A Geometry-Sensitive Quorum Sensing Algorithm for the Best-of-N Site Selection Problem

Grace Cai[1*] and Nancy Lynch[1*]

[1]Computer Science and Artificial Intelligence Laboratory, MIT, Vassar St., Cambridge, 02139, MA, USA.

*Corresponding author(s). E-mail(s): gracecai@mit.edu; lynch@mit.edu;

Abstract

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. 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, 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.

1 Introduction

Swarms of birds, bees, and ants are able to coordinate themselves to make decisions using only local interactions [1–3]. Modelling these natural swarms has inspired many successful swarm algorithms [4]. 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 [5].

Many variations of the best-of-N site selection problem have been studied for swarms [6]. For example, when sites are of equal quality, choosing one is a symmetry-breaking problem [7, 8]. 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) [9].

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) [10, 11]. 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 2 describes the house hunting process of ants and overviews existing swarm models. Section 3 describes our model. We provide details on the implementation of our model in Section **??**. We test the model accuracy and decision time in different geographic situations, and report the results in Section 4. We discuss these results in Section 5. Lastly, we suggest future work in Section 6.

2 Background

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 [2, 12].

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 [13, 14]. Franks [13] 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.

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 [11, 15]. These models, initiated by Pratt [11], 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 nest 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 [15].

The corresponding problem to house hunting in robot swarms is known as the *N-site selection problem* [6]. Agents, starting at a central home site, must find and choose among N candidate nest sites in the environment and move to the site with highest quality. Unlike house hunting models, which do not physically simulate ants in space, swarm models set up ants in a simulated arena and let them physically explore sites and travel between them.

Inspired by ant modelling, [10] and [5] have modeled swarm agents using four main states – Uncommitted Latent, Uncommitted Interactive, Favoring Latent, and Favoring Interactive (with [10] adding a fifth Committed state to emulate having detected a quorum). Uncommitted Latent agents remain in the home nest 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 nest or committed to it. Other swarm models for N-site selection typically use a similar progression through uncommitted, favoring, and committed type phases [16].

One setup where a high quality site was twice as far as a low quality one was successfully solved in [5], but for the most part these models and their variations have mainly been tested in arenas with two candidate sites equidistant from the home nest [10, 17–19]. Our model aims to analyze the behavior of these models in more varied site setups and and improve upon them.

3 Model

We first describe our new discrete geographical model for modeling swarms. Then we discuss the individual restrictions, parameters, and agent algorithms needed for the house hunting problem specifically.

3.1 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 $\mathsf{V} = mn$. 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.

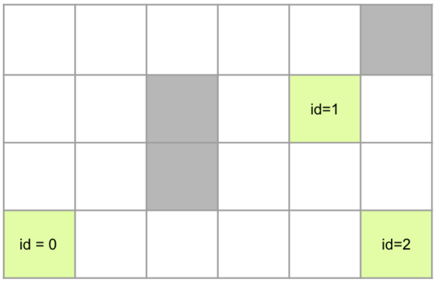

Fig. 1: A visual representation of a global configuration for agents wandering an arena with obstacles. Here, we use the interpretation of vertices as squares. Each square is grey if it is an obstacle, green if it contains an agent, and white if it is empty.

We also 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.

3.1.1 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:

- $svmap : V \rightarrow SV$ is the vertex state mapping, which assigns a vertex-state to each vertex,
- $srmap : R \rightarrow SR$ is the agent state mapping, which assigns an agent-state to each agent, and
- $locmap : R \rightarrow V$ 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 1. In Figure 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.

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.

3.1.2 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,
- $srmap_1 : myagents \rightarrow SR$ is the new agent state mapping, for agents currently at the vertex, and
- $dirmap_1 : myagents \rightarrow \{R, L, U, D, S\}$ gives directions of motion for all the agents currently at the vertex.

3.1.3 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 suggested vertex state sv', agent state sr', and direction of motion $d \in \{R, L, U, D, S\}$.

Phase Two: A 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.

3.1.4 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 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, 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.

