Geometry-Sensitive Swarm Algorithms

by

Grace Cai

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2023

© Massachusetts Institute of Technology 2023. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
January 15, 2023

Certified by .....
Nancy Lynch
Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Geometry-Sensitive Swarm Algorithms

by

Grace Cai

Submitted to the Department of Electrical Engineering and Computer Science
on January 15, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Insect colonies, birds, and fish successfully coordinate themselves to make collective decisions through purely local interactions. Their behaviors have inspired the development of algorithms for robotic swarms. Robotic swarms consist of many simple and often identical agents that interact solely via local sensing and communication. Swarm algorithms seek to produce collective behaviors within the swarm such as aggregation, consensus achievement, and task allocation.

For both biological and robotic swarms, simulation is a powerful tool to facilitate analyzing and improving swarm models. While simulating swarm algorithms is very useful, detailed and realistic simulations can take a long time to run and gather feedback on. This often leads to simplifications, especially in the geometry of the problem, that are not representative of what swarms may see in the real word [4, 13, 44]. In this thesis, we present a new discrete modelling framework for swarm algorithms that allows agents to synchronously transition on a discrete grid. Using such a grid makes it easier to add geometric qualities to the agents' environment and allows for parallel speedups in the simulation time when developing swarm algorithms or models of biological swarms.

Using our new framework, we study the existing swarm problems of house hunting and task allocation, and provide new algorithms which are more geometrically-sensitive than previous work [4, 13, 23, 32, 42, 44].

In Chapter 3, we develop a house hunting algorithm that is able to choose the best nest even when it is very far away from the swarm's home nest or is being blocked by other poorer quality candidates. Our algorithm chooses the best quality nest much more consistently than previous work, which had not considered these geometrically challenging setups.

In Chapter 4, we develop two new task allocation algorithms for agents in an environment with unknown task locations and demands, and test these algorithms in environments of varying task density. We show that one of our algorithms, inspired by the communication of house-hunting agents via a home nest, outperforms Levy flight foraging in environments with sparse task density. Our other algorithm, inspired by communication via virtual pheromones, completes tasks even faster and performs

well at all task densities tested but requires a much larger number of agents and high agent communication.

This thesis contributes our new discrete swarm model and uses that model to develop a new N-site selection algorithm as well as two new task allocation algorithms. Our new algorithms are evaluated under more geometrically varied environments, which is enabled by the geometric simplicity of our general model. We hope that our simulation framework, along with the new algorithms we have contributed, inspire future swarm research and experimentation.

Thesis Supervisor: Nancy Lynch
Title: Professor

Acknowledgments

My history with working on swarm algorithms goes all the way back to high school, where I published my first research paper on N-site selection in my senior year. Coming to MIT, after a year of exploring, I attended by advisor's class and was excited to find that her lab conducted a similar type of research. After a few semesters of undergraduate research with her, I began this thesis in my junior spring. The path I took to getting my MEng at MIT was unusual and challenging at times, and I could not have done it without the help of everyone around me.

I would firstly like to thank my advisor, Nancy Lynch, for her years of guidance, both as an undergraduate and as a masters student. When I reached out to her in my sophomore year, she welcomed me to her lab and she continues to be a supportive and dedicated advisor. Nancy helped me come up with the idea for the general modelling framework presented in this thesis, and has spent countless hours reviewing my ideas, paper drafts, and more. I truly appreciate the time and energy she has put into discussing swarm research with me and admire the vigor she brings to every lab meeting. Chapters 2 and 3 are joint work with her.

I would also like to thank my collaborator, Noble Harasha, for his help and enthusiasm in implementing and testing the PROP algorithm in Chapter 4. I really enjoyed working with him on task allocation and appreciate his thoughtfulness and attention to detail. Chapter 4 is joint work with Noble and Nancy.

Thank you to Zhi Wei Gan and Julian Shun for their collaboration on a parallel implementation of the model presented in this thesis. The parallel speedups that they have been working on are very inspiring and it has been great to learn about the techniques they used to achieve them.

The former and current members of the TDS group helped me significantly as well. I'd like to thank Jiajia Zhao for showing me what the path of a master's student was like while I was in undergrad and for mentoring me in my house hunting research throughout sophomore year. I would like to thank Brabeeba Wang, Sabrina Drammis, and Keith Murray for their feedback on my work at lab meetings.

I am a strong believer that a great personal life is key to doing great work, and my friends and family have given me immeasurable love and support. I appreciate my mom, Qi Zhao, my dad, Duo Cai, and my sister, Elizabeth Cai for their emotional energy and their weekly calls from afar. I appreciate my wonderful friends Paolo Adajar, Allison Borton, Zawad Chowdhury, Matthew Cox, Laura Cui, Ashley Lin, and Andrew Komo (among others) for having many thoughtful conversations with me throughout the years. Last, but certainly not least, the biggest thanks to my partner, Alex Gu, for his joy, creativity, and thoughtfulness, and for always being there for me.

Contents

List of Figures

Chapter 1

Introduction

In nature, large groups of ants, bees, fish, and even humans can work together to make decisions and solve problems in a decentralized manner [9, 41, 46]. Through only local sensing and interactions, these groups are able to work together and come to a consensus when needed. Studying and modelling these biological swarms give us insight into their behaviour and has also inspired algorithms for robotic swarms to solve tasks like foraging, navigation, task allocation, and house hunting [18].

In the study of both natural and artificial swarms, simulating the swarm behavior is very helpful. From a biological perspective, it is useful to compare simulated swarms to real swarms of ants and bees in order to determine if a model correctly replicates the swarms' behavior [42, 16]. From an artificial swarm perspective, simulations are crucial in testing out new algorithms and demonstrating their effectiveness, especially because formal proofs of correctness or convergence are very challenging to show [25, 27]. Unfortunately, simulations of a large number of agents can take a long time to run, creating a bottleneck in the development and testing of swarm algorithms.

We introduce a formal discrete mathematical model for the simulation of swarm algorithms, where agents are positioned among a discrete grid of vertices and can only move from vertex to vertex in synchronous rounds. The model is simple enough to ease simulation and testing time, which can take up a large portion of swarm algorithms research [8]. Furthermore, because motion is discrete, our model may make it simpler to analyze and prove information about swarm algorithms.

We have used our model to develop algorithms for and improve upon the well-known and well-studied swarm problems of house-hunting and task allocation via two main papers in Chapters 3 and 4 respectively.

Our first algorithmic contribution is a house hunting (also known in swarm literature as N-site selection) algorithm that is more sensitive and robust to the geographic placement and distances of candidate sites. Previous work [44, 13, 7] evaluated house hunting algorithms in mostly simple scenarios with two candidate nests equidistant from the home nest. However, real house hunting ants are able to choose the best candidate nest in more geographically complex setups [19]. We show that the strategies used by existing swarm algorithms are far less likely to find the best nest in more challenging environments where the highest quality nest is far away from home or blocked by another lower quality nest. We then provide a new algorithm, tested in our simulation framework, that is able to remain accurate in geometrically challenging environments.

We have also developed two task allocation algorithms for environments with unknown task locations and demands and evaluated their performance on varying task densities. A large majority of previous task allocation algorithms assume known task locations and demands [4, 23], which is unlikely in task allocation situations such as search and rescue or detecting mines. Our first algorithm, the House Hunting Task Allocation algorithm (HHTA), was inspired by the state machine models of house hunting algorithms [44], which revolve around using a home nest as a location for communication between agents. Our second virtual pheromone inspired algorithm (PROP), uses two types of agents, propagators and followers [49]. Propagator agents spread task information to each other, which follower agents use to find nearby tasks they can perform. We compared our two algorithms to the Levy walk foraging algorithm and found that HHTA outperforms the Levy flight best in low task densities, and PROP outperforms both HHTA and the Levy flight in medium to low task densities but requires many more agents. Our findings contribute two new algorithms and show that the geometry of the environment (the task density) greatly affects relative algorithm performance and is a factor that should be considered in more depth.

Lastly, we also detail ongoing work as well as provide ideas for potential future work using our model. We are currently in the process of developing a parallel simulator of our general model that will enable extremely large scale simulations in a faster amount of time. We suggest environmentally beneficial swarm applications for future work such as artificial pollination [11] or oil spill cleanup [2]. We also provide a sketch for a potential task allocation algorithm inspired by the centralized solution of min cost flow used to solve the Optimal Transport Problem [52]. We adapt this centralized algorithm into a swarm algorithm to provide a potential third mechanism for task allocation in unknown environments. Our proposed directions highlight areas of future work that can be researched within the framework of our general model.

Overall, this thesis makes two key contributions. First, we propose a new formal general model for swarm algorithms that can easily be adapted to simulation. Then, we contribute to swarm literature by developing novel algorithms for the problems of N-site selection and task allocation in unknown environments. Through developing our house hunting and task allocation algorithms we also show that our modelling framework is reasonable and capable of adequately representing swarm algorithms. We used our model to examine geographic aspects of these problems that have not been considered in the past and showed how these aspects (site placement in house hunting, and task density in task allocation) affect algorithm performance and should be considered. Our results shed light on the impact of environment geometry on the performance of swarm algorithms and we contribute several algorithms which have been evaluated in a more thorough manner geographically.

Chapter Contents The remainder of this thesis is structured as follows. Chapter 2 provides the formal details of our general swarm model, as well as pseudocode for how the model framework was implemented. Chapter 3 presents and evaluates our new N-site selection algorithm and then provides a pseudocode outline of it. Chapter 4 presents our two new task allocation algorithms, HHTA and PROP, which assign robots to complete tasks in initially unknown environments. Chapter 5 discusses future work. Chapter 6 concludes the thesis.

Contributors Chapters 2 is joint work with Nancy Lynch. Nancy Lynch and Grace Cai came up with the general model and wrote Chapter 2 together. Grace implemented and provided the pseudocode for the general model. Chapter 3 is joint work with Nancy Lynch. Nancy and Grace wrote section 3.3.1 together. Grace came up with the N-site selection algorithm in Chapter 3, implemented and conducted experiments on the algorithm, prepared all figures, and wrote the remainder of Chapter 3. Chapter 4 is joint work with Nancy Lynch and Noble Harasha. Nancy and Grace wrote section 4.3.1 together. Grace came up with, implemented, and tested the HHTA algorithm. Grace, Nancy, and Noble came up with the PROP algorithm together. Noble implemented and tested the PROP algorithm. Grace came up with the experiments and environmental setups in Chapter 4. Chapter 5 is written by Grace Cai. Section 5.1 is joint work with Zhi Wei Gan, Nancy Lynch, and Julian Shun.

Chapter 2

A General Discrete Swarm Model

In this chapter, we provide a formal model for simulating a swarm of agents synchronously in two dimensions. We then provide a detailed pseudocode implementation of our framework.

Our modelling framework is designed to be general enough to design a variety of swarm algorithms. We use our framework for the house hunting algorithm presented in Chapter 3 and the task allocation algorithms presented in Chapter 4.

2.1 General Model

We assume a finite set R of agents, with a state set SR of potential states. Agents move on a discrete rectangular grid of size $n \times m$, formally modelled as directed graph $G = (V, E)$ with $|V| = mn$. Edges are bidirectional, and we also include a self-loop at each vertex. Vertices are indexed as (x, y), where $0 \leq x \leq m-1$, $0 \leq y \leq n-1$. Each vertex also has a state set SV of potential states.

We also define *squares* with the same range of indices; square (x, y) is the one whose lower left vertex is (x, y). This correspondence means that we can think of squares instead of vertices if we prefer.

We use a discrete model so the model can be simulated in a distributed fashion on each vertex to reduce computation time. Also, using a discrete model allows algorithms to be modeled as state machines, which enables analysis using techniques

such as invariant assertions.

2.1.1 Model Configurations

We define four kinds of configurations, global vs. local, and ordinary vs. transitory. The transitory configurations are used as intermediate steps in defining system executions.

A global configuration specifies the information and state of entire grid. For each vertex/square in the grid, it specifies a vertex state. For each agent in R, it specifies the agent's location on the grid as well as the agent's state.

Formally, a *global configuration* C is a triple of mappings, $(svmap, srmap, locmap)$, where:

- $svmap : V \to SV$ is the vertex state mapping, which assigns a vertex-state to each vertex,
- $srmap : R \to SR$ is the agent state mapping, which assigns an agent-state to each agent, and
- $locmap : R \to V$ is the location mapping, which assigns a location to each agent.

A visual example of a global configuration with agents wandering an arena with obstacles can be seen in Figure 2-1. In Figure 2-1, 3 agents wander a $M = 6, N = 4$ sized grid. The agent state is simply an agent id $\in \{0, 1, 2\}$, a unique identifier for each agent. The vertex state specifies whether that vertex is an obstacle or not.

A *local configuration* C' is intended to capture the contents of one vertex/square and thus details the vertex's state, the agents located at the vertex, and the states of those agents. Formally, it is a triple $(sv, myagents, srmap)$, where:

- $sv \in SQ$ is the vertex-state of the given vertex,
- $myagents \subseteq R$ is the set of agents at the vertex, and
- $srmap : myagents \to SR$ assigns an agent-state to each agent at the vertex.

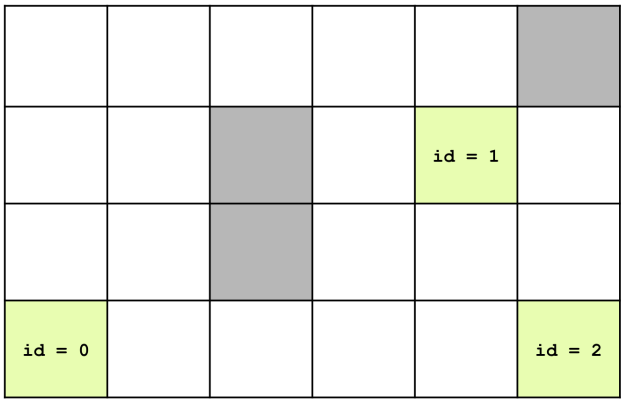

Figure 2-1: A visual representation of a global configuration for agents wandering an arena with obstacles. Here, we use the interpretation of vertices as squares. Each square is grey if it is an obstacle, green if it contains an agent, and white if it is empty.

We also have a notion of transitory configuration, which is used as an intermediate stage between two ordinary configurations, in constructing executions. It represents agents in motion from one vertex to another.

A global transitory configuration, like a global configuration, contains information about the vertex state of each vertex in the grid as well as the agent state of each agent. However, instead of also specifying the location of each agent, it instead specifies for each agent the edge along which it is travelling. For example, agent r travelling along edge (v, v') means that agent r started at vertex/square v and is going to adjacent vertex/square v'.

Formally, a *global transitory configuration* T is a triple of mappings $(svmap, srmap, edgemap)$, where

- $svmap$ and $srmap$ have the same types as for ordinary configurations, and
- $edgemap : R \to E$ assigns a directed edge to each agent.

A local transitory configuration represents newly-computed states for a single vertex and its agents, plus directions of travel for the local agents. Agents at a square can move up (U), down (D), left (L), right (R), or stay (S) at their current vertex.

Formally, a *local transitory configuration* T' is a quadruple $(sv, myagents, srmap, dirmap)$, where

- sv, $myagents$, and $srmap$ are as in the definition of a local configuration, and
- $dirmap : myagents \to \{R, L, U, D, S\}$ is the direction mapping, which assigns a travel direction to each agent. Direction S means to stay at the vertex.

2.1.2 Local transitions

The transition of a vertex v may be influenced by the local configurations of nearby vertices in addition to itself. We define an **influence radius** I, which is the same for all vertices, to mean that vertex indexed at (x, y) is influenced by all vertices $\{(a, b) \mid a \in [x - I, x + I], b \in [y - I, y + I]\}$, where a and b are integers mod n. We can use this influence radius to create a local mapping M_v from local coordinates to the neighboring local configurations. Thus, for a vertex v at location (x, y), we produce M_v such that $M_v(a, b) \to C'(w)$ where w is the vertex located at $(x + a, y + b)$ and $-I < a, b < I$. This influence radius is representative of a sensing and communication radius for agents.

We have a local transition function δ, which maps all the information associated with one vertex and its influence radius at one time to new information that can be associated with the vertex at the following time. It also produces directions of motion for all the agents at the vertex.

Formally, for a vertex v, δ maps the mapping M_v to a probability distribution on local transitory configurations of the form $(sv_1, myagents, srmap_1, dirmap_1)$, where:

- $sv_1 \in SV$ is the new state of the vertex,
- $srmap_1 : myagents \to SR$ is the new agent state mapping, for agents currently at the vertex, and
- $dirmap_1 : myagents \to \{R, L, U, D, S\}$ gives directions of motion for all the agents currently at the vertex.

2.1.3 Local transition function δ

The local transition function δ is further broken down into two phases as follows.

Phase One: Each agent in vertex v uses the same probabilistic transition function α, which maps the agent's state $sr \in SR$, location (x, y), the vertex state of the location $sv \in SV$, and the mapping M_v to a distribution over new proposed vertex state sv', agent state sr', and direction of motion $d \in \{R, L, U, D, S\}$.

Phase Two: A resolution rule L is used to reconcile the different vertex states suggested by each agent at the vertex and select one final vertex state. The rule also determines for each agent whether they may transition to state sr' and direction of motion d or stay at the same location with original state sr.

Formally, L takes in the mappings $(svprop, srprop, dirprop)$ and the local configuration $C'(v)$ of vertex v, where:

- $svprop : myagents \to SV$ is a mapping from agents at vertex v to the vertex state sv' proposed in Phase One
- $srprop : myagents \to SR$ is the agent state mapping from agents at v to the agent state sr' they proposed in Phase One
- $dirprop : myagents \to \{R, L, U, D, S\}$ is the direction mapping from agents at v to the direction of travel d they proposed in Phase One

The rule L uses these mappings and the local vertex configuration to produce the output $(sv_1, srmap_1, dirmap_1)$, which is the output of δ. Specifically, L is used to decide on one final new vertex state $sv_1 \in SV$ and modify the proposed agent states and directions mappings if necessary, producing $srmap_1$ and $dirmap_1$.

2.1.4 Probabilistic execution

The system operates by probabilistically transitioning all vertices v for an infinite number of rounds. During each round, for each vertex v, we obtain the mapping M_v which contains the local configurations of all vertices in its influence radius. We then apply δ to M_v to transition vertex v and all agents at vertex v and sample the resulting distribution to select a local transitory configuration for v. For each vertex v we now have $(sv_v, myagents_v, srmap_v, dirmap_v)$ returned from δ.

For each v, we take $dirmap_v$, which specifies the direction of motion for each agent and use it to map all agents to their new vertices. For each vertex v, its new local configuration is just the new vertex state sv_v, the new set of agents at the vertex, and the *srmap* mapping from agents to their new agent states.

2.2 Model Capabilities

We would like to highlight some useful capabilities that our framework allows for. For example, our model allows for the use of multiple different types of agents, as shown by the PROP algorithm presented in Chapter 4. The PROP algorithm uses two types of agents – propagators, which are simple, mote-inspired agents [49], and followers, which are more complex, task-performing agents. This distinction between different types of agents can either be incorporated into the agent state as a variable, or by easily extending the framework's singular agent set R and agent state set SR to multiple agent sets $R_1, R_2, \ldots$ and agent state sets $SR_1, SR_2, \ldots$ corresponding to the different types of agents.

By using a torus grid, our model also allows for simulations using a rectangular grid as a special case. Let us fix the lower left corner of the grid at vertex $(0, 0)$ and the upper right corner of the grid at vertex $(m-1, n-1)$. If an algorithm designed within our model seeks to use such a rectangular grid for more realistic edge effects, it needs to follow two extra restrictions. Firstly, in generating the agent's direction of motion, agents assuming a grid environment will ensure the direction of motion they are generating is never out-of-bounds of the grid, meaning it never moves between vertices $(0, y)$ and $(m-1, y)$ or between vertices $(x, 0)$ and $(x, n-1)$. Secondly, agents should ignore all information in their influence radius mapping M_v which maps to an "out of bounds" vertex that the agent wouldn't be able to see in a grid.

Our model also allows for the simulation of messages. Since agents have access to all information about other agents within their influence radius when transitioning, a message occurs whenever an agent looks at the state information of another agent, which can be interpreted as receiving a message from that agent. Therefore, the

influence radius not only functions as a sensing radius, but also a communication radius. This makes sense under the assumption that agents must have sensed each other to be able to message each other.

Our model allows for simulating agents without communication abilities as a special case. Such agents would be restricted by not looking at any of the agent state information included in the influence radius mapping M_v, and only using the location of the sensed agents to make decisions. Similarly, our model can simulate agents with no sensing or communication abilities by having agents ignore all information within their influence radii.

The use of a problem-specific resolution rule L allows for a dynamic handling of the conflicts that arise from all agents at a vertex synchronously transitioning. For example, in the task allocation problem, L would need to handle the conflict of more agents trying to claim a task than necessary by picking some winner agents and turning away the rest. This functionality can be seen in detail in Section 4.8.5. An example with a different resolution functionality would be problems which simulate pheromones, in which L would make sure that if multiple agents at a vertex deposited pheromones, all of the pheromone information would be included in the new vertex state.

Lastly, using our discrete model rather than a continuous version brings some additional benefits. Our discrete model has a direct translation to simulation, whereas continuous models require time to be discretized. Furthermore, our model enables discrete analysis, since agents within our model are state machines and the entire environment, with all vertices and agents, could be analyzed as a state machine as well. While this analysis could be very difficult, especially with more complex swarm algorithms, it provides an alternative to the differential equations based analysis that continuous models use.

On the other hand, continuous models have their own merits, such as providing more realism than discrete models. For this reason we would like to mention that there is a direct translation of algorithms that run using our model into a 2D continuous space. This can be achieved by discretizing the continuous space and running the

algorithm as if a grid were present. If we use a very fine-grained discretization then our model is a good approximation of the 2D continuous space.

2.3 Useful Extensions

We would like to propose a few extensions of our model that could be useful in future work. One of the most useful would be to extend our model to use a 3D grid space, where we have influence balls instead of influence radii, and agents can move in any one of 6 directions. This would allow more a more realistic simulation of aerial swarms. A further generalization would be to extend our model to use any general directed graph. Here agents would be able to move along any outgoing edge from the vertex they are at, and an influence radius of x could encompass all nodes that are up to x hops away.

Another extension of our model would be to develop a collision-avoidant version of it, where only one agent can occupy a vertex at any given time. This is useful for problems such as multi-agent pathfinding in warehouse environments [33], where the goal is for agents to avoid colliding paths. In this extension, instead of a resolution rule which prevents multiple agents from making conflicting changes to the same vertex, we would need a collision avoidance rule. This collision avoidance rule should look at all proposed agent directions of motion, and if multiple agents are trying to move to the same vertex v', only one should be allowed to make the move and the remaining agents should stay where they are. The collision avoidance rule is different than the resolution rule because the resolution rule looks at conflicts within agents' current vertex v but the collision avoidance rule looks to prevent conflicts (collision) at agents' new vertex v'. Note that no resolution rule is necessary in this extension since only one agent is at any vertex at any given time, so there are no conflicts that arise from two agents trying to modify the same vertex at once.

