Mathematical modeling of multi-agent search & task allocation

Jonas A. Grønbakken

Thesis submitted for the degree of
Master in Cybernetics and Autonomous Systems
60 credits

Department of Technology Systems
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2023

Mathematical modeling of multi-agent search & task allocation

Jonas A. Grønbakken

© 2023 Jonas A. Grønbakken

Mathematical modeling of multi-agent search & task allocation

`http://www.duo.uio.no/`

Printed: Reprosentralen, University of Oslo

Abstract

Multi-agent search and task allocation(MASTA) has a wide range of applications, including search & rescue, ecological monitoring & sampling, military applications, etc. Considerable difficulty in designing such systems has been a lack of analytical modeling tools, requiring the researchers and engineers to rely on computer modeling & simulation(M&S). While M&S is an extraordinary and important tool, it often does not lend itself easily to human insight, can require a great deal of time and energy, and is frequently not fitting for quick decision-making. In this thesis, an analytical model of MASTA is presented and compared to a MASTA computer implementation as a baseline. The analytical model preforms within a few percent error and allows for greater insight into system behavior and parameter interactions. The model presented might reduce the design time of MASTA systems and allow for greater control of such systems with fast decision-making.

Mathematical modeling of multi-agent search & task allocation

Jonas A. Grønbakken

May 15, 2023

Contents

1 Introduction

Multi-robot task allocation (MRTA) concerns systems of multiple robots that solve tasks that they often cannot solve independently. It can be defined as an optimal assignment problem. [20]. This problem is often complex and is usually NP-hard. [21]. As a result, many different approaches have been developed. One of the most popular decentralized multi-agent systems of interest is market-based methods. In single-item auctions, an agent will start an auction for a task, the other agents will send bids based on their utility, and the auctioneer will choose the winners. We are interested in looking at systems that need to search for a task and then allocate it, so we combine search and task allocation. In real systems, communication radius and the ability to detect tasks is limited and can be prohibitively expensive, especially under water. We propose combining simple, cheap agents adaptively to create mobile phased arrays capable of longer-range communication and task detection. We refer to these agents as constituents, as they form a howl agent (composite) together. To our knowledge, this idea of constituent agents is not explored in previous literature. The optimal formation of composite agents is a complicated problem, as it affects search, communication and task allocation performance. If effective methodologies for forming and interacting composite agents are developed, it could allow for a large group of simple robots to take on tasks of much more expensive equipment.

[32]

[39]

[41] Researchers look into dynamic task allocation for search and retrieval with retrieval constraints. Objects for retrieval are found by searching a set of locations, which individual robots can do. Objects are associated with a type; before search and retrieval, a list is created of when each type of task can be delivered in relation to each other. For example, with types red and blue, given a list {red,blue,red,red}, any red or blue object found can fill the requirement but only following the specified order. First, an extended sequential single-item auction is developed and compared against an implicit coordination (consensus control) approach. They found consensus control completed all the tasks quicker, but agents moved less with the auction approach.

A potential application is underwater tasks where communication and navigation are prohibitively expensive; autonomous underwater vehicles (AUVs) can cost upwards of a hundred thousand dollars. Researchers [13] resent a method with which one expensive underwater robot can guide a swarm of simpler cheap robots using a hydrophone. The simple agents estimate relative heading by Doppler shift of frequencies emitted and distance by amplitude. Constituent agents forming phased arrays might be able to act as leaders, communicating with vehicles closer to the surface for navigation.

[24]

[44]

[34]

[27]

[36]

[18]

4. Swarm

The animal kingdom also sees agents coming together and emitting a stronger signal. . For example, glowworms (fireflies, lightning bugs) attract each other with bio-luminescence during mating season. Their glow increases in intensity as they congregate, attracting glowworms from further away. As a result, certain glowworm species will flash synchronously, many hundred together lighting up trees at a constant frequency with complete darkness in between [19].

[25]

[9]

[45]

[43]

1.1 Aim of thesis

The thesis aims to mathematically model the performance of MAS search and task allocation under a range of parameters such as; the number of agents, detection radius, number of tasks, etc. The models aim to increase understanding of MASTA systems and reduce design time and computational demand. In addition, mathematical models might also allow for the design and development of MASTA controllers. The goal is to answer the following research questions:

1. Is it possible to model MASTA systems mathematically?
2. Do these models increase understanding of MASTA?
3. Do these models reduce computational demand?

1.2 Outline

In section 2, background material is presented, consisting of short summaries of relevant fields for the thesis. Section 3 describes the implemented computer model and software. Section 4 contains experiments and results interwoven with analysis. Section 5 discusses the results from section 4. Finally, in section 6, conclusions are drawn, and ideas for future work are presented.

2 Background

2.1 Multi robot search and task allocation

First, we introduce MRTA (multi-robot task allocation) and define how search is Incorporated. The general MRTA problem is defined as follows, given a set of R robots and T tasks, each subset of robots r_i has an utility u_{ij} associated with each subset of tasks t_j. We wish to find a set of allocations X of robots to tasks s.t $\sum U_X$ is maximized, the utility of all allocations while no hard constraint is broken (e.g. $\sum_i x_{ij} = 1, \forall j$ where x_{ij} is an allocation of robot i to task j, tasks can only be solved once by a single robot). When discussing tasks, it is beneficial to consider them more than just basic operations. Imagine moving a set of planks from one location to another. It is natural to define the act of moving all the planks as one task and moving one or a set of planks as a sub-task. In this case, we can decompose the task into many sets of sub-tasks T_s such that when certain combinations of those sub tasks T_c is completed the task is completed. We call the set (T_s, T_c) a decomposition of task T, not all decomposition's are possible to follow through on, like starting with the plank on the bottom of the pile. When a decomposition is solvable by a MRS we refer to it as allocatable. Further classification of task types and taxonomies of MRTA help organize literature and solve problems with greater ease.

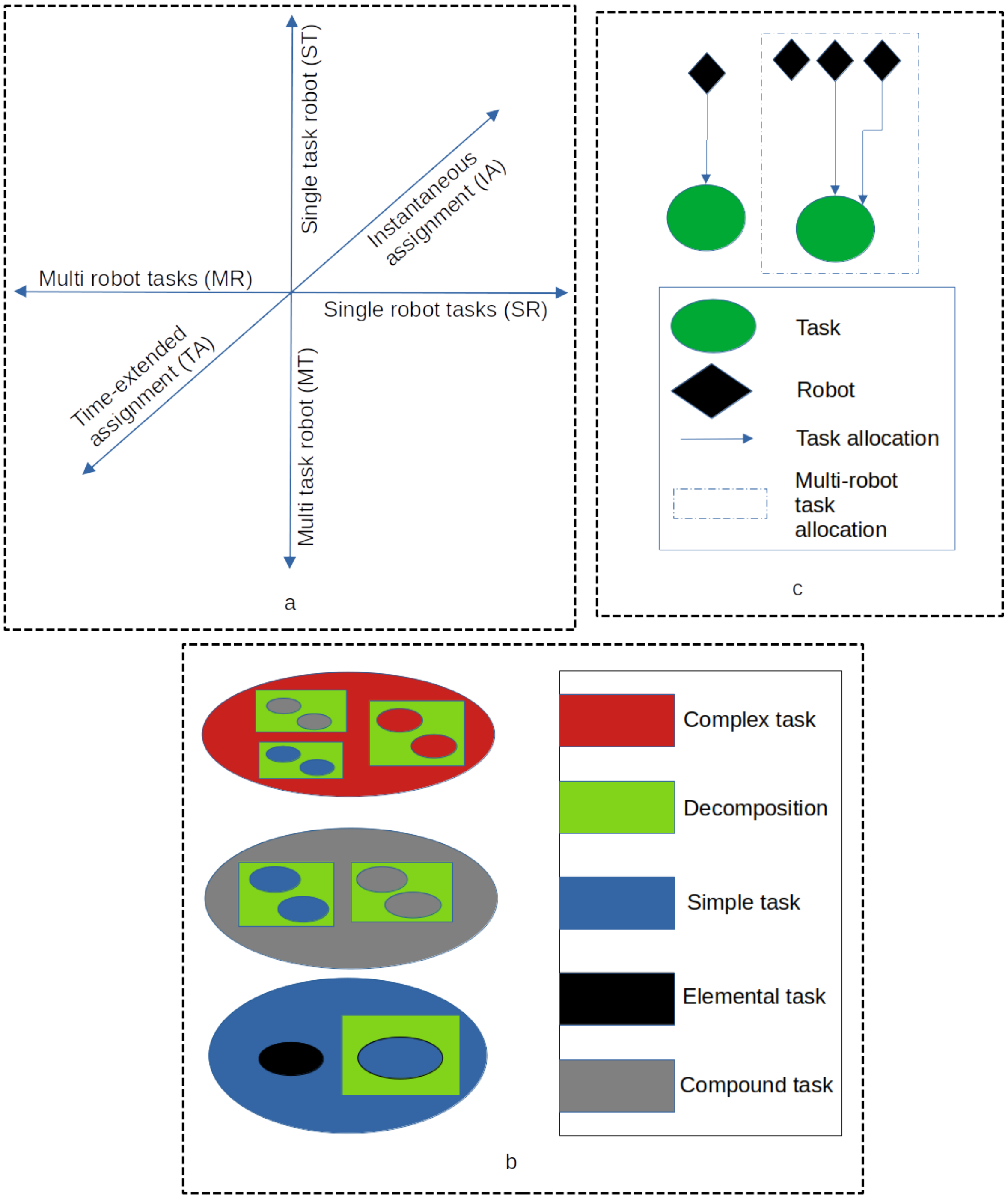


Figure 1: a) The three dimensions in [15] taxonomy, first, tasks who can be solved by one robot (SR) vs task who require multiple robots (MR). second, robots who can only solve one task concurrently (ST) vs robots solving multiple tasks concurrently (MT). Thirdly instantaneous assignment where no regard for future allocations is possible vs time-extend assignment where enough information is avalible such that the optimal assignment requires considering future assignments. b) types of tasks as defined by [46] and c) multi-robot task allocation.

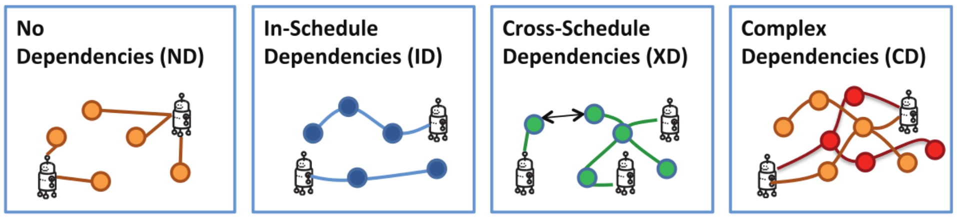

No Dependencies (ND)
In-Schedule Dependencies (ID)
Cross-Schedule Dependencies (XD)
Complex Dependencies (CD)

Figure 2: Scheduling dependencies (from Figure 3 in [21])

Brian P Gerkey and Maja J [15] Introduced in their taxonomy the three axis for classification of MRTA problems. Singel-task robots (ST) robots that only can work on one task at a time vs. multi-task robots (MT), which can work on multiple tasks simultaneously. Singel-robot tasks (SR) can be solved by a single robot vs. mult-robot tasks (MR) who require more than one to be solved. Lastly, they introduced instantaneous assignment (IA) vs. time-extended assignment (TA). With IA problems, there is no possibility for planning. All the information available for the system does not permit it to make future allocations or predictions on task completion/arrival. The opposite is true for TA. There are two other taxonomies on MRTA; the first [46] combines definitions for task decomposition's with scheduling dependencies for tasks and other agents. The second taxonomy focuses on temporal and ordering constraints without [46] task definitions. Lets first describe task decomposition, a decomposition of T is a set (T_S, T_C) where T_S is a set of sub tasks and $T_C$is a combination of those sub tasks such that the tuple satisfies T.

- **Multiple decomposability** refers to task which has more than one decomposition
- **Elemental task** - a task that cannot be decomposed
- **Simple task** - an elemental task or a decomposable simple task
- **Decomposable simple task** - can be decomposed into elemental tasks or decomposable simple sub-tasks (if there exists no multi-robot allocatable decomposition)
  - **Multi-robot allocatability** is a task that can be allocated to a group of robots such that no sub-team or robot solves it alone and the task is complete. That is, if the only allocation of task that satisfies it requires that a single robot or sub-group only completes it, then the allocation is obvious and is not an allocatability problem.
- **Compound task** - can be decomposed into a set of simple or compound tasks requiring the existence of precisely one fixed full decomposition for the task.
  - **Full decomposability** is a set of simple sub-tasks that can be derived within a finite amount of decomposition steps.
- **Complex task** - multiple decomposable tasks for which at least one decomposition is multi-robot allocatable. Sub-tasks may be simple, compound, or complex.

For these task definitions, [21] introduces the dependencies:

- **No dependencies (ND)** - utility of an agent is not affected by other tasks or other agents in the system. Task decomposition and task allocation are decoupled concerns of simple or compound tasks. Linear assignment problems

- **In-schedule dependencies (ID)** - the utility of an agent is affected by its own schedule, but no one else; task decomposition and task allocation are decoupled, simple, or compound tasks. NP-hard.
- **Cross-schedule dependencies (XD)** - the utility is affected by inter-schedule and ID, so an agent's schedule cannot be constructed optimally without considering other agents' schedules. Task decomposition is decoupled from task allocation, simple or compound tasks.
- **Complex-schedule dependencies (CD)** - inter-schedule dependencies for complex tasks, task decomposition, and task allocation depend on each other. Therefore, one must consider the joint problem of task decomposition, allocation, and individual agent schedule for optimal utility.

Secondly [28] introduces a new taxonomy on top of [15]. Where they distinguish between:

- **Time windows** (TW) vs. **synchronization** and **precedence (SP)**
  - **time window** is a specified interval (a, b) of time wherein it is possible to complete a task, if $a = 0$ its also called a deadline.
  - **precedence constraint** required ordering of tasks.
  - **synchronization constraint** are constraints relating the completion of tasks, for instance equal completion times.
- **Hard temporal constraints** vs. **soft temporal constraint**
  - hard constraints means they have to be satisfied
  - soft constraints can be violated but at a cost.
- **Deterministic** vs. **stochastic** models
  - Deterministic models give the same output every time while stochastic models try to represent something uncertain which gives differing results.

Other taxonomies have also been made on the basis of system functions such as [12], where groups are based not on the problem that is to be solved, but for example communication range and topology. We will look away from the later mentioned taxonomy and attempt to define problems in relation to the three mentioned above. One of the simplest problems is linear assignment problems which fall under ND[SR-ST-IA] (single robot-task, single task, instantaneous assignment, and no dependencies) with no temporal or ordering constraints (which will be the case unless otherwise mentioned). Linear assignment problems are solvable in polynomial time by a number of algorithms by, for example, representing it as a linear program; Maximize

$$\sum_{i \in N} \sum_{j \in M} x_{ij} u_{ij}$$

for N agents and M tasks subject to

$$\sum_{i \in N} x_{ij} = 1, \forall j \in M$$

and

$$\sum_{i \in N} x_{ij} = 1, \forall j \in M$$

where

$$x_{ij} \in [0, 1]$$

[15] . This problem is also optimally solvable by an auction algorithm [5], here presented as an optimum matching problem on a graph. In [38] the authors present the problem of moving boxes, some of which require multiple agents. They use a distributed problem-solving (DSP) approach, where agents form a coalition that can overlap to move the boxes. We can classify this problem as XD[MR-MT-IA]:SP with the introduction of precedence constraints. The authors use greedy set partitioning and coverage algorithms to solve the problem, which requires global agent communication. Some work has been done with search. In [16] the robots find and collect items in a generated environment. These items have an associated weight requirement and speed at which they can be returned based on the robot's work capacity. When a task is found, the agent starts an auction where other agents bid on the task. The auctioneer then chooses the winners, which help move the item to a specified location. Leaders of groups (auctioneers) can bid between each other for agents if they need help solving the task they have found. We can classify this as XD[ST-MR-IA] as the coalitions break apart after task completion. When combining search and task allocation, the problem can become very complicated. Search and task allocation can encompass multiple classifications.

2.2 Markov chains

Markov chains, also called Markov processes, are stochastic models describing the state transitions of some systems with the Markov property. They can be used in several ways; Markov decision processes are used for decision-making and control in robotics and autonomous systems. Hidden Markov models are used when an underlying process is assumed to have the Markov property, but its states are not directly observable. Here we will discuss Markov chains with finite states that operate in discrete and continuous time [42]. These models are commonly represented with matrices, which for discrete time Markov chains(DTMC) consist of state transition probabilities. A typical example is a prediction of the weather tomorrow if the weather today is known. Say if it is sunny, then there is a probability α that it will be sunny tomorrow, and a probability $1 - \alpha$ that it will rain. On the other hand, if it is raining today, there is a probability β that it will rain tomorrow, and a probability $1 - \beta$ for the contrary. Then the Markov transition matrix will look like.

$$P = \begin{bmatrix} \alpha & 1-\alpha \\ 1-\beta & \beta \end{bmatrix}$$

We define it so that P_{ij} is the probability of transitioning from state i to state j. Markov chains follow the Markov property, defined as.

$$P\{X_{n+1} = j | X_n = i, X_{n-1} = i_1, .., X_1 = i_{n-1}\} = P\{X_{n+1} = j | X_n = i\}$$

So the probability of going to a state depends only on the current state. Any other information is superfluous and does not lead to any better prediction. The Markov process is often referred to as the memory-less process. For transition matrices in DTMC, each row sums to 1; that is a probability of 1 that we either leave or stay in the current state. To find one-step transitions, we can look at the matrix. For two steps, we need to consider the possible paths, say we start in state 1 and wish to know the probability of ending up in state 2 after 2 steps.

$$P^2_{1,2} = (1-\alpha)\beta + \alpha(1-\alpha)$$

We can either go straight to state 2 and stay there or stay for one step in state 1 and go to state 2. Raising the transition matrix to the power of two we have that.

$$P^2 = \begin{bmatrix} \alpha & 1-\alpha \\ 1-\beta & \beta \end{bmatrix} * \begin{bmatrix} \alpha & 1-\alpha \\ 1-\beta & \beta \end{bmatrix} = \begin{bmatrix} \alpha^2 + (1-\alpha)(1-\beta) & \alpha(1-\alpha) + \beta(1-\alpha) \\ \alpha(1-\beta) + \beta(1-\beta) & (1-\alpha)(1-\beta) + \beta^2 \end{bmatrix}$$

We see that $P^2_{1,2} = P^2_{12}$ the probability of ending in state 2 after 2 steps starting in state 1. This is true in general from the definition of matrix multiplication.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

We see that c_{ij} is the sum of all paths from i to j. The Chapman-Kolmogorov equations tells us that,

$$P^{m+n}_{ij} = \sum_{k=1}^{\infty} P^m_{ik} P^n_{kj}$$

where.

$$P^{m+n}_{ij} = P\{X_{m+n} = j | X_0 = i\}$$

When P is a transition matrix the expression reduces to matrix multiplication

$$P^{m+n} = P^m P^n$$

and in general an n step transition is given by P^n.