3.2 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

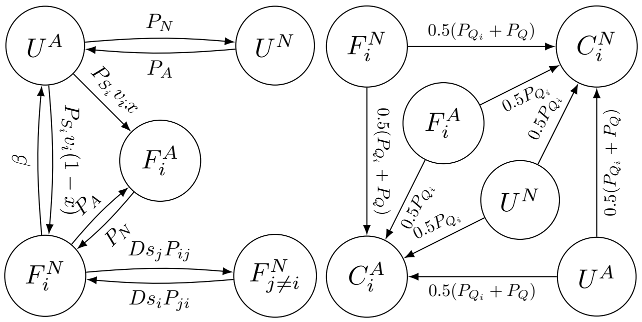

Fig. 2: State model. $\{U, F, C\}$ denote preference states. The superscript $\{N, A\}$ denotes the activity state, and a subscript i denotes that an agent is favoring or committed to site i. The transitions for Uncommitted and Favoring states are shown on the left, and transitions from Uncommitted and Favoring to Committed states are on the right.

s_i also has a quality $s_i.qual \in [0, 1]$. 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.

In order to properly represent sites in our model, the vertex state SV consists of the following variables:

- `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 $s_i.qual$, 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 the pair of pairs $((x_i^1, x_i^2),(y_i^1,y_i^2))$. 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 $[x_i^1, x_i^2]$ and y range $[y_i^1, y_i^2]$.

3.3 Agent States and Transition Function

We first provide a high-level overview of the agent state set SR and 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 2.

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 [20]. 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 $x = \frac{P_A}{P_A+P_N}$ percent of uncommitted agents are active, whereas $1-x$ agents remain in the nest. Prior work [5] 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 $q = \lfloor (q_{MIN} - q_{MAX}) * s_i.qual + q_{MAX} \rfloor$, 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.

In order to execute the core state transitions, the agent state set SR is as follows. 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. The constant variables we use in the agent state are:

- `INFLUENCE_RADIUS`, the influence radius I
- `N`, the height of the grid N
- `M`, the width of the grid M
- `L`, the inverse average site round trip L
- `Q_MIN`, the minimum quorum threshold q_{MIN}
- `Q_MAX`, the maximum quorum threshold q_{MAX}
- `MESSAGE_RATE`, how often the agent sends messages when it has information, in units of 1/rounds. This corresponds to the variable D.
- `LEVY_LOC` and `LEVY_CAP`, levy flight distribution parameters, where `LEVY_CAP` caps the right tail of the distribution

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 $[0, \mid R \mid -1]$ (where R is the agent set)

Lastly, modifiable variables are used to store extra, changeable state. For our house hunting implementation, the following modifiable variables are used:

- `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

Algorithm 1 Uncommitted Nest (UN) Agent Transition for an agent a

```
procedure GENERATE_TRANSITION_UN(local_vertex_mapping))
    ▷Before anything else, check if a quorum was sensed
    $quorum\_sensed, new\_agent\_state \leftarrow$
        check_quorum_sensed($local\_vertex\_mapping$)
    if $quourm\_sensed$ then
        return $a$.location.state, $new\_agent\_state, S$
    $s \leftarrow a$.state
    ▷If the agent just transitioned to UN and is still travelling to the home nest
    if $s$.destination $\neq null$ then
        $new\_direction =$ get_direction_from_destination(s.destination,
            ($a$.location.x, $a$.location.y))
        $new\_location =$ get_coords_from_movement(
            $a$.location.x, $a$.location.y, $new\_direction$)
        if $a$.within_site($new\_location$[0], $new\_location$[1], $s$.home) then
            $s$.destination $\leftarrow$ null
            return $a$.location.state, $s, new\_direction$
    ▷If the agent is already in the home nest
    $active\_chance \leftarrow s$.L/9
    if random_float_from(0,1) $\leq active\_chance$ then
        $s$.activity_type $\leftarrow$ Active
        return $a$.location.state, $s, S$
    else
        return $a$.location.state, $s, S$
```

The agent transition function α uses these state variables to implement the transitions between the six core states. An pseudocode example of how α looks like for Uncommitted Nest agents can be seen in Algorithm 1. In this example, we first check 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 helper functions referenced in Algorithm 1 are described in detail in Appendix B and Appendix C. The full implementation details for the remaining five core states, the resolution rule, and the details for how the model transitions can be found in Appendix A.

