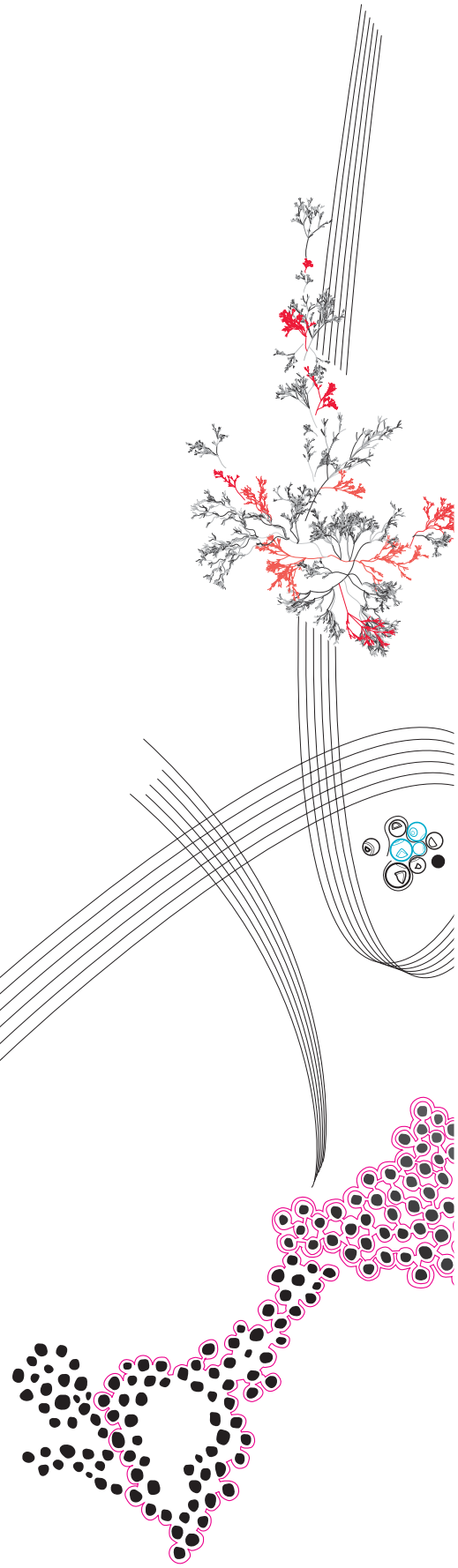

MEng Computer Science
Individual Project

GEM: Online Coverage Path Planning via Greedy Entropy Maximization

Kale Champagnie

Supervisor: Junyan Hu

April, 2023

Department of Computer Science
University College London

Contents

Abstract

We present GEM, a novel approach to online coverage path planning in which a swarm of homogeneous agents act to maximize the entropy of pheromone deposited within their environment. We show that entropy maximization (EM) coincides with many conventional goals in offline coverage path planning, while also generalizing to online settings. We evaluate our approach by measuring the rate at which entropy is maximized within a variety of static and dynamic environments. Our experimental results demonstrate that GEM achieves state-of-the-art performance in online coverage, competitive with offline methods, despite requiring no direct communication among agents. Finally, we comment on potential improvements to GEM and discuss the utility of EM as a general framework for coverage path planning.

Keywords: coverage path planning, multi-agent systems, swarm intelligence.

Chapter 1

Introduction

Coverage Path Planning (CPP) refers to the task of directing one or more mobile agents such that they collectively explore the entirety of a given area, while also avoiding obstacles [13]. Generally, CPP involves two interrelated processes — viewpoint generation and path generation . Viewpoint generation involves specifying a set of regions within the target area (viewpoints), such that their union encompasses the whole area . The geometry and arrangement of these viewpoints is typically contingent on the radius of each agent's sensing device . For instance, coverage path planning algorithms designed for UAVs (Unoccupied Aerial Vehicles) may utilize relatively large viewpoints, owning to the extensive sensing range of drone-mounted cameras .

In this work, we focus on developing a novel CPP approach that takes inspiration from the collective behaviour of ants to explore arbitrary, highly dynamic environments in an efficient, distributed manner. Unlike many existing approaches, our method is entirely decentralized and requires no direct communication between agents, instead relying on indirect pheromonal signalling. We additionally propose a generalised framework for evaluating the performance of CPP solutions based on the notion that *successful coverage*, in most cases reduces to *uniform exploration* of the environment, such that each viewpoint receives a similar number of visits from the team of agents.

CPP has been applied to numerous fields, including search and rescue operations [4], household robotics [23], agricultural monitoring [23], exploration of hazardous environments (e.g. due to unsafe levels of radiation) [10], aerial surveillance [28], and broader applications in mathematics related to efficient coverage of discrete and continuous state-spaces . We elaborate on several of these applications below.

- **Search and Rescue Operations.** CPP has emerged as a critical tool for search and rescue operations, enabling rescue teams to more efficiently explore large areas in the search for missing individuals. In the context of such operations, CPP provides a robust and repeatable method for devising search paths for rescue boats, UAVs, and other mobile agents participating in the search .
- **Household Robotics.** In domestic settings, CPP finds a wide range of applications, involving automated traversal of indoor and outdoor environments. Perhaps the most common is the use of CPP for planning the trajectories of vacuuming robots and other automated cleaning devices. Such applications often involve relatively restricted, local viewpoints and consequently, extensive visitation to each position within the target are .

- **3. Mathematical Applications.** Apart from such practical applications as have been described thus far, effective CPP solutions also yield substantive theoretic importance within the broader fields of discrete mathematics and computer science. This is primarily due to the fact that CPP algorithms are often formulated on the basis that the environment can be represented as a graph with vertices corresponding to particular states and edges corresponding to agent-instigated transitions between said states . CPP then equates to efficiently identifying and exploring each vertex within the state-space.

Overall, CPP is a crucial area of research within the broader field of robot path planning, with numerous applications in a diverse range of fields.

1.1 Contributions

In this work, we make a distinction between two categories of CPP approaches: namely, *offline* and *online* CPP. The latter involves the generation of viewpoints and complete path plans prior to executing those plans in the environment [13]. In contrast, *online* CPP methods construct plans in real-time, i.e. as agents are actively exploring the environment [27]. Both offline and online approaches have respective merits and limitations, however as we later discuss, online methods are often preferable in many real-world environments that exhibit dynamic or unpredictable changes.

In this project, we aim to generalise CPP by proposing a general metric known as *uniformity*. Uniformity measures how evenly distributed visits to each viewpoint are over time. As we show in later sections, maximizing uniformity, naturally results in strong performance on conventional metrics and benchmarks. However, unlike existing metrics, uniformity does not make restrictive assumptions about the environment, such that it is static [13]. Under this framework, we then propose a novel online CPP method known as Greedy Entropy Maximization (GEM). GEM aims to maximize uniformity globally through local *greedy* actions taken by individual agents. Finally, we aim to implement GEM and the uniformity framework as an open-source Python package to aid future students and researchers in contributing to progress in CPP. In summary, our contributions are as follows:

1.1.1 Uniformity

We develop a generalised framework for CPP which applies to both offline and online approaches. This relies on the definition of a generalised *uniformity* metric which measures coverage performance in arbitrary environments.

1.1.2 GEM

We present a novel *online* CPP method (GEM), which maximizes uniformity in a greedy fashion. Using a greedy approach may allow for a simple, elegant and highly computationally efficient solution which also achieves strong performance.

1.1.3 PyCPP

We develop an open-source library for conducting CPP research. The library should specifically provide tooling for simulating CPP algorithms in static, dynamic and interactive (i.e.

user-influenced) environments.

1.2 Methodology

This project is multi-facet, involving both theoretical and practical contributions to the field of CPP. As such, a clear development strategy was required to ensure consistent progress. We discuss our general development strategy in this section.

1.2.1 Development Phases

Development during this project can be divided into two primary stages - preliminary experiments, and active development. The preliminary stage was a period of approximately six weeks reserved for research, prototyping and re-implementing existing CPP methods. The purpose of this phase was simply to become familiar with the field as a whole, as well as concrete solutions developed in previous works. The deliverables of this phase were several implementations of well-known offline CPP algorithms such as DARP [17] in Python, and a list of ideas for novel approaches to the CPP task. This list was gradually reduced as it became apparent which proposals were more likely to be successful.

After this preliminary phase, began active development in which the proposed solution was taken from a prototype into a viable product. This phase was considerably more involved than the preliminary experiments and was organized into 1-2 week long sprints with regular supervisor meetings, and importantly, routine feedback from potential users of our solution.

1.2.2 Development Strategy

The specific strategy utilized during active development may be characterized as a form of Agile with an emphasis on test-driven design (TDD) and user-centred design (UCD) [3]. We discuss this in more detail in chapters 4 and 6. The justification for choosing this approach was that it allowed for flexibility and the ability to quickly produce new prototypes improving upon prior iterations. Constant user-feedback in the form of questionnaires, and later beta-testing was used to align the project's goals so they remained consistent with the genuine needs of the target audience.

1.3 Report Structure

The remainder of this report documents the key aspects of the project, namely a novel online CPP solution (GEM), a generalised framework for CPP based on uniformity, and a concrete implementation of these ideas into an open-source software library for CPP research in Python.

- **Chapter 2 - Background.** Conducts an in-depth review of traditional and state-of-the-art offline and online CPP methods, discussing their formulations, merits and limitations.
- **Chapter 3 - Uniformity.** Introduces a general framework for CPP based on the understanding that successful coverage equates to uniform exploration of the environment over time. We discuss the intuitions behind our framework and then formally prove its generality by encoding conventional coverage objectives in terms of maximizing uniformity.

- **Chapter 4 - GEM: Greedy Entropy Maximization.** Introduces the GEM algorithm for decentralized online coverage path planning. We provide an extensive evaluation of the algorithm and compare it to a theoretically ideal solution and existing state-of-the-art methods.
- **Chapter 5 - Design and Implementation.** Describes our open-source CPP framework in terms of requirements, design principles, architecture and specific implementation details.
- **Chapter 6 - Testing.** Describes our multi-factor approach to testing we used in this project.
- **Chapter 7 - Conclusion.** Gives a final evaluation of the project's results and achievements. We discuss to what extent the project was successful in achieving its goals and requirements.

1.4 Summary

Offline and online coverage path planning are two forms of CPP with respective merits and limitations; while offline approaches allow for rigorous optimization of generated paths, they suffer from limited applicability in dynamic environments due to overly-restrictive assumptions. Online approaches alleviate many of these issues, however are difficult to directly compare due to the lack of consensus about objective goals for coverage algorithms in dynamic settings.

We propose a novel lens with which to view coverage path planning, based on a generalised uniformity metric and develop an efficient online CPP algorithm accordingly. We implement our proposals as an open-source algorithm which both aids in demonstrating the effectiveness of the solutions, as well as providing a useful platform for future researchers and students interested in CPP.

Chapter 2

Background

2.1 Introduction

Coverage path planning (CPP) is the process of directing one or more mobile agents such that they collectively explore the entirety of an area of interest (AOI) [13]. An adaption of regular path planning, CPP finds use whenever a specific region requires extensive attention through automated means. Instances of CPP problems appear within a wide variety of contexts and fields, from inspection of hazardous environments to surface cleaning robots, to game design . Due to the commonality of such problems, there has been wide spread efforts to improve current state-of-the-art (SOTA) in various aspects, including coverage efficiency, scalability , and generality (i.e. how many distinct environments a particular method performs effectively in). Along the same lines, this work aims to develop an entirely general CPP algorithm that can be applied to arbitrary static or dynamic environments. However, before proposing our solution, we use this chapter to engage in a detailed discussion of existing literature and other background to contextualize our method.

2.2 Components of Coverage Path Planning

Although individual CPP problems may be highly specific, they each share some fundamental aspects in common which we can discuss and reason about abstractly.

2.2.1 Environments

A key aspect of any given CPP problem is the environment in which agents will be situated, for instance, urban environments such as city roads and building interiors, or less structured settings such as those found in marine expeditions . In any case, characterizing a particular environment in terms of its structure and dynamics is paramount to designing and deploying CPP solutions effectively.

Model-based Environments

Many CPP algorithms use an explicit model of the environment, typically a data structure encoding the positions of obstacles and agents within the AOI [13] [12]. Environment models are often represented by a discrete structure such as a graph or matrix in cases where it's viable to approximate the geometries of features by quantizing their edges to a grid. CPP methods which operate on such environments are often referred to as grid-based or cellular approaches to reflect such quantization. However, in other cases, either

due to preference or necessity, the environment is assumed to have infinite or arbitrary resolution with objects instead being described by vectors or other non-discrete structures [11]. A notable advantage of continuous environment models is that they allow movement in arbitrary directions [**?**], which may be considerably more realistic than the limited set of directions available in discrete settings. We note however that discrete environment models do not always imply discrete movements in a physical AOI. For instance, a graph-based discrete environment model may connect two locations within an AOI without specifying a particular path between them; allowing concrete trajectories to be decided by an auxiliary path finding algorithm (e.g. A-star). Hence, while continuous models are more realistic, discrete models may also be used abstractly to represent a connected locations within continuous environments.

Apart from the topological structure of an AOI, environment models may also approximate aspects of the dynamics within an environment. For instance, if it is known a priori that certain objects within the environment change according to some regular or predictable pattern, this knowledge may be incorporated into the environment model and consequently into CPP algorithms utilizing the model [24]. We refer to such models as dynamic, in contrast to those that simply assume objects remain fixed.

Whether a dynamic or static model is more appropriate is highly application specific. In some cases, while a dynamic model may be preferable, constructing an effective model may be infeasible due to the degree of unpredictability within the environment where agents will be deployed. Environment dynamics may also change over time (distribution shift), causing previously apt models to become obsolete and inaccurate. To address these limitations, model-free environments are often used instead, which we describe in the following section.

Model-free Environments

Rather than attempting to model the environment agents will be situated in, several recent approaches reduce assumptions to a minimum. While remaining ignorant to a priori knowledge about an environment can have a negative effect on performance, it allows such methods to be applied to a far broader set of CPP problems. A prominent example of a model-free environment approach is StiCo [25], which we discuss further in the related work section. StiCo, makes essentially no presuppositions about how the environment will behave, yet under this premise is able to nonetheless achieve effective coverage of an area. Model-free environments are of particular interest to us in this work, as we are focused on providing a generalised CPP framework which cannot commit only to one type of context where CPP may be used.

Formal Definition

To reason formally about CPP it's necessary to make the notion of environments mathematically precise. In general, we can represent the environment as a graph $G = (V, E)$ where the vertices V constitute a possibly infinite set of states. A state may be any mathematical object, for instance a binary matrix representing the locations of obstacles within the AOI. We connect states via edges in E, typically representing the actions an agent can take from a particular state. Readers may recognise this formulation as essentially equivalent to that of a deterministic automaton from formal language theory. While rather abstract, it will be beneficial in our later discussions to understand environments in this way.

We may also extend this definition such that upon every state transition, the environment emits a certain reward R_t, according to some distribution $p(R|s_1 \rightarrow s_2)$ where $s_1 \rightarrow s_2$ denotes the transition from state s_1 to state s_2. In this case, we essentially have a Markov Decision Process (MDP), as found in reinforcement learning [4]. Working under these very general definitions will allow our results to be applied to a wide range of scenarios. Additionally, it creates an opportunity for seemingly disconnected areas of mathematics to be applied to CPP tasks.

2.2.2 Agents

Apart from environments, CPP problems involve agents, individual decision makers which must collaborate to achieve complete coverage of the AOI. Compared to similar multi-agent planning problems (e.g. multi-agent path finding), CPP involves a relatively large number of constraints with regard to which actions agents can take at a particular instant. For example, many approaches based on decomposing the AOI into non-overlapping regions (e.g. DARP) require that agent paths can at no point intersect [17]. In online methods, such strict requirements necessitate extensive and continual communication between agents to ensure synchronization between each agent's internal map of the environment.

Homogeneity and Heterogeneity

Agents may be homogeneous or heterogeneous. Homogeneous agents are often use simpler w.r.t. to their communication protocols, in some cases requiring almost no communication with other agents [25]. In reward-based approaches such as [16], homogeneity also drastically reduces the complexity of the reward function and means only a single reward-maximizing policy needs to be specified or learned, as apposed to separate policies for individual agents [19]. Nonetheless, heterogeneous agents can potentially implement far more sophisticated coverage strategies due to more effective division of resources. For instance in [19], a core team of homogeneous agents are used to explore the environment in conjunction with a secondary team of additional agents to facilitate state synchronization and failure re-planning.

Viewpoints and Sensing

The sensing devices available to individual agents is an important factor in determining the shape and size of viewpoints during CPP. Viewpoints may be considered as subsets of the AOI, with the union of all viewpoints encompassing the entire area. For instance [22] takes in consideration a variety of physical factors affecting reasonable viewpoints in UAV-based CPP; namely, sensor size, focal length and altitude. In model-free environments, viewpoints may not be prespecified but rather generated on the fly as agents explore the area.

Practical Considerations

Many applications of CPP involve physical agents (i.e. mobile robots) exploring a physical environment. In such applications, practical limitations regarding energy consumption and agent movement difficulties become significant factors in determining whether a particular CPP algorithm will be suitable. Some authors take these considerations within CPP algorithms themselves. For example propose Turn Minimizing Spanning Tree Coverage (TMSTC) [20], which uses a reward function that takes into account the number of turns agents must perform to complete their respective paths. While two distinct spanning trees

may generate equal length paths, the one with least turns is often preferable in actuality as turning is often a bottleneck in agent mobility. Moreover, paths with few turns may be more resource-efficient .

2.3 Coverage Path Planning Approaches

CPP algorithms can be classified in several ways and there is no single taxonomy agreed upon by all authors , however, for our purposes we make a primary distinction between offline and online approaches.

2.3.1 Offline Coverage Path Planning

Offline Coverage Path Planning is a traditional form of coverage path planning which generates viewpoints and path plans prior to executing those plans within the environment. Offline CPP approaches have maintained persistent interest in the literature due a number of valuable properties; for instance, their ability to produce near-optimal solutions in static environments [17]. As planning is conducted offline, the generated agent trajectories can be rigorously optimized to achieve short path length and other desirable qualities. Nonetheless, offline approaches have substantial limitations, as we outline below.

A primary disadvantages of offline approaches is their reliance on rather unrealistic assumptions about the environment agents will be embedded in. Specifically, offline algorithms typically presuppose that the environment is static or stationary - i.e. that obstacles and other environmental features will remain constant over time, or more generally that the environment is entirely predictable. However, many scenarios where CPP could be applied, do not necessarily fit this model. Consider for example a household vacuuming robot tasked with cleaning all uncovered areas of a carpet. In such a scenario, we might very well expect environmental obstacles to change over time. A door initially closed, might be opened, leading to novel, unexplored (and uncleaned) regions. Algorithms which assume the environment is static are generally incapable of adapting to such changes without further assistance. A related problem arising from offline approaches is their reliance on complete a priori knowledge about the environment. However, many realistic scenarios may involve environments which are only partially observable or indeed completely unmapped. Hence, offline approaches become quite inconvenient here.

Another limitation of offline approaches is their scalability. Since generated paths must cover all viewpoints, and the number of viewpoints generally scales quadratically with the size of the target area, offline methods inherently involve at least a quadratic number of operations to produce solutions w.r.t. to a given area form factor. While certainly tractable, this time complexity nonetheless becomes unwieldy for highly granular coverage tasks in large areas. Scalability can also be considered in terms of practical deployability. Many offline approaches are also centralized, meaning agents receive plans from a single coordination unit, which may have limited network reachability .

Although offline approaches have significant merits, their inherent limitations have prompted substantial efforts into developing more flexible and robust approaches which relax various conventional assumptions such that they can be applied to a wider range of realistic settings.

2.3.2 Online Coverage Path Planning

Online Coverage Path Planning methods have been developed in conjunction with conventional approaches to address many of the issues outlined above . Their defining feature is the construction of plans at execution time (either partially or entirely), often in a dynamic fashion which adapts to unanticipated environmental changes [15]. Some of the most successful approaches have utilized or replicated concepts from biological systems such as stigmergic communication (i.e. pheromonal signaling) and other decentralized swarm-like mechanics to achieve real-time coverage without an explicit planning algorithm [13]. However, there are still significant areas for improvement within contemporary approaches. For instance, whereas offline planning algorithms have a clear objective (cover every viewpoint in an efficient manner), there has been less consensus among researchers about which objective goals online methods should strive for. Additionally, to the best of our knowledge, there has been little effort to provide a general theory of effective CPP which unified both offline and online algorithms.

2.3.3 Related Work

As mentioned, many of the most successful CPP solutions have employed bio-inspired algorithms to achieve coverage in a distributed fashion. (Hassan et al., 2020) propose PPCPP (predator-pray coverage path planning), an adaptive approach based on the concepts of foraging and risk-of-predation in predator-pray relationships [15]. Whereas pray represent individual agents, predators represent virtual stationary points that agents perpetually maximize their distance from while covering the map and avoiding obstacles. A notable advantage of PPCPP is its ability to cover arbitrary manifolds embedded in R^3, making it suitable for applications involving "bumpy" surfaces.

Drawing inspiration from the collective behavior of ants, (Sahraei et al.) propose StiCo (Stigmergic Coverage) to achieve blind coverage in the presence of dynamic obstacles. In this approach, agents deposit pheromone along the edges of their individual coverage areas while avoiding pheromone they did not deposit, causing the swarm to rapidly disperse and explore the whole map. This heuristic also leads to high coverage retention (CR) — the ability to quickly retain complete coverage of the map after environmental changes, which is often a primary goal in dynamic coverage path planning. Remarkably, StiCo achieves this despite requiring no direct communication among agents (relying only on stigmergy), and does so more effectively than many non-blind approaches. However, a significant limitation of StiCo is that it does not specify how agents should explore the interior of their coverage areas after circumnavigating the edge. Note that conventional spanning-tree approaches cannot be applied due to the dynamism and blindness of the environment. In this sense, StiCo may not always be an applicable *complete* coverage algorithm for environments with fixed viewpoint sizes. Nevertheless, it highlights the potential for stigmergy and swarm intelligence to form the basis of novel, self-organizing CPP approaches.