2.4 Pseudocode Framework

We now describe our pseudocode framework for implementing our model. We use the classes listed below, each with their own functions and variables, in the implementation of our swarm model.

- Agent, representing the agents in the formal model
- Agent State, representing the state set SR for agents in the formal model
- Configuration, representing the global configuration of agents, vertices, and agent locations within vertices
- Vertex, representing a singular vertex in the grid (a local configuration in the formal model)
- Vertex State, representing the state set SV for vertices in the formal model

2.4.1 Vertex Class

Each vertex in the grid is represented by an instance of the `Vertex` class. The vertex class represents local configurations in the formal model. A `Vertex` has four variables:

- `x`, the x coordinate of the vertex
- `y`, the y coordinate of the vertex
- `state`, the vertex state of the vertex
- `agents`, a list of agents located at the vertex, where each agent is an instance of the Agent class defined below

2.4.2 Vertex State Class

An instance of the vertex state class represents a vertex state in the set SV that an agent can take on in the formal model. The set of possible instances of the vertex

state class forms the set SV in the formal model. The specific variables used in the vertex state depend on the task the swarm is performing.

For example, consider a simple toy example where agents are doing a random walk on the grid, and each vertex stores a count of which agents have landed on it. In that case, the vertex state would consist of the following variable:

- `visited`, a list of agent ids who have visited the vertex

2.4.3 Agent State Class

An instance of the agent state class represents the internal state of an agent and the set of possible instances of the agent state class form the agent state set SR in the formal model. Agent states can store two types of variables – constant variables, and modifiable ones. Constant variables are fixed in value and are the same for all agents. They represent shared knowledge that the agents have already. Three constant variables that all agents in the formal model have are:

- `INFLUENCE_RADIUS`, the influence radius
- `N`, the height of the grid
- `M`, the width of the grid

More constant variables can be added depending on the specific problem. All agents also store an `agent_id` to differentiate them from each other. This number is unique to each agent and cannot be modified:

- `id`, a unique integer in the range $[0, |R| - 1]$ (where R is the agent set)

Lastly, modifiable variables are used to store extra, changeable state for specific problems. Consider our toy example of agents doing a random walk, where we count which agents have been in which squares. Suppose the agents do a simple random walk, in which they walk in one direction (up, down, left, or right) for a fixed number of steps before changing directions. In this case, the agent state would also need to include the following variables:

- `step_length`, a constant variable representing the number of steps an agent taks before changing direction

- `current_direction`, a modifiable variable taking on values in the set $\{U, D, L, R\}$ representing which direction the agent is currently travelling in

- `steps_taken`, a modifiable variable representing how many steps the agent has taken so far in their current direction of travel

2.4.4 Agent Class

Each agent in our agent set R is implemented as an instance of the `Agent` class. An `Agent` has two variables – `location`, the `Vertex` class corresponding to the agent's location, and `state`, the `Agent State` class describing the Agent's State.

Every agent also has the function `generate_transition(local_vertex_mapping)`, which corresponds in the formal model to sampling from the probabilistic transition function α to generate a change in agent location and state each round of the simulation. Note that `is a local_vertex_mapping` is a map from local coordinates (dx, dy) within the agent's influence radius to instances of the Vertex class (local configurations). The local coordinates are translated such that (0,0) maps to the Agent's current location vertex. The code for `generate_transition(local_vertex_mapping)` is problem specific and represents the bulk of the agent's decision making logic in each time step.

Let us consider our toy example again, where agents are doing a random walk and the vertices are storing which agents have been on them. The transition function for this problem would look like Algorithm 1.

In this problem, each agent proposes changing the agent state to include its own id if it hasn't been added already. Each agent then proceeds to generate one step of their random walk. If they have traveled far enough in the current direction, a new direction of travel is generated and their travel distance counter (`steps_taken`) is reset. Otherwise, they continue on their random walk leg.

Algorithm 1 Agent Transition for an Agent a in the Random Walk Example

```
1: procedure GENERATE_TRANSITION(local_vertex_mapping))
2:     s ← a.state
3:     new_agent_state ← a.state
4:     new_vertex_state ← a.location.state
5:     ▷Propose adding the agent's own id to the vertex it just visited
6:     new_vertex_state.visited ← union(new_vertex_state.visited, s.id)
7:     ▷Generate new direction for random walk if we have reached the step length
8:     if s.steps_taken == s.step_length then
9:         new_dir ← random_choice([L, R, D, U])
10:        new_agent_state.current_direction ← new_dir
11:        new_agent_state.steps_taken ← 1
12:    return new_vertex_state, new_agent_state, new_dir
```

2.4.5 Configuration

A configuration represents the global configuration from our formal model and encompasses all agents and vertices within it. Specifically, a configuration has the following variables:

- `N`, the height of the grid
- `M`, the width of the grid
- `vertices`, a map of all of the N*M vertices in the grid, from coordinates in the set $[0, M-1] \times [0, N-1]$ to instances of the Vertex class
- `INFLUENCE_RADIUS`, the influence radius of each agent in the configuration (which matches the `INFLUENCE_RADIUS` variable in the agent state)
- `agents`, a map from agent id to instances of the Agent class

The configuration class also has a `transition()` function (Algorithm 2), which represents stepping forward once in time. This function utilizes two helper functions and is defined as follows:

The function `generate_global_transitory()` (Algorithm 3) generates and returns the global transitory configuration, and the function `execute_transition(global_transitory)` simply executes the updates to

Algorithm 2 Transition function for a configuration `C`

```
1: procedure TRANSITION
2:     global_transitory ← C.generate_global_transitory()
3:     C.execute_transition(global_transitory)
```

the agent locations and states based upon the transitory configuration. The global transitory configuration is a map from coordinates to a triple consisting of the new vertex state v', a mapping from agent id to new proposed agent states for agents currently at v, and a mapping from agent id to new proposed movement directions (of the set {U, D, L, R, S}) for that vertex.

The `generate_global_transitory()` function is further implemented as follows:

Algorithm 3 Generating the global transitory configuration for a configuration `C`

```
1: procedure GENERATE_GLOBAL_TRANSITORY
2:     global_transitory ← {}
3:     for x in 1 ...C.M do
4:         for  y in 1 ...C.N do
5:             local_vertex_mapping ← generate_local_mapping(
6:                 C.vertices[(x, y)], C.INFLUENCE_RADIUS, C.vertices)
7:             global_transitory[(x, y)] ← C.delta(local_vertex_mapping)
8:     return global_transitory
```

For each vertex in the global configuration, we are generating the local vertex mapping (of all vertices within the influence radius) and using it to compute the local transitory configuration for that vertex. We then merge all of these local transitory configurations into the global transitory configuration. The function `generate_local_mapping` takes in a vertex v, influence radius, and global vertex set in order to return a map from local coordinates (dx, dy) within the influence radius centered around v to the corresponding instances of the Vertex class found inside the influence radius. Then, the `delta` function is applied to this local vertex mapping to get the local transitory configuration for that vertex v. All of the local transitory configurations are merged into the global transitory configuration.

The function `delta` (Algorithm 4) corresponds to the δ map in our formal model, taking in the local vertex mapping and returning a new proposed vertex state as well

as maps from agent id to new agent state and movement direction. The delta function is where agents attempt to transition states and modify their environment, and any conflicts between their actions are resolved.

Algorithm 4 δ for a configuration `C`

```
1: procedure DELTA(local_vertex_mapping)
2:     v ← local_vertex_mapping[(0,0)]
3:     if length(v.agents) == 0 then return v.state, {}, {}
4:     proposed_vertex_states ← {}
5:     proposed_agent_states ← {}
6:     proposed_agent_dirs ← {}
7:     for agent in v.agents do
8:         proposed_vertex_state, proposed_agent_state, direction =
9:             agent.generate_transition(local_vertex_mapping)
10:        proposed_vertex_states[agent.id] ← proposed_vertex_state
11:        proposed_agent_states[agent.id] ← proposed_agent_state
12:        proposed_agent_dirs[agent.id] ← direction
13:    return resolution_rule(
14:        proposed_vertex_states, proposed_agent_states, proposed_agent_dirs)
```

In lines 2-3, we get the vertex that this local vertex mapping is centered around. If this vertex does not have any agents inside of it, then the vertex state cannot change in the house hunting problem, so we return the vertex's current state as the proposed new state. We return empty maps for the agent state and direction update since no agents are present. In lines 4-12, we have each agent inside the vertex propose a new vertex state, their own new agent state, and their direction of motion, and store these pieces of information. This corresponds to phase 1 of the formal model, where each agent samples from their α distribution, which proposes agent transitions at each time step.

In line 13, we execute Phase 2 of the formal model, which resolves potential conflict between the vertex states and agents states that agents propose. We do so via a `resolution_rule`, which takes in all the proposed vertex states, agent states, and agent directions. The resolution rule decides what the final new vertex state is by combining the proposed vertex states, and also decides which agent states and agent directions are allowed to transition. If an agent is not allowed to transition, it

remains in the same state it was in before and stays where it is.

Let us examine the resolution rule for our random walk toy example. In this example, each agent proposes to add itself to the list of agents stored by the vertex it is currently on. However, if multiple agents try to add themselves to the same vertex's list, we can resolve their conflicting vertex states via a resolution rule such as Algorithm 5.

Algorithm 5 Random walk example resolution rule

```
1: procedure RESOLUTION_RULE(proposed_vertex_states, proposed_agent_states, proposed_agent_dirs))
2:     new_vs ← VertexState()
3:     for proposed_vs in proposed_vertex_states.values() do
4:         new_vs.visited ← union(proposed_vs.visited, new_vs.visited)
5:     new_agent_states ← proposed_agent_states
6:     new_agent_dirs ← proposed_agent_dirs
7:     return new_vs, new_agent_states, new_agent_dirs
```

In this example resolution rule, we take a union of all of the suggested new vertex states, which only add individual agents, in order to add all new agents to the list of agents who have visited the vertex. We then allow all agents to transition states and directions, since in this example all agents proposed vertex states have been accepted into the final new vertex state. Note that this is not always the case for other problems. For example, in a task allocation problem where multiple agents attempt to claim a single-agent task and transition into doing the task, only one agent should succeed and be allowed to transition.

2.4.6 Summary

We have described the implementation of the general model; the backbone which all swarm algorithms run in our model share. We have also detailed which parts of the implementation are problem-specific. Namely, where different problems differ in the model implementation is in the Vertex State variables, the Agent State variables, the agent transition function for transitions at each time step, and the resolution rule. This implementation cleanly divides out the problem-specific implementations.

Chapter 3

A Geometry-Sensitive N-Site Selection Algorithm

The house hunting behavior of the Temnothorax albipennis ant allows the colony to explore several nest choices and agree on the best one. Their behavior serves as the basis for many bio-inspired swarm models to solve the same problem. However, many of the existing site selection models in both insect colony and swarm literature test the model's accuracy and decision time only on setups where all potential site choices are equidistant from the swarm's starting location [42, 44, 13, 32]. These models do not account for the geographic challenges that result from site choices with different geometry. For example, although actual ant colonies are capable of consistently choosing a higher quality, further site instead of a lower quality, closer site [19], existing models are much less accurate in this scenario. Existing models are also more prone to committing to a low quality site if it is on the path between the agents' starting site and a higher quality site. We present a new model for the site selection problem and verify via simulation that is able to better handle these geographic challenges. Our results provide insight into the types of challenges site selection models face when distance is taken into account. Our work will allow swarms to be robust to more realistic situations where sites could be distributed in the environment in many different ways.

3.1 Introduction

Swarms of birds, bees, and ants are able to coordinate themselves to make decisions using only local interactions [9, 41, 46]. Modelling these natural swarms has inspired many successful swarm algorithms [18]. One such bio-inspired algorithm comes from the house hunting behavior of ants. Models of the ants' behavior when selecting a new nest serve as the basis for swarm algorithms which seek to select the best site out of a discrete number of candidate sites in space [44].

Many variations of the best-of-N site selection problem have been studied for swarms [53]. For example, when sites are of equal quality, choosing one is a symmetry-breaking problem [24, 55]. Situations with asymmetric site qualities and costs (where higher quality sites have a higher cost of being chosen) have also been studied – for example, when one of two candidate sites is significantly larger than the other (making it harder for agents to detect other agents favoring the larger site, even when it is of higher quality) [10].

However, most site selection models are mainly tested on small numbers of candidate nest sites that are equidistant from the agents' starting location (also known as the home nest) [13, 42]. In many applications of the site selection problem such as shelter seeking, sites will not be distributed so uniformly.

This equidistant setup fails to capture two important geographical details that existing algorithms struggle with in making accurate decisions. Firstly, nests that are closer to the home nest are advantaged because they are more likely to be found. Even so, house hunting ants can still choose higher quality sites that are much further than lower quality, closer sites. We have found that existing site selection models often commit to the closer site even when there is a better, further option. Secondly, using sites equidistant from the home nest eliminates the possibility of some nests being in the way of others. Site selection models often trigger consensus on a new site after a certain quorum population of agents have been detected in it. If a low quality nest is on the path from the home nest to a high quality nest, agents travelling between the home nest and the high quality nest could saturate the path and detect a quorum for

the lower quality nest that is in the way instead of the highest quality nest.

This paper aims to create a new algorithm that can successfully account for a more varied range of nest distributions, allowing agents to successfully choose higher quality nests even when they have the disadvantage of being further from the agents' starting location or there are other lower quality nests in the way. The model should also perform with similar accuracy compared to existing models on the default setup with equidistant candidate nest sites. We show via simulation that incorporating a quorum threshold that decreases with site quality allows for increased accuracy compared to previous models. We also show that setups where candidate sites are in the way of each other or are of similar quality can make it harder for site selection models to produce accurate results.

Section 3.2 describes the house hunting process of ants and overviews existing swarm models. Section 3.3 describes our model. We provide details on the implementation of our model, test accuracy and decision time in different geographic situations, and report the results in Section 3.4. We discuss these results in Section 3.5. Lastly, we suggest future work in Section 3.6. The full simulation code can be found at [6].

3.2 Background

3.2.1 Ant House Hunting

When the *T. albipennis* ants' home nest is destroyed, the colony can find and collectively move to a new, high quality nest. To do so, *T. albipennis* scouts first scan the area, searching for candidate nests. When a nest is found, the scouts wait a period of time inversely proportional to the nest quality before returning to the home nest. There, they recruit others to examine the new site in a process known as forward tandem running. Tandem runs allow more ants to learn the path to a new site in case the ants decide to move there. When an ant in a candidate nest encounters others in the site at a rate surpassing a threshold rate (known as the quorum threshold),

ants switch their behavior to carrying other members of the colony to the new nest. Carrying is three times faster than tandem runs and accelerates the move to the new nest [40, 41].

This decision-making process allows ants to not only agree on a new nest, but also to choose the highest quality nest out of multiple nests in the environment. This is true even if the high quality nest is much further from the home nest than the low quality nest [19, 47]. Franks [19] found that with a low quality nest 30 cm from home and a high quality nest 255cm from home, 88% of ant colonies successfully chose the high quality nest even though it was 9 times further.

3.2.2 House Hunting and Site Selection Models

To better study the ants' behavior, models have been designed to simulate how ants change behavior throughout the house hunting process [42, 58]. These models, initiated by Pratt [42], allow simulated ants to probabilistically transition through four phases – the Exploration, Assessment, Canvassing, and Transport phases. The Exploration represents when the ants are still exploring their environment for new sites. When an ant discovers a site, it enters the Assessment phase, in which it examines the quality of the site and determine whether to accept or reject it. If the ant accepts the site, it enters the Canvassing phase, which represents the process of recruiting other ants via forward tandem runs. Finally, if a quorum is sensed, the ant enters the Transport phase, which represents the carrying behavior used to move the colony to the new site.

These models, however, assume that when an ant transitions from the Exploration phase to the Assessment phase, it is equally likely to choose any of the candidate sites to assess. This assumes that any nest is equally likely to be found, which is unlikely in the real world because sites closer to the home nest are more likely to be discovered. To our knowledge, house hunting models have not tried to model situations where nests have different likelihoods of being found, as is the case when nests have different distances from the home nest [58].

The corresponding problem to house hunting in robot swarms is known as the

N-site selection problem [53]. Agents, starting at a central home site, must find and choose among N candidate sites in the environment and move to the site with highest quality. (The home and candidate sites are also referred to as nests, since the problem was inspired by house-hunting ants.) Unlike house hunting models, which do not physically simulate ants in space, swarm models set up agents in a simulated arena and let them physically explore sites and travel between them.

Inspired by ant modelling, [13] and [44] have modeled swarm agents using four main states – Uncommitted Latent, Uncommitted Interactive, Favoring Latent, and Favoring Interactive (with [13] adding a fifth Committed state to emulate having detected a quorum). Uncommitted Latent agents remain in the home site while Uncommitted Interactive agents explore the arena for candidate sites. Favoring Interactive agents have discovered and are favoring a certain site and recruit other agents to the site, while Favoring Latent agents remain in the new site to try and build up quorum. Agents probabilistically transition between these states based on environmental events (e.g. the discovery of a new site) and eventually end up significantly favoring a new candidate site or committed to it. Other swarm models for N-site selection typically use a similar progression through uncommitted, favoring, and committed type phases [37].

One setup where a high quality site was twice as far as a low quality one was successfully solved in [44], but for the most part these models and their variations have mainly been tested in arenas with two candidate sites equidistant from the home site [7, 13, 32, 43]. Our model aims to analyze the behavior of these models in more varied site setups and and improve upon them.

3.3 Model

We first summarize our discrete geographical model (presented in Chapter 2) for modeling swarms. Then we discuss the individual restrictions, parameters, and agent algorithms needed for the house hunting problem specifically.

The pseudocode for our general model can be found in Section 2.4. A pseudocode

description of our N-site selection algorithm can be found in Section 3.7.

3.3.1 General Model

The general model we present in this section is an abbreviated description of our model in Chapter 2. For more formal details, please see Chapter 2.

We use the special case of a rectangular grid instead of a torus. The special case works by having agents treat the graph as a grid by never generating an out-of-bounds direction of motion and by never using any vertices in their influence radius that they wouldn't be able to see on a grid due to edge effects. For more details on this special case, please see Section 2.2.

We assume a finite set R of agents, with a state set SR of potential states. Agents move on a discrete rectangular grid of size $n \times m$, formally modelled as directed graph $G = (V, E)$ with $|V| = mn$. Edges are bidirectional, and we also include a self-loop at each vertex. Vertices are indexed as (x, y), where $0 \leq x \leq n-1$, $0 \leq y \leq m-1$. Each vertex also has a state set SV of potential states.

We use a discrete model so the model can be simulated in a distributed fashion on each vertex to reduce computation time and provide the possibility of discrete analysis via state machine methods.

Local Configurations:

A *local configuration* $C'(v)$ captures the contents vertex v. It is a triple $(sv, myagents, srmap)$, where $sv \in SV$ is the vertex state of v, $myagents \subseteq R$ is the set of agents at v, and $srmap : myagents \to SR$ assigns an agent state to each agent at v.

Local Transitions:

The transition of a vertex v may be influenced by the local configurations of nearby vertices. We define an **influence radius** I, which is the same for all vertices, to mean that vertex indexed at (x, y) is influenced by all valid vertices $\{(a, b) | a \in [x -$

$I, x + I], b \in [y - I, y + I]\}$, where a and b are integers. We can use this influence radius to create a local mapping M_v from local coordinates to the neighboring local configurations. For a vertex v at location (x, y), we produce M_v such that $M_v(a, b) \rightarrow C'(w)$ where w is the vertex located at $(x+a, y+b)$ and $-I < a, b < I$. This influence radius is representative of a sensing and communication radius. Agents can use all information from vertices within the influence radius to make decisions.

We have a local transition function δ, which maps all the information associated with one vertex and its influence radius at one time to new information that can be associated with the vertex at the following time. It also produces directions of motion for all the agents at the vertex.

Formally, for a vertex v, δ probabilistically maps M_v to a quadruple of the form $(sv_1, myagents, srmap_1, dirmap_1)$, where $sv_1 \in SV$ is the new state of the vertex, $srmap_1 : myagents \rightarrow SR$ is the new agent state mapping for agents at the vertex, and $dirmap_1 : myagents \rightarrow \{R, L, U, D, S\}$ gives directions of motion for agents currently at the vertex. Note that R, L, U, and D mean right, left, up, and down respectively, and S means to stay at the vertex. The local transition function δ is further broken down into two phases as follows.

Phase One: Each agent in vertex v uses the same probabilistic transition function α, which probabilistically maps the agent's state $sr \in SR$, location (x, y), and the mapping M_v to a new suggested vertex state sv', agent state sr', and direction of motion $d \in \{R, L, U, D, S\}$. We can think of α as an agent state machine model.

Phase Two: Since agents may suggest conflicting new vertex states, a rule L is used to select one final vertex state. The rule also determines for each agent whether they may transition to state sr' and direction of motion d or whether they must stay at the same location with original state sr.

Probabilistic Execution:

The system operates by probablistically transitioning all vertices v for an infinite number of rounds. During each round, for each vertex v, we obtain the mapping M_v which contains the local configurations of all vertices in its influence radius. We then

apply δ to M_v to transition vertex v and all agents at vertex v. For each vertex v we now have $(sv_v, myagents_v, srmap_v, dirmap_v)$ returned from δ.

For each v, we take $dirmap_v$, which specifies the direction of motion for each agent and use it to map all agents to their new vertices. For each vertex v, it's new local configuration is just the new vertex state sv_v, the new set of agents at the vertex, and the *srmap* mapping from agents to their new agent states.

3.3.2 House Hunting Environment Model

The goal of the house hunting problem is for agents to explore the grid and select the best site out of N sites to migrate to collectively. We model sites as follows.

A set S, $|S| = N$ of rectangular sites are located within this grid, where site s_i has lower left vertex (x_i^1, y_i^1) and upper right vertex (x_i^2, y_i^2). Each site s_i also has a quality $s_i.q \in [0, 1]$. To represent these sites, we let the vertex state set be $SV = S \cup \{\emptyset\}$ for each vertex, indicating which site, if any, the vertex belongs to. Furthermore, we denote the site s_0 to be the *home nest*. In the initial configuration, all agents start out at a random vertex in the home nest, chosen uniformly from among the vertices in that nest.

3.3.3 Agent States and Transition Function

The agent state set SR is best described in conjunction with the agent transition function α. Agents can take on one of 6 core states, each a combination of one of three preference states (Uncommitted, Favoring, Committed), and two activity states (Nest, Active). The state model can be seen in Figure 3-1.