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 [21]. 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).

4.1 Further Nest of Higher Quality

House hunting ants are capable of choosing further, higher quality sites over closer, lower quality ones [13]. 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 [13].

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

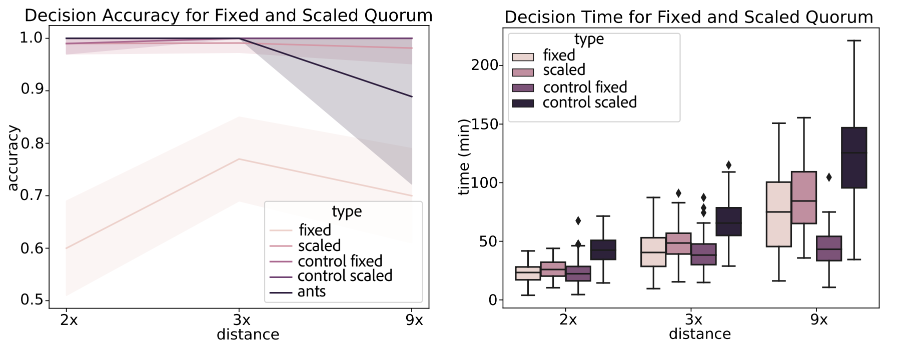

Fig. 3: Decision Time and Accuracy for far nests 2, 3, and 9 times as far from the home nest. Fixed quorum indicates the fixed threshold value of 4, and scaled quorum indicates $q_{MIN} = 4, q_{MAX} = 7$. The accuracy for the actual ants is taken from [13].

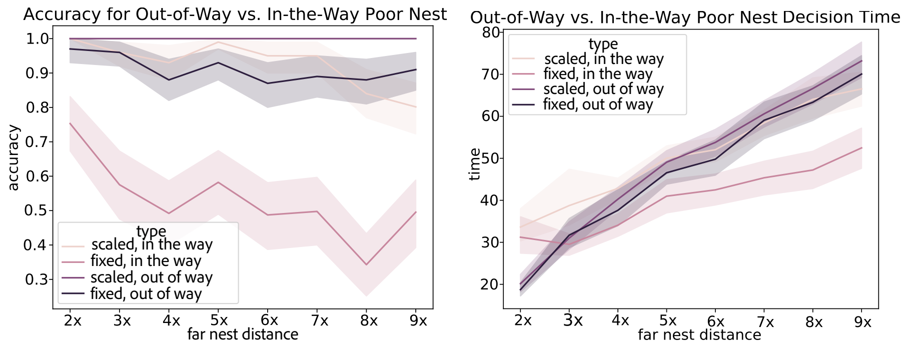

Fig. 4: Decision Time and Accuracy for far nests $2-9$ times further than the close nest for both fixed and scaled quorums. In the in-the-way setup, the home nest, low quality nest, and high quality nest were lined up in that order. In the out of way setup, the low quality nest, home nest, and high quality nest were lined up in that order.

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 4 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.

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 5 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.

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

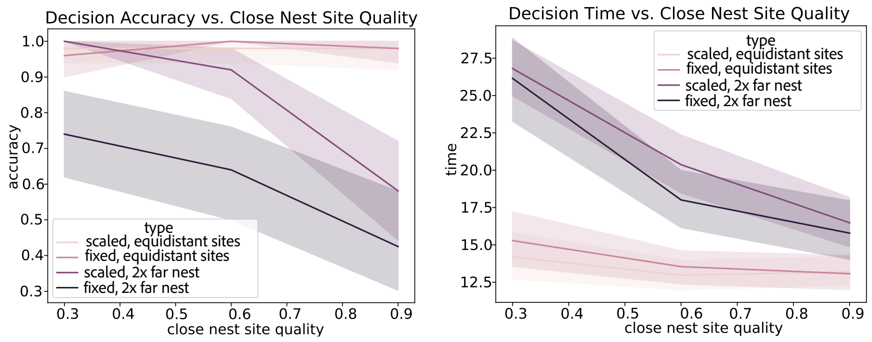

Fig. 5: Decision Accuracy and Time given varying differences in site quality between the near and the far nest.