Chapter 3

Uniformity

3.1 Introduction

The primary aim of coverage path planning is two efficiently visit every viewpoint within a target area. In static environments, we may consider coverage to be complete once every viewpoint has been visited at least once by an agent. We may then measure the efficiency of the solution with respect to how quickly it completes and in terms of other metrics such as the overlap (redundancy) between generated paths. However, in dynamic environments, it may not be sufficient to visit each viewpoint only once. Rather, agents may be instructed to continually maintain an active presence on each viewpoint such that there does not exist a particular viewpoint with significantly less or significantly more attention than all other viewpoints. Additionally, online CPP methods should be capable of adapting to environmental changes which create opportunities to explore previously unreachable regions of the AOI.

While on first inspection, these two notions of successful coverage appear distinct, in this chapter, we introduce a generalised coverage path planning metric and objective function, which allows us to view both offline and online CPP approaches through the same lens. The metric in question is referred to as *uniformity*, which aims to capture the notion of spatially and temporally balanced visitation to each viewpoint within the AOI. It quantifies the extent to which agents effectively distributed their shared resources into covering the AOI as rapidly as possible and maintaining said coverage after perturbations to the environment (e.g. the removal of obstacles, or indeed agents themselves).

Devising a general metric has several advantages. Primarily, it allows authors to directly compare their methods irrespective of taxonomic discrepancies such as whether their methods are online, offline, centralized or decentralized. This may aid researchers in building on each other's work more effectively by adopting a shared framework and language through which CPP methods can be evaluated. A second key advantage of unifying CPP into a single objective function is that it allows seemingly unrelated techniques from the broader fields of mathematics and computer science to find novel applications in path planning. The re-framing of CPP as a maximization problem immediately allows us to import well established algorithms such as genetic optimization and simulated annealing to the coverage path planning domain without significant work [18]. Many other established fields are also applicable, for instance sequential decision theory and reinforcement learning [19], which can use uniformity as a utility or reward function to guide agent decision making and implement self-improving policies via techniques such as deep Q-learning. While, in

this work we do not make use of reinforcement learning, we discuss the largely unexplored potential for reinforcement learning learning in online CPP in Chapter 7.

In this chapter, we discuss uniformity in great detail and show that it formally generalises metrics and goals of conventional coverage path planning, in both offline and online settings. We then introduce a number of derivative metrics based on uniformity to extend the scope of the framework. For instance, we derive the notion of *coverage retention*, to quantify how effectively a team of agents recover from environmental changes that reduce the level of uniformity achieved within the environment. We also show how uniformity can take into consideration the relative importance of different viewpoints through uneven weightings during calculations.

3.2 Coverage Path Planning Metrics

In this section, we discuss the objectives of both offline and online Coverage Path Planning (CPP) approaches and the metrics used to measure their performance. In static environments, the goal of CPP is to efficiently visit every viewpoint within a target area at least once, while in dynamic environments, agents must maintain an active presence on each viewpoint. We provide formal definitions of several offline and online metric, which will aid us in later proving the generality of uniformity using a reduction-style argument.

3.2.1 Offline Coverage Metrics

Offline coverage objectives and metrics generally rest on two assumptions:

- **Staticity.** The environment, or at least the set of viewpoints, remains constant and does not undergo any changes over time or throughout the duration of the coverage task. This assumption allows us to unambiguously denote when coverage has completed — namely after each viewpoint has been visited at least once . An offline CPP method is said to be incomplete if it generates plans which do meet this criterion, except in case of obstacles, which we explain below.
- **Observability.** The environment is wholly observable. An offline CPP should at least have knowledge of the locations of every obstacle within each view point. While the detailed geometry of each obstacle may be unknown, offline approaches generally assume the scope of each obstacle (i.e. how many points within the AOI it covers) is known in advance. We then define a viewpoint as inaccessible if every point within the viewpoint is within the scope of an obstacle. Extending the previous criterion, we permit viewpoints to be "skipped" during coverage if they are inaccessible. Some setups extend the scope of obstacles significantly farther than their physical form factor to account for the time required for agents to physically avoid the object.

Under these two assumptions, authors have formulated a number of metrics for offline coverage planning which aim to capture the objectives of different algorithms. We now outline a number of important metrics used within offline coverage path planning, discuss their appropriateness to different scenarios and give their formal definitions.

Coverage Completeness (CC)

Perhaps the most important aspect of any CPP approach is that it does ensure each viewpoint is indeed visited at least once (complete coverage). While exact algorithms such as DARP may always achieve complete coverage, the same cannot be said for many other solution which rely on more heuristics or other approximate strategies. For example, during the preliminary phase of this project, we experimented with implementing a decentralized rendition of DARP using deep learning to approximate the distribution of DARP-allocated sub-areas for a range of map sizes. In this case, the method was only able to achieve complete coverage in 90% of scenarios with many solutions having unvisited viewpoints, particularly at the borders of adjacent agent sub-areas. Coverage completeness was a fundamental metric we used to evaluate the viability of this approach.

Coverage Completeness (CC) is most often quantified as the ratio between visited and unvisited accessible viewpoints, and commonly expressed a percentage — a coverage completeness score of 90% would indicate that for every nine visited viewpoints there is one unvisited viewpoint. To make this precise, let V denote the fixed set of accessible viewpoints (i.e. those not occupied by obstacles). Let $A = \{A_t\}$ denote a coverage plan such that $A_t \subseteq V$ is the subset of viewpoints occupied by agents at time t. For offline coverage, we say that a viewpoint v is covered if and only if $v \in A_t$ for some time t. This may be written as a unary predicate.

$$C(v) \iff \exists t(v \in A_t)$$

Later it will also be helpful to consider a time-constrained variant of this predicate, denoted $C_t(v)$. This is the subset of accessible vertices which have been covered up to and including time t.

$$C_t(v) \iff (\exists t' \leq t)(v \in A_t)$$

Then, we can specify coverage completeness (CC) as the ratio of covered to uncovered viewpoints.

$$\text{CC} = \frac{|C|}{|V|}$$

Weighted Coverage Completeness (WCC)

Quantifying coverage completeness as a ratio between visited and unvisited viewpoints may not always be sufficient to capture nuance regarding the precedence of certain viewpoints, i.e. the notion that some viewpoints are more important than others. Precedence may manifest either temporally (i.e. there is a preference for visiting certain viewpoints first) or spatially (i.e. there is a preference for visiting certain viewpoints more often than

others). Temporal precedence is beyond the scope of this project, however we discuss spatial precedence below.

Spatial precedence is most often incorporated into coverage metrics by prescribing weights to each viewpoint so that some viewpoints contribute more strongly to the overall score. Weighted Coverage Completeness (WCC) is a simple modification of CC which multiplies each viewpoint in C by its respective weight. Formally this is given by

$$\text{WCC} = \frac{\sum_{a \in C} w(a)}{\sum_{v \in V} w(v)}$$

where $w : V \to R$ returns the weight of a particular vertex. CC can be seen as a special case of WCC where $w(x) = 1$ for all $x \in V$.

WCC can be used to capture various types of spatial precedence, such as the preference for visiting viewpoints in densely populated areas more often than those in sparsely populated areas, or the preference for visiting viewpoints with higher informational value etc. Ultimately the choice of how weights are assigned depends on the specific application and the objectives of the coverage problem. For example, in a surveillance application, the weights of viewpoints may be proportional to the probability of detecting an event or an object in the area covered by the viewpoint. In a wildlife monitoring application, the weights of viewpoints may be proportional to the density of animal populations in the area covered by the viewpoint.

Coverage Time (CT)

Coverage Time quantifies the time it takes for an agent or a team of agents to achieve complete coverage. More specifically, it is time until every viewpoint in V has been visited at least once. How time is measured may depend on whether the coverage algorithm uses a continuous or discrete environment model. In discrete cases, we can count time steps or simply the number of actions performed by the agents until complete coverage is achieved. In continuous scenarios, it may be more appropriate to take into account the time required for agents to continuously move between viewpoints. In any case, we can generally formalise coverage time as follows:

$$\text{CT} = \min\{t \mid C_t = V\}$$

In other words, CT is the earliest time t at which the set of covered viewpoints encompasses all viewpoints in V. For offline planning, we typically assume that the algorithm terminates subsequently terminates after reaching such a state.

Coverage Time is an important metric to consider when evaluating the effectiveness of a CPP solution. In many applications, there may be constraints on the time available for coverage, meaning solutions . For example, in a search and rescue scenario, the time available to locate and rescue a missing person may be limited. In a surveillance application, there may be a need to monitor a large area in a short amount of time in order to detect potential threats.

The complexity of the environment, the number of viewpoints to be covered, and the number of agents involved are all factors that can affect coverage time. In general, coverage time is expected to increase as the number of viewpoints or the complexity of the environment increases. The number of agents involved may bare a less trivial relationship to coverage time; adding more agents may speed up coverage (by allowing more viewpoints to be visited simultaneously) or slow it down (by introducing communication and coordination overheads).

Coverage Rate (CR)

Coverage Rate (CR) (also referred to as coverage velocity) is a related metric to coverage time which measures how coverage changes between successive time steps. This Coverage rate typically varies with the number of agents, often in a linear or polynomial fashion such that increasing the number of agents participating results in swifter coverage. While both CR and CT measure how rapidly coverage occurs, CR offers greater insight into how the efficiency of coverage changes throughout the duration of a run. As we discuss later in this chapter, it becomes especially significant when evaluating online approaches in dynamic environments and aids in understanding how coverage efficiency changes in response to moving obstacles or other perturbations.

CR may be defined as the number of new viewpoints occupied per unit time, commonly computed by taking a running average over n previous time steps. Formally, the coverage rate at time t is given by

$$\mathrm{CR}_t = \frac{|C_t| - |C_{t-n}|}{n}$$

Coverage Overlap (CO)

Coverage Overlap (CO) measures the extent to which coverage paths contain redundancies due to overlapping with each other at certain points. To achieve complete coverage in static environments, it is unnecessary for viewpoints to be visited more than once, and as such any overlap which does occurred can be considered an inefficiency . Note that in some applications, some degree of overlap may in fact be desirable as additional redundancy may help the system to remain robust against individual agent failures. However, excessive overlap is generally seen as a negative quality which ought to be minimized by the planning algorithm. CO can be calculated by measuring the proportion of viewpoints in V that have been visited more than once, relative to the total number of viewpoints.

To define coverage overlap, we first consider the set of viewpoints which are visited more than once during a particular plan A. We can express this as an extension of $C(v)$, such that two distinct times t_1 and t_2 must exist where $v \in A_{t_1}$ and $v \in A_{t_2}$. Formally,

$$O(v) \iff \exists t_1 \exists t_2 (t_1 \neq t_2)(v \in A_{t_1} \wedge v \in A_{t_2})$$

If $O(v)$, then v experiences overlap as it is visited more than once.

Now we can formally define the coverage overlap (CO) metric as the proportion of viewpoints which individually experience overlap.

$$\mathrm{CO} = \frac{|O|}{|V|}$$

If no overlap occurs then CO = 0. In contrast, when excessive overlap occurs, we may expect values above 0.5.

CO is particularly relevant in situations where minimizing the total distance travelled by the agents is important, as overlap can lead to extraneous travel. Another reason is that overlap can result in redundant data collection, which can be wasteful in applications where data storage or processing resources are limited. For example, in environmental monitoring or mapping tasks, overlapping coverage paths can result in the collection of duplicate data.

Coverage Path Length (CPL)

Coverage Path Length (CPL) is a simple metric which measures the length of a particular agent's coverage path, frequently in order to gauge how evenly resources are being allocated among agents. For instance, consider a path plan A involving two agents. Ideally, we would like this plan to prescribe equal path lengths to both agents in order to expend the least amount of energy from each agent individually. Moreover, grossly unbalanced path lengths cause an increase to total coverage time as parallelism is effectively reduced.

CPL for an individual agent path can be measured by counting the number of time steps which occur until the agent no longer moves. That is,

$$\mathrm{CPL}_i = \min\{t \mid \left(\forall t' > t\right)\left(A^i_{t'} = A^i_t\right)\}$$

To quantify the balance (BL) between path lengths, many authors compute a ratio between the shortest and longest paths:

$$\mathrm{BL} = \frac{\min_i \mathrm{CPL}_i}{\max_i \mathrm{CPL}_i}$$

If BL = 1, then there is perfect balance, and every path has the same length. Otherwise, if BL < 1, there is some degree of dis-balance between path lengths. Generally values lower than 0.5 (the longest path is twice as long as the shortest), are considered unacceptable.

In summary, offline CPP methods are subject to a number of key metrics which can be unambiguously specified in terms of the viewpoints occupied by agents at different times during a run. However, as discussed previously, offline methods have only have limited applicability dynamic and/or partially observable environments. To address this, many works have developed online CPP algorithms, and a corresponding set of metrics distinct from those used in offline CPP. In the following section, we highlight several online metrics and compare them to their offline counterparts.

3.2.2 Online Coverage Path Planning Metrics

While the basis of successful offline coverage is coverage completeness the same metric cannot trivially be used to evaluate online approaches, as in such scenarios, coverage is typically perpetual. Nonetheless, we can find an online analogue for coverage completeness, and indeed several other offline metrics using the concept of *visiting periods* [5].

Visiting Period (VP)

Previously we denoted the set of viewpoints by V, and assumed it remains static. However, to accommodate the potentially dynamic environments encountered in online CPP, we now use indexical notation so that V_t denotes the set of viewpoints at time t. We also allow the edges between viewpoints to vary and denote this set with E_t. Finally, as given previously, we denote the subset of viewpoints occupied by agents at time t with A_t. Hence, the complete environment state at time t may be specified as the 3-tuple (E_t, V_t, A_t).

With this notation in mind, we then proceed to define the visiting period (VP) of a particular viewpoint, which is the elapsed time since it was last visited by an agent. Formally,

$$\mathrm{VP}_t(v) = t - \max_{t' \leq t} \left\{ t' \mid v \in A_{t'} \right\}$$

If $\mathrm{VP}_t(v) = 0$, then the viewpoint v is currently occupied by an agent. Otherwise, a positive value indicates that some time has passed since it was last visited. Throughout this section, we will use VP as a basis for other online metrics.

Online Coverage Completeness (OCC)

Online Coverage Completeness (OCC) is a straightforward generalisation of coverage completeness to online approaches. It revolves around the idea of a lifespan given to each viewpoint. The lifespan of a viewpoint is the amount of time which may elapse since its last visit before it's considered uncovered. In this context, we re-purpose $C_t(v)$ to mean the following.

$$C_t(v) \iff \mathrm{VP}_t(v) \leq k$$

where k is the lifespan.

Now we define the online coverage completeness (OCC) at time t as the proportion of viewpoints currently covered:

$$\mathrm{OCC}_t = \mathrm{CC}_t = \frac{|C_t|}{|V|}$$

This definition is almost identical to the conventional formulation of coverage completeness, however expresses the coverage predicate $C_t(v)$ in terms of visiting period rather than whether v exists in A_t for some time t. To implement precedence among viewpoints, we may use a weighted variant of OCC, or (more commonly), use a varying lifespan such that more important viewpoints expire more rapidly and hence require more frequent attention. Choosing an infinite lifespan value reduces OCC to CC.

Online Coverage Rate (OCR)

In a similar vein, we can define an online analogue to coverage rate, again using VP to determine which viewpoints are covered at a particular point in time. Online coverage rate (OCR) is given by

$$\mathrm{OCR}_t = \frac{|C_t| - |C_{t-n}|}{n}$$

where C_t is as defined above.

3.2.3 Generalised Coverage Metrics

So far we have discussed a number of key metrics used offline and online coverage path planning. We see that offline metrics can be formalised in terms of the existence of vertices within a particular coverage path plan A. In contrast, online metrics rely on the concept of a visiting for each viewpoint. Nonetheless, the resulting expressions for offline metrics and their online analogues bare considerable resemblance, suggesting there may be a way to generalise further. In the following section, we introduce one such generalised metric known as uniformity

3.3 Uniformity

We now come to one of the primary contributions of this work — a generalised metric which allows offline and online CPP approaches to be viewed through a unified lens. We refer to this metric as uniformity as it essentially quantifies how evenly spread visits to viewpoints are over the AOI. In this section, we provide a precise definition of uniformity and show that maximizing uniformity implies optimizing for the offline and online metrics discussed previously. In addition, we discuss several derivative metrics such as coverage retention which provide further insight into how online methods respond to disruptive environment changes.

3.3.1 Motivation

The core motivation behind uniformity is the observation that *what we mean* by "ideal coverage" is typically that agents are able to visit each viewpoint within the AOI at a uniform and sufficiently frequent rate. For instance, in offline coverage, we require that every viewpoint is visited exactly once throughout the duration of a particular coverage plan A. In this case, the rate is $1/|A|$ and is uniform since every viewpoint is subjected to it. Indeed, the existence of viewpoints which do not receives visits at this rate would entail that the plan is either incomplete (some viewpoints receive too few visits), or contains overlap (some viewpoints receive too many visits).

A similar analysis can be applied to online CPP metrics. Consider Online Coverage Completeness (OCC) which measures, at time t, the proportion of viewpoints that have been visited recently enough to be considered covered. Suppose that, every viewpoint is visited at a uniform rate and than this rate is sufficiently frequent. Then, after some initial exploration overhead, OCC will be perpetually maximized — every viewpoint will be considered covered. We may also use this general notion of uniformity to extend coverage overlap to dynamic settings. Namely, if certain viewpoints are visited significantly more frequently than others, then we may conclude that there exists redundancy within the plan.

We find that this rather intuitive notion of uniform and sufficient visitation generalises a number of specific metrics used for both offline and online CPP evaluation. In the next section we provide a formal definition of uniformity and show that it formally generalises said metrics, allowing us to develop a unified framework for CPP.

3.3.2 Formal Definition

To make uniformity precise, we must translate our intuitions into formal mathematics. We first define a measure for the uniformity of an environment at time t. We then, use this definition to define the uniformity of a coverage path plan, as the cumulative uniformity achieved throughout the duration of the plan.

We define the state of an environment at time t as a 3-tuple $S_t = (V_t, E_t, A_t)$ where:

- V_t is the set of viewpoints at time t.
- E_t is the set of edges (i.e. accessible paths) between viewpoints at time t.
- $A_t \subseteq V_t$ is the subset of viewpoints occupied by agents at time t.

To quantify the uniformity of such an environment, we equip each viewpoint with a pheromone level which decays over time. We may also call this the temperature or energy of a viewpoint. Whenever an agent visits the viewpoint, the pheromone level is replenished and increased by a constant amount. Otherwise, the pheromone level is depleted and reduced by a constant factor (i.e. exponential decay). The rate at which pheromone decays is an important parameter which essentially reflects how many frequently visits must be made to a particular viewpoint. We discuss this in more depth in Chapter 3.

Let $v \in V$ be a viewpoint. Then its pheromone level at time t is given by

$$L_t(v) = \alpha L_{t-1}(v) + k1_{A_t}(v), \quad L_0(v) = 0$$

where:

- α is the decay rate coefficient — for instance $\alpha = 0.98$ causes the pheromone to decay by 2% on every iteration.
- k is the amount of pheromone deposited by agents when visiting the viewpoint, typically $k = 1$.
- $1_{A_i}(v)$ is an indicator function, returning one if $v \in A_i$ or zero otherwise.

Using this recurrence relation, we can obtain a pheromone distribution for the environment at time t. This is a probability distribution encoding the relative pheromone levels of each viewpoint. We may compute it by simply normalizing over all viewpoints in V:

$$p_t(v) = \frac{L_t(v)}{\sum_{u \in V} L_t(u)}$$

Now, we would like to quantify the uniformity of this distribution. A semantically reasonable choice is to compute the Shannon entropy of a hypothetical random variable distributed according to p_t. That is,

$$\mathrm{H}[p_t] = \mathrm{H}[X], \quad X \sim p_t$$

According to information theory, $\mathrm{H}[X]$ may be interpreted as minimum number of bits required to communicate a particular outcome in X. However, for our purposes we needn't make use of this interpretation. Instead, we can simply see $\mathrm{H}[X]$ as measuring how evenly spread pheromones are among the set of viewpoints. If most of the pheromone is concentrated in only a few viewpoints, then the entropy will be low. As we will show, this occurs precisely when conventional CPP objectives such as low overlap fail to be achieved.

Hence, the uniformity of the environment at time t, may be concisely written as

$$U(S_t) = \mathrm{H}[p_t] = -\sum_{v \in V_t} p_t(v) \log_2 p_t(v)$$

For completeness, we also define the uniformity of an entire coverage path plan A, as the cumulative uniformity achieved at each time step.

$$U(A) = \sum_{t \leq |A|} U(S_t)$$

3.4 Generality

Throughout this chapter we have made the claim that uniformity generalises both offline and online coverage metrics. In section 3.1., we provided an intuitive explanation of why this is the case, however did not provide a formal proof. To support the variety of our claim, we now devise such a proof using the formal definitions of uniformity and conventional CPP metrics. We show that in each case, maximizing uniformity entails optimizing for the metric in question.

3.4.1 Uniformity and Offline Coverage Completeness

Increasing Uniformity Eventually Implies Increasing Coverage Completeness

We show that if $U(S_{t'}) - U(S_t) > c$ then $\mathrm{CC}_{t'} > \mathrm{CC}_t$.

In other words, uniformity cannot be increased beyond a certain amount without also increasing coverage completeness.

Suppose that at time t, we have $|C_t| = k$. Then only k viewpoints have ever been visited.

A viewpoint u which has never been visited has no pheromone since $L_0(u) = 0$ and there exists no time $t' \leq t$ such that $1_{A_{t'}}(v) = 1$.

Hence, all pheromone must be entirely concentrated on the k visited viewpoints. The minimal uniformity occurs, when all pheromone is concentrated on a single viewpoint u, such that

$$p_t(v) = \begin{cases} 1 & \text{if } v = u \\ 0 & \text{otherwise} \end{cases}$$

Then entropy of this distribution is exactly zero; there is no uncertainty about where pheromone is located.

Now suppose that at time $t' > t$, we again have $|C_{t'}| \leq k$. The maximal uniformity occurs when $p_t(v) = 1/k$ for all $v \in V_t$. In this case the entropy is exactly $\log_2 k$.

Hence, the maximum increase in uniformity between t and t' is

$$\max U(S_{t'}) - \min U(S_t) = \log_2 k - 0 = \log_2 k$$

Hence, if $U(S_{t'}) - U(S_t) > \log_2 k$, the assumption that $|C_{t'}| \leq k$ is false, and $|C_{t'}| > |C_t|$.

Therefore, $U(S_{t'}) - U(S_t) > c$ implies $\mathrm{CC}_{t'} > \mathrm{CC}_t$ where $c = \log_2 \mathrm{CC}_t$.