Uncommitted Nest (U^N) agents stay in the home nest to prevent too many agents from flooding the environment. They have a chance of transitioning to Uncommitted Active (U^A) agents, which try to explore the arena and discover new sites. U^A agents move according to the Levy flight random walk, which has been shown to be used by foraging ants [51]. U^N agents transition to U^A with probability P_A, and U^A agents transition to U^N agents with probability P_N. This results in an expected $x = \frac{P_A}{P_A+P_N}$

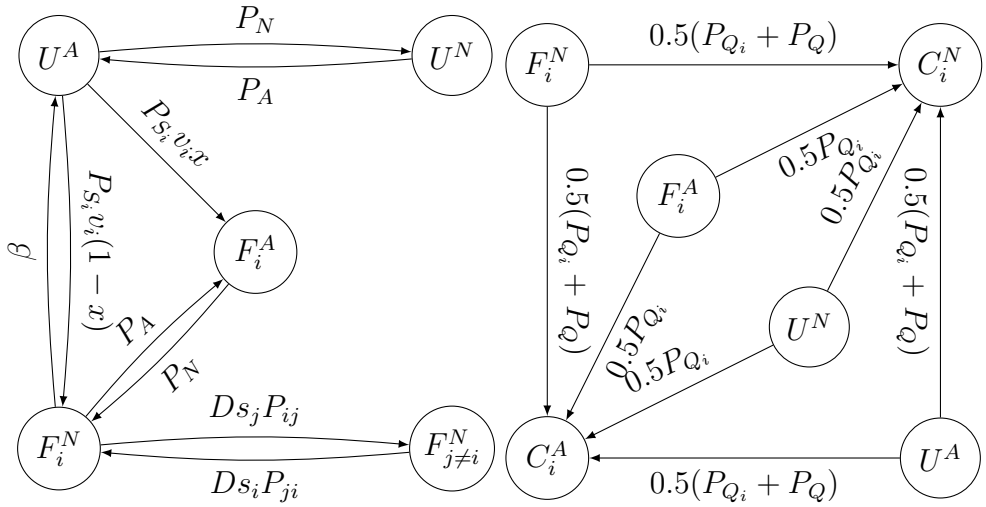

Figure 3-1: State model. $\{U, F, C\}$ denote preference states. The superscript $\{N, A\}$ denotes the activity state, and a subscript i denotes that an agent is favoring or committed to site i. The transitions for Uncommitted and Favoring states are shown on the left, and transitions from Uncommitted and Favoring to Committed states are on the right.

percent of uncommitted agents are active, whereas $1 - x$ agents remain in the nest. Prior work [44] lets $P_N = 9P_A = L$, where L is the inverse of the average site round trip time, chosen to promote sufficient mixing. This leads to 10% of the agent population being active.

Uncommitted Active agents have a chance P_{S_i} of discovering a new nest, which is 1 if a new nest is within influence radius and 0 otherwise. If they discover a nest s_i, they explore and accept it with probability $s_i.q$ (the quality of s_i). They then have an x% chance of transitioning to Favoring Active, and a $(1-x)$% chance of transitioning to Favoring Nest.

Favoring agents (F_i^A, F_i^N) prefer the site s_i that they discovered. Favoring Active (F_i^A) agents remain in site s_i to build quorum. Favoring Nest (F_i^N) agents return to the home nest to recruit others to site s_i. Favoring Nest agents transition to Active with the same probability P_A and Favoring Active agents transition back to Nest agents with probability P_N, creating the same effect where an expected 90% of the favoring agent population is F_i^A while the rest are F_i^N.

F_i^N agents have a probability β of abandoning their nest, which is 1 if the time

spent without seeing other agents surpasses t_β. F_i^A agents can be inhibited by other F_i^A agents as follows. The chance an agent favoring nest i is converted to favoring nest j is Dr_jP_{ij}, where the factor of D is the probability of agents messaging each other (to prevent excessive messaging). r_j is the number of agents favoring s_j that have the agent within their influence radius. After an agent hears of the new site s_j, it visits the site to evaluate $s_j.q$ and changes its preference to s_j if $s_j.q > s_i.q$. Thus, the condition P_{ij} is 1 when $s_j.q > s_i.q$ and 0 otherwise.

U^A agents and F_i^N agents can detect a quorum and commit to a site when q agents in the site are within their influence radius. The quorum size scales with site value as $q = \lfloor (q_{MIN} - q_{MAX}) * qual(i) + q_{MAX} \rfloor$, where q_{MAX} and q_{MIN} are the maximum and minimum possible quorum threshold respectively. The condition P_Q is 1 when quorum is satisfied and 0 otherwise. Agents in any Favoring or Uncommitted state will transition to the committed state, if they encounter an agent already in quorum. The condition P_{Q_i} is 1 when another quorum agent for s_i is encountered and 0 otherwise. Furthermore, agents have an $\frac{1}{2}$ chance of transitioning to Committed Active (C_i^A) and a $\frac{1}{2}\%$ chance of Committed Nest C_i^N after having detected or been notified of a quorum.

C_i^N agents head to the home nest to inform others of the move, while C_i^A agents randomly wander the grid to find stragglers. Agents in quorum states continue to wander until they have sensed quorum for t_Q time steps, whereupon they return to the new selected site s_i.

The resulting agent state set SR is a product of the 6 core states needed in the state model as well as a number of auxiliary variables such as an agent's destination, the names of the sites it favors or has sensed quorum for, and parameters for an agent's random walk when exploring the grid.

Since in the house hunting problem (unlike other problems like task allocation), an agent never modifies the environment, an agent's proposed new vertex state is always the same as the old vertex state. Therefore, phase two of δ is not needed to reconcile conflicting vertex state suggestions from agents.

The transition function α, which for each agent returns a proposed new vertex

state sv', agent state sr' and direction of motion works as follows. The agent never modifies the grid, so $sv' = sv$. The agent state sr' and direction d are calculated according to the core transitions and the auxiliary variables needed to keep track of those transitions. For example, when an agent is headed towards a site, the direction d is calculated to be the next step towards the site. When an agent is staying within a site, the direction d is calculated to be a random walk within the site boundaries.

The total set of variables parameters is $\{P_A, P_N, D, t_Q, t_\beta, q_{MIN}, q_{MAX}\}$, as well as the site locations $(x_i^1, y_i^1), (x_i^2, y_i^2)$ and quality $s_i.q$. In Section 3.4, we explore how changes in q_{MIN}, q_{MAX}, and the site locations and quality impact the accuracy, decision time, and split decisions made by the model.

3.4 Results

The model was tested in simulation using Pygame, with each grid square representing 1cm^2. Agents moved at 1 cm/s, with one round representing one second. We chose this speed because even the lowest cost robots are still able to move at 1cm/s [48]. Agents had an influence radius of 2. All simulations were run using 100 agents, and a messaging rate of 1/15. We let the abandonment timeout $t_\beta = \frac{5}{L}$ and the quorum timeout $t_Q = \frac{1}{L}$.

For each set of trials, we evaluated accuracy (the fraction of agents who chose the highest quality nest), decision time (the time it took for all agents to arrive at the nest they committed to), and split decisions (the number of trials where not all agents committed to the same nest).

3.4.1 Further Nest of Higher Quality

House hunting ants are capable of choosing further, higher quality sites over closer, lower quality ones [19]. When the far site and the near site are of equal value, ants consistently choose the closer one. To test our model's ability to produce the same behavior, we replicated the experimental setups in [19].

Three different distance comparisons were tested, with a further, higher quality

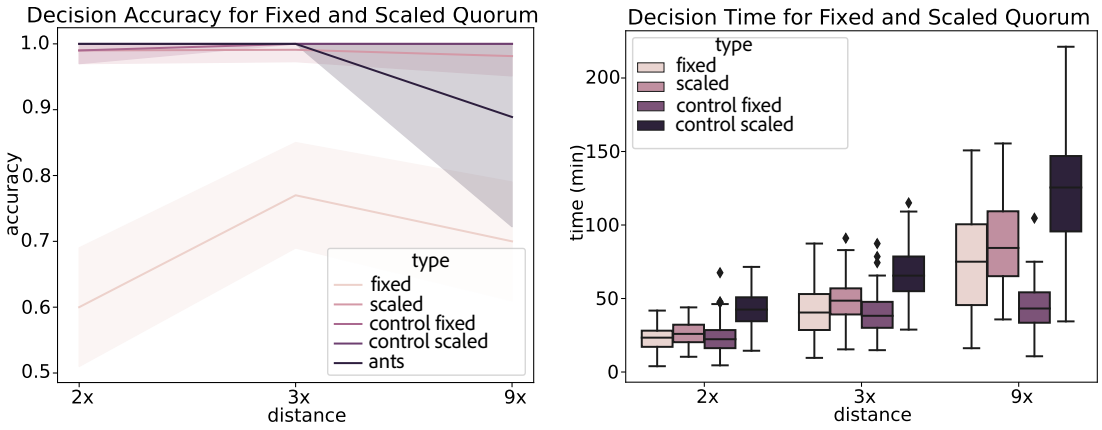

Figure 3-2: Decision Time and Accuracy for far nests 2, 3, and 9 times as far from the home nest. Fixed quorum indicates the fixed threshold value of 4, and scaled quorum indicates $q_{MIN} = 4, q_{MAX} = 7$. The accuracy for the actual ants is taken from [19].

nest of quality 0.9 being 2x, 3x, and 9x as far as a lower quality nest of quality 0.3 on the path from the high quality nest to the home nest. We included a control setup for each of these distance comparisons where both the far and close nest were quality 0.3. The arena size was $N = 16, M = 80$ for the 2x case, $N = 18, M = 180$ for the 3x case, and $N = 18$, $M = 300$ for the 9x case.

We tested our model using two different quorum parameters. In one test, we had $q_{MIN} = q_{MAX} = 4$, intended to represent the behavior of previous models with a fixed quorum threshold. In the other setup, $q_{MIN} = 4$ and $q_{MAX} = 7$, allowing our model to use the new feature of scaling the quorum threshold with site quality. We ran 100 trials for each set of parameters.

As seen in Figure 3-2, using a scaled threshold significantly improved accuracy from using a fixed one. In the control case, both the fixed and scaled quorum threshold achieved high accuracy, with all accuracies being greater than 99%. In cases where the far site was of higher quality, the decision time for fixed and scaled quorum was comparable. However, the scaled quorum threshold took significantly (Welch's T-test, p=0.05) more time in the control case to decide.

Furthermore, as seen in Figure 3-2, our model successfully chose the further site with comparable (or significantly higher in the 9x case) accuracy than ants themselves, indicating that our model is on par with the ants.

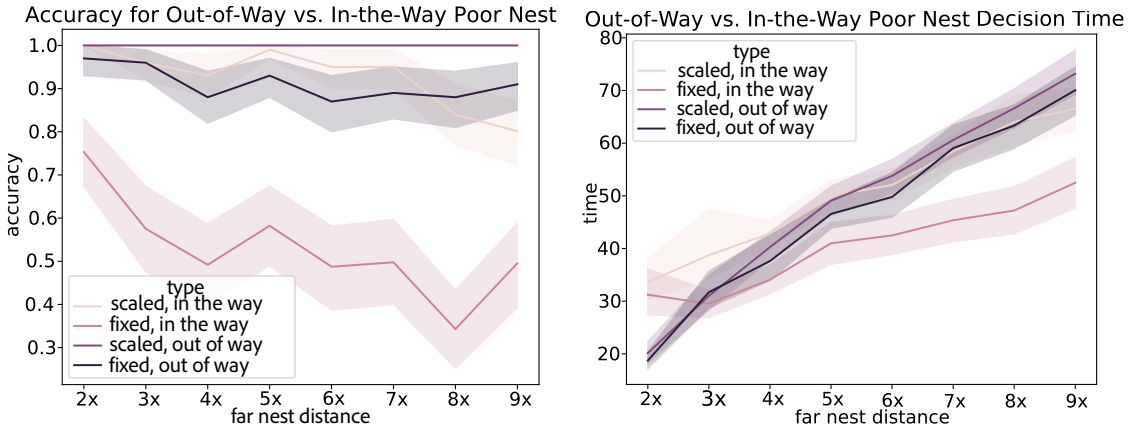

Figure 3-3: Decision Time and Accuracy for far nests $2-9$ times further than the close nest for both fixed and scaled quorums. In the in-the-way setup, the home nest, low quality nest, and high quality nest were lined up in that order. In the out of way setup, the low quality nest, home nest, and high quality nest were lined up in that order.

3.4.2 Effects of Lower Quality Nest Being in the Way

To isolate the effects of the low quality nest being in the way of the high quality nest, we tested our model where the high quality nest (quality 0.9) was one of $\{2, 3, 4, 5, 6, 7, 8, 9\}$ times further than the low quality nest (quality 0.3), but in opposite directions of the home nest. We compared model performance when the low quality nest was in the way of the home nest. We ran tests with $N = 18$, $M = 300$, with the low quality nest always 30cm from home. We again tested a fixed ($q_{MIN} = q_{MAX} = 4$) and scaled ($q_{MIN} = 4, q_{MAX} = 7$) quorum threshold on these setups. 100 trials were conducted for each set of parameters.

Figure 3-3 shows that for the out-of-way setup, the scaled quorum performs significantly (Welch's T-test, p=0.05) more accurately than the fixed quorum on all far nest distances. For the in-the-way setup, the scaled quorum performs significantly better (Welch's T-test, p=0.05) when the far nest is 3x further or more. Note it is harder for the fixed quorum to solve the in-the-way problem accurately compared to the out-of-way problem (Welch's T-test, p=0.05). It is likewise harder for the scaled quorum to solve the in-the-way problem when the far nest is $\{3, 4, 6, 7, 8, 9\}$ times further (Welch's T-test, p=0.05), showing that the in-the-way problem is harder to solve for site selection algorithms.

For distances 3x or further, there is no significant difference between the decision times for the fixed out-of-way, scaled out-of-way, and scaled in-the-way setups. For distances 5x and further, the fixed quorum takes significantly less time than the other setups but suffers in decision accuracy (Welch's T-test, p=0.05) compared to the other three setups.

3.4.3 Effects of Magnitude of Difference in Site Quality

Because site quality affects the quorum threshold, we expect it to be harder for agents to correctly choose a high quality far site when it is only slightly better than than nearby lower quality sites. This is because the difference in quorum threshold is less pronounced for sites of similar quality. For two equidistant nests, the algorithm should consistently choose the best site as it has in past work, so the absolute difference in site quality should not matter.

To test these effects, we used the setup in Section 4.1 where the further nest was 2x (60 cm) as far as the in-the-way close nest (30 cm), and compared it to an equidistant setup where both candidate nests were 30 cm away from the home nest in opposite directions. We tested both a fixed quorum $q_{MAX} = q_{MIN} = 4$ and a scaled quorum on these setups $q_{MAX} = 7, q_{MIN} = 4$. We varied the quality of the near nest in the set of potential values $\{0.3, 0.6, 0.9\}$, corresponding to quorum thresholds of $\{6, 5, 4\}$ respectively, with the far nest having quality 1.0. (In the equidistant case, we varied the quality of one nest while the other had quality 1.0.) Figure 3-4 shows the resulting accuracy and decision time.

As predicted, a smaller difference in site quality / quorum threshold led to significantly (Welch's T-test, p=0.05) lower decision accuracy for the non-equidistant setup. In the equidistant setup, agents were able to achieve a near-100% outcome regardless of magnitude of differences in site quality. However, in the unbalanced setup, we confirmed that for larger differences in site quality, the algorithm comes to a more accurate decision, showing that non-equidistant candidate nest setups cause sensitivity to absolute site value differences that can't be seen in the equidistant setup.

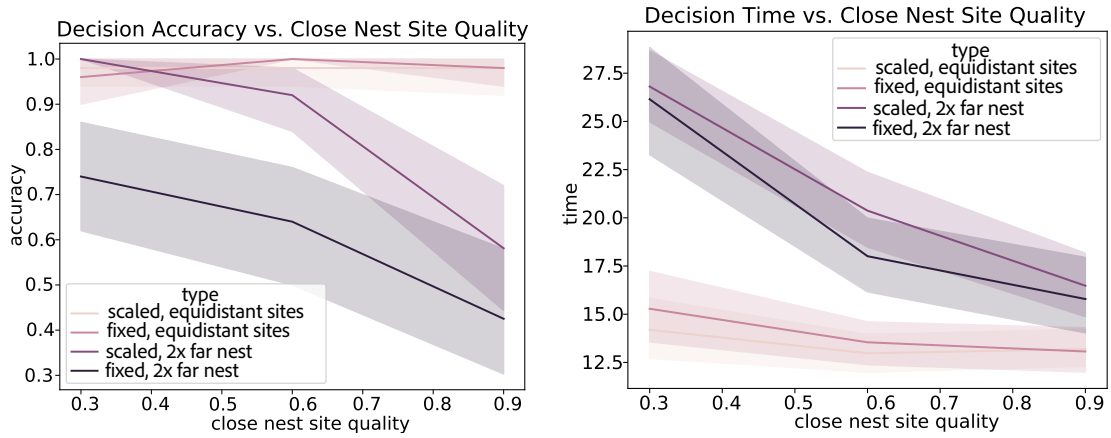

Figure 3-4: Decision Accuracy and Time given varying differences in site quality between the near and the far nest.

3.5 Discussion

The results demonstrate our model's ability to improve accuracy when choosing from a higher quality, further site and a lower quality, closer site. This improvement comes at the cost of a higher decision time when converging on a lower quality site, because the quorum threshold for low quality sites is higher in our model. This higher decision time is reasonable and represents hesitance when committing to a poor quality option in the hopes of finding a better one.

Our model also demonstrated the extra difficulty that comes with a lower quality site being in the path from the home nest to a high quality site. Qualitative observation showed that agents travelling back and forth between the far site and the home nest often unintentionally contributed to a quorum in the poor quality, in-the-way site as they travelled through it. We showed that using a scaled quorum threshold as opposed to a fixed one is an effective way of significantly increasing decision accuracy. However, even if the closer, poor quality site is completely out of the way of the far, high quality site, Figure 3-3 shows that using a scaled quorum can still help to improve accuracy.

Figure 3-4 shows that our model is still successful when candidate sites are equidistant from home, as is most commonly tested. We also show that an equidistant setup is not influenced by the absolute difference between candidate site qualities. Contrar-

ily, in the setup with a further, high quality nest, it is harder to make an accurate decision the smaller the quality difference between the high and low quality nests. Note that it is also less grievous of an error to choose the low quality nest when the quality difference is small.

We observed a shorter decision time in conjunction with lower accuracy, similar to the time-accuracy trade-off in natural swarms [12, 26]. In each set of 100 trials run, there were at most 2 split decisions, indicating our model succeeds in keeping the swarm together even when migrating to the further nest.

3.6 Future Work

While our model has made strides in being more accurate when choosing between sites with different geographical distributions, many site setups have yet to be tested. Future work could introduce obstacles to the environment, add uniquely shaped sites, or try to evaluate the house hunting algorithm on a more realistic simulator.

Our model suggests that a quorum threshold that scales with site quality leads to more accurate site selection. Future work could explore if actual ants do the same and use this information to create more accurate models.

Lastly, while our model is hard to analyze without making simplifications (because it involves agents physically moving in space), future work could try to develop analytical bounds. One method we envision is simplifying the chance of each site being discovered to a fixed probability and trying to model agent population flow between the different model states, similar to [43], which does this for candidate sites all with an equal chance of discovery.

3.7 Formal Pseudocode

Now, we describe the details of the house hunting model in terms of pseudocode framework in Section 2.4. Specifically, we specify the `Vertex State`, `Agent State`, agent transition function, and resolution rule, which are all problem specific.

3.7.1 Vertex State Class

An instance of the vertex state class has three variables. For the house hunting problem, the vertex state is synonymous with site information, and there will only be $N + 2$ unique vertex states for N sites (one for each candidate site, one for the home nest, and one meaning no site is present at this vertex).

- `name` is a string representing the name of the site that the vertex is a part of (for example, `Home` or `Site A` or `Site B`). We chose to use strings for human readability, but using an integer for site identifiers could work as well. If the vertex is not part of any site, the string is `null`.

- `value` is a string representing the value (a float from 0 to 1) of the site the vertex is a part of. If the vertex is not part of any site, the value is irrelevant and can take on any value.

- `location` is a pair of pairs ((a, b),(c,d)). If the vertex is part of a site (meaning `name` $\neq$ `null`), then `location` indicates that the site is located at the rectangle with x range [a, b] and y range [c, d].

Sites

Note that in order for the site desciptions in the vertex state to be geographically consistent, any vertex state with the same non-null `name` must also have the same `value` and `location`, since they are all part of the same site. Therefore, sites are uniquely defined by a corresponding vertex state. In later parts of the code, we store the vertex state of a vertex in a specific site and use that vertex state as the site information itself.

3.7.2 Agent State Class

An agent state has the following constant variables in addition to the constants from the general model. Constants are fixed in an execution of the simulator for all agents. Each agent has these constants as a part of its state, and the values never change.

- `L`, the inverse average site round trip
- `Q_MIN`, the minimum quorum threshold
- `Q_MAX`, the maximum quorum threshold
- `MESSAGE_RATE`, how often the agent sends messages when it has information, in units of 1/rounds
- `LEVY_LOC` and `LEVY_CAP`, levy flight distribution parameters, where `LEVY_CAP` caps the right tail of the distribution

Agents also have the following modifiable parts of their state, which may change during execution.

- `preference_type`, the preference type of the agent (Uncomitted, Favoring, Committed) as a String
- `activity_type`, the activity type of the agent (Active, Nest) as a String
- `home`, the information of the home nest (specifically `home.location` and `home.value`) which are defined the same way they are in the vertex state
- `angle`, the angle in radians of the current random walk the agent is taking (if there is one)
- `starting_point`, the (x,y) coordinates of the agent's starting point for the current straight-line leg of their random walk (if there is one)
- `travel_distance`, how many more steps the agent has to venture in the direction specified by `angle` and `starting_point`
- `destination`, the (x,y) coordinates of a point within the agent's destination site, if any
- `destination_site`, the vertex state of the vertex at `destination`, which contains the site's name, boundaries, and site value

- `exploring_site`, a boolean indicating whether the agent is currently exploring within a site
- `exploration_cooldown`, a cooldown time representing how many rounds to wait before exploring a new site after just having rejected a site. this prevents agents from constantly sensing the site they just explored and exploring it over and over again.
- `time_since_neighbor`, the time since a neighboring agent was last seen (used for an agent to determine whether or not to abandon a site)
- `favoured_site`, the vertex state of a vertex within the site the agent is favoring, if any
- `committed_site`, the vertex state of a vertex within the site the agent has committed to, if any
- `terminated`, a boolean keeping track of whether the agent is still running the house hunting algorithm or has finished

3.7.3 Agent Transition Functions

We now consider the generation of agent transitions at each timestep. This can be broken down into different logic based on the six main agent states – Uncommitted Nest, Uncommitted Active, Favoring Nest, Favoring Active, Quorum Nest, and Quorum Active. We describe the algorithm for each main state below. The pseudocode implementations utilize some utility functions in their implementation. The functionality of these utility functions is described in Appendix A.

Uncommitted Nest Transitions

In the main Uncommitted Nest state, an agent has not favored or committed to any sites yet and has the role of staying in the home nest and waiting to be recruited by others or to transition into the Uncommitted Active state. The code for the

transition function first checks if the Uncommitted Nest agent has sensed a quorum within influence radius (from other agents who are in the quorum state). If it has, the agent converts into the quorum state as well. Otherwise, we check if the agent has just transitioned to UN, in which case it may be outside of the home nest still, with the intention to travel there. If this is the case, the agent moves in the direction of the home nest. Otherwise, the agent is already at the home nest. In this case, it has a $L/9$ chance of transitioning to UA. Otherwise, it remains at the home nest and does not move. The pseudocode for this state can be seen in Algorithm 6.

