Near-Optimal Continuous Patrolling with Teams of Mobile Information Gathering Agents

R. Stranders[a], E. Munoz de Cote[b], A. Rogers[a], N.R. Jennings[a,c]

[a] *Electronics and Computer Science*
University of Southampton
Southampton, United Kingdom
[b] *Department of Computer Science,*
National Institute of Astrophysics, Optics and Electronics,
Tonantzintla, Mexico
[c] *Department of Computing and Information Technology*
King Abdulaziz University
Jeddah, Saudi Arabia

Abstract

Autonomous unmanned vehicles equipped with sensors are rapidly becoming the *de facto* means of achieving situational awareness — the ability to make sense of, and predict what is happening in an environment. Particularly in environments that are subject to continuous change, the use of such teams to maintain *accurate* and *up-to-date* situational awareness is a challenging problem. To perform well, the vehicles need to patrol their environment continuously and in a coordinated manner.

To address this challenge, we develop a near-optimal multi-agent algorithm for continuously patrolling such environments. We first define a general class of multi-agent information gathering problems in which vehicles are represented by information gathering agents — autonomous entities that direct their activity towards collecting information with the aim of providing accurate and up-to-date situational awareness. These agents move on a graph, while taking measurements with the aim of maximising the cumulative discounted *observation value* over time. Here, observation value is an abstract measure of reward, which encodes the properties of the agents' sensors, and the spatial and temporal properties of the measured phenomena. Concrete instantiations of this class of problems include monitoring environmental phenomena (temperature, pressure, etc.), disaster response, and patrolling environments to prevent intrusions from (non-strategic) attackers.

In more detail, we derive a single-agent divide and conquer algorithm to compute a continuous patrol (an infinitely long path in the graph) that yields a near-optimal amount of observation value. This algorithm recursively decomposes the graph, until high-quality paths in the resulting components can be computed outright by a greedy algorithm. It then constructs a patrol by concatenating these paths using dynamic programming. For multiple agents, the algorithm sequentially computes patrols for each agent in a greedy fashion, in order to maximise its marginal contribution to the team. Moreover, to achieve robustness, we develop algorithms for repairing patrols

Email addresses: `rs06r@ecs.soton.ac.uk` (R. Stranders), `jemc@inaoep.mx` (E. Munoz de Cote), `acr@ecs.soton.ac.uk` (A. Rogers), `nrj@ecs.soton.ac.uk` (N.R. Jennings)

when one or more agents fail or the graph changes.

For both the single and the multi-agent case, we give theoretical guarantees (lower bounds on the solution quality and an upper bound on the computational complexity in the size of the graph and the number agents) on the performance of the algorithms. We benchmark the single and multi-agent algorithm against the state of the art and demonstrate that it typically performs 35% and 33% better in terms of average and minimum solution quality respectively.

Keywords: Multi-agent Systems, Information Gathering Agents, Mobile Sensors, Multi-Robot Path Planning, Patrolling, Sequential Decision Making

1. Introduction

Unmanned autonomous vehicles equipped with sensors are rapidly becoming the *de facto* means of achieving situational awareness — the ability to make sense of, and predict what is happening in an environment — in application domains that are highly dynamic in nature, such as disaster management, military reconnaissance and climate research. In these domains, and many others besides, the use of autonomous vehicles reduces the need for exposing humans to hostile, impassable or polluted environments. Moreover, in such environments the deployment of wireless sensor networks (WSNs) is difficult, potentially not cost effective for temporary use, or cannot be performed in a sufficiently timely fashion. Now, operating as a team, rather than a collection of individuals, these unmanned vehicles can provide up-to-date coverage of a large area by coordinating their movements, and improve the robustness of this coverage by compensating for the failure of one or more vehicles. For example, in the aftermath of a major earthquake, a team of unmanned aerial vehicles (UAVs) can support first responders by patrolling the skies overhead (Maza et al., 2011). By working together, they can supply real-time wide-area surveillance on the movements of crowds and the spread of fires and floods (Delle Fave et al., 2012). In similar vein, teams of UAVs can be used to track and predict the path of hurricanes (Fitzpatrick, 2009). Since these UAVs face hostile environmental conditions, extra care must be taken to deal with the potential loss of one or more UAVs. Under such conditions, the coverage provided by the vehicles can be made to degrade gracefully if the responsibilities of their failed counterparts are shared through teamwork.

In light of this, the main challenge we address in this paper is the use of a team of autonomous vehicles in order to monitor the continuously changing state of the phenomena in their environment — or put differently — to provide *accurate* and *up to date* situational awareness. In order to do so they need to patrol the environment continuously in a coordinated manner.

We model this challenge as a general class of information gathering problem, in which vehicles are represented by *information gathering agents* — autonomous entities that direct their activity towards collecting information with the aim of providing accurate and up-to-date situational awareness. The generality of this class of problems is achieved through the use of the concept of an *information value function.* This function encodes the properties of the agents' sensors (such as their accuracy and range), the spatial and temporal properties of the measured phenomenon (e.g. how much correlation exists between measurements at two different locations and how quickly these measurements becomes stale), and a metric of the agents' performance (e.g. the prediction accuracy of the current and future temperature across the environment, or the maximum time that has elapsed between two observations of the same location). The physical layout of the environment is modelled by a graph, which defines the allowed movements of the agents. Given this, the main objective of the team of agents is to patrol the graph continuously so as to maximise the cumulative discounted observation value over time.

Solving this problem optimally for all but the smallest instances is impossible due to the exponential growth of the number of possible paths in the size of the graph and the number of agents (to be more precise, it is known to be NP-hard (Karp, 1982)). Therefore, we develop a multi-agent algorithm that computes patrols that are *boundedly optimal*, i.e. policies that are guaranteed to be within a (small) factor of the optimal one. This algorithm solves two interrelated issues: (i) specifying how individual agents should behave, and (ii) determining how a team of agents should interact in order to maximise team performance. The structure of our multi-agent algorithm reflects this dichotomy; it invokes a single-agent algorithm for each individual agent to maximise its marginal contribution to the performance of the multi-agent team.

In more detail, the single-agent algorithm uses a divide and conquer strategy to efficiently compute a high quality patrol (in terms of observation value). This algorithm uses three main operations: Divide, Conquer and Merge. Divide recursively decomposes the graph using a graph clustering algorithm, until the diameter of the resulting components is "small enough". Conquer then solves the problem within these components using a greedy algorithm that computes high quality paths through them. Finally, Merge concatenates these paths using dynamic programming into a patrol for the top level problem (the entire graph).

To compute patrols for multiple agents, the single-agent algorithm is invoked incrementally for each agent with the aim of maximising its marginal contribution to the team. More precisely, the observation value collected by agent i is maximised subject to the patrols for agents $1, \ldots, i-1$. This is commonly referred to in the economic cost-sharing literature as a *marginal contribution* scheme (Nisan et al., 2007). In this scheme, each agent is given a reward proportional to the welfare it contributes to the team. Effectively, under this reward structure, agent i's goal becomes to collect the reward left behind by agents $1, \ldots, i-1$. We show that this results in greatly reduced computational overhead compared to searching the joint solution space of patrols for all i agents.

We provide theoretical guarantees on the performance and computation of both the single and multi-agent algorithms. Given the potential life critical nature and sensitivity of the application domains in which the agents are often operating, these guarantees — particularly on the worst-case performance — are important when the algorithm is applied in the real world. Specifically, we show that the multi-agent patrol obtained through greedy computation is at least 63% as good as the best set of single-agent patrols (i.e. of the type computed by the single-agent algorithm). Moreover, we show that our algorithm scales well with the size of the environment. However, it scales exponentially with the number of agents, but we empirically demonstrate that this can be kept in check by pruning those states in the MDP that are not reachable from a given initial state of the environment.

Next, we make the multi-agent algorithm robust against component failure (which may occur in hostile environments) by developing efficient algorithms for repairing patrols in the event of failure of one or more agents, or when the layout graph changes. In the former case, agents fill the gap left behind by the failed agent by adopting the patrol of their predecessors. Rather than recomputing these patrols from scratch, existing patrols can be (partially) reused, resulting in a reduction of up to 50% in the number of states that needs to be searched (which is proportional to the computation time required) without losing solution quality. In the latter case, the recursive nature of the single-agent algorithm is exploited to limit the recomputation to the affected subproblem in which the graph changes occurs.

Finally, to ascertain how the algorithm performs in practical settings, and how this relates to the theoretical performance guarantees, we provide an extensive empirical evaluation of our algorithms in two challenging information gathering scenarios. The first models a general patrolling task, in which agents are tasked with minimising the intra-visit time of different areas of their environment. The second resembles a disaster response scenario, in which the agents' goal is to monitor a continuously changing environmental phenomenon (e.g. temperature, radiation, pressure and gas concentration). Our findings show that our algorithm outperforms three state-of-the-art benchmark algorithms taken from the literature in terms of minimising average and maximum intra-visit time (in the first scenario) and average and maximum root-mean-square error (in the second scenario). Specifically, it typically reduces the former metric by 35% and the latter metric by 33% for 6 agents. Moreover, using a best-response algorithm in an attempt to compute the optimal multi-agent policy, we demonstrate that our algorithm achieves approximately 91% optimality in the problem instances we consider, providing strong evidence for the near-optimality of the multi-agent algorithm.

Now, recent work has addressed similar challenges in environments that are static over time, or are changing at a rate that is negligible compared to the time required to traverse them (Meliou et al., 2007; Singh et al., 2009). Similarly, techniques from the literature on continuous localisation and mapping (SLAM) (Thrun et al., 2005, Chapter 10), while relevant, also typically assume that the environment is static, and as such agents need not revisit areas in order to update their beliefs about the environment. As a consequence, these algorithms compute finite length paths, which tend not to return to previously visited locations, since no additional information can be obtained from doing so. In contrast, in continuously changing environments, it is imperative that agents periodically revisit locations in order to provide up-to-date situational awareness. As a result, these existing approaches fall short of explicitly dealing with the rapid rate of change within the agents' environment that we consider here.

However, two algorithms from related work address this shortcoming (see Section 2 for more details). First, the decentralised algorithm proposed by Stranders et al. (2010a) is specifically geared towards multi-agent patrolling in environments subject to rapid change. It allows agents to patrol continuously by planning new portions of their paths using receding horizon control. Since it has a limited look-ahead, however, it does not provide guarantees on long term solution quality — a drawback given the potential life critical nature of the applications. Second, Elmaliach et al. (2009) consider the problem of repeatedly visiting all cells in a grid with an maximal frequency. Their algorithm computes a circular path (which we call a patrol), on which multiple robots are then are deployed equidistantly. However, our problem formulation is more general; the intra-visit time is but one of the possible performance metrics. Moreover, in our problem, moving equidistantly on a circular path is not necessarily optimal, and our algorithm therefore attempts to maximise the marginal contribution of each agent to the team instead.

To summarise, the primary contributions of this paper are:

- A new general class of information gathering problems involving multiple information gathering agents moving on a graph. This class relies on the concept of an information value function to encompass a large spectrum of concrete applications of UAVs, UGVs, and even mobile cleaning robots. It captures the spatial and temporal dynamics of the phenomenon of interest, the sensing capabilities of the agents, and the performance metric of the situational awareness achieved by the agents.

  The novelty of our problem formulation, compared to that of Singh et al. (2009) on which it is based, lies in the property of *temporality.* This property models the change of the environment over time.

- A non-myopic[1] divide and conquer algorithm for computing near-optimal patrols for *single* information gathering agents. The key novelty of this algorithm lies in the fact that it computes continuous patrols for patrolling rapid and continuously changing environments.

- An algorithm for computing near-optimal patrols for *multiple* agents by iteratively computing single-agent policies. This is done by maximising the marginal contribution of each agent, by collecting the reward that other agents were unable to collect. We achieve this by avoiding both *synchronous* double-counting, which occurs when two or more agents patrol the same cluster, and *asynchronous* double-counting, which occurs when an agent i patrols a cluster before another agent j ($i > j$), thereby effectively transferring the received reward from agent i to agent j.

  The novelty of this approach lies in the application of the sequential allocation technique (Singh et al., 2009) for the computation of a joint continuous patrol, which allows the algorithm to (empirically) scale much better than an algorithm that searches the entire joint space of patrols. While the latter does not scale beyond two agents in the settings we consider, our algorithm computes policies for six agents in less than one minute on a standard desktop machine.

- Algorithms for improving the robustness of the multi-agent patrols in the event of failure of one or more agents or changes in the graph. These algorithms repair the offline computed patrols during their execution. Both can save a significant amount of computation by reusing results from the offline stage (typically in excess of 50%), making them efficient methods for coping with *a priori* unknown events.

- Theoretical guarantees on both solution quality and computation cost of both the single-agent and multi-agent algorithms.

[1] We refer to a non-myopic algorithm as one that uses a multiperiod optimisation criterion, as opposed to a myopic optimisation one. Note that cyclic patrols use a multiperiod (or more accurately, infinite period) optimisation criterion.

- An empirical evaluation of our multi-agent algorithm by benchmarking it against a range of state of the art algorithms, such as the decentralised receding horizon control (RHC) algorithm (Stranders et al., 2010a), the global greedy algorithm (Vidal et al., 2001) and an algorithm that solves a modified Travelling Salesman Problem (TSP) to make the agents periodically observe the entire environment (Sak et al., 2008). We demonstrate that our multi-agent algorithm typically performs 35% better in terms of the average quality of situational awareness, and 33% better in terms of minimum quality. Finally, we empirically evaluate the near-optimality of the multi-agent algorithm by attempting to improve the multi-agent patrols using a best-response algorithm, whereby agents repeatedly compute the best patrol in response to their peers. While this algorithm is moderately effective (yielding up to 9% improvement), the improvement it achieves comes at a considerable computational cost (it searches 10–100 times more states than the multi-agent algorithm). We consider this evidence for the relative effectiveness and efficiency of our algorithms.

The remainder of the paper is organised as follows. In Section 2 we discuss the state of the art. In Section 3 we formally define the problem of multi-agent information gathering. In Section 4 we describe our algorithm for computing patrols for single and multiple information gathering agents. In Section 5 we derive bounds on the solution quality and the computational complexity of this algorithm. In Section 6 we describe the algorithms for repairing patrols in response to *a priori* unknown events. In Section 7 we empirically evaluate the algorithms. We conclude in Section 8.

2. Related Work

Recent work in (multi) sensor/robot patrolling can be classified along four orthogonal dimensions pertaining to algorithmic properties:

- *Offline vs. online.* Offline algorithms compute patrols before sensors are deployed, while online algorithms control the sensors' motion during operation. As a result, online algorithms are better able to revise patrols after (the sensors' belief of) the environment has changed, or when sensors fail unexpectedly during their mission.

- *Finite vs. infinite planning horizon.* Finite planning horizon algorithms compute patrols that maximise reward (or performance) over a finite horizon, infinite horizon (non-myopic) algorithms maximise an expected sum of rewards over an infinite horizon.

- *Continuous patrolling vs. single traversal.* Continuous patrolling is geared towards monitoring dynamic environments. These include those found in military and security domains, in which intruders attempt to breach a perimeter — which has to be continuously patrolled — or disaster management scenarios, in which decision makers continuously need accurate and up-to-date situational awareness. Single traversals are useful when the aim is to obtain a one-off snapshot of an environment. The work on single traversal is relevant, because techniques for computing single traversals can be exploited to compute infinitely long continuous patrols by concatenating single traversals in different parts of an environment.

- *Strategic vs. non-strategic patrolling.* Strategic patrolling attempts to reduce the loss caused by intrusions or attacks from perfectly rational (i.e. expected payoff maximising) intruders. Non-strategic patrolling takes place in the absence of such strategic entities, for example when monitoring nature or searching for confused civilians after a disaster. This work is non-strategic, but we discuss strategic patrolling work as well in light of future extensions.

Furthermore, these approaches assume or exploit properties of the environments in which the sensors are situated:

- *Spatial or spatio-temporal dynamics.* In environments with spatial dynamics only, observations vary only along the spatial dimensions, while in environments with spatio-temporal dynamics, observations are a function of both their spatial and temporal coordinates. The former is consistent with phenomena that stay (almost) fixed over time, such as terrain height or the layout of a building. The latter is consistent with phenomena that vary in space and time, such as weather conditions, radiation or gas concentration.[2]

Since the primary contribution of this paper is a non-myopic algorithm for computing patrols in environments with spatio-temporal dynamics, we will use the two corresponding dimensions — myopia and spatio-temporal dynamism — to discuss the state of the art. Table 1 summarises these, as well as the other two aforementioned properties.

Non-myopic spatial algorithms. Previous work in the class of infinite horizon spatial algorithms are based on the assumption that the environment is static over time. Under this assumption, it suffices to traverse the environment once, while ensuring that the informativeness of the observations made along the path is maximised. Since visiting the same location twice does not result in new information, these algorithms will attempt to avoid this. This is in contrast with our assumption that the environment varies in time as well as space, in which case revisiting locations is a necessary requirement for optimality. Algorithms found in this non-myopic spatial class consist primarily of approximation algorithms for the single-sensor non-adaptive (Singh et al., 2007) and multi-sensor adaptive (Singh et al., 2009) setting with energy constraints (e.g. finite battery or mission time). Both works exploit an intuitive property of diminishing returns that is formalised in the notion of *submodularity*: making an observation leads to a bigger improvement in performance if the sensors have made few observations so far, than if they have made many observations. This property holds in a wide range of real-life sensor applications, and is an assumption that our work shares with that of Singh et al. However, apart from solving a different problem (i.e. single traversal vs. continuous patrolling) the solution proposed by Singh et al. (2009) also differs algorithmically from ours. While they define a two-step algorithm for computing high quality single traversals through the environment, our solution is a full divide and conquer algorithm. In more detail, in the first step the algorithm of Singh et al. divides the environment into clusters, and computes high-quality paths

[2] Environments that vary along the temporal dimension only can monitored with a single fixed sensor, and are not of relevance here.

	Property				
Algorithm	Online[3]	Infinite look-ahead	Continuous Patrolling	Spatio-Temporal	Strategic
Singh et al. (2007)		×			
Singh et al. (2009)	×	×			
Meliou et al. (2007)	×	×		×	
Paruchuri et al. (2007)		×			×
Tsai et al. (2010)		×			×
Basilico et al. (2009)		×	×	×	×
Agmon et al. (2008)		×	×	×	×
Elmaliach et al. (2009)	×	×	×	×	
Grocholsky (2002)	×		×		
Fiorelli et al. (2006)	×		×		
Martinez-Cantin et al. (2007)	×		×		
Ahmadi and Stone (2006)	×		×	×	
Stranders et al. (2009)	×		×	×	
Stranders et al. (2010a)	×		×	×	
Our algorithm	×	×	×	×	

Table 1: The properties of the state of the art.

through these clusters. In the second step, these paths are concatenated to yield the desired traversal. The two steps bear similarity to the first two operations used in our algorithm (Divide and Conquer). However, our algorithm uses completely different techniques (sequential decision making) for concatenating paths within a single cluster into infinite-length patrols, which, unlike their solution, are recursively applied to increasingly smaller subdivisions of the environment, until the patrolling problem within these subdivisions becomes efficiently solvable.

Within this class of algorithms we also find work on deploying fixed security checkpoints to prevent intrusions by strategic opponents (Paruchuri et al., 2007; Tsai et al., 2010). In this context, both authors develop an efficient procedure for generating checkpoint deployments that prevent intrusions or at least minimise their impact. The work focuses on tractability, directly dealing with the exponential explosion incurred in the attacker's and defender's strategy spaces. Although finding checkpoint deployments is not directly linked to our work — i.e. there is no temporal element in their approach, and they use Stackelberg games to model the problem (instead of MDPs) — it is a problem somewhat related to the one studied here.

Non-myopic spatial-temporal algorithms. In the class of infinite horizon spatio-temporal algorithms we find a variation of the aforementioned work of Singh et al. (2007) that addresses the setting wherein the environment changes slowly over time (Meliou et al., 2007). Here, the challenge is to compute a sequence of informative paths conditioned on the observations collected along previous paths. The time it takes to patrol environments is assumed to be negligible compared to the rate at which they change. Therefore, the environment is considered static while sensors traverse paths. We do not make this assumption. Rather, we assume that environments often change at a rate that is too high for a single sensor to take an accurate snapshot.

Other work in this class focuses on patrolling in the presence of strategic evaders or intruders; a problem that is characterised by (possibly multiple) attackers attempting to avoid capture or breach a perimeter. The agents' main challenge in such cases is

[3] Our algorithm is online in that it is able to repair offline computed patrols in response of the failure of agents and changes in the graph (see Section 6). It does not perform online path planning.

to detect and capture these attackers in an effort to minimise loss. A good patrolling policy is one that frequently revisits locations, since attackers can appear anywhere at any time. Several offline and centralised optimal algorithms have been proposed (see for example Basilico et al. (2009); Agmon et al. (2008)) that compute non-myopic policies for multiple mobile robots. Due to the strategic nature of the problem, these require different techniques than the ones we use in our algorithm — Stackelberg games and partially observable stochastic games instead of the Markov decision processes. We will investigate the use of these techniques for strategic patrolling as a future extension of our work. However, since the problem of strategic patrolling is NP-hard (Basilico et al., 2009) (as is the problem we address in this paper), these optimal algorithms scale poorly and are only capable of solving small problem instances. As a consequence, approximation algorithms — such as the ones we propose in this paper — are needed to solve most practical patrolling problems in large graphs with many sensors.

A more similar approach to ours is that of Elmaliach et al. (2009), who consider the problem of repeatedly visiting all cells in a grid with an optimal frequency (similar to the TSP benchmark used in Section 7, but more scalable). Their solution is to find a circular path (which we call a patrol) based on a spanning tree (induced by the grid's topology). Multiple robots are then deployed equidistantly on this circular path. This process induces a Hamiltonian cycle and every cell will be visited with the same frequency. Nevertheless, there are several differences with our work. Our problem definition is more general: (i) it supports general (undirected) graphs instead of just grids, (ii) agents can observe more than one vertex at a time, (iii) different parts of the graph are not necessarily of equal importance, and (iv) the frequency of visiting vertices is but one of the possible performance metrics (see Experiments 1 and 2 in Section 7). Because of this, ensuring that robots move equidistantly on the circular path is not necessarily optimal (or even desirable). Therefore, our algorithm attempts to maximise the marginal contribution of each agent to the team instead.

Finite horizon spatial algorithms. The class of finite horizon spatial algorithms can be further categorised by the length of look-ahead. Some algorithms use greedy (i.e. single step look-ahead) gradient climbing techniques to reduce entropy in the prediction model (thus reducing the prediction error) (Grocholsky, 2002), optionally in combination with potential fields that force groups of agents into desirable formations for observing their environment (Fiorelli et al., 2006). Other algorithms use an increased (but finite) look-ahead by applying receding or finite horizon control. One application of this technique is to control a single agent whose goal is to minimise uncertainty about its own position as well as the location of various targets in its environment (Martinez-Cantin et al., 2007). Unfortunately, algorithms within this class cannot give performance guarantees due to their finite look-ahead.

Finite horizon spatial-temporal algorithms. Finally, the class of finite horizon spatio-temporal algorithms also use receding horizon control. To this end Ahmadi and Stone (2006) use decentralised negotiation to continuously refine the dimensions of the partitions which the agents are assigned to patrol. Stranders et al. (2009) and Stranders et al. (2010a) use the max-sum algorithm for decentralised coordination (Rogers et al., 2011) at predefined intervals to plan the next sequence of moves that yields the highest probability of non-strategic target capture (in the pursuit evasion domain), or the lowest prediction error (for monitoring environmental phenomena). While shown to be adaptive and effective, however, neither give performance guarantees on long-term performance.

This paper seeks to address the aforementioned shortcomings of the state of the art in the context of the central problem addressed in this paper, by not only taking the immediately received reward over a finite number of moves into account, but also the reward received over the remainder of the sensors' mission, making it non-myopic. As a result, the algorithm presented in this paper computes infinite length patrols for multiple sensors in highly dynamic environments. In addition, it can repair these offline computed patrols online in the event of failure of one or more agents.[4] Before discussing our algorithm in detail, however, we first present the formalisation of the problem it aims to solve.

