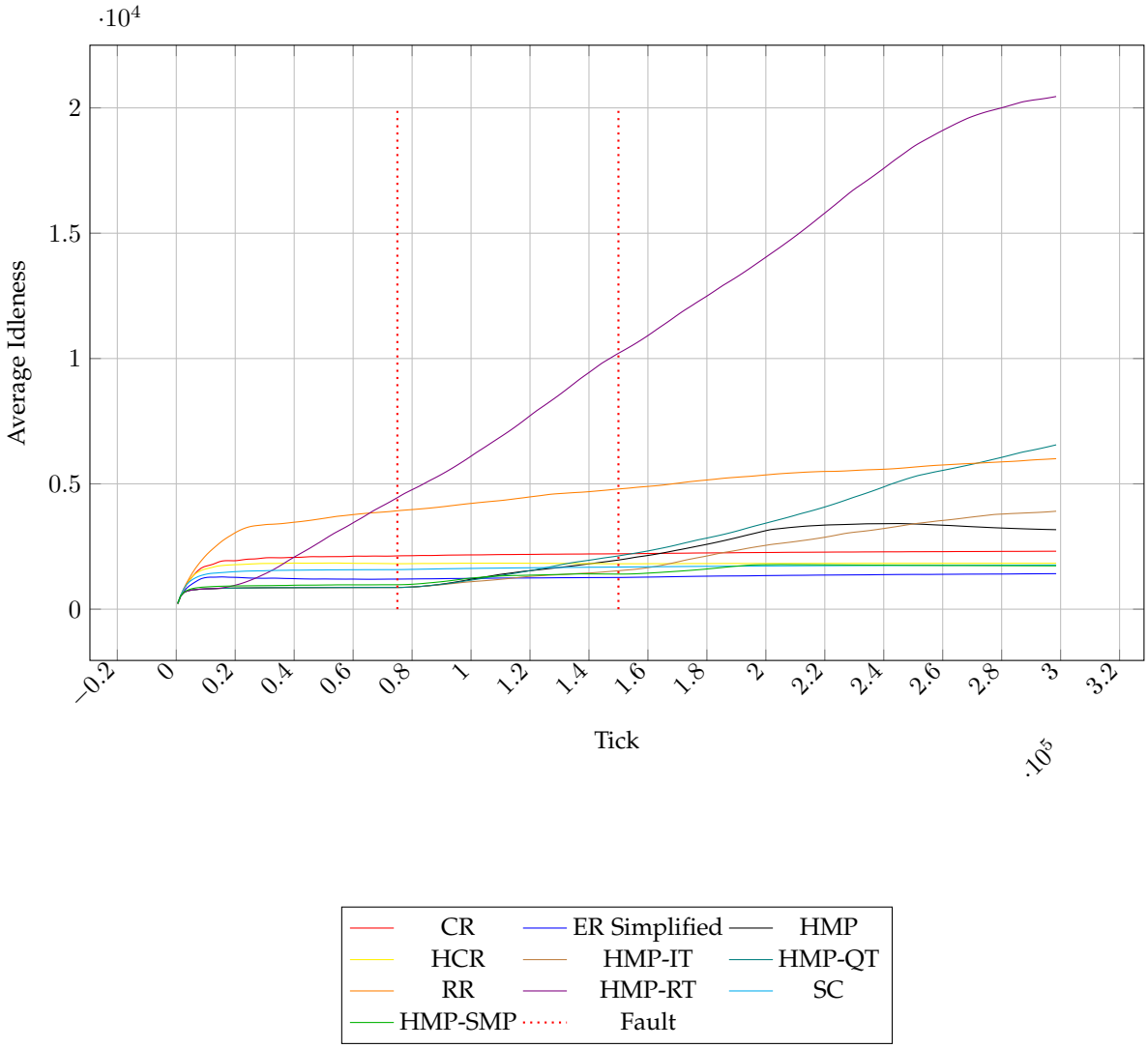


Fig. 30: Experiment 3: Average Idleness over time on the Building Map on with fault at tick 75000,150000