Algorithm 6 Uncommitted Nest (UN) Agent Transition for an agent a

```
1:  procedure GENERATE_TRANSITION_UN(local_vertex_mapping))
2:      ▷Before anything else, check if a quorum was sensed
3:      quorum_sensed, new_agent_state ←
4:          check_quorum_sensed(local_vertex_mapping)
5:      if quourm_sensed then
6:          return a.location.state, new_agent_state, S
7:      s ← a.state
8:      ▷If the agent just transitioned to UN and is still travelling to the home nest
9:      if s.destination ≠ null then
10:         new_direction ← get_direction_from_destination(s.destination,
11:             (a.location.x, a.location.y))
12:         new_location ←
13:             get_coords_from_movement(a.location.x, a.location.y, new_direction)
14:         if a.within_site(new_location[0], new_location[1], s.home) then
15:             s.destination ← null
16:             return a.location.state, s, new_direction
17:     ▷If the agent is already in the home nest
18:     active_chance ← s.L/9
19:     if random_float_from(0,1) ≤ active_chance then
20:         s.activity_type ← Active
21:         return a.location.state, s, S
22:     else
23:         return a.location.state, s, S
```

Uncommitted Active Transitions

In the main Uncommitted Active state, an agent is patrolling the environment looking for new sites to explore. The pseudocode for this state can be seen in Algorithm 7. Once again, we begin by checking if a quorum has been sensed in the environment. If it has, the agent converts to the quorum state. Otherwise, the Uncommitted Active agent could either still be random walking the environment, or has found a site and is currently exploring it by doing a random walk inside of it. If the agent is random walking, we check if there are any sites nearby that it could discover and head towards. The agent also has a chance of transitioning to the Uncommitted Nest state as per the state transition diagram between the core states. If the agent is headed towards a site it is planning to explore, we let it continue heading towards the site. If the agent has arrived at the site it is planning to explore, we set it up to begin the random walk within the site. If the agent has finished exploring the site, it comes to a decision on whether to favor or reject the site. If it favors the site, it transitions to one of the favoring states. Otherwise, it enters an short exploration cooldown period to prevent it form exploring the same site it just left, and continues exploring.

Algorithm 7 Uncommitted Active (UA) Agent Transition for an agent a

```
1: procedure GENERATE_TRANSITION_UA(local_vertex_mapping))
2:     ▷Before anything else, check if a quorum was sensed
3:     quorum_sensed, new_agent_state ←
4:         check_quorum_sensed(local_vertex_mapping)
5:     if quourm_sensed then
6:         return a.location.state, new_agent_state, S
7:     s ← a.state
8:     ▷If the agent has not found a site to explore, look for one
9:     if s.destination is null and s.exploration_cooldown == 0 then
10:        nearby_site, site_location = find_nearby_site(local_vertex_mapping)
11:        ▷Explore the site the agent just found
12:        if nearby_site is not null then
13:            s.destination ← site_location
14:            s.desintation_site ← nearby_site
```

Algorithm 7 Uncommitted Active (UA) algorithm (continued)

```
    else if s.exploration_cooldown > 0 then
        ▷Decrement exploration cooldown if it is in effect
        s.exploration_cooldown -= 1
    ▷The agent is headed towards a site to explore
    if s.destination_site is not null then
        if s.location.state.site_name == s.destination_site.site_name and not s.exploring_site then
            ▷The agent has just arrived at the site
            s.exploring_site ← True
            s.travel_distance ← 10
            s.angle ← random_float_from(0, 2π)
            s.starting_point ← (s.location.x, s.location.y)
        else if not s.exploring_site then
            ▷The agent is still headed towards the site
            new_direction ← get_direction_from_destination(
                s.destination, (s.location.x, s.location.y))
            return s.location.state, s, new_direction
    ▷Calculate if agent should become nest agent
    nest_chance ← s.L
    if random_float_from(0,1) < nest_chance and not s.exploring_site then
        s.destination ← s.random_location_in_site(s.home)
        s.activity_type ← Nest
        return s.location.state, new_agent_state, new_direction
    ▷The agent has finished exploring the site
    if s.travel_distance == 0 and s.exploring_site then
        s.destination ← null
        s.exploring_site ← False
        s.favored_site ← s.destination_site
        s.destination_site ← null
        if random_float_from(0,1) < s.location.site_value then
            ▷Choose to favor the site
            if random_float_from(0,1) < 9/10 then
                s.destination ← s.random_location_in_site(s.home)
                s.destination_site ← s.home
                s.preference_type ← Favoring
                s.activity_type ← Nest
                return s.location.state, s, S
            else
                s.preference_type ← Favoring
                s.activity_type ← Active
                return s.location.state, s, S
```

Algorithm 7 Uncommitted Active (UA) algorithm (continued)

```
55:         else
56:             ▷Reject the site
57:             s.exploration_cooldown ← 10
58:             s.favored_site ← null
59:     ▷Agent continues exploring the arena
60:     new_direction, s ← get_travel_direction(s)
61:     return s.location.state, s, new_direction
```

Favoring Active Transitions

In the Favoring Active (FA) state, the agent has found a site to favor and remains in their favored site to build up quorum. Once again, we first check to see if a quorum has been sensed for the favored site. If the agent just transitioned to Favoring Active and is not yet at it's favored site, it continues heading towards the site. If the agent is already in the favored site, it has a chance of transitioning into Favoring Nest, or remaining as Favoring Active. Agents who remain as Favoring Active continue to perform random walks within their favored nest to try and build up quorum. The pseudocode for this main state can be seen in Algorithm 8.

Algorithm 8 Favoring Active (FA) Agent Transition for an agent a

```
1: procedure GENERATE_TRANSITION_FA(local_vertex_mapping))
2:     ▷Check if a quorum was sensed
3:     quorum_sensed, new_agent_state ←
4:         check_quorum_sensed(local_vertex_mapping)
5:     if quourm_sensed then
6:         return a.location.state, new_agent_state, S
7:     s ← a.state
8:     ▷Agent still headed towards their favored site
9:     if s.destination is not null then
10:        new_direction ← get_direction_from_destination(
11:            s.destination, (s.location.x, s.location.y))
12:        new_loc ← get_coords_from_movement(
13:            s.location.x, s.location.y, new_direction)
```

Algorithm 8 Favoring Active (FA) algorithm (continued)

```
14:         ▷Agent has just reached their favored site
15:         if s.within_site(new_loc[0], new_loc[1], s.favored_site) then
16:             s.destination ← null
17:         return s.location.state, s, new_direction
18:     ▷Agent is inside favored site
19:     nest_chance ← s.L
20:     if random_float_from(0,1) < nest_chance then
21:         s.destination ← s.random_location_in_site(s.home)
22:         s.desintation_site ← s.home
23:         s.activity_type ← Nest
24:         return s.location.state, s, S
25:     else
26:         new_direction ← s.get_travel_direction()
27:         return s.location.state, s, new_direction
```

Favoring Nest Transitions

In the Favoring Nest (FN) state, the agent has found a site to favor and returns to the home nest to try and recruit other Favoring Nest agents to support their favored site. First, if the time since the agent has seen any neighbors exceeds $5/L$, the agent abandons the site and becomes Uncommitted Active again. This is so that if everyone in the home nest has already moved to a new committed site, the favoring agent is able to abandon its own site and try to random walk the arena in search of the committed agents. Afterwards, once again, we check to see if a quorum has been detected for the favored site or some other site. If the agent has just turned into a favoring nest agent and is still headed towards to home nest, we calculate the next step in that direction. If the agent is already at the home nest, it tries to communicate with it's neighbors to see if they favor a higher quality nest. If they do, the favoring agent goes to that nest to examine it. If a favoring agent is headed towards a better quality site to explore it, we calculate the next step for it to continue exploring. After exploring the better site, the agent decides to favor the new site instead and changes state accordingly.

Algorithm 9 Favoring Nest (FN) Agent Transition for an agent a

```
1: procedure GENERATE_TRANSITION_FN(local_vertex_mapping))
2:     s ← a.state
3:     ▷Determine whether to abandon site
4:     should_abandon, new_state ← should_abandon_site(local_vertex_mapping, s)
5:     if should_abandon then
6:         return a.location.state, new_state, S
7:     ▷Check if a quorum was sensed
8:     quorum_sensed, new_agent_state ←
9:         check_quorum_sensed(local_vertex_mapping)
10:    if quourm_sensed then
11:        return a.location.state, new_agent_state, S
12:    if s.destination is null then
13:        ▷Communicate with other agents to see if a better nest can be found
14:        better_nest ← find_better_nest(local_vertex_mapping)
15:        if better_nest is not null then
16:            s.destination ← random_location_in_site(better_nest)
17:            s.destination_site ← better_nest
18:    if s.destination is not null then
19:        if s.location.site_name == s.destination_site.site_name and
   s.destination_site != s.home_nest and not s.exploring_site then
20:            ▷Agent has just arrived at the better site it heard of
21:            s.exploring_site ← True
22:            s.travel_distance ← 10
23:            s.angle ← random_float_from(0, 2π)
24:            s.starting_point ← (s.location.x, s.location.y)
25:        else if s.destination_site == s.home_nest and s.location.state.site_name
   == s.destination_site.site_name then
26:            ▷Agent just became FN and just arrived at home
27:            s.destination ← null
28:            s.destination_site ← null
29:        else if not s.exploring_site then
30:            ▷Agent still headed towards either home or a better nest
31:            new_direction ← get_direction_from_destination(
32:                s.destination, (s.location.x, s.location.y))
33:            return s.location.state, s, new_direction
34:    ▷FN agents have chance of becoming FA
35:    active_chance ← s.L/9
36:    if random_float_from(0,1) < active_chance and not s.exploring_site)
   then
37:        s.destination ← random_location_in_site(s.favored_site)
38:        s.activity_type ← Active
39:        return s.location.state, s, S
```

Algorithm 9 Favoring Nest (FN) algorithm (continued)

```
    ▷Agent finished exploring better site; changes to favoring it
    if s.travel_distance == 0 and s.exploring_site then
        s.destination ← null
        s.exploring_site ← False
        s.favored_site ← s.destination_site
        s.destination_site ← null
        if random_float_from(0, 1) < 9/10 then
            s.destination ← random_location_in_site(s.home)
            s.destination_site ← s.home_nest
            return s.location.state, s, S
        else
            s.activity_type ← Active
            return s.location.state, s, S
```

Committed Active Transitions

In the Committed Active state, agents have already sensed a quorum and are wandering the arena to broadcast the quorum to any straggling agents. After the agent has travelled a distance of 1/L (as set in the `check_quorum_sensed)` function definition, the agent returns to the home nest and finishes the algorithm. The algorithm for this state can be seen in Algorithm 10.

Algorithm 10 Committed Active (CA) Agent Transition for an agent a

```
procedure GENERATE_TRANSITION_CA(local_vertex_mapping))
    s ← a.state
    ▷Agent done with algorithm
    if s.terminated then
        return s.location.state, s, S
    ▷Agent finishing algorithm, headed to committed site
    if s.destination_site == s.quorum_site or s.travel_distance == 0
then
        s, new_dir ← committed_agent_state_and_dir(s)
    else
        new_dir ← get_travel_direction()
    return s.location.state, s, new_dir
```

Committed Nest Transitions

In the Committed Nest state, agents have already sensed a quorum and are either going towards / are inside the home nest to broadcast the quorum to other agents, or have finished broadcasting and are back inside the committed site. (Once committed agents return to the committed site, they have finished the algorithm and will no longer take any actions). The code for the committed nest transitions can be seen in Algorithm 11.

Algorithm 11 Committed Nest (CN) Agent Transition for an agent a

```
1: procedure GENERATE_TRANSITION_CN(local_vertex_mapping))
2:     s ← a.state
3:     ▷Agent done with algorithm
4:     if s.terminated then
5:         return s.location.state, s, S
6:     ▷Agent headed towards home nest to broadcast quorum
7:     if s.destination_site == s.home then
8:         if s.location.state == s.home then
9:             s.destination ← null
10:            s.destination_site ← null
11:            s.travel_distance ← int(1/s.L)
12:            s.angle ← random_float_from(0, 2π)
13:            s.starting_point ← (s.location.x, s.location.y)
14:    else
15:        new_direction ← get_direction_from_destination(
16:            s.destination, s.location.x, s.location.y))
17:        return s.location.state, s, new_direction
18:    ▷Agent finishing algorithm, headed to committed site
19:    if s.destination_site == s.quorum_site or s.travel_distance == 0
   then
20:        s, new_dir ← committed_agent_state_and_dir(s)
21:    else
22:        new_dir ← get_travel_direction()
23:    return s.location.state, s, new_dir
```

3.7.4 Resolution Rule

In the house hunting model, agents do not affect vertex states in any way (they are not acting on their environment) so their new proposed vertex state always remains the same as the old vertex state. Therefore, all agents have the same understanding of the grid when they are transitioning, so all agents should be allowed to execute their new proposed state and direction of motion. Therefore, we use a naive resolution rule which just picks a random one of the new vertex states (since they all should be the same) and accepts all proposed agent state and direction changes. The naive resolution rule can be seen below:

Algorithm 12 Naive resolution rule

```
1: procedure NAIVE_RESOLUTION(proposed_vertex_states, proposed_agent_states, proposed_agent_dirs))
2:     new_vertex_state ← proposed_vertex_states.values()[0]
3:     new_agent_states ← proposed_agent_states
4:     new_agent_dirs ← proposed_agent_dirs
5:     return new_vertex_state, new_agent_states, new_agent_dirs
```

Note that while the resolution rule for house hunting is simple, the resolution rules for other problems, such as task allocation, where agents are modifying their environment, will need to actively choose a new proposed vertex state and may have to block some agents from transitioning state and direction if their vertex state proposal was rejected.

Chapter 4

Two Task Allocation Algorithms in Unknown Environments with Varying Task Density

Task allocation is an important problem for robot swarms to solve, allowing agents to reduce task completion time by performing tasks in a distributed fashion. Many existing task allocation algorithms assume prior knowledge of task location and demand or fail to consider the effects of the geometric distribution of tasks on the completion time and communication cost of the algorithms [4, 5, 23, 56]. In this thesis, we examine an environment where agents must explore and discover tasks with positive demand and successfully assign themselves to complete all such tasks. We propose two new task allocation algorithms for initially unknown environments – one based on N-site selection and the other on virtual pheromones. We analyze the effect of algorithm-specific parameters on each algorithm and also evaluate the effectiveness of the two algorithms in dense vs. sparse task distributions. Compared to the Levy walk, which has been theorized to be optimal for foraging [54], our virtual pheromone inspired algorithm is much faster in all but very high task densities but is communication and agent intensive. Our site selection inspired algorithm also outperforms Levy walk in sparse task densities and is a less resource-intensive option than our virtual pheromone algorithm for this case. At very high task densities, Levy walk

outperforms our algorithms because the time spent communicating about tasks is not worth it when tasks are very easily found. Because the performance of both algorithms relative to random walk is dependent on task density, our results shed light on how task density is important in choosing a task allocation algorithm in initially unknown environments.

4.1 Introduction

Robot swarms are simple, distributed units that are able to work together to achieve emergent collective behaviours [18]. Swarm algorithms often draw inspiration from swarms in nature such as birds, ants, and bees [41, 46, 31]. Swarm algorithms provide a scalable and fault-tolerant solution to problems such as search-and-rescue [14] and environmental monitoring [17]. One of the most well-studied swarm problems is task allocation [22], which aims to assign agents to tasks in an optimal manner.

Many classes of task allocation algorithms assume that task locations and demand for agents are known, and try to optimize an assignment of agents to tasks [5, 56, 4]. However, in many applications, such as finding and defusing mines [50], task information is not initially known. Algorithms which do consider task allocation in unknown environments run limited testing on the effects of task density. However, the density of tasks in the environment affects relative algorithm performance.

In this paper, we consider the problem of assigning agents to tasks with positive demand in an initially unknown environment. We assume each agent can only be assigned to one task. Within this framework, we contribute two new algorithms and compare them to the Levy Walk (RW), which is used in nature for foraging [45]. We also show how task density makes our different algorithms better suited towards different task environments.

The first algorithm, our house hunting task allocation algorithm (HHTA), is inspired by swarm house-hunting models [44]. While the house hunting problem aims for agents to agree on one of many locations in the environment, the task allocation problem aims for agents to split themselves proportionally to task demand amongst

all tasks in the environment. In our HHTA algorithm, agents use their starting location as a home base that they can return to after discovering tasks in the environment. The home base functions as a central point of communication and allows for agents to recruit each other to do tasks, serving the same function as the home nest in many swarm house hunting algorithms such as our new algorithm in Chapter 3.

The second algorithm is a propagation-based algorithm (PROP), which uses a regular grid of cheap, simple agents to propagate task demand information outwards to neighboring propagator agents. We assign a separate type of agent with more advanced computing powers to read the information and use it to probabilistically decide which task to head towards. The propagation of task demand information via cheap agents is inspired by virtual pheromones [1, 3], a commonly used nature-inspired technique in swarm algorithms.

By comparing both algorithms to the Levy flight, we show that it is harder for PROP to do well with very dense tasks, as a large influx of propagated information can confuse agents. Our other algorithm, HHTA, does worse when tasks are mid to high density because inter-agent communication about tasks is not worth it compared to a random walk, which is highly likely to encounter tasks quickly. However, it does better than RW when tasks are sparse as the cost of communication about task location is justified when tasks are harder to find. It is also less resource intensive compared to PROP. We also evaluate the effects of varying individual parameters within several task densities in order to better understand our new algorithms.

Our results demonstrate how different task allocation algorithms do well in environments with different task density and invite further examination on the performance of other task allocation algorithms in different types of task environments.

Section 4.2 provides the inspiration for our two proposed algorithms, explaining house hunting and virtual pheromones in greater depth. Section 4.3 describes our general formal model and our task allocation problem statement. Section 4.4 dives into our two new algorithms. Our simulation results and comparison between the two new algorithms in sparse and dense task environments can be found in Section 4.5. Section 4.6 discusses our results, and Section 4.7 concludes our findings and provides

ideas for future work. Section 4.8 provides pseudocode details for the two algorithms we introduce.

4.2 Background

Task allocation is a well studied problem and several variants of the problem have been studied. Per the taxonomy defined in [22] our task allocation problem is of the single-task agents, multi-robot tasks variety, which means that agents can only do one task at a time, but tasks may require multiple agents.

When task demands and locations are known, this problem becomes a coalition formation problem, where we wish to form agents into groups that are best suited to do each task. This problem can be thought of as a set partitioning problem [22], and adaptations to distributed swarms have been proposed [5, 56].

Other strategies for when tasks are fixed at known locations model tasks as a graph where agents can travel between edges [4, 30, 23]. These algorithms optimize for a flow rate between edges in the graph so that agents can satisfy all task demands quickly. Another strategy in this case, based on Optimal Mass Transport [52], is to treat the tasks with demands as sinks and the tasks with agents as sources in a min cost flow problem. However, both strategies require prior knowledge of task locations.

Our problem differs from coalition formation and the graph-based task allocation problems because we are assuming that agents have no initial knowledge of task location or demands. In this case, we want to discover tasks and communicate information about them as quickly as possible so that agents can satisfy all task demands.

One solution to task allocation in an environment with unknown tasks is to have agents form local clusters and run Optimal Mass Transport locally [57]. Other task allocation algorithms, such as auction-based algorithms, perform a similar type of agent clustering to assign tasks [28]. Our two algorithms by contrast are fully distributed and computationally simple, without the need for grouping to locally run a complex centralized algorithm. This allows us to save the time needed to form agent clusters and allows agents to be cheaper to implement due to low computation cost.

4.2.1 Levy Flight

The Levy Flight is a random walk shown to be optimal in certain foraging scenarios. When the resources being foraged are destructive (resources do not reappear after being foraged) or are non-destructive (resources can be visited any number of times) and sparse, then the Levy Flight is the optimal random walk for foraging [54]. The Levy flight has been observed in foraging animals and has been adapted to swarm algorithms [45, 20] as well. As such, we will be using this random walk as a baseline to compare against for our two new algorithms.

4.2.2 House Hunting

Several ant species engage in a house-hunting behaviour when their home nest is destroyed [41, 40]. First, ants explore nearby for nest sites. If a site is found, the ant waits a period of time inversely proportional to the site quality before returning to the home nest to lead others to the new nest. This process of recruitment is known as forward tandem running (FTR). Once the encounter rate of other ants in the candidate nest reaches a critical threshold known as the quorum threshold, ants switch to carrying members of the colony to the new nest. This carrying behaviour is 3 times faster than FTR and accelerates the move to the new site [41].

Ant house hunting has inspired the corresponding swarm problem of N-site selection [53], where agents must choose the highest quality site from N initially unknown candidates. One common N-site selection model has agents transition between four main states: Uncommitted Interactive, Uncommitted Latent, Favoring Interactive, and Favoring Latent [44]. Some works also include a fifth Committed state [32, 13, 7]. In this type of model, Uncommitted Interactive agents explore the arena for new sites, while Uncommitted Latent agents stay in the home nest. Once an Uncommitted Interactive agent discovers a site, it can decide to favor the site. Favoring agents can be interactive, meaning they return to the home nest to recruit other favoring agents, or latent, meaning they stay in their favored site to build up quorum. Lastly, if agents detect a sufficient number of others in a new candidate site, they can transition into

the committed state to finalize their decision.

Task allocation can be thought of as a variant of the house hunting problem, where instead of trying to send all agents to one location, we want to send agents to multiple locations according to the demand at each one. This idea has been used in Berman [4] and Halasz's [23] work to develop task allocation algorithms for a known graph of tasks where agents can traverse along the edges. We extend this idea further by using inspiration from site selection algorithms to develop our HHTA algorithm. In HHTA, agents use a home nest which functions as a location for recruiting other agents to tasks and communicating with other agents. The four main states of the HHTA algorithm share parallels to the Uncommitted Interactive, Uncommitted Latent, Favoring Active, and Committed states described above which are further explained in Section 4.4.1.

4.2.3 Virtual Pheromones and Potential Fields

Ants leave pheromones in their environment when foraging to guide other ants to any discovered food sources [1]. This strategy of leaving information in the environment has inspired swarms to implement virtual pheromones (pheromones represented by computational data instead of chemical signals). For example, [3] used physically deployable beacons that robots could leave in the environment to store information in, and [36] set up a virtual pheromone approach with a pre-deployed network of beacons that acted as a grid of locations to leave information in. One cheap way to implement virtual pheromones is using wireless sensor motes to store and propagate information [49].