3. Problem Definition

In this section we present a general formalisation of the multi-agent information gathering problem. This formulation is domain independent, and therefore does not reference any domain specific properties (such as targets, environmental phenomena, or intruders). This is accomplished through the use of the concept of *observation value* (cf. Meliou et al. (2007)), which abstracts from the chosen representation of the environment, and defines the value of observations in terms of their contribution towards improving the accuracy of situational awareness. Put differently, the collected observation value is a metric of how well the agents are performing.

Our formalisation is inspired by that of Singh et al. (2007), which we extend with a temporal dimension through the property of *temporality*. This property models the dynamism in the environment which is one of the central foci of this paper. In what follows, we introduce the three main aspects of the problem: (i) the physical environment, (ii) the information gathering agents and their capabilities and (iii) the agents' objective. A summary of the notation introduced in this section and used throughout this paper can be found in Table 2.

3.1. The Physical Environment

The physical environment is defined by its spatial and temporal properties. The former is encoded by a layout graph, which specifies how and where agents can move:

Definition 1 (Layout Graph)**.** *A layout graph is an undirected graph $G = (V, E)$ that represents the layout of the environment, where the set of spatial coordinates V is embedded in Euclidean space and edges E encode the movements that are possible between them.*

By modelling the physical layout of the environment as an *undirected graph*, we assume that agents can move along edges in both directions. While this is common in most environments, there are certain cases in which this assumption does not hold, for

[4] Since our algorithm is based on sequential decision making in order to compute infinitely long patrols, our repair algorithm is not related to plan repair in classical planning (which is characterised by the existence of a predefined goal state). A discussion of plan repair in classical planning is therefore considered beyond the scope of this paper.

Symbol	**Meaning**
$\mathbf{A} = \{1, \dots, M\}$	A set of agents
$\mathbf{A}_{-i}$	The set of agents $\{1, \dots, i-1\}$
$G = (V, E)$	An undirected graph encoding the physical layout of an environment (Definition 1)
$adj_G(v)$	The set of vertices adjacent to v in a graph G
$\mathrm{d}(u, v)$	The Euclidean distance between coordinates u and v (Definition 2)
$\mathrm{d}_G(u, v)$	The shortest path (Dijkstra) distance between vertices u and v in graph G (Definition 3)
$\mathrm{diam}(G)$	The diameter of a graph G, i.e. the length of the longest shortest path between any pair of vertices of G
$T = \{1, 2, \dots\}$	A discrete set of time steps (Definition 4)
$O = V \times T$	The set of all observations, i.e. the set of all spatio-temporal coordinates (Definition 7)
$O^t_{\mathbf{A}}$	The observations made by all agents at time t
O^t_i	The observations made by agent i at time t
$\mathbf{O}^t_{\mathbf{A}}$	The observations made by all agents at or before time t
$C = (V_C, E_C)$	A cluster (Definition 15)
$T = (V_T, E_T)$	A transit node (Definition 17)
$G[\mathcal{C}] = ((\mathbf{C} \cup \mathbf{T}), E_{\mathcal{C}})$	A cluster graph: an undirected bipartite graph of transit nodes and clusters in a given graph G. Edges $E_{\mathcal{C}}$ encode the connections between the two
$\mathbf{C}_{\max}$	The maximum number of clusters Divide creates before further recursive division is necessary
(T, C, T')	A subpatrol starting from transit node T through cluster C to transit node T' (Definition 19)
$P_{T,C,T'}$	The sequence of vertices in G visited by subpatrol (T, C, T')
$c(P_{T,C,T'})$	The length (number of vertices) of subpatrol (T, C, T')
$I(C, \lambda_C, T, T')$	The value of subpatrol (T, C, T') given that cluster C was patrolled λ_C time steps ago
λ_C	The number of time steps since cluster C was last visited
B	The maximum number of time steps allocated to an agent to patrol a cluster
f	A set function $f : 2^O \to \mathbb{R}^+$ that assigns *observation value* to a set of observations (Definition 10)
δ	The minimum distance between two observations for these to be considered independent (Property 3)
τ	The minimum time between two observations for these to be considered independent (Property 4)
γ	The discounting factor
π_i	The policy of agent i
s_i	The state of agent i
$S_r(s)$	The set of states reachable from state s

Table 2: A summary of the notation used throughout this paper.

instance in the presence of one-way streets or corridors. The reason for the focus on undirected graphs in this paper is that dealing with directed graphs requires the existence of appropriate graph clustering algorithms that satisfy the requirements stated in Section 4.1.1. This issue is discussed in further detail in Section 4.1.1.

In this paper we use two different measures of distance related to spatial coordinates V and layout graph G:

Definition 2 (Euclidean Distance)**.** *The Euclidean distance between two spatial coordinates $v_1 \in V$ and $v_2 \in V$ is denoted by $\mathrm{d}(v_1, v_2)$.*

Definition 3 (Dijkstra Distance)**.** *The Dijkstra distance between two spatial coordinates $v_1 \in V$ and $v_2 \in V$ is equal to the length of the shortest path and is denoted by $\mathrm{d}_G(v_1, v_2)$.*

The temporal properties of the physical environment are defined as follows:

Definition 4 (Time)**.** *Time is modelled by a discrete set of temporal coordinates $T = \{1, 2, 3, \dots\}$ (henceforth referred to as* time steps*) at which the agents observe the environment and at which their performance is evaluated.*

3.2. Information Gathering Agents

Agents are situated in the physical environment defined above.

Definition 5 (Information Gathering Agent)**.** *An information gathering agent (agent for short) is a physical mobile entity capable of taking observations. The set of all information gathering agents is denoted as $\mathbf{A} = \{1, \dots, M\}$.*

The movement and observation capabilities of agents are defined as follows:

Definition 6 (Movement)**.** *At all time steps T, all agents are positioned at one of the vertices of layout graph G. Multiple agents can occupy the same vertex. Movement is atomic, i.e. takes place within the interval between two subsequent time steps, and is constrained by layout graph G, i.e. an agent positioned at a vertex $v \in V$ can only move to a vertex $v' \in adj_G(v)$ that is adjacent to v in graph G. The speed of the agents is assumed to be sufficient to reach an adjacent vertex within a single time step.*

Definition 7 (Observation)**.** *An observation is a pair (v, t), where $v \in V$ is the spatial and $t \in T$ is the temporal coordinate at which it is taken. The set of all possible observations is denoted by $O = V \times T$.*

Definition 8 (Taking observations)**.** *Agents take observations at each time step at or near their current position. The time it takes to collect an observation is assumed to be negligible. Depending on type of the sensors they are equipped with, agents are able to observe one or more vertices in V at once. For example, an agent equipped with a camera can observe all vertices in the line of sight, possibly up to a certain distance. However, with a standard temperature sensor, an agent can only observe the current position. Hence, our model supports both types of sensors.*

There are several different sets of observations used in the formalisation of the optimal solution and our algorithms. To formalise these, we use the following notation:

- O_i^t: the set of observations made by agent i at time t.

- $O^t_{\mathbf{A}} = \bigcup_{i \in \mathbf{A}} O^t_i$: the set of observations made by all agents at time t.
- $\mathbf{O}^t_{\mathbf{A}} = \bigcup_{t' \leq t} O^{t'}_{\mathbf{A}}$: the set of observations made by all agents *at or before* time t. For convenience, we define $\mathbf{O}^t_{\mathbf{A}} = \varnothing$ for $t < 0$.

3.3. The Objective

As stated in the introduction, the objective of the agents is to maximise the quality of the situational awareness they provide. This quality is measured in terms of *observation value*:

Definition 9 (Observation Value)**.** *The observation value of a set of observations is proportional to the increase in situational awareness it brings about. Put differently, the better a set of observations allows the agents to understand and predict what is happening in their environment, the higher its observation value.*

The observation value of a set of observation is calculated by an *observation value function*:

Definition 10 (Observation Value Function)**.** *An observation value function f is a set function $f : 2^O \to \mathbb{R}^+$ that assigns* observation value *to a set of observations.*

To ensure generality of our model, the semantics of the observation value function are deliberately left abstract, as they can vary significantly depending on the type of environment, the agents' mission, and the phenomena they observe within it. It is important to note, however, that the observation value function encodes the following information:

- Any information about the dynamics of the process that is known *a priori*, such as the type of phenomenon that is monitored, the speed at which the environment is changing, and the correlation between observations along the temporal and spatial dimensions.
- The metric of the agents' performance. Concrete examples are mutual information (Guestrin et al., 2005), entropy (Ko et al., 1995), area coverage (Stranders et al., 2010b), and probability of capturing a non-strategic target (Stranders et al., 2010a), all of which are a measure of the quality of the picture of their environment compiled by the agents, subject to the properties of the environment mentioned in the first item.

We will see concrete instances of observation value functions in Example 1 and Section 7.

Now, there are a number of properties which many observation value functions have in common, which are exploited by our solution. These properties are:

Property 1 (Non-decreasing)**.** *Observation value functions are non-decreasing: $\forall A, B$ such that $A \subseteq B \subseteq O$, $f(A) \leq f(B)$. Thus, acquiring more observations never 'hurts'.*

Property 2 (Submodularity)**.** *Observation value functions are submodular: $\forall A, B$ such that $A \subseteq B \subseteq O$ and $\forall o \in O$:*

$$f(A \cup \{o\}) - f(A) \geq f(B \cup \{o\}) - f(B)$$

This property encodes the diminishing returns of observations, i.e. making an additional observation is more valuable if the agents have only made a few prior observations, than if they have made many.

Many aforementioned observation value functions exhibit the submodularity property in the context of information gathering, such as entropy, mutual information, area coverage and target capture probability.

Locality and *temporality* are two additional properties of observation value functions that formalise the (in)dependency of observations taken at two (possibly different) points in time and space:

Property 3 (Locality)**.** *Observations taken sufficiently far apart in space are (almost) independent (cf. Krause et al. (2006)). That is, there exists a distance $\delta \geq 0$, and a $\rho \geq 0$, such that for any two sets of observations A and B, if $\min_{a \in A, b \in B} \mathrm{d}(a, b) \geq \delta$, then:*

$$f(A \cup B) \geq f(A) + f(B) - \rho$$

Thus, the smaller δ, the less information an observation at a given spatial coordinate provides about a different spatial coordinate.

We assume the locality parameters ρ and δ do not change over time. This is because these values encode knowledge about the environment that is known *a priori*. This does not mean that the (future) measurements of the phenomena within the environment are known *a priori*, but it does imply that their dynamics do not change over time.

Property 4 (Temporality)**.** *Observations taken sufficiently far apart in time are (almost) independent. Formally, let $\sigma_t(\cdot)$ be a function that selects only those observations made at or after t:*

$$\sigma_t(A) := \{(v, t') \in A \mid t' \geq t\}$$

Then, there exists a $\tau \geq 0$, such that for all $A \subseteq O$, $\epsilon \geq 0$:

$$f(\sigma_{t-\tau}(A)) \geq f(A) - \epsilon$$

Thus, the smaller τ, the more dynamic the environment.

Note that if $\delta = \tau = \infty$, all observations are dependent. In such cases, we can no longer speak of locality or temporality. Thus, unlike Properties 1 and 2, an observation value function can exhibit locality and temporality in degrees and need not have these properties at all.

Given the formalisation above, we can now express the team's goal of maximising situational awareness (captured by observation value function f). To do this, we first need the concept of a patrolling policy:

Definition 11 (Patrolling Policy)**.** *A patrolling policy $\pi : 2^O \rightarrow 2^O$ specifies which observations $O_{\mathbf{A}}^t \subset O$ should be made at time step t, given that observations $\mathbf{O}_{\mathbf{A}}^{t-1} \subset O$ were made at the time steps before t, subject to movement and observation constraints imposed by layout graph G (Definition 6) and the agents' sensors (Definition 8). Put differently, $\pi(\mathbf{O}_{\mathbf{A}}^{t-1}) = O_{\mathbf{A}}^t$.*

The team's goal can now be expressed formally in terms of the optimal policy π^*. We will use $s_t = \mathbf{O}_{\mathbf{A}}^{t-1}$ and $a_t = O_{\mathbf{A}}^t$ for sake of simplicity.

Definition 12 (The Agents' Objective)**.** *The agents' objective is to maximise the expected value*[5] *of using a policy* π *of the discounted incremental observation value over time:*

$$\max_{\pi} E^{\pi}\left[\sum_{t\in T}\gamma^t r(s_t,a_t) \mid \pi(s_t)=a_t\right] \tag{1}$$

where $0 \leq \gamma \leq 1$ *is the discount factor that determines how much observations made in the near future are worth compared to those made in the further future. The instantaneous reward* $r(s_t,a_t) = f(s_t \cup a_t) - f(s_t)$*, i.e. the so-called "incremental value" of the observations* $a_t = O^t_{\mathbf{A}}$ *made by all agents at time* t*, given that observations* $s_t = \mathbf{O}^{t-1}_{\mathbf{A}}$ *were already made at or before time* $t-1$*.*

Finding the optimal patrolling policy π^* calls for quantifying the quality of any policy π, this can be computed using a recursive definition known as the Bellman (1957) equations for computing the state value function for policy π:

Definition 13 (Value Function)**.** *A value function* V^{π} *computes the discounted future incremental observation value that is received by following policy* π *from a given state (i.e. a set of collected observations* $\mathbf{O}^t_{\mathbf{A}}$*):*

$$V^{\pi}(s_t) := r(s_t,a_t) + \gamma V^{\pi}(a_t \cup s_t) \tag{2}$$

Using Definitions 11 and 13, we can now define the optimal patrolling policy that maximises Equation 1:

Definition 14 (Optimal Patrolling Policy)**.** *The optimal patrolling policy maximises the discounted incremental observation value and is defined as,*

$$\pi^*(s_t) := \arg\max_{a_t}\Big(r(s_t,a_t) + \gamma V^{\pi^*}(a_t \cup s_t)\Big) \tag{3}$$

where $a_t = O^{t+1}_{\mathbf{A}}$ *is a set of observations that can be collected by the agents at time step* $t+1$*, subject to movement and observation constraints and* $s_t = \mathbf{O}^t_{\mathbf{A}}$*.*

Proposition 1. *The optimal patrolling policy is the solution to the teams' objective.*

Proof. Recall Equation (3), let $s_{t+1} = a_t \cup s_t$ and note that,

$$\begin{aligned}
&r(s_t,a_t) + \gamma V^{\pi^*}(s_{t+1}) \\
&= r(s_t,a_t) + \gamma\Big(r(s_{t+1},a_{t+1}) + \gamma V^{\pi^*}(s_{t+2})\Big) \\
&= r(s_t,a_t) + \gamma\Big(r(s_{t+1},a_{t+1}) + \gamma\Big(r(s_{t+2},a_{t+2}) + \gamma V^{\pi^*}(s_{t+3})\Big)\Big) \\
&\vdots \\
&= \sum_{t\in T}\gamma^t r(s_t,a_t)
\end{aligned}$$

then, $\pi^*(s_t) = \arg\max_{a_t}\sum_{t\in T}\gamma^t r(s_{t+1},a_{t+1})$, which is equivalent to Equation 1. □

[5] Note that while the policy defined in Definition 11 is deterministic, we will treat π as non-deterministic, since this results in more familiar notation.

Note that policy π^* (or indeed any other policy satisfying Definition 11) is defined over the set of all possible observation histories $\mathbf{O}_{\mathbf{A}}^t$. The size of this set is exponential in the number of time steps t that have elapsed and the number of agents M. As a result, computing the optimal policy is computationally intractable for all but the smallest of problems.[6] The algorithms we develop in the next section avoid this problem by exploiting the properties of observation function f mentioned above, making a significant compression of the observation history possible. By doing so, we obtain efficient algorithms that, instead of the optimal solution, compute *bounded approximate* policies, i.e. policies that are guaranteed to be within a (small) factor of the optimal one.

We conclude this section with the following example, which illustrates the model formulated in this section.

Example 1. *Figure 1 shows four discrete time steps of a team of four agents patrolling a small graph. For illustration purposes, $\gamma = 1$ and observation value function f assigns a value to observation at vertex v that is equal to the number of time steps that have elapsed since v was last observed, with a maximum of* 4. *This models an environment in which observations become stale after $\tau = 4$. Moreover, agents can only observe the vertex at which they are currently positioned.*

The size of the vertices in Figure 1 are proportional to the observation value that can be received at their coordinates in the next time step. Thus, the amount of observation value that the agents receive as a team in each of these four time steps is 4, 12, 16 *and* 13 *respectively. Note that the decision by agent* 3 *to go back to its previous position at* $t = 4$ *(Figure 1(d)) results in a suboptimal observation value (at least, over these four time steps).*

4. Near-Optimal Non-Myopic Patrolling

Given the model formulated in the previous section, we first develop an approximate non-myopic algorithm for the single-agent case (Section 4.1), which is later used as a building block for the computing multi-agent policies (Section 4.2). For the purpose of clarity, we only discuss the algorithmic steps in this section — the theoretical analysis of these algorithms is deferred to Section 5.

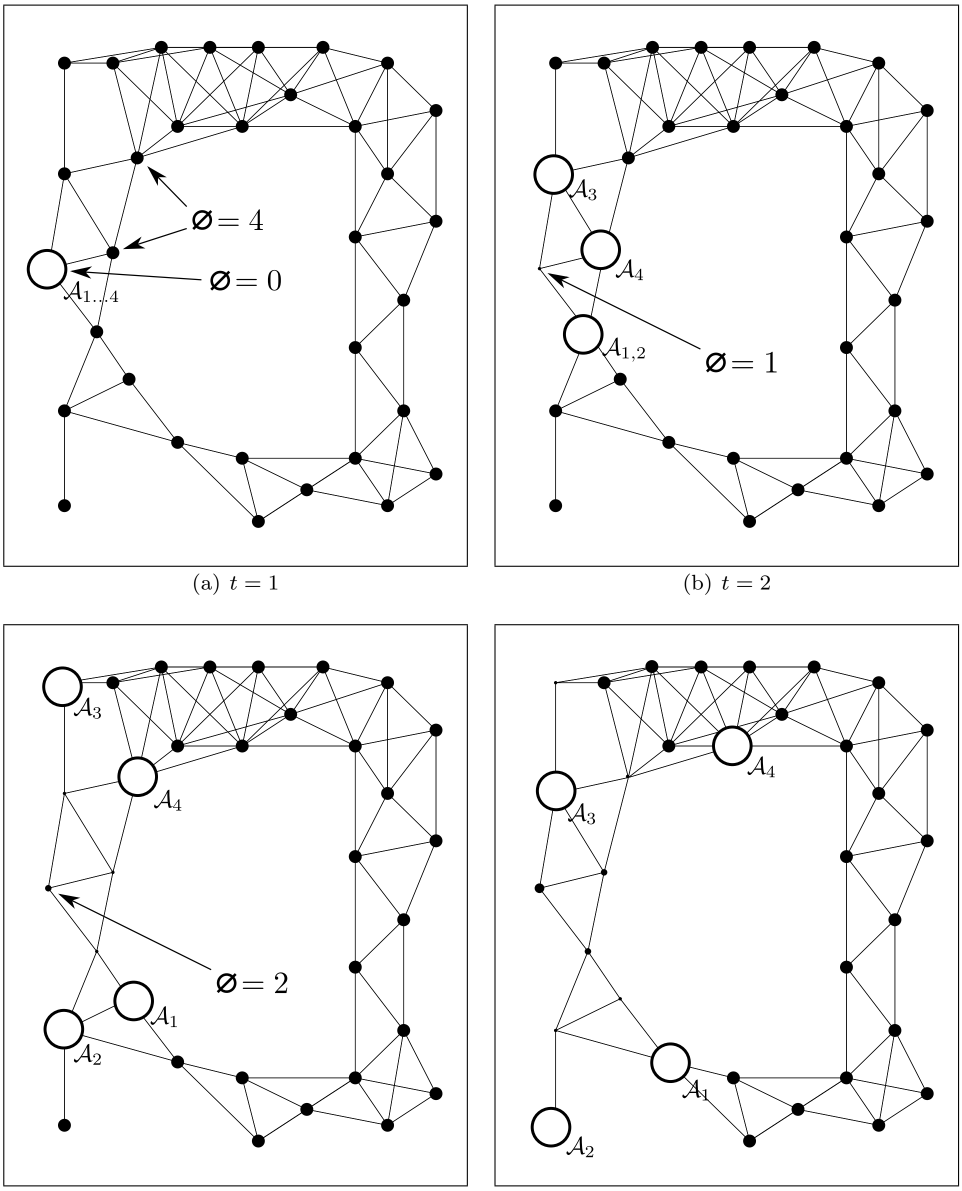
Before continuing, we formally define a number of the most important concepts used in the formalisation of the single and multi-agent algorithms:

Definition 15 (Cluster)**.** *A cluster $C = (V_C, E_C)$ is a connected subgraph of layout graph G.*

Definition 16 (Atomic Cluster)**.** *An atomic cluster is a cluster with a diameter less or equal to D and is not subdivided into smaller clusters.*

Definition 17 (Transit Node)**.** *A transit node is a maximal connected subgraph of layout graph G whose vertices lie on the boundary between one or more clusters. More*

[6] For example, consider the scenario from Example 1. The vertices in the layout graph G have an average degree of 4.2 (33 vertices and 70 edges). Therefore, after only 10 time steps with 4 agents, there are $(4.2^{10})^4 \approx 8 \cdot 10^{24}$ different paths the agents could have jointly taken, with an even greater number of possible observation histories.



(a) $t = 1$

(b) $t = 2$

(c) $t = 3$

(d) $t = 4$

Figure 1: Four discrete time steps of a team of agents $\mathbf{A} = \{1, 2, 3, 4\}$ moving in an environment whose layout is defined by a graph $G = (V, E)$. The diameter of the vertices (indicated by the $\varnothing$ symbol) is proportional to the observation value that is received by moving there in the next time step.

formally, if $T = (V_T, E_T)$ is a transit node, $(v, v') \in E_T$ iff $v \in V_C$ and $v' \in V_{C'}$, where $C = (V_C, E_C)$ and $C' = (V_{C'}, E_{C'})$ are distinct clusters.

Definition 18 (Cluster graph)**.** *A cluster graph $G[\mathcal{C}] = ((\mathbf{C} \cup \mathbf{T}), E_{\mathcal{C}})$ is a bipartite graph of transit nodes and clusters. An edge exists between a transit node T and a cluster C iff at least one vertex in T is adjacent to C in layout graph G.*

Definition 19 (Subpatrol)**.** *A subpatrol (T, C, T') is a path from a transit node T, through a cluster C to a transit node T' (it is possible that $T = T'$). A subpatrol originates at a vertex $v \in V_T \cap V_C$ and terminates at a vertex $v' \in V_{T'} \cap V_C$.*

Definition 20 (Patrol)**.** *A patrol is an infinitely long path through layout graph G obtained by concatenating subpatrols $(T^{(1)}, C^{(1)}, T^{(2)}), (T^{(2)}, C^{(2)}, T^{(3)}), \ldots$.*

4.1. The Single-Agent Algorithm

The prime objective of the single-agent algorithm is to compute a patrol along which a large amount of observation value can be collected (we will shortly quantify "large amount") in a computationally efficient way. It does so through employing a *divide and conquer* strategy: recursively dividing the layout graph into smaller and smaller components (clusters), until the patrolling problem within those clusters can be efficiently solved. As a result, the exponentially large set of possible observation histories $\mathbf{O}^t_{\mathbf{A}}$ in Equation 3 is compressed into a more manageably sized set of *world states*, such that it becomes computationally feasible to conduct searches over the space of patrols. This compression comes at a cost, as the resulting patrol is not optimal. However, as we will show in Section 5, it is possible to derive theoretical guarantees on its quality.