Maximizing Uniformity Implies Maximizing Coverage Completeness

We show that if uniformity is maximal, coverage completeness must be maximal.

Assume that $U(S_t)$ is maximal. Then $p_t(v) = 1/|V_t| > 0$ for every viewpoint. Then for every viewpoint

$$L_t(v) = \alpha L_{t-1}(v) + k1_{A_t}(v) > 0$$

which implies $1_{A_{t'}}(v)$ for some $t' \leq t$.

which implies $C_t(v)$ for every $v \in V_t$.

which implies CC = 100% (maximal).

3.4.2 Uniformity and Online Coverage Completeness

Maximizing Uniformity Implies Maximizing Online Coverage Completeness

We show that if uniformity is maximal, online coverage completeness must be maximal.

Let l be the lifespan such that if $C_t(v) \iff \text{VP}_t(v) < l$.

We can always choose a decay rate coefficient α such that $p_t(v) = 0$ if and only if $\text{VP}_t(v) \geq l$. In other words, we can choose a decay rate which assigns probabilities greater than 0 to viewpoints only if they were visited within the lifespan. Essentially we are encoding l in α.

Maximum uniformity is achieved when $p_t(v) = 1/|V_t|$ for all $v \in V_t$.

Then for all v we have that $p_t(v) > 0$ which implies $\text{VP}_t(v) < l \implies C_t(v)$.

Hence, when uniformity is maximized under α, every viewpoint is covered under the lifespan l, so online coverage completeness is also maximized.

3.5 Derivative Metrics

Now that we have justified the notion of uniformity on intuitive and formal grounds, we introduce several derivative metrics which may expressed in terms of uniformity.

3.5.1 Perplexity (PPL)

Uniformity is measured logarithmically in bits (or nats if we choose to use the natural logarithm). However, it is more natural for us to work with linear scales when comparing and evaluating coverage algorithms. The most straightforward method for linearising entropy is to work with perplexity, which simply inverts the logarithm.

$$\mathrm{PPL}(S_t) = 2^{U(S_t)}$$

Throughout the remainder of this report, we will generally work with PPL rather than U directly when presenting results.

3.5.2 Relative Perplexity (RPPL)

A further transformation we may apply to U, is to compute perplexity, relative to the maximum possible perplexity achievable within an environment. Maximum perplexity is achieved when $p_t(v) = 1/|V_t|$ for all $v \in V_t$ and has a value of $2^{|V_t|}$. Hence, relative perplexity (RP) is given by

$$\mathrm{RPPL}(S_t) = \frac{2^{U(S_t)}}{2^{|V_t|}} = 2^{U(S_t)-|V_t|}$$

When RPPL $= 1$, we have ideal coverage. Values less than 50% are generally considered unsatisfactory. Note that the RPPL achieved by an algorithm is dependant on the decay rate coefficient α. Higher values of α (lower decay rate) make achieving high RPPL easier, whereas lower values represent resource-intensive coverage problems where each viewpoint requires more frequent attention.

3.5.3 Online Coverage Rate (OCR)

We previously defined online coverage rate in terms of visiting periods, however we can also define it more flexibly using uniformity. Specifically, we can consider the change in uniformity over the past n time steps.

$$\mathrm{OCR}_t = \frac{U(S_t) - U(S_{t-n})}{n}$$

A CPP which achieves a high uniformity and does so rapidly is considered efficient.

3.5.4 Coverage Retention (CR)

One important aspect of online CPP methods is their ability to adapt to changing environments. In such settings, we're often interested how online coverage completeness and other time-varying metrics are affected as perturbations occur. Coverage retention (CR) allows us to measure precisely how such changes influence the uniformity achieved by a particular CPP algorithm. It is defined as the OCR achieved after an environmental change. The

nature of the change to the environment is circumstantial, however two primary changes which we focus on in later sections is the insertion and removal of obstacles. Effective online CPP should typically maintain their current coverage rate when obstacles are inserted and increase their coverage rate when obstacles are removed.

3.6 Summary

In this chapter we presented a general framework for offline and online CPP which allows both approaches to be viewed as maximizing the entropy of pheromone deposited within the environment. We showed that the uniformity objective formally implies optimizing several conventional metrics, while also creating opportunities for the development of new measures such as coverage retention. In the next chapter, we present a novel approach to online CPP which explicitly aims to maximize uniformity through greedy actions taken by a swarm of homogeneous agents.

Chapter 4

GEM: Greedy Entropy Maximization

4.1 Introduction

In Chapter 3 we introduced the notion of uniformity a general measure of coverage performance that can be applied to both offline and online CPP scenarios. In this chapter, we present a concrete uniformity-based coverage path planning algorithm which takes inspiration from the collective behaviour of ants. We provide an extensive evaluation of our solution and compare it to existing state-of-the-art approaches.

Our approach is referred to as GEM (Greedy Entropy Maximization) and aims to maximize uniformity (entropy) using a swarm of homogeneous agents that make decisions greedily (i.e. only using local information). GEM makes almost no a priori assumptions about the environment (apart from the assumption that it's discrete). GEM can also work with an arbitrary number of agents and yields a time complexity which is invariant to map size. In addition, agents require no direct communication as the algorithm instead relies on implicit/indirect stigmergic communication via pheromones.

4.2 Problem Formulation

The primary goal of GEM is to maximize the uniformity of the environment $U(S_t)$, and more generally the cumulative uniformity obtained over the entire course of a run, denoted $U(A)$. Readers may recognise a similarity between this objective and the objective reinforcement learning (RL) algorithms. Indeed, to make that connection precise, we now formulate online CPP as a Markov-decision process (MDP), to support later discussions regarding the application of RL to CPP.

Let $\{(S_0, R_0), \dots, (S_t, R_t), \dots\}$ be a Markov-decision process where $S_t = (V_t, E_t, A_t)$ is the state of the environment at time t and R_t is the immediate reward obtained from transitioning into state S_t from S_{t-1}. The agent (which in our case refers to the entire swarm of mobile agents), emits a corresponding sequence of actions $\{a_0, \dots, a_t, \dots\}$ which cause the environment to emit a new state and reward according to the conditional distribution $p\left(R_{t+1}, S_{t+1} | S_t, a_t\right)$.

4.2.1 Actions

The swarm emits a sequence of actions $\{a_0, \ldots, a_t, \ldots\}$ describing the movements of each agent at every time step. For simplicity, we can let $a_t = A_t$, such that each action just lists the viewpoints each agent should occupy. Some actions are invalid in particular states; namely actions which would cause multiple agents to occupy the same viewpoint at the same time, or actions which place agents in viewpoints occupied by obstacles. For the sake of generality, we avoid defining precisely how obstacles are represented, however define an inaccessible viewpoint as one which has no inbound edges. We say $v \in V_t$ is inaccessible if

$$(\forall u \in V_t)((u, v) \notin E_t)$$

In contrast, an accessible viewpoint is one with at least one inbound edge:

$$(\exists u \in V_t)((u, v) \in E_t)$$

In GEM, agents may only move between adjacent viewpoint such that

$$(\forall v \in A_t)(\exists u \in O_{t-1})((u, v) \in E_{t-1})$$

i.e., every viewpoint in A_t must be reachable from a viewpoint in A_{t-1} (the prior locations of each agent), along an edge in E_{t-1}.

4.2.2 States

The state of the environment is essentially a graph with vertices representing viewpoints and edges representing accessibility between those viewpoints. For instance, in grid-based CPP, each grid cell may be assigned a particular viewpoint, with edges between call geometrically adjacent cells and itself (reflexivity).

We generally assume that V_t remains fixed such that $\forall t(V_t = V_0)$. In most environment models, there is no need to change V_t as all environmental perturbations can be represented through alterations to the set of edges E_t. For instance, again considering grid-based CPP, walls may be represented simply by the lack of an edge between two adjacent cells.

4.2.3 Rewards

Rewards are signals transmitted to an agent in addition to states, which signal whether a previous action was positive or negative. In the case of GEM, a natural choice for this reward signal is just the uniformity of the current state. That is,

$$R_t = U(S_t)$$

We could also use the change in uniformity between consecutive states; in RL solutions, this may be preferable as it provides a bounded reward signal with negative rewards helping to negatively reinforce actions that reduce coverage. However, for GEM, which is not based on RL, we can uniformity directly.

Then cumulative reward received by the agent is simply given by

$$R = \sum_{t<t_{\max}} R_t$$

The agent's goal is then to select a set of actions A such that

$$R = \max_{A \in \mathcal{A}} R$$

where $\mathcal{A}$ is the set of all possible action sequences the agent could take.

This completes our decision-theoretic formulation of online CPP. We now discuss how GEM provides an effective solution.

4.3 Greedy Entropy Maximization Algorithm

GEM is a completely decentralized online CPP algorithm in which a swarm of homogeneous agents act to maximize the entropy of pheromone deposited within the environment. Each agent is assumed to have minimal sensing such that it can only detect the state of viewpoints immediately adjacent to the viewpoint it's currently stationed on. Moreover, we make no assumptions about whether agents have memory nor whether they are capable of communicating with each other remotely. Such relaxed assumptions might appear to preclude any reasonable method for successful coverage. However, we find that through pheromone-based stigmergic communication, a swarm of such limited agents can nonetheless achieve strong performance on uniformity coverage benchmarks.

In GEM, each agent within the swarm must have access to the pheromone level $L_t(v)$ deposited on every viewpoint adjacent to its own position. A decision heuristic function $h(\cdot)$ is then applied to determine which adjacent viewpoint the agent should move to. Formally,

$$A_t^i = _{v \in \text{adj}(A_{t-1}^i)} \, h(L_t(v))$$

where $\text{adj}(A_{t-1}^i)$ is the set of viewpoints adjacent to A_{t-1}^i.

The simplest heuristic, and the one we use our implementation is just the identity function $h(x) = x$. In this case, each agent simply moves to whichever viewpoint has the least pheromone deposited on it.

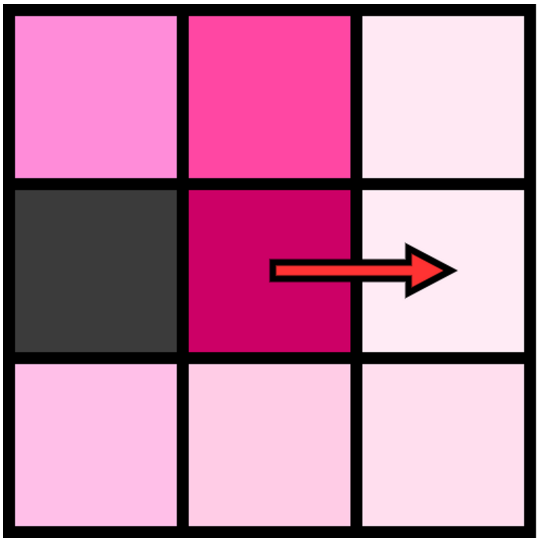

FIGURE 4.1: Actions taken by a GEM agent following the least pheromone heuristic with pheromone levels depicted by colour intensity.

$$A_t^i = \text{argmin}_{v \in \text{adj}(A_{t-1}^i)} L_t(v)$$

Figure 4.1 shows a hypothetical scenario in a grid-based CPP problem with an agent stationed at the centre cell. The intensity of each cell indicates the relative amount of pheromone deposited on it. Under the identity heuristic, the agent chooses to move to the adjacent cell with least pheromone, indicated by the arrow. An obstacle is shown on the left in grey.

Once an agent moves to a new viewpoint, it deposits fresh pheromone to it, which increases the pheromone level. Pheromone levels decay over time according to the equation introduced in chapter 3.

$$L_t(v) = \alpha L_{t-1}(v) + k1_{A_t}(v), \quad L_0(v) = 0$$

where α is the decay rate coefficient and k is the amount of new pheromone deposited.

Intuitively, we might expect this heuristic to perform moderately since it actively peruses viewpoints that haven't been visited recently. However, quite surprisingly, we find it performs exceptionally competing with offline CPP algorithms such as DARP.

4.4 Pseudo-code Implementation

The GEM algorithm is very simple and as such requires very little code to implement. We provide pseudo-code for the algorithm below.

for $t \in \{0, \ldots, t_{max}\}$ **do**
  for $A_t^i \in A_t$ **do**
    $A_{t+1}^i = \text{argmin}_{v \in \text{adj}(A_{t-1}^i)} L_t(v)$
  end
  for $v \in V_t$ **do**
    $L_{t+1}(v) = \alpha L_t(v) + k 1_{A_t}(v)$
  end
end

Algorithm 1: GEM

At every time-step, each agent moves to whichever viewpoint has the least pheromone deposited on it. Every viewpoint's pheromone level is then updated according to exponential decay and whether an agent is current stationed on it.

4.5 Complexity Analysis

A primary advantage of GEM over several existing methods is that it's computational complexity is scales linearly or constantly with respect to important parameters such as the number of agents, or number of viewpoints within the environment.

As discussed, agents are entirely independent of each other and do not rely on each other to make decisions. As such, we first consider the time and space complexity of GEM, when running as a concurrent algorithm, i.e. where agents run in parallel. In this case, at every time step, each agent simply needs to observe the pheromone present on every viewpoint adjacent to its own. This operation is $O(n)$ in the number of adjacent viewpoints. In many cases, such as in grid-based CPP problems, this factor is constant, so the operation takes $O(1)$ time. Identifying the viewpoint with least pheromone is consequently also $O(n)$ or $O(1)$.

Similarly, pheromone updates can be performed in a concurrent fashion as $L_t(v)$ depends only on $L_{t-1}(v)$. In settings, where pheromones have a physical implementation, this operation make take $O(1)$ time. In other environments, such as simulations, the operation nonetheless be performed efficiently using various concurrency techniques, such as GPU programming.

Overall, in physical environments, the algorithm can have complexities as low $O(1)$ in both the number of agents, and the number of viewpoints. In simulated environments, the algorithm's time complexity is $O(n)$ in the number of agents and $O(n)$ in the number of viewpoints. However, using concurrency techniques, the concrete time required to update pheromone levels for each viewpoint may be reduced considerably.

4.6 Experiments

The paths generated by GEM might be classified as chaotic (in the chaos-theoretic sense), since they are highly sensitive to initial conditions, such as the number of agents participating in coverage and the precise location of obstacles. Such a high degree of sensitivity makes formally analysing GEM's performance difficult, if not intractable. To overcome this issue, we instead rely on a number of empirical experiments which demonstrate how GEM performs in a wide variety of environments. We make use of statistical techniques such as p-values to estimate the reliability of our results.

We evaluate GEM on environments including static and dynamic features, and compare its performance against existing state-of-the-art algorithms. Our results show that GEM consistently competes with or outperforms other approaches in terms of uniformity and other uniformity-based metrics. Furthermore, we demonstrate that GEM is robust to changes in the number of agents and can handle complex environments with both narrow passages and multiple rooms. Overall, our experiments demonstrate the effectiveness of using stigmergy-based approaches for coverage path planning, and the potential of GEM as a practical solution for real-world applications.

4.6.1 Experimental Design

Our experiments aim to ascertain how well GEM performs under a range of environmental conditions, including static and dynamic environments. In each experiment, we compare GEM against two other algorithms: a control and a target The control algorithm serves as an insufficient baseline, essentially an algorithm which we deem as ineffective in performing CPP. In our case, we use a random walker which moves agents in random directions at every time step, essentially implementing Brownian motion. If GEM performs similarly (or worse) than this control algorithm, we can also deem GEM as ineffective. In contrast, the target algorithm serves the opposite purpose, essentially providing model solutions. In our case, we use a variant of DARP known as Replannable DARP (R-DARP). R-DARP essentially recomputes the DARP-optimal solution whenever the environment changes. R-DARP achieves very strong performance in dynamic environments, however is often infeasible to implement in practice — primarily because it requires omniscience; i.e. complete knowledge of the environment state at all times. If GEM can compete with or surpass R-DARP, then we may deem GEM as highly effective. Apart from R-DARP, we also consider the *theoretically optimal* CPP algorithm for any environment, which simply stations an agent at every accessible viewpoint. This algorithm is similarly intractable however serves to provide an absolute upper-bound on performance. Where GEM lies between these three alternatives will inform our overall evaluation of its performance and utility.

In our experiments, we simulate a wide variety of environment types, broadly falling into three categories:

- **Random Obstacle Environments.** static and dynamic environments where each viewpoint is occupied with probability p. In dynamic cases, the environment is regenerated every so often (e.g. every 10 time-steps). In static cases, it's generated only once. This category of environment allows us to gauge how well GEM performs

in arbitrary AOIs, however may not accurately reflect its performance in many real-world settings. This is addressed by the next two environment categories.

- **Maze-like Environments.** Some real-world environments may take the form of sparse or dense maze with many interlocking passages. To estimate GEM's performance in such settings, we use a conventional maze generation algorithm (namely, recursive backtracking) to generate random mazes of varying sizes. We then measure how effectively GEM can cover all accessible viewpoints within the AOI. Maze-like environments are typically the most challenging form of static environment for CPP algorithms as they foster many opportunities for collisions between agents. As GEM does not perform planning in advance, our expectation prior to conducting experiments is that it will perform significantly worse than the target in these settings.

- **Room-like Environments.** A more commonly encountered setting is that consisting of several rooms separated by walls, with doors that occasionally open. To generate these environments, we randomly generate nonuniform grids, by intersecting a number of horizontal and vertical lines. We then randomly select locations along each line to use as doors which open with probability p.

4.6.2 Random Obstacle Environments

Qualitative Analysis

Random obstacle environments are the simplest form of environment commonly used by authors to evaluate CPP algorithms. In such environments, the accessibility of each viewpoint or has a Bernoulli distribution, typically uniform across the whole AOI. The probability p that a viewpoint is occupied may be referred to as the obstacle density of the environment. In our experiments, we investigate how well GEM performs in environments with densities between 0.1 and 0.9. Note that when measuring relative perplexity (RPPL), the presence of more obstacles is automatically taken into account. RPPL is measured in terms of accessible viewpoints only. An example of a random obstacle environment generated with obstacle density $p = 0.3$ is shown in figure 4.2.

The heat-map shows the state of a 40x40 AOI after 1000 iterations of GEM with four agents and a pheromone decay coefficient of $\alpha = 0.99$. The hue of each cell designates the amount of pheromone deposited on the corresponding viewpoint. As can be seen, the distribution is highly uniform with a RPPL score of 1537/1563 (98%).

A smaller 20x20 random obstacle environment with $p = 0.98$ is shown below figures 4.3 and 4.4 after 3000 iterations of R-DARP and GEM respectively.

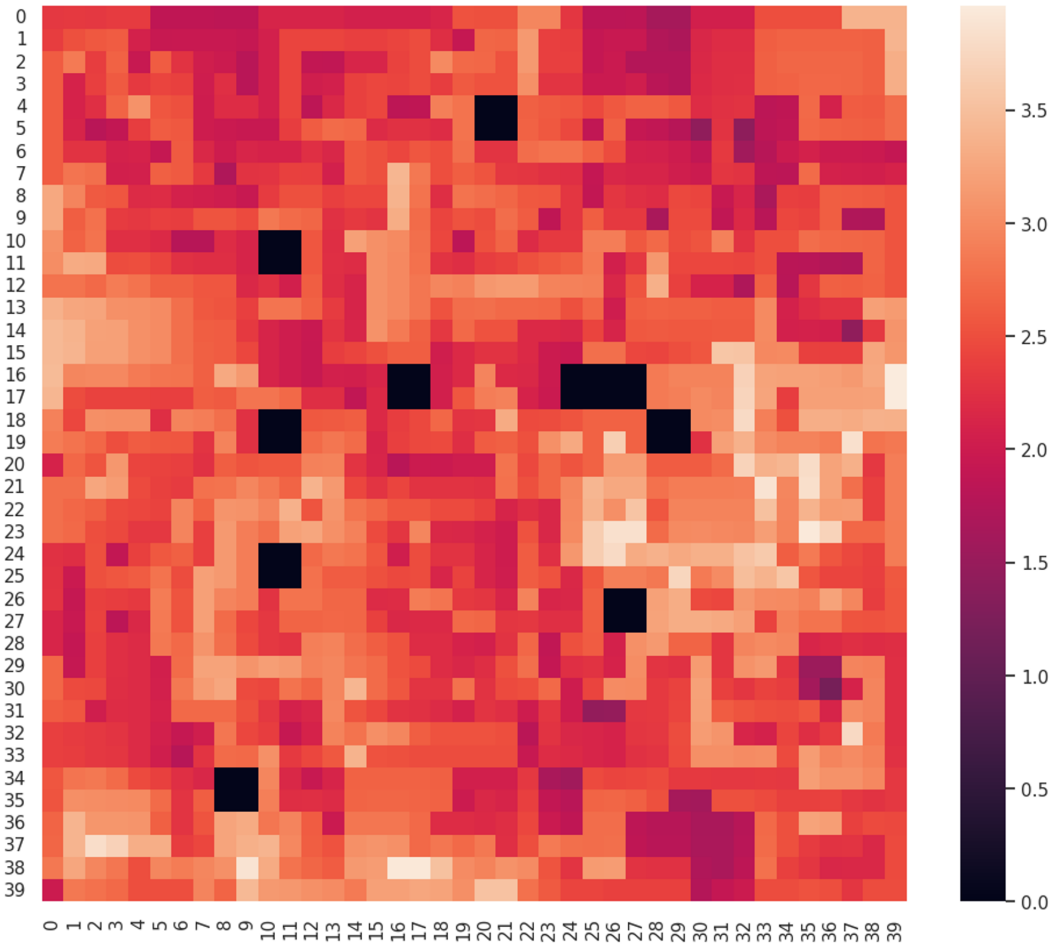

FIGURE 4.2: A heat-map showing the state of a 40x40 AOI after 1000 iterations of GEM with four agents, a pheromone decay coefficient of $\alpha = 0.99$ (i.e. $\beta = 0.01$), and an obstacle density of $p = 0.3$

While each algorithm produces significantly different paths, we find that they both achieve strong RPPL scores of 91% and 99% respectively. R-DARP produces pheromone trails which are highly uniform between contiguous viewpoints. This can be attributed to the fact that each viewpoint is only ever visited by a single agent (in static environments). However, this property also leads to some degree of self-overlap, which results in non-ideal uniformity. In contrast, GEM produces pheromone which is less uniform between contiguous viewpoints, however more evenly distributed overall, leading to a superior RPPL score of 99%. The qualitative difference between these two distributions can be examined further by plotting the PPL score achieved for each iteration between $t = 0$ and $t = 3000$. These plots are shown below:

For comparison, we also render the same plots for our control algorithm (random walker), and the theoretically optimal CPP algorithm.