Pheromones are frequently used in conjunction with potential fields or particle swarm optimization techniques. Potential field algorithms model objects in the environment as either positive charges or negative charges, with agents experiencing attraction or repulsion from the objects based on the electric force between them. Particle swarm optimization [39] follows a similar physics approach, except the attractive and repulsive forces were based on springs as opposed to charges. These techniques are employed in navigation tasks, where potential fields and pheromones

can work together to guide robots around obstacles and towards a target in space [38]. Pheromones are also employed in foraging tasks to help robots efficiently find what they are foraging for [29].

We utilize the ideas of virtual pheromones in our propagation-base algorithm, which uses simple mote-like agents to leave task information in the environment. Task-performing robots use this information when searching for tasks in the task allocation process. The use of virtual pheromones allows us to easily notify task-performing robots of nearby tasks. We also use potential fields as inspiration for how a robot's motion should be influenced when it learns of multiple potential tasks through pheromones in the environment. Robots are more attracted to tasks with higher demand and tasks that are closer to their current location, so tasks can be thought of like charges which robots can feel the force of.

4.3 Model

We first summarize our new discrete general model (presented in Chapter 2) for modeling swarms. Then we discuss the individual restrictions, parameters, and agent algorithms needed for task allocation.

The pseudocode for our general model can be found in Section 2.4. A pseudocode description of our task allocation algorithms can be found in Section 4.8.

4.3.1 General Model

The general model we present in this section is an abbreviated description of our model in Chapter 2. For more formal details, please see Chapter 2.

We use the special case of a rectangular grid instead of a torus. The special case works by having agents treat the graph as a grid by never generating an out-of-bounds direction of motion and by never using any vertices in their influence radius that they wouldn't be able to see on a grid due to edge effects. For more details on this special case, please see Section 2.2.

We assume a finite set R of agents, with a state set SR of potential states. Agents

move on a discrete rectangular grid of size $N \times M$, formally modelled as directed graph $G = (V, E)$ with $|V| = MN$. Edges are bidirectional, and we also include a self-loop at each vertex. Vertices are indexed as (x, y), where $0 \leq x \leq N - 1$, $0 \leq y \leq M - 1$. Each vertex also has a state set SV of potential states.

Local Configurations: A *local configuration* $C'(v)$ captures the contents of vertex v. It is a triple $(sv, myagents, srmap)$, where $sv \in SV$ is the vertex state of v, $myagents \subseteq R$ is the set of agents at v, and $srmap : myagents \to SR$ assigns an agent state to each agent at v.

Local Transitions: The transition of a vertex v may be influenced by the local configurations of nearby vertices. We define an **influence radius** I, which is the same for all vertices, to mean that vertex indexed at (x, y) is influenced by all valid vertices $\{(a, b)|a \in [x - I, x + I], b \in [y - I, y + I]\}$, where a and b are integers. We can use this influence radius to create a local mapping M_v from local coordinates to the neighboring local configurations. For a vertex v at location (x, y), we produce M_v such that $M_v(a, b) \to C'(w)$ where w is the vertex located at $(x + a, y + b)$ and $-I < a, b < I$. This influence radius is representative of a sensing and communication radius. Agents can use all information from vertices within the influence radius to make decisions.

We have a local transition function δ, which maps all the information associated with a vertex and its influence radius at one time to new information that can be associated with the vertex and the agents at that vertex for the following time.

Formally, for a vertex v, δ probabilistically maps M_v to a quadruple of the form $(sv_1, myagents, srmap_1, dirmap_1)$, where $sv_1 \in SV$ is the new state of the vertex, $srmap_1 : myagents \to SR$ is the new agent state mapping for agents at the vertex, and $dirmap_1 : myagents \to \{R, L, U, D, S\}$ gives directions of motion for agents currently at the vertex. Note that R, L, U, and D mean right, left, up, and down respectively, and S means to stay at the vertex. The local transition function δ is further broken down into two phases as follows.

Phase One: Each agent in vertex v uses the same transition function α, which probabilistically maps the agent's state $sr \in SR$, location (x, y), and the mapping

M_v to a new suggested vertex state sv', agent state sr', and direction of motion $d \in \{R, L, U, D, S\}$. We can think of α as an agent state machine model.

Phase Two: Since agents may suggest conflicting new vertex states, a rule L is used to select one final vertex state. The rule also determines for each agent whether they may transition to state sr' and direction of motion d or whether they must stay at the same location with original state sr.

Probabilistic Execution: The system operates by probablistically transitioning all vertices v for an infinite number of rounds. During each round, for each vertex v, we obtain the mapping M_v which contains the local configurations of all vertices in its influence radius. We then apply δ to M_v to transition vertex v and all agents at vertex v. For each vertex v we now have $(sv_v, myagents_v, srmap_v, dirmap_v)$ returned from δ.

For each v, we take $dirmap_v$, which specifies the direction of motion for each agent and use it to map all agents to their new vertices. For each vertex v, its new local configuration is just the new vertex state sv_v, the new set of agents at the vertex, and the *srmap* mapping from agents to their new agent states.

4.3.2 Task Allocation Problem Definition

Consider T tasks $[t_0 \dots t_{T-1}]$ arranged at a subset of vertices in our general model, with at most one task at each vertex. Specifically, the task locations can be described as $l = [(x_0, y_0), \dots, (x_{T-1}, y_{T-1})]$, where $l_i = (x_i, y_i)$ is the vertex location of task t_i and $i \neq j \rightarrow l_i \neq l_j$ (each task has a distinct location). We wish to distribute agents among the tasks to achieve a certain distribution $a = [a_0, ..., a_{T-1}]$ where a_i represents the number of agents doing task i and $\sum a_i = kR < R$ (meaning a fraction k of all agents is enough to complete all the tasks).

We assume that when an agent senses a task within its influence radius, it is able to detect the demand of that task. This assumption follows from our general model, which assumes agents have access to all vertex states of the vertices within their influence radius. Since agents can also detect how many agents are at the task, they can use this information to compute the *residual demand*, defined as the difference

between the task demand and the number of agents already at the task. We denote the residual demand at task i by t_i^{rd}. We assume that the desired task distribution does not change over time, and that the task is complex enough that each agent can only do one task over the course of the algorithm.

In order to properly represent tasks in both of our algorithms, the vertex state set SV contains the following variables: `is_task`, whether the vertex is a task; `demand`, the task demand if the vertex is a task; `residual_demand`, the residual demand if the vertex is a task; `task_location`, the x, y coordinates of the vertex if it is a task.

We go into more detail on the agent states and transitions for our two algorithms in Sections 4.4.1 and 4.4.2. One other detail to note about task allocation is that in phase two of δ, we reconcile conflicting proposed vertex states. This shows up in task allocation when multiple agents attempt to claim the same task. When this happens, if there are s agents trying to claim the task but only $rd < s$ residual demand, then only rd agents are allowed to transition their state to having claimed the task. Otherwise, if $rd > s$, all agents will be allowed to claim the task.

4.4 Algorithms

In this section we present our two new task allocation algorithms, HHTA and PROP, which are designed to run within the general model framework described in Section 4.3.

4.4.1 House Hunting Task Allocation Algorithm

In our house-hunting inspired algorithm (HHTA), agents start out at a square home nest with lower left corner (x_h^1, y_h^1) and upper right corner (x_h^2, y_h^2). Call the set of vertices that make up the home location $\mathcal{H}$. We assume that $\forall i, l_i \notin \mathcal{H}$, meaning no tasks are located at any home nest vertex. In HHTA, the vertex state set SV needs the additional variable `is_home`, indicating whether the vertex is a home vertex or not.

Agents can be in one of four core states: Home (H), Exploring (E), Recruiting (R),

or Committed (C). Home agents wait in the home nest for news of tasks. Exploring agents explore the arena for tasks. Recruiting agents go to the home nest and try to recruit others to go to a task they have found. Committed agents are already committed to a task do not participate in any further decision making. The algorithm is parameterized by the following three variables, which are used in the transitions between these four core states:

- P_c, the base probability of committing to a task after discovering it
- P_e, the expected fraction of exploring agents
- r_m, the probability for Recruiting agents to send a message to another agent at any given time step

All other variables used in the description of the algorithm that cannot be directly calculated from $\{P_c, P_e, r_m, N, M\}$ (where N and M are the size of the arena) are unknown quantities that could be determined via simulation.

Home agents have a P_E chance of converting to exploring agents, where P_E is defined as $\frac{L*P_e}{1-P_e}$. Here, P_e is the expected fraction of exploring agents and L is defined as $1/(M+N)$ where M is the width of the grid, and N is the height of the grid. Exploring agents have a P_H chance of converting to Home agents, where P_H is defined as L. Note that L is independent of the number of agents, the number of tasks, and the total task demand.

We let P_{t_i} denote the probability that an exploring agent finds task i at any given time step. Once it finds task i, it has a $c = \max(P_c, 1/t_i^{rd})$ chance of becoming a Committed agent, where P_c is an algorithm parameter and t_i^{rd} is the residual demand of task i as defined in Section 4.3.2. The agent has a $1-c$ chance of becoming a Recruiting agent instead.

Committed agents have fully committed to a task and stay at that task. Recruiting agents head back to the home nest to tell Home agents about the task they have found. Agents recruiting for site i have a $1/t_i^{rd}$ chance to stop recruiting and become committed to task i.

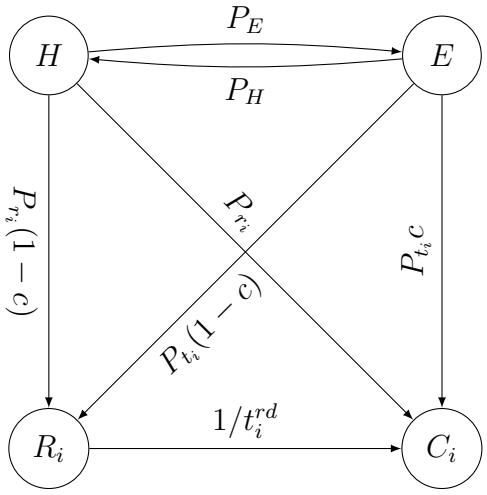

Figure 4-1: State model of the four core states. The subscript i denotes that an agent is recruiting for or committed to site i.

Recruiting agents have a r_m chance of sending a message to each agent within their influence radius at each time step, where r_m is the message rate. We let P_{r_i} denote the probability that a Home agent receives at least one recruiting message for task i at any given time step. If a Home agent does receive a message from a recruiting agent, it has a P_c chance of committing to the task and heading towards it, and a $1 - P_c$ chance of recruiting for the task. Note that the residual demand information for C and R agents may become stale as more agents commit to tasks. A diagram of the transitions between these core states can be found in Figure 4-1.

In order to execute the core state transitions, the agent state set SR consists of the following variables: `core_state`, which can be H, E, R or C; `id`, the agent id, taking on values from $0 \ldots |R| - 1$; `L`, defined as $L = 1/(M + N)$; `P_commit`, the probability P_c; `P_explore`, the probability P_e; `message_rate`, the message rate r_m; `angle`, the agent's current angle of travel; `starting_point`, a random walk parameter tracking where the agent started from; `travel_distance`, the length of the current leg of the random walk; `destination_task`, the agent's destination if they have just found a task or are headed towards their committed task; `home_destination`, the agent's destination if they are headed to a home vertex; `recruitment_task`, the task an agent is recruiting for; and `committed_task`, the task an agent has committed to. The agent transition function α uses these state variables to implement the transitions

between the four core states.

We would now like to provide some intuition for the transitions presented in the HHTA algorithm. Firstly, the transition probability P_H from the Exploring to the Home state (with L being defined as $\frac{1}{M+N}$ and P_H being defined as L) indicates that agents are expected to explore for $M + N$ time steps (enough to reach the corners of the grid) before returning home. The factor of L is inspired by house hunting algorithms, where L is defined as the inverse of the average site round trip so that exploring agents will have enough time to reach candidate sites before returning home. The transition P_E in the other direction from Home to Exploring ensures that the expected fraction of Exploring agents out of the total number of Exploring and Home agents is P_e.

When an Exploring agent discovers a task, its probability of becoming Committed was mentioned to be $\max(P_c, 1/t_i^{rd})$. Here, P_c is the base probability of committing. We take the max of P_c with $1/t_i^{rd}$ to ensure that if task i has low enough residual demand t_i^{rd} such that $1/t_i^{rd} > P_c$, then agents have a higher chance than the base probability P_c of committing to the task right away. If a task has residual demand 1, for instance, any agent which discovers it will commit to the task right away instead of trying to recruit others for it.

We would like to mention that P_{r_i}, the probability that a Home agent receives at least one recruiting message for task i at any given time step, can be defined as $P_{r_i} = I_{1-r_m}(R_{t_i} - 1, 2)$ where R_{t_i} represents the number of agents recruiting for task i that are within sensing radius, and I is the regularized incomplete beta function. This formula can be derived by seeing that the distribution of the number of messages that the Home agent receives about task i is a binomial distribution with $n = R_{t_i}, p = r_m$ (where n is the number of independent experiments and p is the probability of success). Thus, the probability that at least k messages are received is the CDF of this binomial distribution evaluated at k, which is given by $I_{1-p}(n-k, 1+k)$. Since we are looking for the probability that at least one message about task i is received, $k = 1$. Plugging in for p, n, k gives $P_{r_i} = I_{1-r_m}(R_{t_i} - 1, 2)$.

4.4.2 Task Propagation Algorithm

In our task propagation algorithm (PROP), we distinguish between two types of agents – $N * M$ propagators and F followers. Propagators are simple, mote-like agents. One of them is assigned to each vertex to allow vertices to propagate task information to each other. Followers are more advanced agents which are able to perform the tasks in the task allocation problem. Followers follow the signals left by propogators in order to find tasks.

Similarly to HHTA, all agents are initially deployed at a square home location with lower left corner (x_h^1, y_h^1) and upper right corner (x_h^2, y_h^2). However, agents in PROP do not utilize this home location after starting the algorithm. We assume this home location to be at the center of the grid to ease deployment. First, all $N * M$ propagators travel to the vertex which they are assigned to, taking $\frac{M+N}{2}$ time for all agents to reach their assigned vertex.

Each propagator has an influence radius of 1 and also stores in its state a mapping $\mathcal{M}_T$ from task locations l_i to residual demands $t_i^{rd'}$, representing that it has heard that task i at location l_i has residual demand $t_i^{rd'}$. After all propagators are in place, propagators that are at a task vertex i spread the tuple $(t_i^{rd'}, l_i)$ to all other propagators in their influence radius. Every t_p time steps, a propagator takes all new task information (if it has new information it did not already propagate) it has received and spreads that information to all other propagators in their influence radius with the following conditions: information about task i can only be spread to agents whose assigned vertex v is located within the bounds $[x_i - I_M, x_i + I_M]$ for the x coordinate and $[y_i - I_M, y_i + I_M]$ for the y coordinate, and the Euclidian distance between i and v must be less than or equal to d_p. Here, t_p is the integer propagation timeout and d_p is the maximum propagation radius.

Because the residual demand of a task changes over time, the propagator at task i will have to send new information whenever the residual demand decreases. When a propagator which already has task information $\mathcal{M}_\mathcal{T}(l_i) \rightarrow t_i^{rd'}$ receives new information about a task $(t_i^{rd''}, l_i)$, it updates the task information for task i to be

$\mathcal{M}_{\mathcal{T}}(l_i) \rightarrow min(t_i^{rd'}, t_i^{rd''})$ in order to have the most up-to-date information. Since the residual demand of a task is always decreasing as more and more agents join the task, we know the smaller residual demand is the more accurate one.

Followers try to use the information of propagators in order to find tasks to head towards. Followers only begin their algorithm after all propagator agents have reached their assigned vertex and begun the propagation process described above. At every time step, a follower first checks the vertices within its influence radius for a task with non-zero residual demand, and starts moving towards that task if it exists. If no task is found in its influence radius, a follower located at (x, y) looks at the propagator assigned to location (x, y) in order to get information about potential task locations it could head towards. It compiles all non-zero residual demands into the resulting mapping M_F, which maps from task locations l_i to residual demands $t_i^{rd'}$. If M_F is non-empty (there is at least one task location with non-zero residual demand) then the probability that a follower located at (x, y) heads towards task location $l_i \in M_F$ is:

$$\frac{\frac{M_F(l_i)}{L_2(l_i,(x,y))^2}}{\sum_{l_j \in D(M_F)} \frac{M_F(l_j)}{L_2(l_j,(x,y))^2}} \tag{4.1}$$

This means that the probability of a follower heading towards a task has an inverse square relationship with L_2 distance between the task location and the agent's location, and is also weighted by the residual demand of the task itself. This equation is determined so that agents are less likely to travel to tasks that are further away from them, but more likely to travel to a task if it has higher residual demand. If the mapping M_F is empty (the agent has no task information), it takes a random step in one direction $\{L, D, R, U\}$ (following a Levy flight random walk) in order to explore.

Once a follower agent reaches a task with non-zero residual demand, it stays there indefinitely, "completing the task" and decrementing the task's residual demand by one.

In order to execute the algorithm, the agent state set contains the following variables: `type`, the type of agent, which can be 'propagator' or 'follower' and `id`, the agent id, which takes on values from $0 \ldots |N \cdot M + F| - 1$. The following additional

variables are in SR and are only used by propagator agents: `task_info`, the mapping $\mathcal{M}_{\mathcal{T}}$; `propagation_rate`, the propagation timeout t_p; and `propagation_ctr`, the number of rounds since an agent last propagated task information. Lastly, the variables in SR used only by follower agents are: `destination_task`, the agent's destination if they have just found a task or are headed towards their committed task; `committed_task`, the task an agent has committed to; `angle`, a random walk parameter denoting angle of travel; `starting_point`, a random walk parameter tracking where the agent started from; and `travel_distance`, the length of the current leg of the random walk.

4.5 Results

Our algorithms were tested in simulation using Pygame on a grid of size $M = N = 50$. Each vertex had an area of 1cm^2, meaning that agents moved at 1cm/s, a speed which simple, low-cost robots are able to move at [48]. All simulations were run using 100 task-performing agents and the total task demand summed to 80. In the trials for the HHTA algorithm, agents had an influence radius of 2. In the trials for the PROP algorithm, propagators had an influence radius of 1 and followers had an influence radius of 2.

For each set of trials, we evaluated task completion time, defined as the time necessary for the total residual demand to become 0. In subsections 4.5.1 and 4.5.2, we also measure the average number of messages sent per run per agent. For the HHTA algorithm, whenever a Home agent is notified of a task by a Recruiting agent, the Recruiting agent's message count is incremented. For the PROP algorithm, the message count is incremented when a propagator shares new task information with one of its neighbors. We do not track the message count for follower agents since it is a negligible portion of total messages.

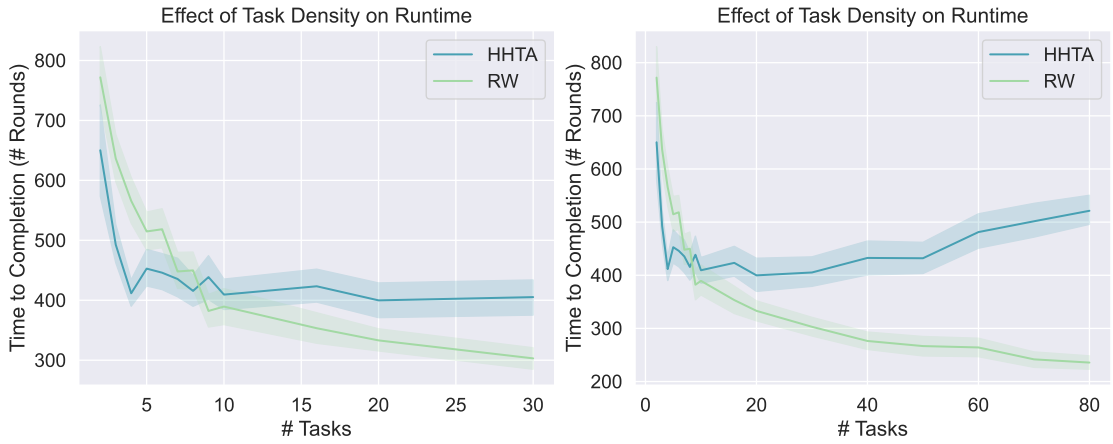

Figure 4-2: The effect of number of tasks on completion time for HHTA and RW, tested on up to 30 tasks (left) and on up to 80 tasks (right). The graph on the left highlights the sparser task densities where HHTA performs best.

4.5.1 Effects of Task Density on HHTA Performance

To examine the effects of task density on the HHTA algorithm's performance, we measured task completion time and average number of messages sent per agent for $T \in \{2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80\}$. For each value of T (the number of tasks), we ran 100 trials with $m_r = \frac{1}{6}$, $P_e = \frac{2}{3}, P_c = \frac{3}{10}$. Note that T directly determines task density since the size of our arena was held fixed. Therefore, we refer to low T as low task density and high T as high task density.

We compared the performance of HHTA to the Levy Flight random walk (RW). The Levy Flight random walk sampled random walk step lengths from a Levy distribution with $\mu = 10$. Figure 4-2 shows the resulting average task completion time for varying task densities. The HHTA algorithm outperforms the random walk by about 100 rounds in very sparse task setups when $T \leq 6$ and performs comparably when $7 \leq T \leq 10$, but for denser task setups, the cost of returning to the home nest to recruit others is too high compared to the random walk (Welch's T-test, p=0.05). We can approximate the area covered by detectable tasks as $\frac{T(2I+1)^2}{NM}$, where $(2I+1)^2$ is the size of the influence radius (in reality, the ratio would be a bit smaller as the detectable range for tasks can intersect). This means that for our choice of parameters, the HHTA algorithm outperforms the random walk when about 6% or less of the total task area has an immediately detectable task.

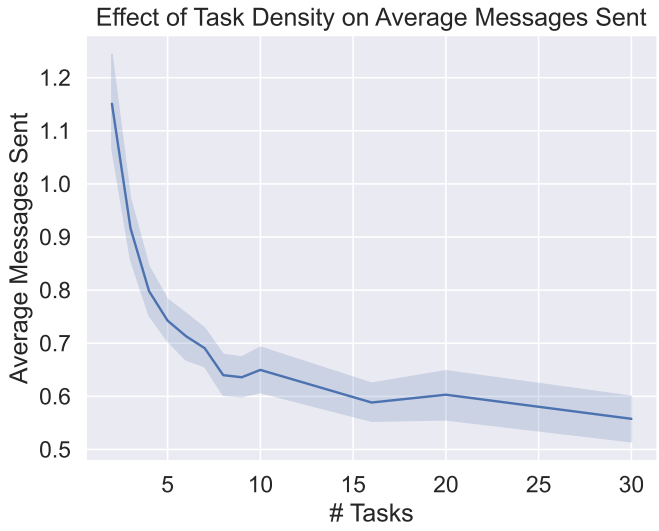

Figure 4-3: The effect of number of tasks on average messages sent per agent for HHTA

Figure 4-3 shows the average number of messages sent per agent for the HHTA algorithm. (Note that the random walk algorithm uses no communication). Note that on average, each agent sends fewer than 1.2 messages per round using HHTA. Note also that agents send fewer messages on average as density increases. Since the total task demand is fixed at 80, a larger number of tasks indicates less demand per task on average, making agents in the HHTA algorithm less likely to enter the task state (where messages are sent) and remain in it.