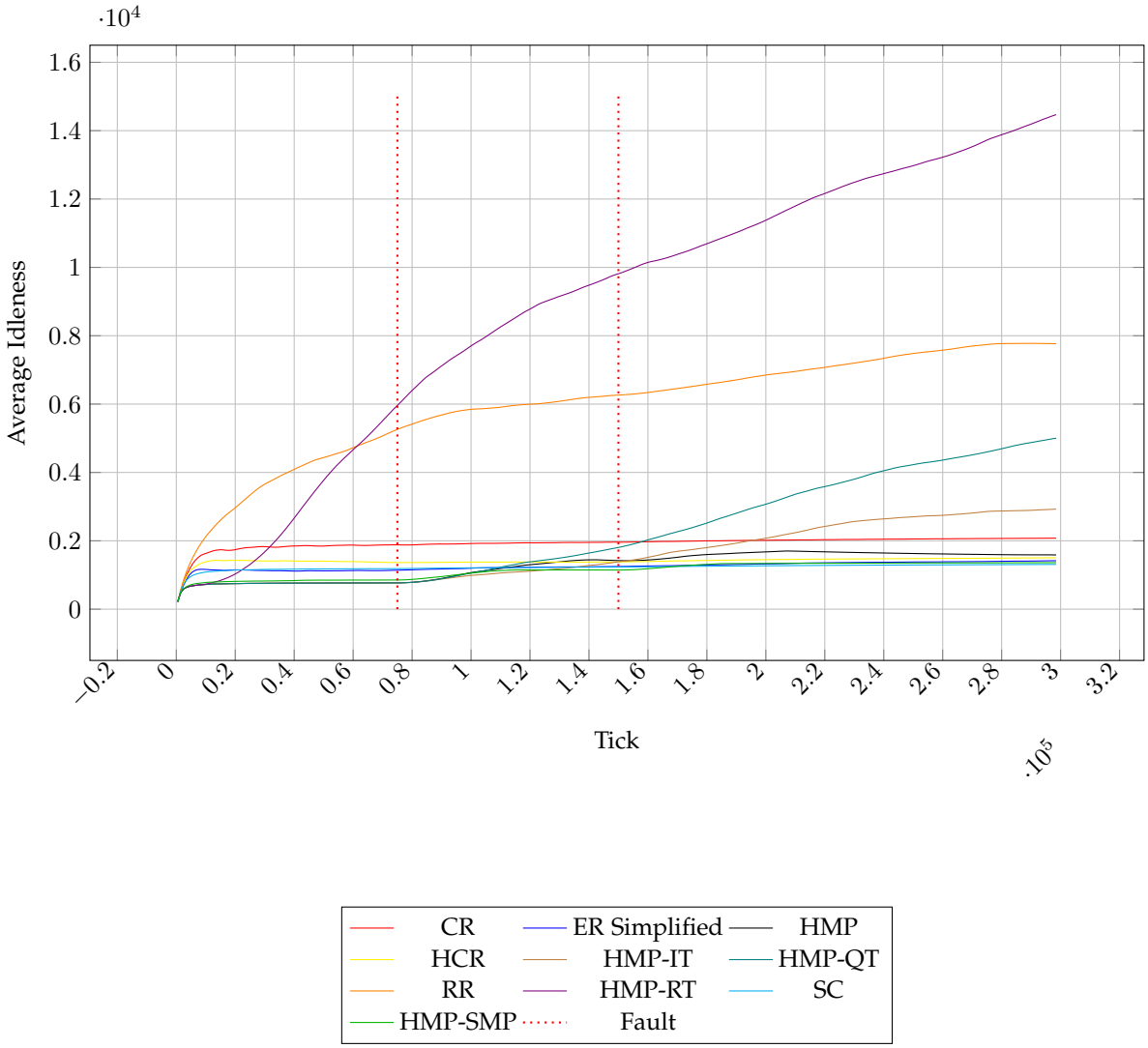


Fig. 31: Experiment 3: Average Idleness over time on the Cave Map on with fault at tick 75000,150000