State classifications, a state j is *accessible* from state i if $P^n_{ij} > 0$ for some $n \geq 0$, this is not always a two way relationship, we can have what is referred to as an *absorbing* state after which no other state is entered like the system.

$$\begin{bmatrix} 0.1 & 0.5 & 0.4 \\ 0.8 & 0.1 & 0.1 \\ 0 & 0 & 1 \end{bmatrix}$$

In this example state 2 is accessible from state 1 and vice versa, we say that state 1 and 2 *communicate* . Additionally we see that both state 1 and 2 are *transient,*that is.

$$\sum_{n=1}^{\infty} P^n_{11} < \infty$$

Since the chain will eventually get stuck in state 3, a state is called *recurrent* if

$$\sum_{n=1}^{\infty} P^n_{ii} = \infty$$

A set of states that communicate are referred to as a *class,* if a Markov chain consists of multiple classes it is referred to as reducible, since different classes effectively divides the state space.

If we want to know the probability of being in some state far ahead in the future, say $n \to \infty$ we can in some cases simply multiply P with itself until it converges, but this is not generally possible as some Markov chains are periodic, that is a state might only be reachable every k'th step. Instead we define $\pi_j = 1/m_j$ the long run proportion of time spent in state j, where

$$m_j = E[N_j | X_0 = j]$$

Where

$$N_j = min\{n > 0 | X_n = j\}$$

That is m_j is the expected number of transitions after leaving state j to enter it again, this is a general valid definition for irreducible and recurrent Markov chains[35]. To find the vector π we note that.

$$\pi_j = \sum_i \pi_i P_{ij} \to \pi = P^T \pi$$

and that $\sum_i \pi_i = 1$, say

$$P = \begin{bmatrix} 0.3 & 0.1 & 0.6 \\ 0.3 & 0.5 & 0.2 \\ 0.4 & 0.5 & 0.1 \end{bmatrix}$$

then we can find π by creating

$$y = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

And

$$M = \begin{bmatrix} [P^T - I]_{1:2 \times 3} \\ 1, 1 \end{bmatrix}$$

We have that $(P^T - I)\pi = 0$ then

$$M\pi = y \to \pi = M^{-1} y \approx \begin{bmatrix} 0.33 \\ 0.37 \\ 0.3 \end{bmatrix}$$

In general we can find π with $y = [0, .., 0, 1]^T \in R^n$ and

$$M = \begin{bmatrix} [P^T - I]_{1:n-1 \times n} \\ 1, .., 1 \end{bmatrix} \in R^{n \times n}$$

Systems that change state continuously, require continuous-time Markov chains (CTMC); instead of jumps during discrete jumps, these Markov chains can transition at any time. However, since we still want to retain the Markov property, the time we spend at a state cant change the probability of changing the state that is.

$$P\{X(t+s) = j | X(t) = i\} = P\{X(s) = j | X(0) = i\}$$

The only continuous distribution with this property is the exponential distribution $\lambda e^{-\lambda t}$, having mean $E[t] = \frac{1}{\lambda}$, where λ is referred to as the rate. For CTMC we use what is called a rate matrix, whose rows sum to 0.

$$Q = \begin{bmatrix} -\lambda_{12} - \lambda_{13} & \lambda_{12} & \lambda_{13} \\ \lambda_{21} & -\lambda_{21} - \lambda_{23} & \lambda_{23} \\ \lambda_{31} & \lambda_{32} & -\lambda_{31} - \lambda_{32} \end{bmatrix}$$

So if $Q_{ij} < Q_{ih}$ then we expect to transfer to state h more often as the transition, on average, takes less time. The minimum of a set of exponential distributed random variables $\{x_1, .., x_n\}$with rates $\lambda_1, .., \lambda_n$ is exponentially distributed with mean.

$$E\{min\{x_1, .., x_n\}\} = \frac{1}{\lambda_1 + .. + \lambda_n}$$

So the higher the rates of a state, the less time we will spend in that state [30]. We can find the stationary distribution π for CTMC by using the constraints $Q^T\pi = 0$ and $\sum \pi = 1$, using a similar trick to the one above, we define

$$M = \begin{bmatrix} Q^T_{1:n-1\times n} \\ 1, ..., 1 \end{bmatrix} \in R^{n\times n}$$

And

$$y = [0, .., 0, 1]^T \in R^n$$

Then

$$\pi = M^{-1}y$$

The time evolution given by the Kolmogorov forward equation

$$\frac{dP(t)}{dt} = P(t)Q$$

which has the solution

$$P(t) = P(0)e^{tQ} = e^{tQ}$$

where e is the matrix exponential. Different interpretations of CTMC are equivalent, so we will not discuss them here.

2.3 Scientific Verification and Validation

Oberkampf and Roy mention four important parts of modeling and simulation credibility in [29].

Quality of the analysts In this case, the analyst is me, a master's student, which carries low credibility (quality). But the supervisor and institute have a lot of credibility, constituting the most important part of analyst credibility.

Quality of physics modeling The quality of physics modeling may not directly apply to this project, but in essence, it is the quality of model choices and comprehensiveness. Of course, comprehensiveness, in our case, is only important when it comes to generalizations of results. Still, our model is quite abstract, and one cannot assume that the results directly apply to some other system without good arguments for why it would be. This part of credibility is then covered by the model description in 3.1 and the extent to which that section covers modeling choices.

Verification & validation activities Validation is building the right model, while verification is building the model right [7]. In our case, validation could be done by using real robots, but it will not be conducted in this project. Verification, on the other hand, is the most crucial part of our credibility. We will discuss verification in greater depth shortly.

Uncertainty quantification & sensitivity analyses Uncertainty quantification is the process of quantifying elements of the model or model implementation that could cause uncertainties in the results. Model assumptions, initial conditions, etc., could cause these uncertainties. One could point to many potential uncertainties in our model, such as boundary shape, and how these would affect the translatability of our results to other experiments or studies. But as this project has a limited scope, we study the model described as is and don't consider uncertainty quantification. Sensitivity analysis is the process of connecting output uncertainties to all the parts that make up the model, usually conducted with uncertainty quantification. While also essential to modeling and simulation credibility, it is outside this project's scope.

As stated before, verification is building the model right. In our case, this refers to the software implementation created to run simulation experiments with the model. The precondition required to do verification is the model description given in 3.1. The implementation can only be as precise as the description. "There is no silver bullet to guarantee software correctness and, consequently, all available techniques for fault detection and correction should be used." [22], software verification is a complex topic, and many different approaches have been developed. The first defensive line against software errors is good programming practices, such as coding conventions. A great deal of effort did go into code design. However, since my expertise is questionable and the code is not reviewed by anybody else, its quality is questionable. In this project, agile programming was used, [29], as iterative software development suits small scientific projects well. Other software development methods would probably lead to higher correctness, such as the Vienna Development Method [22]; where an abstract model is refined to code level hand in hand with a formal specification. One can formally prove that the program is correct at each refinement step. This approach falls into the category of formal methods, where one uses rigorous mathematical approaches to prove that a program is correct regarding its specification (successful verification). While there are many tools to assist in such efforts like [4], with an automated theorem prover, this approach would be too time consuming given the scope of our project and expertise at hand. The most important part of our verification is black-box testing, where we consider the system a black box, designing a set of tests with inputs and expected outputs. Black-box testing is often the best way of finding faults, [22], but like with any test, it does not show that software is without fault; rather, tests' inability to detect faults increases our trust in the software implementation.

ABMS is unique regarding verification, as most ABMS is exploratory in nature [14], as is the case in our project. Being exploratory implies that we do not know all outputs for a given input. Even though running simulations do not create information [1] unavailable through the model definition, emergent behavior is not exempt from this either. But human insight is still generated, which is why we do modeling and simulation in the first place. We can design test cases for simpler parts of our system that provide confidence in the software implementation. [17] argues for testing ABMS at different levels, micro, meso and macro levels. At the micro level, we test individual agents or elements that make up the agents and environmental elements. At the meso level, we test agent interaction, such as communication, while macro-level tests consider the entire system. The exploratory nature of ABMS gives rise to one more challenge: artifacts [14] which are significant changes in system behavior created not by some implementation fault but by faulty model assumptions. A clear example of this in our case is agent activation; when agents are not randomly activated each step, they can form groups that outperform systems without this behavior. To deal with these artifacts and further inspect our system, we use visual verification to see how the agents move and interact with the environment in real-time. Visual verification is part of face validity. [17] While relying on the quality of the analysts, it has been a vital tool for us to asses program correctness, which the supervisor could easily inspect.

3 Method and Implementation

3.1 Model description

Agents are distributed uniformly in the search space. They search by choosing a random direction until they reach the boundary, where a new direction is determined uniformly, and the process repeats. This is achieved by each iteration for each agent checking if they are out of bounds and changing the required dimension of their velocity. Agents also check each iteration to see if they can see a task. If that is the case, they will move towards the task and flag themselves as on task. If multiple tasks are within the detection radius, which can happen after a task is solved or after initialization in pseudo-code:

Algorithm 1 search

```
$r_d$=detection radius
s=speed
l=side length of search space
$d_{ms}$=maximum distance at which a task can be solved
Initialize:

    current_task=Null
    on_task=false
    for each agent:
        1. agent.position=$[x_1, x_2] * l$,  $x_1, x_2 \sim U(0,1)$
        2. agent.velocity=normalize($[v_1, v_2]$) $* s$,  $v_1, v_2 \sim U(-1,1)$

for each iteration:

    for each agent:
        for i in each dimension:
            if agent.position.i>l
              agent.velocity.i=U(-1,0)
              agent.position.i=l
            else if agent.position.y<0
              agent.velocity.i=U(0,1)
              agent.position.i=0
            else
              agent.velocity.i=U(-1,1)
        agent.velocity=normalize(agent.velocity)*s
    if !on_task
        if any(d(agent,tasks)<$r_d$)

            current_task=task s.t d(agent,task)==min(d(agent,tasks))
            on_task=true
            agent.velocity=normalize(task.position-agent.position)*s
    elseif d(agent,current_task)<$d_{ms}$
        agent.velocity=0
```

tasks have an associated requirement to be solved, t_r while agents each have an associated potential a_p when $t_r \leq \sum_{i=1}^{n} a_p^i$ then a task can be solved. If all agents have equal potential then we can use $t_r \leq na_p$ where n is the number of agents. For each task, we check if the task is solvable for each iteration. When it gets solved, all agents are revealed of the task and rerun initialization, as seen above. The task is reset and receives a new random position, and a counter increases for the number of tasks solved.

Algorithm 2 Solving task

```
for each iteration:

    for each task:
        if $t_r \leq na_p$
            for each agent on task
                reinitialize(agent)
            reset(task)
            task_count+=1
```

To measure the performance of our systems we use a metric, the mean number of tasks solved per iteration. $t_s(i)$ the number of tasks solved at iteration i, them $E[t_s] = \frac{1}{n}\sum_i^n t_s(i)$ is our primary metric. It has some valuable properties, and it converges to a fixed value over time for any system where t_s is a distribution whose mean is smaller than i. In general, this will be true for most of our simulations. We don't expect a task or multiple tasks to be solved in each iteration or for the number of tasks solved in each iteration to accelerate. That is $d^2E[t_s] = 0$ in steady state, so for systems where this measure does not converge $dE[t_s] = k$ so we will get a constant slope with respect to i. For all of our simulations, we have a maximum number of tasks. If that number is 3, then at most $k = 3$, but this is usually not the case in search and task allocation systems, as the search takes time and so does solving the tasks.

For algorithms with communication, we need to decide on task inference capability. There are a number of things that agents can infer about a task and other agents. One is if agents can see task requirements at the same distance, they can detect a task. It is not evident that this would be the case as it is easy to imagine being able to sense a task and its direction with less information than what is required to tell the type of task and its requirements. If the system is set out to solve a single type of task, then the agents would hold this information. If there are multiple types of tasks, they might be unable to infer immediately. Sometimes it might be unknown, and they would have to figure it out.

A second inference is an ability to see if other robots are at the task, that is, within d_{ms}, say an agent detects the task and wants to start an auction. Selecting fewer agents than required will be optimal if no robots are already at the task. This inference can be sidelined by assuming that all agents behave the same and that we communicate globally. Then if an agent is on the task, the other agents would already know as an auction would be taking place. It is indeed vital when detection and task requirement inference distance is larger than communication distance. In general, starting an auction as early as possible sounds optimal not to waste time moving to the task, but that might be detrimental if it makes more agents than required quit their searching process.

Thirdly there is a possible inference about agents moving toward the task. In the most optimal system (assuming one also knows the agents at the task location), one would have this information before approaching a task or starting an auction. On the other hand, if enough agents are already moving toward a task to solve, then it is better to continue searching. For the performance of communication behaviors, it is essential to know how many agents to call.

-auction

-call-out

t_r	Task requirement, a scalar which is associated with each tasks, the amount of work

	required to solve a task
t_c	Task capacity, a scalar associated with each agent, the agents capacity to do work
n	Number of agents
n_t	Number of tasks
l	Side length of search area
r_d	Detection radius, associated with each agent, the distance at which a task can be detected
r_c	Communicated radius, associated with each agent, the distance at which agents can communicate
N_r	Number of runs, number of repeated simulations with same parameters
N_s	Number of steps or iterations per simulation

3.2 Software

Agent based modeling and simulation (ABMS) is used in a wide area of domains, from economics and industry to natural sciences and military applications. A recent review of ABMS tools is found in [2] looking at 85 distinct software tools. Constructing arguments for which tool choices are optimal in this project would be very time-consuming. Since most programming languages and many software libraries could support this project, it seemed the most time efficient to see if the software we were familiar with was satisfactory for our purpose, and indeed it was.

In this project, we use the programming language Julia [8] designed for high-performance and scientific computing with accessible parallel computing. Julia performs similarly to C [8] for single-core while dynamically and faster typed than most other languages. Julia also has many other features, such as meta-programming and directly calling code from other programming languages such as C, Python, R, Fortran, etc. Julia is a relatively new language, so calling on packages from other languages is an excellent way to plug the holes where they exist. The syntax of Julia is similar to Python's, so code development is relatively fast. In this project, a range of Julia packages was used. Agents.jl [11] is package for agent based simulation with a multitude of features while having far higher performance than Mesa, Netlogo and Mason [3]. Dr Watson [10] has also been of seminal help with running simulations and organizing the project. A large amount of other packages has also been used from file readers to statistical analysis to symbolic and numeric integration.

4 Simulation experiments, results and analysis

This section covers all experiments and results that were interesting. In addition, we analyze the results and construct a mathematical model. Before we can run experiments, we need to establish the credibility of our simulator.

4.1 Verification

We are interested in studying steady-state behavior. Transient behavior is largely subject to initial conditions and randomness, making it more difficult to understand. In addition, transient behavior is of little importance with longer-running systems. To start our verification and simulation process, we need to know when a steady state is reached with respect to our metric.

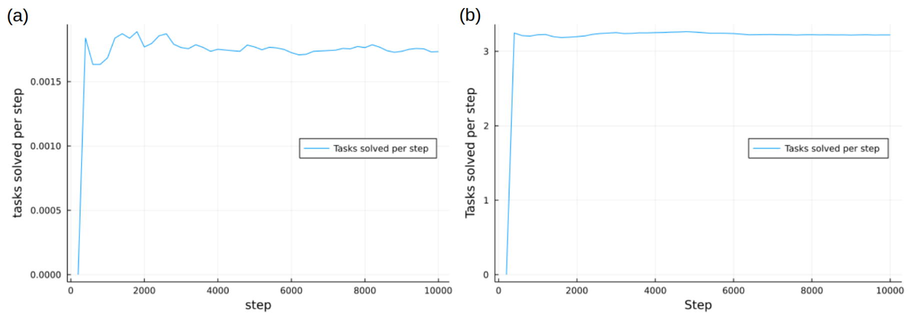


Figure 3: Time until steady state, both figures show 10k iterations with $N_r = 24$. (a) with one task, one agent, $r_d = 2.5$ we see steady state is reached at around 4k iterations. (b) with 30 agents, 5 tasks, $r_d = 10$ steady state is reached before 2k steps.

As seen in figure 3 systems that solve tasks more often reach steady state sooner, this is not surprising since with rarer events one needs more samples to estimate the mean. We will run most simulations with $N_r = 24$ and $N_s = 10k$ this is probably far more than required, meaning its almost guaranteed that we reach steady state with any parameter set. If a parameter set needs even higher N_r or N_s that will be mentioned. In figure 4 we see the performance of our system for different communication protocols, this results coincides well with [26] which did a master thesis on a similar topic. This serves as a good indication that our software is working correctly.