The single-agent algorithm exploits the following properties of the problem defined in Section 3:

1. We exploit the *locality* property of an observation value function f (Property 3), by partitioning graph G into a set of clusters $\mathbf{C} = \{C_1, \ldots, C_{|\mathbf{C}|}\}$ (Definition 15), such that observations taken in different clusters are independent. The problem of maximising observation value can then be solved independently for each cluster. This property is exploited in a similar way by (Singh et al., 2009).
2. We exploit the *temporality* property of f (Property 4) by discarding observations older than τ. These observations are independent of observations taken now or in the future, and can thus safely be ignored.
3. Just as space is divided in clusters, time is divided into non-overlapping intervals of length $B \in \mathbb{N}$. During each interval, an agent enters a cluster, patrols it, and moves on to the next. The path taken within a cluster is called a *subpatrol* (Definition 19). Parameter B is chosen such that the agent can collect a *reasonable* amount of observation value within the cluster.

By exploiting these properties, we can define a divide and conquer algorithm, which is defined by the following three operations (see Figure 2 for a visual overview):

Divide Exploit the locality property by recursively subdividing the problem into more manageable subproblems. This is done by dividing a (sub)problem with layout graph G into a set of clusters $\mathbf{C} = \{C_1, \ldots, C_{|\mathbf{C}|}\}$ such that the distance between them is sufficient to ensure observations taken in different clusters are independent (cf. Singh et al. (2009)).

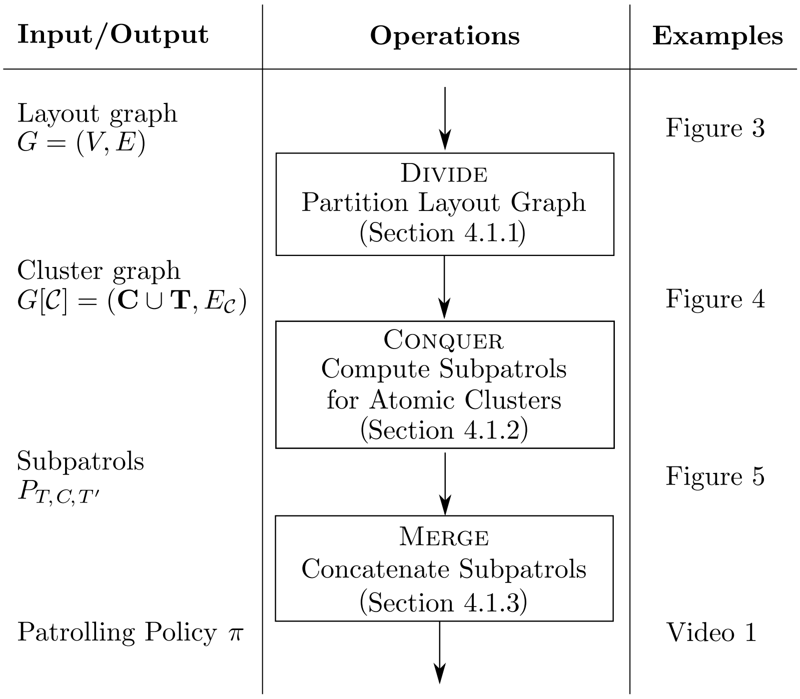


Figure 2: An overview of the single-agent algorithm.

By dividing the layout graph, we have transformed the problem from one in which the agents move from vertex to vertex (of which there are many), into one where agents move from cluster to cluster (of which there are few). The graph that captures these new topological relations is a bipartite graph $G[\mathcal{C}]$ called a *cluster graph* (Definition 18).

Conquer When a cluster obtained in the previous step is small enough, the patrolling problem within that cluster can be solved. Here "small enough" means that the diameter, $\mathrm{diam}(C)$, of a cluster C is sufficiently small to allow an agent to collect a large amount of observation value within B time steps. In Section 5 we will quantify the relation between $\mathrm{diam}(C)$ and B that achieves this.

By solving the patrolling problem for the atomic clusters, we obtain a set of *subpatrols* (Definition 19) within those clusters. A subpatrol is a path within an atomic cluster of length B or less. Each subpatrol corresponds to a movement allowed within the cluster graph $G[\mathcal{C}]$ obtained from the Divide operation.

Merge The Merge operation reverses the recursion by concatenating the subpatrols of subproblems into a (sub)patrol of a higher level (sub)problem. It does so by constructing a Markov Decision Process (MDP) in which states represent the position of the agent and the time λ_C each cluster C was last visited. A solution to this MDP is a policy that instructs the agent which cluster to patrol next, given its current position and the last time the clusters were visited. When the Merge operation arrives at the root of the tree, which corresponds to the patrolling problem in the original layout graph, it yields the final output of the algorithm: a patrol for a single agent.

Video 1. *A video demonstrating each operation, as well as the complete algorithm can be found at* `http://player.vimeo.com/video/20220136`*.*

In what follows, we first provide detailed algorithms for each operation and then construct the complete algorithm in Section 4.1.4. As will prove in Section 5.2, our algorithm provides a good trade-off between computation and solution quality, while simultaneously providing performance guarantees. However, we wish to note that different algorithms may be used in the Divide and Conquer operations to yield subpatrols that strike a different balance between computation and quality. We will discuss a (non-exhaustive) set of alternatives where appropriate.

4.1.1. Divide*: Partition the Layout Graph*

The objective of the Divide operation of the algorithm can be defined as:

Objective 1. *Partition graph G into a set of clusters $\mathbf{C} = \{C_1, \ldots, C_{|\mathbf{C}|}\}$ ($|\mathbf{C}| \leq \mathbf{C}_{\max}$), while minimising the maximum diameter across all clusters. If $\mathrm{diam}(G) \leq D$, graph G is not further subdivided as it is small enough such that a path of length B can visit at least k vertices ($k \ll |V_C|$). Furthermore, ensure that vertices in different clusters are at least a distance of δ apart.*

The reason for imposing the constraints on the diameter of atomic clusters is that it allows us to later derive theoretical guarantees on the quality of the computed solution, as will become clear in Section 5. Parameter $\mathbf{C}_{\max}$ determines how many subproblems may be created until further subdivision is needed. The resulting partitions are

Algorithm 1 The Divide algorithm for clustering layout graph G into cluster graph $G[\mathcal{C}]$.

```
Require: $G = (V, E)$: the layout graph
Require: $D$: the maximum diameter of an atomic cluster
Require: $\mathbf{C}_{\max}$: the maximum number of clusters $G$ is divided into
Ensure: Cluster graph $G[\mathcal{C}] = ((\mathbf{C} \cup \mathbf{T}), E_{\mathcal{C}})$, such that:
    • $\mathbf{C} = \{C_1, \ldots, C_{|\mathbf{C}|}\}$ is a set of clusters;
    • $\mathbf{T} = \{T_1, \ldots, T_{|\mathbf{T}|}\}$ is a set of transit nodes;
    • $\forall v \in C_i, \forall v' \in C_j, i \neq j : \mathrm{d}(v, v') \geq \delta$, i.e. vertices within a cluster are at least a
      distance $\delta$ away from vertices in other clusters;
    • $|\mathbf{C}| \leq \mathbf{C}_{\max}$
 1: procedure Divide($G, D, \mathbf{C}_{\max}$)
 2:     if $\mathrm{diam}(G) \leq D$ then
 3:         return $G[\mathcal{C}] = (\{G\}, \emptyset)$          ▷ Lowest level of division has been reached
 4:     end if
        ▷ Step 1: Partition graph G:
 5:     $\mathbf{C} \leftarrow$ Cluster($G, D, \mathbf{C}_{\max}$)          ▷ Cluster $G$ in at most $\mathbf{C}_{\max}$ clusters
        ▷ Step 2: Identify transit nodes $\mathbf{T}$ by detecting vertices $V_B$ that lie on the boundary
        between two clusters:
 6:     $V_B \leftarrow \bigcup_{C \in \mathbf{C}} \{v \in C \mid \exists v' \in V : (v, v') \in E\}$
 7:     $\mathbf{T} \leftarrow$ ConnectedComponents($G[V_B]$)
        ▷ Compute edges $E_{\mathcal{C}}$ of graph $G[\mathcal{C}]$ that encode connections between clusters and transit
        nodes:
 8:     $E_{\mathcal{C}} \leftarrow \{(C, T) \mid C \in \mathbf{C}, T \in \mathbf{T}, \exists v \in C, \exists v' \in T : (v, v') \in E\}$
        ▷ Step 3: Strip away vertices less than $\frac{1}{2}\delta$ away from vertices in different clusters:
 9:     $V_\delta \leftarrow \varnothing$
10:     $V_\delta = \bigcup_{C \in \mathbf{C}} \{v \in C \mid \exists v' \in V \setminus C : \mathrm{d}(v, v') \leq \frac{1}{2}\delta\}$
11:     for $C \in \mathbf{C}$ do
12:         $C \leftarrow C \setminus V_\delta$
13:     end for
14:     return $G[\mathcal{C}] = ((\mathbf{C} \cup \mathbf{T}), E_{\mathcal{C}})$
15: end procedure
```

transformed into a cluster graph $G[\mathcal{C}]$ that encodes how agents can move between clusters.

In terms of alternative algorithms, any (recursive) decomposition of the graph in principle reduces the complexity of the problem that needs to be solved by operations Divide and Conquer. However, the quality of the solution and the theoretical guarantees on the solution depend on the decomposition of the graph, which in turn depends on the type of graph and the graph clustering algorithm used. Fortunately, the literature on graph clustering is abundant. As a result, our algorithm can be applied to a wide spectrum of graphs by selecting an appropriate clustering algorithm that satisfies the properties mentioned in Objective 1.

Algorithm 1 performs the necessary steps. First, it checks whether the graph is sufficiently small and does not need further subdivision (line 2). If this is not the case, it partitions the graph into a set of clusters $\mathbf{C} = \{C_1, \ldots, C_{|\mathbf{C}|}\}$ (line 5). There are many different ways of doing this, however, we are interested in obtaining a minimum number of atomic clusters that satisfy the maximum diameter requirement. With this in mind, we use the algorithm proposed by Edachery et al. (1999) as a subroutine, which we will refer to as Cluster$(G, D, \mathbf{C}_{\max})$.

As we will discuss in further detail in Section 5.2, this algorithm is approximate. It satisfies the minimum diameter requirement of each cluster, but requires more than the minimum (optimal) number of clusters to do so. We choose this algorithm for its computational efficiency. However, depending on the type of graph, there might be other algorithms worth investigating. For an overview of different approaches, we refer the reader to Schaeffer (2007). In addition, this algorithm is only applicable to undirected graphs. This is the reason for focusing exclusively on undirected graph in this paper (see Section 3.1). To the best of our knowledge, there are no graph clustering algorithms for directed graphs that provide a bound on the diameter of the resulting clusters. There are, however, several algorithms for clustering directed graphs without these guarantees (e.g. Meila and Pentney (2007); Satuluri and Parthasarathy (2011)). As a result, these algorithms may be used the Divide operation of our algorithm, but lead to the loss of the performance guarantees described in Section 5.2.

Now, the Cluster$(G, D, \mathbf{C}_{\max})$ algorithm takes as input a graph G, maximum diameter D and the maximum number of clusters $\mathbf{C}_{\max}$ and returns a set of clusters $\mathbf{C}$, such that $|\mathbf{C}| \leq \mathbf{C}_{\max}$ while attempting to reduce the maximum diameter of these clusters to D. In more detail, Cluster solves a slight variation of the *pairwise clustering problem*. The pairwise clustering problem involves finding the smallest n-clustering of G, such that each of the n clusters has a diameter smaller than D.

In the second step (lines 6–7), Algorithm 1 identifies the *transit nodes* $\mathbf{T}$ between clusters $\mathbf{C}$. These transit nodes are connected components of graph G that lie on the boundary of the clusters. To compute these, the algorithm identifies the boundary vertices V_B of the clusters, i.e. those vertices that have at least one adjacent vertex in a different cluster. $G[V_B]$ is the subgraph induced by V_B, so the set of transit nodes $\mathbf{T} = \{T_1, \ldots, T_{|\mathbf{T}|}\}$ corresponds to the set of connected components in $G[V_B]$.

The third step (lines 9–13) of the algorithm ensures independence of observations made in different clusters, by removing all vertices that are less than $1/2\delta$ away from vertices in other clusters (Property 3).

The resulting clusters, transit nodes and their connections are represented as a bipartite graph $G[\mathcal{C}] = ((\mathbf{C} \cup \mathbf{T}), E_{\mathcal{C}})$. The set of edges $E_{\mathcal{C}}$ of $G[\mathcal{C}]$ contains an edge (C, T) between a cluster $C \in \mathbf{C}$ and a transit node $T \in \mathbf{T}$ if and only if the original graph G contains an edge e that has endpoints in both C and T (line 6). This graph represents valid high-level movements between clusters, and is used in the Merge operation to define the actions of the agent within the MDP.

The following example illustrates the operation of Divide:

Example 2. *Figure 3 shows a single-level clustering (i.e. non-recursive) of the layout graph G of the Agents, Interaction and Complexity (AIC) lab at the University of Southampton with $\delta = 0$ (i.e. observations made at different spatial coordinates are independent) and $D = 25$. This results in six clusters and seven transit nodes. Figure 4 depicts the cluster graph $G[\mathcal{C}]$ representing the interconnections between the clusters and transit nodes in Figure 3.*

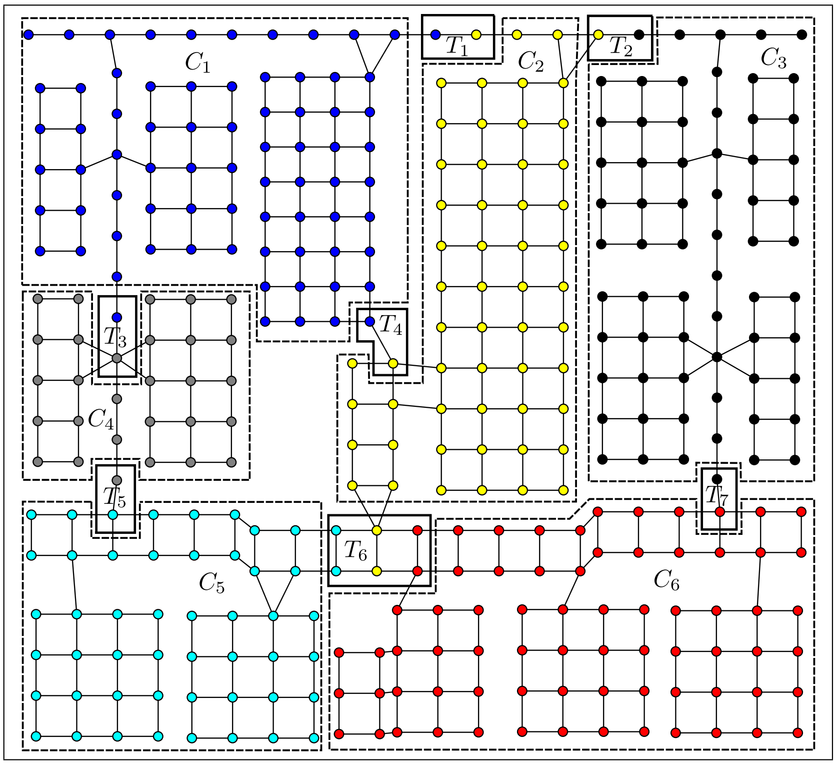


Figure 3: The clusters and transit nodes obtained by applying the DIVIDE operation on the layout graph of the AIC lab with $\mathbf{C}_{\max} = 6$ and $D = 25$. The dotted lines indicate the boundaries of the clusters $\mathbf{C}$ and the solid lines outline the seven transit nodes $\mathbf{T}$ that connect the clusters. The original layout graph has 350 vertices and 529 edges.

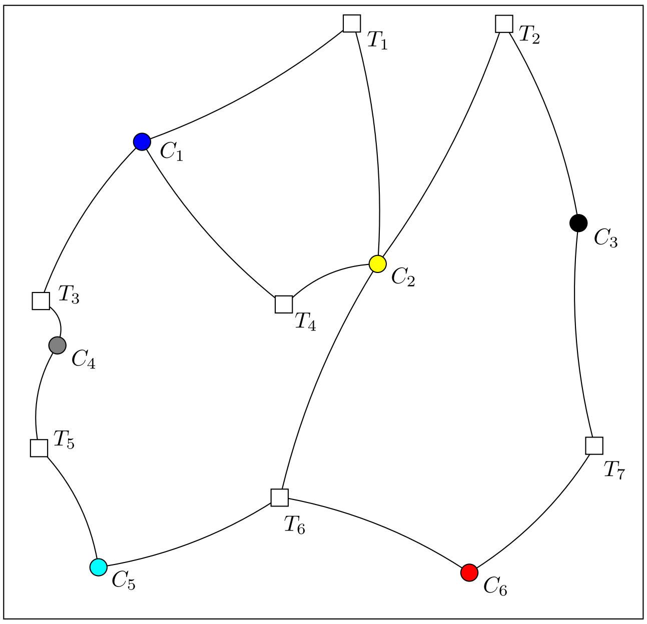


Figure 4: The cluster graph $G[\mathcal{C}]$ that represents the topological relations between the clusters (coloured circles) and transit nodes (white squares) in Figure 3.

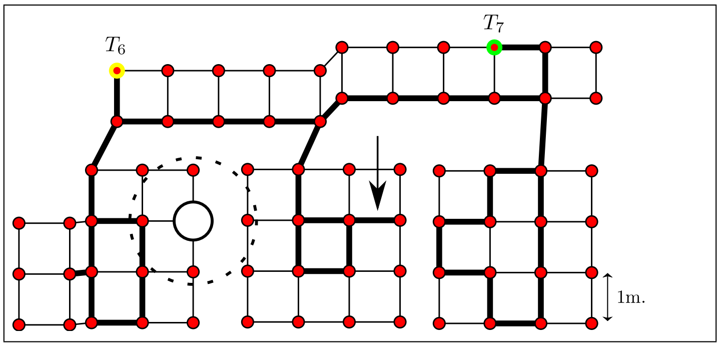


Figure 5: Example 4: A patrol within cluster C_6 from transit node T_7 to T_6 for an information gathering agent with an observation radius of 1.5m. The arrow indicates a redundant move of the agent, i.e. one that, when omitted, yields the same amount of observation value.

4.1.2. Conquer*: Compute Subpatrols in Atomic Clusters*

By recursively clustering the layout graph, we have now decomposed the problem of finding a path of high value through the original (large) graph to a set of easier independent subproblems that involve finding subpatrols within the small atomic clusters. Recall that these subproblems were made independent by ensuring that observations made in different clusters are independent and by limiting the time available to patrol an atomic cluster to B. Based on this, we can now state the objective of the Conquer operation of the algorithm:

Objective 2. *For each atomic cluster, compute* subpatrols *of length of at most B between each pair of the cluster's adjacent* transit nodes*.*

The reason for this objective is that the agent's high-level movement is constrained by graph $G[\mathcal{C}]$, in which agents enter and exit clusters through the clusters' adjacent transit nodes. A subpatrol is the path in layout graph G that corresponds to such a movement.

Example 3. *For cluster C_6 in Figure 4 there are four possible subpatrols: (T_6, C_6, T_6), (T_6, C_6, T_7), (T_7, C_6, T_6), (T_7, C_6, T_7)*

In more detail, the problem is now to find a sequence of vertices within an atomic cluster that maximises the value of observations, subject to a finite time budget B. Since this problem is NP-complete (Karp, 1982),[7] solving this problem optimally is computationally intractable for arbitrary clusters. Therefore, instead, the patrolling subroutine is chosen to be approximate. That is, it computes subpatrols of near-optimal value that are shorter than B. This subroutine is based on the near-optimal sensor placement algorithm of Krause et al. (2006) (which was also used in Singh et al. (2009)).

Algorithm 2 shows the necessary steps for computing these subpatrols. For each cluster, this algorithm is used to compute subpatrols between each pair of the cluster's adjacent transit nodes.

In more detail, for a given cluster C, entry T and exit T', Algorithm 2 proceeds in two steps. First (lines 2–6), it orders the vertices of C by their incremental value — the value obtained by greedily adding the observations O_v made at v to the already selected set $\mathbf{O}$, such that the incremental value $f(\mathbf{O} \cup O_v) - f(\mathbf{O})$ of observations collected at v is maximised. This results in a sequence of vertices $s_G = (v^{(1)}, \ldots, v^{(|V_C|)})$. In the second step (lines 7–14), it seeks to find a subpatrol $P_{T,C,T'}$ from T to T' with a length of at most B and maximises the length n of the prefix of s_G (i.e. its first n elements) that is visited along the path. This problem can be encoded as an instance of the TSP where we seek to find a minimum cost (in terms of time) cycle $(T, v^{(1)}, \ldots, v^{(n)}, T', T)$. Here, the time of moving between two vertices v_i and v_j equals the length of the shortest path between them, and the time taken by moving between T and T' equals 0. Since solving the TSP itself is NP-complete (Karp, 1982), we use the heuristic algorithm by Christofides (1976), which has the best known performance guarantee ($\frac{3}{2}$) of any approximate algorithm for the TSP (Gross and Yellen, 1999).

Example 4. *Consider an agent (the white circle in Figure 5) that is capable of perfectly observing all vertices within a sensing radius of $1.5m$ (the dashed circle) and let value*

[7] It is easy to see that the decision variant of the TSP can be reduced to this problem.

Algorithm 2 The Conquer algorithm for computing a subpatrol of atomic cluster C from entry T to exit T'.

```
Require: $C = (V_C, E_C)$: a cluster
Require: $f$: the observation value function
Require: $T$: the entry transit node
Require: $T'$: the exit transit node
Require: $B$: the budget
Ensure: $P_{T,C,T'}$: a subpatrol in $C$ from $T$ to $T'$ taking $c(P_{T,C,T'}) \leq B$ time steps.
procedure CONQUER($C, f, T, T', B$)
    ▷ Step 1: Sort vertices by their incremental observation value:
    $s_G \leftarrow ()$
    while $V_C \setminus s_G \neq \emptyset$ do
        Let $O_v$ be the observations made at $v$, and $\mathbf{O} \leftarrow \bigcup_{v \in s_G} O_v$.
        $s_G \leftarrow s_G || \operatorname*{arg\,max}_{v \in C \setminus s_G} f(\mathbf{O} \cup O_v) - f(\mathbf{O})$
    end while
    ▷ Step 2: Find the maximum $n$ such that the time taken by traversing the subpatrol that
    visits the first $n$ elements of $s_G$ does not exceed $B$:
    $n \leftarrow 0$
    $P' \leftarrow (T, T')$
    repeat
        $n \leftarrow n + 1$
        $P_{T,C,T'} \leftarrow P'$
        $s_G^n \leftarrow \text{prefix}(s_G, n)$                    ▷ Select first $n$ elements of $s_G$
        $P' \leftarrow TSP(T, s_G^n, T')$
    until $c(P') > B$
    return $P_{T,C,T'}$
end procedure
```

function f be defined in terms of the number of vertices that are observed. Figure 5 shows the subpatrol P_{T_7,C_6,T_6} through C_6 in the graph in 3 computed by Algorithm 2 with $B = 50$.

Note that this patrol is not optimal, in the sense that the same number of vertices (i.e. all of them) could have been observed within 44 time steps (instead of 46) by removing the path element indicated by an arrow.

4.1.3. Merge*: Concatenate Subpatrols*

The third and final operation of the algorithm achieves the following objective:

Objective 3. *For a given (sub)problem identified by* Divide*, compute a patrol (see Definition 20) by concatenating the subpatrols in lower level clusters such that the observation value received along that patrol is maximised (subject to the computed subpatrols).*

Thus, using Merge we start at the level of atomic clusters by concatenating subpatrols computed by Conquer and move up to higher level clusters until the patrolling problem involving the entire layout graph (i.e. root-level cluster) has been solved.