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 4 shows that using a scaled quorum can still help to improve accuracy.

Figure 5 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. Contrarily, 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 [22, 23]. 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.

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, or try to adapt the house hunting model to an arena with continuous site values.

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 [19], which does this for candidate sites all with an equal chance of discovery.

7 Declarations

7.1 Ethics approval and consent to participate

Not applicable.

7.2 Consent for publication

Not applicable.

7.3 Availability of data and material

The simulation data used to generate our graphs is available at `https://github.com/ssgcai/geo-swarm/tree/main/house_hunting`. The model code is also available at the same link for ease of replication.

7.4 Competing interests

The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

7.5 Funding

This work was supported by NSF awards CCF-1461559, CCF-1810758, and CCF-2139936.

7.6 Authors' contributions

N.L and G.C wrote section 3.1 together. G.C. wrote the remainder of the manuscript text and prepared all figures. All authors reviewed the manuscript.

7.7 Acknowledgements

Special thanks to Jiajia Zhao for her advice and ideas about early versions of this work.

References

[1] Camazine, S., Visscher, P.K., Finley, J., Vetter, R.S.: House-hunting by honey bee swarms: collective decisions and individual behaviors. Insectes Sociaux **46**(4), 348–360 (1999)

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

[3] Reynolds, C.W.: Flocks, herds and schools: A distributed behavioral model. In: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, pp. 25–34 (1987)

[4] Fan, X., Sayers, W., Zhang, S., Han, Z., Ren, L., Chizari, H.: Review and classification of bio-inspired algorithms and their applications. Journal of Bionic Engineering **17**(3), 611–631 (2020)

[5] Reina, A., Valentini, G., Fernández-Oto, C., Dorigo, M., Trianni, V.: A design pattern for decentralised decision making. PloS one **10**(10), 0140950 (2015)

[6] Valentini, G., Ferrante, E., Dorigo, M.: The best-of-n problem in robot swarms: Formalization, state of the art, and novel perspectives. Frontiers in Robotics and AI **4**, 9 (2017)

[7] Hamann, H., Schmickl, T., Wörn, H., Crailsheim, K.: Analysis of emergent symmetry breaking in collective decision making. Neural Computing and Applications **21**(2), 207–218 (2012)

[8] Wessnitzer, J., Melhuish, C.: Collective decision-making and behaviour transitions in distributed ad hoc wireless networks of mobile robots: Target-hunting. In: European Conference on Artificial Life, pp. 893–902 (2003). Springer

[9] Campo, A., Garnier, S., Dédriche, O., Zekkri, M., Dorigo, M.: Self-organized discrimination of resources. PLoS One **6**(5), 19888 (2011)

[10] Cody, J.R., Adams, J.A.: An evaluation of quorum sensing mechanisms in collective value-sensitive site selection. In: 2017 International Symposium on Multi-Robot and Multi-Agent Systems (MRS), pp. 40–47 (2017). https://doi.org/10.1109/MRS.2017.8250929

[11] Pratt, S.C., Sumpter, D.J., Mallon, E.B., Franks, N.R.: An agent-based model of collective nest choice by the ant temnothorax albipennis. Animal Behaviour **70**(5), 1023–1036 (2005)

[12] Pratt, S.C.: Behavioral mechanisms of collective nest-site choice by the

ant temnothorax curvispinosus. Insectes Sociaux **52**(4), 383–392 (2005)

[13] Franks, N.R., Hardcastle, K.A., Collins, S., Smith, F.D., Sullivan, K.M., Robinson, E.J., Sendova-Franks, A.B.: Can ant colonies choose a far-and-away better nest over an in-the-way poor one? Animal Behaviour **76**(2), 323–334 (2008)