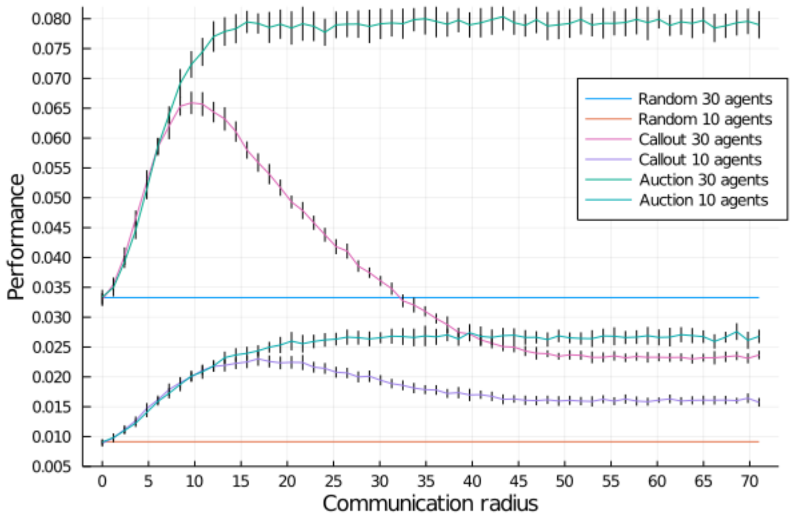


Figure 4: Validation of the simulator shows that auction performance is best, followed closely by call-out until a breakpoint is reached. Where the agents start to aggregate, and search capability falls.

Further, we have designed a set of black box tests at the micro,meso, and macro level, described in Table 1, In addition, we have data processing software. Therefore, the results will be evaluated concurrently as we use the data processing segment to make graphs showing our behavior and metrics measurements. The results can be seen in Figures 5 on the next page and 6 on page 20.

Table 1: Black-box tests

Parameter/Function	Metric	Test setup/expected result
speed,time step & step count	distance traveled	generate random values of velocity, time step, and step count. In simulations with no tasks. The distance traveled should be exactly velocity*time step*step count.
task detection radius	tasks solved	Simulations with increasing radius, at 0, tasks solved should be zero. Tasks solved should increase with task detection radius, until it covers the search area.
tasks solved	tasks solved	One task and one agent with full detection radius. The number of tasks solved should be equal to the number of time steps.
task requirement & task capacity	tasks solved	Simulations with one agent with full detection radius and one task. Let the agent have task capacities [1,..,n] and the task have requirement [1,..,n]. Then a heat map of the performance should be triangular, where one side has 0 performance and the other has a constant performance.
maximum solving distance	tasks solved	Simulations with increasing radius, performance should increase as radius increases.
Multi agent task requirement and capacity	tasks solved	Multiple agents with same task capacity, simulate span in number of agents,task capacity and task requirement. Agents should be able to solve tasks when $nt_c \geq t_r$ otherwise not.
communication radius	tasks solved	Run simulations of call-out,auction & random with increasing communication radius. At zero radius call-out and auction should be equal to random. Tasks solved increases for auction until it reaches steady state. Call-out performance increases until it peaks after which performance falls.

Auction	distance	Have one task and n agents, with task capacity 1, task requirement $<$n and full communication. After a task is solved wait with placing a new task so agents get to spread out. When a task is found, the distance to the agent who is the furthest away and won should on average be equal to the k'th order metric of the distance between one point and (n-1) others uniformly in the search space.

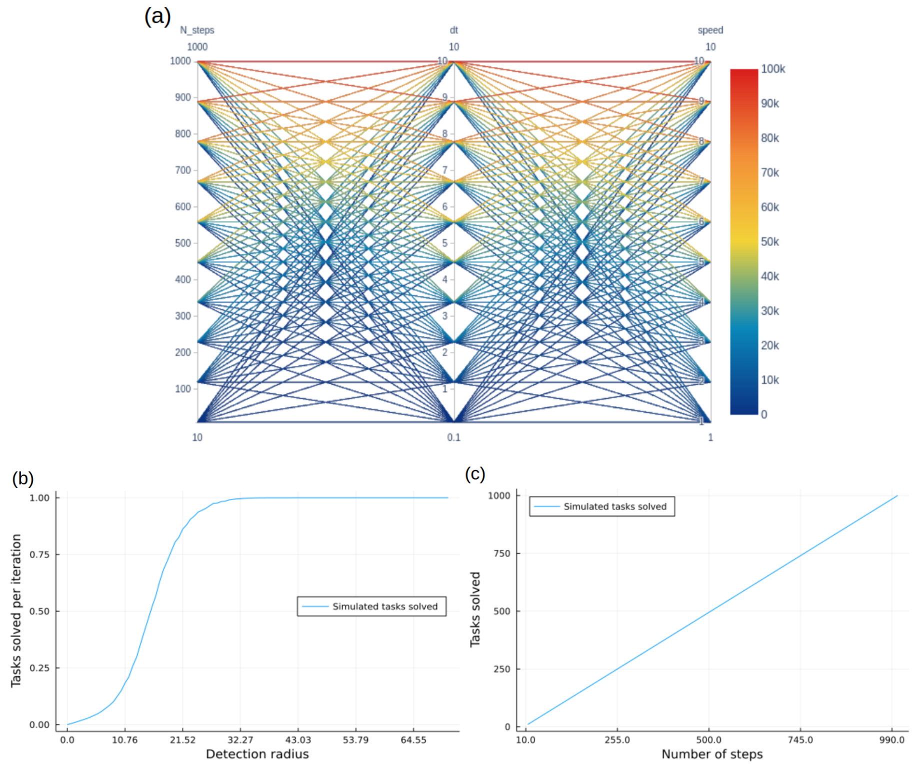


Figure 5: Verification tests (a) Parallel coordinate plot showing that total distance traveled is multiplicative with the number of steps, velocity, and step size. In (b), the number of tasks solved per iteration increases with the detection radius until one task is solved per iteration. (c) With a configuration that can solve one task each iteration, we see that a correct number of tasks are solved.

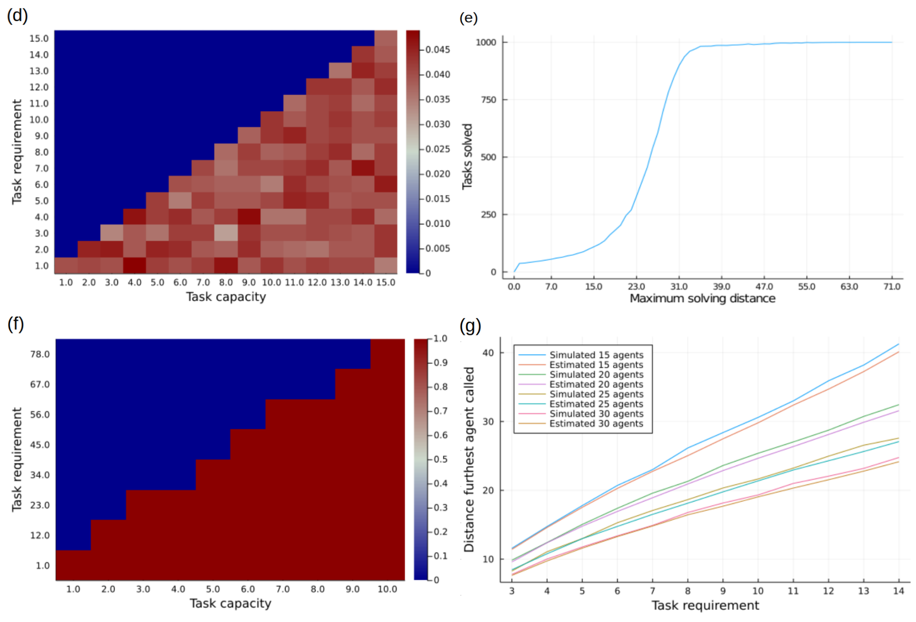


Figure 6: (d) We see that tasks only can be solved when task requirement $\leq$ task capacity. (e) As the maximum solving distance increases the agent spends less time traveling to tasks, and eventually solves them instantly. (f) With $n = 11$ we see that tasks are only solved when $t_r \leq nt_c$. (g) We see the distance to the furthest agent required to solve a task is approximately equal to what is expected with uniformly distributed points.

We see that all the results agree with our predictions, which increases our trust in the system.

Since the measure on our simulation will vary according to thousands of uniform random effects, the performance distribution on the same parameter set should be normally distributed. Showing this is true would also allow us to run statistical tests assuming the normal distribution, which will be very helpful later. If the results are not normally distributed, it probably hints at some error in the simulator. To find out whether the results are normally distributed we use the Shapiro-Wilk test [37], a widely used normality test, histograms and qq plots. Shapiro-Wilks is implemented by Pingouin package in Julia, using algorithm AS R94. We find it sufficient to choose one set of parameters, $r_d = d_m = 10$, $n = n_t = v = dt = 1, l = 50$. $N = 10000$ and $N_{runs} = 50000$ so we get 50000 points of data to plot. For Shapiro-Wilks we choose 200 random data points, which seems to return true every time the test is ran, with $W = 0.993565$ and $p = 0.53349$. Though the test started to fail for over 500 data points, normality tests generally do not deal well with too many samples as they weigh outliers too high. The results can be seen on figure 7

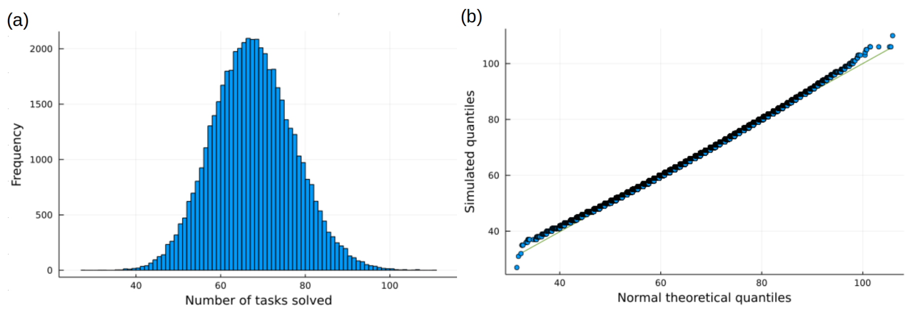


Figure 7: Evaluation of normality, (a)histogram and (b)q-q plot. We see that our data is normally distributed.

All taken into account, we believe our model implementation and, therefore, simulation results are credible within the limitations of the project

4.2 Parameter choices & initial conditions

[26] found that uniform initial distribution of agents led to a faster steady state. Therefore, we will use the same initial distribution in this project. Additionally, tasks are distributed uniformly.

To improve transferability and repeatability, we wish to normalize some parameters with respect to others. Normalization of detection radius, communication radius, and velocity concerning the side length of the search area seems natural, and there is no apparent reason why this would not work. So we generate a set of non-normalized and normalized data with equal parameters. Simulations with a continuous auction are slow with a low communication radius because a lot of computation is involved. We, therefore, only let agents hold a single auction. As we can see in the comparative plots in Figure 8 on the following page the normalization works well. We can assume that representing speed, detection radius and communication radius as a proportion of side length is transferable to any equivalent simulation with equal proportions.

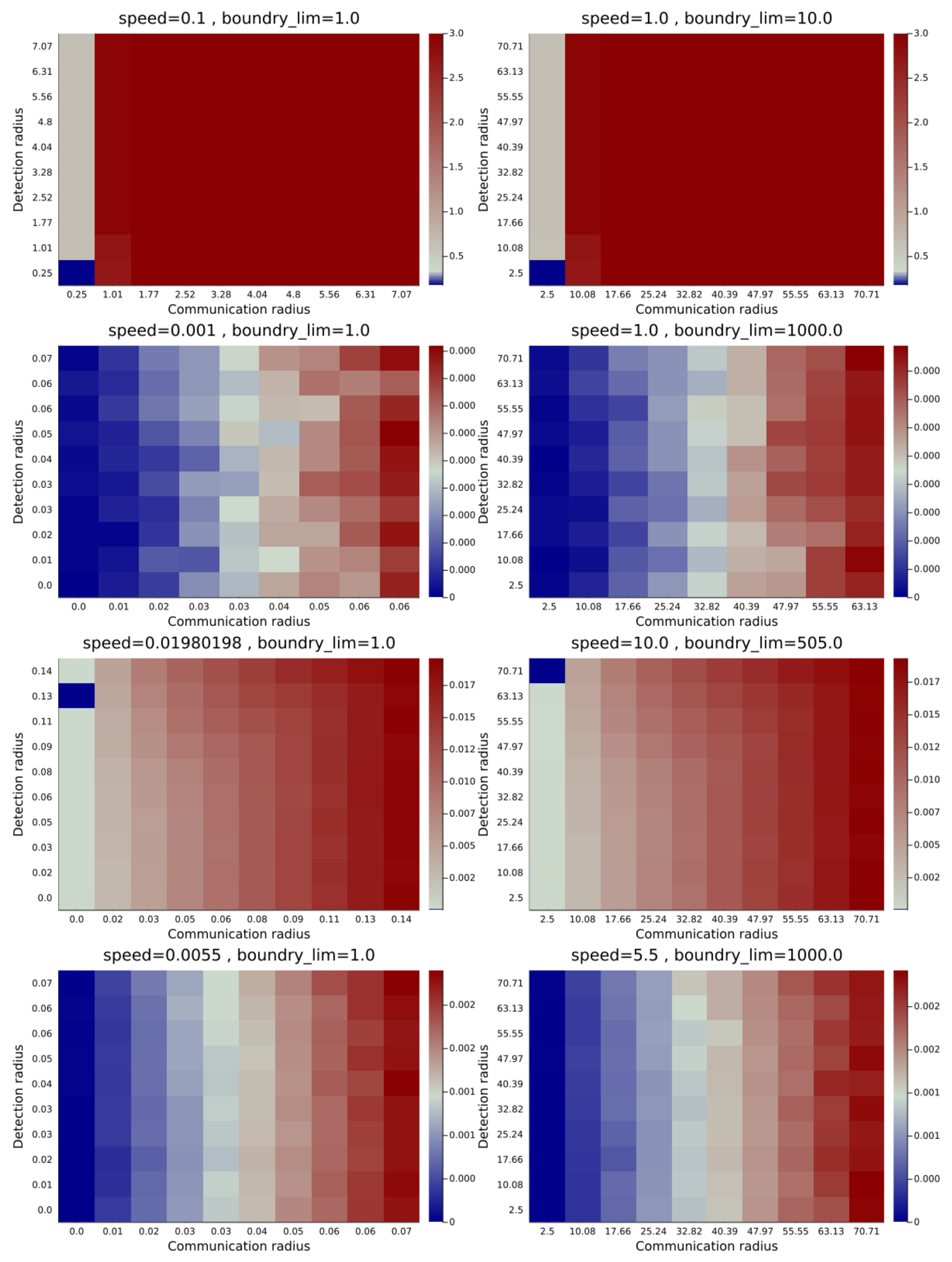


Figure 8: Here we see simulations ran with different speed and boundary limits (side length of search space) with communication and detection radius's that span from 2.5 to $50\sqrt{2}$. The normalized heat-maps are on the left, we see virtually no difference except for the occasional noise.

Another critical parameter is the time step, dt. Reducing dt increases the simulation's time resolution by reducing agent movement and increasing sampling frequency and communication frequency per time unit. However, reducing dt also shortens the effective length of a simulation if $dt = 0.1$ and $N_s = 1000$ the simulation runs for 100 time units, as we know from 4.1 the more tasks we solve per step the faster we reach steady state. When dt is low we might need to increase N_s to reach steady state, increasing computation time. Further when dt is high there is a chance of passing by a task without detecting it, we can analyze this phenomenon to increase our understanding of how the choice of dt affects the simulations.

For each time step the agents checks if a task is within their detection radius. We then have a sampling frequency

$$f_s = \frac{1}{dt}$$

sampling once every step. Given a sampling frequency, we encounter a problem; for any time step finitely bigger than 0, there is a probability a collision (overlap between detection area and task) will not be detected. Understanding the relationship between this probability and the parameters we choose for our simulator is essential, as it will affect our primary metric. The parameters that affect this problem are; sampling frequency, velocity, and detection radius. The tasks are treated as particles, while our agents move in straight paths.

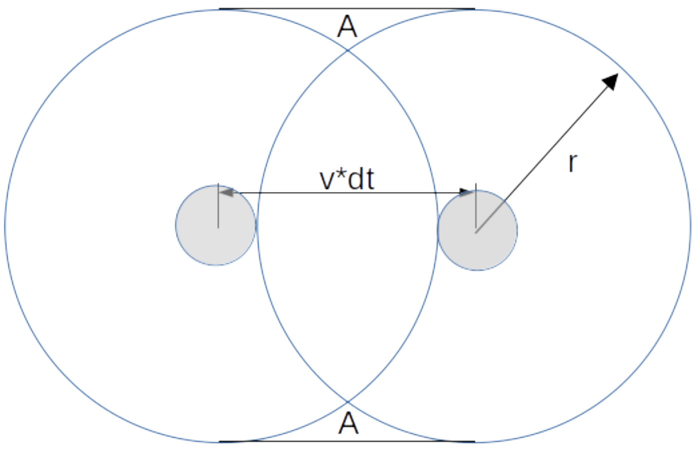


Figure 9: agent (gray circle) moves a distance vdt with detection radius r. A contains the area in which a task can stay undetected

In Figure 9 we see that the probability of a task going undetected per step is the areas in which a task can stay undetected divided by the newly sampled area. We find a function that adjusts for this probability, then run some simulations to see how well it accounts for variability in dt.