To achieve this objective, Merge (Algorithm 3) solves a MDP over the patrolling problem in clustered graph $G[\mathcal{C}]$. This (deterministic) MDP is a 4-tuple $(S, A, \delta(\cdot,\cdot), R(\cdot,\cdot))$ where:

Algorithm 3 The Merge algorithm for solving the patrolling problem within a non-atomic cluster.

```
Require: $G[\mathcal{C}]$: a cluster graph of graph $G$
Require: subpatrols: the set of all subpatrols for all clusters in $G[\mathcal{C}]$
Require: $B_G$: the budgeted time for patrolling $G$
Require: entry: the vertex where the agent enters $G$
Require: exit: the vertex where the agent should exit $G$ within $B_G$ time steps or $\emptyset$
    if $B_G = \infty$
Ensure: A (sub)patrol for $G$ of length no greater than $B_G$ starting at entry and
    terminating at exit
procedure MERGE($G[\mathcal{C}]$, subpatrols, $B_G$, entry, exit)
    return The patrol obtained by concatenating subpatrols using the MDP for
    $G[\mathcal{C}]$ defined in this section with parameters $\gamma$, $B_G$, entry, and exit.
end procedure
```

- S is a set of states encoding the current position of the agent and the time each cluster was last visited.
- A is a set of actions. In this context, each action in this set corresponds to following a subpatrol (computed by Conquer) that start from the agent's current position, a transit node, through a cluster to another transit node.
- $s' = \delta(s, a)$ is the state obtained from following subpatrol a in state s. Thus, δ is a deterministic transition function.
- $R(s, a)$ is the observation value received by following subpatrol a in state s.

In what follows, we discuss each item in more detail.

State Space. The state space S consists of *patrolling states*:

Definition 21 (Patrolling State)**.** *A patrolling state is a triple* $(T, \boldsymbol{\lambda}, B_r)$ *where:*

1. $T \in \mathbf{T}$ *is the agent's position, which is one of the transit nodes.*
2. $\boldsymbol{\lambda} = [\lambda_{C_1}, \ldots, \lambda_{C_{|\mathbf{C}|}}]$ *is a vector in which each element is the number of time steps since each cluster was last patrolled.*
3. $B_r \in \mathbb{N}^+$ *is the remaining budget for patrolling the graph.*

By exploiting the *temporality* property (Property 4), we know that observations made longer than τ time steps ago are independent of new observations. Therefore, the entries of $\boldsymbol{\lambda}$ never exceed τ.

Furthermore, keeping track of the exact number of time steps since a cluster was last visited yields $\tau^{|\mathbf{C}|}$ distinct possible states, causing the problem to become intractable for even a very small number of clusters or a small value of τ. However, by exploiting the knowledge that an agent takes B time steps to patrol an atomic cluster, and if we furthermore choose B to be a divisor of τ, we can ensure that $\lambda_C \in \{0, B, 2B, \ldots, \tau\}$. This drastically reduces the number of distinct possible visit states of a single cluster from $\tau + 1$ to $\frac{\tau}{B} + 1$. Thus, combining this result with the number of possible positions for the agent $|\mathbf{T}|$, the state space for a single agent consists of $|\mathbf{T}|(\frac{\tau}{B} + 1)^{|\mathbf{C}|}$ states. We discuss the effect of this on computational complexity in Section 5.2.

Action Space. The action space of the MDP consists of *patrolling actions* which are defined as follows:

Definition 22 (Patrolling Action)**.** *A patrolling action is a sequence (T, C, T') where $C \in \mathbf{C}$ is a cluster and $(T, T') \in \mathbf{T}^2$ are transit nodes. A patrolling action corresponds to a subpatrol starting from T which moves through C and terminates at T'.*

The set of valid actions $A(s)$ for state s is defined as:

Definition 23 (Valid Patrolling Action)**.** *Let s be the state $(T, \boldsymbol{\lambda}, B_r)$. The set of valid patrolling actions $A(s)$ for s is:*

$$A(s) = \{(T, C, T') \mid c(P_{T,C,T'}) \leq B_r \wedge C \in adj_{G[\mathcal{C}]}(T) \wedge T' \in adj_{G[\mathcal{C}]}(C)\}$$

Thus, the set of available actions contains all subpatrols starting at T for which the agent has sufficient remaining budget.

Example 5. *The valid patrolling actions in the cluster graph shown in Figure 4 for state $s = (T_1, \cdot, \infty)$ are $A(s) = \{(T_1, C_1, T_1), (T_1, C_1, T_3), (T_1, C_1, T_4),$ $(T_1, C_2, T_1), (T_1, C_2, T_2), (T_1, C_2, T_4), (T_1, C_2, T_6)\}$.*

Transition Function. The transition function formalises how the state of the MDP transitions under a given action a, and is defined as:

Definition 24 (Patrolling Transition Function)**.** *The patrolling transition function is a deterministic function $\delta(s, a) = s'$ which formalises the transition from state $s = (T, [\lambda_{C_1}, \ldots, \lambda_{C_{|\mathbf{C}|}}], B_r)$ under valid action (see Definition 23) $a = (T, C_i, T') \in A(s)$:*

$$\delta(s, a) = (T', [\hat{\lambda}_{C_1}, \ldots, \hat{\lambda}_{C_{i-1}}, 0, \hat{\lambda}_{C_{i+1}}, \ldots, \hat{\lambda}_{C_{|\mathbf{C}|}}], B_r - c(P_{T,C_i,T'}))$$

where $\hat{\lambda}_{C_j} = \min(\lambda_{C_j} + c(P_{T,C_i,T'}), \tau)$.

Thus, the patrolling transition function states that when an agent patrols a cluster C_i by performing action $a = (T, C, T')$, the process transitions to state s', in which the agent is positioned at T' and the visitation time of the cluster λ_C is reset to 0. Furthermore, since the agent takes a number of time steps equal to the length of the subpatrol to visit a cluster, the visitation times of clusters C_j $(j \neq i)$ are incremented by the length of the patrol $c(P_{T,C_i,T'})$, if not already equal to τ, and the remaining budget is decreased by $c(P_{T,C_i,T'})$.

This transition function enables us to further reduce the size of the state space defined earlier, by only considering the states $S_r(\overline{s})$ that are reachable from the initial state $\overline{s} = (entry, [\tau, \ldots, \tau])$ in which none of the states have been visited yet and the agent is at the *entry* transition node (see Algorithm 3). As an example of a state that cannot be reached in the setting of Figure 3, consider $(T_1, [\tau, \tau, 0, \tau, \tau, \tau], \cdot)$ which encodes that cluster C_3 was just patrolled by the agent and then moved to a transit node that is inaccessible from C_3. The set of states $S_r(s)$ reachable from a state s is defined as:

$$S_r(s) = \{s\} \cup \bigcup_{a \in A(s)} S_r(\delta(s, a)) \tag{4}$$

Reward Function. The reward function of the MDP is defined as follows:

Definition 25 (Patrolling Reward Function)**.** *The reward $R(s,a)$ received for performing patrolling action (T,C,T') in state $s = (T, [\lambda_{C_1}, \ldots, \lambda_{C_{|\mathbf{C}|}}], B_r)$ is given by:*

$$R\left((\mathbf{T}, [\lambda_{C_1}, \ldots, \lambda_{C_{|\mathbf{C}|}}], B_r), P_{T,C,T'}\right) = \begin{cases} I(C, \lambda_C, T, T') & \textit{if } B_r \leq c(P_{T,C,T'}) \\ 0 & \textit{if } A(\delta(s,a)) = \emptyset \wedge T' = \textit{exit} \\ -\infty & \textit{otherwise} \end{cases} \tag{5}$$

where $I(C, \lambda_C, T, T')$ is the value of the observations made along subpatrol $P_{T,C,T'}$, given that cluster C was visited λ_C time steps ago and is given by:

$$I(C, \lambda_C, T, T') \equiv \sum_{i=1}^{n} \gamma^{t_i} \cdot \left[f\left(O_{v^{(i)}} \cup \bigcup_{j=1}^{i-1} O_{v^{(j)}} \cup \mathbf{O}_C^{-\lambda_C} \right) - f\left(\bigcup_{j=1}^{i-1} O_{v^{(j)}} \cup \mathbf{O}_C^{-\lambda_C} \right) \right] \tag{6}$$

Here, $\mathbf{O}_C^{-\lambda_C}$ denotes the set of observations made λ_C time steps ago at each vertex of C, the set O_v denotes the observations made at v (as before), and t_i is the time at which $v^{(i)}$ is visited, which is the time it takes to arrive at $v^{(i)}$ traversing subpatrol $P_{T,C,T'}$:[8]

$$t_i = \sum_{j=1}^{i-1} \mathrm{d}_G\left(v^{(j)}, v^{(j+1)}\right)$$

A couple of important points need to be made about this reward function. First, it is unknown which subpatrol was previously used to visit C, we assume that all vertices of C were visited simultaneously λ_C time steps ago, at which point a set of observations was made, which we denote as $\mathbf{O}_{\mathbf{C}}^{-\lambda_C}$. Thus, the incremental value of the observations made along $P_{T,C,T'}$ with respect to $\mathbf{O}_{\mathbf{C}}^{-\lambda_C}$ yields a conservative estimate (i.e. lower bound) on the true reward for action (T,C,T'), since observation value function f is strictly decreasing with the time elapsed since observations were made.

Second, note that the reward $R(s,a)$ of performing $a = (T,C,T')$ in state $s = (T, [\lambda_{C_1}, \ldots, \lambda_{C_{|\mathbf{C}|}}])$ is the sum of incremental values of observations made along subpatrol $P_{T,C,T'} = (T, v^{(1)}, \ldots, v^{(n)}, T')$. Thus, $R(s,a)$ depends exclusively on the visitation state λ_{C_i} of cluster C and the entry T and exit T' of the subpatrol used to visit C; the visitation states of the clusters other than C are irrelevant for computing the action's reward. Therefore, we defined an auxiliary function $I(C, \lambda_C, T, T')$ that computes the value of a subpatrol with only the relevant parameters.

Third, the reward function ensures that the agent arrives at the *exit* transit node (see Algorithm 3) before the remaining budget B_r has been exhausted. This is done by assigning the value $-\infty$ to transitions to states in which this constraint is not met.

Solving the MDP. A solution of the MDP $(S, A, \delta(\cdot,\cdot), R(\cdot,\cdot))$ defined above is a policy of the form $\pi(s) = a$ that, for every possible state $s \in S_r(s_0)$ reachable from initial

[8] Recall that $\mathrm{d}_G(v, v')$ is the length of the shortest path in G from v to v'.

state s_0, yields action a that maximises the expected discounted reward. This policy is characterised by the following equations:

$$\pi(s) = \arg\max_a \{R(s,a) + \hat{\gamma} V^{\pi}(\delta(s,a))\} \tag{7}$$

$$V^{\pi}(s) = R(s, \pi(s)) + \hat{\gamma} V^{\pi}(\delta(s, \pi(s))) \tag{8}$$

Here, $V^{\pi}(s)$ is referred to as the *state value* of s under policy π, which equals the discounted sum of rewards to be received by following policy π from state s. Many algorithms can be used to compute policy π, such as policy iteration (Howard, 1960), modified policy iteration (Puterman and Shin, 1978), and prioritised sweeping (Moore and Atkeson, 1993). However, one of the simplest is value iteration (Puterman, 1994). This algorithm repeatedly applies the following update rule:

$$V(s) = \max_a \left\{R(s,a) + \hat{\gamma} V(\delta(s,a))\right\} \tag{9}$$

until the maximum difference between any two successive state values falls below a predefined threshold $\epsilon > 0$. After termination, the value of each state under policy π is within ϵ of the optimal value. This policy is returned by the MERGE operation (see Algorithm 3). When executed, it yields the desired (sub)patrol for the given graph.

4.1.4. Putting it all together: The complete algorithm

Now that we have defined all three necessary operations, we can now construct the single-agent patrolling algorithm (Algorithm 4). This algorithm calls the COMPUTESUBPATROLDNC as subroutine with an infinite budget, since we require a *continuous* patrol from *entry* at the transit node at which the agent is located.[9] Algorithm 5 shows the operation of the recursive COMPUTESUBPATROLDNC which performs the actual computation.

First, it checks whether the graph is small enough for solving the problem outright with CONQUER (lines 2–4). If not, it DIVIDEs the problem into smaller clusters. Then, using a recursive call to itself, COMPUTESUBPATROLDNC computes subpatrols in each of the identified clusters. The allocated budget of these subpatrols is computed using the COMPUTEBUDGET subroutine in Algorithm 6, which ensures that clusters on the same level are given equal budget. Finally, the subpatrols for clusters **C** are MERGEd into a subpatrol for graph G (line 14).

When the call to COMPUTESUBPATROLDNC in Algorithm 4 returns, it has computed a patrolling policy (Definition 11), which, when executed, yields the desired patrol in the full layout graph G.

Consider the following example which explains the operation of the complete algorithm.

Example 6. *Consider the layout graph in Figure 6. This graph is obtained by connecting nine copies of the AIC lab (Figure 3) in a three by three grid. Using* DIVIDE *with $D = 20$, the graph is first divided into six top-level clusters. Each of these top-level clusters are then Divided again into six second-level clusters. Finally, some of*

[9] If the agent is not located at a transit node, we compute the value of starting at all transit nodes and discount this value with the length of the shortest path from the agent's starting location to each of these nodes. We then choose the best starting transit node.

Algorithm 4 The single-agent algorithm for computing a patrol of graph G in a divide and conquer fashion.

```
Require: $G = (V, E)$: a layout graph
Require: $f$: the observation value function
Require: $\gamma \in [0, 1\rangle$: the discount factor
Require: $B \in \mathbb{N}$: the maximum time that may be spent in an atomic cluster.
Require: $D \in \mathbb{N}$: the maximum diameter of an atomic cluster.
Require: $entry \in V$: the starting location of the agent
Require: $\mathbf{C}_{\max}$: the maximum number of clusters in which $G$ may be divided.
Ensure: a patrol for graph $G$
procedure computePatrolDnC($G, f, \gamma, B, D, entry, \mathbf{C}_{\max}$)
    return computeSubpatrolDnC($G, f, \gamma, B, D, \infty, entry, \emptyset, \mathbf{C}_{\max}$)
                                                                    ▷ Algorithm 5
end procedure
```

Algorithm 5 A divide and conquer (DnC) algorithm for computing a subpatrol in a (subgraph of a) layout graph.

```
Require: $G = (V, E)$: a layout graph
Require: $\gamma \in [0, 1\rangle$: the discount factor
Require: $f$: the observation value function
Require: $B \in \mathbb{N}$: the maximum time that may be spent in an atomic cluster
Require: $B_G \in \mathbb{N}$: the maximum time that may be spent in $G$
Require: $D \in \mathbb{N}$: the maximum diameter of an atomic cluster.
Require: $entry \in V$: the vertex from which the layout graph is entered
Require: $exit \in V$: the vertex from which the layout graph should be exited.
Require: $\mathbf{C}_{\max}$: the maximum number of clusters in which $G$ may be divided.
Ensure: $P_{entry,G,exit}$: a (sub)patrol for graph $G$ starting at entry and terminating at
    exit of length no greater than budget
procedure computeSubpatrolDnC($G, f, \gamma, B, B_G, D, entry, exit, \mathbf{C}_{\max}$)
    if $diam(G) \leq D$ then                          ▷ The graph is small enough: conquer
        return Conquer($G, f, entry, exit, B$)                            ▷ Algorithm 2
    else                                                ▷ The graph is too big: divide
        $G[\mathcal{C}] = (\mathbf{C} \cup \mathbf{T}, E_{\mathcal{C}}) \leftarrow$ Divide($G, D, \mathbf{C}_{\max}$)              ▷ Algorithm 1
        $subpatrols \leftarrow \{\}$
        for $C \in \mathbf{C}$ do                         ▷ Compute subpatrols for each cluster
            $B_C \leftarrow$ computeBudget($C, B, D, \mathbf{C}_{\max}$)                 ▷ Algorithm 6
            for $T \in adj_{G[\mathcal{C}]}(C), T' \in adj_{G[\mathcal{C}]}(C)$ do
                $P_{T,C,T'} \leftarrow$ computeSubpatrolDnC($C, f, \gamma, B, B_C, D, T, T', \mathbf{C}_{\max}$)
                $subpatrols \leftarrow subpatrols \cup \{P_{T,C,T'}\}$
            end for
        end for
        return Merge($G[\mathcal{C}], subpatrols, \gamma^{B_G}, B_G, entry, exit$)        ▷ Algorithm 3
    end if
end procedure
```

Algorithm 6 An algorithm for computing the time that may be spent in a cluster C.

```
Require: $C$: a cluster
Require: $B \in \mathbb{N}$: the maximum time that may be spent in an atomic cluster
Require: $D \in \mathbb{N}$: the maximum diameter of an atomic cluster.
Require: $\mathbf{C}_{\max}$: the maximum number of clusters in which $C$ may be divided
procedure computeBudget($C, B, D, \mathbf{C}_{\max}$)
    if diam($C$) $\leq D$ then
        return $B$
    else
        $G[\mathcal{C}] = (\mathbf{C} \cup \mathbf{T}, E_{\mathcal{C}}) \leftarrow$ Divide($G, D, \mathbf{C}_{\max}$)
        return $|\mathbf{C}| \times \max_{C' \in \mathbf{C}}$ computeBudget($C', B, D, \mathbf{C}_{\max}$)
    end if
end procedure
```

the 36 second-level clusters are Divided one more time resulting in 64 atomic clusters. Figure 7 shows a tree which represents the recursive division of the clusters. In this tree the root represents the complete layout graph, and the leafs the atomic clusters.

Each of the atomic clusters (coloured nodes in Figure 7) are solved using Conquer. *Going up the tree, we find the non-atomic clusters. Each of these was clustered to obtain a cluster graph (line 5 in Algorithm 4). For example, the cluster graph of the root graph is shown in Figure 8(a). Similarly, the cluster graph of the top-level cluster in the bottom right of Figure 6 is shown in 8(b). The left two columns of Figure 7 shows the* maximum *budget (computed using Algorithm 6) and discount factor used as parameters to the* Merge *operation for solving patrolling problems on the corresponding levels of the tree.*

4.1.5. Determining Parameters

Algorithm 4 takes three parameters: D, B and $\mathbf{C}_{\max}$, all of which affect the algorithm's performance. Here we briefly discuss their effect and describe a methodology of selecting appropriate values:

- An increase in D yields an increase in the number of atomic clusters, but a reduction in their size. Smaller clusters are easier to solve by Conquer, but they increase the amount work performed by Merge by increasing the recursion depth of Algorithm 4. Unfortunately, it is not possible to make general statements about the optimal value of D for arbitrary graphs and observation value functions. As a rule of thumb, we chose the size of the clusters such that an agent is capable of visiting at least $k = 10$ of the greedily selected vertices in Algorithm 2. Furthermore, metrics such as the number of edge cuts and the variance in diameter indicate the quality of the clustering for a given graph. These metrics depend highly on the type of graph and the type of graph clustering algorithm used. We refer the reader to Schaeffer (2007) for an overview of these algorithms.

- B determines the trade-off between intra and inter cluster patrolling. As B is increased, agents spend more time patrolling atomic clusters before moving on to the next. At a certain point, the additional value obtained within clusters no longer compensates for the decrease in discount factor γ^B. Given this, a good

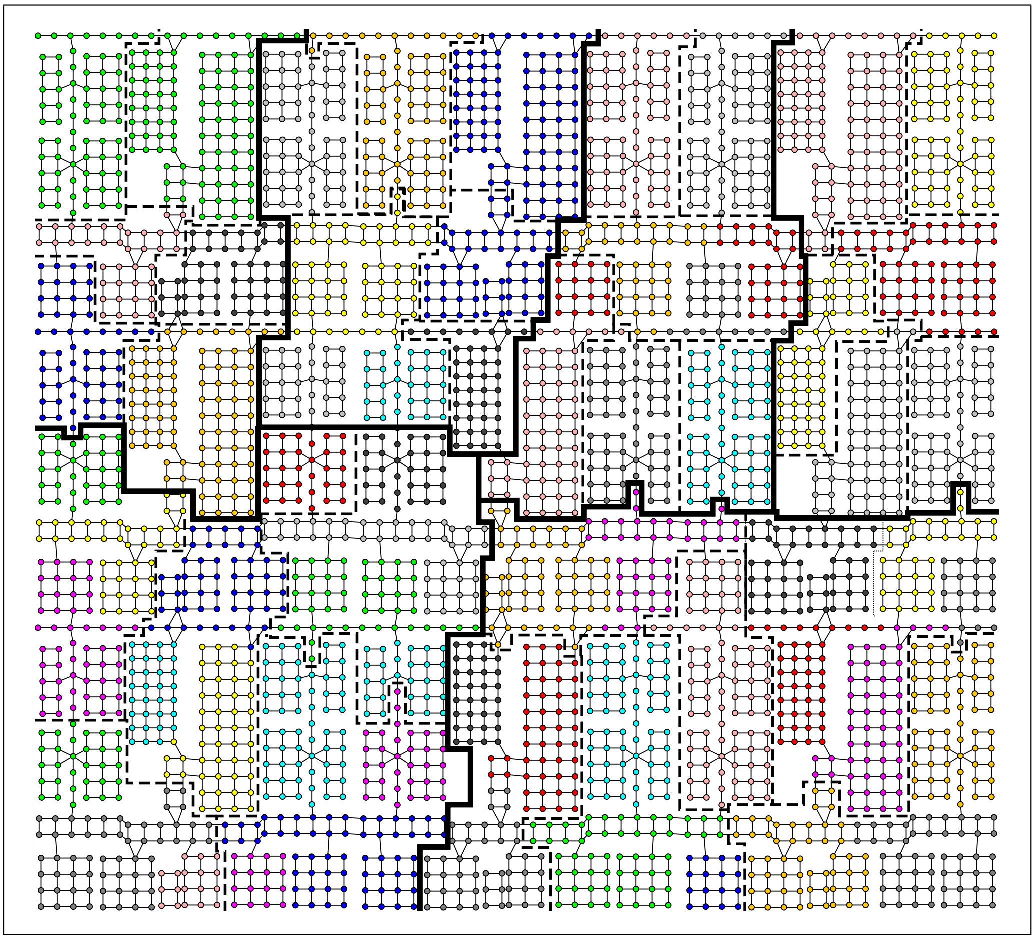


Figure 6: Nine copies of the layout graph in Figure 3 laid out in a three by three grid. The graph has been recursively clustered on three levels. The six top-level clusters are demarcated with bold lines and the six second-level with dashed lines. The 64 atomic clusters are distinguishable by the colour of their vertices.