[14] Robinson, E.J., Smith, F.D., Sullivan, K.M., Franks, N.R.: Do ants make direct comparisons? Proceedings of the Royal Society B: Biological Sciences **276**(1667), 2635–2641 (2009)

[15] Zhao, J., Lynch, N., Pratt, S.C.: The power of social information in ant-colony house-hunting: A computational modeling approach. bioRxiv, 2020–10 (2021)

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

[17] Cai, G., Sofge, D.: 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, pp. 1853–1855 (2019)

[18] Khurana, S., Sofge, D.: Quorum sensing re-evaluation algorithm for n-site selection in autonomous swarms. In: ICAART (1), pp. 193–198 (2020)

[19] Reina, A., Marshall, J.A., Trianni, V., Bose, T.: Model of the best-of-n nest-site selection process in honeybees. Physical Review E **95**(5), 052411 (2017)

[20] Sims, D.W., Humphries, N.E., Bradford, R.W., Bruce, B.D.: 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)

[21] Rubenstein, M., Ahler, C., Nagpal, R.: Kilobot: A low cost scalable robot system for collective behaviors. In: 2012 IEEE International Conference on Robotics and Automation, pp. 3293–3298 (2012). IEEE

[22] Chittka, L., Dyer, A.G., Bock, F., Dornhaus, A.: Bees trade off foraging speed for accuracy. Nature **424**(6947), 388–388 (2003)

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

Appendix A House Hunting Model Implementation Details

We use the below classes, each with their own functions and variables, in the implementation of our house hunting model.

- Agent, representing the agents in the formal model
- Agent State, representing the state set SR for agents in the formal model. The agent state is described in Section 3.3.
- 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. The vertex state is described in Section 3.3

A.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

A.2 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.

A.3 Agent Transition Function

We now consider the generation of agent transitions at each timestep of the house hunting algorithm. 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. The transition algorithm for the Uncommitted Nest state was described in Section 3.3. We describe the algorithm for each remaining main state below.

The pseudocode for the transition function uses a number of simple utility functions whose I/O specifications can be found in Appendix B. It also uses a few more complex utility functions. The definitions and explanations for these functions can be found in Appendix C.

A.3.1 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 2. 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 2 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:      $quorum\_sensed, new\_agent\_state \leftarrow$
4:          check_quorum_sensed($local\_vertex\_mapping$)
5:      if $quourm\_sensed$ then
6:          return $a$.location.state, $new\_agent\_state$, S
7:      $s \leftarrow a$.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:         $nearby\_site, site\_location \leftarrow$
11:             find_nearby_site($local\_vertex\_mapping$)
```

Algorithm 2 Uncommitted Active (UA) algorithm (continued)

```
12:         ▷Explore the site the agent just found
13:         if nearby_site is not null then
14:             s.destination ← site_location
15:             s.desintation_site ← nearby_site
16:     else if s.exploration_cooldown > 0 then
17:         ▷Decrement exploration cooldown if it is in effect
18:         s.exploration_cooldown -= 1
19:     ▷The agent is headed towards a site to explore
20:     if s.destination_site is not null then
21:         if s.location.state.site_name == s.destination_site.site_name
    and not s.exploring_site then
22:             ▷The agent has just arrived at the site
23:             s.exploring_site ← True
24:             s.travel_distance ← 10
25:             s.angle ← random_float_from(0, 2π)
26:             s.starting_point ← (s.location.x, s.location.y)
27:         else if not s.exploring_site then
28:             ▷The agent is still headed towards the site
29:             new_direction ← get_direction_from_destination(
30:                 s.destination, (s.location.x, s.location.y))
31:             return s.location.state, s, new_direction
32:     ▷Calculate if agent should become nest agent
33:     nest_chance ← s.L
34:     if random_float_from(0,1)<nest_chance and not s.exploring_site
    then
35:         s.destination ← s.random_location_in_site(s.home)
36:         s.activity_type ← Nest
37:         return s.location.state, new_agent_state, new_direction
38:     ▷The agent has finished exploring the site
39:     if s.travel_distance == 0 and s.exploring_site then
40:         s.destination ← null
41:         s.exploring_site ← False
42:         s.favored_site ← s.destination_site
43:         s.destination_site ← null
```

A.3.2 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

Algorithm 2 Uncommitted Active(UA) algorithm (continued)