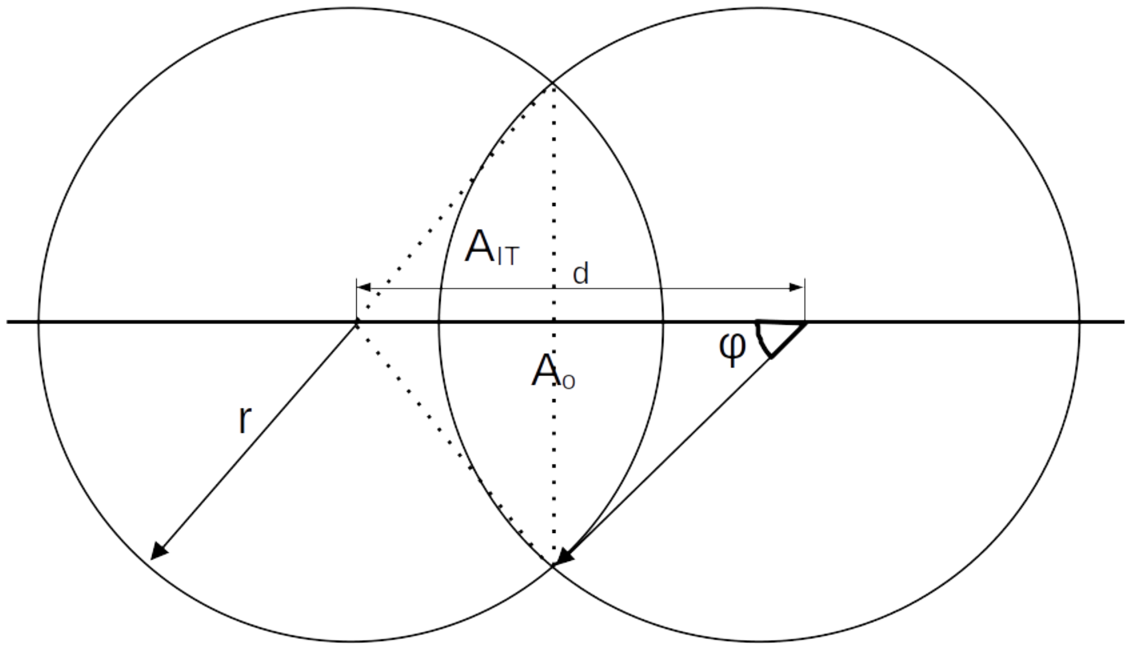


Figure 10: Here we see an agent moving a distance d, with detection radius r, A_0 is the overlap area, A_{IT} internal triangle,φ angle to detection radius intersection

We want to find A_0 in Figure 10, because of symmetry we know that the intersection point occurs at $\frac{d}{2}$ on the x-axis. Then

$$rcos(\varphi) = \frac{d}{2} \rightarrow \varphi = arccos(\frac{d}{2r})$$

, so the area of the circular segment is φr^2. We see that

$$A_{IT} = 2(\frac{1}{2}\frac{d}{2}rsin(\varphi)) = \frac{d}{2}rsin(arccos(\frac{d}{2r})) = \frac{d}{2}r\sqrt{1-(\frac{d}{2r})^2} = \frac{d}{2}\sqrt{r^2 - \frac{d^2}{4}} = \frac{d}{4}\sqrt{4r^2-d^2}$$

$\varphi r^2 - A_{IT}$ is half the area of the lens A_0, then

$$A_0 = 2(\varphi r^2 - A_{IT}) = 2r^2 arcos(\frac{d}{2r}) - \frac{d}{2}\sqrt{4r^2-d^2}$$

Then the new area we discover after moving a distance d is

$$A_d = \pi r^2 - 2r^2 arcos(\frac{d}{2r}) + \frac{d}{2}\sqrt{4r^2-d^2}$$

Knowing A_0 we can find A in Figure 9 on the previous page by noting that

$$A_1 = \frac{4r^2 - \pi r^2}{4} = \frac{r^2}{4}(4-\pi)$$

is one fourth of the area enclosed between a circle of radius r and the square that bounds it. This area contains a proportion of A, we can find $\frac{A}{2}$ by removing the extra enclosed area A_2. We can find A_2 by first making a rectangle from $\frac{d}{2}$(mid point between the circles) to where the circle intersect the x axis, giving it a width of $r - \frac{d}{2}$, then we give it a height r. This rectangle contains the extra area in A_1 in addition to one fourth of the lens between the circles. We then see that

$$A_2 = r(r - \frac{d}{2}) - \frac{A_0}{4}$$

so

$$A = A_1 - A_2 = \frac{r^2}{4}(4-\pi) - r(r-\frac{d}{2}) + \frac{1}{4}(2r^2 arcos(\frac{d}{2r}) - \frac{d}{2}\sqrt{4r^2-d^2}) = \frac{rd}{2} + \frac{1}{2}r^2 arcos(\frac{d}{2r}) - \frac{r^2\pi}{4} - \frac{d}{8}\sqrt{4r^2-d^2}$$

.

We can now adjust for the probability to not see a task by

$$\frac{A}{A_d} = \frac{\frac{rd}{2} + \frac{1}{2}r^2 arcos(\frac{d}{2r}) - \frac{r^2\pi}{4} - \frac{d}{8}\sqrt{4r^2-d^2}}{\pi r^2 - 2r^2 arcos(\frac{d}{2r}) + \frac{d}{2}\sqrt{4r^2-d^2}}$$

.

In Table 2 we run simulations with different values of dt, with $N_r = 100$, n=10, $n_t = 10$ and v=1. For every simulation the amount of total time is kept constant, so with $N = 10000$ for dt=1, the equivalent total time for $dt = 20$ gives N=500

Table 2: tasks solved total

dt	$r_d = 2.5$	$r_d = 2.5$,adjusted	$r_d = 10$	$r_d = 10$,adjusted
0.01	1769	1769	8229	8229
0.1	1768	1768	8172	8172
1	1714	1725	7712	7715
r_d	1565	1639	3755	3933
$2r_d$	1226	1687	2159	2971

When $r_d = 2.5$ we can adjust fort dt quite well. When $dt = 2r_d = 5$ we get $N = 2000$, and most of the loss here seems to be because we don't detect tasks we could have with a lower dt. When the probability of finding a task per sampling is high, then reducing the number of samples has a much higher cost on performance than the increase in dt. When $r_d = 10$ and dt=1 we solve a task almost every iteration, increasing dt increases the amount of area sampled per iteration, but since the probability of finding a task is already high it cant make up for the loss in total number of samples. We get the same results when varying v and keeping dt constant, if there was a fixed distance at which tasks could be solved that is $r_d \neq d_m$ then high vdt might cause the agents to jump around the tasks greatly reducing efficiency. Lastly we could have a separate sampling frequency, but as we have no specific target values for sampling frequency this makes little sense and will increase computation time. A good trade off seems to be keeping $dt = 1$, so we will use that value unless otherwise mentioned.

4.3 Modeling search

In this section, search is modeled separately from task allocation. Search on its own is the process of finding a task. After an agent finds a task, it is solved immediately. This implies that the max solving distance. $d_m = r_d$ and that the task requirement $t_r = 1$. While one can model a multi-agent system where the agents must also travel to the task itself, it is not considered part of the search process here. Though the final model does extend to cases like this. Data is handled as follows unless otherwise stated, the results from N_r runs is stored in a csv file along with the parameters of that run. This can be transformed into a matrix $M \in R^{nxN_r}$ with n recorded values and N_r columns, then we average the final recorded value, that is performance$=\sum_{i=1}^{N_r} M[n,i]\frac{1}{N_r}$. One could also average the last few recorded values, but it does create some challenges so it is avoided unless mentioned. Simulations are ran with $l = 50$ and $v = 1$, the normalized equivalent is $l = 1$ and $v = 0.02$ unless otherwise stated.

4.3.1 single task single agent

In this scenario $n = 1$, $n_t = 1$,agents sampling the same area repeatedly introduces some modeling challenges, which will be addressed later. But for the time being, each agent samples the area around it after it has moved a distance of $2r_d$, vastly reducing correlation between sample's. The data is scaled by $\frac{2r_d}{vdt}$ o make it easier to compare and represent. It makes the data look like the sampling frequency equals dt. Since we solve tasks instantly after they are found from a distance, the number of tasks solved per iteration should equal the number of tasks found per iteration. We can estimate the probability of finding a task by the proportion of the search space sampled. We call the predictive function H.

$$H_1(r_d, l) = min(\frac{\pi * {r_d}^2}{l^2}, 1) \tag{1}$$

since

$$H_1(r_d, l) = 1 \rightarrow r_d = \sqrt{\frac{l^2}{\pi}} \approx 0.564189584$$

would limit the possible values of r_d. Further r_d is normalized by l so

$$H_1(r_d, l) = min(\frac{\pi * {r_d}^2}{l^2}, 1) = min(\pi r_d^2, 1) = H_1(r_d)$$

to simplify further expressions. A data adjusting function B is also defined.

$$B_1(r_d, v, dt) = \frac{2r_d}{vdt} \tag{2}$$

The full prediction is then done by the combination of applying B to the data and using H to predict output after application, the pair (H_i, B_j).

As the data produced is normally distributed, confidence intervals via Student's-t distribution is used. Detection radius is normalized to between 0 and $\sqrt{2}$ as shown before to be possible.

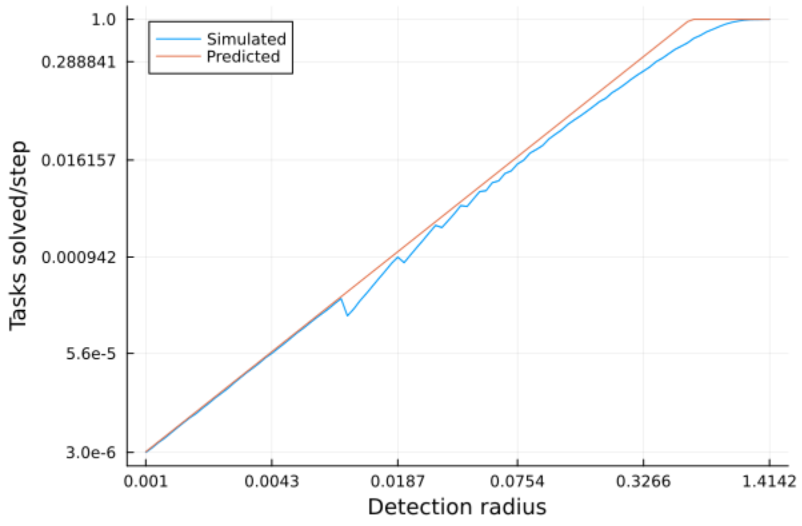


Figure 11: predicted and simulated results with (H_1, B_1)

In Figure 11 the simulation is ran with $dt = 0.01, N = 1000000, N_r = 1000$. The prediction looks reasonably close for small values of r_d, but there is some artifacts in the data causing negative spikes in performance . Since $dt > 0$ the sampling doesn't always occur after an agent has moved 2r, but instead when the distance moved is between 2r and $2r + dt * v$. The number of samples in a continuous simulation would be

$$f_c = \frac{vT}{2r_d}$$

in the discrete case we have

$$f_d = floor\left(\frac{vT}{dt * ceil\left(2r_d\frac{1}{dt}\right)}\right)$$

Where $T = Ndt$, the total simulation time. Assuming that the performance of this system is directly proportional to the number of samples an agent takes, this can be corrected by multiplying the result with $\frac{f_c}{f_d}$. Then

$$B_2(r_d, v, N, dt) = \frac{2r_d}{vdt}\frac{f_c}{f_d} \tag{3}$$

with (H_1, B_2) we get.

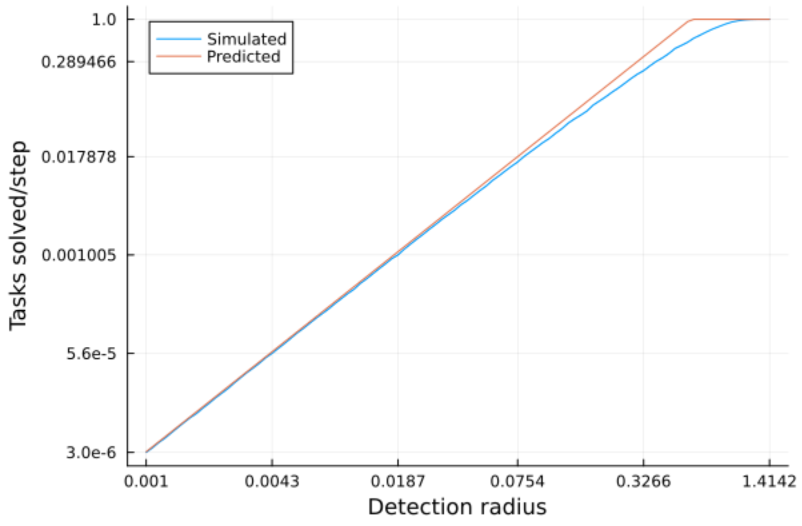
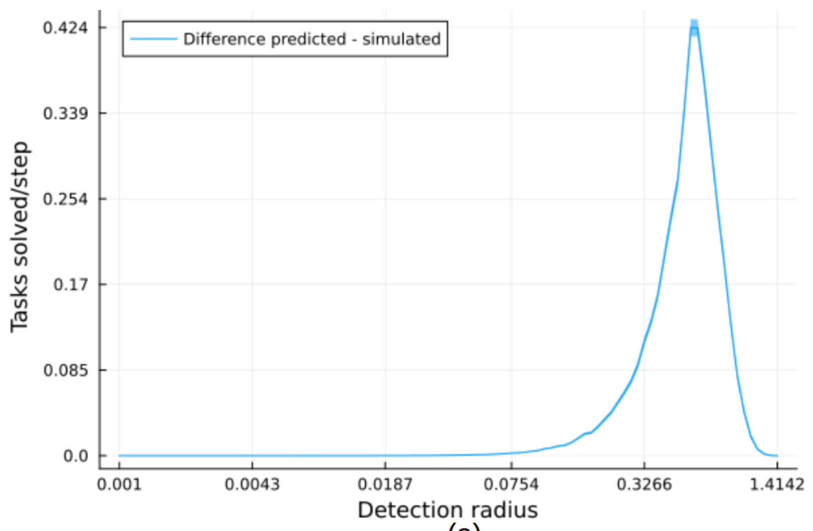


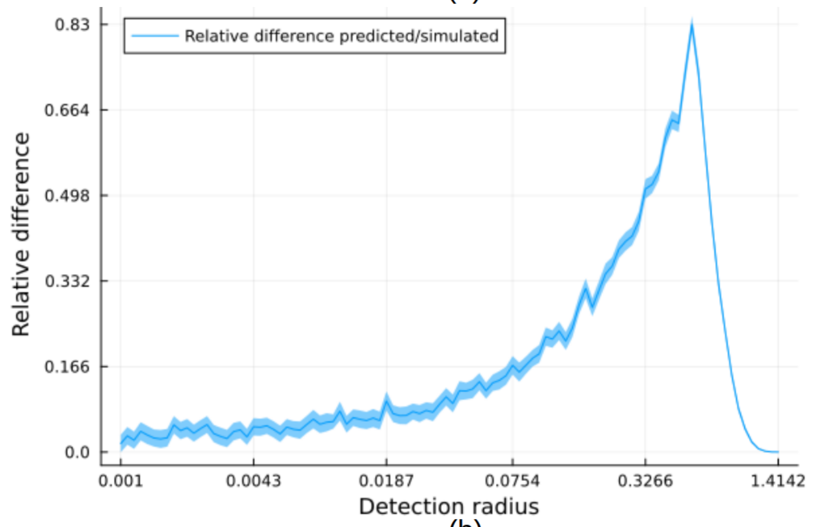
Figure 12: Adjusted simulation vs prediction

in Figure 12 results with adjustment are presented, H looks to be a good predictor for single-task single-agent search (STSA-S) with a low detection radius. Here there is an average relative difference (error)

$$\frac{1}{N}\sum_{i}^{N}|\frac{predicted[i]-simulated[i]}{simulated[i]}| \approx 0.17$$



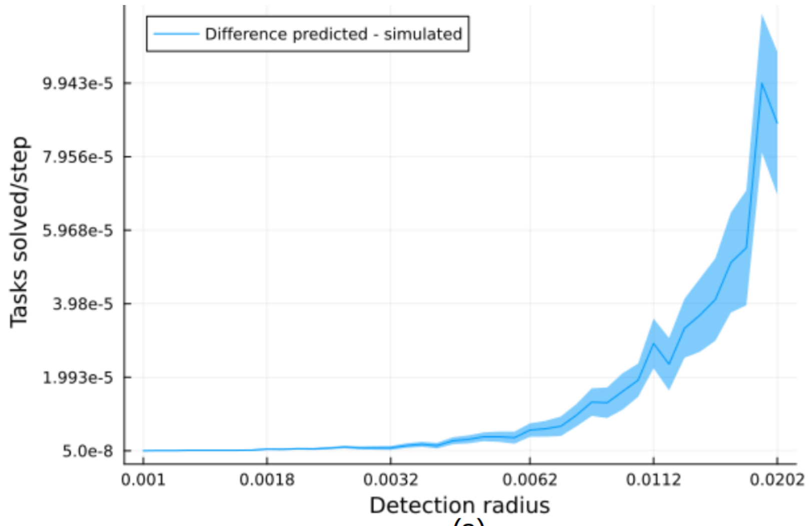
(a)



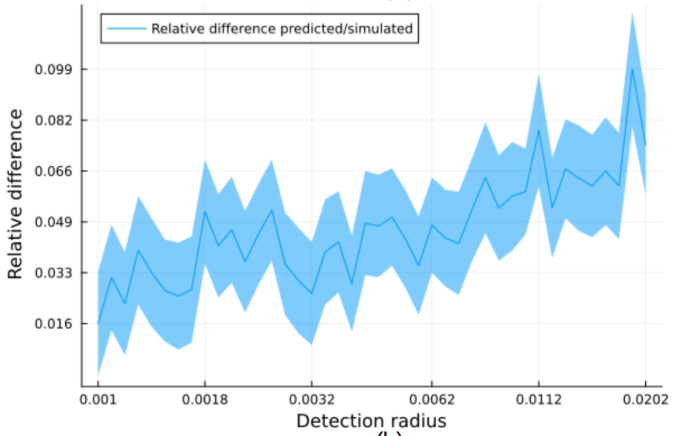
(b)

Figure 13: (a) difference between predicted and simulated, (b) relative difference. In both metrics the difference is low when r is either very low or very high

As seen in Figure 13 (H_1, B_2) is a good approximation of detection probability when the detection radius of agents is below 0.15 or over 0.95. Most conceivable real-world multi-agent search systems would operate with a low detection radius, so (H_1, B_2) will be a good predictor for the performance of these systems. To investigate further when this prediction might be used, plots for lower detection radii are made.