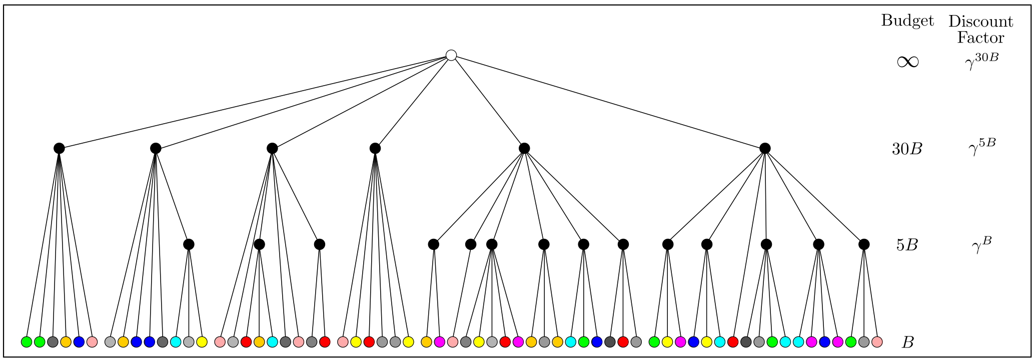
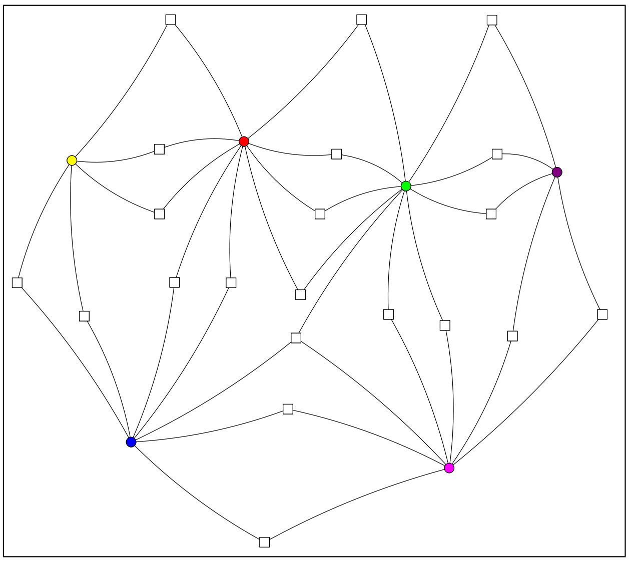
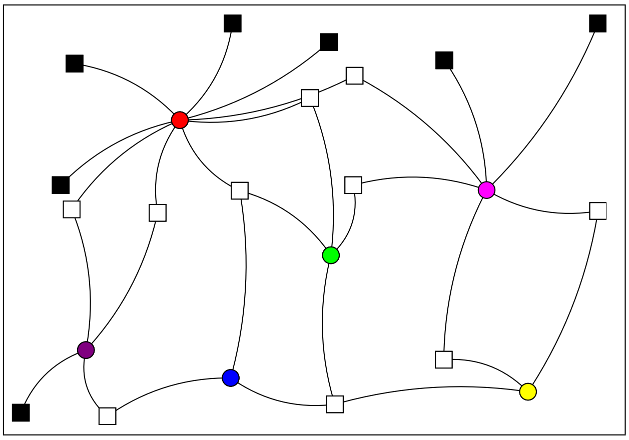


Figure 7: The relations between top-level, second-level and atomic clusters in Figure 6 represented as a tree. Children of a cluster are shown in order of their clock-wise appearance in Figure 6.



(a) The cluster graph of the root graph in Figure 7. Squares represent transit nodes, circles represent clusters.



(b) The cluster graph of the bottom right top-level cluster in Figure 7. White squares represent transit nodes between clusters, black squares represent the transit nodes to other top-level clusters.

Figure 8: Cluster graphs representing the topological relations between child-clusters of two clusters in Figure 7.

heuristic is to find the value of B^* which maximises the average discounted value of all subpatrols computed by Conquer:

$$B^* = \max_{B \in [D, |V_C|]} \sum_{P_{T,C,T'}} \gamma^B I(C, 0, T, T')$$

As this value depends on observation value function f, it is problem dependent. However, it can be efficiently computed using a simple hill climbing algorithm.

- $\mathbf{C}_{\max}$ is a parameter that determines the trade-off between computation and solution quality. The larger $\mathbf{C}_{\max}$, the larger the set of subpatrols that can be concatenated, which generally results in higher solution quality. However, the empirical results in Section 7 show that the increase in solution quality diminishes fairly quickly as $\mathbf{C}_{\max}$ is increased. Thus, as a rule of thumb, $\mathbf{C}_{\max}$ should be increased until the available computing capabilities or time constraints no longer warrant the increase in solution quality (both of which are application dependent).

This concludes the description of the single-agent algorithm. Using this algorithm as a building block, we can now derive the multi-agent algorithm, which is described next.

4.2. The Multi-Agent Algorithm

Now that we have defined the single-agent algorithm, we can extend it to compute policies for the multi-agent problem. A straightforward, but somewhat naïve, way of doing this is to extend the MDP constructed in the Merge operation to multiple agents. The state space of this multi-agent MDP contains the position of each agent, and its action space is defined as the Cartesian product of the action spaces of the single agents. However, in so doing, the size of the state and action space grow exponentially with the number of agents M, allowing only the smallest of problem instances to be solved.[10]

The key property of our multi-agent algorithm is that it avoids this problem by sequentially computing policies for single agents, instead of computing a joint policy for the team. More specifically, our approach computes a nearly optimal policy for a team of agents (we prove this in the next section), by greedily computing single-agent policies for each agent i, conditioned on the previously computed policies of agents $\mathbf{A}_{-i} = \{1, \ldots, i-1\}$.

This greedy method is similar to the *sequential allocation* of multiple agents proposed by Singh et al. (2007). However, the problem they address is to compute finitely long paths for each agent, instead of policies. This makes a straightforward application of their *sequential allocation* method impossible.

In more detail, under the assumption of Singh et. al., it is possible to define a new observation value function f' that computes the marginal value of observations O_i

[10] While testing this naïve approach on the setting in Experiment 1 in Section 7 with only 2 agents, it consistently ran out of memory on a machine with 12GB of RAM after expending $\gg$ 2 hours of computation.

made by agent i conditioned on observations made by agents $\mathbf{A}_{-i}$, i.e.:

$$f'(O_i) = f\left(O_i \cup \bigcup_{j=1}^{i-1} O_j\right) - f\left(\bigcup_{j=1}^{i-1} O_j\right)$$

However, this implicitly assumes there exists an order in which the agents make observations; agent 1 traverses the environment first, agent 2 second, etc. Clearly, no such ordering is possible with paths of *infinite* length (i.e. the policies computed by the single-agent algorithm). Thus, we need to fundamentally redesign the reward function used in the single-agent algorithm developed in the previous section in order to correctly allocate rewards to agents, and thus be able to perform sequential allocation.

To be able to incrementally compute the team policy, we modify the single-agent MDP defined in Section 4.1.3 such that the goal of agent i becomes to maximise the received marginal reward, or equivalently, to collect the observation value left behind by agents $\mathbf{A}_{-i}$. Put differently, agent i operates on a modified MDP that changes according to the policies of $\mathbf{A}_{-i}$. To accomplish this, we make the transition function of agent i reflect the effect of the policies of agents $\mathbf{A}_{-i}$, while agents $\mathbf{A}_{-i}$ are unaware of the existence of agent i.

The MDP that captures this process can be obtained from the single-agent MDP discussed in the previous section by making the following modifications:

State Space Agent i now takes into account the positions and states of agents $\mathbf{A}_{-i}$ (but not *vice versa*) in order to determine how the world will change under their policies. States thus become composite (or recursive).

Transition Function The transition function now reflects the effect of agent i's actions, as well as the policies executed by agents $\mathbf{A}_{-i}$.

Reward Function The reward function now rewards agent i only for the received marginal observation value, i.e. the observation value left behind by agents $\mathbf{A}_{-i}$.

The relations between states, policies and transition functions in this modified MDP are shown in Figure 9. In the remainder of this section we shall discuss each modification in more detail.

State Space. The new MDP takes into account the effect of agent i's actions, as well as those of agents $\mathbf{A}_{-i}$ who are executing their policies beyond agent i's control. In order to determine these actions, the MDP needs to include knowledge of the policies of agents $\mathbf{A}_{-i}$, as well as their current states.

Thus, we define composite states, which combine the *atomic* state — the states of the single-agent MDP defined in Section 4.1.3 — of agent i with the composite state of agent j:

Definition 26 (Multi-Agent Patrolling State). *Let $\tilde{s}$ denote the atomic states of the form $(T, \boldsymbol{\lambda})$ as in Definition 21. The multi-agent state for agent i is given by the following recursive relation:*

$$\begin{aligned} s_1 &= \quad \tilde{s}_1 \\ s_2 &= \quad (\tilde{s}_2, s_1) \\ &\vdots \\ s_i &= \quad (\tilde{s}_i, s_{i-1}) \end{aligned} \tag{10}$$

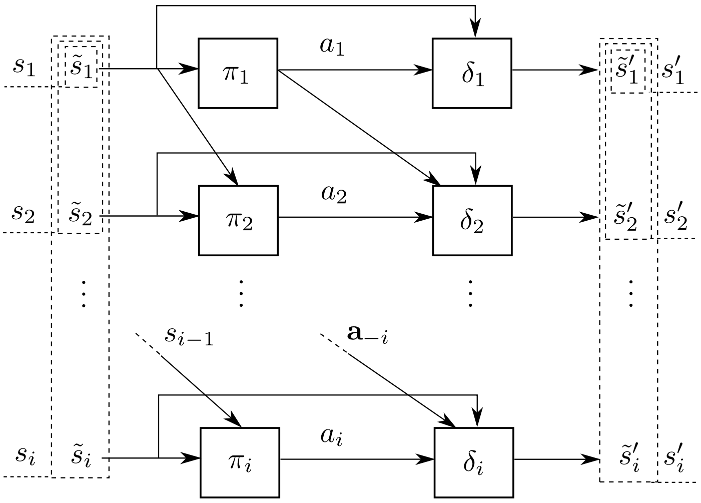


Figure 9: The recursive state space of agent i.

Transition Function. To determine the successor state s'_i obtained by applying action a_i of agent i, the multi-agent transition function first determines the state s'_{i-1} that results from the actions of agents $\mathbf{A}_{-i}$. State s'_i is then obtained by applying action a_i to s'_{i-1}.

With this in mind, we define the multi-agent transition functions as follows:

Definition 27 (Multi-Agent Patrolling Transition Function)**.** *The multi-agent patrolling transition function δ_i for agent i is recursively defined as:*

$$\begin{aligned} {s'}_1 = &\quad \delta_1(s_1, a_1) \\ {s'}_2 = &\quad \delta_2(\delta_1(s_1, \pi_1(s_1)), a_2) \\ &\vdots \\ {s'}_i = &\quad \delta_i(\delta_{i-1}(s_{i-1}, \pi_{i-1}(s_{i-1})), a_i) \end{aligned}$$

where δ_1 is equal to the patrolling transition function for single agents as in Definition 24.

The following example demonstrates the multi-agent state space and transition function.

Example 7. *Consider the environment in Figure 3 and bipartite graph $G[\mathcal{C}]$ in Figure 4 with two agents. At time step t, the atomic states $\tilde{s}$ of these agents are $\tilde{s}_1 = (T_7, [\tau, \tau, \tau, \tau, \tau, 0], \cdot)$ and $\tilde{s}_2 = (T_6, [\tau, \tau, 0, \tau, \tau, 0], \cdot)$ (and the composite state of agent 2 is $s_2 = (\tilde{s}_2, s_1)$). Thus, agent 1 has just patrolled cluster C_6 and is now positioned at T_7. Similarly, agent 2 has just patrolled cluster C_3 and is now positioned at T_6. Note that agent 2 is aware of the fact that agent 1 patrolled C_6, but agent 1 — being unaware of the existence of agent 2 — does not know about the new state of cluster C_7.*

Reward Function. To ensure the reward function only takes into account marginal observation value, we need to exclude double counting. There are two types of double counting. First, *synchronous* double counting, which occurs when two agents patrol the same cluster within the same time step. In this case the reward for patrolling the cluster is received twice. Second, *asynchronous* double counting, which is a little more subtle. For ease of exposition, we will illustrate this with an example.

Example 8. *(Continued from Example 7) At time step t, agent 1 patrols C_3 by choosing action (T_7, C_3, T_2) and transitions to $(T_3, [\tau, \tau, 0, \tau, \tau, 0])$. The reward for this transition is equal to the observation value obtained from patrolling cluster C_3 in state τ. In reality, however, much less value is obtained, since agent 2 patrolled C_3, and reset its visitation time λ_3 to 0. Put differently, agent 2 "stole" the reward of agent 1 for patrolling C_3.*

Thus, asynchronous double counting occurs whenever an agent i patrols a cluster C before agent j ($j < i$), such that j's belief of λ_C is less than its true value.

To prevent double counting — both synchronous and asynchronous — we introduce a penalty P for agent i that compensates for the reduction of reward of the agent j ($j < i$) that patrols C next, as follows:

$$R_i(s, (T, C, T')) = R(s, (T, C, T')) - P \tag{11}$$

Here, $R(\cdot, \cdot)$ is the reward function defined in Section 4.1.3, and P is the loss incurred by agent j ($j < i$) that will patrol cluster C next. This is the (discounted) difference between the expected reward (which agent j would have received in the absence of agent i) and its actual reward, discounted by the number of time steps t_n that will elapse before agent j patrols C:

$$P = \gamma^{t_n}(R_{expected} - R_{actual}) \tag{12}$$

The rewards $R_{expected}$ and R_{actual} are defined as:

$$R_{expected} = I(C, \min(\tau, \hat{\lambda}_C + t_n), \hat{T}_{start}, \hat{T}_{end}) \tag{13}$$

$$R_{actual} = I(C, t_n - B, \hat{T}_{start}, \hat{T}_{end}) \tag{14}$$

where $I(C, \lambda_C, T, T')$ is the value of a subpatrol (Equation 6), $\hat{\lambda}_C$ is the last visitation time of cluster C in agent j's current state; $\hat{T}_{start}$ and $\hat{T}_{end}$ are the entry and exit transit nodes chosen by agent j for its next visit to C.

The following example illustrates the behaviour of the new reward function.

Example 9. *Consider a scenario with two agents and a single cluster C. Agent 1 patrols this cluster at $t = 0$ and $t = 6$, and agent 2 at $t = 3$. Furthermore, suppose that the maximum reward for patrolling C is 1, that $\tau = 6$ and that the reward increases 0.2 every time step the cluster is not patrolled. Figure 10 shows the function of* potential *reward as a function of time for this scenario, which is realised only when the cluster is patrolled. The two lines in Figure 10 represent the beliefs agents 1 and 2 have of this reward.*

The rewards received by the agents are as follows (see Table 3). First, agent 1 patrols C at $t = 0$ and receives a reward of 1. Second, agent 2 patrols the cluster at $t = 3$ and receives a reward of 0.4. At this point, the beliefs of the agents diverge,

Time step	**Actual Reward**		**Marginal Reward**	
	Agent 1	Agent 2	Agent 1	Agent 2
t_0	1		1	
t_3		$0.4\gamma^3$		$0.4\gamma^3 - 0.6\gamma^6$
t_6	$0.4\gamma^6$		γ^6	
Total	$1 + 0.4\gamma^3 + 0.4\gamma^6$		$1 + 0.4\gamma^3 + 0.4\gamma^6$	

Table 3: The actual and marginal rewards received by the agents in Example 9.

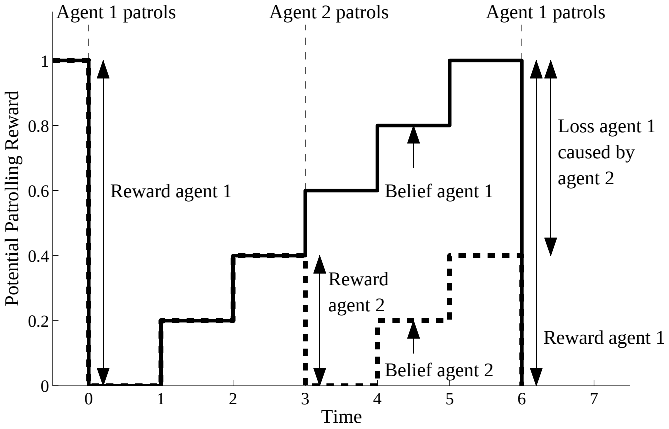


Figure 10: The potential reward for patrolling cluster C in the scenario of Example 9.

because agent 1 *is not aware of agent* 2*'s actions. Finally, agent* 1 *patrols the cluster at* $t = 6$*. Contrary to its beliefs, it receives a reward of* 0.4 *instead of* 1*. In total the team receives a (discounted) reward of* $1 + 0.4\gamma^3 + 0.4\gamma^6$*.*

Now, consider the marginal rewards of the agents, i.e. the additional observation value received by adding an extra agent. To compute these rewards for agent 1*, we need only consider the beliefs of agent* 1*, because it believes it is alone. It patrols the cluster twice when the reward equals* 1 *(at time step 0 and 6), so its reward is* $1 + \gamma^6$*. For agent* 2*, we need to consider its reward for patrolling the cluster at time step* 3*, but also the loss of reward of agent* 1 *at time step* 6 *for which it is responsible. This loss is* $0.6\gamma^6$*, which makes its marginal reward* $0.4\gamma^3 - 0.6\gamma^6$*. To see that these penalties are correct, note that the sum of marginal rewards is equal to the sum of actual rewards, as desired.*

This concludes the definition of the MDP for multiple agents. Using value iteration to solve this MDP as before, we obtain a policy for each individual agent, which, when combined, form a policy for the entire team. This team policy is not optimal, since the policy for agent i is computed greedily with respect to the policies of agents $\mathbf{A}_{-i}$. Despite this, we can still derive performance guarantees on the observation value obtained by the team, as we show in the next section.

5. Theoretical Analysis

As mentioned in the introduction, performance guarantees are important in critical domains such as space exploration and security, since the existence of pathological behaviour should be ruled out. In this section, we will therefore derive performance guarantees on the solution quality achieved by the algorithm presented in the previous section, as well as bounds on its computation overhead.

5.1. Solution Quality

We will first derive a lower bound on the solution quality of the single-agent algorithm, by proving the following lemma:

Lemma 1. *If* $\mathrm{diam}(C) \leq D = \frac{2}{3}B\left(\sqrt{\frac{\pi k}{2}} + \mathcal{O}(1)\right)^{-1}$*, Algorithm 2 computes a subpatrol* $P_{T,C,T'}$ *with an observation value* $I(C, \lambda_C, T, T')$ *of at least*

$$\gamma^B\left(1 - \left(\frac{k-1}{k}\right)^k\right) f(O^*)$$

Here, $f(O^*)$ *is the value of the optimal set of observations made at* k *vertices of* C*, ignoring the movement constraints of* G*.*

Proof. The proof consists of two steps. In the first, we use a result by Moran (1984) to prove that any TSP in a graph with k vertices with diameter D has a cost less than B. Moran (1984) proved a bound on the length L of the TSP of an arbitrary graph with k vertices. Specifically, for a graph G embedded in two-dimensional Euclidean space, the following relation holds:

$$L \leq \left(\sqrt{\frac{\pi k}{2}} + \mathcal{O}(1)\right)\mathrm{diam}(G)$$

By applying this relation to line 11 of Algorithm 2, we know that $n \geq k$ holds when this algorithm terminates. The extra cost of including T and T' (which are contained in C) into the TSP is compensated by the fact that we set the cost of moving between T and T' to 0 (since we require a *path* from T to T', not a cycle). As mentioned earlier, instead of solving the TSP optimally (which is an NP-complete problem), we use the approximation algorithm by Christofides (1976). This algorithm has an approximation ratio of $\frac{3}{2}$, which accounts for the factor of $\frac{2}{3}$ on bound of the $\mathrm{diam}(C)$.

In the second step of this proof, we apply the following theorem by Nemhauser and Wolsey (1978) for obtaining a bound on the value of the greedily selected vertices (lines 2–5 of Algorithm 2):

Theorem 1. *Let* $f : 2^E \rightarrow \mathbb{R}$ *be a non-decreasing submodular set function. The greedy algorithm that iteratively selects the element* $e \in E$ *that has the highest incremental value with respect to the previously chosen elements* $I \subset E$*:*

$$e = \arg\max_{e \in E \setminus I} f(e \cup I) - f(I)$$

until the resulting set I *has the desired cardinality* k*, has an approximation bound* $\frac{f(I_G)}{f(I^*)}$ *of at least* $1 - \left(\frac{k-1}{k}\right)^k$*, where* $I^* \subset E$ *is the optimal subset of cardinality* k *that maximises* f*.*

This theorem states that the ratio between the value of the first k greedily selected elements and the value of the optimal k elements is at least $1 - \left(\frac{k-1}{k}\right)^k$. The factor of γ^B stems from the fact that it is unknown in which order these k elements are visited by the TSP. However, it is known that these elements are visited within B time steps. Thus, we obtain a lower bound by discounting the incremental values obtained at these k elements by B time steps, which completes the proof. □

The Merge operation of the algorithm (Section 4.1.3) uses these subpatrols and concatenates them into a single overarching patrol. The problem of finding an optimal sequence of subpatrols is represented as an MDP, which is optimally solved by value iteration. Consequently, the following holds for the value of the initial state $\overline{s}$, which is equal to the discounted observation value received by the agent by following policy π (Equation 8):

$$V^{\pi}(\overline{s}) \geq \frac{\gamma^B}{1-\gamma^B}\left(1 - \left(\frac{k-1}{k}\right)^k\right) f_{\min}(O^*) \tag{15}$$

where $f_{\min}(O^*)$ is the minimum value of $f_{\min}(O^*)$ over all clusters $\mathbf{C}$.

To prove a bound on the solution quality of the multi-agent algorithm, we prove that the observation value of a set of policies is submodular. To do this, we define a set function g over a set of *single-agent* policies $[\pi_1, \ldots, \pi_M]$, that computes the discounted observation value of a set of policies:

$$g(\pi_1, \ldots, \pi_M) = \sum_{i=1}^{M} V^{\hat{\pi}_i}(\overline{s}_i)$$

Here, $\hat{\pi}_i$ is a policy for agent i of the form discussed in Section 4.2, which behaves identically in the presence of agents $1, \ldots, i-1$ as policy π_i does in isolation. Thus, policy $\hat{\pi}_i$ visits the same clusters as π_i, and in the same order. Since the discounted marginal observation value of a single policy $\hat{\pi}_i$ received from initial state $\overline{s}$ is equal to $V^{\hat{\pi}_i}(\overline{s})$, function g computes the discounted observation value of a team of agents $1, \ldots, M$.

We can now state the following result:

Lemma 2. *Function g is a non-decreasing submodular set function.*

Proof. The non-decreasing property follows trivially from the fact that adding more agents never reduces the observation value they receive as a team (since existing agents do not change their policies). To prove submodularity, we need to show that, for every set of policies $\boldsymbol{\pi}' \subseteq \boldsymbol{\pi}$ and policy $\pi \notin \boldsymbol{\pi}'$ the following holds:

$$g(\{\pi\} \cup \boldsymbol{\pi}') - g(\boldsymbol{\pi}') \geq g(\{\pi\} \cup \boldsymbol{\pi}) - g(\boldsymbol{\pi})$$

To prove that this holds, we just need to prove that adding a policy π to a set of policies $\boldsymbol{\pi}$ instead of $\boldsymbol{\pi}' \subseteq \boldsymbol{\pi}$ reduces reward and increases penalty (Equation 11). To prove the former, observe that agent i's belief of the last visitation time λ_C^i of cluster C is non-increasing in i, and Equation 6 is non-increasing in λ_C^i. Thus, adding predecessors to agent i reduces its reward for any subpatrol in any cluster. To prove the latter, observe that, with additional predecessors, the number of time steps t_n before any predecessor visits the same cluster C decreases or remains unchanged. Since penalty P is a strictly increasing function of t_n (see Equations 12, 13, and 14), adding π to $\boldsymbol{\pi}$ instead of $\boldsymbol{\pi}' \subseteq \boldsymbol{\pi}$ indeed increases the penalty. □

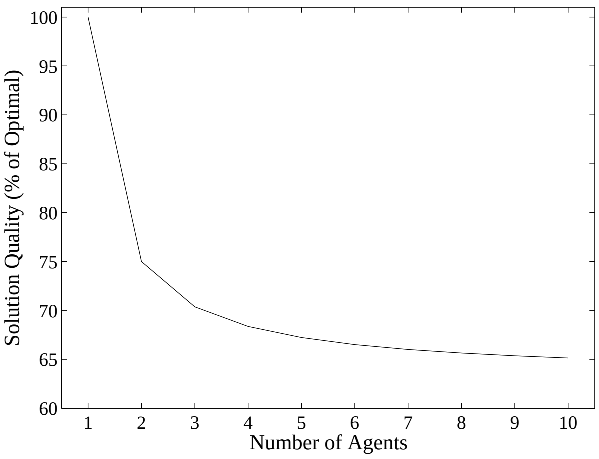


Figure 11: The worst-case bound on the approximation ratio of the multi-agent algorithm proved in Corollary 1 as a function of the number of agents M.