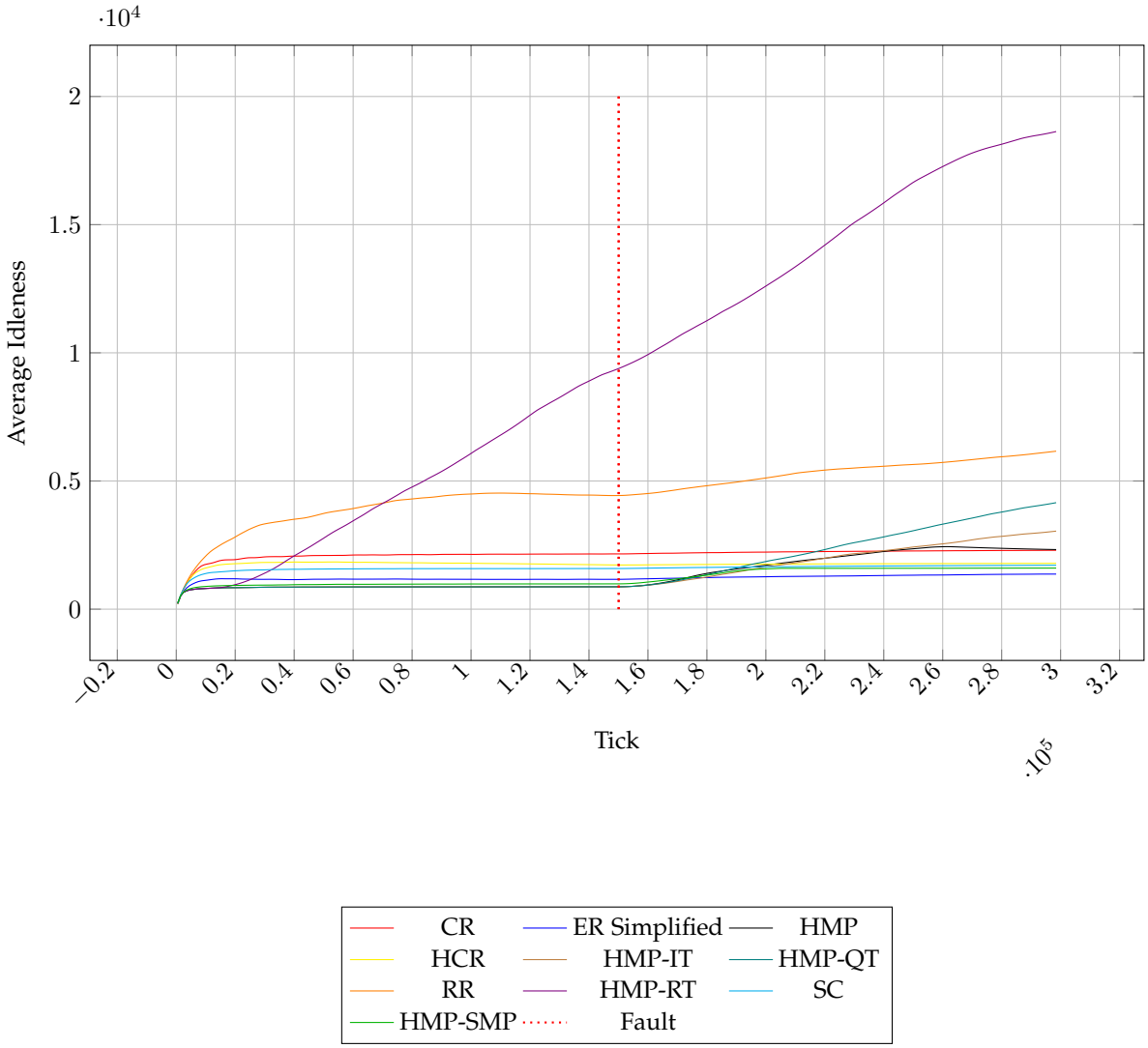


Fig. 32: Experiment 3: Average Idleness over time on the Building Map on with fault at tick 150000,150150