(a)



(b)

Figure 14: (a) Difference between predicted and simulated (b) Relative difference predicted and simulated

Figure 14 Shows that (H_1, B_2) is a very good predictor when it comes to tasks solved at low detection radii. Since the system solves several tasks, the relative difference does not make up a significant actual difference. (H_1, B_2) will most likely improve as a predictor as r_d gets smaller. While (H_1, B_2) can be used for systems with low detection radius, a more general predictor would be useful to more systems.

Instead, the expected distance between uniform points is used since the tasks are placed uniformly, and agents are approximately uniformly distributed over time .In [31] we find the PDF

$$g_s(s) = \begin{cases} -2\frac{\sqrt{s}}{a^2b} - 2\frac{\sqrt{s}}{ab^2} + \frac{\pi}{ab} + \frac{s}{a^2b^2}, & 0 < s \leq a^2 \\ -2\frac{\sqrt{s}}{a^2b} - \frac{1}{b^2} + \frac{2}{ab}arcsin\left(\frac{a}{\sqrt{s}}\right) + \frac{2}{a^2b}\sqrt{s-a^2}, & a^2 < s \leq b^2 \\ -\frac{1}{b^2} + \frac{2}{ab}arcsin\left(\frac{a}{\sqrt{s}}\right) + \frac{2}{a^2b}\sqrt{s-a^2} - \frac{1}{a^2} & \\ +\frac{2}{ab}arcsin\left(\frac{b}{\sqrt{s}}\right) + \frac{2}{ab^2}\sqrt{s-b^2} - \frac{\pi}{ab} - \frac{s}{a^2b^2} & b^2 < s \leq a^2+b^2 \end{cases}$$

for the squared distance between two uniform points in a box with side lengths $a < b$. In our simulator $a = b = l$ so we can omit the middle term and rewrite.

$$g_s(s) = \begin{cases} \frac{\pi}{l^2} + \frac{s}{l^4} - 4\frac{\sqrt{s}}{l^3} & 0 < s \leq l^2 \\ \frac{4}{l^2} arcsin\left(\frac{l}{\sqrt{s}}\right) + \frac{4}{l^3}\sqrt{s - l^2} - \frac{2}{l^2} - \frac{\pi}{l^2} - \frac{s}{l^4} & l^2 < s \leq 2l^2 \end{cases}$$

The distribution of the squared distance is of little use, so $g_d(d)$ where $d = \sqrt{s}$ is derived as follows.

$$P(\sqrt{S} \leq d) = P(S \leq d^2) = G_{\sqrt{S}}(d) = G_S(d^2)$$

Then

$$g_d(d) = g_{\sqrt{S}}(d) = \frac{d}{dd}G_S(d^2) = 2dg_s(d^2)$$

So

$$g_d(d) = 2dg_s(d^2)$$

where d is distance. then

$$g(d) = \begin{cases} 2d\left(\frac{\pi}{l^2} + \frac{d^2}{l^4} - 4\frac{d}{l^3}\right) & 0 < d \leq l \\ 2d\left(\frac{4}{l^2} arcsin\left(\frac{l}{d}\right) + \frac{4}{l^3}\sqrt{d^2 - l^2} - \frac{2}{l^2} - \frac{\pi}{l^2} - \frac{d^2}{l^4}\right) & l < d \leq l\sqrt{2} \end{cases}$$

The CDF will give the probability that a uniform point is within a distance d from another point, making it more useful,

$$G(d) = \int g(d)dd$$

we have that

$$g(d)_1 = 2d\left(\frac{\pi}{l^2} + \frac{d^2}{l^4} - 4\frac{d}{l^3}\right) = \frac{1}{l}2\frac{d}{l}\left(\pi + \frac{d^2}{l^2} - 4\frac{d}{l}\right)$$

so we can substitute $x = \frac{d}{l}$

$$G(x)_1 = \int 2x\left(\pi + x^2 - 4x\right)dx$$

and find that

$$G(x)_1 = \frac{x^2\left(3x^2 - 16x + 6\pi\right)}{6} + C$$

but $C = 0$ since we want $G(0)_1 = 0$,then

$$G(x)_1 = \frac{x^2\left(3x^2 - 16x + 6\pi\right)}{6}; 0 \leq x \leq 1$$

.

For $G(d)_2$we have

$$G(d)_2 = 2d\left(\frac{4}{l^2} arcsin\left(\frac{l}{d}\right) + \frac{4}{l^3}\sqrt{d^2 - l^2} - \frac{2}{l^2} - \frac{\pi}{l^2} - \frac{d^2}{l^4}\right)$$

$$= \frac{1}{l} 2 \frac{d}{l} \left(4arcsin\left(\frac{l}{d}\right) + 4\sqrt{\left(\frac{d}{l}\right)^2 - 1} - 2 - \pi - \frac{d^2}{l^2} \right)$$
$$= \frac{1}{l} 2 \frac{d}{l} \left(4arcsin\left(\frac{l}{d}\right) + 4\sqrt{\frac{d^2 - l^2}{l^2}} - 2 - \pi - \frac{d^2}{l^2} \right)$$

so again we can substitute $x = \frac{d}{l}$,then

$$G(x)_2 = \int 2x \left(4arcsin\left(\frac{1}{x}\right) + 4\sqrt{x^2 - 1} - 2 - \pi - x^2 \right) dx$$

$$= \frac{16\left(x^2 - 1\right)^{\frac{3}{2}} - 3x^4 + x^2\left(24arcsin(\frac{1}{x}) - 6\pi - 12\right) + 24x\sqrt{1 - \frac{1}{x^2}}}{6} + C; 1 \leq x \leq \sqrt{2}$$

Since we want $G(\sqrt{2})_2 = 1$ we have that $C = \frac{1}{3}$ then

$$G(d) = \begin{cases} \frac{x^2\left(3x^2 - 16x + 6\pi\right)}{6} & 0 \leq x \leq 1 \\ \frac{16\left(x^2-1\right)^{\frac{3}{2}} - 3x^4 + x^2\left(24arcsin(\frac{1}{x}) - 6\pi - 12\right) + 24x\sqrt{1-\frac{1}{x^2}}}{6} + \frac{1}{3} & 1 \leq x \leq \sqrt{2} \end{cases} \tag{4}$$

This distribution arises from the convolution of multiple uniform random numbers. For example, say we throw an n-sided die two times, and then in a coordinate system, mark the first throw on the x-axis and the second on the y-axis. Calling this our first point, then we repeat creating a second point. Now we can measure the distance between these points as illustrated in Figure 15.

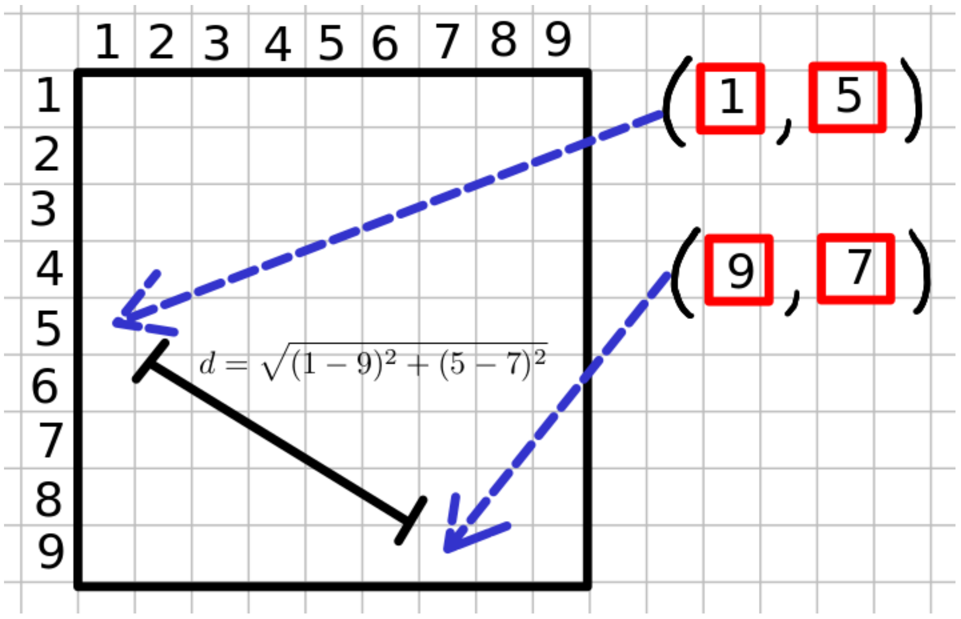


Figure 15: 4 dice rolls creating two points (1,5) and (9,7) with the distance between them given by d

We can double check that G(d) follows this same behavior with the simple Algorithm 3 then plot the relative difference between its output and the result given by G(d)

Algorithm 3 Distance between uniform points

```
N #number of repeats
m # distance resolution
dists=zeros(m) #initialization of distance vector
for _ in 1:N
    P1=[U(0,1),U(0,1)] #Point by two draws from the uniform distribution
    P2=[U(0,1),U(0,1)]
    d=distance(P1,P2)
    index=Int(floor(d*(m)))+1 # Index that corresponds to distance d
    dists[index]+=1 #add one to distance vector at index
end
dists/=sum(dist) #normalize dists
D=[sum(dists[1:i]) for i in 1:m] # Cumulative distribution of dists
```

To compare the results given by Algorithm 3 with $g_d(d)$ and $G_d(d)$ they first need to be averaged over the intervals that dists uses.

Algorithm 4 Average function over intervals

```
res=1000000 #resolution to approximate continuity
d=LinRange(0,$\sqrt{2}$,res) # vector of evenly spaced values from 0 to $\sqrt{2}$
m=100 #number of intervals
f_vec=zeros(m) #Initialization of results vector
f(d)=$g_d(d)$ # the function that is being averaged as a vector
for i in 1:m
    ind_min=$\frac{(i-1)N}{m}+1$
    ind_max=$\frac{iN}{m}+1$
    f_vec[i]=sum(f(d)[ind_min:ind_max])$\frac{m}{N}$
end
return f_vec
```

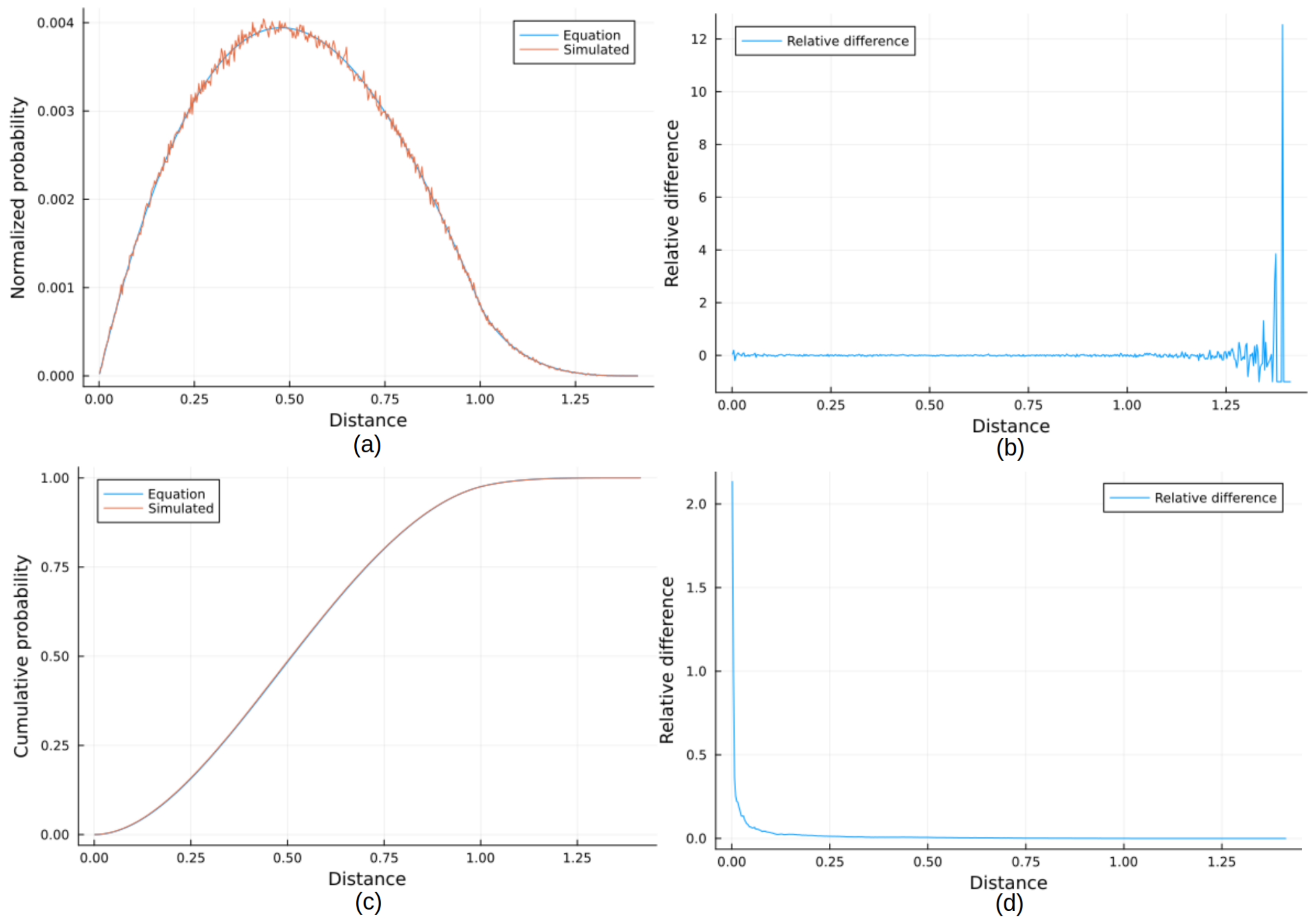


Figure 16: (a) Simulated distance distribution by 3 and average normalized(AN) $g_d(d)$. (b) Relative difference between simulated and $g_d(d)$ (AN). (c) Averaged $G_d(d)$ and simulated cumulative distribution. (d) Relative difference between averaged $G_d(d)$ and simulated cumulative distribution.

The simulated distribution is normalized by its sum, hence, $g_d(d)$ must be normalized by its sum. In addition Algorithm 4 is used, so the distribution is averaged and normalized.As seen in Figure 16 both cumulative and normal distributions overlap almost exactly. There is some difference at the endpoints in the relative difference plots, but this is most likely caused by the low probability of having good statistics with such low and high distances. The difference that we can see mostly highlights a need for mathematics when things become very costly to compute, and it is quite convincing that $g_d(d)$ follows the behavior described.

Now predictions are done with

$$H_2(r_d) = G_d(r_d) \tag{5}$$

and B_2

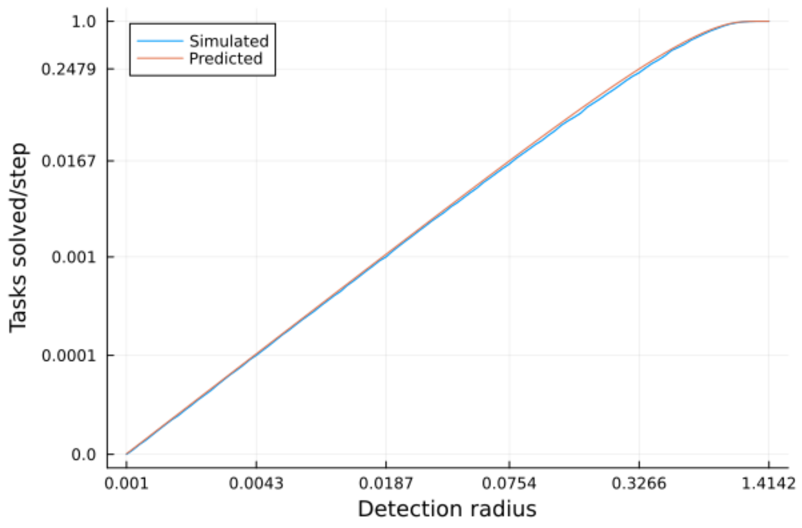


Figure 17: Simulated with 95% confidence interval and sampling error correction vs predicted with g_d

As seen in Figure 17 the prediction fits much better over the detection radius range. Relative average error is

$$\frac{1}{n}\sum abs\left(\frac{predicted(i) - simulated(i)}{simulated(i)}\right) \approx 0.058$$

.