```
44:         if random_float_from(0,1) < s.location.site_value then
45:             ▷Choose to favor the site
46:             if random_float_from(0,1) < 9/10 then
47:                 s.destination ← s.random_location_in_site(s.home)
48:                 s.destination_site ← s.home
49:                 s.preference_type ← Favoring
50:                 s.activity_type ← Nest
51:                 return s.location.state, s, S
52:             else
53:                 s.preference_type ← Favoring
54:                 s.activity_type ← Active
55:                 return s.location.state, s, S
56:         else
57:             ▷Reject the site
58:             s.exploration_cooldown ← 10
59:             s.favored_site ← null
60:     ▷Agent continues exploring the arena
61:     new_direction, s ← get_travel_direction(s)
62:     return s.location.state, s, new_direction
```

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 3.

Algorithm 3 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_location ← get_coords_from_movement(
13:             s.location.x, s.location.y, new_direction)
14:         ▷Agent has just reached their favored site
```

Algorithm 3 Favoring Active (FA) algorithm (continued)

```
15:         if $s$.within_site($new\_location$[0], $new\_location$[1], $s$.favored_site)
  then $s$.destination ← null
16:         return $s$.location.state, $s$, $new\_direction$
17:     ▷Agent is inside favored site
18:     $nest\_chance \leftarrow s$.L
19:     if random_float_from(0,1) < $nest\_chance$ then
20:         $s$.destination ← $s$.random_location_in_site($s$.home)
21:         $s$.desintation_site ← $s$.home
22:         $s$.activity_type ← Nest
23:         return $s$.location.state, $s$, S
24:     else
25:         $new\_direction$ ← $s$.get_travel_direction()
26:         return $s$.location.state, $s$, $new\_direction$
```

A.3.3 Favoring Nest Transitions

In the Favoring Nest (FN) state, the agent has found a site to favor and returns home to try and recruit other FN agents to support their favored site. 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 4 Favoring Nest (FN) Agent Transition for an agent a

```
1: procedure GENERATE_TRANSITION_FN(local_vertex_mapping))
2:     $s \leftarrow a$.state
3:     ▷Determine whether to abandon site
4:     $should\_abandon, new\_state \leftarrow$
5:         should_abandon_site($local\_vertex\_mapping, s$)
6:     if $should\_abandon$ then
7:         return $a$.location.state, $new\_state$, S
```

Algorithm 4 Favoring Nest (FN) algorithm (continued)

```
8:      ▷Check if a quorum was sensed
9:      $quorum\_sensed, new\_agent\_state \leftarrow$
10:         check_quorum_sensed($local\_vertex\_mapping$)
11:     if $quourm\_sensed$ then
12:         return $a$.location.state, $new\_agent\_state$, S
13:     if $s$.destination is null then
14:         ▷Communicate with other agents to see if a better nest can be found
15:         $better\_nest \leftarrow$ find_better_nest($local\_vertex\_mapping$)
16:         if $better\_nest$ is not null then
17:             $s$.destination $\leftarrow$ random_location_in_site($better\_nest$)
18:             $s$.destination_site $\leftarrow$ $better\_nest$
19:     if $s$.destination is not null then
20:         if $s$.location.site_name == $s$.destination_site.site_name and
    $s$.destination_site != $s$.home_nest and not $s$.exploring_site then
21:             ▷Agent has just arrived at the better site it heard of
22:             $s$.exploring_site $\leftarrow$ True
23:             $s$.travel_distance $\leftarrow$ 10
24:             $s$.angle $\leftarrow$ random_float_from(0, $2\pi$)
25:             $s$.starting_point $\leftarrow$ ($s$.location.x, $s$.location.y)
26:         else if $s$.location.state.site_name==$s$.destination_site.site_name
     and $s$.destination_site == $s$.home_nest then
27:             ▷Agent just became FN and just arrived at home
28:             $s$.destination $\leftarrow$ null
29:             $s$.destination_site $\leftarrow$ null
30:         else if not $s$.exploring_site then
31:             ▷Agent still headed towards either home or a better nest
32:             $new\_direction \leftarrow$ get_direction_from_destination(
33:                 $s$.destination, ($s$.location.x, $s$.location.y))
34:             return $s$.location.state, $s$, $new\_direction$
35:     ▷FN agents have chance of becoming FA
36:     $active\_chance \leftarrow$ $s$.L/9
37:     if random_float_from(0,1) < $active\_chance$ and not $s$.exploring_site
    then
38:         $s$.destination $\leftarrow$ random_location_in_site($s$.favored_site)
39:         $s$.activity_type $\leftarrow$ Active
40:         return $s$.location.state, $s$, S
41:     ▷Agent finished exploring better site; changes to favoring it
42:     if $s$.travel_distance == 0 and $s$.exploring_site then
43:         $s$.destination $\leftarrow$ null
44:         $s$.exploring_site $\leftarrow$ False
45:         $s$.favored_site $\leftarrow$ $s$.destination_site
46:         $s$.destination_site $\leftarrow$ null
47:         if random_float_from(0, 1) < 9/10 then
```