Since the multi-agent algorithm maximises the incremental value of g by greedily computing a policy of agent i with respect to the policies of agents $1, \ldots, i-1$, Theorem 1 by Nemhauser and Wolsey (1978) can be directly applied to obtain the following result:

Corollary 1. *For M agents, the policies computed by the multi-agent algorithm are at least $\left(1 - \left(\frac{M-1}{M}\right)^M\right)$ as valuable as the optimal M policies of the type computed by the single-agent algorithm.*

See Figure 11 for an illustration of the bound as a function of the number of agents M. Thus, for $M \to \infty$, the multi-agent policy yields at least $\approx 63\%$ observation value as the best policy obtained by searching the joint policy space for M agents. Note that the latter policy is not optimal, because the single agent algorithm from Section 4.1 is not optimal. However, by sacrificing at most $\approx 37\%$ observation value, a significant amount of computation can be saved by computing the multi-agent policy sequentially, instead of searching the joint policy space.

5.2. Computational Complexity

The computational complexity of Algorithm 5 can be decomposed into the complexity of its three operations:

- The complexity of Divide is determined by subroutine Cluster$(G, D, \mathbf{C}_{\max})$. It solves the problem of finding a clustering that minimises the maximum diameter of the (at most) $\mathbf{C}_{\max}$ clusters, which is known to be NP-hard (Schaeffer, 2007). To ensure computational efficiency, we choose an approximation algorithm that requires more than the optimum number of clusters to satisfy the maximum diameter requirement. In particular, as mentioned in Section 4.1.1, we select the algorithm proposed Edachery et al. (1999), which finds a partitioning in time $O(|V|^3)$.

- The majority of the computation required by CONQUER is attributable to computing the TSP in line 13 in Algorithm 2. As mentioned earlier, the complexity of computing an optimal TSP is exponential in $|V|$ (assuming P $\neq$ NP). However, if we use the heuristic proposed by Christofides (1976), which has the most competitive performance bounds, this is reduced to $O(|V|^3)$.
- Lastly, the complexity of MERGE is determined by value iteration to solve the MDP. Value iteration requires a number of iterations that is polynomial in $1/(1-\gamma), 1/\epsilon$, and the magnitude of the largest reward (Littman et al., 1995). Moreover, a single iteration of value iteration (Equation 9) can be performed in $O(|A||S|)$ steps.[11]

  For the single-agent case:

  $$|S| = |\mathbf{T}| \left(\frac{\tau}{B} + 1\right)^{|\mathbf{C}_{\max}|} = O\left(|\mathbf{C}_{\max}|^2 \left(\frac{\tau}{B} + 1\right)^{|\mathbf{C}_{\max}|}\right)$$

  The size of the action space $|A|$ depends on the connectedness of the bipartite graph $G[\mathcal{C}]$, but is $O(|\mathbf{C}_{\max}^2|)$ in the worst case.

Thus, the MERGE operation dominates the complexity of Algorithm 5; its computational complexity is exponential in parameter $\mathbf{C}_{\max}$. This leads to the following theorem:

Theorem 2. *The computational complexity of the single-agent algorithm (Algorithm 4) is:*

$$O\left(|V||\mathbf{C}_{\max}|^3 (\frac{\tau}{B} + 1)^{|\mathbf{C}_{\max}|}\right)$$

Proof. Let $T(|V|)$ be the work required by Algorithm 5 for graph $G(V, E)$. $T(|V|)$ is given by the following recursive relation:

$$T(|V|) = O\left(|\mathbf{C}_{\max}|^3 (\frac{\tau}{B} + 1)^{|\mathbf{C}_{\max}|}\right) + \mathbf{C}_{\max} \cdot T\left(\frac{|V|}{\mathbf{C}_{\max}}\right) \tag{16}$$

This is because the algorithm involves one call to MERGE (line 14) and $\mathbf{C}_{\max}$ recursive calls on each of the $\mathbf{C}_{\max}$ subproblems of about equal size (line 10). The application of the master theorem (Cormen et al., 2009, p. 97) completes the proof. □

Theorem 2 states that the complexity of the single-agent algorithm is exponential in $\mathbf{C}_{\max}$. More importantly, however, it states that it is *linear* in the size of the graph. Hence, the algorithm scales well with the size of the graph.

A similar result can be obtained for the multi-agent algorithm:

Corollary 2. *The computational complexity of the multi-agent algorithm with M agents is:*

$$O\left(|V||\mathbf{C}_{\max}|^{(|M|+2)} (\frac{\tau}{B} + 1)^{|\mathbf{C}_{\max}|}\right)$$

[11] This is because our transition function is deterministic. For *non-deterministic* transition functions, value iteration needs $O(|A||S|^2)$ steps.

Proof. This follows directly from Theorem 2 and the fact that a state in the multi-agent MDP keeps track of the position of the M agents. □

Thus, the complexity of the multi-agent algorithm is also linear in the size of the graph, but exponential in the number of agents. It is important to note that this exponential growth applies to the full set of states, many of which are not reachable from the initial state of the environment (Equation 4). For this reason, we will empirically quantify the considerable savings that result from disregarding unreachable states in Section 7.

6. Further Extensions

In this section we discuss several extensions to extend the applicability and improve the robustness of the single and multi-agent algorithms in real-life applications. These two requirements were identified in the introduction to allow agents to operate in life-critical and hostile environments.

6.1. Dealing with Limited Mission Time

Often, in real-life applications, the mission time of information gathering agents is limited. For example, UAVs carry limited fuel and ground robots have a limited battery life. Similarly, a cleaning robot has limited capacity for carrying collected dirt. In these cases, the agents either have to be taken out of commission after their power source has been depleted or have to be recharged. Both cases can be handled by extending the MDP used in the Merge operation of the single-agent algorithm:

No Recharging If no possibility of recharging exists, agents with depleted energy sources have to return to a base station. This is easily enforced by replacing the call to computeSubpatrolDnC in Algorithm 4 with:

$$\textsc{computeSubpatrolDnC}(G, f, \gamma, B, D, E_0/R_{EC}, \mathit{entry}, \mathit{base}, \mathbf{C}_{\max})$$

where E_0 is the energy capacity at the start of the mission, R_{EC} is the energy consumption per time step and *base* is the location of the base station.

Possibility of Recharging If recharging/refuelling stations are present, agents have to arrive at those stations before their energy source is depleted. Therefore, handling this case is therefore similar to the previous one, except that the transition function δ in Definition 24 is replaced by δ':

$$\delta'((T, [\lambda_{C_1}, \ldots, \lambda_{C_{|\mathbf{C}|}}], B_r), a) = \begin{cases} \delta(s, a) & \text{if } a = (T, C_i, T') \\ (T, [\lambda_{C_1}, \ldots, \lambda_{C_{|\mathbf{C}|}}], B_r + R_T \cdot B_G) & \text{if } a = CHARGE \end{cases}$$

where $CHARGE$ is a (new) action of staying put while charging, R_T is the recharging rate for the recharging station at T (which is 0 if T does not have a recharging station), and $B_G = \textsc{computeBudget}(G, B, D, \mathbf{C}_{\max})$ is the budget for top-level clusters as computed by Algorithm 6. Note that this change applies only to the top-level call to computeSubpatrolDnC in Algorithm 4 which solves the patrolling problem in the full layout graph, not to subsequent recursive calls.

6.2. Repairing Patrols in Case of Component Failure

As mentioned in the introduction, the fact that the agents are operating in potentially hostile environments (space, severe weather conditions, or battlefields) makes them vulnerable to component failure. As shown in the previous section, the multi-agent policy that results from sequentially allocating agents is near-optimal. However, since it is computed at deployment, it is not inherently capable of responding to *a priori* unexpected events, such as the failure of one or more agents. Put differently, they are not *robust* against failure.

Thus, with the multi-agent algorithm in its current form, if agent i fails, it leaves a "gap" between agents $i-1$ and $i+1$, which can lead to a severe performance degradation depending on the value of i (with $i = 1$ being the worst-case scenario). In what follows next, we develop an algorithm for repairing patrols when one or more agents fail to close this gap. This algorithm relies on the recursive nature of the sequentially computed policies, i.e. the fact that agent i knows the policies of $\mathbf{A}_{-i}$. Thus, by adopting the policies of their predecessors, agents are capable of filling the gap left behind by a malfunctioning agent. After the repairing algorithm has completed, the agents end up with policies that are identical to those obtained from the multi-agent algorithm when computing policies from scratch. Moreover, the repairing algorithm is guaranteed to reduce the number of states that need to be searched as compared to computing from scratch, since it reuses the value function for the states that were searched previously.

In more detail, let agent i be the failed agent. In response to discovering that agent i has failed, each agent j ($j > i$) performs three operations:

- It adopts patrolling policy π_{j-1}. This policy is known by agent j, since it was required to compute π_j (see Section 4.2).
- It removes the atomic state (see Equation 10) of agent i from its recursive state:

$$s_j = (\tilde{s}_j, (\dots, (\tilde{s}_{i+1}, (\tilde{s}_{i-1}, (\dots)))))$$

- It updates policy π_{j-1} for states $S_r(s_j) \setminus S_r(\overline{s}_{j-1})$, which were unreachable from the initial state $\overline{s}_{j-1}$ of agent $j-1$. This policy update can be efficiently performed by running value iteration on $S_r(s_j) \setminus S_r(\overline{s}_{j-1})$ *only*; the values of states $S_r(s_{j-1})$ remain unchanged (Levin et al., 2009, Chapter 1). This is because the value of a state depends on values of its successors only (Equation 8), and states $S_r(\overline{s}_{j-1})$ are not successors of states $S_r(s_j) \setminus S_r(\overline{s}_{j-1})$ by definition. Thus, by updating policy π_{j-1} for the newly discovered states, instead of recomputing it from scratch, significant savings can be made. In the next section, we show that these savings typically amount to a 50% reduction in the number of states that need to be searched.

It is important to note that in the case of multiple failing agents, this algorithm has to be invoked for each failed agent. For instance, if all but one agents fail, the algorithm is executed $M(M-1)/2$ times, and the multi-agent team degenerates to the single agent case.

Figure 12 shows an example of the effect of repairing policies in a scenario with two agents. Here, agent 1 fails after 500 time steps. At this point, the agents' performance drops significantly; depending on the dynamism of the environment expressed by parameter τ, the average time between visiting vertices in graph G increases by

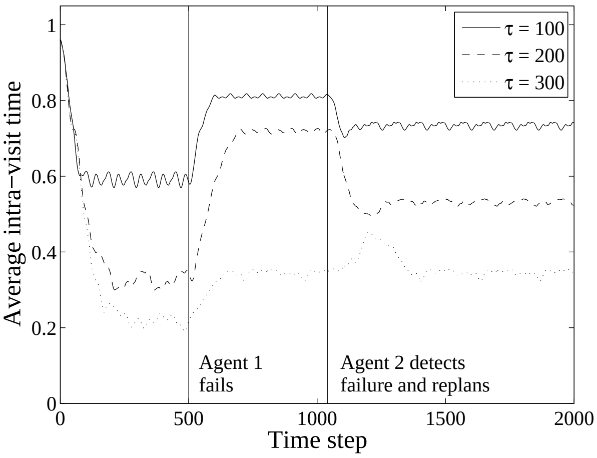


Figure 12: An example of repairing policies with two agents for different values of τ. Agent 1 fails after 500 time steps. Agent 2 replans at time step 1050.

50%–130%. Failure of agent 1 is detected by agent 2 at time step 1050,[12] after which it adopts agent 1's policy and replans for its own current state.

Note that in the discussion above, we did not mention the method whereby agents detect failure of their peers. This was done deliberately, because failure detection depends on the type and communication capabilities of the agents. For instance, if the agents are always in communication range of each other, or if a central communication hub exists, failure can be assumed after a prolonged time of radio silence. In the absence of centralised communication, agents can keep track of the last "signs of life" of each agent, and exchange this information with others using a multi-hop wireless communication network or a "graffiti-style" method communication involving RFID tags dropped in the environment (Tammet et al., 2008). Due to the implementation dependent nature of failure detection, and the general nature of our model and algorithms, we consider further discussion of this issue beyond the scope of this paper.

The next section deals with repairing patrols in response to the changes in the layout of the environment.

6.3. Repairing Patrols in Case of Changes in Environmental Layout

Besides unexpected failure of agents, changes in the layout graph are another source of *a priori* unknown events that can have a detrimental effect on the agents' performance. In this section, we provide an algorithm for repairing patrols in response to such events.

The recursive nature of Algorithm 4 allows for a natural way of repairing patrols. Algorithm 7 shows the procedure for repairing a patrol when vertex v_r is removed from the graph.[13] First, it recursively searches for the affected component of the graph. When found, there exist several possibilities:

[12] This number is chosen for illustration purposes only. In practice, it is highly dependent on the communication capabilities of the agents, their communication range and communication protocols.

[13] For the purpose of clarity, we only consider the removal of a single vertex. The necessary modifications to deal with the removal of edges or multiple vertices are trivial.

1. The patrol does not move through v_r. No recomputation necessary.
2. Vertex v_r is contained in an atomic cluster C. Recompute the subpatrol through C using Conquer.
3. The removal of vertex v_r disconnects the containing cluster C, causing it to fall apart in multiple components. To repair the patrol, the patrols for the parent cluster of C are recomputed from scratch using Algorithm 5. A special case is if C is the root cluster (i.e. the layout graph of C). In this case multiple independent problems are obtained corresponding to each component with an agent in it.
4. Vertex v_r is contained in a transit node T, which, in turn, is contained in non-atomic cluster C. Two possibilities exist:
   a Transit node T does not lose connectivity to adjacent clusters. For each subpatrol that traverses T, compute the new shortest path through T between adjacent clusters. No further recomputation is necessary since the patrol is not blocked.
   b T becomes disconnected, but C remains connected. Recompute each subpatrol through C using Merge.

If recomputation is deemed necessary, the algorithm propagates back up the hierarchy until one of two conditions is met. Either it arrives at the root, or it is terminated at an intermediate stage to save computation. The latter happens as soon as the loss due to the removal of v_r falls below a user-set percentage. This is not shown to preserve readability of Algorithm 7, but a check can be inserted before line 35 to compare the original and new values of the repaired subpatrols.

Unfortunately, it is not possible to make general statements about the incurred loss (except for trivial cases 1 and 4a). In cases 2, 3 and 4b, the loss of a vertex can cut off important paths, causing the agent to take radically different paths (for example, consider the effect of removing one of the vertices in T_7 in Figure 3). However, in case 2, the loss is likely to be at most $f\left((v_r, t)\right)$, which is the observation value if only v_r is observed at current time t (so no discount is incurred). In practice, this value is likely to be much less, since f is submodular and the loss of not observing v_r is compensated by observing the vertices around it. In case 3, the use of Algorithm 5 means the performance guarantees from Section 5 apply for the newly obtained problem.

This concludes the discussion of the extensions for improving robustness. In the next section, we empirically evaluate both the single and multi-agent algorithms.

7. Empirical Evaluation

Having presented the offline multi-agent algorithm for computing multi-agent patrols in Section 4, we now demonstrate that our approach outperforms the state of the art in the area of continuous multi-agent patrolling.

The reasons for performing empirical evaluation, in addition to having proved theoretical bounds on computation and solution quality in Section 5.1, are threefold. First, to ascertain how the algorithm performs in real-life inspired application settings, and how this relates to the theoretical bounds. These may differ (significantly), since the bounds from Section 5.1 are pessimistic, and relate to the worst possible scenario.

Algorithm 7 An algorithm for repairing a patrol in the event of a vertex becoming inaccessible.

Require: $P_{entry,C,exit}$: the previously computed subpatrol through C
Require: v_r: the vertex that has been removed
Ensure: $P'_{entry,C,exit}$: the repaired subpatrol through C

```
procedure REPAIRSUBPATROL($P_{entry,C,exit}, v_r$)
    if $v_r \notin P_{entry,C,exit}$ then                                        ▷ Case 1
        return $P_{entry,C,exit}$
    end if
    if $diam(C) \leq D$ then                                                     ▷ Case 2
        return CONQUER($C[V_C \setminus \{v_r\}], f, entry, exit, B$)
    else
        $G[\mathcal{C}] = (\mathbf{C} \cup \mathbf{T}, E_{\mathcal{C}}) \leftarrow$ DIVIDE($C, D, \mathbf{C}_{\max}$)
                                                    ▷ Reused from Line 5 in Algorithm 5
        if $\exists T_r \in \mathbf{T} : v_r \in V_{T_r}$ then                   ▷ Case 4
            $T'_r \leftarrow T_r[V_{T_r} \setminus \{v_r\}]$        ▷ Compute subgraph of $T$ induced by removing $v_r$
            if $adj_{G_{\mathcal{C}}}(T_r) = adj_{G_{\mathcal{C}}}(T'_r)$ then   ▷ Case 4a
                return $P_{entry,C,exit}$ with $v_r$ removed and replaced by shortest path
                        between vertices before and after $v_r$ in $P_{entry,C,exit}$
            else                                                                 ▷ Case 4b
                $G'[\mathcal{C}] =\leftarrow$ DIVIDE($C \setminus \{v_r\}, D, \mathbf{C}_{\max}$)
                return MERGE($G'[\mathcal{C}], subpatrols, \gamma^{B_C}, B_C, entry, exit$)
                                          ▷ $subpatrols$ reused from Line 11 in Algorithm 5
            end if
        else
            let $C_r \in \mathbf{C}$ be the cluster s.t. $v_r \in V_{C_r}$
            $subpatrolsToRepair \leftarrow \{P_{T,C,T'} \mid P_{T,C,T'} \in subpatrols \wedge C = C_r\}$
                                          ▷ $subpatrols$ reused from Line 11 in Algorithm 5
            $repairedSubpatrols \leftarrow \{\}$
            for $P_{T,C_r,T'} \in subpatrolsToRepair$ do
                if $C_r[V_{C_r} \setminus \{v_r\}]$ is connected then
                    $P'_{T,C_r,T'} \leftarrow$ REPAIRSUBPATROL($P_{T,C_r,T'}, v_r$)      ▷ Recurse
                else                                                             ▷ Case 3
                    $P'_{T,C_r,T'} \leftarrow$ COMPUTESUBPATROLDNC($C_r[V_{C_r} \setminus \{v_r\}]$,
                                                 $f, \gamma, B, B_C, T, T', \mathbf{C}_{\max}$)
                end if
                $repairedSubpatrols \leftarrow repairedSubpatrols \cup \{P'_{T,C,T'}\}$
            end for
            $subpatrols' \leftarrow subpatrols \setminus subpatrolsToRepair \cup repairedSubpatrols$
            return MERGE($G[\mathcal{C}], subpatrols', \gamma^{B_C}, B_C, entry, exit$)
                                                    ▷ $B_C$ reused from line 8 Algorithm 5
        end if
    end if
end procedure
```

Second, to ascertain how the computational cost of the algorithm scales with the number of agents and the dynamism of the environment, as compared to the theoretical complexity results from Section 5.2. Third, since the overlap between the state spaces of the broken and repaired policies are *a priori* unknown, we need to determine the performance of the repairing algorithm from Section 6.2 as a means of improving the robustness of offline computed policies.

Before detailing the experimental setup, we first summarise our key findings:

- Our algorithm outperforms a wide range of benchmark algorithms, of which the decentralised receding horizon control (RHC) algorithm developed by Stranders et al. (2010a) is the strongest competitor. We demonstrate that our multi-agent algorithm typically performs 35% better in terms of the average quality of situational awareness, and 33% better in terms of minimum quality.

- The algorithm searches a sub-exponential number of states as the number of agents increases, in contrast to what the theoretical results suggest. For 6 agents, for example, it typically searches less than 1 in 10^4 to 1 in 10^6 of the theoretically possible states (depending on the level of dynamism of the environment).

- The multi-agent patrols can only be marginally improved (up to 9%) using a thorough search of the state space (requiring 10–100 times more states to be searched). We consider this evidence for the near optimality of the multi-agent algorithm.

- The algorithm for repairing multi-agent patrols in the event of a failed agent (Section 6.2) reuses a significant amount of computation from the offline stage (typically in excess of 50%), making it an efficient method for coping with agent failure.

In what follows, we first describe the experimental setup. Then, we discuss the results for both experiments in detail.

7.1. Experimental Setup

To demonstrate the algorithm's versatility and the expressiveness of the model presented in Section 3, we present three sets of experiments in different topologies and application domains:

- In the first, the agents' goal is to patrol an environment uniformly, by minimising the time between observing each vertex of layout graph G. This corresponds to a generic patrolling task, where the agents need to periodically and homogeneously observe all locations in their environment. In fact, if "observe" is substituted by "clean", this scenario can also represent a continuous maintenance task for mobile cleaning robots in an environment where dirt accumulates at a constant rate.

- The second is similar to the first, except that we significantly scale up the size of the environment to ascertain the scalability of our algorithms.

- In the third, we change the environment entirely to demonstrate the generality of our algorithm, and task the agents tasked with monitoring an environmental phenomenon, such as temperature or gas concentration. This setting models a disaster response scenario, in which agents are deployed to keep track of the quickly changing environmental conditions, and supply commanders with up to date situational awareness.

In both domains, we derive an appropriate observation value function f, and use the following two metrics, which are based on this function, to assess the algorithm's performance:

- Average uncollected observation value over all locations of the layout graph G, averaged over all time steps T:

$$f_{avg} = \frac{1}{|V||T|} \sum_{t \in T} f \left(\bigcup_{v \in V} o_{v,t} \setminus \mathbf{O}^t_{\mathbf{A}} \right) \tag{17}$$

  For the first and second experiment, this represents the average time between two visits. For the third, it is the average root mean square error (RMSE) of the agents' estimation of the environmental phenomenon. This metric captures the average quality of the provided situational awareness over the entire space.

- Maximum uncollected observation value over all locations of the layout graph G, averaged over all time steps T:

$$f_{max} = \frac{1}{|T|} \sum_{t \in T} \max_{v} f \left(\{o_{v,t}\} \setminus \mathbf{O}^t_{\mathbf{A}} \right) \tag{18}$$

  For the first and second experiment, this is the maximum time between visits averaged over time. For the third, it is the maximum RMSE averaged over time. This metric is of key importance in safety-critical applications commonly found in disaster management and security domains, since it is a measure of the maximum risk involved in relying on the situational awareness the agents provide.

In both cases, the lower the metric, the better the algorithm performs.

Finally, we benchmarked the following algorithms each with different properties and characteristics:

GG Global Greedy, a state of the art algorithm, proposed for pursuit-evasion by Vidal et al. (2001). Global Greedy is an algorithm which moves the agents towards the vertex where the most observation value can be obtained. This algorithm exemplifies an uncoordinated approach which requires no communication and is included in our benchmark to ascertain the value coordination at the cost of increased complexity.

TSP An algorithm proposed by Sak et al. (2008), which computes the shortest closed walk that visits all vertices (similar to the TSP[14]). Agents are spaced equidistant from each other on the closed walk to minimise redundant coverage. This algorithm is included in our benchmark for the same reasons as GG.

[14] To compute the TSP cycle of the graph, we used Concorde. (`http://www.tsp.gatech.edu/concorde.html`).

RHC The receding horizon control algorithm proposed by Stranders et al. (2010a), which uses the max-sum message passing algorithm for decentralised coordination to maximise the observation value received as a team over a finite (and receding) planning horizon. Replanning occurs every 15 time steps (as per Stranders et al. (2010a)). The RHC algorithm is included to ascertain the value of planning ahead further than a finite time horizon.

NM-γ Our non-myopic multi-agent algorithm with discounting factor γ. We configure our approach with different values of γ to determine its effect on the f_{avg} and f_{max} metrics.

7.2. A Note on Statistical Significance

In the plots that follow, we use two different measures for determining statistical significance of our results. Which measure is used depends on the type of question that we attempt to answer:

1. *Is algorithm A better than algorithm B?* When comparing the performance of two or more algorithms we present the *95% confidence intervals* as determined by a paired Welch's t-test. This allows us to determine whether the performance of one algorithm is significantly (in the statistical sense of the word) better than another algorithm by determining whether their confidence intervals overlap. If they do not, we can state that the difference is significant ($p = 0.05$).
2. *What is the magnitude of a performance metric?* When determining the value of a specific performance metric (e.g. the number of states searched), we use the *standard error of the mean.* This is the standard deviation of the sample means, which gives us a good indication of the average performance of an algorithm.