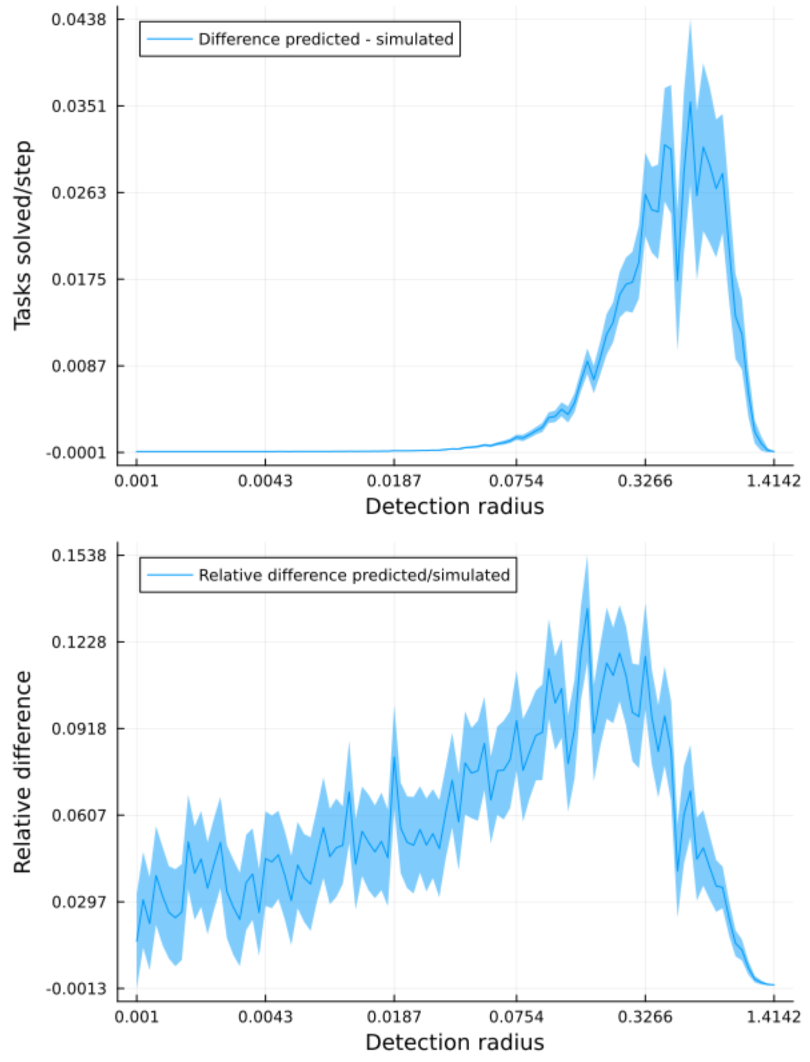


Figure 18: relative difference and difference between prediction and simulation with 95% confidence interval

In Figure 18 the relative error decreases as the radius increases. The error still present might be caused by agents sampling the same area, when r_d gets large the probability that a loss caused by overlap will matter decreases, since tasks are detected anyways. The difference between the two predictive functions gets smaller as r_d gets smaller,

$$\frac{G_d(r_d)-H_1(r_d)}{H_1(r_d)} = \frac{\frac{r_d^2(3r_d^2-16r_d+6\pi)}{6}-\pi r_d^2}{\pi r_d^2} = \frac{3r_d^2-16r_d}{6\pi} \ ,$$

$$\text{then } \lim_{r_d \to 0}\left(\frac{G_d(r_d)-H_1(r_d)}{H_1(r_d)}\right) = 0.$$

As an example, when $r_d = 0.001$ the relative difference between the predictions are ≈ 0.00085, which for most uses is undetectable.

A reasonable explanation for the error is the overlap between the agents, as there is a loss when an area is re-sampled. Assuming that the agent position is uniform, the location of two samples from the agent is uniform. This is not the case but for larger agents moving $2r_d$ a boundary interaction will change its direction uniformly, and then the distance from the previous sampling location will have some distribution. Assuming this distribution is random, we can model the expected overlap.

We know that the loss from two agents overlapping is

$$A_{ol}(r_d, d) = \frac{A_o}{2} = r_d^2 arcos(\frac{d}{2r_d}) - \frac{d}{4}\sqrt{4r_d^2 - d^2}$$

(see equation 4.2) since the points are uniform we can use

$$G_d(d)A_{ol}(r_d, d)$$

as the probability weighted loss given d and r_d. Then the integral

$$E_{ol}[r_d] = \frac{1}{2r_d}\int_0^{2r_d} G_d(d)A_{loss}(r_d, d)dd$$

is the expected overlap loss given r_d, using SageMath [40] we symbolically integrate and find that

$$E_{ol}[r_d] = -\frac{2}{3}\pi r_d^6 + \frac{128}{75}r_d^7 + \frac{16}{9}\pi r_d^5 - \frac{16}{105}\left(8r_d^6 + 7\pi r_d^4\right) r_d; r_d < 0.5$$

For $r_d > 0.5$ we need to solve this integral numerically. There is of course a probability that three samples will overlap without a task being found, but that probability is very low so its excluded here. To incorporate the loss r_d is corrected,

$$r_d^* = \sqrt{\frac{\pi r_d^2 - E_{ol}[r_d](1 - G_d(r_d))}{\pi}} \tag{6}$$

the loss needs to be multiplied by

$$(1 - G_d(r_d))$$

since its only relevant if a task is not detected. then the update prediction is given by

$$H_3(r_d) = G_d(r_d^*) \tag{7}$$

.

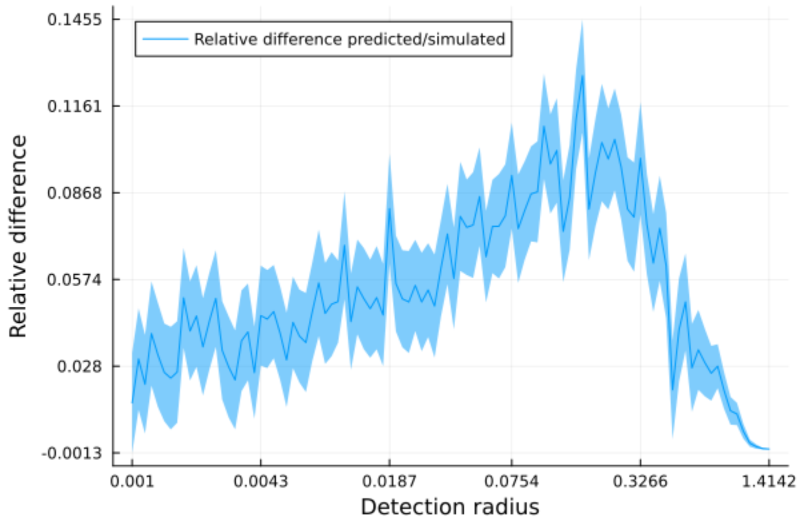


Figure 19: Prediction adjusted with expected overlap given uniform distribution

in Figure 19 relative difference is slightly reduced, mostly for higher values of r_d the average error is now 0.054%

To further investigate where the prediction error might stem from, the actual positional distribution of the agents is recorded. They are not completely uniform, as assumed in the model. The agent distribution is reconstructed by setting up a $m \times n$ grid and recording when an agent is within each box. This can be done for different parameter sets by normalizing the position matrix s.t $\sum_{i=1}^{m} \sum_{j=1}^{m} M_{ij} = 1$We obtain a 2-dimensional PDF representing the probability that an agent is within one of the boxes. With M, rejection sampling is used to simulate the probability of finding a task where agents have a distribution given by M. The description of this algorithm is given at algorithm 5.

Algorithm 5 Detection probability simulation with agent distribution M and uniform task distribution

```
$M \in R^{nxm}$ positional PDF for agents
r #detection radius
z=maximum(M) # maximum value of M used for rejection sampling
N #number of iterations
c=0 #counter
ns=0 #tasks solved
for 1:N
    i=U(1,n) #Uniform random index
    j=U(1,m) #Uniform random index
    k=U(0,z) #Uniform random acceptance value
    task_position=[U(0,1),U(0,1)]
    if $M_{i,j} \leq k$ #if k is greater use the position $M_{i,j}$
        c+=1 #increment counter
        #if task is within radius solve task and increment task counter
        if distance($M_{i,j}$,task_position)$\leq r$
            ns+=1
        end
    end
end
return $\frac{ns}{c}$
```

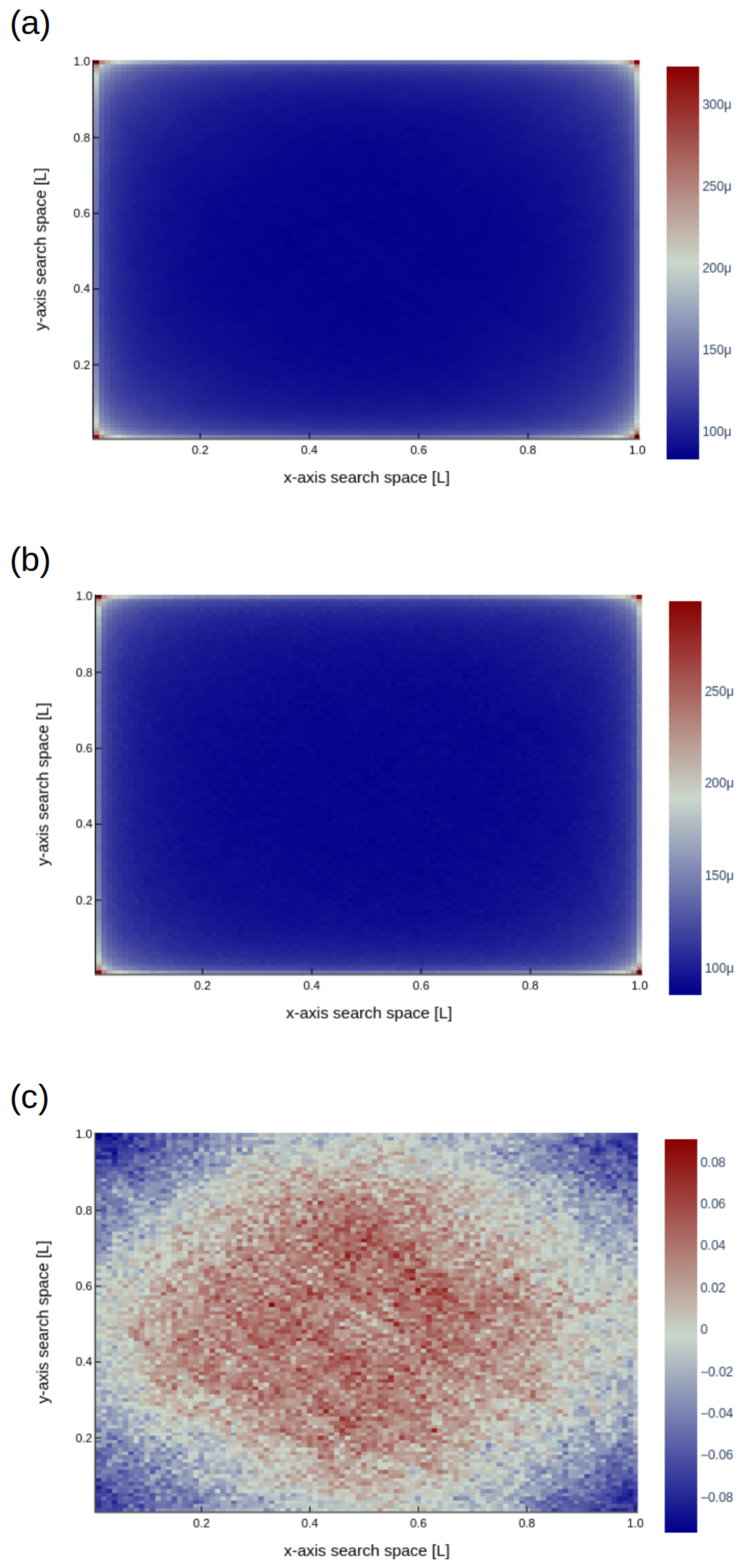
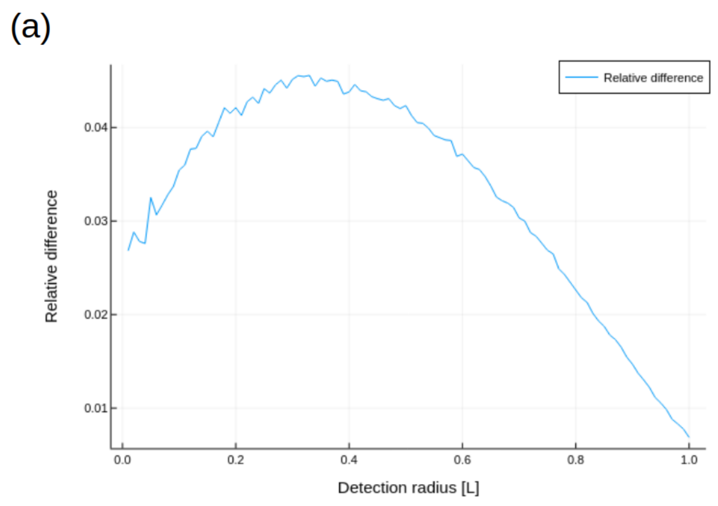


Figure 20: (a) Distribution with n=1,n_t =1 and r=0.05 (b) distribution with n=10,n_t =3 and r=0.4 (auction mechanism) (c) relative difference between the distributions where positive values indicate a higher probability of finding agents from (b)

As seen in Figure 20 systems with a higher density of tasks and agents spend more time in the middle of the search space compared to systems with a lower density of tasks and agents. This is expected as agents change direction when they find or are called to a task via the auction mechanism. Agents that do not find tasks must move to the boundary to change direction. To investigate if the actual agent distribution can be used to account for the modeling error directly, simulations using this distribution can be compared to the model. Suppose the difference and relative difference of the simulations with the approximate distribution matches that of the simulation vs. model. In that case, it is a strong indication that the error comes from the assumption of uniform distribution.



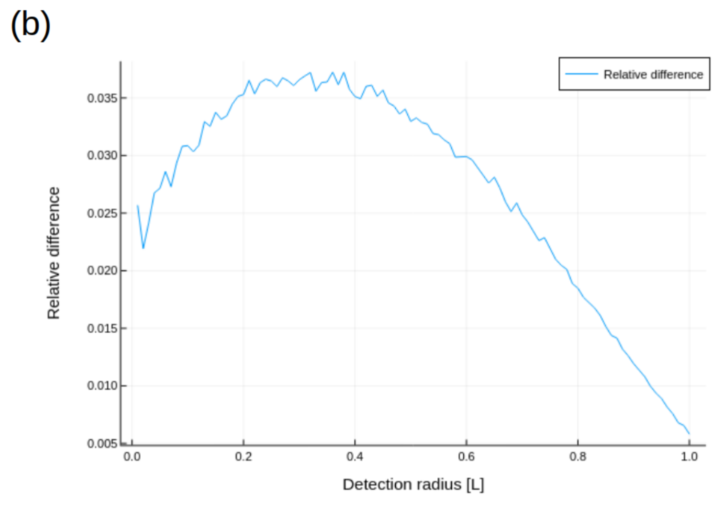


Figure 21: Relative difference between model and simulations on the two distributions show in figure 20 (a) distribution with n=1,n_t=1 and r=0.05 (b) distribution with n=10,$n_t = 3$ and r=0.4 (auction mechanism)

As seen in Figure 21 the relative difference between simulations with approximate distributions and model prediction does not match the findings earlier, so its not straightforward to use this knowledge to adjust for model error.

A second approach would be to investigate the agent movement pattern in the short term. As agents change direction on the boundary it is likely that their new path will overlap with their old in some way.

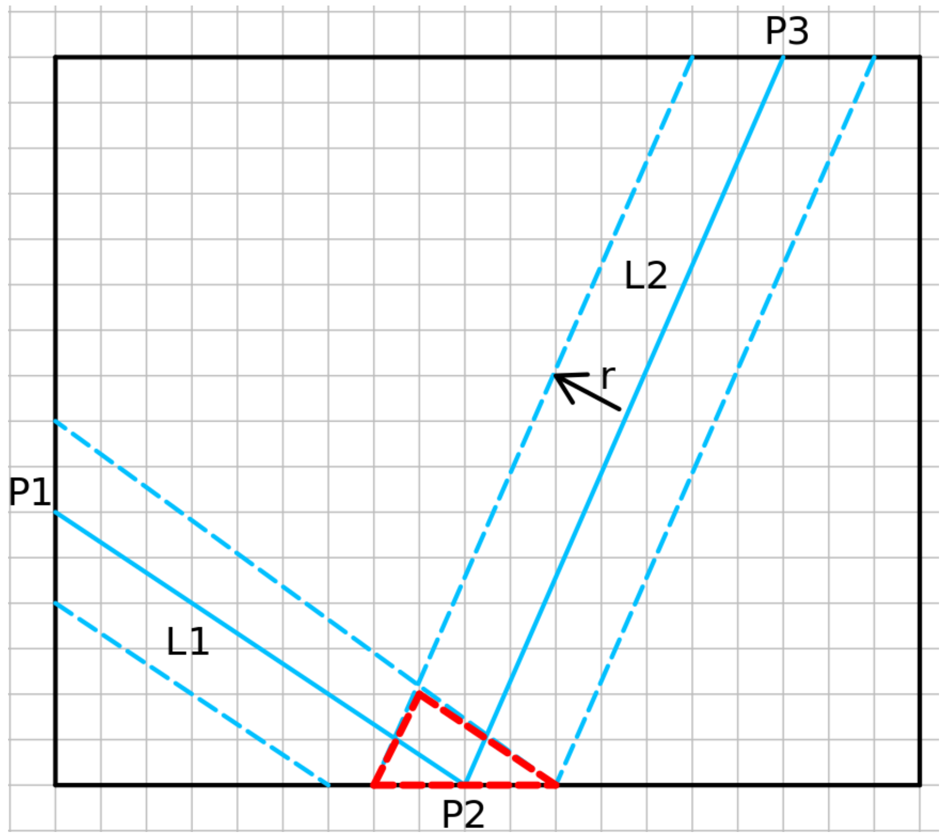


Figure 22: Agent path from P1 to P2 to P3, (blue line) detection area within the dashed blue lines. Agent detection area self overlaps in the red triangle by P2.