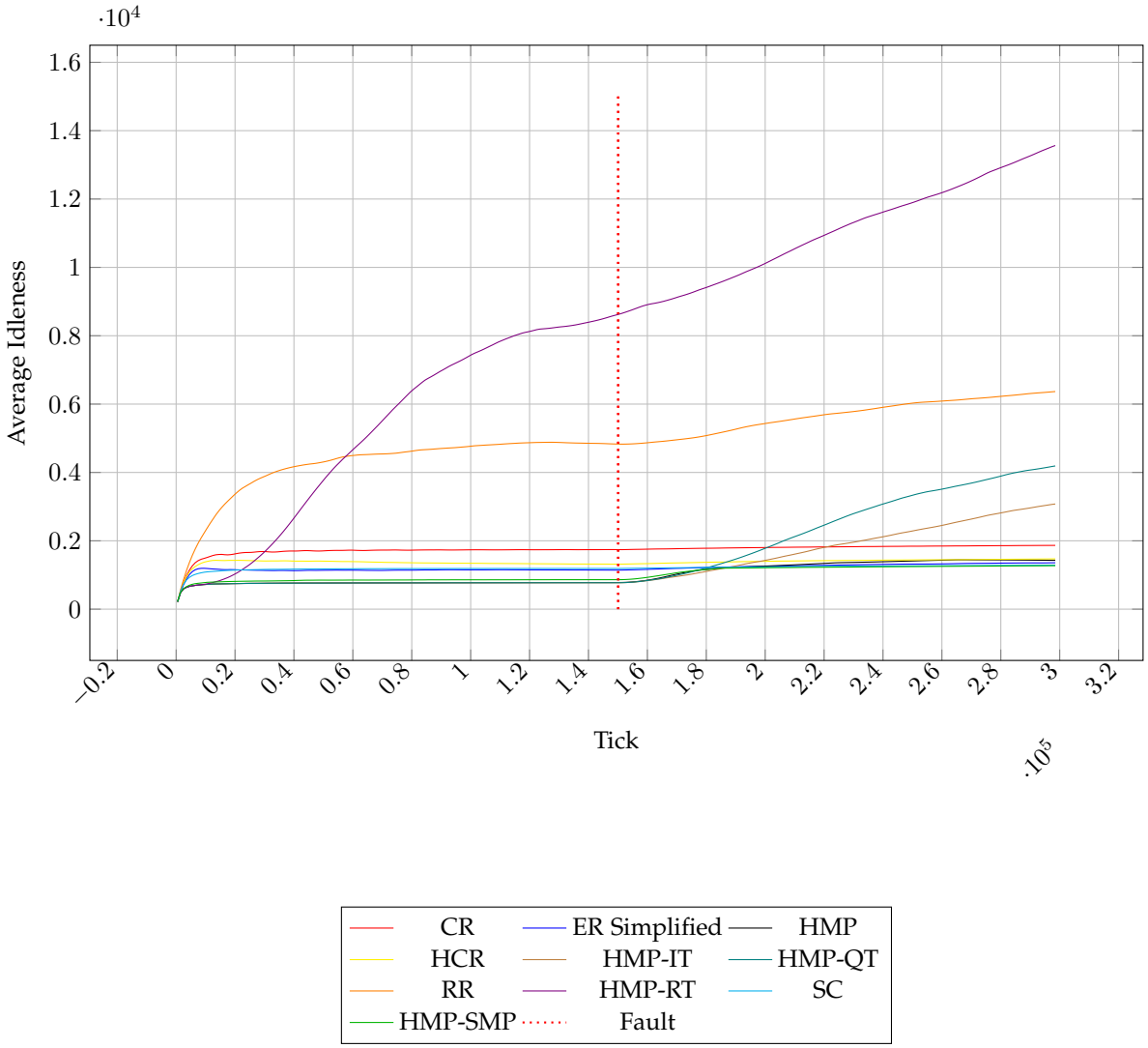


Fig. 33: Experiment 3: Average Idleness over time on the Cave Map on with fault at tick 150000,150150