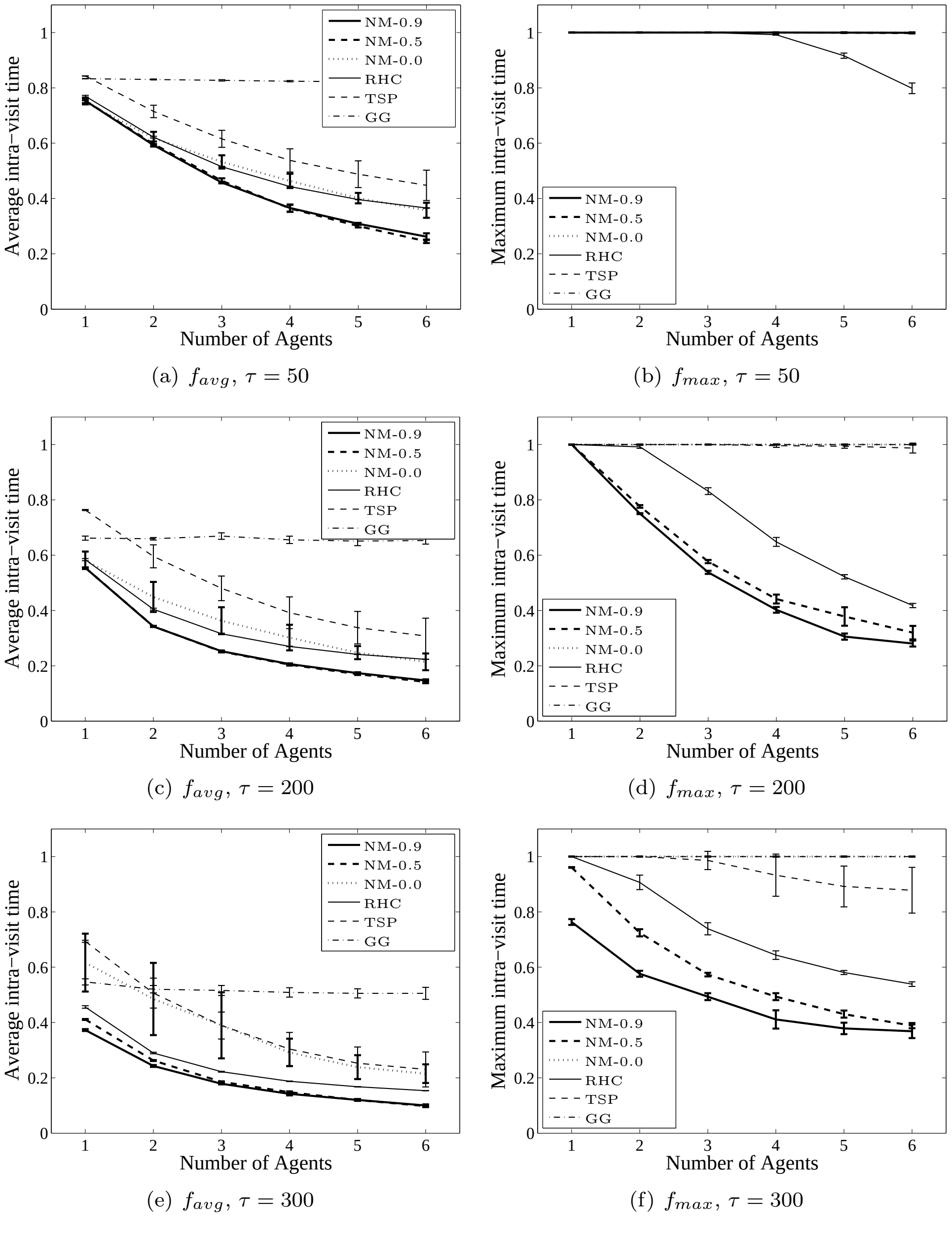
We will clearly indicate which measure is used in the captions of the plots.

7.3. Experiment 1: Minimising Intra-Visit Time

The first experiment is set in the AIC lab from Figure 3. We consider a scenario in which the value of observing a vertex is equal to the number of time steps that have elapsed since it has last been observed, with a maximum of τ (clearly, this makes observations older than τ independent from observations made at the current time step). Thus, the agent's goal is to minimise the time between two successive observations of each vertex in graph G. All agents have a circular observation area with a diameter of $1.5m$.[15]

We set $\mathbf{C}_{\max} = 6$ and $D = 25$, which results in a single-level clustering with the six clusters shown in Figure 3. We applied the methodology from Section 4.1.5 and found that a budget B of 50 leads to a partitioning of the graph in six clusters, such that agents are capable of observing all vertices within the allotted time. We then applied the algorithms listed in the previous section with a varying number of agents.

[15] Since the graph consists of lattice graphs in which the distance between adjacent vertices is 1m (Figure 5), an agent is capable of observing around 9 vertices simultaneously.



(a) f_{avg}, $\tau = 50$

(b) f_{max}, $\tau = 50$

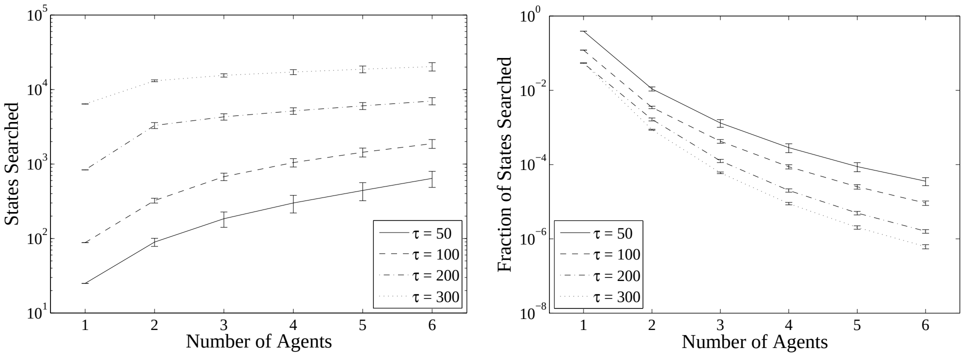
(c) f_{avg}, $\tau = 200$

(d) f_{max}, $\tau = 200$

(e) f_{avg}, $\tau = 300$

(f) f_{max}, $\tau = 300$

Figure 13: Experiment 1: the agents' performance as a team in terms of f_{avg} (Equation 17) and f_{max} (Equation 18). Visit times are normalised to 1. Error bars indicate the 95% confidence intervals.



(a) Total number of states searched by the team of agents.

(b) Reachable states as a fraction of all states.

Figure 14: The number of reachable states searched by the non-myopic algorithm. Error bars indicate the standard error of the mean.

7.3.1. Solution Quality

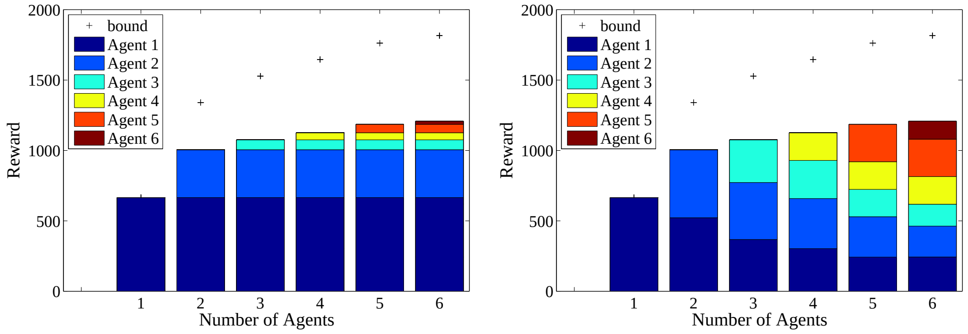
First, we analysed solution quality in terms of the average and maximum intra-visit time over 1000 time steps. To this end, we varied the temporal parameter τ; the smaller this parameter, the more dynamic the environment, the greater the need for an increased number of agents to accurately monitor the faster changing environment. Results are shown in Figure 13. From Figures 13(a), 13(c), and 13(e), we can conclude that the f_{avg} achieved by our algorithm with 6 agents is at least 33% (for $\tau = 50$) and at most $\tau = 37\%$ (for $\tau = 300$) lower than that RHC, the closest competitor. By inspecting the 95% confidence intervals, we can state that these results are statistically significant ($p = 0.05$). Furthermore, we can observe that the performance of our NM-γ algorithm with $\gamma = 0.5$ and $\gamma = 0.9$ is statistically indistinguishable in terms of f_{avg}, indicating that a long look-ahead is unnecessary in this domain.

However, Figures 13(b), 13(d), and 13(f), support a different conclusion in terms of f_{max}. Particularly for smaller number of agents and $\tau = 300$, NM-0.9 clearly outperforms NM-0.5. Moreover, the purely greedy approaches (NM-0.0 and GG) are unsuitable for minimising the maximum uncollected observation value. With a notable exception of $\tau = 50$, NM-0.9 again outperforms RHC (achieving 33% lower f_{max}). This is due to NM patrolling the graph in a more regular fashion, such that all clusters (and therefore all vertices) are visited in fixed intervals. In contrast, the RHC algorithm tends to move to a different area immediately after the majority (but not all) of value has been obtained. For $\tau = 50$, all algorithms except RHC (and 6 agents) yield $f_{max} = 1$. The reason for this is that, in order to reduce f_{max}, agents need to revisit locations within 50 time steps. Using the NM algorithm, agents spend $B = 50$ time steps in each cluster. Thus, with six agents and six clusters, f_{max} cannot drop below 1 (normalised).

Finally, as expected, Figure 13 shows that the less dynamic the environment, the better all algorithms perform.

7.3.2. Computational Overhead

Second, we considered the computational overhead of our algorithm. Figure 14(a) shows the number of states that were searched. This number is proportional to the



(a) Marginal Reward

(b) Actual Reward

Figure 15: Marginal and actual observation value received by individual agents for a single typical run of NM-0.9.

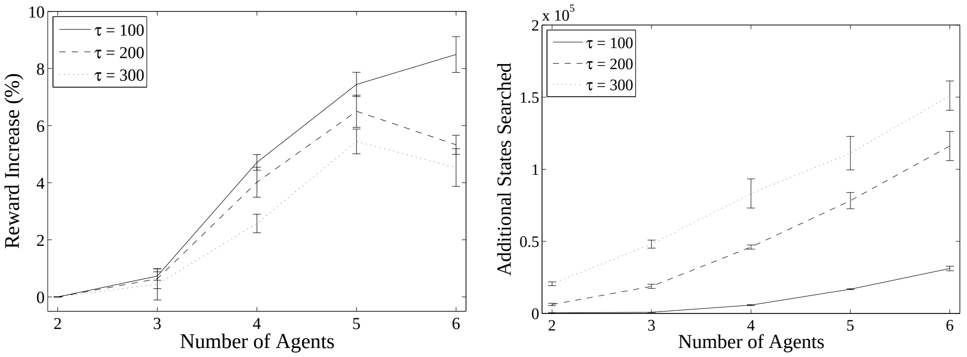
running time of the value iteration algorithm (see Section 4.1.3), which represents the bulk of the total running time of our algorithm.[16] This figure confirms that the number of states grows exponentially with τ, as indicated by the complexity results from Section 5. However, in contrast to what these results suggest, we found that the number of states is roughly linear in the number of agents, indicating that only a very small fraction of the exponentially large state space is reachable from the initial state. This is confirmed by Figure 14(b), which shows the size of the reachable state space $S_r(\overline{s})$ as a fraction of the $|\mathbf{T}|^M(\frac{\tau}{B}+1)^{|\mathbf{C}|}$ states (see Section 5.2). For 6 agents only 1 in 10^4 ($\tau = 50$) or 1 in 10^6 ($\tau = 300$) states is reachable, and needs to be searched.

7.3.3. Marginal vs. Actual Observation Value

Third, we analysed the effect of adding additional agents on the amount of observation value received by the team. Figure 15(a) shows the marginal reward for a typical run of the algorithm. Recall from Section 4.2 that the marginal reward is equal to the observation value collected by an agent, minus the penalty incurred by reducing the observation value of other agents. From this figure, it can be seen that marginal values are roughly but not strictly decreasing. This is due to the fact that some agents benefit more from their (randomly chosen) starting position than others.

Figure 15(b) shows the observation value that is actually received by the individual agents (i.e. by ignoring the penalty). Clearly, the way the penalty is defined allows for an efficient and relatively equal reallocation of reward.

Finally, both panels of Figure 15 show the theoretical bound of Corollary 1 on the observation value of the optimal policy, given the observation value obtained in this particular problem instance. For example, for 6 agents, the optimal reward is at most 50.4% higher than the reward obtained by the multi-agent algorithm. In the next section, we show that the optimal might in fact be much closer than this.



(a) Increase in Observation Value. Error bars indicate the 95% confidence intervals.

(b) Additional states searched. Error bars indicate standard error of the mean.

Figure 16: Attempting to improve policies using DSA.

7.3.4. Improving the Multi-Agent Policy

To ascertain how tight these bounds are, we attempted to find the optimal policies by searching the full joint action space, instead of sequentially computing single-agent policies. However, since not only does the state space grow exponentially with the number of agents (Section 5.2), so does the action space. This means we were unable to compute the optimal policies even for two agents. As mentioned in Section 4.2, our attempts resulted in our simulations running out of memory.

Instead, we attempted to improve the computed policies by finding joint deviations to the policies that yield higher reward. By comparing the extra effort involved in terms of the number of additional states that needs to be searched with the improvement in solution quality, we can estimate the distance between the performance of our algorithm and the (unknown) optimal.

In more detail, while being in states $s_1, \ldots, s_M$ the agents try to find a *joint* action $\mathbf{a} = [a_1, \ldots, a_M]$ that yields a higher discounted reward than following policies $\pi_1, \ldots, \pi_M$, such that the following inequality holds:

$$\sum_{i=1}^{M} R(s_i, a_i) + \gamma^B V^{\pi_i}(\delta(s_i, a_1, \ldots, a_i)) >$$
$$\sum_{i=1}^{M} R(s_i, \pi(s_i)) + \gamma^B V^{\pi_i}(\delta(s_i, \pi_1(s_1), \ldots, \pi_i(s_i))) \quad (19)$$

Computing a joint action that satisfies this equation raises two challenges. Firstly, the value functions V^{π_i} have been computed only for those states $S_r(\overline{s})$ that are reachable from the initial state $\overline{s}$ (Equation 4), given that policies of agents $\mathbf{A}_{-i}$ are fixed. Thus, joint action $\mathbf{a}$ that deviates from these policies is likely to cause several agents (with the notable exception of agent 1) to end up in a state $\hat{s} \notin S_r(\overline{s})$. Secondly,

[16] A fair comparison with RHC in terms of computational overhead is difficult, because it needs to replan every few time steps. In contrast, NM plans only once, and planning time is amortised over the mission time of the agents.

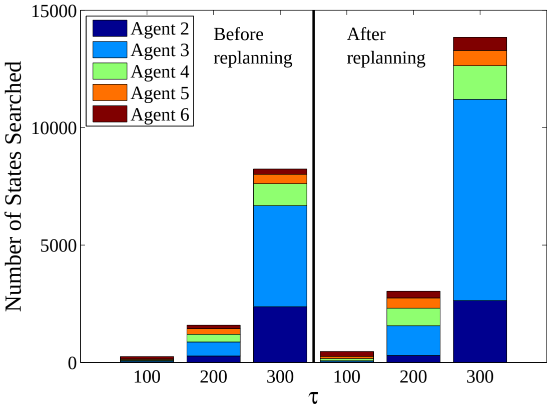


Figure 17: Impact of replanning after failure of agent 1 on number of states searched ($\gamma = 0.9$).

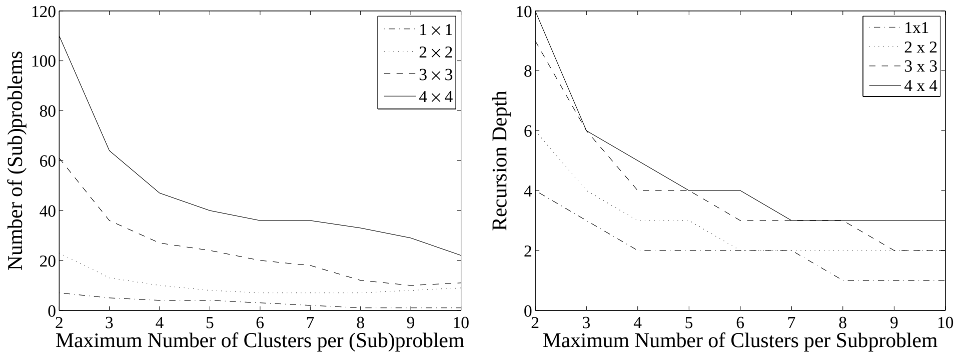
finding an action **a** that satisfies Equation 19 requires the evaluation of possibly many joint actions. As a result, evaluating Equation 19 for each of these actions can be very expensive.

Therefore, we applied the distributed stochastic algorithm (DSA) (Fitzpatrick and Meertens, 2003); an approximate decentralised coordination algorithm. While this algorithm does not necessarily yield the optimal solution, it does provide a good trade-off between computation and solution quality, and is thus a good algorithm to avoid the two problems mentioned above. Using DSA, agents iteratively take turns and choose the action conditioned on the actions chosen by others. This process is guaranteed to converge, at which point none of the agents can deviate to improve the reward received as a team.

Figure 16(a) shows that DSA is moderately effective in improving the received observation value. The improvement is statistically significant for > 4 agents ($p = 0.05$), and yields an improvement in observation value of up to 8.5%. Compare this to the bounds in Figure 15 and Corollary 1, which indicate a theoretical 50% room of improvement for 6 agents. From this, we conclude that it is likely that our algorithm performs much closer to the optimal than Corollary 1 suggests. This is largely as we expected, since it states the theoretical lower bound on achievable performance. Moreover, in order to achieve this improvement using DSA, a significantly larger portion of the state space needs to be searched. For $\tau = 100$ it searches ≈ 100 times and for $\tau = 300$ it searches ≈ 10 times more states than the multi-agent algorithm (cf. Figure 14(a)). We consider the relative lack of effectiveness of DSA as evidence for the efficiency and effectiveness of our algorithm.

7.3.5. Replanning after Agent Failure

Finally, we determined the efficiency of the replanning algorithm described in Section 6.2. We do not report the effect of repairing on the solution quality, as it critically depends on the time between failure and detection by the other agents. This is mainly determined by implementation issues, such as the communication range and protocols. However, since a team of n agents operates as a team of size $n - 1$ after plan repair, the effect of agent failure in the long run can be seen in Figure 13 by comparing the performance of n agents with that of $n - 1$ agents.



(a) The number of subproblems

(b) The height of the tree (recursion depth of the algorithm)

Figure 18: The number of subproblems and the recursion depth for solving patrolling problems in the $n \times n$ graphs.

Now, Figure 17 shows the number of states searched before and after replanning in response of the failure of agent 1, averaged over 100 runs. From this figure, we can conclude that replanning requires between 67% (for $\tau = 300$) and 91% (for $\tau = 200$) additional states to be searched. Conversely, this means that between 52% (for $\tau = 200$) and 59% (for $\tau = 300$) of the computation needed for computing the policies from scratch could be avoided, making it an efficient method of improving the robustness of the offline computed policies.

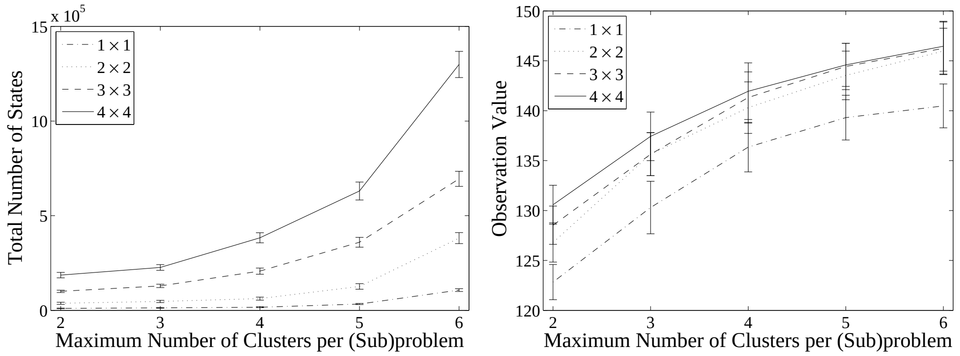
7.4. Experiment 2: Scaling Up

In the previous experiment, we used the single-level clustering of Figure 4. In this experiment, we significantly increase the size of the environment to ascertain the scalability of our full divide and conquer algorithm. To do this, we use graphs similar to Figure 6, which consists of multiple copies of the AIC lab in Figure 4 laid out in a square grid.

First, we study the effect of varying parameter $\mathbf{C}_{\max}$ (the maximum number of clusters identified by Divide) on the number of subproblems and the height of the resulting tree (cf. Figure 7). For $D = 20$, this relation is shown in Figures 18(a) and 18(b). The former corresponds to the number of calls to Algorithm 5)), while the latter is a measure of the depth of the recursion required to solve the problem. As expected, the figures show that both the number of subproblems and the height of the tree decrease with the $\mathbf{C}_{\max}$. However, since an increase in $\mathbf{C}_{\max}$ causes a worst-case exponential increase in computation (Section 5.2), further empirical study is required to ascertain the effect of this on computation, as well as on the solution quality.

7.4.1. A Single Agent

To do this, we applied the single-agent algorithm to the graphs of 1 × 1, 2 × 2, 3 × 3 and 4 × 4 copies of the AIC lab with $\tau = 200$ and $\gamma = 0.9$ (given the good results obtained for this value in Experiment 1). Figure 19(a) shows the relation between the total number of (reachable) states in the MDP created by the Merge operation. We report results $\mathbf{C}_{\max} \in [2, 6]$. Beyond this interval, our implementation of the algorithm



(a) Total number of states

(b) The observation value collected by a single sensor

Figure 19: Computation vs. solution quality as a function of $\mathbf{C}_{\max}$ for the $n \times n$ graphs.

ran out of memory due to the large number of states. As expected, the number of states increases exponentially with parameter $\mathbf{C}_{\max}$. However, it is very important to note that the number of states does *not* grow exponentially with the size of the environment, but rather linearly (division by n^2 yields a number of states that is statistically indistinguishable across the size of the graphs). The explanation for this can be found in Figure 18(a): the increase in the number of subproblems is roughly polynomial in n for the $n \times n$ graphs. From this we can conclude that our algorithm scales poorly in $\mathbf{C}_{\max}$, but well in the size of the layout graph of the problem.

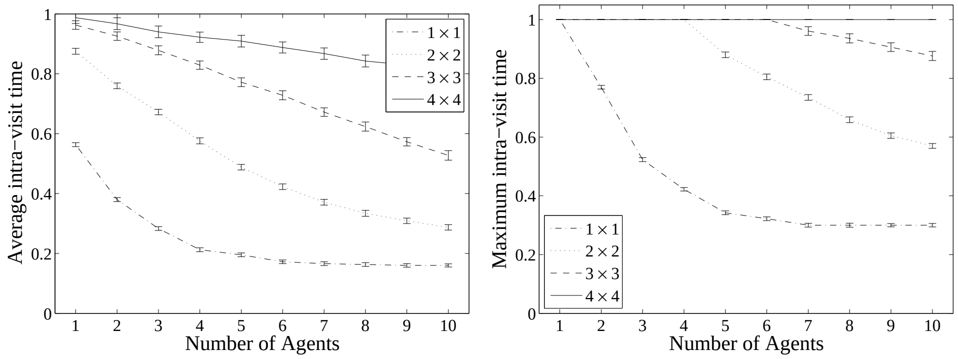
The main question is now: what is the effect of $\mathbf{C}_{\max}$ on solution quality? Figure 19(b) shows the observation value collected by the single agent.[17] As can be observed in this figure, the observation value collected by the agent increases monotonically with $\mathbf{C}_{\max}$. The probable explanation for this is that as $\mathbf{C}_{\max}$ increases, the number of movements of the agent within the graph increases. As a result, increasing $\mathbf{C}_{\max}$ means that the solution space is more thoroughly searched, leading to better solutions. However, it is important to note that this increase seems to level off. Thus, increasing $\mathbf{C}_{\max}$ yields an exponential increase in computation, but a decrease in marginal solution quality.