To calculate the overlap area in Figure 22 Polyhedra [23] package is used. The polygon areas and their intersection is found as described in Algorithm 6

Algorithm 6 Pathing overlap

```
#Lines L1(P1,P2) and L2(P2,P3) extend a distance $r_d$ normal to the line
#creating quadrilaterals that overlap.
N # number of repeats
$r_d$ # detection radius
function $R(\theta)$ # rotation matrix
$E_{spo} = 0$ #initialization of expected self pathing overlap
for i in 1:N

    # uniform random points on boundary,
    # no two consecutive points on same boundary face
    P1,P2,P3
    $m_1 = (P2_y - P1_y)/(P2_x - P1_x)$ # L1 gradient
    $x_1 = \frac{r_d}{\sqrt{m_1^2+1}}$ # x change s.t. $||[x_1, x_1m_1]|| == r_d$
    $v_1 = [x_1, x_1m_1]$
    $P1^* = P1 - v_1$
    $P2^* = P2 + v_1$
    Poly1=Polygon($P1^* + R(\pi/2)v, P1^* + R(-\pi/2)v, P2^* + R(-\pi/2)v, P2^* +$
    $R(\pi/2)v$)
    #repeat with $P2 = P3$ and $P1 = P2$ to make Poly2
    Poly3=Poly1$\cap$Poly2 #intersection of polygons Poly1 and Poly2
    #intersection area divided by total combined area
    $E_{spo}$+=(area(Poly3)/(area(Poly1)+area(Poly2)-area(Poly3)))/N

end
return $E_{spo}$
```

The E_{spo} is then used to adjust the prediction, by

$$r_d^* = \sqrt{\frac{r^2}{E_{spo}(1 - G_d(r_d))}} \tag{8}$$

then we predict with

$$H_4 = G_d(r_d^*) \tag{9}$$

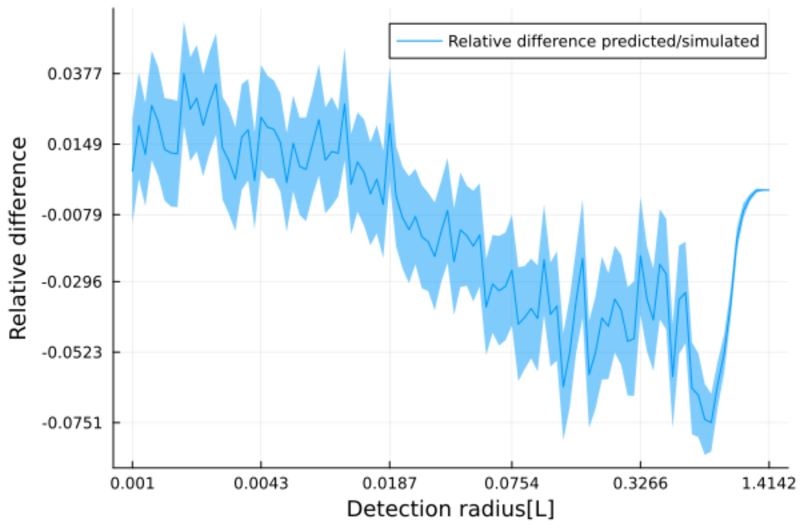


Figure 23: Relative difference when using new prediction

As seen in the figure above, the error is reduced significantly. Now the mean error is 2.5%. However, there are some peaks, especially for high detection radii. The remaining error is probably due to discretization. We assume the agents move continuously, which is not the case, one can go one step further and recreate this discrete movement, but it is difficult to do so without essentially recreating the simulator itself.

To look into this further an algorithm is made Algorithm 7 to simulate agents walking in straight paths in a more direct way than they do in the main simulator. While this algorithm does recreate much of the behavior of the main simulator with one agent and one task, it specifies more strictly how agents move from point to point, distilling the effect.

Algorithm 7 Movement algorithm

```
end_pos # target position on boundary
pos # position
$r_d$ # detection radius
dt # time step
f=$\frac{2r_d}{dt}$ # sampling frequency
v # velocity
solved=0 # tasks solved counter
N # total number of steps
c_steps=0 # step counter
function getRandomBoundaryPoint()
    if U(0,1)>0.5
        point=[$\mathcal{U}(0,1)$,U(0,1)]
    else
        point=[U(0,1),$\mathcal{U}(0,1)$]
    end
    return point
end
function newEndPosition(pos)
    end_pos=getRandomBoundryPoint()
    v=normalize([end_pos[1]-pos[1],end_pos[2]-pos[2]])
    return v,end_pos
end
pos=getRandomBoundryPoint()
v,end_pos=newEndPosition(pos)
for 1:N
    if pos==end_pos
        v,end_pos=newEndPosition(pos)
    end
    if steps$\geq$f
        if distance(pos,task_pos)$\leq$r
            task_pos=[U(0,1),U(0,1)]
            solved+=1
            steps=0
        end
    end
    pos+=v*dt
    steps+=1
end
```

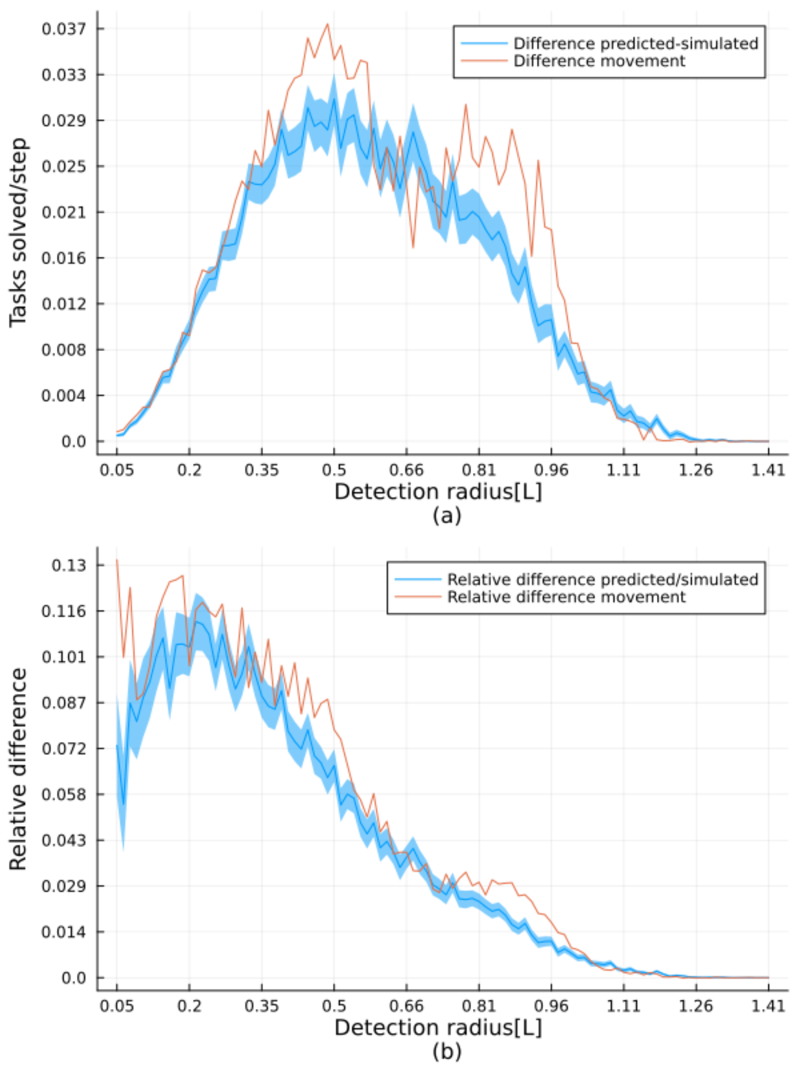


Figure 24: Difference and relative difference of predicted H_2 vs simulated and as found by the movement algorithm 7

As seen in Figure 24 the movement algorithm acts as a much better predictor of difference and the relative difference between predicted and simulated results. In the absence of other explanations, this mechanism is the most likely cause of the prediction error seen.

4.3.2 Multi task one agent

Further model performance on systems with multiple tasks is investigated. To predict the performance of this configuration, with multiple tasks, the distribution $g\left(\frac{r}{l}\right)$ is sampled $n_t$times. Only one task can be found in each iteration, so the performance prediction becomes the probability of finding at least one task. That is

$$H_5(r_d, n_t) = 1 - \left(1 - G\left(r_d\right)\right)^{n_t} \tag{10}$$

, a tasks is found or its not, so the result is binomial statistic. As seen in Figure 25 the prediction differs more for multi-task than single-task systems. But the error given by the movement algorithm has a very high correlation with the prediction error. Though the scaling is slightly off, this is the most likely cause of the error.

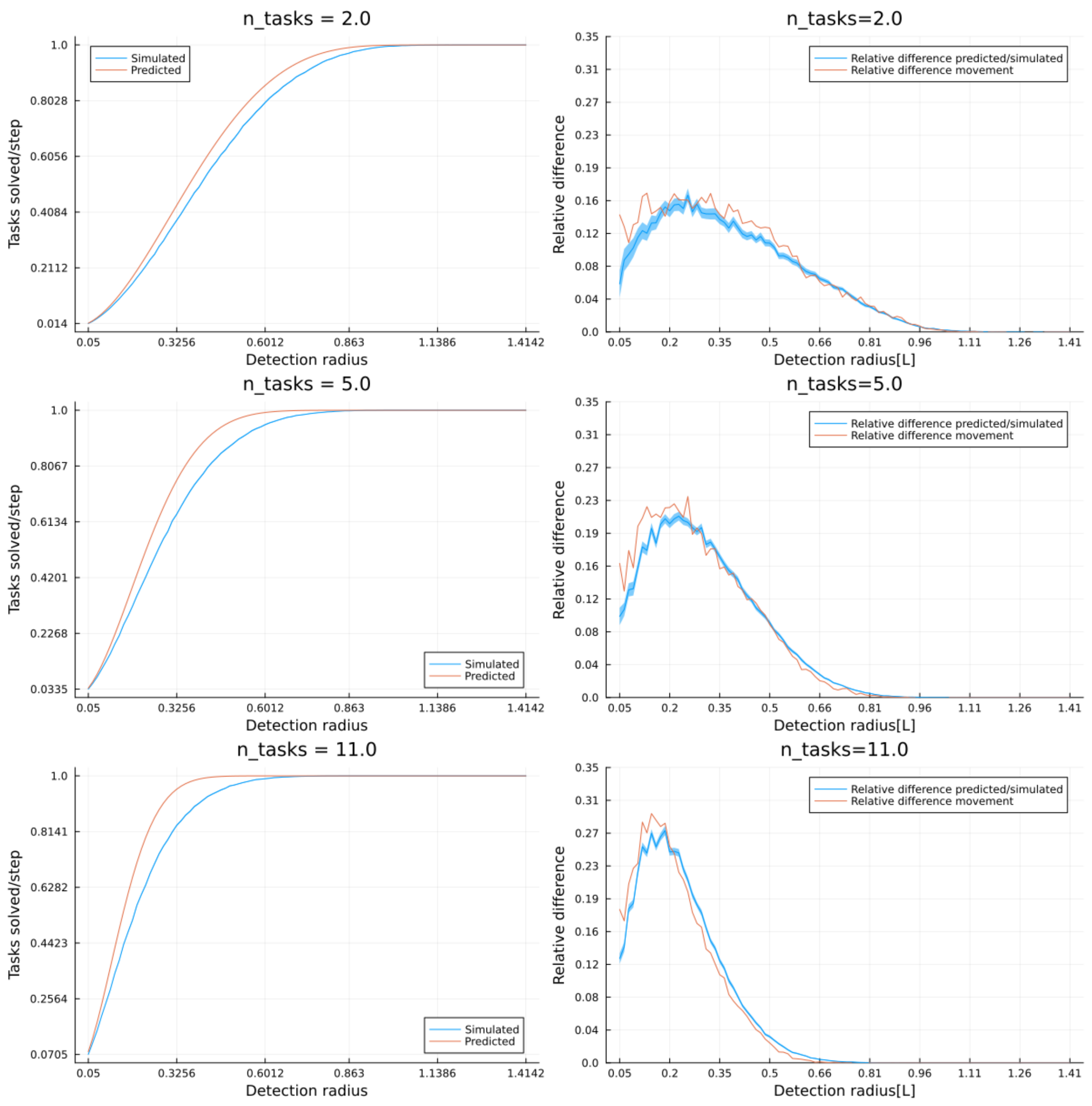


Figure 25: Simulated vs predicted for different number of tasks, seen in the title of each graph. Side by side with relative difference of prediction, simulation and movement algorithm.

When we apply pathing overlap as in equation 8 to detection radius ranging from 0.001 to $\sqrt{2}$ we get a much smaller error.

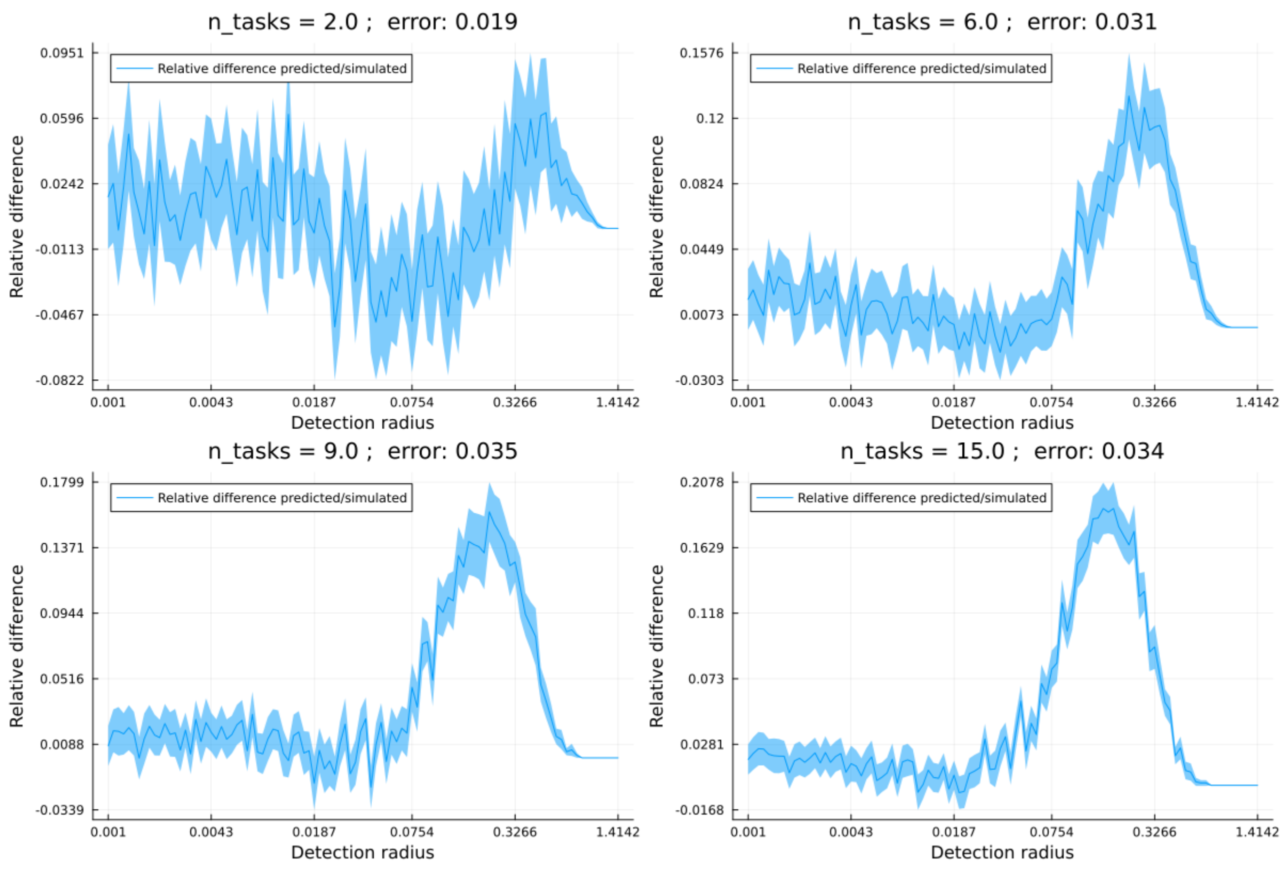


Figure 26: Relative difference with pathing overlap with average linear error

In Figure 26 the error is calculated by taking a random uniform number between 0 and $\sqrt{2}$ then taking the closest corresponding relative error value and averaging the result from many such samples as described in Algorithm 8. We see that on a linear scale, the error is relatively low, but we have high peak errors of around 18% when there are many tasks and $r_d \in (0.075, 0.33)$.