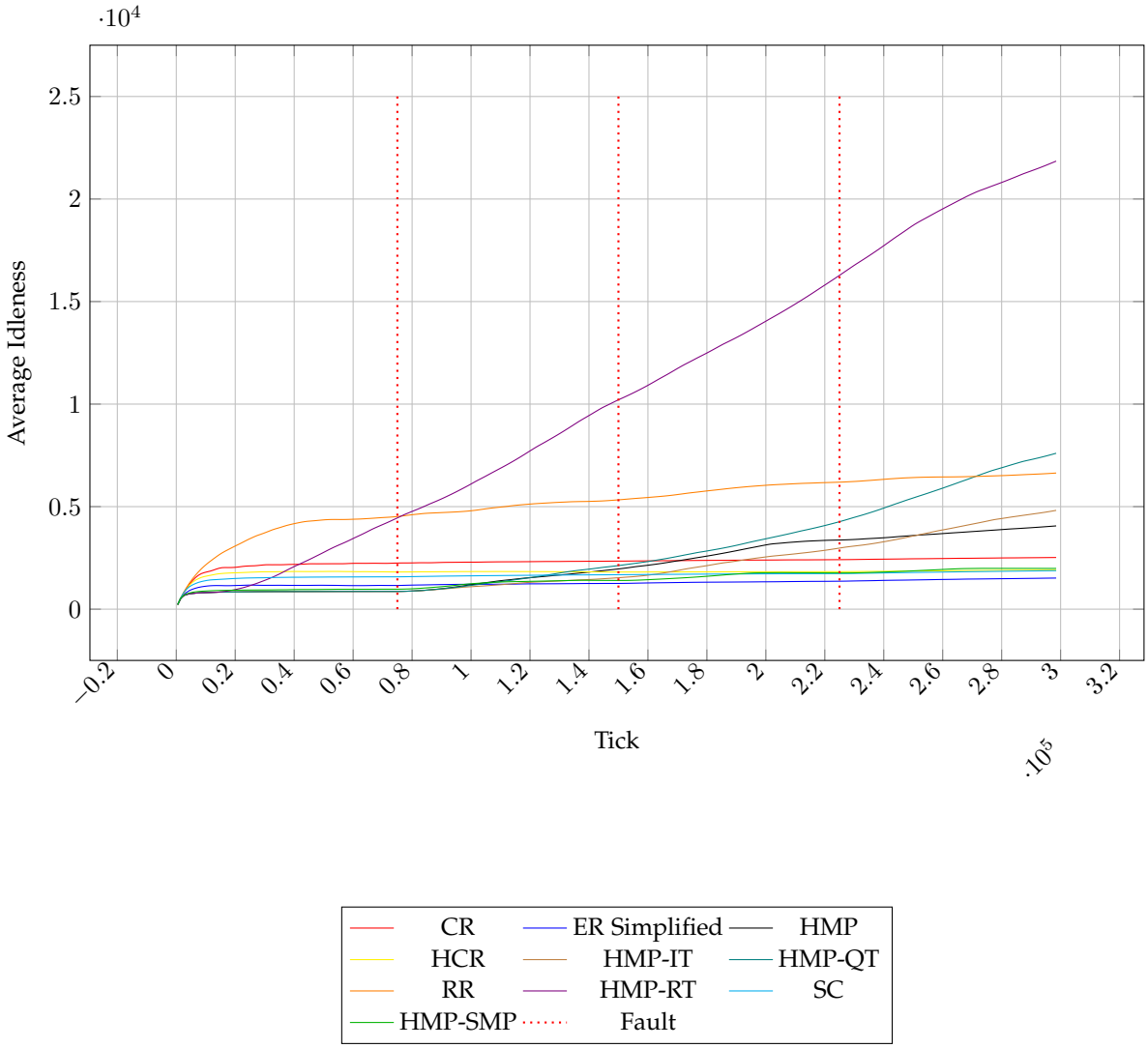


Fig. 34: Experiment 3: Average Idleness over time on the Building Map on with fault at tick 75000,150000,225000