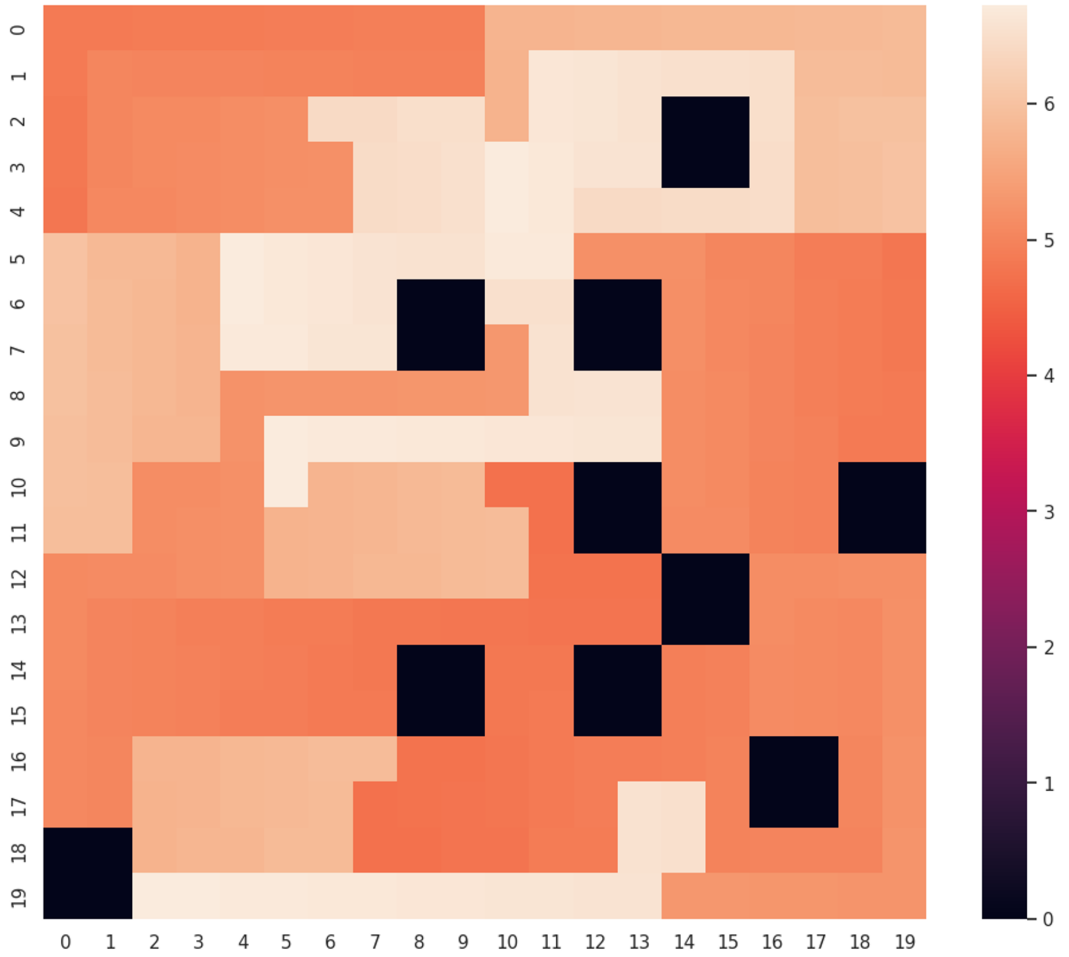

FIGURE 4.3: R-DARP pheromone distribution after 3000 iterations. RPPL = 331/360 = 91%

There are several important observations to make regarding these plots. Firstly, we see that R-DARP has a higher coverage rate (CR) than GEM, between $t = 0$ and $t = 250$, with R-DARP achieving a PPL above 300, before GEM. However, GEM ultimately converges to a higher, near-optimal PPL of 358 within 500 iterations, whereas R-DARP only obtains a maximum of 331. We also observe that R-DARP features periodic drops in PPL, corresponding to periods when all agents are experiencing some degree of self-overlap simultaneously. R-DARP's mean PPL score of 306 is hence significantly lower than GEM's at 343.

Nonetheless, GEM's initial coverage rate is substantially less than the theoretical optimum, suggesting there is still room for significant improvement in this regard.

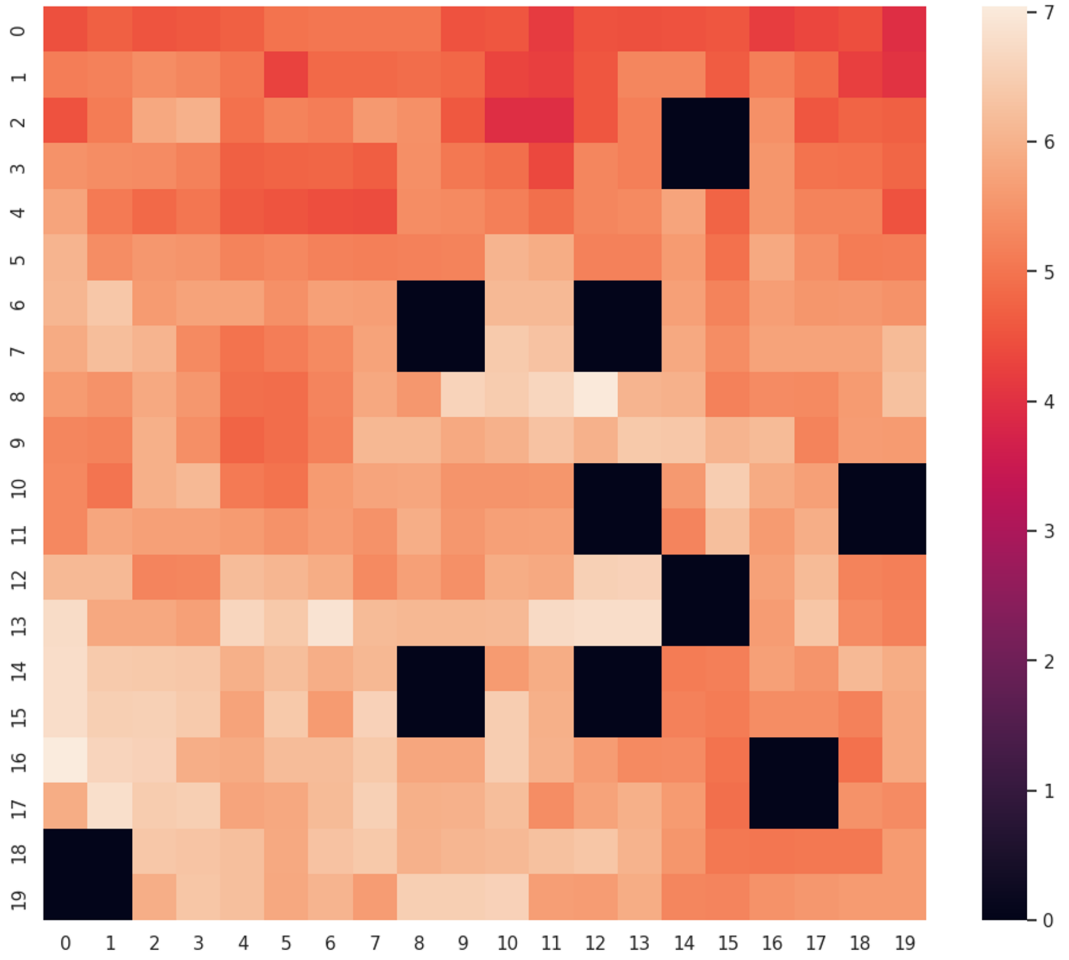

FIGURE 4.4: GEM pheromone distribution after 3000 iterations. RPPL = 358/360 = 99%

Quantitative Analysis

In the previous section we provided a number of qualitative examples demonstrating how GEM performs in comparison to R-DARP. However, in order to obtain confidence in GEM, we require a rigorous quantitative analysis. In this analysis, we are primarily interested in how uniformity-based metrics (PPL/RPPL, CR) vary with respect to CPP parameters including:

- n — the number of agents used to conduct CPP
- p — the obstacle density of the environment
- β — the pheromone decay rate
- A — the number of viewpoints in the AOI (i.e. its area)

For a particular assignment of these parameters, we measure the following response variables:

- $m = \max R$ — the maximum RPPL achieved within k iterations.

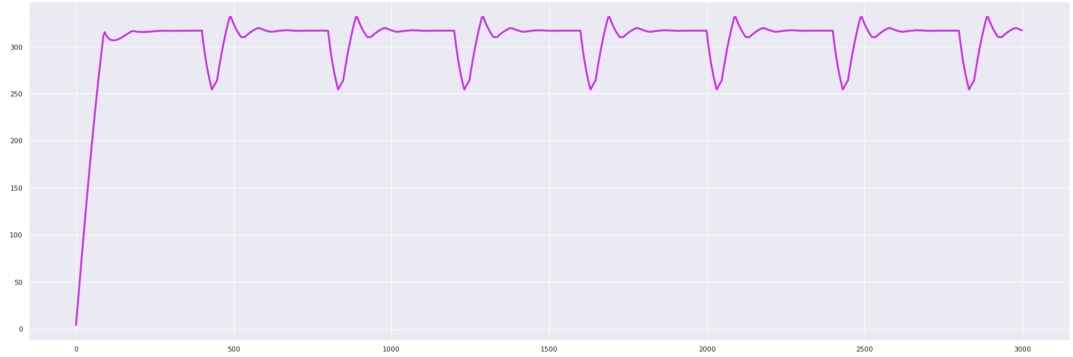

FIGURE 4.5: R-DARP PPL over time; max = 331, $\mu = 306$

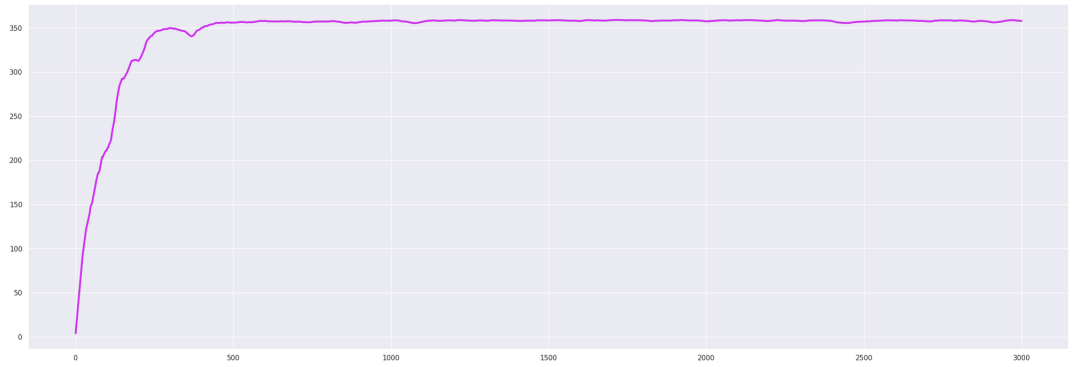

FIGURE 4.6: GEM PPL over time; max = 358, $\mu = 343$

- $\mu = \mathrm{E}[R]$ — the mean RPPL achieved within k iterations.
- $\sigma = \sqrt{\mathrm{V}[R]}$ — the standard deviation of RPPL achieved within k iterations.

Results

We randomly generate 1000 CPP problems on a 20x20 AOI with a fixed obstacle density $p = 0.1$ and constant decay rate of $\beta = 0.01$. We vary the number of agents and observe how this affects the expected RPPL achieved within $k = 1000$ iterations, for GEM, R-DARP and RandomWalker.

n vs. μ with $k = 1000$, $p = 0.01$, $A = 20^2$, $\beta = 0.01$			
Algorithm	$n = 1$	$n = 5$	$n = 10$
GEM	**0.45**	**0.85**	**0.93**
R-DARP	0.41	0.73	0.88
RandomWalker	0.16	0.47	0.55

To investigate the relationship of obstacle density and μ, we generate 1000 random CPP problems on a 20x20 AOI with a fixed number of agents $n = 5$ and a decay rate of $\beta = 0.01$. We vary the obstacle density between 0% and 50%.

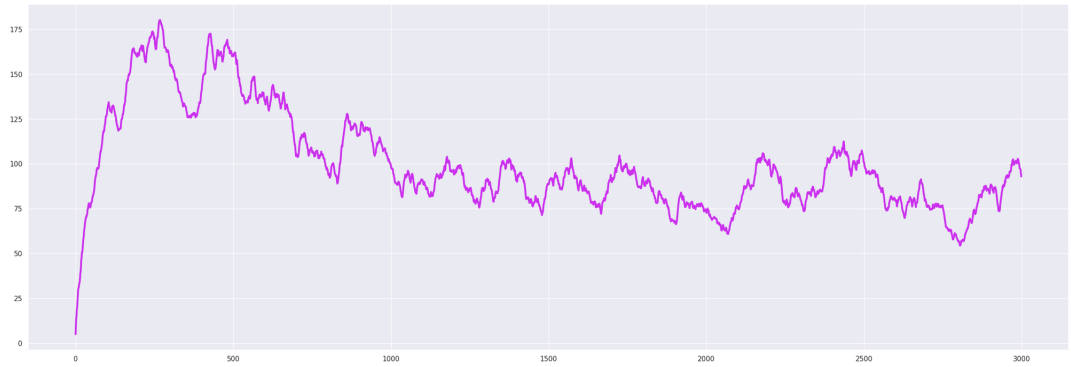

FIGURE 4.7: Random Walker PPL over time; max = 180, $\mu = 99$

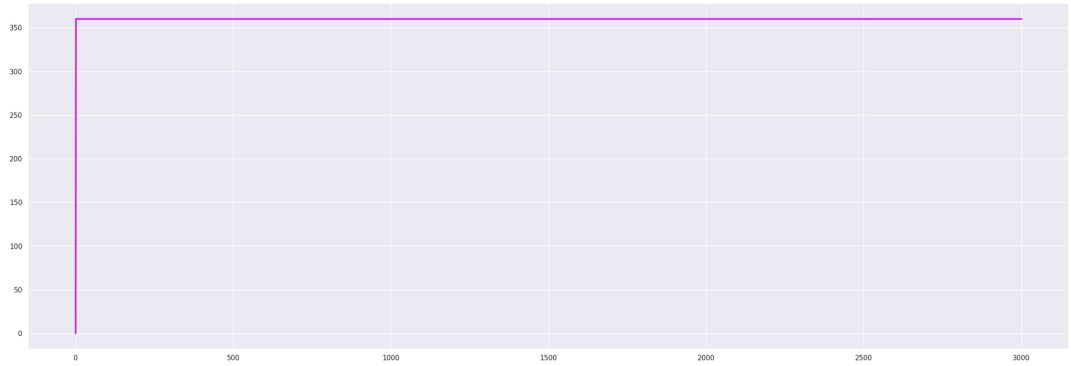

FIGURE 4.8: Optimal PPL over time; max = 360, $\mu = 360$

p vs. μ with $k = 1000$, $n = 5$, $A = 20^2$, $\beta = 0.01$			
Algorithm	$p = 0.0$	$p = 0.25$	$p = 0.5$
GEM	**0.94**	0.72	**0.20**
R-DARP	0.88	**0.74**	0.14
RandomWalker	0.16	0.47	0.07

We repeat the same experiment with $A = 40^2$.

p vs. μ with $k = 1000$, $n = 5$, $A = 40^2$, $\beta = 0.01$			
Algorithm	$p = 0.0$	$p = 0.25$	$p = 0.5$
GEM	**0.64**	0.45	**0.23**
R-DARP	0.59	**0.45**	0.17
RandomWalker	0.16	0.47	0.07

Evaluation

Our results show that GEM consistently performs effectively in arbitrary environments, often able to achieve a mean RPPL of above 90% when given a suitable number of agents for the target decay rate. We find that GEM performs competitively with R-DARP in terms of how many agents are required to reach a certain level of coverage. It generally surpasses R-DARP in terms of the stability of coverage throughout the duration of a run,

leading to higher mean RPPL. In future, we would like evaluate GEM more rigorously in a wider range of realistic environment settings; for instance, involving *adversarial* agents.

Chapter 5

Design and Implementation

5.1 Introduction

In addition to its primary objective, this project aims to address a significant challenge in the CPP research: the lack of easily accessible, open-source software and tools for developing CPP algorithms and simulations. We hope to alleviate the difficulties encountered by researchers and developers in accessing and integrating CPP algorithms into their own work, due to the absence of properly maintained implementations on package indices for popular languages.

To address these challenges, this project aims to provide an open-source framework and library that streamlines CPP algorithm and simulation development using a modular, extensible API. The hope is that, researchers and developers may incorporate the framework into their own projects such that they can focus on what's important: implementing CPP algorithms, without the overhead of implementing a simulation engine from scratch. Ultimately, we hope our software can serve to accelerate progress in the field of CPP by providing a welcoming ecosystem which can be used by beginners and experts alike.

5.2 Motivations

During the preliminary research stage of this project, it became apparent that the availability of resources related to CPP algorithms and software tools was rather limited. Despite extensive online searches, we were unable to locate any published libraries or frameworks on popular package indices for languages such as Python and JavaScript that were specifically designed to support CPP algorithm development or research. Subsequently, we expanded our search to other software repositories on websites such as GitHub and CodeBerg. However, it remained exceedingly difficult to identify any resources targeted at general CPP research and development. Our findings are corroborated by [21] which discusses in more detail the sparsity issue. We were able to locate a very limited number of Python-based implementations for existing CPP methods, however these were generally poorly maintained, outdated and severely lacking in documentation. In some cases, these implementations were bug-prone and difficult to run reliably.

Additionally, we observed that paper authors often failed to provide easily accessible implementations of their algorithms, or access to benchmarking datasets, making objective

comparison between methods a painstaking process, sometimes involving a degree of reverse engineering. The challenges posed by these factors were a considerable hindrance towards the development of our project, and significantly reduced the rate at which we would implement GEM and compare it against other algorithms. Therefore, it became essential to address these issues by developing a standardized open-source framework and library for CPP algorithm development and presentation.

To prevent the challenges we encountered during the preliminary research stage from arising in the future, we devised a strategy to ensure that any code we produce is readily integratable into other projects. The most direct way of achieving this was to divide development into two phases — framework development, and implementing GEM. The framework would be entirely decoupled from our specific GEM implementation, and possible to use within any other project targeting CPP. This approach was useful, both in terms of making our code more maintainable and modular, as well as providing a large proportion of code we could export and publish as an open-source package.

5.3 Requirements

Before commencing the development of any software project, it is essential to conduct a thorough and systematic approach to requirements gathering. This process serves as a foundation for the project, enabling the identification of user needs, system constraints, and functional requirements, and thereby ensuring that the end product fulfills its intended purpose. Effective requirements gathering is a critical aspect of software development, and can significantly reduce the risk of costly errors and delays during the development cycle.

In line with this, our project dedicated at least three weeks to the requirements gathering phase, employing a multi-faceted approach. We employed various techniques for gathering requirements, including the identification of open research areas within the field of CPP to inform high-level requirements. We also conducted user interviews and developed user personas to gain insight into the needs and expectations of potential users of our software. Our supervisor additionally provided valuable feedback and guidance throughout the requirements gathering process, which helped us to refine our understanding of the project's scope and objectives.

5.3.1 User-centred Requirements

The user-centered requirements we gathered reflect the needs and expectations of our target users, which include researchers and developers working in the field of CPP, as well as non-experts who would simply like to integrate existing CPP algorithms into their work. To gather user-centered requirements, we utilized three primary mediums for engaging with users. Firstly, we created a basic questionnaire consisting of open-ended and closed questions regarding potential features for our CPP framework. We also conducted in-person interviews with a small number of users to gain a more qualitative understanding of user needs. Finally, we constructed several personas that characterize our typical users in terms of their background, goals and motivations. As a result of these procedures, we were able to identify the following requirements for our framework:

Non-functional Requirements

- **Modularity.** The framework should be designed using a modular architecture so that new CPP algorithms can be implemented and/or used without requiring fundamental structural changes to the code base.
- **Ease of Use.** The framework should be easy to install, configure, and use. Unlike the existing implementations we found, this framework should not require an extensive, laborious list of shell commands to install. Importing a particular CPP method from the framework should also be a simple, streamlined process which involves only a few lines of code.
- **Generality.** The framework should facilitate a wide range of coverage path planning approaches so that users are not constrained in which types of algorithms they can use or implement. For instance, both offline and online CPP algorithms should be implementable within the framework.
- **Performance.** The framework should place an emphasis on efficiency regarding simulation tools and basic constructs such as updating the state of an environment. This is necessary to provide a pleasant user-experience and allows users with limited hardware to nonetheless access the framework.

Functional Requirements

- The framework should allow users to implement and use arbitrary environments, agents and evaluation metrics; i.e. none of these aspects should be "hard-coded."
- The framework should provide a way to "mix and match" agents with different environments. In other words, there should be essentially no coupling between agents and environments. In general there should also be very little coupling between any two components in the system.
- The framework should support interactivity whereby the user can influence the environment in real-time for online CPP methods. This is a key feature for experimentation and presentation of methods.
- The framework should provide an easy way to export or import maps and environment configurations in order to promote the development of open-source CPP datasets.

5.3.2 Additional Requirements

In addition to user-centered requirements, we identified a range of additional requirements that are essential for the successful implementation and deployment of our project.

Non-functional requirements

- **Scalability.** The framework should be designed in a way that allows for easy scaling to accommodate larger and more complex CPP scenarios, without compromising performance or stability.
- **Portability.** The framework should be platform-independent, and capable of running on a variety of operating systems, including Windows, macOS, and Linux.
- **Maintainability.** The framework should be well-documented and emphasise high-quality readable code, allowing for easy maintenance and future development by multiple contributors.
- **Security.** The framework should be designed with security-first and defensive programming principles. While the framework is unlikely to be directly used in high-security contexts, it may feature indirectly within a larger application.
- **Verifiability.** It should be easy to test and verify the correctness of components within the system. The lack of tests accompanying open-source implementations of CPP algorithms in an area for improvement.

Functional requirements

- The framework should provide visualization tools that allow users to view and analyze the results of their CPP simulations.
- The framework should support multiple types of maps and environments.This may be achieved by adopting a generalised graph data structure for representing environments.
- The framework should support both single-agent and multi-agent CPP scenarios.
- The framework should provide a comprehensive set of tools for evaluating and comparing different CPP algorithms, including metrics for coverage, and tools for testing/verifying the correctness of a particular solution.