7.4.2. Multiple Agents

We also studied the performance of the multi-agent algorithm on these large problem instances. We set $\mathbf{C}_{\max} = 6$, which is the maximum value before the experiments ran out of memory. The results in terms of f_{avg} and f_{max} are shown in Figures 20(a) and 20(b).

These results largely conform the results of Experiment 1. In addition, we now see that in larger graphs, there is a more constant decrease of f_{avg} as the number of agents is increased compared to smaller graphs. For example, in the 1×1 graph, f_{avg} no

[17] We do not report metrics f_{avg} or f_{max} here because a single agent is not able to patrol these large environment by itself. As a result, a single agent performs much better in terms of these metrics in smaller graphs than in larger ones. As a result, unlike the amount of observation value collected by the agent, f_{avg} and f_{max} are not comparable.



(a) f_{avg}

(b) f_{max}

Figure 20: Experiment 2: the agents' performance as a team in terms of f_{avg} and f_{max} in $n \times n$ graphs.

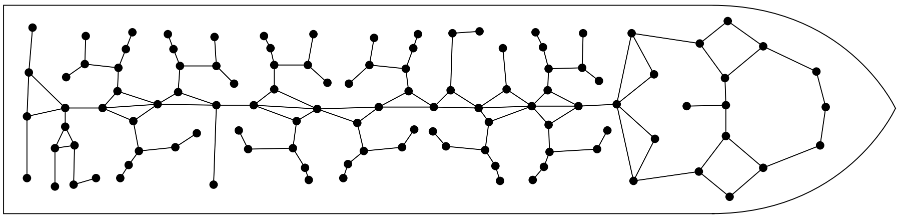


Figure 21: The ship layout graph used in Experiment 3.

longer decreases after 7 agents have been deployed, while f_{avg} keeps decreasing in the 2×2 graph. The explanation for this is that agents are less likely to overlap in larger graphs, and therefore contribute more. Furthermore, we see that a limited number of agents with limited sensing range are increasingly less capable of reducing f_{max}, indicating that the large 4×4 graphs need a considerable number of agents to patrol effectively. Unfortunately, for these instances we were unable to add more agents, because of limited memory on the machine used for these experiments. However, it is worth noting that our algorithm was typically capable of solving the hardest problem within two hours.

7.5. Experiment 3: Monitoring Environmental Phenomena

In third experiment, we study the performance of the algorithm in a different layout graph with a different observation value function. In more detail, the agents are tasked to patrol a ship (Figure 21) while monitoring an environmental phenomena that is represented by a real valued field that varies over space and time (such as temperature, radiation, pressure and gas concentration). The key challenge in monitoring such phenomena is to predict its value at unobserved coordinates (in both time and space) based on a limited number of observations. Recent work has addressed this challenge by modelling the spatial and temporal dynamics of the phenomena using Gaussian processes (GPs) (Rasmussen and Williams, 2006). GPs are a powerful Bayesian approach for inference about functions, and have been shown to be an effective tool for capturing the dynamics of spatial phenomena (Cressie, 1993). In this experiment, we use a GP to model the environmental phenomenon, and obtain a principled measure of observation value.[18]

In more detail, let $\mathbf{X}$ denote the matrix with the coordinates at which observation were made, and vector $\mathbf{y}$ the observed value of field $\mathcal{F}$ at those coordinates. Then, the predictive distribution of the observation at coordinates $\mathbf{x}_*$ is Gaussian with mean μ and variance σ^2 is given by:

$$\mu = K(\mathbf{x}_*, \mathbf{X})K(\mathbf{X}, \mathbf{X})^{-1}\mathbf{y} \tag{20}$$

$$\sigma^2 = K(\mathbf{x}_*, \mathbf{x}_*) - K(\mathbf{x}_*, \mathbf{X})K(\mathbf{X}, \mathbf{X})^{-1}K(\mathbf{X}, \mathbf{x}_*) \tag{21}$$

where $K(\mathbf{X}, \mathbf{X}')$ denotes the covariance matrix for all pairs of rows in $\mathbf{X}$ and $\mathbf{X}'$. This matrix is obtained by evaluating a function $k(\mathbf{x}, \mathbf{x}')$, called a covariance function, which encodes the spatial and temporal correlations of the pair $(\mathbf{x}, \mathbf{x}')$. Generally, covariance is a non-increasing function of the distance in space and time. A typical choice for modelling smooth phenomena is the squared exponential function where the covariance decreases exponentially with this distance:

$$k(\mathbf{x}, \mathbf{x}') = \sigma_f^2 \exp\left(-\tfrac{1}{2}|\mathbf{x} - \mathbf{x}'|^2/l^2\right) \tag{22}$$

where σ_f and l are called *hyperparameters* that model the signal variance and the length-scale of the phenomenon respectively. The latter determines how quickly the phenomenon varies over time and space.[19]

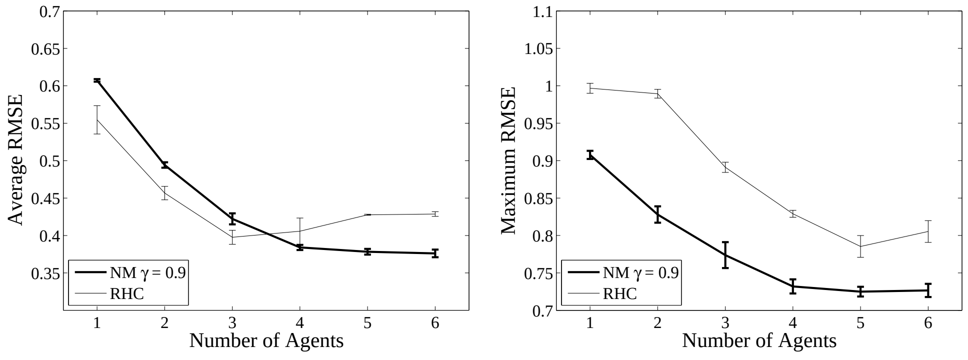
One of the features of the GP is that the posterior variance in Equation 21 is independent of actual measurements $\mathbf{y}$. This allows the sensors to determine the variance reduction that results from collecting samples along a certain path without the need of actually collecting them. By exploiting this feature, we define the value $f(O)$ to be the *reduction in entropy* that results from making observations O. The magnitude of this entropy reduction is a function[20] of the variance computed in Equation 21.

For this experiment, we simulated an environmental phenomenon with a spatial length-scale of 5 and a temporal length-scale of 50. This corresponds to $\rho = 10$ (with $\epsilon < 0.01$) and $\tau = 100$ (with $\delta < 0.01$). These parameters were chosen during initial calibration to generate difficult problem instances. If, for example, the temporal length-scale is very short, the phenomena changes so rapidly that each time step presents a new and independent problem from the last, for which the trivial solution is to ensure the agents are spread out sufficiently to minimise redundant coverage. In such settings, our algorithm assigns the first agent to repeatedly patrol the biggest cluster. The second agent is assigned to repeatedly patrol the second biggest cluster, etc. Agents never leave their assigned cluster, because after every sweep through a cluster all previously made observations within that cluster have become stale, and the next sweep again results in maximum observation value. If, in contrast, the temporal length-scale is very long, the phenomenon is almost static, in which case a single traversal of the environment suffices. In such settings, our algorithm computes a patrol

[18] While GPs are flexible and powerful tools for performing inference over a large class of functions, it should be noted that our choice for this experiment is by no means central to our work, and other approaches could equally well be applied.

[19] A slightly modified version of Equation 22 allows for different length-scales for the spatial and temporal dimensions of the process.

[20] The entropy of a normal distribution with variance σ^2 is $\frac{1}{2}\ln(2\pi e\sigma^2)$



(a) f_{min} (b) f_{max}

Figure 22: Experiment 3: the agents' performance as a team in terms of f_{avg} (Equation 17) and f_{max} (Equation 18). RMSE is normalised to 1. Error bars indicate the 95% confidence intervals.

in which the agents visit each cluster once as soon as possible, while simultaneously attempting to prioritise the bigger clusters (due to discount factor γ). After each cluster has been visited, no more observation value can be obtained, and the agents "aimlessly wander" around the graph.

For the more challenging problem instances we consider here, however, the environmental phenomenon has a strong correlation along the temporal dimension (i.e. it varies slowly over time), and relatively weak correlations along the spatial dimension (i.e. it varies quickly in space). As a result, the agents' goal is a mix between the two extreme settings described earlier (i.e. with a very long or a very short temporal length-scale). As a result, the agents' priority is to visit each cluster as quickly as possible while prioritising more valuable clusters, before settling into a patrolling routine that revisits clusters regularly.

The results in terms of f_{avg} and f_{max} are shown in Figure 22. We benchmarked the best NM configuration ($\gamma = 0.9$) from Experiment 1 with its closest competitor, RHC, in 100 simulations.[21] The pattern observed here is similar to that in the first experiment, in that NM outperforms the RHC algorithm in terms of both f_{avg} and f_{max}. More specifically, NM reduces average RMSE by approximately 14% and maximum RMSE by 10% for 6 agents compared to the RHC algorithm (results are statistically significant with $p = 0.05$). Moreover, while the marginal performance increase exhibited by the non-myopic algorithm is guaranteed to be positive (adding agents never hurts), the performance of the RHC algorithm starts to decline after adding the fourth agent.[22]

In summary, the empirical results presented in this section show that our algorithm outperforms the RHC algorithm (Stranders et al., 2010a) (which does not give performance guarantees) in most cases, both in terms of average performance, as well as

[21] The results for the TSP and GG algorithms were similar to Experiment 1 and are omitted.

[22] This is caused by the max-sum algorithm that lies at the foundation of the RHC algorithm; as the coordination graph between the agents becomes denser (i.e. agents need to coordinate with increasingly many neighbours), the factor graph contains more cycles causing max-sum to output increasingly less optimal solutions (Rogers et al., 2011).

worst-case performance. Furthermore, to test the tightness of the theoretical bound on the optimal multi-agent patrol, we attempted to improve the policies using a decentralised coordination algorithm. We found that the room for improvement is much smaller than the theory suggests, indicating that the multi-agent policies are closer to the optimal than the bound indicates. In terms of efficiency, the results showed that the approach of sequential allocation empirically results in a sub-exponential increase of the number of states searched, in contrast to the computational complexity results. Finally, the results show that the repair algorithm is efficient, reducing computation by more than half compared to computing policies from scratch.

8. Conclusions and Future Work

In this paper, we developed an approximate non-myopic algorithm for continuous patrolling with multiple information gathering agents. Unlike previous work, this algorithm is geared towards environments that exhibit the property of *temporality*, which models very dynamic situations. As a consequence, agents must periodically (and infinitely often) revisit locations to provide up-to-date situational awareness. This algorithm gives strong performance guarantees, making it suitable for deployment in security and life-critical domains.

In more detail, the single-agent algorithm uses a divide and conquer strategy and, as such, follows a three-step computation: (i) decompose the environment into clusters, (ii) compute subpatrols within each cluster, and (iii) concatenate these subpatrols to form the desired patrol. The multi-agent algorithm computes a near-optimal multi-agent patrol by iteratively computing single-agent patrols that maximise their marginal contribution to the team. To do this, we modified the reward structure of single agents to incentivise agents to the reward left behind by their predecessors. The novelty of this algorithm is the application of sequential allocation for the computation of a joint policy, which allows the algorithm to (empirically) scale much better than an algorithm that searches the entire joint policy space.

We also developed two repairs algorithm to improve the robustness of the multi-agent patrols in the event of failure of one or more agents or changes in the graph. Using the former algorithm, the remaining functioning agents compensate for the loss of possibly multiple malfunctioning agents by adopting the patrols of their predecessors. Once repaired, the agents' patrols are identical to those obtained through recomputing them from scratch using the multi-agent algorithm. We show that significant parts of the previously computed patrols can be reused (typically in excess of 50%), making it an efficient method for coping with system failure. The latter algorithm exploits the recursive nature of the divide and conquer algorithm to repair the portion of the patrol that was affected by the removal of a vertex in the graph. In so doing, it reuses as much computation as possible from the single-agent algorithm.

Our algorithms can be applied in a wide range of domains which exhibit the general properties of submodularity, temporality and locality. We demonstrated this by benchmarking their performance in two distinct domains, in which the agents' goal was to minimise the time between visiting locations, and to monitor a continuously changing environmental phenomenon. Compared to the closest competitor (RHC), our algorithm typically performs 35% better in terms of the average quality of situational awareness, and 33% better in terms of minimum quality. Crucially, unlike the RHC algorithm, our algorithm provides strong performance guarantees, which are essential in safety critical domains where a lower bound is often required on the worst-case behaviour.

Future work will focus on extending our algorithm to settings with strategic opponents. In this paper, we have considered environments that are non-strategic, i.e. these environments do not behave so as to further their interest, since they simply had none, or were assumed to have none. However, some scenarios are clearly intrinsically strategic. An example of this is the pursuit evasion domain, in which the agents' objective is to capture a moving target, whose goal is to prevent itself from being captured. Moreover, assuming the environment behaves strategically — even when it does not — is equivalent to being fully risk averse, in the sense that good solutions to this problem seek to minimise the maximum risk the agents (and their owners) are exposed to. In safety-critical and hostile scenarios, this is often a desirable trait.

The main challenge in extending our work to strategic patrolling is the need for radically different techniques. In particular, as discussed in Section 2, the strategic nature of the opponent requires game theoretic concepts and algorithms to address this challenge. For instance, a common assumption in strategic patrolling is that the attacker has full knowledge of the agents' strategy. This is often modelled using the game theoretic solution concept of a Stackelberg equilibrium. This solution concept is characterised as a two phase game: the agents choose their patrols first, after which the attacker chooses the attack location that maximises its expected payoff in response to these patrols. Solving a Stackelberg equilibrium involves solving a partially observable stochastic game (POSG), which, in turn, is often solved using mathematical (integer) programming (Basilico et al., 2009). Future work will focus on investigating whether a POSG can replace the MDP used in the Merge operation, while using Divide and Conquer in their current form. In addition, since the divide and conquer strategy of our algorithm results in approximate solutions, the solution obtained by a strategic version of it is highly likely to be approximate as well. It will be particularly interesting to bound the quality of the solution, for example by determining whether an algorithm can be developed that computes an ϵ-Stackelberg equilibrium, i.e. one that is at most ϵ away from the true Stackelberg equilibrium.

A different way of extending our work to strategic patrolling is through the use *automated abstractions* (Sandholm and Singh, 2012; Basilico and Gatti, 2011), a technique for reducing the state and action space of a stochastic game by combining atomic actions (e.g. moves) into short patrols. Similar to our method of recursively subdividing the physical environment, automated abstractions yield theoretical guarantees on the (suboptimal) solution, while ensuring scalability of the algorithms on large problem instances. An interesting direction of future research would be to augment our divide and conquer strategy with this technique to solve adversarial patrolling games.

Acknowledgements

The work in this paper was done as part of the ORCHID project (`www.orchid.ac.uk`).

References

Agmon, N., Kraus, S., Kaminka, G. A., 2008. Multi-robot perimeter patrol in adversarial settings. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 2339–2345.

Ahmadi, M., Stone, P., 2006. A multi-robot system for continuous area sweeping tasks. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 1724–1729.

Basilico, N., Gatti, N., 2011. Automated abstractions for patrolling security games. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence. pp. 1096–1101.

Basilico, N., Gatti, N., Amigoni, F., 2009. Leader-follower strategies for robotic patrolling in environments with arbitrary topologies. In: Proceedings of the Eighth International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS). Budapest, Hungary, pp. 57–64.

Bellman, R., 1957. Dynamic programming. Princeton University Press.

Christofides, N., 1976. Worst-case analysis of a new heuristic for the travelling salesman problem. Tech. rep., Carnegie Mellon University, Graduate School of Industrial Administration.

Cormen, T. H., Stein, C., Rivest, R. L., Leiserson, C. E., 2009. Introduction to Algorithms, 3rd Edition. McGraw-Hill Higher Education.

Cressie, N., 1993. Statistics for Spatial Data. Wiley-Interscience.

Delle Fave, F. M., Rogers, A., Xu, Z., Sukkarieh, S., Jennings, N. R., 2012. Deploying the max-sum algorithm for coordination and task allocation of unmanned aerial vehicles for live aerial imagery collection. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 469–476.

Edachery, J., Sen, A., Brandenburg, F., 1999. Graph clustering using distance-k cliques. In: Kratochvyl, J. (Ed.), Graph Drawing. Vol. 1731 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 98–106.

Elmaliach, Y., Agmon, N., Kaminka, G. A., 2009. Multi-robot area patrol under frequency constraints. Annals of Math and Articial Intelligence journal (AMAI) 57, 293–320.

Fiorelli, E., Leonard, N., Bhatta, P., Paley, D., Bachmayer, R., Fratantoni, D., 2006. Multi-AUV control and adaptive sampling in Monterey Bay. IEEE Journal of Oceanic Engineering 31 (4), 935 – 48.

Fitzpatrick, P., 2009. Unmanned aircraft hurricane reconnaissance. In: Proceedings of the Twenty-Fifth Gulf of Mexico Information Transfer Meeting. pp. 47–48.

Fitzpatrick, S., Meertens, L., 2003. Distributed coordination through anarchic optimization. In: Lesser, V., Ortiz, Jr., C. L., Tambe, M. (Eds.), Distributed Sensor Networks. Kluwer Academic Publishers, Ch. 11, pp. 257–295.

Grocholsky, B., 2002. Information-theoretic control of multiple sensor platforms. Ph.D. thesis, University of Sydney.

Gross, J., Yellen, J., 1999. Graph theory and its applications. CRC Press, Inc., Boca Raton, Florida, USA.

Guestrin, C., Krause, A., Singh, A. P., 2005. Near-optimal sensor placements in gaussian processes. In: Proceedings of the Twenty-Second International Conference on Machine Learning (ICML). pp. 265–272.

Howard, R. A., 1960. Dynamic Programming and Markov Processes. The MIT Press, Cambridge, Massachusetts, USA.

Karp, R. M., 1982. Dynamic programming meets the principle of inclusion and exclusion. Operations Research Letters 1 (2), 49 – 51.

Ko, C. W., Lee, J., Queyranne, M., 1995. An exact algorithm for maximum entropy sampling. Operations Research 43 (4), 684–691.

Krause, A., Guestrin, C., Gupta, A., Kleinberg, J., 2006. Near-optimal sensor placements: Maximizing information while minimizing communication cost. In: Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN). ACM Press, New York, NY, USA, pp. 2–10.

Levin, D. A., Peres, Y., Wilmer, E. L., 2009. Markov Chains and Mixing Times. American Mathematical Society.

Littman, M. L., Dean, T. L., Kaelbling, L. P., 1995. On the complexity of solving Markov Decision Problems. In: Proceedings of the Eleventh International Conference on Uncertainty in Artificial Intelligence (UAI). Montreal, Quebec, Canada, pp. 394–402.

Martinez-Cantin, R., de Freitas, N., Doucet, A., Castellanos, J. A., 2007. Active policy learning for robot planning and exploration under uncertainty. In: Proceedings of Robotics: Science and Systems.

Maza, J. I., Caballero, F., Capitan, J., Martinez de Dios, J. R., Ollero, A., 2011. Experimental results in multi-uav coordination for disaster management and civil security applications. Journal of Intelligent and Robotic Systems 61 (1-4), 563–585.

Meila, M., Pentney, W., 2007. Clustering by weighted cuts in directed graphs. In: Proceedings of the Seventh SIAM International Conference on Data Mining.

Meliou, A., Krause, A., Guestrin, C., Hellerstein, J. M., 2007. Nonmyopic informative path planning in spatio-temporal models. In: Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI). Vancouver, British Columbia, Canada, pp. 602–607.

Moore, A. W., Atkeson, C. G., 1993. Prioritized sweeping: Reinforcement learning with less data and less time. Machine Learning 13 (1), 103–130.

Moran, S., 1984. On the length of optimal TSP circuits in sets of bounded diameter. Journal of Combinatorial Theory, Series B 37 (2), 113 – 141.

Nemhauser, G. L., Wolsey, L. A., 1978. An analysis of approximations for maximising submodular set functions—I. Mathematical Programming 14 (1), 265—294.

Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V. V., 2007. Algorithmic Game Theory. The MIT Press.

Paruchuri, P., Pearce, J., Tambe, M., Ordonez, F., Kraus, S., 2007. An efficient heuristic approach for security against multiple adversaries. In: Proceedings of the Sixth International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS). Honolulu, Hawaii, USA, pp. 1–8.

Puterman, M. L., 1994. Markov Decision Processes—Discrete Stochastic Dynamic Programming. John Wiley & Sons, Inc., New York, New York, USA.

Puterman, M. L., Shin, M. C., 1978. Modified policy iteration algorithms for discounted markov decision problems. Management Science 24 (11), 1127–1137.

Rasmussen, C. E., Williams, C. K. I., 2006. Gaussian Processes for Machine Learning. The MIT Press.

Rogers, A., Farinelli, A., Stranders, R., Jennings, N. R., 2011. Bounded approximate decentralised coordination via the max-sum algorithm. Artificial Intelligence 175 (2).

Sak, T., Wainer, J., Goldenstein, S. K., 2008. Probabilistic multiagent patrolling. In: Proceedings of the Brazilian Symposium on Artificial Intelligence (SBIA). pp. 124–133.

Sandholm, T., Singh, S., 2012. Lossy stochastic game abstraction with bounds. In: Proceedings of the Thirteenth ACM Conference on Electronic Commerce. ACM, pp. 880–897.

Satuluri, V., Parthasarathy, S., 2011. Symmetrizations for clustering directed graphs. In: Proceedings of the 14th International Conference on Extending Database Technology. pp. 343–354.

Schaeffer, S. E., 2007. Graph clustering. Computer Science Review 1 (1), 27–64.

Singh, A., Krause, A., Guestrin, C., Kaiser, W., 2009. Efficient informative sensing using multiple robots. Journal of Artificial Intelligence Research (JAIR) 34, 707–755.

Singh, A., Krause, A., Guestrin, C., Kaiser, W. J., Batalin, M. A., 2007. Efficient planning of informative paths for multiple robots. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI). Hyderabad, India, pp. 2204–2211.

Stranders, R., Delle Fave, F. M., Rogers, A., Jennings, N. R., 2010a. A decentralised coordination algorithm for mobile sensors. In: Proceedings of the Twenty-Fourth National Conference on Artificial Intelligence (AAAI). Atlanta, Georgia, USA, pp. 874–880.

Stranders, R., Farinelli, A., Rogers, A., Jennings, N. R., 2009. Decentralised coordination of mobile sensors using the max-sum algorithm. In: Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI). Pasadena, California, USA, pp. 299–304.

Stranders, R., Rogers, A., Jennings, N. R., 2010b. A decentralised coordination algorithm for maximising sensor coverage in large sensor networks. In: Proceedings of the Ninth International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS). Toronto, Canada, pp. 1165–1172.

Tammet, T., Vain, J., Puusepp, A., Reilent, E., Kuusik, A., 2008. RFID-based communications for a self-organising robot swarm. In: Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems. pp. 45–54.

Thrun, S., Burgard, W., Fox, D., 2005. Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). The MIT Press.

Tsai, J., Yin, Z., Kwak, J., Kempe, D., Kiekintveld, C., Tambe, M., 2010. Urban security: Game-theoretic resource allocation in networked domains. In: Proceedings of the Twenty-Fifth National Conference on Artificial Intelligence (AAAI). Atlanta, Georgia, USA, pp. 881–886.

Vidal, R., Rashid, S., Sharp, C., Jin, S., Sastry, S., 2001. Pursuit-evasion games with unmanned ground and aerial vehicles. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA). pp. 2948–2955.