4.5.2 Effects of Task Density on PROP Performance

To examine the effects of task density on the PROP algorithm's performance, we once again measured task completion time and average number of messages sent per propagator agent for $T \in \{1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 25, 30, 35, 40, 50, 60, 70, 80\}$. For each value of T (the number of tasks), we ran 20 trials with $t_p = 3$ and $d_p = 25$. Note again that T directly determines task density since the size of our arena was held fixed. Therefore, we refer to low T as low task density and high T as high task density.

We compared the performance of PROP to the same Levy Flight random walk (RW) that we compared HHTA with. The Levy Flight random walk sampled random

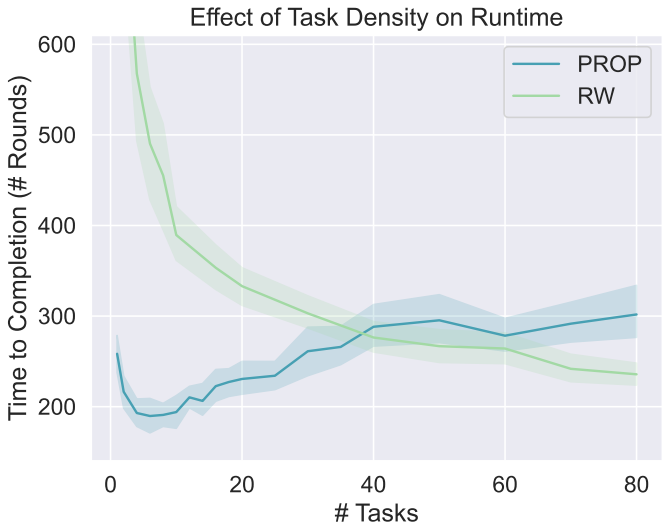

Figure 4-4: The effect of number of tasks on completion time for PROP and RW

walk step lengths from a Levy distribution with $\mu = 10$. Figure 4-4 shows the resulting average task completion time for varying task densities. The PROP algorithm outperforms the random walk significantly (Welch's T-test, p=0.05) in sparser task setups ($T \in [1, 25]$), with fewer tasks exaggerating this performance gap nonlinearly. In moderately dense task setups ($T \in [30, 60]$), the two algorithms' runtimes are comparable, and in our most dense task setups ($T \in [70, 80]$), the random walk begins to increasingly outperform the PROP algorithm to a significant extent (Welch's T-test, p=0.05). Intuitively, as the density of tasks in the environment increases, follower agents are more likely to find tasks in their influence radius (benefiting RW). Conversely, more tasks means more task information within each propagator agent, overloading and misguiding the follower agents during their decision process (harming PROP). This overload occurs as a result of having too many nearby tasks as choices for followers to head towards. Because followers decide a new direction of motion at each time step based on propagator agents' information, too many task options increases a follower's chance of indecision (choosing different tasks at each time step to head towards). After a certain point, too much propagated information results in that information declining in its specificity and thus usefulness.

Figure 4-5 shows the average number of messages sent per propagator agent per round (for the PROP algorithm). Note that on average, each propagator agent sends

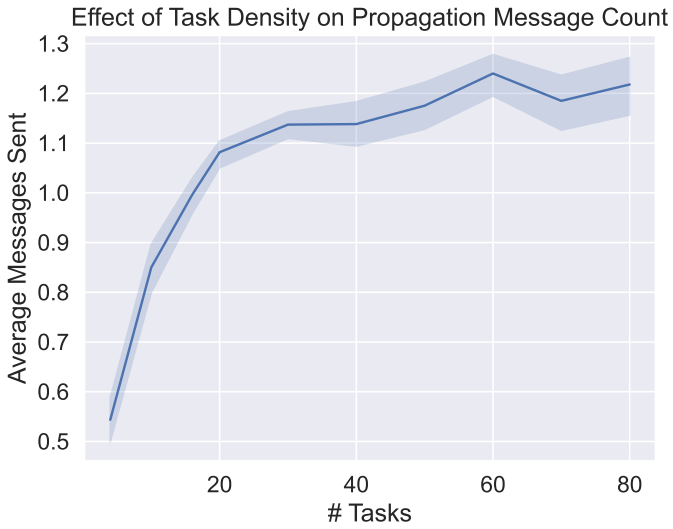

Figure 4-5: The effect of number of tasks on average messages sent per propagator agent per round (for PROP)

fewer than 1.3 messages to other agents per round; however, given the grid space's size, a large communication cost is still incurred as there are $2,500$ propagator agents. Regarding the effect of task density on these communication costs, agents generally send more messages as the number of tasks increases. When there are more tasks and thus more vertices close to tasks, it takes less time for most of the propagator agents to receive some information initially that they can begin propagating. Additionally, having more tasks means that *some* task's demand gets updated more often, resulting in there being new information (as messages) that needs to be propagated more often. This increasing trend becomes less dramatic at higher task densities, likely due to the fact that with enough tasks, the overlapping propagation radii all cover roughly the same area. As shown in Figure 4-4, recall that higher task densities result in higher completion times for the PROP algorithm. Therefore, higher task densities do not only result in agents sending more messages per round, but the total number of messages sent over an entire run increases even more dramatically along with the number of tasks.

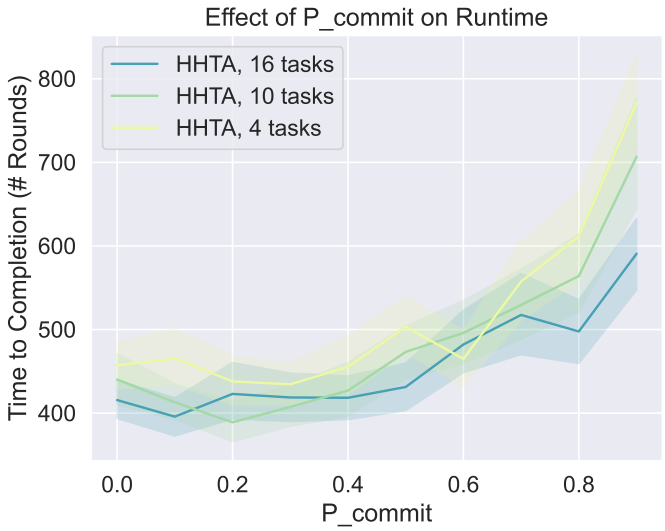

Figure 4-6: The effect of P_c on HHTA completion time for $\{4, 10, 16\}$ tasks

4.5.3 Effects of P_c on HHTA Completion Time for Varying Task Density

We explored the effects of varying P_c, the base probability of committing to a task instead of recruiting for it, on completion time for varying T. We ran 100 trials for each value of $P_c \in \{\frac{i}{10}, 0 \leq i \leq 9\}$ using $m_r = \frac{1}{6}$ and $P_e = \frac{2}{3}$. The results can be seen in Figure 4-6.

For $P_c \leq 0.7$, there was no significant difference between the HHTA completion time at different task densities (Welch's T-test, p=0.05). However, for $P_c \in \{0.8, 0.9\}$, the completion time for $T = 4$ is higher than the completion time for $T = 16$ (Welch's T-test, p=0.05). Our results show that only for large P_c do we see a significant difference in performance at different task densities. This makes sense, as a larger proportion of committing agents means agents mostly find tasks by discovering them independently, which is harder in the sparse case. We also note that from $P_c = 0.4$ to $P_c = 0.8$, task completion time follows an increasing trend, indicating that higher recruitment (lower P_c) allows agents to complete tasks faster.

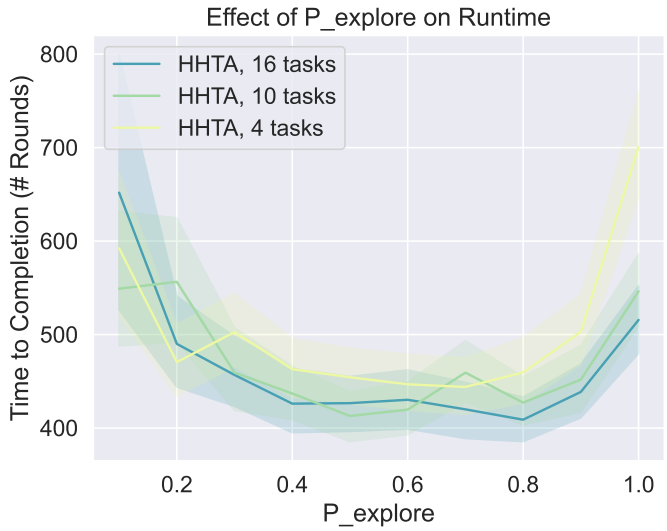

Figure 4-7: The effect of P_e on HHTA completion time for $\{4, 10, 16\}$ tasks

4.5.4 Effects of P_e on HHTA Completion Time for Varying Task Density

We also explored varying P_e, the probability of exploring vs. staying at home, running 100 trials each for $P_e \in \{\frac{i}{10}, 1 \leq i \leq 10\}$, with $T \in 4, 10, 16$ and using $m_r = \frac{1}{6}$ and $P_c = \frac{3}{10}$. The results can be seen in Figure 4-7.

For $P_e \leq 0.8$, there was no significant difference between the HHTA completion time at different task densities (Welch's T-test, p=0.05) with the exception of $T = 10$ vs. $T = 4$ at $P_e = 0.2$, with $p = 0.03$. However, there was a significant difference in completion time between $T = 4$ and $T = 16$ when $P_e = 1.0$ and $P_e = 0.9$. Our results show that HHTA has a consistent completion time regardless of task density other than for large $P_e \in \{0.9, 1.0\}$, meaning a large majority of agents are exploring (making the algorithm more similar to random walk). When P_e is high, it is harder to complete the sparse problem because exploring is harder in a sparse environment.

Our results also show that a more even balance of P_e (the proportion of Exploring agents) vs. $1 - P_e$ (the proportion of Home agents) leads to a faster completion time. When P_e is too low, not enough agents are exploring, making it harder to find tasks. When P_e is too high, not enough agents are available in the home nest to be recruited when tasks are found.

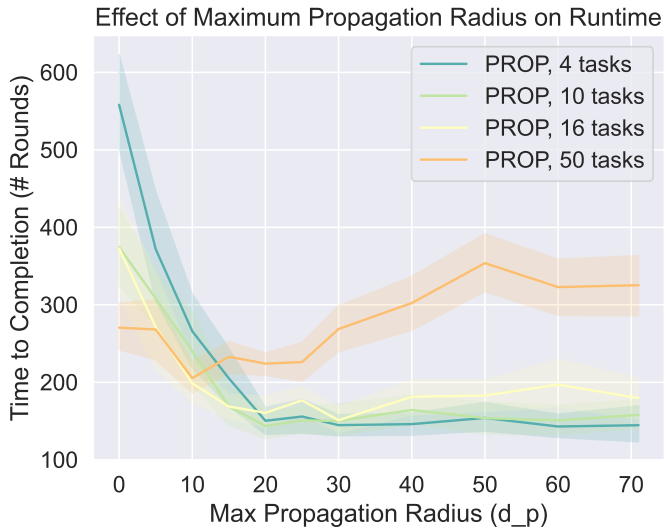

Figure 4-8: The effect of maximum propagation radius (d_p) on PROP completion time for $\{4, 10, 16, 50\}$ tasks

4.5.5 Effects of d_p on PROP Completion Time for Varying Task Density

We explored the effects of varying d_p, the maximum propogation radius, on completion time for varying T. We ran 20 trials for each unique pair of $d_p \in \{0, 5, 10, 15, 20, 25, 30, 40, 50, 60, 50\sqrt{2}\}$ (Note that $d_p = 50\sqrt{2}$ means all tasks' information can be propagated over the entire grid space) and $T \in \{4, 10, 16, 50\}$, using $t_p = 3$. The results can be seen in Figure 4-8.

For reasonably sparse task setups ($T \in \{4, 10, 16\}$), larger maximum propagation radii correlate with faster runtimes (Welch's T-test, p=0.05) but after a certain point, completion time is mostly unchanged. In contrast, for very dense task setups ($T = 50$), besides a slight improvement in completion time moving from around $d_p = 0$ to $d_p = 10$, larger maximum propagation radii result in slower completion times (Welch's T-test, p=0.05). Increasing d_p results in more propagator agents having more task information, which allows (1) for follower agents to find tasks even if they are far way and (2) for follower agents to leverage this extra task information to prioritize tasks with higher demand. This causes the initial decline in completion time for increasing d_p values. However, if d_p is too large, there is too much information being propagated, diluting the agents' strategy. This adverse effect is likely not

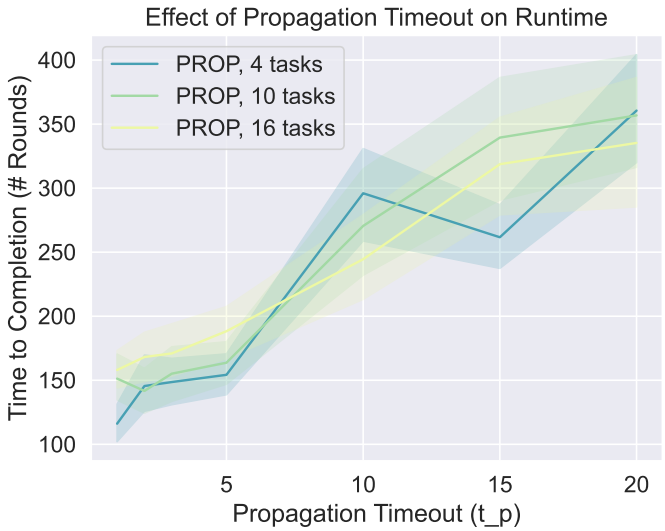

Figure 4-9: The effect of integer propagation timeout (t_p) on PROP completion time for $\{4, 10, 16\}$ tasks

seen with sparser environments because even with every single propagator having information about every single task, each mapping of task info is still bounded in size by this smaller number of tasks. It is also reasonable to infer that the turning point in each plot's trend (when completion time either becomes constant or starts increasing) is related to the d_p value at which every propagator agent receives *some* task information.

4.5.6 Effects t_p on PROP Completion Time for Varying Task Density

We explored the effects of varying t_p, the number of rounds a propagator must wait before sharing new task information with its neighbors, on completion time for varying T. We ran 20 trials for each unique pair of $t_p \in \{1, 2, 3, 5, 10, 15, 20\}$ and $T \in \{4, 10, 16\}$, using $d_p = 25$. The results can be seen in Figure 4-9.

There is a clear, mostly linear trend between t_p and completion time, where increasing the propagation timeout results in increasing completion times. The trend is fairly consistent across all distinct task densities that were tested. This relationship between t_p and completion time is to be expected, as smaller t_p means that task information is moved about the environment more quickly, causing the information that is

used by follower agents to decide which task to move towards to be more up-to-date. Besides at the very beginning, t_p has no effect on the locations of task information, so none of the adverse phenomena we have seen in which there is "too much" propagated information occur when varying t_p. It is the same task information, simply better when when the timeout is smaller. It is worth noting, though, that smaller values of t_p involve more message passing.

4.6 Discussion

Our results demonstrate for both HHTA and PROP that when the total demand for agents is held fixed, task density significantly affects algorithm performance. HHTA performs better than RW when tasks are very sparse, and worse when the number of tasks is high because communicating about individual tasks matters less when there are many of them (Figure 4-2). RW performs very poorly with sparse tasks because it becomes harder over time for the remaining agents to find tasks. PROP also performs better than RW until the number of tasks is very high, as agents struggle to arrive at tasks when too much task information is being propagated (Figure 4-4). PROP's completion time increases for very sparse tasks ($T \leq 6$), though it still outperforms RW and HHTA. We would also like to point out that PROP consistently outperforms HHTA for all task densities tested, which can be seen by comparing Figures 4-2 and 4-4. Though PROP has a faster completion time and is more distributed than HHTA, it is much more resource and communication intensive. This is because it requires a propagator agent at every grid cell in order to spread information.

In relevant task allocation problems such as search-and-rescue or mine detection, the number of tasks in the environment is expected to be sparse, so both algorithms provide a speed-up in completion time compared to the Levy walk. HHTA provides a less agent intensive and less communication intensive approach but requires a central communication location. Contrarily, PROP provides a quicker and more distributed approach for sparse and mid-density environments but is more resource-intensive. Since the Levy flight has been shown to optimize search efficiency and

can be observed in many species in nature, it makes sense that for very dense task environments with a low demand per task, the Levy flight outperforms both algorithms. Such environments are a very similar problem to foraging itself. On the other hand, environments with fewer tasks that require more agents benefit more from the coordination and communication of more advanced algorithms.

We also analyzed both algorithms' mechanics individually, showing the importance of recruitment in HHTA as well as the importance of an even balance of Exploring vs. Home agents. For PROP, we showed as expected that generally, higher d_p leads to better performance, though it is more communication-intensive. We also showed that as propagation timeout increases, time to completion increases, since task demands are stale for longer periods of time.

We also note that in extreme parameter settings, HHTA completion time was similar regardless of task density while varying algorithm parameters like P_c and P_e. However, this is untrue for PROP, which had a higher completion time for sparser environments at low d_p, and a higher completion time for denser environments at high d_p. This behavior makes sense because as d_p approaches 0, PROP reduces to RW, which is similarly affected with a higher completion time for sparse tasks.

4.7 Future Work

The environmental parameter we focused on varying was the number of tasks, while we kept the arena size and total task demand fixed. We kept total task demand fixed to ensure a fixed ratio of number of agents to total task demand even as we varied the task density (number of tasks). However, there are still many other environmental parameters that we could explore the effects of and use to evaluate HHTA and PROP. One example is varying the arena size while keeping the total task demand and number of agents fixed. We would expect it to be harder to find tasks in a larger arena compared to a smaller one. Another example is varying k, the fraction of agents needed to complete all tasks. Varying k could be done by fixing the number of agents but changing the total task demand. We would expect larger k to have a longer

completion time for both algorithms since a larger fraction of agents need to find and complete tasks.

Future work could explore experiments in a dynamic setting, where new tasks can appear over time and agents can search for a new task after their existing tasks are finished. It could also evaluate other environment parameters, such as the ratio of total task demand to total number of agents. A larger such ratio would make the task allocation problem harder to solve, as there are fewer and fewer extra agents available to communicate.

Future work could also combine the strengths of the PROP and HHTA algorithms, where one agent for each task is assigned to propagate by leaving information in the vertex state of nearby vertices or communicating task information directly to any nearby agents like HHTA does. This algorithm would have a much smaller agent cost than the PROP algorithm while still being able to propagate task information. It would also not require a central home nest like the HHTA algorithm does, instead opting to induce agent communication all around the arena.

The results we obtained were using a relatively small number of agents and tasks. Our general model allows for easy parallelization and we are exploring the possibility of running much larger experiments with hundreds of thousands of agents via parallelization [21]. We have used our parallel implementation in [21] already to run the density estimation algorithm in [35] on one hundred thousand agents and verify high-probability bounds on the algorithm. Evaluating our task allocation algorithms on a larger scale would be useful for understanding their scalability.

Lastly, future work could aim for analytical bounds on the expected task completion time of our two algorithms. Because the algorithms are relatively simple compared to many swarm algorithms, high probability bounds may be possible to obtain.

4.8 Formal Pseudocode

Now, we describe the details of the house hunting model in terms of pseudocode. Specifically, we reiterate the `Vertex State` used, which dictates the environment of both the PROP and HHTA algorithms. Then we reiterate the `Agent State` and specify the agent transition functions for the PROP and HHTA algorithms individually. Lastly, we specify the resolution rule needed for the task allocation problem.

4.8.1 Vertex State Class

In order to properly represent tasks in both of our algorithms, the vertex state set SV contains the following variables:

- `is_task`, whether or not the vertex is a task
- `demand`, the task demand if the vertex is a task; otherwise `0`
- `residual_demand`, the residual demand if the vertex is a task; otherwise `0`
- `task_location`, the x, y coordinates of the vertex if it is a task

4.8.2 HHTA Agent State Class

An agent state has the following constant variables in addition to the constants from the general model. Constants are fixed in an execution of the simulator for all agents. Each agent has these constants as a part of its state, and the values never change.

- `L`, defined as $L = 1/(M + N)$
- `P_COMMIT`, the probability P_c
- `P_EXPLORE`, the probability P_e
- `MESSAGE_RATE`, the message rate r_m
- `LEVY_LOC` and `LEVY_CAP`, levy flight distribution parameters, where `LEVY_CAP` caps the right tail of the distribution

Agents also have the following modifiable parts of their state, which may change during execution.

- `core_state`, the core state described above, which can be H, E, R or C
- `angle`, a random walk parameter denoting angle of travel
- `starting_point`, a random walk parameter tracking where the agent started from
- `travel_distance`, the length of the current leg of the random walk
- `destination_task`, the agent's destination if they have just found a task or are headed towards their committed task
- `home_destination`, the agent's destination if they are headed to a home vertex
- `recruitment_task`, the task an agent is recruiting for, initially `None`
- `committed_task`, the task an agent has committed to, initially `None`

4.8.3 HHTA Agent Transition Function

We now consider the generation of agent transitions at each time step for our HHTA algorithm. This can be broken down into different logic based on the four core states – Home, Exploring, Recruiting, or Committed. The helper functions used in the pseudocode for the transition functions can be found in Appendix B.

Home Agent Transitions

Home agents stay in the home nest to receive communications from other agents. In the Home agent transition function, we first check if the agent has just transitioned to this state and has not reached the home nest yet. If so, the agent continues moving in the direction of the home nest. If the agent is already at the home nest, it may receive a message via the states of other agents about a new discovered task, in which case it will either transition to Committed and try to do the task, or to the Recruiting state

and try to recruit for the task. Agents who have not heard of any task information have a chance of converting to exploring and searching for tasks. The pseudocode for Home agents can be seen in Algorithm 13.