Algorithm 4 Favoring Nest (FN) algorithm (continued)

```
            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
```

A.3.4 Committed Nest Transitions

In the Committed Nest state, agents have already sensed a quorum and are either going towards or are inside the home nest to broadcast the quorum to other agents. Once they have finished broadcasting, they return to their 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 5.

Algorithm 5 Committed Nest (CN) Agent Transition for an agent a

```
procedure GENERATE_TRANSITION_CN(local_vertex_mapping))
    s ← a.state
    ▷Agent done with algorithm
    if s.terminated then
        return s.location.state, s, S
    ▷Agent headed towards home nest to broadcast quorum
    if s.destination_site == s.home then
        if s.location.state == s.home then
            s.destination ← null
            s.destination_site ← null
            s.travel_distance ← int(1/s.L)
            s.angle ← random_float_from(0, 2π)
            s.starting_point ← (s.location.x, s.location.y)
    else
        new_direction ← get_direction_from_destination(
            s.destination, s.location.x, s.location.y))
        return s.location.state, s, new_direction
    ▷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
```

A.3.5 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 6.

Algorithm 6 Committed Active (CA) Agent Transition for an agent a

```
procedure GENERATE_TRANSITION_CA(local_vertex_mapping))
    $s \leftarrow a$.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
        $s$, $new\_dir \leftarrow$ committed_agent_state_and_dir($s$)
    else
        $new\_dir \leftarrow$ get_travel_direction()
    return $s$.location.state, $s$, $new\_dir$
```

A.4 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 7), which represents stepping forward once in time. This function utilizes two helper functions and is defined as follows:

The function `generate_global_transitory()` (Algorithm 8) generates and returns the global transitory configuration, and the function `execute_transition(global_transitory)` simply executes the updates to the agent locations and states based upon the transitory configuration. The global transitory configuration is a map from coordinates to a triple consisting

Algorithm 7 Transition function for a configuration `C`

```
procedure TRANSITION
    $global\_transitory$ ← C.generate_global_transitory()
    C.execute_transition($global\_transitory$)
```

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 8 Generating the global transitory configuration for a configuration `C`

```
procedure GENERATE_GLOBAL_TRANSITORY
    $global\_transitory$ ← {}
    for $x$ in 1 ...C.M do
        for $y$ in 1 ...C.N do
            $local\_vertex\_mapping$ ← generate_local_mapping(
                C.vertices[($x,y$)], C.INFLUENCE_RADIUS, C.vertices)
            $global\_transitory$[($x,y$)] ← C.delta($local\_vertex\_mapping$)
    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 9) 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.

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

Algorithm 9 δ for a configuration C