5.3.3 Use-cases

After gathering requirements from users, we were able to identify several specific use-cases of the software, which we discuss below.

- **Run Offline CPP Algorithms.** This use-case involves a user selecting and running an offline CPP algorithm from the framework, using their own environment and agent configurations as inputs. The output of the algorithm, including the path plan, is visualized in the framework's interface.
- **Run Online CPP Algorithms.** This use-case is similar to the previous one, but the algorithm is designed to operate in real-time, allowing the user to interact with the environment and agent during the simulation. This use-case is particularly useful for testing and experimentation with online CPP algorithms.

- **Compare CPP Algorithms.** This use-case involves a user comparing the performance of multiple CPP algorithms, either through visual inspection or quantitative analysis.
- **Evaluate CPP Algorithm.** This use-case involves a user evaluating the performance of a CPP algorithm under various conditions and settings. The framework should provide tools for generating statistics for a particular algorithm, allowing the user to gain insights into its strengths and weaknesses.
- **Debug/Improve CPP Algorithm.** This use-case involves a user debugging and optimizing a CPP algorithm.
- **Integrate CPP Algorithm.** This use-case involves a user integrating a CPP algorithm into a larger application. The framework should provide effective means by which CPP algorithms can be deployed in production.
- **Develop New CPP Algorithms.** This use-case involves a user developing a new CPP algorithm using the framework. The framework should provide a comprehensive set of building blocks and interfaces for developing new algorithms, as well as extensive documentation and examples to guide the user through the process.

Overall, these use-cases cover a wide range of tasks and workflows related to CPP research and development, and demonstrate the versatility and flexibility of our framework.

5.3.4 Requirements Analysis

In this section, we discuss how we used our gathered requirements to derive a first pass for our design.

Modularity

Perhaps the most important non-functional requirement suggested by potential users was for the system to be modular and extensible. It should be easy for users and contributors to compose different aspects of the framework, as well as implement new features without requiring an in-depth understanding of the whole system. This informed our design in several ways. Firstly, it suggested that we should focus heavily on the APIs and points of communication between components in the system. It was clear from the outset that total decoupling would be a necessary to achieve a desirable degree of modularity.

Performance

Another important requirement was for the system to be performant, particularly when dealing with large-scale problems. To achieve this, we focused on minimizing the overhead caused by framework book-keeping and used concurrency when possible to reduce the cost of CPU-intensive operations. In particular, this requirement informed our use of NumPy (a numerical computing library for Python, optimized for vector calculations). Our emphasis on performance also informed our use of concepts such as lazy module loading, where components are only loaded in to memory when required thus allowing the system to scale effectively.

Ease of Use

A third requirement that we took into account was for the system to be easy to use, particularly for non-expert users who may not have a background in CPP. To achieve this, we focused on creating minimal APIs which make use of parameter defaults. The advantage of this approach is that it allows new users to work with those APIs without extensive knowledge; while simultaneously allowing more advanced users to achieve more specific goals by tweaking the default parameters.

Verifiability

Finally, we took into account the need for the system to be verifiable, particularly in academic settings where reproducibility is an important aspect of research. To achieve this, our design focused heavily on determinism in code. Essentially this means limiting side-effects and making the flow of information through the system explicit such that it's easier to reason about and test. Naturally, this lead our design to borrow concepts from functional programming such as pure functions. It also lead us to incorporate architectural patterns which have become popular in web applications, such as streams and immutable state management.

Security

Another important non-functional requirement was for the system to be secure, and fault-tolerant in how it handles unexpected input or circumstances. To achieve this, we adopted a defensive programming approach, where components assume data arriving from other locations is potentially tainted or otherwise inaccurate. For instance, as Python is a weakly-typed language, the data type of a particular parameter may be invalid. Hence, each component performs some degree of verification that the data it receives lies in accordance with its intended schema. This requirement also informed our use of test-driven development (TDD) to ensure that APIs are implemented correctly and handle invalid conditions appropriately and deterministically.

5.4 Architecture

Our implementation of the GEM algorithm consists of two primary decoupled components - a Python-based CPP environment engine called PyCPP, and the GEM algorithm itself. As discussed, the PyCPP framework serves as a versatile platform for implementing, simulating, and comparing CPP algorithms, and constitutes the majority of our code base. Its modularized architecture groups source files according to the conceptual role they play in the formulation of a CPP problem. For instance, environments for different CPP scenarios a grouped together (however still decoupled); similarly different agents are grouped together. The organization of our code-base into these categories aims to reduce the work required in translating abstract formulations of CPP algorithms into concrete solutions. Users will not have to "reformulate" their algorithm in order to implement it; they only need to translate each component (agents, states, actions, environments) into a corresponding self-contained piece of code. It also allows us to inherit several desirable properties from mathematical descriptions of CPP algorithms, such as purity and ease of verification. Purity refers to the proliferation of pure, deterministic functions that have no side-effects, while ease of verification enables the correctness of an implementation to be more readily verified.

Our highly decoupled architecture implements points of interaction between components using standardized, minimal APIs which do not divulge the internal state of any particular component. There are several advantages to this approach:

- **Modularity.** Decoupling components and using standardized APIs allows for greater modularity and easier maintenance of the code-base. Changes to one component should not require fundamental changes to any other components. Thus the code base can be built up concurrently with different teams working on different aspects in parallel.
- **Flexiblity.** This modular approach provides flexibility to users, allowing them to mix and match components to create new solutions. For instance, users can write a single agent that works with several alternative environments.
- **Scalability.** The code-base can scale almost arbitrarily without requiring fundamental structural changes. Each particular component can essentially be treated as a "plug-in", which is unnecessary for the system to operate. Components can be added, removed or replaced without disrupting the system as a whole. Additionally, adding components does not incur a larger memory footprint as components can be lazily-loaded (i.e. loaded only when required).
- **Simplicity.** By encapsulating the implementation details of each component and porting instrumentation to APIs, new users can easily work with previously developed components without requiring an in-depth understanding of their internal structure.
- **Reusability.** Our modular approach encourages reusability of components. For example, the implementation of an agent in one CPP scenario could potentially be reused in another CPP scenario with only minimal modification.
- **Testability.** Finally, this approach promotes the use of test-driven design (TDD) in which components are tested on the basis of whether they correctly implement certain APIs. As discussed in Chapter 6 — Testing, TDD is particularly effective in increasing the speed of development, which was suitable for this project.

5.5 Overview

In this section, we offer a condensed overview of the architecture of our CPP framework in terms of its structure and components. In later sections we discuss individual components in more depth.

5.5.1 Package Structure

Our framework is packaged using the Python package manager Poetry [2]. Poetry is a modern Python packaging and dependency management tool that simplifies package management by consolidating several common packaging tasks into a single workflow. It provides a simple command-line interface (CLI) to manage package dependencies, virtual environments, and builds.

The use of Poetry has several advantages for the development and distribution of our package. First, it allows us to define the package's dependencies in a simple and easily readable `pyproject.toml` file, which is then used by Poetry to automatically install and manage the necessary dependencies when the package is installed. This eliminates the need for users to manually install dependencies and reduces the chances of version conflicts or other issues that can arise from manual dependency management. Second, Poetry provides an isolated environment for the package and its dependencies, which provides separation from other Python projects on the same host machine. This serves to reduce the change of dependency conflicts and also aids in portability.

In terms of the package structure, Poetry uses a standardized directory layout that is easy to understand and navigate. The key files used by Poetry to manage the package are the `pyproject.toml` file, which contains metadata about the package, its dependencies, and build settings, and the `poetry.lock` file, which is automatically generated by Poetry and contains the exact versions of all the package's dependencies, ensuring that the package can be installed consistently across different systems. The initial file structure created after running `poetry new` is listed below:

```
src/
src/__init__.py
tests/
tests/__init__.py
poetry.lock
pyproject.toml
README.md
```

This scaffold provides the basic skeleton for the project.

In summary, Poetry was a suitable choice for packaging our project due to its elegant dependency management, environment isolation features, and a simple directory structure. The resulting package is highly portable, and straightforward to install.

Building off of Poetry's basic scaffold, we flesh out the `src/` directory with project-specific directories and source files. Leading to the following structure:

```
./data
./scripts
./src
./src/actions
./src/agents
./src/environments
./src/observations
./src/parameters
./src/rewards
./src/states
./src/streams
./tests
```

The directory structure of our CPP framework is designed to be modular and scalable, allowing for easy addition of new components or modification of existing ones. The `src/` directory serves as the main hub for all the source files, while the `tests/` directory holds all unit tests for the framework.

We note the shallow depth of our structure. In recent years, particularly within the web development community, there has been a trend to towards shallow directory layouts as they offer some substantial advantages [14]. One advantage of shallow directory layouts is that they reduce the mental effort required to navigate a given project. This is especially significant in large-scale projects where the number of directories and files can quickly become overwhelming. Additionally, shallow directory layouts may improve the build times of a project as the build system does not have to traverse through multiple nested directories to find the required files. This can result in faster compilation times and therefore an overall reduction to overhead during development. Moreover, shallow directory layouts can facilitate code reuse and promote modularity by providing a clear separation of concerns between different components of the project. This can help make the code-base more maintainable and easier to extend in the future. Overall, the use of a shallow directory structure in our CPP framework is intended to make the development process more streamlined and efficient.

Located inside the `src/` directory, we have the following sub-directories:

```
./src/actions
./src/agents
./src/environments
./src/observations
./src/parameters
./src/rewards
./src/states
./src/streams
```

These directories correspond to the principle concepts used within the framework; for instance, `environments/` corresponds to the concept of a CPP environment — which is essentially a function that transforms state according to a given action. A similar correspondence between theoretical concepts and concrete implementations exists for each of the other directories.

Two directories we would like to highlight here are `parameters/` and `streams/`. The former is a primary instance of where we incorporate functional programming into our design. Parameters are immutable objects which hold configuration options for agents, environments and other variable aspects of the framework. They provide an elegant way to decouple functions and classes from their data-dependencies, allowing said functions to remain entirely pure — acting as transformations to data rather than processes with side-effects. We discuss this concept in more depth in later sections of this chapter.

On the other hand, streams provide a standardized protocol for inter-component communication. A stream is essentially a buffer which stores messages temporarily until they

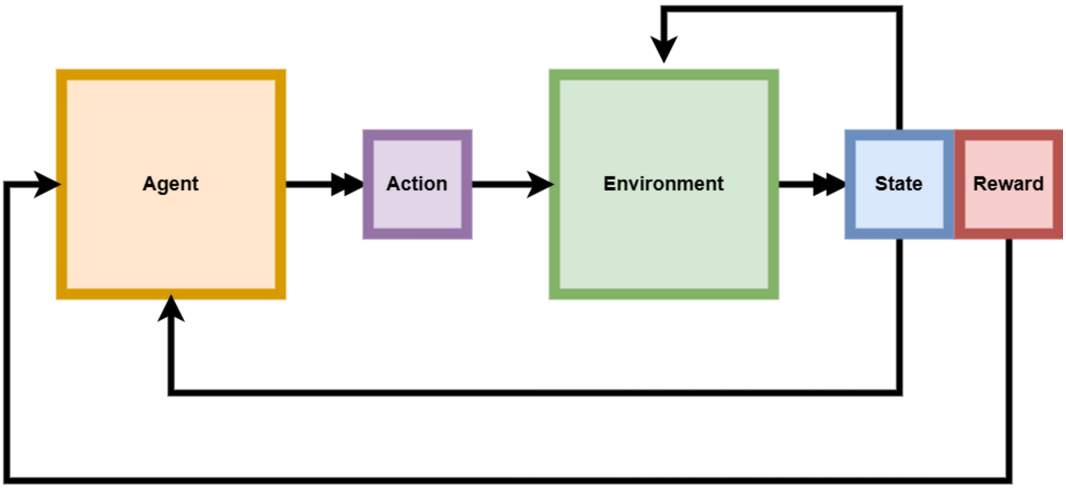

FIGURE 5.1: PyCPP Framework Architecture

are read by a subscriber. Streams allow components to exchange various forms of information such as user events and state changes, without needing to laboriously implement distinct methods for each interaction.

In addition to these core directories, we have also included the `data/` and `scripts/` directories. The `data/` directory holds any static data files that may be required by the framework, such as default AOI layouts, or configuration files. The `scripts/` directory contains utility scripts that can be used to automate certain tasks such as exporting benchmark results.

Overall, our directory structure is designed to promote modularity, ease-of-use, scalability, and transferability of work between different parties.

5.5.2 Components

Figure 5.1 illustrates the essential flow of information and interaction between different components within our framework. Each square represents a particular object of function. Single-headed arrows indicate that the object receives a particular item, whereas double-headed arrows indicate that the object emits an item. For instance, the `Agent` takes in a `State` and `Reward` object and returns an `Action` object. Similarly, the 'Environment' takes in the action produced by the agent and the prior state, and returns a new state, along with a 'Reward' object. This flow closely mimics the state-action-environment paradigm often used to formulate sequential decision making scenarios including CPP and reinforcement learning problems.

Our architecture adopts an entirely functional approach to implement interactions between components. In this approach, objects passed between different components are immutable, and once an item has been created, it cannot be modified by any component. Instead, state updates are implemented by creating new instances of objects. This design decision has several benefits over traditional object-oriented programming.

Firstly, the functional approach makes the system far more deterministic and predictable as a whole. At any given point in time, we can reason about how the system will behave

by simply observing the content of items passed between components such as `Action` and `State`. This contrasts with a system where components rely on external factors such as the state of a file in memory, data which cannot be easily serialized. Such external factors can lead to unexpected results and make the system unpredictable.

Secondly, the functional approach allows events to be replayed simply by running the system from a particular state. This feature can aid in debugging as well as "undo" and "redo" mechanics. The state of the system at any point is entirely transparent, recoverable and serializable.

Moreover, the functional approach enhances the code's readability and maintainability, making it easier for developers to reason about the interactions between components. We can express all mutations to the system as a set of deterministic pure function applications. Due to Python's increasing support for functional programming, this style of code can be written very concisely using `dataclasses` and other features.

The fundamental building blocks of our CPP framework are represented by a collection of interdependent and modular components that work in tandem to form a comprehensive and versatile system capable of implementing a wide range of CPP algorithms. We outline these components below:

States

The `State` component is responsible for defining the current state of the environment in which agents are embedded. States can hold arbitrary data, however in most cases represent some encoding of the AOI; for instance as a graph containing each viewpoint and the edges between viewpoints, or a matrix encoding the locations of obstacles. States may also define some utility functions to perform basic transformations which several environments might want to use — e.g. moving an agent left or right within the AOI.

Agents

The `Agent` component is an abstract class representing the concept of an agent. Note that, in this context "agent" refers to a *single* decision maker, which may or may not correspond to a team of mobile robots. In multi-agent CPP, the individual robots are essentially virtualized as part of the environment state. The `Agent` then decides how every robot in the team should move at every time step.

The specific way in which an agent decides to application specific, however is assumed to depend only on the current state of the environment. That is, we make the Markov assumption. This allows us to represent agents as pure functions which take in a `State` object and return an `Action` object. Nonetheless, to support iterative approaches such as Q-learning, we also provide a way for agents to receive reward signals from the environment via `Observation` objects.

Actions

The `Action` component represents the action that an agent takes in a particular state. The action object encapsulates all the necessary information about the agent's decision. Actions are often represented using enumerations (e.g. `(NorthAction, EastAction, SouthAction,`

`WestAction)`), or immutable structures containing specific data about the action being performed (e.g. `MoveAction(direction = 0, agent = 4)`).

Environments

The `Environment` component provides an abstract class for arbitrary CPP environments. Remaining with our functional paradigm, an environment can be seen as the transition function for a deterministic finite-state automaton (DFA), in which the alphabet consists of `Action` objects, and the states are `State` object. Environment also provide an optional reward to agents, such as uniformity in the case of GEM. We require that the reward depends only on the current state of the environment. Hence, like agents, environments are pure functions. Also note that environments can be paired with arbitrary agents, assuming the agent emits actions the environment is able to interpret. There is no coupling between agents and the environment they inhabit. In fact, agents require no knowledge of their environment, since all relevant information is encapsulated in the given `State` object. An agent can inter-operate with an environment if the agent and environment use the same action-protocol, however, otherwise the two components are entirely independent. This provides considerable flexibility when simulating CPP algorithms, and especially when comparing solutions, as users can easily switch between agents and environments.

Rewards

The `Reward` component is an abstract class for implementing reward functions and coverage metrics. `Reward` objects are simply pure functions that take a `State` object and return a real-valued reward. They are entirely independent from agents and environments, however necessarily rely on the `State` having a particular format. For instance, grid-based rewards necessarily require `State` objects that encode a grid. To reduce any potential limitations incurred by this dependence, users can implement general-purpose state representations such as graphs and design rewards that operate on said graphs. Another possible design pattern which can be applied is *Adaptor*. In this case, an auxiliary class is used to translate a given `State` into a format which a particular reward function can interpret. In practice, we find that the dependence of rewards on states is not a hindrance as CPP algorithm authors generally specify reward functions and metrics in the same context where they specify state representations.

Observations

The `Observation` component is an abstract class which simply encapsulates an `Action`, the `State` prior to the action, the state after the action and the reward received by performing the action. This information is communicated to agents after they perform an action in the environment in order to support reinforcement learning, capture of an agent's decisions and debugging.

Parameters

The `Parameters` component is responsible for storing the parameters that are used to configure the environment and agents. The parameters object encapsulates all the necessary information about the configuration of the CPP problem, such as the number of agents, the size of the environment, the time horizon, etc. Extracting these variables into reusable objects greatly simplifies and reduces the code required to configure environments, agents

and other entities at run time. It also allows simulations to be shared more easily as the entire configuration can be serialized and stored in a file. In conjunction with 'State' objects, this makes all data in the system entirely deterministic, reproducible and portable.

Streams

The `Stream` component is an abstract class providing boilerplate functionality for message streams. As previously described, streams provide a high-bandwidth, visible method for inter-component communication. In our GEM implementation, we use streams to communicate user interactions asynchronously to the environment.

In summary, using these components we have been successful in devising a highly general, modular and flexible framework for representing and simulating CPP problems that rests on established paradigms from functional programming and closely resembles how CPP is typically formally defined in the literature.

Documentation

In our CPP framework, we utilized Python doc-strings [1] to provide in-line documentation for all modules, functions, classes, and methods. Python doc-strings are string literals that appear at the beginning of definition. By adhering to a standardized format, doc-strings allow us to easily generate high-quality documentation from the source code itself.

One of the major advantages of using doc-strings is that they make it easier for us to maintain the documentation and keep it up-to-date as the code-base evolves. When a function's signature or behavior is altered, we can update the corresponding doc-string simultaneously to reflect the changes. Locating documentation with the code it references is helpful in two directions. Firstly, it motivates documentation to accurately reflect the code itself. Secondly, it motivates code to be well-organized and type-safe using `types` and assertions.

Furthermore, automated tools like Sphinx can use doc-strings to generate documentation in a variety of output formats, such as HTML, PDF, and LaTeX. This can save a significant amount of time and effort that would otherwise be required to manually create and maintain documentation in separate formats.

5.5.3 Back-end

Previously we delved into the abstract classes provided by the PyCPP framework, which offer a foundational architecture for modeling coverage path planning problems. These classes prescribe a general blueprint for designing CPP algorithms and simulations, albeit lacking in concrete implementations. To develop a comprehensive solution, we proceeded to construct concrete implementations of States, Actions, Environments, Agents, and Parameters, which form integral components of the system.

The system can be divided into two primary sub-systems: the front-end and back-end. The back-end implements core logic and algorithms for States, Actions, Environments, Agents, and other essential functions. In contrast, the front-end functions as a thin-client, transmitting commands to the back-end and receiving updates about the environment's state. The front-end then renders the environment state to the user in a suitable format,

such as a graphical representation. In this section, we discuss our back-end components in detail.

5.5.4 Parameters

The `Parameters` component, plays a crucial role in the implementation of GEM. It provides a structured way of storing configuration options, which can be passed to arbitrary components. In our implementation, we extend (inherit) from the abstract base class to create a concrete instantiation called `MapParameters`. This class has several attributes that can be used to customize the grid world for GEM simulations. For instance, `width` and `height` attributes specify the size of the map. The `numberOfAgents` attribute indicates how many agents will be placed on the map at the beginning of each simulation. Similarly, the `numberOfObstacles` attribute sets the number of obstacles on the map. By defining these attributes in the `MapParameters` class, we can easily customize the map to create different grid-based CPP problems. The default map size of 10x10 is a commonly used choice, but larger map sizes (e.g., 20x20) can also be specified. `MapParameters` is formally defined as follows:

```
MapParameters {
        width (int) default: 10
        height (int) default: 10
        numberOfAgents (int) default: 5
        numberOfObstacles (int) default: 10
}
```

5.5.5 States

To represent the state of the environment at a particular point in time, we inherit from 'State' to create `MapState`. This state component is responsible for storing the pheromone distribution at time t, as well as the locations of each agent and obstacle within the AOI; all of which are stored using 2D-arrays (matrices) with NumPy. This structure is given below:

```
MapState {
        agentCoordinates (MapCoordinates) optional
        obstacleCoordinates (MapCoordinates) optional
        pheromone (NumPy Array) optional
}
```

In addition to providing a state representation, `MapState` additionally defines a number of common operations, expected to find repeated use in third-party code using the component.

For instance, `updatePheromone`; this operation updates the pheromone distribution according to the exponential decay rule discussed in previous chapters. It utilizes generic programming whereby the specific function applied to each pheromone value is passed in as an argument rather than hard-coded.

Another notable programming technique used within this operation is *vectorized* operations through the NumPy library. In the code below, we see that he pheromone distribution matrix `pheromone` is added to a bitmap representing the locations of agents within the AOI `self.agentCoordinates.bitmap`. In general, such an operation would require iterating through each pheromone level and adding it to a corresponding bit within the `agentCoordinates` matrix. However, using NumPy, such an operation can be parallelised such that it runs in constant, or significantly reduced time.

```
def updatePheromone(self, updateFunction):
    return MapState(
      parameters = self.parameters,
      agentCoordinates= self.agentCoordinates,
      obstacleCoordinates = self.obstacleCoordinates,
      pheromone = updateFunction(self.pheromone) \
        + self.agentCoordinates.bitmap,
    )
```

Finally, we note the use of immutability within this operation — rather than mutating state directly, `updatePheromone` in fact returns a new `MapState` instance, with any differences applied. While instantiating a new object does incur a small degree of overhead, we argue that this is a small price to may for the substantial gains in predictability and visibility such an approach yields. All other operations implemented by the `MapState` component are designed according to this functional paradigm.