Algorithm 13 Agent transition function α for a Home agent with state s at vertex v with coordinates (x, y)

```
1:  procedure GENERATE_TRANSITION(local_vertex_mapping)
2:      new_agent_state ← s
3:      ▷Agent just transitioned, headed home
4:      if s.home_destination is not null then
5:          new_direction ← dir_from_dest(
6:              s.home_destination, s.location.x, s.location.y))
7:          new_location ← coords_from_dir(
8:              s.location.x, s.location.y, new_direction)
9:          if within_home(new_location) then
10:             new_agent_state.home_destination ← null
11:             return s.location.state, new_agent_state, new_direction
12:     ▷Learn about new task
13:     if s.home_destination is null then
14:         task_state ← get_task_info(local_vertex_mapping, new_agent_state)
15:         if task_state is not null then
16:             committed_chance ← s.P_commit
17:             if random_float_from(0,1) < committed_chance then
18:                 new_agent_state.core_state ← Committed
19:                 new_agent_state.destination_task ← task_state
20:             else
21:                 new_agent_state.core_state ← Recruiting
22:                 new_agent_state.recruitment_task ← task_state
23:             return s.location.state, new_agent_state, S
24:     ▷Chance of converting to exploring
25:     c ← (1 − s.P_explore)/(s.P_explore)
26:     if c != 0 then
27:         explore_chance ← s.L/c
28:     else
29:         explore_chance ← 1
30:     if random_float_from(0,1) < explore_chance then
31:         new_agent_state.core_state ← Exploring
32:     return s.location.state, new_agent_state, S
```

Exploring Agent Transitions

Algorithm 14 Agent transition function α for an Exploring agent with state s at vertex v with coordinates (x, y)

```
1: procedure GENERATE_TRANSITION(local_vertex_mapping)
2:     new_agent_state ← s
3:     ▷Search for nearby tasks, has chance of converting to committed
4:     nearby_tasks ← find_nearby_tasks(local_vertex_mapping)
5:     if nearby_tasks != [] then
6:         sum_rd ← 0
7:         weights ← 0
8:         for task_state in nearby_tasks do
9:             sum_rd += task_state.residual_demand
10:            weights.append(task_state.residual_demand)
11:        for i in range(len(weights) do
12:            weights[i] /= sum_rd
13:        chosen_task ← choose_weighted(nearby_tasks, weights)
14:        committed_chance ←
15:            max(s.P_commit, 1/chosen_task.residual_demand)
16:        if random_float_from(0,1) < committed_chance then
17:            new_agent_state.core_state ← Committed
18:            new_agent_state.destination_task ← chosen_task
19:        else
20:            new_agent_state.core_state ← Recruiting
21:            new_agent_state.home_destination ← random_loc_in_home()
22:            new_agent_state.recruitment_task ← chosen_task
23:        return s.location.state, new_agent_state, S
24:    ▷Chance of converting to home
25:    home_chance ← s.L
26:    if random_float_from(0,1) < home_chance then
27:        new_agent_state.home_destination ← random_loc_in_home()
28:        new_agent_state.core_state ← Home
29:        return s.location.state, new_agent_state, S
30:    ▷Keep random walking
31:    new_direction ← get_travel_direction(new_agent_state)
32:    return s.location.state, new_agent_state, new_direction
```

Algorithm 14 describes the transition function for an exploring agent. Exploring agents first look within their influence radius for tasks. If tasks are found, they have a probability of choosing each task weighted by residual demand. After a task is chosen, the agent either becomes Committed or Recruiting for that task. If an

exploring agent has not found any tasks, it has a chance of converting to a Home agent. Otherwise, it random walks in search of tasks.

Recruiting Agent Transitions

Algorithm 15 describes the transition function for a recruiting agent. We first check if the recruiting agent has reached the home nest to recruit yet. If they haven't, they keep heading towards the home nest by stepping one step in that direction. If they have reached the home nest, they have a $1/t_i^{rd}$ chance of transitioning to the Committed core state. Otherwise, they remain in the Recruiting state.

Algorithm 15 Agent transition function α for a Recruiting agent with state s at vertex v with coordinates (x, y)

```
1: procedure GENERATE_TRANSITION(local_vertex_mapping)
2:     new_agent_state ← s
3:     if s.home_destination is not None then
4:         new_direction ←
5:             dir_from_dest(s.home_destination, x, y)
6:         new_location ←
7:             coords_from_dir(x,y,new_direction)
8:         if within_home(new_location) then
9:             new_agent_state.home_destination ← None
10:            return v.state, new_agent_state, new_direction
11:    committed_chance ←
12:        1/s.recruitment_task.residual_demand
13:    if random_float_from(0,1) < committed_chance then
14:        new_agent_state.core_state ← Committed
15:        new_agent_state.destination_task ←
16:            new_agent_state.recruitment_task
17:        new_agent_state.recruitment_task ← None
18:        return v.state, new_agent_state, S
19:    return s.location.state, new_agent_state, S
```

Committed Agent Transitions

Algorithm 16 describes the transition function of a Committed agent. If the agent is still heading towards the task it committed to, it continues moving towards the task.

If the agent has arrived at the task and discovers it to already be full (with residual demand 0), it returns to being an exploring agent. Otherwise, it tries to complete the committing process by decrementing the task's residual demand. Otherwise, the committed agent has already committed to a task and merely remains at the task for the rest of the algorithm.

Algorithm 16 Agent transition function α for a Committed agent with state s at vertex v with coordinates (x, y)

```
1:  procedure GENERATE_TRANSITION(local_vertex_mapping)
2:      new_agent_state ← s
3:      if s.destination_task is not None then
4:          ▷Agent arrived at committed task after recruiting)
5:          if s.location.coords() == s.destination_task.task_location then
6:              ▷Task is full, continue exploring
7:              if s.location.state.residual_demand == 0 then
8:                  new_agent_state.destination_task ← null
9:                  new_agent_state.core_state ← Exploring
10:                 return s.location.state, new_agent_state, S
11:             ▷Try to commit to task
12:             new_agent_state.committed_task ← s.destination_task
13:             new_agent_state.destination_task ← null
14:             new_vertex_state ← s.location.state
15:             new_vertex_state.residual_demand ←
16:                 s.location.state.residual_demand-1
17:             return new_vertex_state, new_agent_state, S
18:         else
19:             ▷Head towards committed task after recruiting
20:             new_direction ← dir_from_dest(
21:                 s.destination_task.task_location, s.location.coords()
22:             return s.location.state, new_agent_state, new_direction
23:     else
24:         ▷Once committed, stay committed
25:         return s.location.state, new_agent_state, S
```

4.8.4 PROP Agent Transition Function

We now consider the agent transition functions for follower and propogator agents independently. The helper functions used in the pseudocode for the transition functions

can be found in Appendix B.

Propagator Agent Transitions

The propagator agent transitions can be seen in Algorithm 17. If the propagator agent is at a task, it first checks if the residual demand information has changed and stores that new information in the message that it will later propagate. Then, for all propagator agents, if they are allowed to propagate at this round (agents can only propagate every `propagation_rate` rounds), then they do so. Otherwise, they keep waiting until they reach a round where they can propagate.

Algorithm 17 Agent transition function α for a propagator agent with state s at vertex v with coordinates (x, y)

```
1: procedure GENERATE_TRANSITION(local_vertex_mapping)
2:     new_agent_state ← s
3:     if v.state.is_task then
4:         new_agent_state.task_info[(x, y)] ←
5:             v.state.residual_demand
6:     if s.propagation_ctr >= s.propagation_rate then
7:         self.propagate()
8:         new_agent_state.propagation_ctr ← 0
9:     else
10:        new_agent_state.propagation_ctr ←
11:            new_agent_state.propagation_ctr + 1
12:    return v.state, new_agent_state, S
```

Follower Agent Transitions

The follower agent transitions can be seen in Algorithm 18. First, if the follower agent is still looking for a task, it scans the propagator agent of the vertex it is located at to find nearby tasks. If a task is found, it heads toward that task. Otherwise, it moves in a random direction. If the follower agent has arrived at the destination task and the residual demand is 0, it stops committing to that task. Otherwise, it tries to commit to the task and will be accepted as long as others are not trying to commit to the same task at the same time (if this is the case, winners will be chosen by our resolution function).

Algorithm 18 Agent transition function α for a follower agent with state s at vertex v with coordinates (x, y)

```
1: procedure GENERATE_TRANSITION(local_vertex_mapping)
2:     new_agent_state ← s
3:     if s.committed_task is None then
4:         if s.destination_task is None then
5:             if find_nearby_task(v) is not None then
6:                 new_agent_state.destination_task ←
7:                     find_nearby_task(v)
8:                 return v.state, new_agent_state, S
9:             else
10:                return v.state, new_agent_state,
11:                    dir_from_propagator()
12:        else
13:            if s.destination_task.state.residual_demand is 0 then
14:                new_agent_state.destination_task ← None
15:                return v.state, new_agent_state, S
16:            if s.destination_task is v then
17:                new_agent_state.committed_task ←
18:                    s.destination_task
19:                new_agent_state.destination_task ← None
20:                new_vertex_state ← v.state
21:                new_vertex_state.residual_demand ←
22:                    new_vertex_state.residual_demand − 1
23:                return new_vertex_state, new_agent_state, S
24:            else
25:                return v.state, new_agent_state,
26:                    dir_to((x,y), s.destination_task)
27:    else
28:        return v.state, new_agent_state, S
```

4.8.5 Resolution Rule

In the task allocation problem, agents cause changes to the vertex states of tasks when they commit to them, by decrementing the task's residual demand. This creates the potential for conflicts if a task has current residual demand rd but $x > rd$ agents are trying to claim the task at the same time. In this case, an arbitrary subset of the x

agents should be allowed to move forward and commit to the task, whereas the other $rd - x$ agents will not be allowed to commit. The new residual demand is now $rd' = 0$ (Note that if $x < rd$, all x agents should be allowed to commit and the new residual demand $rd' = rd - x$.) Our resolution rule accomplishes this by looking at all agents who wish to commit in this round (being careful to ignore agents who have already committed to the task in earlier rounds) and arbitrarily choosing a subset of them as the "winners" who get to commit to the task. The pseudocode for this task-claiming resolution rule can be found in Algorithm 19.

Algorithm 19 Resolution rule for agents trying to claim tasks in a task allocation problem

```
1: procedure TASK_CLAIMING_RESOLUTION(proposed_vertex_states, proposed_agent_states, proposed_agent_dirs))
2:     if not vertex.state.is_task then
3:         ▷No resolution needed for non-task vertices
4:         return naive_resolution(proposed_vertex_states,
5:             proposed_agent_states, proposed_agent_dirs)
6:     agents ← list(proposed_agent_states.keys())
7:     contenders ← []
8:     available_slots ← vertex.state.residual_demand
9:     attempted_claims ← 0
10:    for agent_id in agents do
11:        ▷Check if agent is trying to claim the task
12:        if proposed_vertex_states[agent_id].residual_demand == vertex.state.residual_demand-1 then
13:            attempted_claims += (vertex.state.residual_demand -
14:                proposed_vertex_states[agent_id].residual_demand
15:            contenders.append(agent_id)
16:    if attempted_claims <= available_slots then
17:        ▷Everyone allowed to claim task
18:        vertex.state.residual_demand← available_slots - attempted_claims
19:        return vertex.state, proposed_agent_states, proposed_agent_dirs
20:    else
21:        ▷Choose winners to claim task
22:        new_proposed_agent_states ← {}
23:        winners ← choose_n_from(contenders)
24:        for agent_id in agents do
25:            new_agent_states[agent_id] ←
26:                proposed_agent_states[agent_id]
```

Algorithm 19 Task allocation resolution (continued)

```
27:             if agent_id not in winners and agent_id in contenders then
28:                 new_agent_states[agent_id].committed_task ← None
29:                 new_agent_states[agent_id].destination_task ← None
30:                 if algorithm is HHTA then
31:                     new_agent_states[agent_id].core_state ← Exploring
32:         vertex.state.residual_demand ← 0
33:         return vertex.state, new_agent_states, proposed_agent_dirs
```

Note that the resolution rule slightly differs between HHTA and PROP in that if an HHTA agent tries to claim a task but fails, it's state is changed from the Committed state to the Exploring state. For PROP, by setting the agent's `committed_task` to None, it will automatically resume exploring.

Chapter 5

Future Work

This chapter describes our work-in-progress in addition to some new algorithmic ideas that could be implemented within our general model. Chapter 5.1 discusses our ongoing work on a parallel implementation of our general model in C++. Chapter 5.2 presents some ideas for swarms being applied to environmental problems such as pollination and oil spill cleanup. Chapter 5.3 presents an algorithmic sketch for task allocation in unknown environments which is inspired by the centralized min-cost flow solution to the Optimal Mass Transport problem.

5.1 Simulation Speedups via Parallelization

Our model is highly distributed on individual vertices and easily lends to parallelization. We are in the process of developing PAR-SWARM, a C++ parallel implementation of our modelling framework, which provides significant speedups to large-scale simulations (on the order of 10^6 vertices and agents or larger). These large-scale simulations can be useful when demonstrating high probability bounds on agent behavior such as the density estimation work in [35]. These demonstrations can require a large number of agents, and parallelization makes simulations much easier and faster to conduct.

We have a manuscript in progress describing our parallelization work [21]. Zhi Wei Gan wrote the parallel simulator and is evaluating it. Julian Shun provided advice on

parallelization techniques, and Grace Cai and Nancy Lynch provided details about the original general model.

5.2 Applications of Swarms to the Good of the Earth

Robot swarms are well suited to tackle far-reaching and complex tasks for the good of the environment. Many swarm robots and swarm prototypes have been developed with the goals of pollinating in the place of bees, cleaning up oil spills in the ocean, and more [11, 34]. However, many of these designs focus on tackling the physical aspects of designing these robots rather than the algorithmic ones. For example, [11] seeks to develop robotic bees (RoboBees) to enable pollinating behavior, but focuses on the necessary mechanical components such as mechanisms for swimming, flying, and perching on surfaces. Another example is [34], which develops a design for a swarm robot that can skim oil off the surface of the ocean. While these physical mechanisms are necessary and simple random-walk algorithms could suffice on the coordination front, more advanced algorithms could save time and swarm resources.

Some existing work has been done algorithmically, with [2] getting a simulated swarm to surround the perimeter of an oil spill and [15] introducing a programming language for RoboBees with an example algorithm of random walk alfalfa monitoring and pollination. However, more effective algorithms could be developed given more precise problem statements that seek to minimize completion time or other metrics of evaluation.

One problem that would be interesting to explore is developing an algorithm that not only successfully surrounds the oil spill, but then cleans it up as efficiently as possible. Agents should maximize the percentage of the oil spill that they collectively clean while also trying to complete the process quickly. Another interesting problem would be to design an artificial pollination algorithm that outperforms the random walk presented in [15], where the goal for agents would be to pollinate as many flowers as possible in a fixed arena within a fixed period of time. More complicated variants of the pollination problem could consider multiple species of plants at the same time

and try to develop an algorithm that fairly pollinates all species according to a desired pollination rate.

Developing algorithms for the problem statements we have suggested in artificial pollination and oil spill cleanup is a promising step for future work within our general modelling framework. The discrete nature of our model along with its modifiable vertex states are beneficial to testing and developing environment-based algorithms. Our model would make it simple to set up complex environments such as irregular oil spill shapes or flower locations for pollination.

5.3 A Swarm Task Allocation Algorithm Inspired by Centralized Min Cost Flow

As we have mentioned in Chapter 4, one centralized solution for getting a group of agents to tasks with known locations and demands is to build a graph with vertices at the task locations and the agents' starting location. We can then run a min cost flow algorithm on this graph with the agent locations connected to a super-source and the task locations connecting to a super-sink. The min cost flow algorithm determines how agents should travel in order to minimize the total distance travelled by all agents.

The type of task allocation problem that this algorithm solves differs from the scenario proposed for our HHTA and PROP algorithms in several ways. First, it assumes known task locations and demands. Second, it seeks to minimize the total distance travelled by all agents as opposed to the fastest time it tasks for all tasks to be completed. The metric of minimizing total distance travelled is more relevant when fuel or other resource costs for agents are important to reduce.

We propose an idea for a distributed version of this algorithm in the situation where tasks locations and demands are initially unknown. The distributed version of this algorithm makes use of propagator agents similar to our PROP algorithm in Chapter 4. Future work could evaluate the completion time and messages sent in this

algorithm and compare it to that of PROP and HHTA. Note that our distributed version no longer minimizes total distance travelled by agents, as it includes extra steps at the beginning to discover task information and share it with agents.

5.3.1 Algorithm Description

In the centralized algorithm (CENT), we assume two types of agents – propagator agents and follower agents. Propagator agents are simple, mote-like agents [49] and one is assigned to each vertex. Follower agents perform tasks and will rely on propagator agents to direct them where to go. We assume that follower agents all start in a home nest at the center of the arena, similar to HHTA and PROP. Suppose there are T tasks, and task i has demand a_i.

Phase 1 (Propagator agents share information): Propagator agents spread their task's location in the same manner as they do in the PROP algorithm but do not spread any demand information. Follower agents wander the grid and head towards the first task they sense from a propagator agent's information. (If the agent hears of multiple tasks at once, it randomly chooses a task). After some time, each follower agent is located at one of the tasks in the grid. We assume this fact for the rest of the algorithm. Suppose that there are r_i followers located at task i. Of the r_i agents, as many followers as possible should be assigned to a task at their starting site i. Formally, for every task i, $\min(r_i, a_i)$ agents should automatically be assigned to task i. Tasks where $r_i > a_i$ we call *sources* because they have excess agents that can be distributed elsewhere. Tasks where $r_i < a_i$ we call *sinks* because they still need more agents to arrive. Tasks where $r_i = a_i$ have their demand satisfied and are neither sources nor sinks. The graph now contains S sources with $r_i - a_i$ agents and T sinks with $a_i - r_i$ agents needed, where $S + T < N$. (This problem of transporting supplies to multiple demanders is more generally known as the Optimal Mass Transport problem [52]). Thus we have reduced our problem to finding how to distribute the agents at the sources to the tasks at the sinks while minimizing total distance travelled.

By the end of Phase 1, each propagator agent knows whether its task is a source,

a sink, or neither. If the task is a source, the agent knows $r_i - a_i$ (how many leftover agents there are). If the task is a sink, the agents know $a_i - r_i$ (the residual demand).

Phase 2 (Learning all Sinks / Sources): Note that the tasks form a complete graph G_T, an overlay network on our model grid G. Tasks a and b are connected in G_T with weight $w(a, b)$ equal to the taxicab distance between l_a and l_b. We wish for the propagator agent at each task to know the full graph G_T.

In order to do so, each propagator agent spreads its task surplus and demand to its neighbors within one influence radius. Whenever a propagator agent receives information about a new task, it in turn propagates that information to it's own neighbors. In this way, after $N + M$ rounds, all propagator agents will know the full graph G_T.

Phase 3 (Run Min-Cost Flow): Since every propagator at a task knows the location of each task, each propagator can independently build the graph G_T. Distributing the agents from sources to sinks is now an instance of the min cost flow problem. To solve this problem we connect a super-source s to each source s_j with capacity equal to the number of agents at s_j, and connect each sink t_j to a super-sink t with capacity equal to the number of agents needed at t_j. Call the graph with the addition of the super-source and super-sink G'_T. The flow we wish to send over G'_T is equal to the total remaining demand from the sinks. We let all propagators located at tasks use cycle cancelling or other min cost flow algorithms to determine the optimal flow of agents through each edge. (Note that here, optimal means minimizing the total distance travelled by all agents). By the end of Phase 3, all propagator agents at tasks should have the same optimal flow graph.

Phase 4 (Send Agents to Sinks): After all propagators at tasks have computed the optimal flows, any propagator which is at a source node looks at its outgoing edge flows and sends its agents in the correct distribution along those edges. Because the edge weights in G_T follow the triangle inequality and G_T is complete, the flow should go directly from the sources to the sinks in one hop (it will always be more efficient to go directly from a source to a sink than via any intermediate node). At this point all tasks should have their demands satisfied, and the allocation is complete.

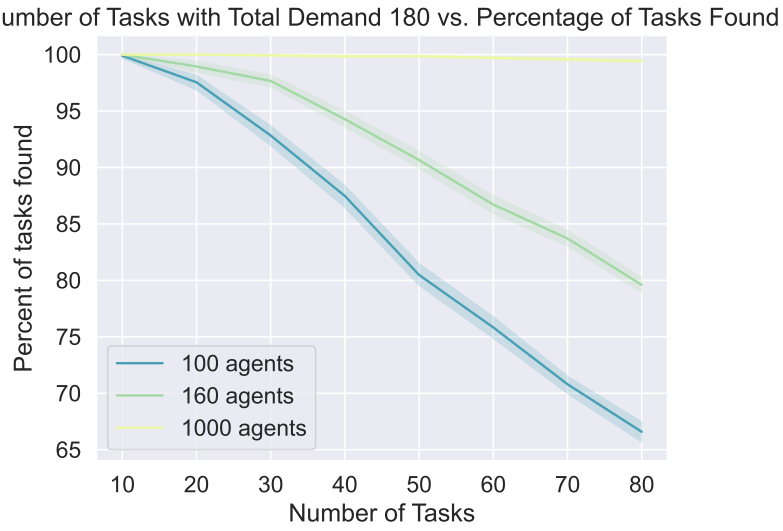

Figure 5-1: Plot of percentage of tasks found via initial random walk for varying task density and varying numbers of agents

Note we assume that propagator agents have advanced enough computation to perform a min cost flow algorithm. Furthermore, while the algorithm can tolerate failures in propagator agents not located at tasks as long as the propagator agents remain connected, it cannot tolerate failures in the propagator agents located at tasks themselves, since they are each performing a centralized computation of min-cost flow. More detailed analysis and evaluation of this algorithm is left to future work.

5.3.2 Motivation for Propagator Agents

An alternative idea that was initially considered for Phase 1 of our centralized algorithm was to only have one type of agent, have them initially do a random walk to find the first task they see, and cluster agents at tasks in that way. Then one agent at each task would be chosen as a leader, and the leaders would try to share their task's information with the agents located at every other task. However, the assumption that at least one agent would end up at each task is not very realistic, especially when the number of agents is to the number of tasks in the arena.

Figure 5-1 shows the percentage of tasks found by agents using this random walk strategy in a $N = 50, M = 50$ grid where there were $T = \{10, 20, 30, 40, 50, 60, 70, 80\}$ tasks and the total task demand was 80. We tested this strategy using $|R| =$

$\{100, 160, 1000\}$ agents. Note that when other than at very sparse task densities, not all tasks were found when using 100 agents or 160 (twice the task demand) agents. Only by using 1000 task performing agents would we see almost 100% of tasks being found no matter the task density.

The inability to guarantee that all tasks are found before starting the centralized task allocation portion of the algorithm motivates our use of propagator agents. Using propagator agents, one agent is always at each task and this all tasks are made known to other propagator agents before running the min cost flow algorithm. Having the propagator functionality separated out also makes it so that we only need the number of follower agents to be equal to T to satisfy task demand. This way propagator agents can be simpler and have the sole roles of sharing information and computing the min flow.

Chapter 6

Conclusion

This thesis first presents our new discrete geometric swarm model, and then dives into two main projects in which we have used our model as an algorithmic framework. In the first project, presented in Chapter 3, we present a more geographically robust N-site selection algorithm compared to previous work [44, 13]. Specifically, our model is able to accurately select the best quality nest even when it is disadvantaged by being further away from the home nest, or by being blocked by poorer, lower quality nests. We showed that the use of a quorum threshold which scales on site quality is particularly useful in more challenging site setups. Our results show the usefulness of our general model in easily testing areas with different site setups and provide an N-site selection algorithm that is more effective than past work at dealing with these new setups. Furthermore, our work also poses questions to the ant research community as to whether quorum sensing ants also employ a variable quorum threshold in their decision making.