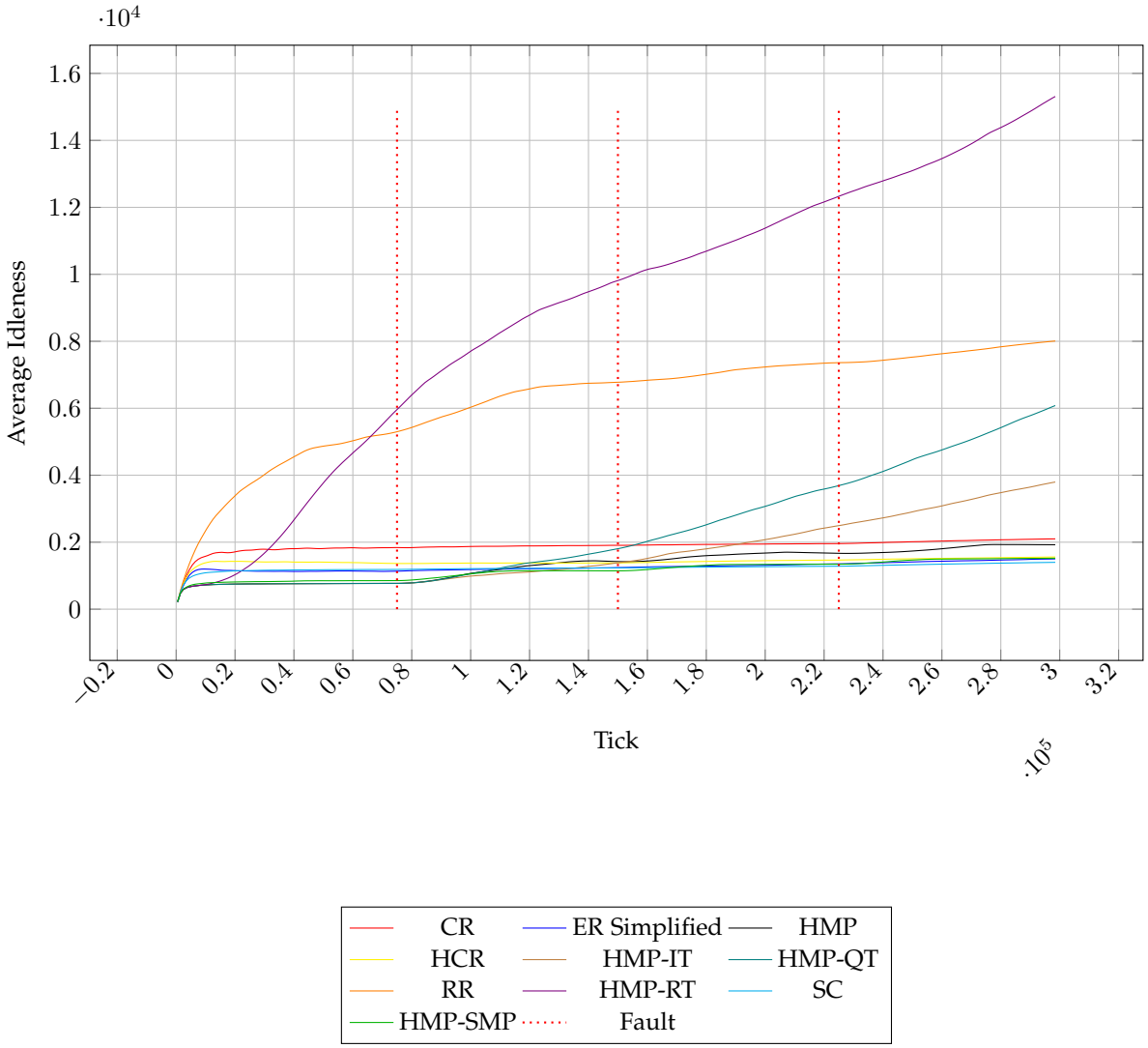


Fig. 35: Experiment 3: Average Idleness over time on the Cave Map on with fault at tick 75000,150000,225000