Actions

The `Action` is an abstract bass class used to represents the actions an agents can take in its environment. If we treat environments as DFAs, then actions can be considered the alphabet of the DFA encoding the environment model.

To this end, we implement the `MapAction` class, which is a concrete instantiation of the abstract `Action` class provided by the PyCPP framework. The `MapAction` class is specifically designed to capture the positions of each agent on the next time step. This simple structure is sufficient to encode the necessary information required to execute the agents' intended behaviors.

The `MapAction` class contains a `List` of `MapCoordinate` objects, which represents the positions of each agent in the environment. This encoding allows the agents to make individual decisions about their next positions while taking into account the positions of the other agents in the environment. The `MapAction` structure is defined as follows:

```
MapAction {
  agentCoordinates (MapCoordinates)
}
```

Here, `MapCoordinates` is a type defined as a `List` of `MapCoordinate` objects. The `MapCoordinate` object is a simple data structure that contains the `x` and `y` coordinates of a position in the environment. By defining the `MapAction` class in this way, we are able to represent the actions taken by the agents in a concise and efficient manner. This allows us to perform simulations with a large number of agents and a complex environment without incurring significant computational overhead.

5.5.6 Rewards

In CPP problems, the design of reward functions is crucial for the effectiveness of an agent's decision-making process. The purpose of a `Reward` class is to determine the relative utility of an agent's actions by computing the reward obtained by the agent when transitioning from one state to another. In other words, the reward function is a mapping from the state-action space to the real numbers, and it quantifies the desirability of taking a certain action from a given state. A `Reward` class is a pure function of the state-action pair, which means that the same state-action pair will always yield the same reward value.

The `UniformityMapReward` class is a concrete instantiation of the abstract base class `MapReward`, which generalises reward functions for grid-based planning problems.

`UniformityMapReward` implements the `next_reward` method, which takes two `MapState` objects as input (the current state and the next state), and returns the reward obtained by the agent. In this case, the reward is simply the `pheromonePerplexity` of the next state, which is a measure of how uniform the pheromone distribution is across the environment. A more uniform distribution yields a higher reward. By using this reward function, the agents are incentivized to explore the AOI uniformly, which ensures that the pheromone distribution is spread evenly across the environment.

The code for `UniformityMapReward` is shown below:

```
@dataclass
class UniformityMapReward(MapReward):
  parameters: MapParameters

  def nextReward(
    self,
    state: MapState,
    nextState: MapState
  ) -> float:
    return next_state.pheromonePerplexity
```

5.5.7 Environments

Up to this point, we have discussed various components that play a passive role in the our implementation of GEM. These components, such as `MapState`, `UniformityMapReward`, and `MapParameter`, are used to represent the state of the environment, the utility of a state transition, and to store configuration options for grid-based planning problems, respectively. However, to create a fully functioning simulation, we require two active aspects

- an `Environment` and an `Agent` that emits actions within this environment. In this section, we turn our attention to the implementation of the `MapEnvironment` component, which is essentially a DFA operating on `MapState` and `MapAction` instances.

`MapEnvironment` is defined as an abstract class as follows:

```
@dataclass
class MapEnvironment(Environment):
  parameters: MapParameters

  def initialState(self) -> MapState:
    pass

  def nextState(
    state: MapState,
    action: MapAction
  ) -> MapState:
    pass
```

Two primary methods are declared (however not defined): `initialState` and `nextState`. The former method is used to construct an default initial `MapState` for the environment, for instance, containing agents and obstacles located at random positions, or indeed simply a blank state with no agents or obstacles. While simulations can be instagated from any state, providing an initial default helps to reduce boilerplate code. The former method, `nextState`, as the name suggests, is intended to be used as a state-transition function for the environment model. The simplest example of such a transition function, would be the identity function in which actions have no influence whatsoever on the environment. However, in most real-world scenarios, actions do affect the environment in some way, and hence the `nextState` method should implement a state transition function that captures these dynamics. The implementation of `nextState` varies depending on the specifics of the problem. For instance, in a grid-based planning problem, a `nextState` method might update the location of an agent based on a valid action that has been taken. It might also update the state of obstacles or add new objects to the environment.

To implement a transition function that responds appropriately to `MapAction` actions, we can simply update the current `MapState`, such that the positions of agents are set to that specified in a given `MapAction` instance. To see this, recall the structures for both `MapState` and `MapAction`:

```
MapState {
        agentCoordinates (MapCoordinates) optional
        obstacleCoordinates (MapCoordinates) optional
        pheromone (NumPy Array) optional
}
```

```
MapAction {
  agentCoordinates (MapCoordinates)
```

}

Since `MapAction` instances explicitly specifies the coordinates each agent should move to on the next time step, we can simply replace the current state's `agentCoordinates` attribute with the that given by the `MapAction`. This simple transformation is sufficient to implement a static CPP environment in which agents can move freely. However, such an environment does not meet all of the functional requirements we outlined in section 3.1. In particular, the following requirement has not been met:

The framework should support interactivity whereby the user can influence the environment in real-time for online CPP methods.

Fortunately, we have already devised an aspect of the PyCPP framework which makes implementing such a feature quite straightforward, namely *streams*. Streams, provide an asynchronous, high-bandwidth and general purpose method for communicating state between components. To implement a stream, we inherit from the `Stream` abstract base class, defined as follows:

```
@dataclass
class Stream:
  messages: List[StreamMessage]

  @property
  def hasMessages(self) -> bool:
    return bool(self.messages)

  def write(self, message: StreamMessage) -> None:
    self.messages.append(message)

  def read(self):
    while self.hasMessages:
      yield self.messages.pop(0)
```

As shown in this code, a `Stream` is effectively a queue containing `StreamMessage` structures, in which writing to the stream corresponds to enquing a particular message, and reading corresponds to dequing a message. Note however, that unlike regular queues, streams emit values using the `yield` keyword, which allows applications to iterate through currently buffered messages (automatically dequing each message after being read).

To apply this construct to the task of implementing an interactive map environment, we first define a sub-class of `StreamMessage` called `MapInteractionStreamMessage` in order to encapsulate the type of interactions we would like to support. At a minimum, we would like to support the following types of interactions:

- `AddMapInteraction` — an interaction corresponding to the action of adding an object to the environment — either a CPP agent or an obstacle.
- `RemoveMapInteraction` — an interaction corresponding to he action of removing an object from the environment.

After some analysis, we find that both types of interaction essentially involve the same pieces of information, namely, the location of the object to be added or removed, and the type of object to be added or removed (agent or obstacle). The only distinguishing factor between these two interactions is whether they perform insertion or removal. As such, we generalise to the following `MapInteraction` structure:

```
MapInteraction {
        action (MapAction)
        coordinate (MapCoordinate)
        entity (MapEntity)
}
```

where `MapAction` and `MapEntity` are enumerations defined as follows:

```
MapAction {
        AddMapAction,
        RemoveMapAction,
}

MapEntity {
        AgentMapEntity,
        OobstacleMapEntity,
}
```

Finally, a `MapInteractionStreamMessage` is defined as a `StreamMessage` with a single field `interaction` containing the `MapInteraction` to be communicated to the environment.

```
MapInteractionStreamMessage {
        interaction (MapInteraction)
}
```

Through this taxonomy of structures, we can elegantly communicate a range of interactions to the environment in real-time by transmitting them as stream messages. The environment can then read the interaction stream and implement logic for applying said interactions. We implement this behaviour into a `MapEnvironment` sub-class called `InteractiveMapEnvironment`.

Considering the components, discussed so far, we provide a more detailed diagram of our system architecture in Figure 5.2.

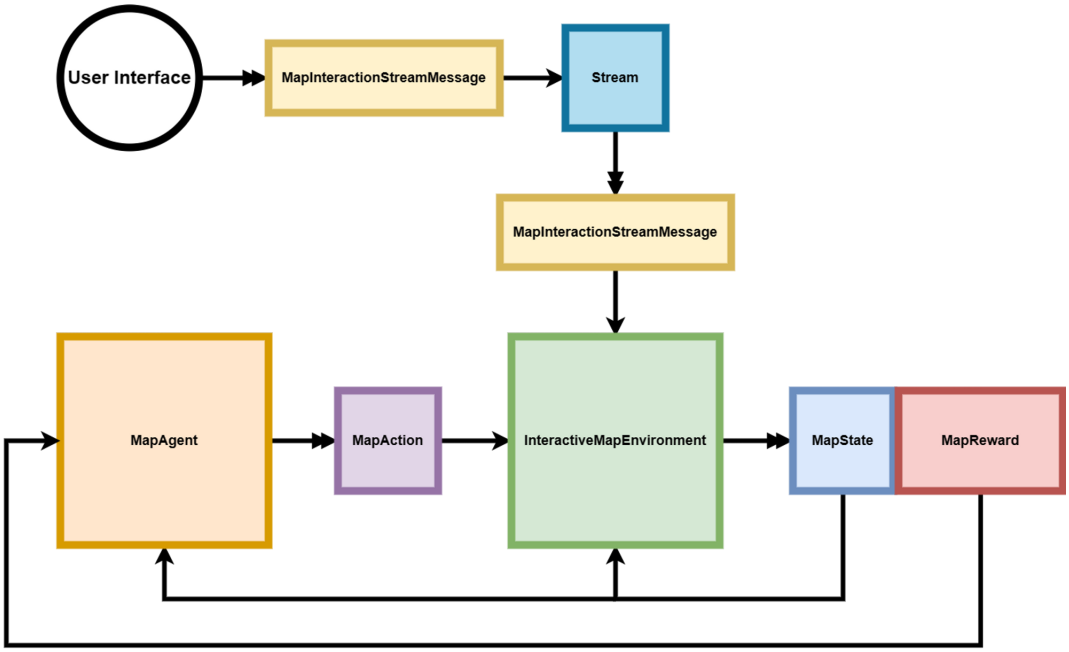

FIGURE 5.2: PyCPP Framework Architecture (Revised)

In the diagram, the user interface (UI) provides an entry point for all user-based interactions with the simulation. The UI detects keyboard and mouse inputs, using built-in APIs and emits a series of corresponding `MapInteractionStreamMessage` messages which are delivered to the environment's interaction stream. The environment `InteractiveMapEnvironment` utilizes the received interactions as well as actions originating from `MapAgent` to produce a `MapState` and `MapReward`. The asynchronous nature of the stream allows interactions to be delivered at any point in time during the simulation.

5.5.8 Agents

The final key component of our system is of course the agents themselves. Agents are pure functions which take in a state (an optionally a reward), and produce an action. They implement all decision making capabilities of a particular CPP solution. For our purposes, it was sufficient for us to implement only three agents:

- `GreedyMapAgent` — an agent which acts according to the GEM algorithm, directing individual mobile robots to move to whichever adjacent cell has the least pheromone deposited on it.
- `RandomWalkAgent` — an agent which causes each robot to move in a random direction at every time step. This was used as a control to aid in evaluating GEM; if GEM were to perform worse than a random walker, it would certainly be considered defective as a CPP solution. We also used this agent rather extensively during development to help test the framework's efficiency and during debugging to identify problems unrelated to the agent component.
- `ExpertMapAgent` — an agent which uses an existing, well established CPP algorithm to make decisions. We specifically used a variant of DARP known as Replannable DARP (R-DARP), which recomputes the DARP-optimal solution upon detection of changes to the environment. Such an solution would be inefficient and cumbersome

to use in practice, however it provided a strong baseline we would use to benchmark against GEM.

As mentioned at the beginning of this chapter, there is a notable scarcity of publicly available implementations for DARP and other CPP algorithms. We were able to find one correct implementation. However this implementation was poorly documented, difficult to install and rather poor in terms of code quality. We refactored, repackaged and partially re-implemented this solution into a refined Python package called `darpy`. We then included this as a third-party dependency within our R-DARP implementation .

The code for our `GreedyMapAgent` is shown below:

```
@dataclass
class GreedyMapAgent(MapAgent):
  parameters: MapParameters

  def move_agent(
    self,
    state: MapState,
    coordinate: MapCoordinate
  ) -> MapCoordinate:

    neighbours = state.availableNeighbours(coordinate) \
      .coordinates

    if not neighbours:
      return coordinate

    # Find neighbour with least pheromone

    leastPheromone = float('inf')
    leastNeighbour = None

    for neighbour in neighbours:
      neighbourPheromone = state\
        .pheromone[neighbour.y, neighbour.x]

      if neighbourPheromone < leastPheromone:
        leastPheromone = neighbourPheromone
        leastNeighbour = neighbour

    return leastNeighbour
```

The agent is implemented as a pure function that takes in the current `MapState` and the `MapCoordinate` of a particular mobile robot, and returns the new coordinate the robot should move to. As discussed, `MapState` holds the current state of the environment, including the location of each robot and the amount of pheromone deposited on each cell

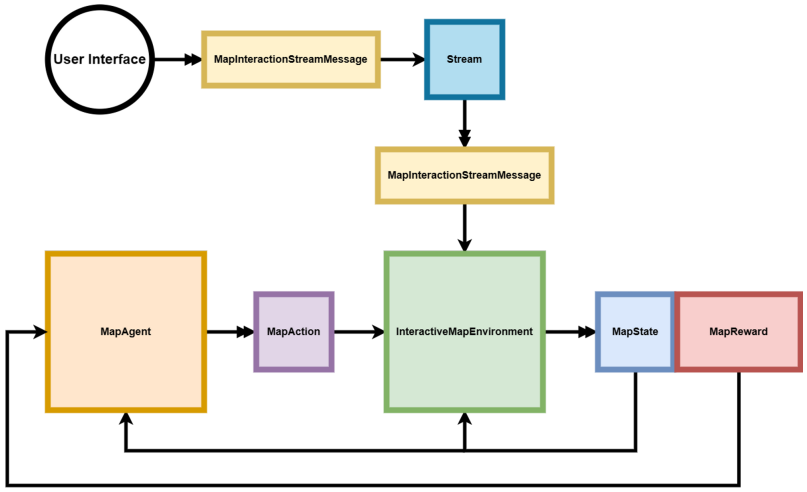

FIGURE 5.3: PyCPP Framework Architecture (Revised)

of the AOI. The `availableNeighbours` method of the `MapState` object is used to find all neighbouring cells the robot may move to (i.e. those which are unoccupied and within bounds of the AOI). If there are no available neighbours, the function simply returns the current coordinate of the robot, since there is nowhere for the robot to move. If there are available neighbours, the function iterates through each neighbouring cell and retrieves the amount of pheromone deposited on it via the 'pheromone' matrix. The `leastPheromone` and `leastNeighbour` variables are used to keep track of the neighbouring cell with the least amount of pheromone deposited on it. At the end of the loop, the function returns the coordinate of the neighbouring cell with the least amount of pheromone, which is the new location that the robot should move to.

5.5.9 Front-end

In Figure 5.3, we can see that the front-end user interface (UI) is entirely decoupled from the rest of the system (the back-end). This separation allows us to develop multiple front-end interfaces for a given back-end. In our implementation, we provide two alternative interfaces to the GEM simulation. We firstly implement a graphical client which supports adding and removing obstacles within the environment. It also clearly renders the locations of obstacles and agents in the AOI, as well as the pheromone intensity of each cell.

In addition, we also provide a read-only console client which provides an ASCII-based rendering of the pheromone distribution for each time step, as well as the current RPPL score. Both front-ends make use of the `next()` built-in function to access the next state and reward of the environment an in iterative fashion.

Finally, we utilize a stripped down variant of the console client in Jupyter notebooks for use in statistical evaluation of GEM compared to alternative approaches.

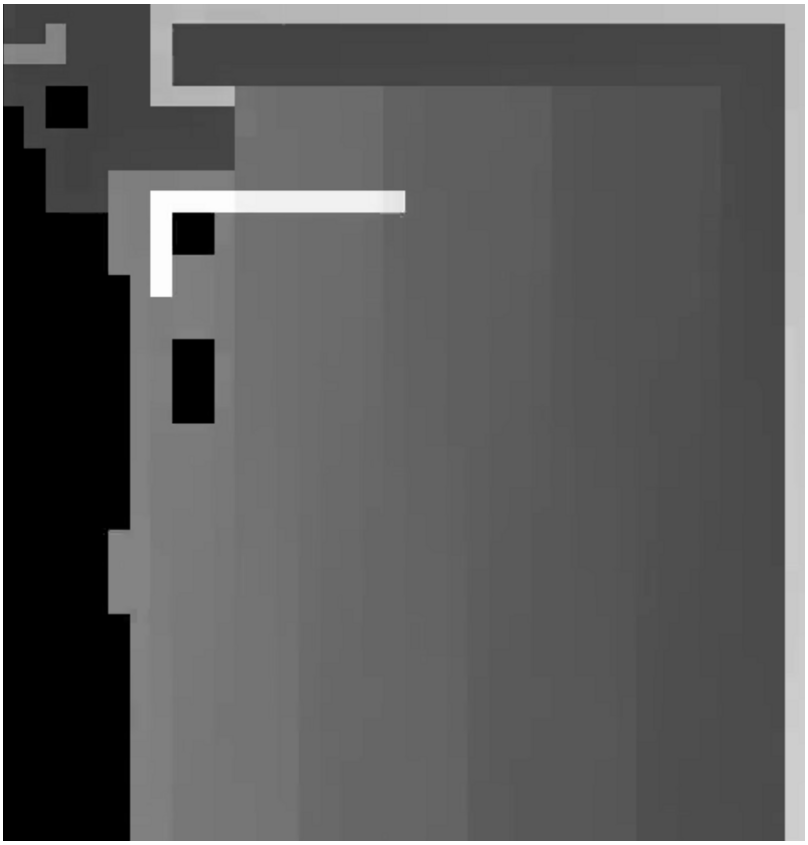

FIGURE 5.4: An early version of the graphical client, showing the pheromone intensity of each viewpoint by the cell brightness.

5.6 Summary

In this chapter we provided an in-depth explanation of our design and implementation of a general purpose framework for developing, testing and simulating CPP scenarios, as well as a concrete implementation of the GEM algorithm for distributed online coverage path planning. We discussed our requirements gathering process and the ways in which those requirements informed our selection of design principles and the overall architecture of our solution. Specifically, our software places a strong emphasis on functional programming and predictability through the use of immutable states, pure functions and by designing organizing components within the framework such that they closely resemble their mathematical counterparts. Based on subsequent user feedback and testing, we find that this approach was ultimately successful in achieving the goals we set out to achieve. In the next chapter, we describe our approach to rigorously testing each component within the system, as well as the software as a whole.

FIGURE 5.5: A more developed version of the graphical client with clearer depictions of agents, and real-time statistics, taking into account user-feedback.

FIGURE 5.6: Several obstacles have been inserted into the environment by the user through mouse controls.

```
[[ 0  0  0  0  0  0  6  6  7  7  7  7  7  7  7  6  6  6  6  6]
 [ 0  0  0  0  0  0  0  6  0  8  8  8  8  8  8  8  6  8  8  9  7]
 [ 0  0  0  0  0  0  0  6  0  8  0  8  7  0  8  8 18 18  9  9  7]
 [ 0  0  0  0  0  0  0  6  0  7  0  8  7  0  8  8  9  9  9  9  7]
 [ 0  0  0  0  0  0  0  6  0  7  0  9  7  0  0  0  0  0  0  9  7]
 [ 0  0  0  0  0  0  0  0  0  7  0  9  7  0  0  0  0  0  0  9  7]
 [ 0  0  0  0  0  0  0  0  0  7  0  9  7  0  0  0  0  0  0 10  7]
 [ 0  0  0  0  0  0  0  0  0  7  0  9  7  0  0  0  0  0  0  0  7]
 [ 0  0  0  0  0  0  0  0  0  7  0  9  7  0  0  0  0  0  0  0  8]
 [ 0  0  0  0  0  0  0  0  0  7  0 10  6  0  0  0  0  0  0  0  8]
 [ 0  0  0  0  0  0  0  0  0  6  0  0  6  0  0  0  0  0  0  0  8]
 [ 0  0  0  0  0  0  0  0  0  6  0  0  6  0  0  0  0  0  0  0  8]
 [ 0  0  0  0  0  0  0  0  0  6  0  0  6  0  0  0  0  0  0  0  8]
 [ 0  0  0  0  0  0  0  0  0  6  0  0  6  0  0  0  0  0  0  0  8]
 [ 0  0  0  0  0  0  0  0  0  6  0  0  6  0  0  0  0  0  0  0  9]
 [ 0  0  0  0  0  0  0  0  0  6  0  0  0  0  0  0  0  0  0  0  9]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  9]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  9]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  9]
 [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0 10]]
iteration: 25
RPPL: 23.96
```

FIGURE 5.7: A read-only console client showing pheromone levels explicitly.

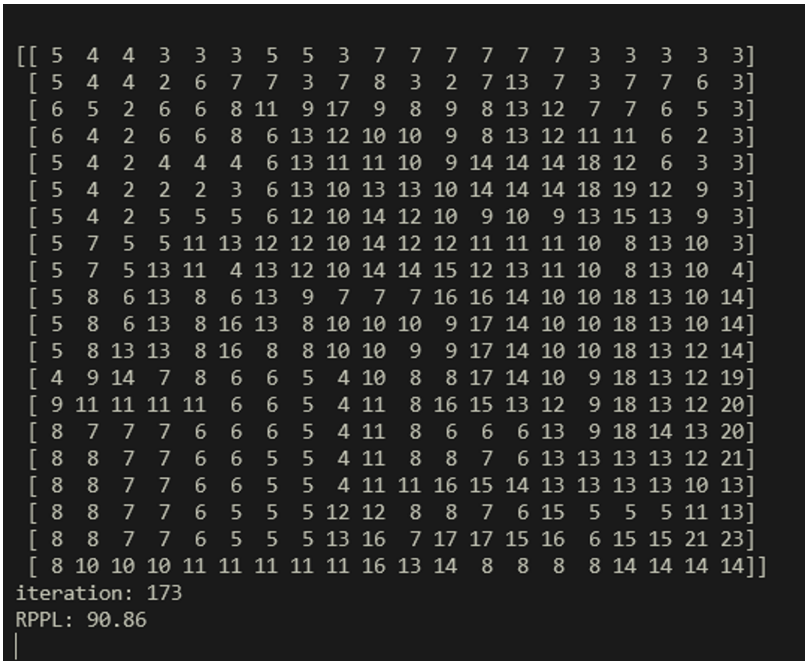

FIGURE 5.8: The same console client, after 173 iterations.