Our second project, presented in Chapter 4, presents two new task allocation algorithms meant for dealing with environments where task locations and demands are initially unknown. We evaluate these algorithms with varying task densities in our arena and discover that both algorithms, as well as the Levy walk we are comparing them to, have varying performance with different task density. We draw inspiration from the fields of virtual pheromones to introduce one algorithm using propagator agents to spread task information. We draw inspiration from house hunting to intro-

duce another algorithm which uses agent communication in a home nest to spread task information. When keeping total task demand fixed, we find that our house-hunting based algorithm (HHTA) is effective on small task densities. Our propagation-based algorithm (PROP) is effective in all but extremely high task densities, but requires many more messages sent between propagator agents. At very high task densities, the Levy flight outperforms both of our algorithms since the task demand is very small. A small task demand of one or two agents makes the problem very similar to foraging, which the Levy flight has been theorized to be optimal for [54]. Our results again highlight the usefulness of our model in evaluating algorithms, as our ability to easily configure the task density within our model revealed how algorithm performance depends on task density.

We supplement our general model as well as all of the new algorithms we present with detailed pseudocode at the end of the relevant chapters and hope that it is useful for both clarity and reproducibility purposes.

Lastly, we detail ongoing work on our model in the form of parallelization using C++, which will allow for running simulations much faster and at a much larger scale. We also include some ideas for environmental problems where our model could be employed to develop swarm solutions, as well as a sketch of a new centralized task allocation algorithm that could be implemented in our model.

We hope that our model and the results we have obtained in house hunting and task allocation inspire further geometric swarm research.

Appendix A

N-Site Selection Utility Functions

A.1 Utility Functions for Agent Transition Function With Definitions

We include a few commonly used utility functions with definitions because they contribute to modifying the agent state and should be defined for completeness.

The function `check_quorum_sensed(local_vertex_mapping)` first calls the `quorum_sensed(local_vertex_mapping)` to see if a quorum site has been found. It then returns a boolean representing whether the quorum has been sensed, and a `new_agent_state` representing if a quorum has been sensed. The new agent state has a 50/50 chance of being either a Quorum Nest or Quorum Active state. The Quorum Nest agent state is set up for the agent to travel back to the home nest, while the Quorum Active agent state is set up for the agent to perform a random walk around the environment. The pseudocode can be seen in Algorithm 20.

The function `committed_agent_state_and_dir(s)` calculates the proper new agent state and direction for a committed agent returning to it's committed site for the final time. Here, `s` is the current agent state. If the agent has just arrived at the committed site, the agent terminates the house hunting algorithm. If the agent has just decided to move to the committed site or is already headed towards it, this function returns the direction the agent needs to move and any state modifications

Algorithm 20 Determining agent state transitions after quorum is sensed for agent a

```
1: procedure CHECK_QUORUM_SENSED(local_vertex_mapping)
2:     quorum_site ← quorum_sensed(local_vertex_mapping)
3:     s ← a.state
4:     if quorum_site is not null then
5:         s.quorum_site ← quorum_site
6:         if random_float_from(0,1) < 0.5 then
7:             s.destination ← random_location_in_site(s.home)
8:             s.destination_site ← s.home
9:             return True, s
10:        else
11:            s.travel_distance ← int(1/s.L)
12:            s.angle ← random_float_from(0,2π)
13:            s.starting_point ← s.location.x, s.location.y
14:            s.destination ← null
15:            s.destination_site ← null
16:            return True, s
17:    return False, null
```

(for example, setting the committed site as the agent's destination site). The code for how this function works can be seen in Algorithm 21.

Algorithm 21 Determining agent state and direction for agents a moving to the committed site

```
1: procedure COMMITTED_AGENT_STATE_AND_DIR(s)
2:     new_agent_state ← s
3:     if s.destination_site == s.quorum_site then
4:         ▷Agent arrived at committed site
5:         if s.location.x == s.destination[0] and s.location.y ==
   s.destination[1] then
6:             new_agent_state.destination ← null
7:             new_agent_state.destination_site ← null
8:             new_agent_state.terminated ← True
9:             return new_agent_state, S
10:        else
11:            ▷Agent still headed towards committed site
12:            new_direction ← get_direction_from_destination(
13:                s.destination, (s.location.x, s.location.y))
14:            return new_agent_state, new_direction
```

Algorithm 21 Determining agent state and direction for agents a moving to the committed site (cont.)

```
    ▷Agent just finished broadcasting, should return to committed site
    if s.travel_distance == 0 then
        new_agent_state.destination ←
            random_location_in_site(s.quorum_site)
        new_agent_state.destination_site ← s.quorum_site
        return new_agent_state, S
```

The function `should_abandon_site(local_vertex_mapping, s)` calculates whether a favoring nest agent with agent state s should stop favoring a site based on when the last time it saw a neighbor was. It returns a pair of a boolean, indicating whether or not to abandon the favored site, and an agent state. If the time since a neighbor was last seen exceeds $5/L$ time steps, we decide to abandon the site and have the agent transition to Uncommitted Active. The specifics can be seen in Algorithm 22.

Algorithm 22 Determining whether a favoring agent a should abandon its site

```
1:  procedure SHOULD_ABANDON_SITE(local_vertex_mapping, s)
2:      new_agent_state ← s
3:      ▷Update time since last neighor was seen
4:      if num_neighbors(local_vertex_mapping) > 1 then
5:          new_agent_state.time_since_neighbor ← 0
6:          ▷Abandon the favored site
7:      else
8:          new_agent_state.time_since_neighbor += 1
9:      if new_agent_state.time_since_neighor >= 5/s.L then
10:         new_agent_state.angle ← 0
11:         new_agent_state.starting_point ←
12:             (s.location.x, s.location.y)
13:         new_agent_state.travel_distance ← 0
14:         new_agent_state.destination ← null
15:         new_agent_state.destination_site ← null
16:         new_agent_state.exploring_site ← False
```

Algorithm 22 Determining whether a favoring agent a should abandon its site (cont.)

```
17:         new_agent_state.exploration_cooldown ← 0
18:         new_agent_state.favored_site ← None
19:         new_agent_state.time_since_neighbor ← 0
20:         new_agent_state.preference_type ← Uncommitted
21:         new_agent_state.activity_type ← Active
22:         return True, new_agent_state
23:     else
24:         return False, s
```

A.2 Utility Functions for Agent Transition Function With I/O Specifications

Utility functions are functions that we will use in the agent pseudocode to help with certain computations but will not offer the pseudocode for as they should be simple enough to derive. We specify the input and output of these utility functions, which will be referenced in the remainder of the pseudocode. Note that all utility functions are implemented within the agent state and have access to current agent state variable values.

- `quorum_sensed(local_vertex_mapping)` is a function that looks at the neighboring squares in the local vertex mapping and checks if a quorum is detected. If so, it returns the vertex state of the site it sensed a quorum for. If not, it returns `null`. Remember that we detect a quorum for a site based on whether the number of neighbors within a site if the number of agents there surpasses the scaled quorum threshold (based on site value), or if we see a committed agent who has already detected quorum for a specific site.

- `get_coords_from_movement(x,y,dir)` takes in the agent's current x-y coordinates `x` and `y` as well as the intended direction of travel `dir` $\in \{L, R, D, U, S\}$ and returns the new corresponding x-y coordinates if the agent moves one step in direction `dir`. The coordinates are returned as a tuple $[x, y]$.

- `within_site(x,y,site)` takes in the agent's x-y coordinates and the vertex state representing a site to check whether the agent is within the site or not. It

returns true if the agent is inside the site, and false otherwise.

- `random_float_from(a,b)` generates a uniformly random float in the range $[a, b]$

- `find_nearby_site(local_vertex_mapping)` is a function that looks at the neighboring squares in the local vertex mapping and checks to see if any site is visible amongst them. It takes the first site it finds and returns the vertex state of the site it found, and the location ((x, y) coordinates) of the vertex that it found to be within a site.

- `find_better_site(local_vertex_mapping)` is a function for favoring agents that looks at the local vertex mapping and checks to see if there are other favoring agents who are favoring a higher quality site. If so, it returns the vertex state of the higher quality site. Otherwise, it returns null.

- `num_neighors(local_vertex_mapping)` is a function that looks at the local vertex mapping and counts the number of visible agents (neighbors) in the local radius. It returns this count as an integer.

- `get_direction_from_destination(destination, curr_loc)` is a function that takes in the agents current x-y location `curr_loc` and the agent's `destination` (also in x-y coordinates) and outputs the direction in the set $\{L, R, D, U, S\}$ that will bring the agent closest to their destination.

- `get_travel_direction(new_agent_state)` is a function that computes the next step of agent random walks, and makes sure that agents do not go out of bounds. The function uses the random walk parameters $travel\_distance, angle$, and $starting\_point$ to determine which direction the agent should head in next. If the random walk runs the agent into the edge of the grid, we regenerate a new angle of travel for the random walk. If an agent is currently within a site, the boundaries for the random walk become the site boundaries instead. The function outputs the new travel direction, in the set $\{L, R, D, U\}$ and the modified agent state with the new random walk parameters in place.

Appendix B

HHTA and PROP Utility Functions

Utility functions are functions that we will use in the agent pseudocode to help with certain computations but will not offer the pseudocode for as they should be simple enough to derive. We specify the input and output of these utility functions, which will be referenced in the remainder of the pseudocode. Note that all utility functions are implemented within the agent state and have access to current agent state variable values.

B.1 HHTA Utility Functions

The following utility functions are used in the HHTA pseudocode.

- `dir_from_dest(destination, curr_loc)` is a function that takes in the agents current x-y location `curr_loc` and the agent's `destination` (also in x-y coordinates) and outputs the direction in the set $\{L, R, D, U, S\}$ that will bring the agent closest to their destination.

- `coords_from_dir(x,y,dir)` takes in the agent's current x-y coordinates `x` and `y` as well as the intended direction of travel `dir` $\in \{L, R, D, U, S\}$ and returns the new corresponding x-y coordinates if the agent moves one step in direction `dir`. The coordinates are returned as a tuple $[x, y]$.

- `within_home(location)` takes in the agent's location (a tuple of x,y coordinates) to check whether the agent is within the home location or not. It returns true if the agent is inside the site, and false otherwise.

- `random_float_from(a,b)` generates a uniformly random float in the range $[a, b]$

- `get_travel_direction(new_agent_state)` is a function that computes the next step of agent random walks, and makes sure that agents do not go out of bounds. The function uses the random walk parameters $travel\_distance, angle$, and $starting\_point$ to determine which direction the agent should head in next. If the random walk runs the agent into the edge of the grid, we regenerate a new angle of travel for the random walk. If an agent is currently within a site, the boundaries for the random walk become the site boundaries instead. The function outputs the new travel direction, in the set $\{L, R, D, U\}$ and the modified agent state with the new random walk parameters in place.

- `choose_weighted(tasks, weights)` takes in a list of tasks and weights and returns one task using a weighted probability according to the weights provided.

- `find_nearby_tasks(vertex)` looks within the influence radius of a vertex and finds all tasks, returning them as a list.

- `get_task_info(local_vertex_mapping, new_agent_state)` looks within the influence radius of a vertex and gathers messages from any agents who are recruiting for a particular task. As soon as a Recruiting agent is found, the function returns the task the agent is recruiting for.

B.2 PROP Utility Functions

The following utility functions are used in the PROP pseudocode.

- `find_nearby_task(vertex)` looks within the influence radius of a vertex and finds all tasks, returning them as a list

- `dir_to(location, destination_task)` computes and returns the direction an agent should travel to head to a given destination task

- `dir_from_propagator()` looks at the propagator agent of the vertex an agent is located at and uses the propagator agent's task information to choose a task to head towards and compute the direction the agent needs to head. If the propagator has no task information, this function returns a random direction.

Bibliography

[1] Athula B Attygalle and E David Morgan. Ant trail pheromones. In *Advances in insect physiology*, volume 18, pages 1–30. Elsevier, 1985.

[2] Fidel Aznar, Mireia Sempere, Mar Pujol, R Rizo, and MJ Pujol. Modelling oil-spill detection with swarm drones. In *Abstract and Applied Analysis*, volume 2014. Hindawi, 2014.

[3] Eric J Barth. A dynamic programming approach to robotic swarm navigation using relay markers. In *Proceedings of the 2003 American Control Conference, 2003.*, volume 6, pages 5264–5269. IEEE, 2003.

[4] Spring Berman, Adám Halász, M Ani Hsieh, and Vijay Kumar. Optimized stochastic policies for task allocation in swarms of robots. *IEEE transactions on robotics*, 25(4):927–937, 2009.

[5] Mickaël Bettinelli, Michel Occello, and Damien Genthial. Coalition formation problem: a group dynamics inspired swarming method. In *International Conference on Swarm Intelligence*, pages 282–289. Springer, 2020.

[6] Grace Cai. Geometric Swarm Modelling, January 2023. Available at https://github.com/ssgcai/geo-swarm.

[7] Grace Cai and Don Sofge. An urgency-dependent quorum sensing algorithm for n-site selection in autonomous swarms. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1853–1855, 2019.

[8] Cindy Calderón-Arce, Juan Carlos Brenes-Torres, and Rebeca Solis-Ortega. Swarm robotics: Simulators, platforms and applications review. *Computation*, 10(6):80, May 2022.

[9] Scott Camazine, Peter K Visscher, Jennifer Finley, and Richard S Vetter. House-hunting by honey bee swarms: collective decisions and individual behaviors. *Insectes Sociaux*, 46(4):348–360, 1999.

[10] Alexandre Campo, Simon Garnier, Olivier Dédriche, Mouhcine Zekkri, and Marco Dorigo. Self-organized discrimination of resources. *PLoS One*, 6(5):e19888, 2011.

[11] Yufeng Chen, Hongqiang Wang, E Farrell Helbling, Noah T Jafferis, Raphael Zufferey, Aaron Ong, Kevin Ma, Nicholas Gravish, Pakpong Chirarattananon, Mirko Kovac, et al. A biologically inspired, flapping-wing, hybrid aerial-aquatic microrobot. *Science robotics*, 2(11):eaao5619, 2017.

[12] Lars Chittka, Adrian G Dyer, Fiola Bock, and Anna Dornhaus. Bees trade off foraging speed for accuracy. *Nature*, 424(6947):388–388, 2003.

[13] Jason R. Cody and Julie A. Adams. An evaluation of quorum sensing mechanisms in collective value-sensitive site selection. In *2017 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, pages 40–47, 2017.

[14] Micael Santos Couceiro. An overview of swarm robotics for search and rescue applications. *Artificial Intelligence: Concepts, Methodologies, Tools, and Applications*, pages 1522–1561, 2017.

[15] Karthik Dantu, Bryan Kate, Jason Waterman, Peter Bailis, and Matt Welsh. Programming micro-aerial vehicle swarms with karma. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 121–134, 2011.

[16] Gloria DeGrandi-Hoffman, Stephen A Roth, GL Loper, and EH Erickson Jr. Beepop: a honeybee population dynamics simulation model. *Ecological modelling*, 45(2):133–150, 1989.

[17] Miguel Duarte, Jorge Gomes, Vasco Costa, Tiago Rodrigues, Fernando Silva, Víctor Lobo, Mario Monteiro Marques, Sancho Moura Oliveira, and Anders Lyhne Christensen. Application of swarm robotics systems to marine environmental monitoring. In *OCEANS 2016-Shanghai*, pages 1–8. IEEE, 2016.

[18] Xumei Fan, William Sayers, Shujun Zhang, Zhiwu Han, Luquan Ren, and Hassan Chizari. Review and classification of bio-inspired algorithms and their applications. *Journal of Bionic Engineering*, 17(3):611–631, 2020.

[19] Nigel R Franks, Katherine A Hardcastle, Sophie Collins, Faith D Smith, Kathryn ME Sullivan, Elva JH Robinson, and Ana B Sendova-Franks. Can ant colonies choose a far-and-away better nest over an in-the-way poor one? *Animal Behaviour*, 76(2):323–334, 2008.

[20] Ryusuke Fujisawa and Shigeto Dobata. Lévy walk enhances efficiency of group foraging in pheromone-communicating swarm robots. In *Proceedings of the 2013 IEEE/SICE International Symposium on System Integration*, pages 808–813, 2013.

[21] Zhi Wei Gan, Julian Shun, Grace Cai, and Nancy Lynch. Par-swarm: A c++ framework for evaluating distributed algorithms for robot swarms.

[22] Brian P Gerkey and Maja J Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International journal of robotics research*, 23(9):939–954, 2004.

[23] Adám Halász, M Ani Hsieh, Spring Berman, and Vijay Kumar. Dynamic redistribution of a swarm of robots among multiple sites. In *2007 IEEE/RSJ international conference on intelligent robots and systems*, pages 2320–2325. IEEE, 2007.

[24] Heiko Hamann, Thomas Schmickl, Heinz Wörn, and Karl Crailsheim. Analysis of emergent symmetry breaking in collective decision making. *Neural Computing and Applications*, 21(2):207–218, 2012.

[25] Heiko Hamann and Heinz Wörn. A framework of space–time continuous models for algorithm design in swarm robotics. *Swarm Intelligence*, 2(2):209–239, 2008.

[26] Richard P Heitz. The speed-accuracy tradeoff: history, physiology, methodology, and behavior. *Frontiers in neuroscience*, 8:150, 2014.

[27] David Hiebeler et al. The swarm simulation system and individual-based modeling. Santa Fe Institute Santa Fe, NM, USA, 1994.

[28] Matthew Hoeing, Prithviraj Dasgupta, Plamen Petrov, and Stephen O'Hara. Auction-based multi-robot task allocation in comstar. In *Proceedings of the 6th international joint conference on autonomous agents and multiagent systems*, pages 1–8, 2007.

[29] Nicholas R Hoff, Amelia Sagoff, Robert J Wood, and Radhika Nagpal. Two foraging algorithms for robot swarms using only local communication. In *2010 IEEE International Conference on Robotics and Biomimetics*, pages 123–130. IEEE, 2010.

[30] M Ani Hsieh, Ádám Halász, Spring Berman, and Vijay Kumar. Biologically inspired redistribution of a swarm of robots among multiple sites. *Swarm Intelligence*, 2(2):121–141, 2008.

[31] Dervis Karaboga and Bahriye Akay. A survey: algorithms simulating bee swarm intelligence. *Artificial intelligence review*, 31(1):61–85, 2009.

[32] Shreeya Khurana and Donald Sofge. Quorum sensing re-evaluation algorithm for n-site selection in autonomous swarms. In *ICAART (1)*, pages 193–198, 2020.

[33] Abderraouf Maoudj and Anders Lyhne Christensen. Decentralized multi-agent path finding in warehouse environments for fleets of mobile robots with limited communication range. In *International Conference on Swarm Intelligence*, pages 104–116. Springer, 2022.

[34] Nirmal Joshua Mathews, Tesbin K Varghese, Prince Zachariah, and Ninos Aji Chirathalattu. Fabrication of solar powered oil skimmer robot. 2018.

[35] Cameron Musco, Hsin-Hao Su, and Nancy A. Lynch. Ant-inspired density estimation via random walks. *Proceedings of the National Academy of Sciences*, 114(40):10534–10541, 2017.

[36] Keith J O'hara, Daniel B Walker, and Tucker R Balch. The gnats—low-cost embedded networks for supporting mobile robots. In *Multi-Robot Systems. From Swarms to Intelligent Automata Volume III*, pages 277–282. Springer, 2005.

[37] Chris A. C. Parker and Hong Zhang. Cooperative decision-making in decentralized multiple-robot systems: The best-of-n problem. *IEEE/ASME Transactions on Mechatronics*, 14:240–251, 2009.

[38] H Van Parunak, Michael Purcell, and Robert O'Connell. Digital pheromones for autonomous coordination of swarming uav's. In *1st UAV Conference*, page 3446, 2002.

[39] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[40] Stephen C Pratt. Behavioral mechanisms of collective nest-site choice by the ant temnothorax curvispinosus. *Insectes Sociaux*, 52(4):383–392, 2005.

[41] Stephen C Pratt. Quorum sensing by encounter rates in the ant temnothorax albipennis. *Behavioral Ecology*, 16(2):488–496, 2005.

[42] Stephen C Pratt, David JT Sumpter, Eamonn B Mallon, and Nigel R Franks. An agent-based model of collective nest choice by the ant temnothorax albipennis. *Animal Behaviour*, 70(5):1023–1036, 2005.

[43] Andreagiovanni Reina, James AR Marshall, Vito Trianni, and Thomas Bose. Model of the best-of-n nest-site selection process in honeybees. *Physical Review E*, 95(5):052411, 2017.

[44] Andreagiovanni Reina, Gabriele Valentini, Cristian Fernández-Oto, Marco Dorigo, and Vito Trianni. A design pattern for decentralised decision making. *PloS one*, 10(10):e0140950, 2015.

[45] A. M. Reynolds and C. J. Rhodes. The lévy flight paradigm: Random search patterns and mechanisms. *Ecology*, 90(4):877–887, 2009.

[46] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, 1987.

[47] Elva JH Robinson, Faith D Smith, Kathryn ME Sullivan, and Nigel R Franks. Do ants make direct comparisons? *Proceedings of the Royal Society B: Biological Sciences*, 276(1667):2635–2641, 2009.

[48] Michael Rubenstein, Christian Ahler, and Radhika Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *2012 IEEE international conference on robotics and automation*, pages 3293–3298. IEEE, 2012.

[49] Katherine Russell, Michael Schader, Kevin Andrea, and Sean Luke. Swarm robot foraging with wireless sensor motes. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 287–295. Citeseer, 2015.

[50] F Sahin et al. A swarm intelligence based approach to the mine detection problem. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 3, pages 6–pp. IEEE, 2002.

[51] David W Sims, Nicolas E Humphries, Russell W Bradford, and Barry D Bruce. Lévy flight and brownian search patterns of a free-ranging predator reflect different prey field characteristics. *Journal of Animal Ecology*, 81(2):432–442, 2012.

[52] Justin Solomon. Optimal transport on discrete domains. *AMS Short Course on Discrete Differential Geometry*, 2018.

[53] Gabriele Valentini, Eliseo Ferrante, and Marco Dorigo. The best-of-n problem in robot swarms: Formalization, state of the art, and novel perspectives. *Frontiers in Robotics and AI*, 4:9, 2017.

[54] Gandimohan M Viswanathan, Sergey V Buldyrev, Shlomo Havlin, MGE Da Luz, EP Raposo, and H Eugene Stanley. Optimizing the success of random searches. *nature*, 401(6756):911–914, 1999.

[55] Jan Wessnitzer and Chris Melhuish. Collective decision-making and behaviour transitions in distributed ad hoc wireless networks of mobile robots: Target-hunting. In *European Conference on Artificial Life*, pages 893–902. Springer, 2003.

[56] Bo Xu, Zhaofeng Yang, Yu Ge, and Zhiping Peng. Coalition formation in multi-agent systems based on improved particle swarm optimization algorithm. *International Journal of Hybrid Information Technology*, 8(3):1–8, 2015.

[57] Dandan Zhang, Guangming Xie, Junzhi Yu, and Long Wang. Adaptive task assignment for multiple mobile robots via swarm intelligence approach. *Robotics and Autonomous Systems*, 55(7):572–588, 2007.

[58] Jiajia Zhao, Nancy Lynch, and Stephen C Pratt. The power of social information in ant-colony house-hunting: A computational modeling approach. *bioRxiv*, pages 2020–10, 2021.