```
procedure DELTA(local_vertex_mapping)
    $v \leftarrow$ local_vertex_mapping[(0,0)]
    if length($v$.agents) == 0 then return $v$.state, {}, {}
    proposed_vertex_states $\leftarrow$ {}
    proposed_agent_states $\leftarrow$ {}
    proposed_agent_dirs $\leftarrow$ {}
    for agent in $v$.agents do
        proposed_vertex_state, proposed_agent_state, direction =
            agent.generate_transition(local_vertex_mapping)
        proposed_vertex_states[agent.id] $\leftarrow$ proposed_vertex_state
        proposed_agent_states[agent.id] $\leftarrow$ proposed_agent_state
        proposed_agent_dirs[agent.id] $\leftarrow$ direction
    new_vertex_state, new_agent_states, new_agent_dirs = resolution_rule(
        proposed_vertex_states, proposed_agent_states, proposed_agent_dirs)
    return new_vertex_state, new_agent_states, new_agent_dirs
```

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.

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:

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.

Algorithm 10 Naive resolution rule

```
1: procedure NAIVE_RESOLUTION(proposed_vertex_states, proposed_agent_states, proposed_agent_dirs))
2:     $new\_vertex\_state \leftarrow proposed\_vertex\_states$.values()[0]
3:     $new\_agent\_states \leftarrow proposed\_agent\_states$
4:     $new\_agent\_dirs \leftarrow proposed\_agent\_dirs$
5:     return $new\_vertex\_state, new\_agent\_states, new\_agent\_dirs$
```

Appendix B 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 C 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 11.

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 (for example, setting the committed site as the agent's destination site). The code for how this function works can be seen in Algorithm 12.

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

Algorithm 11 Determining agent state transitions after quorum is sensed for agent a

```
procedure CHECK_QUORUM_SENSED(local_vertex_mapping)
    quorum_site ← quorum_sensed(local_vertex_mapping)
    s ← a.state
    if quorum_site is not null then
        s.quorum_site ← quorum_site
        if random_float_from(0,1) < 0.5 then
            s.destination ← random_location_in_site(s.home)
            s.destination_site ← s.home
            return True, s
        else
            s.travel_distance ← int(1/s.L)
            s.angle ← random_float_from(0,2π)
            s.starting_point ← s.location.x, s.location.y
            s.destination ← null
            s.destination_site ← null
            return True, s
    return False, null
```

Algorithm 12 Determining agent state and direction for agents a moving to the committed site

```
procedure COMMITTED_AGENT_STATE_AND_DIR(s)
    new_agent_state ← s
    if s.destination_site == s.quorum_site then
        ▷Agent arrived at committed site
        if s.location.x == s.destination[0] and s.location.y == s.destination[1] then
            new_agent_state.destination ← null
            new_agent_state.destination_site ← null
            new_agent_state.terminated ← True
            return new_agent_state, S
        else
            ▷Agent still headed towards committed site
            new_direction ← get_direction_from_destination(
                s.destination, (s.location.x, s.location.y))
            return new_agent_state, new_direction
    ▷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
```

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 13.

Algorithm 13 Determining whether a favoring agent a should abandon its site

```
procedure SHOULD_ABANDON_SITE(local_vertex_mapping, s)
    new_agent_state ← s
    ▷Update time since last neighor was seen
    if num_neighbors(local_vertex_mapping) > 1 then
        new_agent_state.time_since_neighbor ← 0
        ▷Abandon the favored site
    else
        new_agent_state.time_since_neighbor += 1
    if new_agent_state.time_since_neighor >= 5/s.L then
        new_agent_state.angle ← 0
        new_agent_state.starting_point ←
            (s.location.x, s.location.y)
        new_agent_state.travel_distance ← 0
        new_agent_state.destination ← null
        new_agent_state.destination_site ← null
        new_agent_state.exploring_site ← False
        new_agent_state.exploration_cooldown ← 0
        new_agent_state.favored_site ← None
        new_agent_state.time_since_neighbor ← 0
        new_agent_state.preference_type ← Uncommitted
        new_agent_state.activity_type ← Active
        return True, new_agent_state
    else
        return False, s
```