Chapter 6

Testing

6.1 Introduction

Testing is a crucial aspect of any software product, and our work is no exception. In the context of this project, we identified three primary aspects which require rigorous testing in order for us to gain confidence in the viability of our solution:

- **Algorithms.** To what extent are the algorithms used by our solution successful in terms of efficiency (e.g. time and space complexity) and reliability?
- **APIs.** To what extent is our solution implemented correctly, regarding the behaviour of APIs, including user-facing APIs and inter-component APIs?
- **User Experiences.** To what extent does our solution offer a positive user experience? How intuitive is the software?

Each of these aspects can be tested most effectively using a specific strategy. For instance, algorithms can be tested through benchmarking and runtime analysis to discover bottlenecks and inefficient operations. In contrast, APIs are more effectively tested using test-driven development (TDD) whereby tests for the correctness of an API implement are written prior to writing the implementation itself. TDD can be performed at several levels of abstraction to ensure that low-level components operate correctly, and higher-level systems utilizing those components also abide by well defined specifictions. TDD is a hilghy effective testing strategy and particulary well-suited to our modular architecture, however is says nothing about how usable our solution is in practice. This is where our final testing strategy, beta testing, plays a signficant role. In addition to technical tests, beta testing allows us to recieve invaluable user feedback relating to their experience of using the software. Beta testing attempts to ascertain how generally intuitive and understandable the software is to a someone we would expect to use it. Our specific approach here revolved around informed and uninformed beta tests. Informed beta tests equip users with a guide on how to use the software; whereas uninformed tests are more open-ended and require users to learn features and functionality from scratch.

Overall, the testing phase of the project was critical in ensuring that our solution was effective, reliable, and user-friendly. By following a rigorous testing process, we were able to identify and address issues early on in the development process, ultimately leading to a more robust final product.

6.2 Algorithm Testing

Algorithm testing aims to measure the efficiency and reliability of algorithms used within a software solution. To measure efficiency, one can generally take two approaches: theoretical or empirical. In the theoretical approach, the goal is to derive a closed-form expression for the asymptotic time and space complexity of an algorithm with respect to certain varying parameters (e.g. the size of an array). We have already conducted this form of analysis in Chapter 4 where we derive the time complexity of GEM for physical and simulated scenarios. The advantage of this approach is that is generalises to any system in which the algorithms find use (assuming the model of computation is equivalent), and does not depend on specific hardware discrepancies. However, for medium or large-scale projects with many interacting components, deriving precise complexity bounds for each part of the code can become laborious and time consuming. Hence, a more fruitful approach is to measure efficiency empirically by gathering statistics about how the run-time varies with respect to certain parameters. In many cases, an approximate asymptotic complexity can be inferred inductively by plotting run-time against some size parameter; for instance, if the underlying complexity is quadratic, this will be rather straightforward to recognise through visual inspection of the plot. The facilitate this statistical approach, we utilized several tools, including Python's builtin `timeit` and `cProfile` modules in order to estimate the run-time of various components within the system. We also used more sophisticated graphical profilers including Pyflame, vprof and Scalene which are capable of generating rich summaries of how execution time is distributed among the various operations performed during a program's execution. Visual profilers are particularly useful as they more effectively communicate where potential bottlenecks may lie. After experimenting with each profiling technique,

A second important aspect of algorithm testing (and software testing more broadly) is reliability testing. In this case, we are concerned with how the algorithm operates under extreme or limiting input conditions. Extreme inputs are those who's characteristics are in some way much greater in magnitude than what is typically expected by the algorithm. For instance, consider an algorithm designed to locate all occurrences of a regular expression within a given text file. In this case, feeding the algorithm a large, non-textual file such as a video could be considered extreme input — while technically valid input it most certainly contradicts the algorithm's intended use. Handling of extreme input can generally be achieved by firstly detecting its presence, and secondly deciding either to reject the input, or transform it into a more palatable form (e.g. through truncation). It is also essential to limiting inputs or consider edge cases during algorithm testing. Edge cases are inputs that are at the boundaries of the input space and are often the most likely to cause failures or inconsistencies within an algorithm. For example, consider an algorithm intended to sort lists of integers. In this scenarios, edge cases could include an empty list or a list with only one element. Similarly to extreme inputs, such edge cases may undermine assumptions about how an algorithm will be used while it was being developed. In general, it's crucial to develop algorithms with explicit awareness of any assumptions being made.

6.2.1 Profiling

Profiling with Pyflame

Pyflame is a profiling tool for Python applications that generates flame graphs to visualize the execution of code. It can be used to identify performance bottlenecks by showing which functions are taking up the most time during the execution of a program. Pyflame works by attaching to a running Python process and sampling the call stack at regular intervals to collect data on how much time is spent in each function. This data is then used to generate a flame graph, which is a visualization that shows the call stack as a series of nested rectangles, with each rectangle representing a function and its size proportional to the amount of time spent in that function. The resulting graph can be used to quickly identify where the majority of time is being spent in a program and which functions are potential candidates for optimization.

To incorporate Pyflame into our test suite, we wrote a simple Python script, that loads a particular module from our `src/` directory, obtains the current process' PID (process ID) using Python's built-in `os.getpid()` function, and then executes Pyflame as a console command, with the aforementioned PID as a parameter. The code for this script is show below:

```
import os
import subprocess
import sys
from _thread import start_new_thread as thread

module = sys.argv # The module to test.
pid = os.getpid()  # Get the current process' PID.

# Load the module's code.

with open(module, 'r') as file:
        module_code = file.read()

# Run the module in a new thread.

thread(exec, (module_code, ))

# Run Pyflame.

command = f"pyflame -p {pid} > profile.txt"
subprocess.run(command, shell=True)
```

This script can be used to construct a Pyflame profile for any particular module within our PyCPP framework or GEM implementation. As such, we used it extensively when evaluating the performance of our front- and back-end components.

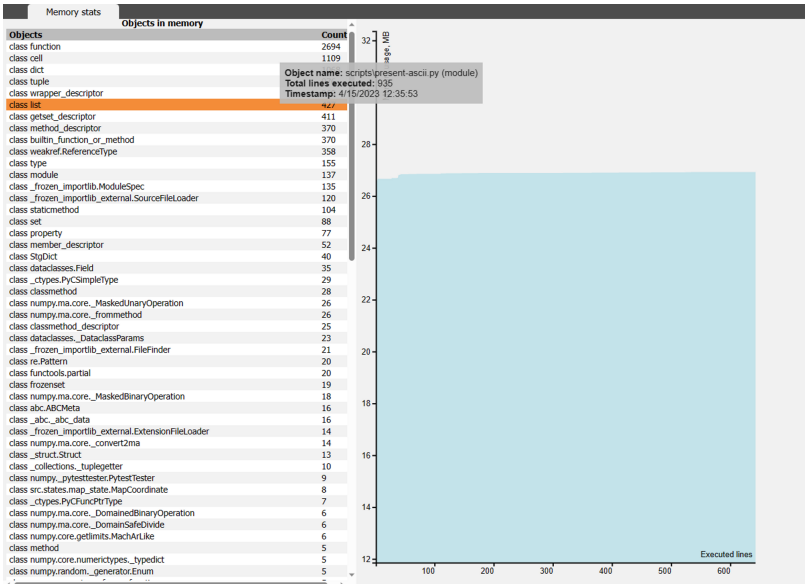

FIGURE 6.1: vprof UI.

Profiling with vprof

In addition to Pyflame, we also utlized vprof as it provides a rich user-interface for exploring various aspects of a program's runtime performance, namely it's memory and CPU footprint.Unlike Pyflame, vprof is web-based and supports remote access, which aided in real-time demonstrations and explanations of our solution's performance.

Profiling with Scalene

Finally, we used a more recently developed profiler called Scalene. In comparison to Pyflame and vprof, Scalene stands out for its superior performance and minimal overhead, often running orders of magnitude faster, while providing richer statistics. Another unique feature, although one we did not utilize, is its ability to suggest code improvements through large language models.

Scalene provides profiling at several degrees of granularity simultaneously (i.e. both line and function-level profiling). This was beneficial for us as our systems contains a mixture of script-like and library-like code whereby the former is more amenable to line-level profiling and he latter is more suited to function-level profiling.

Scalane can be used either as a console command or programmatically as a Python module within code. To emphasize reproducibility of our testing approach, we chose to utilize the second option. For instance, to profile the time required to perform a state transition within the 'InteractiveMapEnvironment' environment, we used the we used Scalene's 'start()' and 'stop()' profiling methods:

```
from scalene import scalene_profiler

...
```

```
simulation = environment.simulate(
  state = environment.initial_state(),
  agent = greedy_agent,
  reward = reward,
  iterations = NUMBER_OF_ITERATIONS,
)

# Turn profiling on...

scalene_profiler.start()

# Perform a state transition...

next(simulation)

# Turn profiling off...

scalene_profiler.stop()
```

In summary, our approach to profiling during this project was rather comprehensive and involved using several popular profiling tools, each offering distinct advantages which were applicable to our specific goals. Namely, Pyflame provided a simple method for quickly generating flame graphs during development, allowing us to actively identify points of inefficiency. Meanwhile, vprof and Scalene provided more sophisticated profiling tools with detailed line- and function-level statistics. These were more useful during the later phases of development where we engaged in refactoring and aggressive optimization.

6.2.2 Reliability

To test the reliability of our algorithms, we employed a mixture of unit tests and more advanced techniques including fuzzing.

Unit Testing with PyTest

Unit testing involves testing individual components or functions of the software in isolation to ensure that they behave as expected. This is done by writing test cases that specify the input and expected output of the function, and then running these test cases to verify that the function behaves as expected. We used the PyTest framework for writing and running unit tests.

PyTest is a popular testing framework that offers several advantages over others. For instance, it allows for easy configuration of tests through the use of fixtures, which help to reduce code duplication by allowing developers to share common test resources across multiple test cases. PyTest also provides a powerful assertion library, which makes it easy to write test cases that are both expressive and easy to read. Finally, PyTest is simply rather easy to install and work with.

To integrate tests into our project, we placed them into the `tests` directory. This made it easy to keep tests organized and separate from the main code-base. PyTest automatically discovers and executes these upon issuing the `pytest` command. Due to the functional nature of our system (discussed in Chapter 5), writing an extensive suite of tests was not particularly difficult. For instance, to be confident that state transitions would always work as expected in `InteractiveMapEnvironment` it sufficed to check whether that component alone implemented the transition function correctly. Since it is decoupled from other components, and since states are immutable, we did not need to test how it behaved conditional on external factors such as "which user interface is being used." We were able to apply the same localized testing strategy to all other components of our system as they all similarly revolve around pure functions.

Fuzzing with Atheris

To test extreme and limiting inputs, we utilized a testing technique known as fuzzing. Fuzzing involves repeatedly running a target program on random or unusual input in order to identify poorly handled edge cases and other obscure inputs which lead to errors or crashes. The fuzzing framework we used, known as Atheris is based off `libFuzz` and allows us to conduct fuzzing programmatically.

Our fuzzing scripts successfully identified a number of edge-case inputs which, if left unchecked could lead to unexpected crashes, creating a negative user experience. One example was the edge scenario in which the environment contained more agents than accessible viewpoints. In this case, some agents would occupy the same viewpoint, leading to a failed assertion, in turn causing the simulation to crash. To fix this issue, we added logic to ensure users cannot specify more agents than there are unoccupied viewpoints.

Overall, by combining unit testing and fuzzing, we were able to gain a high degree of confidence in the reliability of our algorithms, ensuring that our solution is both efficient and robust.

6.3 API Testing

In conjunction with testing the granular efficiency and reliability of the algorithms used within each component, an equally important aspect of our testing strategy was integration and APIs. Here we are primarily concerned with the correctness of each component in terms of whether it accurately implements a specification, and the extent to which different components communicate effectively. The predominant testing paradigm we used for this endeavour was test-driven development (TDD).

TDD is a software development approach that emphasizes writing automated tests before writing concrete implementations. The TDD process involves the following steps:

- 1. Write a test that fails.
- 2. Write the minimum amount of code required to make the test pass.
- 3.Refactor the code to improve its design and maintainability while ensuring that all tests still pass.

This methodology is highly applicable to modular systems such as our PyCPP framework. If each API is implemented correctly, and each component utilizes other components according to their API specifications, then the system as a whole should operate correctly. In practice, side-effects can introduce nondeterministic aspects which jeopardise the soundness of that statement. However, fortunately, in our architecture, we specifically minimize the presence of side-effects, thus increasing the utility of TDD in our process. We again chose to write TDD tests with PyTest owning to its simplicity.

6.4 User Experience Testing

User experience (UX) testing is a critical aspect of software development that aims to evaluate how users interact with a software solution and the extent to which it meets their expectations. In our project, we adopted a comprehensive approach to UX testing that involved both informed and uninformed beta testing, as well as post-usage interviews and user-feedback surveys.

Informed beta testing refers to the process of giving users a guide or manual that outlines how to use the software before they begin testing. This type of testing is particularly useful for identifying usability issues that may not be immediately apparent to users who are not familiar with the software. For instance, we provided users with a step-by-step guide on how to use each of the features of our PyCPP framework, and asked them to provide feedback on their experience using the software. By observing how users navigate through the software and where they encounter difficulties, we were able to identify areas where we could improve the usability of the software.

In contrast, uninformed beta testing involves giving users access to the software without any prior guidance or instruction. This type of testing is useful for assessing how intuitive the software is and how easily users can figure out how to use it without any external help. We asked a group of users who were unfamiliar with our PyCPP framework to test the software and provide feedback on their experience. By observing how users interacted with the software and identifying where they encountered difficulties or confusion, we were able to identify areas where we could improve the user interface and overall user experience.

Post-usage interviews and user-feedback surveys were also an important component of our UX testing strategy. We conducted interviews with users who had completed beta testing and asked them about their experience using the software. We also asked them to rate various aspects of the software, such as its ease of use, intuitiveness, and overall effectiveness. Additionally, we distributed a user-feedback survey that asked users to provide more detailed feedback on their experience with the software, including suggestions for improvements and areas where they encountered difficulties.

These elements of testing were performed concurrently with algorithm testing and API testing throughout the course of the project. For instance, several users recommended displaying reward statistics for GEM and other algorithms as part of the simulation GUI. This recommendation later implemented into our primary front-end and tested using the API and algorithm testing methods previously discussed.

Based on our latest user-feedback, we believe the current iteration of our software delivers a generally positive user experience and achieves one of our primary aims — to simplify the process of developing CPP algorithms and simulations for research and practical applications.

6.5 Summary

In this chapter, we discussed the critical role that testing plays in the development of software. We identified three primary aspects of software that require rigorous testing: algorithms, APIs, and user experience.

We explained that algorithm testing aims to measure the efficiency and reliability of algorithms used within a software solution and can be done theoretically or empirically. API testing, on the other hand, focuses on the correctness of each component in terms of whether it accurately implements a specification and the extent to which different components communicate effectively. We highlighted that test-driven development (TDD) is a highly applicable methodology for API testing.

Finally, we discussed user experience testing, which aims to assess the usability and intuitiveness of a software solution. We explained that we conducted both informed and uninformed beta testing and post-usage interviews and user-feedback surveys to gain feedback and improve the user experience of our PyCPP framework. Overall, we emphasized that by following a rigorous testing process, we were able to identify and address issues early on in the development process, ultimately leading to a more robust final product.

Chapter 7

Conclusion

7.1 Achievements and Evaluation

The aim of this project was three-fold. Primarily, our goal was to develop a novel approach to online coverage path planning which competes with or surpasses state-of-the-art methods. Secondly, and as part of achieving this goal, we also aimed to provide a generalised theoretical framework for CPP which unifies the objectives of offline and online CPP approaches. Finally, on observing the general scarcity of publicly available resources concerning CPP, we aimed to develop an open-source, well-maintained and easy to use Python framework for researchers, students and developers working with CPP and related problems. The framework aimed to provide a blueprint for developing CPP algorithms in a well-organized, functional manner which closely resembles mathematical formalism.

In evaluation, we believe we have been successful in achieving all free of these goals to an extent. We introduced GEM, a simple, elegant online CPP algorithm based on stigmergic communication and swarm intelligence. In Chapter 4, we showed through empirical evaluation that GEM performs competitively with ideal online CPP approaches such as R-DARP. Despite its simplicity and naive approach to environment modelling, we find that it nonetheless quickly covers the target AOI using only a few agents, whilst avoiding obstacles and minimizing re-coverage of saturated regions. Through changes to the pheromone decay function it can also be adapted to support importance-weighted coverage whereby certain viewpoints require more frequent visits.

To evaluate GEM and R-DARP, we proposed a novel metric known as uniformity which generalised conventional metrics used in offline and online coverage. We prove several interesting mathematical properties of uniformity, such that maximizing uniformity implies maximizing coverage completeness and minimizing overlap. Thus, it offers a viable means for casting CPP as an optimization problem, and/or as a Markov decision process (MDP). Due to these properties, we argue uniformity achieves the goal of providing a general framework for CPP; however more work is required to solidify its relationship to application-based metrics such as number of turns.

Finally, we successfully developed PyCPP, as an open-source framework for modelling, simulation and comparison of different CPP algorithms. Our rigorous testing procedure, coupled with an overt focus on functional programming and determinism have resulted in a framework which is highly reliable and elegantly organized. While still in its early stages, we hope PyCPP will be used to accelerate future research into coverage path planning,

among both beginners and experts. One area for improvement is our documentation. We have provided documentation in the form of READMEs on GitHub and auto-generated documentation from Python doc-strings. However, in future we would like to provide more extensive documentation hosted on `readthedocs.io` or another website with more in-depth tutorials showing how to use the framework. Additionally, we would like to refactor some of our separate GEM implementation code to improve readability.

Overall, this project has been successful in achieving its primary objectives, corroborated by positive user-feedback from our most recent iteration.

7.2 Future Work

To the best of our knowledge, our work has introduced several novel aspects to the CPP space, namely GEM and uniformity. GEM represents just one of many algorithms that could be developed to maximize uniformity via a team of mobile agents. In particular, we see reinforcement learning (RL) as a clear next step in terms of research. RL, in particular deep RL, could provide more rapid maximization of uniformity by utilizing inter-agent communication and more comprehensive features from the environment (i.e. wider field-of-views). We have an upper-bound on the speed at which uniformity can be maximized in discrete-time environments, and GEM, while effective, does not come very close. As such, this suggests there is significant room for improvement within the general area of uniformity-based CPP. We are excited for potential future approaches which surpass GEM.

7.3 Final Thoughts

In conclusion, this project has made significant contributions to the field of coverage path planning. We have developed a novel online CPP algorithm, GEM, that performs competitively with state-of-the-art methods. Moreover, we have introduced a new metric, uniformity, which provides a general framework for CPP by unifying offline and online approaches. The development of PyCPP has also been a significant accomplishment, providing a reliable and well-organized framework for researchers, students, and developers working with CPP. While we have achieved our primary objectives, there is still much room for improvement and further research. We look forward to seeing future developments in this exciting field.

Bibliography

[1] PEP 257 – Docstring Conventions | peps.python.org.

[2] Poetry - python dependency management and packaging made easy.

[3] What is user centered design?, Apr 2023.

[4] Bo Ai, Maoxin Jia, Hanwen Xu, Jiangling Xu, Zhen Wen, Benshuai Li, and Dan Zhang. Coverage path planning for maritime search and rescue using reinforcement learning. *Ocean Engineering*, 241:110098, 2021.

[5] Richard Bormann, Florian Jordan, Joshua Hampp, and Martin Hägele. Indoor coverage path planning: Survey, implementation, analysis. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1718–1725, 2018.

[6] Kale Champagnie. Github - greedy-entropy-maximization/darpy: About a user-friendly implementation of the darp algorithm for multi-agent coverage path planning (mcpp).

[7] Kale Champagnie. Github - greedy-entropy-maximization/gem: A fast, online coverage path planning algorithm based on stigmergic communication.

[8] Kale Champagnie. Github - greedy-entropy-maximization/pycpp: A flexible, elegant framework for coverage path planning in python.

[9] Kale Champagnie. Github - greedy-entropy-maximization/statistical-path-planning: Statistical models for multi-agent path planning in dynamic environments.

[10] Chris Denniston, Thomas R. Krogstad, Stephanie Kemna, and Gaurav S. Sukhatme. Planning safe paths through hazardous environments, 2018.

[11] Lefteris Doitsidis, Stephan Weiss, Alessandro Renzaglia, Markus Achtelik, Elias B. Kosmatopoulos, Roland Y. Siegwart, and Davide Scaramuzza. Optimal surveillance coverage for teams of micro aerial vehicles in gps-denied environments using onboard vision. *Autonomous Robots*, 33:173–188, 2012.

[12] Sumit Gajjar, Jaydeep Bhadani, Pramit Dutta, and Naveen Rastogi. Complete coverage path planning algorithm for known 2d environment. In *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, pages 963–967, 2017.

[13] Enric Galceran and Marc Carreras. A survey on coverage path planning for robotics. *Robotics and Autonomous Systems*, 61(12):1258–1276, 2013.

[14] family=Kelly given i=D, given=Daniel. How to structure a large scale vue.js application - vue school articles.

[15] Mahdi Hassan and Dikai Liu. Ppcpp: A predator–prey-based approach to adaptive coverage path planning. *IEEE Transactions on Robotics*, 36(1):284–301, 2020.

[16] Muzaffer Kapanoglu, Mete Alikalfa, Metin Ozkan, Ahmet Yazıcı, and Osman Parlaktuna. A pattern-based genetic algorithm for multi-robot coverage path planning minimizing completion time. *J. Intell. Manuf.*, 23(4):1035–1045, aug 2012.

[17] Athanasios Ch. Kapoutsis, Savvas A. Chatzichristofis, and Elias B. Kosmatopoulos. Darp: Divide areas algorithm for optimal multi-robot coverage path planning. *J. Intell. Robotics Syst.*, 86(3–4):663–680, jun 2017.

[18] Wen-Hao Li, Tao Zhao, and Song-Yi Dian. A multi-robot coverage path planning method based on genetic algorithm. In *2021 International Conference on Security, Pattern Analysis, and CyberneticsSPAC)*, pages 13–18, 2021.

[19] Yongkai Liu, Jiawei Hu, and Wei Dong. Decentralized coverage path planning with reinforcement learning and dual guidance, 2022.

[20] Junjie Lu, Bi Zeng, Jingtao Tang, and Tin Lun Lam. Tmstc*: A turn-minimizing algorithm for multi-robot coverage path planning, 2022.

[21] Gonzalo Mier, Joã o Valente, and Sytze de Bruin. Fields2cover: An open-source coverage path planning library for unmanned agricultural vehicles. *IEEE Robotics and Automation Letters*, 8(4):2166–2172, apr 2023.

[22] Arman Nedjati, Arman Nedjati, Béla Vizvári, and Jamal Arkat. Complete Coverage Path Planning for a Multi-UAV Response System in Post-Earthquake Assessment. *Robotics*, 5(4):26, 12 2016.

[23] DongKi Noh, WooJu Lee, Hyoung-Rock Kim, Il-Soo Cho, In-Bo Shim, and SeungMin Baek. Adaptive coverage path planning policy for a cleaning robot with deep reinforcement learning. In *2022 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–6, 2022.

[24] DongKi Noh, WooJu Lee, Hyoung-Rock Kim, Il-Soo Cho, In-Bo Shim, and SeungMin Baek. Adaptive coverage path planning policy for a cleaning robot with deep reinforcement learning. In *2022 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–6, 2022.

[25] Bijan Ranjbar-Sahraei, Gerhard Weiss, and Ali Nakisaee. A multi-robot coverage approach based on stigmergic communication. In Ingo J. Timm and Christian Guttmann, editors, *Multiagent System Technologies*, pages 126–138, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[26] Nils Reimers and Iryna Gurevych. Making monolingual sentence embeddings multilingual using knowledge distillation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2020.

[27] Mina G. Sadek, Ahmed M. El-Garhy, and Amr E. Mohamed. A dynamic cooperative multi-agent online coverage path planning algorithm. In *2021 16th International Conference on Computer Engineering and Systems (ICCES)*, pages 1–9, 2021.

[28] Marina Torres, David A. Pelta, José L. Verdegay, and Juan C. Torres. Coverage path planning with unmanned aerial vehicles for 3d terrain reconstruction. *Expert Systems with Applications*, 55:441–451, 2016.

Appendix A

User-centred Design

In this appendix, we provide a number of artefacts we utilized during the design process to gather user requirements and feedback.

A.1 User Personas

In software development (and product development more broadly), personas are simplified, fictional representations of the *typical* users. Personas are generally informed by interactions with groups of real users via interviews, or early access testing. In our case, we interviewed several dozen students in order to gauge what typical users of our software might want to achieve, given their goals, motivations and background. We utilized `thispersondoesnotexist.com` to generate random profile images for each persona. Two of our personas are shown in figures A.1 and A.2. High quality renders of these personas are linked in Appendix C.

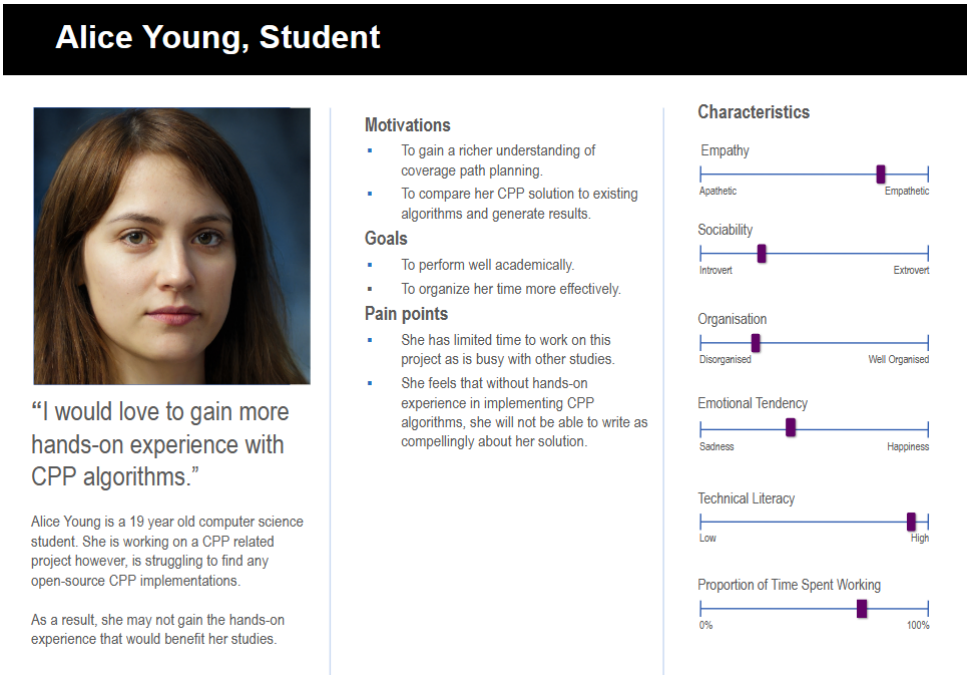

FIGURE A.1: Person, Alice Young.

A.2 Questionnaires

To inform our personas, as well as various other user-centred aspects of our design, we utilized Google Forms to deliver questionnaires. In these questionnaires we included a variety of closed and open-ended questions regarding the features of our software.

A.2.1 Questionnaire for Requirements Gathering

- 1. How would you rate your degree of experience with coverage path planning on a scale of 1 to 5?
- 2. Based on the description we provided to you about PyCPP, which features would you rate as most important to include?
- 3. List up to five additional features you think would be desirable in the PyCPP framework.
- 4. List up to five reasons you might have to use PyCPP.
- 5. How many times a week would you expect to use PyCPP?
- 6. Are you are already a user of a CPP-related application? If so, which features do you find most useful?

A.2.2 Questionnaire for User Feedback

- 1. How easy was it for you to use the PyCPP GEM simulation on a scale of one to five?
- 2. If you chose a number below three, please list any difficulties you encountered.

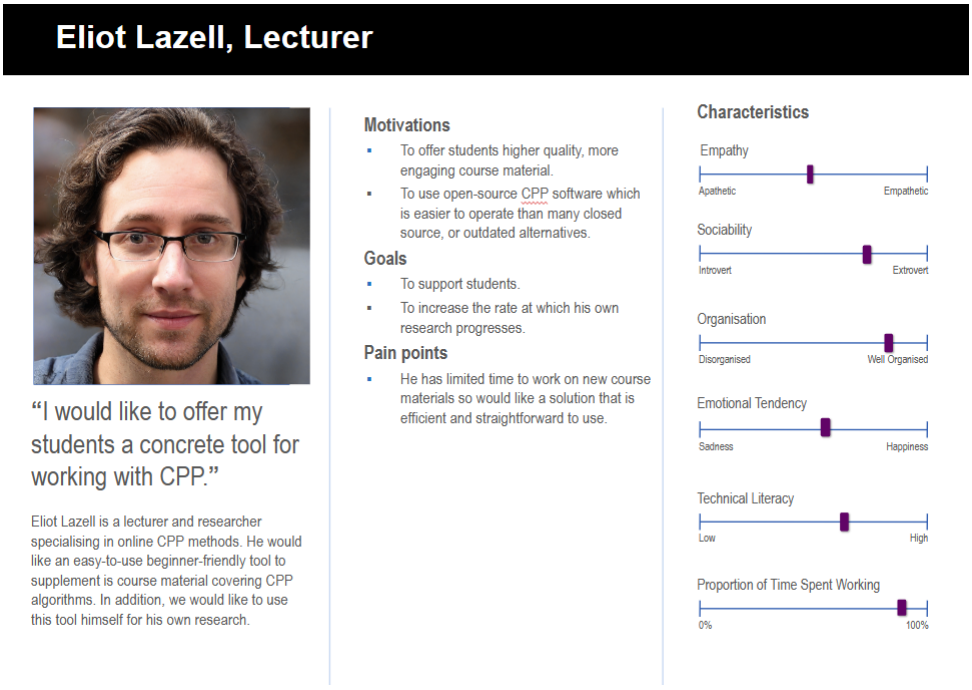

FIGURE A.2: Persona, Eliot Lazell.

- 3. If you chose a number above three, please list which aspects you found most intuitive and which you found more cumbersome.
- 4. To what extent were you able to achieve the intentions and goals you outlined in our prior discussion using the software?
- 5. When using the software, did you encounter any bugs or erroneous behaviour? If so, please describe the nature of the behaviour and if possible which action you took leading up to the event.
- 6. How would you rate the ease of installation on a scale of one to five?
- 7. How would you rate the ease of executing/launching the application on a scale of one to five?
- 8. How would you rate the performance of the application on a scale of one to five?
- 9. Do you have any additional feedback you would like to offer?

After collecting several dozen responses to our questionnaire, we sorted the responses into several clusters based on sentiment and the specific information provided. This was done automatically using `sentence-transformers` [26] and Scikit's k-means clustering implementation. This allowed us to identify common feedback topics and hence informed are subsequent design decisions.

Appendix B

Source Code

In this appendix, we provide condensed source code for our PyCPP framework, as well as the concrete implementation of GEM. The full code can be found on our GitHub (linked in Appendix C).

B.1 Actions

B.1.1 init

```
from src.actions.action import Action
from src.actions.map_action import MapAction
```

B.1.2 Action

```
from dataclasses import dataclass


@dataclass
class Action:
    """
    Defines an abstract base class for all actions.
    """
    pass
```

B.1.3 MapAction

```
from src.actions.action import Action
from src.states.map_state import MapCoordinates
from dataclasses import dataclass


@dataclass
class MapAction(Action):
    """
    Defines the actions which can be performed under
    the `MapEnvironment` environment.
```

```
    Parameters
    ----------

    agent_coordinates: MapCoordinates

        The set of coordinates agents should move to
        on the next time step.
    """

    agent_coordinates: MapCoordinates
```

B.2 Agents

B.2.1 init

```
from src.agents.agent import Agent
from src.agents.map_agent import MapAgent
from src.agents.random_walk_map_agent import RandomWalkMapAgent
from src.agents.ideal_map_agent import IdealMapAgent
from src.agents.greedy_map_agent import GreedyMapAgent
```

B.2.2 Agent

```
from src.actions import Action
from src.observations import Observation
from src.parameters import Parameters
from src.rewards import Reward
from src.states import State
from dataclasses import dataclass


@dataclass
class Agent:
    """
    Defines an abstract base class for all agents.

    Parameters
    ----------

    parameters: Parameters

        A Parameters object.

    Methods
    -------

    next_action(state: State) -> Action
```

```
        Emits an action based only on the current
        state of the environment.

    observe(observation: Observation) -> None

        Recieves an observation from the environment
        containing the reward recieved from taking an
        action.
    """

    parameters: Parameters

    def next_action(self, state: State) -> Action:
        pass

    def observe(self, observation: Observation) -> None:
        pass
```

B.2.3 MapAgent

```
from src.actions import MapAction
from src.agents import Agent
from src.parameters import MapParameters
from src.rewards import MapReward
from src.states import MapState
from src.states.map_state import MapCoordinate, MapCoordinates
from dataclasses import dataclass


@dataclass
class MapAgent(Agent):
    """
    Defines an abstract base class for all agents which work with
    the `MapEnvironment` environment.

    Parameters
    ----------

    Inherits from `Agent`.


    Methods
    -------

    Inherits from `Agent`.

    move_agent(state: MapState, coordinate: MapCoordinate)
```

```
        Moves a single agent.
    """

    parameters: MapParameters

    def next_action(self, state: MapState) -> MapAction:

        state = MapState(
            parameters = self.parameters,
            agent_coordinates = state.agent_coordinates,
            obstacle_coordinates = state.obstacle_coordinates,
            pheromone = state.pheromone,
        )

        for i, agent_coordinate in \
            enumerate(state.agent_coordinates.coordinates):

            new_coordinate = self.move_agent(state, agent_coordinate)
            state.agent_coordinates.coordinates[i] = new_coordinate


        return MapAction(
            agent_coordinates = state.agent_coordinates,
        )

    def move_agent(
        self,
        state: MapState,
        coordinate: MapCoordinate,
    ) -> MapCoordinate:

        return coordinate
```

B.2.4 GreedyMapAgent

```
from src.actions import MapAction
from src.agents.map_agent import MapAgent
from src.parameters import MapParameters
from src.rewards import MapReward
from src.states import MapState
from src.states.map_state import MapCoordinate
from dataclasses import dataclass
import random


@dataclass
class GreedyMapAgent(MapAgent):
    """
    Implements the GEM algorithm as a `MapAgent`.
```

```
    Parameters
    ----------

    Inherits from `MapAgent`.

    Methods
    -------

    Inherits from `MapAgent`.
    """

    parameters: MapParameters

    def move_agent(
        self,
        state: MapState,
        coordinate: MapCoordinate
    ) -> MapCoordinate:

        neighbours = state.available_neighbours(coordinate) \
            .coordinates

        if not neighbours:
            return coordinate

        # Find neighbour with least pheromone

        least_pheromone = float('inf')
        least_neighbour = None

        for neighbour in neighbours:
            neighbour_pheromone = state.pheromone[neighbour.y, neighbour.x]

            if neighbour_pheromone < least_pheromone:
                least_pheromone = neighbour_pheromone
                least_neighbour = neighbour

        return least_neighbour
```

B.2.5 RandomWalkMapAgent

```
from src.actions import MapAction
from src.agents.map_agent import MapAgent
from src.parameters import MapParameters
from src.rewards import MapReward
from src.states import MapState
from src.states.map_state import MapCoordinate
from dataclasses import dataclass
```

```
import random


@dataclass
class RandomWalkMapAgent(MapAgent):
    """
    Implements a random walker as a `MapAgent`.

    Parameters
    ----------

    Inherits from `MapAgent`.

    Methods
    -------

    Inherits from `MapAgent`.
    """

    parameters: MapParameters

    def move_agent(
        self,
        state: MapState,
        coordinate: MapCoordinate
    ) -> MapCoordinate:

        neighbours = state.available_neighbours(coordinate) \
            .coordinates

        if not neighbours:
            return coordinate # Can't move

        else:
            return random.choice(neighbours)
```

B.3 Environments

B.3.1 __init__

```
from src.environments.environment import Environment
from src.environments.map_environment import MapEnvironment
from src.environments.interactive_map_environment \
    import InteractiveMapEnvironment
```

B.3.2 Environment

```
from src.actions import Action
```

```
from src.agents import Agent
from src.observations import Observation
from src.parameters import Parameters
from src.rewards import Reward
from src.states import State
from typing import Iterator, Tuple
from dataclasses import dataclass


@dataclass
class Environment:
    """
    Defines an abstract base class for all environments.

    Parameters
    ----------

    parameters: Parameters

        A parameters object.

    Methods
    -------

    initial_state() -> State

        Returns the initial state for the environment.

    next_state(state: State, action: Action) -> State:

        Returns the next state for the environment based only
        on its current state an a given action.

    simulate(
        state: State,
        agent: Agent,
        reward: Reward,
        iterations: int = 10,
    )

        Simulates the environment with a given agent and
        reward function for `iterations` time steps. Returns
        an iterator of `Observation` objects.
    """

    parameters: Parameters

    def initial_state(self) -> State:
        pass
```

```
    def next_state(state: State, action: Action) -> State:
        pass


# Simulate an environment.

def simulate(
    environment: Environment,
    state: State,
    agent: Agent,
    reward: Reward,
    iterations: int = 10,
) -> Iterator[Tuple[State, State, Action, float]]:

    for i in range(iterations):

        next_action = agent.next_action(state)
        next_state = environment.next_state(state, next_action)
        next_reward = reward.next_reward(state, next_state)

        observation = Observation(
            state = state,
            next_state = next_state,
            next_action = next_action,
            next_reward = next_reward,
        )

        agent.observe(observation)
        state = next_state

        yield observation


Environment.simulate = simulate
```

B.3.3 MapEnvironment

```
from src.actions import MapAction
from src.agents import MapAgent
from src.environments.environment import Environment
from src.observations import Observation
from src.parameters import MapParameters
from src.rewards import MapReward
from src.states import MapState
from typing import Iterator
from dataclasses import dataclass
```

```
@dataclass
class MapEnvironment(Environment):
    """
    Defines an abstract base class for all map environments.

    Parameters
    ----------

    Inherits from `Environment`.

    Methods
    -------

    Inherits from `Environment`.
    """

    parameters: MapParameters

    def initial_state(self) -> MapState:
        pass

    def next_state(state: MapState, action: MapAction) -> MapState:
        pass
```

B.3.4 IteractiveMapEnvironment

```
from src.actions import MapAction
from src.agents import MapAgent
from src.environments.map_environment import MapEnvironment
from src.observations import Observation
from src.parameters import MapParameters
from src.rewards import MapReward
from src.states import MapState

from src.streams import Stream
from src.streams.map_event_stream_message import (
    MapAction,
    AddMapAction,
    RemoveMapAction,
    MapEntity,
    AgentMapEntity,
    ObstacleMapEntity,
    MapEvent,
    MapEventStreamMessage,
)

from typing import Iterator
from dataclasses import dataclass
```

```
@dataclass
class InteractiveMapEnvironment(MapEnvironment):
    """
    Implements an interactive map environment supporting user
    interactions via the `map_event_stream` stream.

    Parameters
    ----------

    Inherits from `MapEnvironment`.

    map_event_stream: Stream = Stream()

        Provides a stream for sending user iteractions to the
        map.
    """

    parameters: MapParameters
    map_event_stream: Stream = Stream()

    def initial_state(self) -> MapState:
        return MapState(self.parameters).empty

    def next_state(self, state: MapState, action: MapAction) -> MapState:

        # Apply the agent's actions.

        state = MapState(
            parameters = state.parameters,
            agent_coordinates = action.agent_coordinates,
            obstacle_coordinates = state.obstacle_coordinates,
            pheromone = state.pheromone,
        )

        state = state.update_pheromone(
            lambda x: x * self.parameters.alpha
        )

        # Apply state changes from user interactions.

        for message in self.map_event_stream.read():
            event = message.event

            coordinates = isinstance(event.entity, AgentMapEntity) \
                and state.agent_coordinates \
                or state.obstacle_coordinates

            if isinstance(event.action, AddMapAction):
```

```
                coordinates.coordinates.append(event.coordinate)

            elif isinstance(event.action, RemoveMapAction):
                for i, coordinate in enumerate(coordinates.coordinates):
                    if coordinate.index == event.coordinate.index:
                        coordinates.coordinates.pop(i)
            else:
                raise TypeError(
                    'Invalid MapAction type in map_event_stream.'
                )

        # Return the new state

        return state
```

B.4 Parameters

B.4.1 __init__

```
from src.parameters.parameters import Parameters
from src.parameters.map_parameters import MapParameters
```

B.4.2 Parameters

```
from dataclasses import dataclass


@dataclass
class Parameters:
    """
    Defines an abstract base class for all parameters.
    """

    pass
```

B.4.3 MapParameters

```
from src.parameters.parameters import Parameters
from dataclasses import dataclass


@dataclass
class MapParameters(Parameters):
    """
    Defines an abstract base class for all map parameters.
    """

    width: int = 10
```

```
    height: int = 10
    number_of_agents: int = 5
    number_of_obstacles: int = 10
    alpha: float = 0.99
```

B.5 Rewards

B.5.1 init

```
from src.rewards.reward import Reward
from src.rewards.map_reward import MapReward
from src.rewards.uniformity_map_reward import UniformityMapReward
```

B.5.2 Reward

```
from src.parameters import Parameters
from src.states import State
from dataclasses import dataclass


@dataclass
class Reward:
    """
    Defines an abstract base class for all rewards.
    """

    parameters: Parameters

    def next_reward(
        self,
        state: State,
        next_state: State
    ) -> float:
        pass
```

B.5.3 MapReward

```
from src.parameters import MapParameters
from src.states import MapState
from src.rewards.reward import Reward
from dataclasses import dataclass


@dataclass
class MapReward(Reward):
    """
    Defines an abstract base class for all map rewards.
    """
```

```
    parameters: MapParameters

    def next_reward(
        self,
        state: MapState,
        next_state: MapState
    ) -> float:
        pass
```

B.5.4 UniformityMapReward

```
from src.parameters import MapParameters
from src.states import MapState
from src.rewards.map_reward import MapReward
from dataclasses import dataclass


@dataclass
class UniformityMapReward(MapReward):
    """
    Implements the uniformity map reward. Returns the perplexity
    of the current map state.
    """

    parameters: MapParameters

    def next_reward(
        self,
        state: MapState,
        next_state: MapState
    ) -> float:

        return next_state.pheromone_perplexity
```

B.6 States

B.6.1 init

```
from src.states.state import State
from src.states.map_state import MapState
```

B.6.2 State

```
from src.parameters import Parameters
from dataclasses import dataclass
```

```
@dataclass
class State:
    """
    Defines an abstract class for all states.
    """

    parameters: Parameters
```

B.7 Streams

B.7.1 init

```
from src.streams.stream import Stream
from src.streams.stream_message import StreamMessage
from src.streams.map_event_stream_message import MapEventStreamMessage
```

B.7.2 MapEventStream

```
from src.streams.stream_message import StreamMessage
from src.states.map_state import MapCoordinate
from dataclasses import dataclass


class MapAction: pass
class AddMapAction(MapAction): pass
class RemoveMapAction(MapAction): pass


class MapEntity: pass
class AgentMapEntity(MapEntity): pass
class ObstacleMapEntity(MapEntity): pass


@dataclass
class MapEvent:
    action: MapAction
    coordinate: MapCoordinate
    entity: MapEntity


@dataclass
class MapEventStreamMessage(StreamMessage):

    # Stream message for editing the map

    event: MapEvent
```

Appendix C

Resources

In this final appendix, we provides links to online resources related to or directly produced by this project. We link to our GitHub account where you will be able to find the full source code for PyCPP and our GEM implementation, in addition to other libraries such as `darpy` and a sub-project about statistical path planning conducted during preliminary research.

- GitHub - greedy-entropy-maximization/darpy: About A user-friendly implementation of the DARP algorithm for multi-agent coverage path planning (MCPP) [6].
- GitHub - greedy-entropy-maximization/gem: A fast, online coverage path planning algorithm based on stigmergic communication [7].
- GitHub - greedy-entropy-maximization/pycpp: A flexible, elegant framework for coverage path planning written in Python [8].
- GitHub - greedy-entropy-maximization/statistical-path-planning: Statistical models for multi-agent path planning in dynamic environments [9].