Algorithm 8 Random linear error single axis

```
xvals # x-axis values
yvals # y-axis values, relative difference values
N # number of repetitions
error=0 #initialization of error
for _ in 1:N

    #generate uniform random x between min and max of xvals
    x=minimum(xvals)+U(0,1)*maximum(xvals)
    ind=minimumIndex(abs(xvals-x)) # the index on xvals thats closest to x
    error+=abs(yvals[ind]) # add the absolute relative difference value at ind

end
return error/N
```

4.3.3 Single task multi agent

To predict single-task multi-agent performance, we use mostly the same equation as in multi-task single agents. Since now there is one task that is either found or not found by n agents, we want to have the probability that this task is found by any of the agents, which is.

$$H_6(r_d, n) = 1 - (1 - G_d(r_d))^n \tag{11}$$

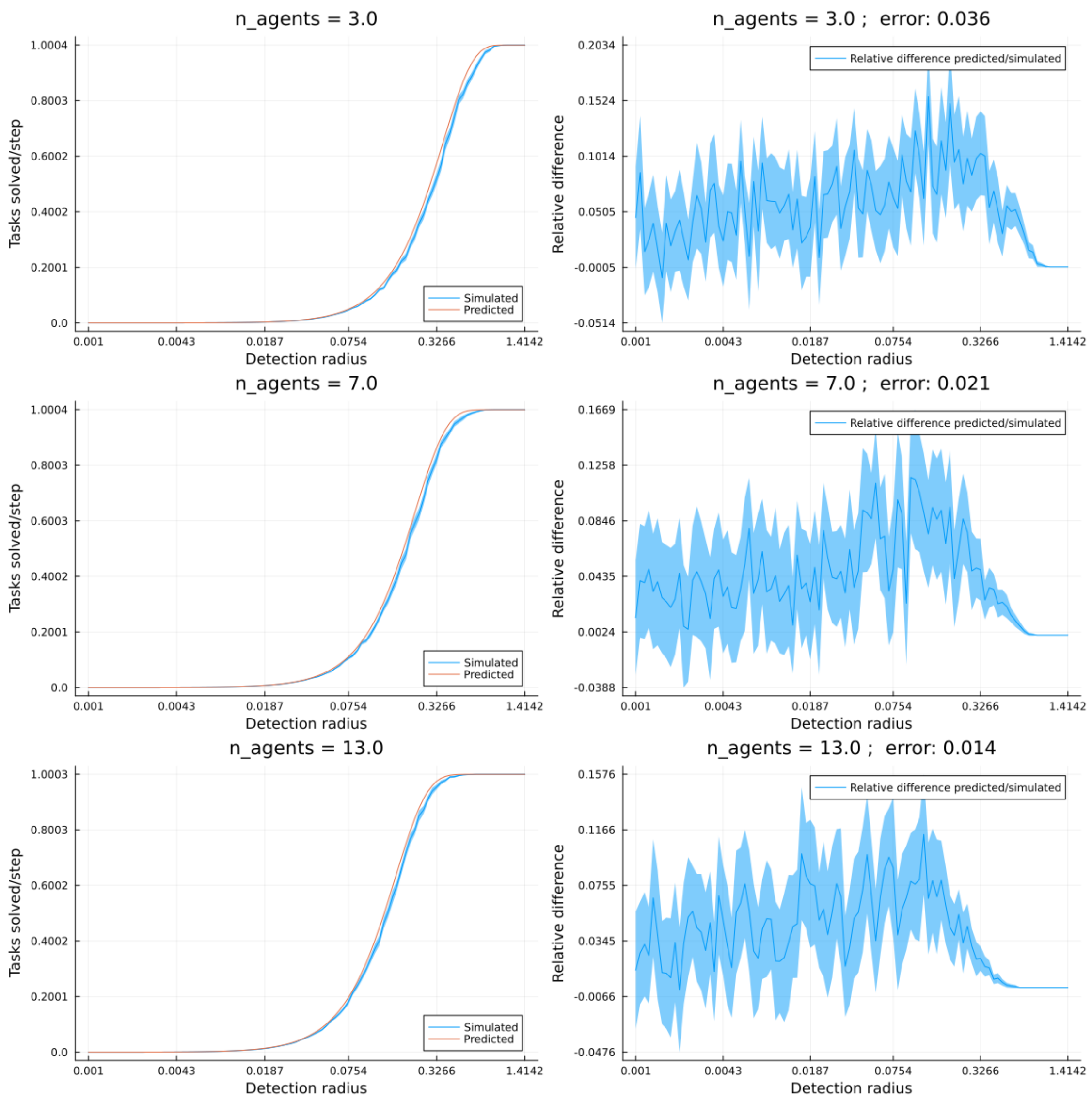


Figure 27: Simulated and predicted tasks solved per step with corresponding relative difference and linear error

The error seen in Figure 27 might be explained by agent overlap. By using self pathing overlap 8 the error is further reduced.

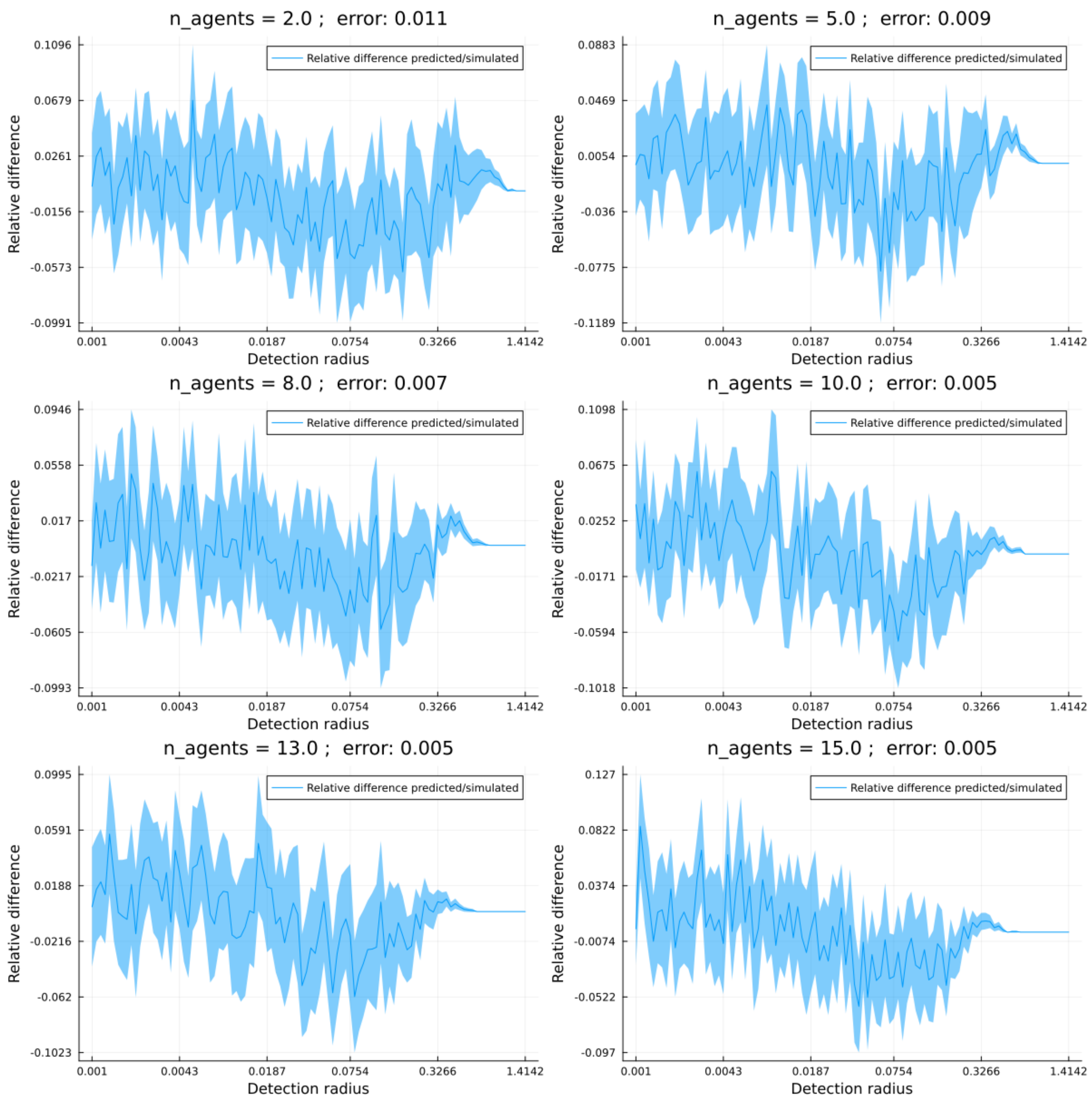


Figure 28: Relative difference predicted/simulated with self pathing overlap adjustment

in Figure 28 we see that the average error is almost nonexistent, though there are error peaks of upwards 10%.

4.3.4 Multi task multi agent

With multi-task multi-agent systems, there are more moving parts, but we can view it as a sequential search since there is no interaction between the agents. Agents activate sequentially in each iteration.

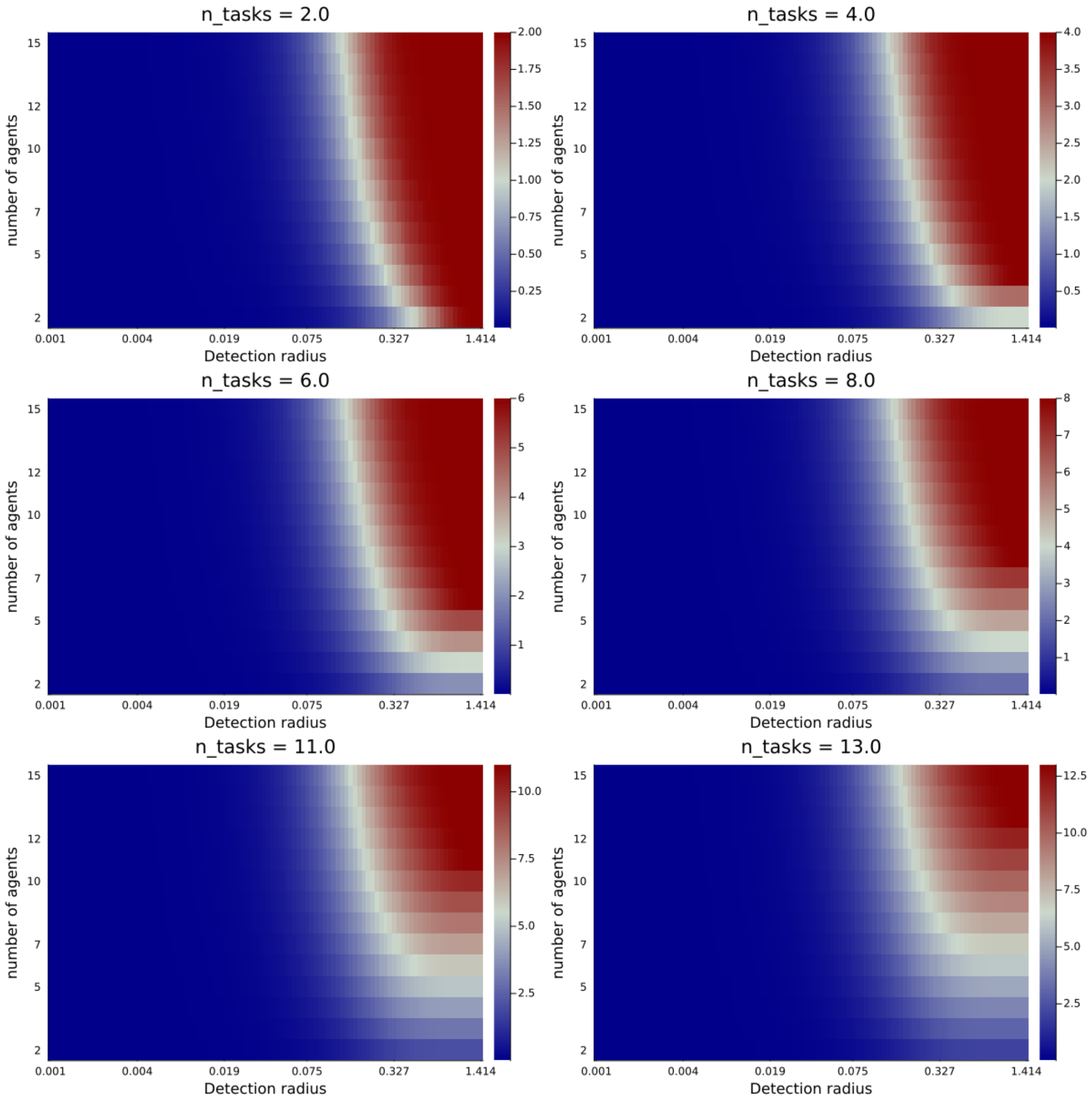


Figure 29: Heat maps of tasks solved per step for different number of tasks, with number of agents on the y-axis and detection radius on x-axis

In Figure 29 we see performance as expected. When the detection radius and the number of agents increase, the number of tasks solved increases up to a maximum of $minimum([n, n_t])$.

To predict performance in multi-agent multi-task systems, we must combine approaches from MTSA and STMA. One approach would be to multiply the MTSA equation by n, then

$$n(1-(1-G_d(r_d)))^{n_t} \tag{12}$$

. This equation work for low values of r_d where n is not very high compared to n_t, but when these conditions are not met it can overestimate the number of tasks solved. Call the maximum number of tasks solved $m_{nt} = minimum([n, n_t])$. Then we can predict with

$$H_6(r_d, n, n_t) = minimum\left(n(1-(1-G_d(r_d)))^{n_t}, m_{nt}\right) \tag{13}$$

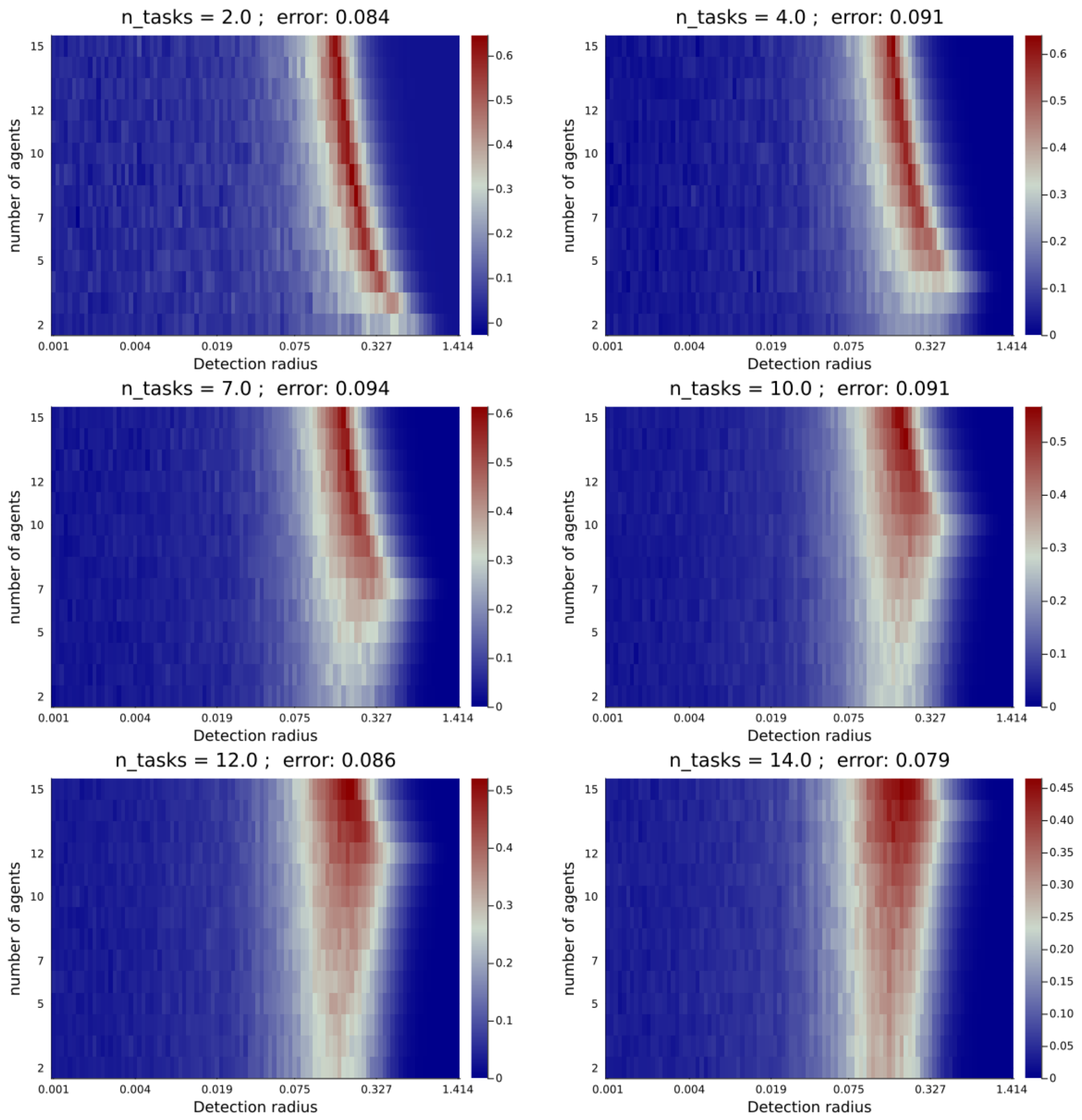


Figure 30: Relative difference between prediction by H_6 and simulated data with B_2 with linear average error.

In figure Figure 30 we see that the prediction preforms well at low detection radii and very high detection radii, but there are high peak errors of over 60% with $r_d \approx 0.2$ and $n_t = 15$. The error is calculated as shown in Algorithm 9.

Algorithm 9 Linear error calculation matrix

```
xvals # values on x-axis
yvals # values on y-axis
reldifdata # relative difference matrix
res #resolution
#Vector of evenly spaced values between min(xvals) and max(xvals)
x_vec=LinRange(min(xvals),max(xvals),res)
#Vector of evenly spaced values between min(yvals) and max(yvals)
y_vec=LinRange(min(yvals),max(yvals),res)
error=0 #initialization of error
for i in 1:res
    #Index of xvals value closest to x_vec[i]
    x_ind=min_index(abs(xvals-x_vec[i]))
    for j in 1:res
        #Index of yvals value closest to y_vec[j]
        y_ind=min_index(abs(yvals-y_vec[j]))
        # add relative difference at [x_ind,y_ind] to error
        error+=reldifdata[x_ind,y_ind]
    end
end
return error/(res^2)
```

These agents still overlap themselves by how they move so E_{spo} can be used to account for it. In addition agents overlap each other, as long as there isn't a very large amount of agents we can use $(n-1)E_{lo}$ to estimate this loss. Then r_d is updated as

$$r_d^* = \sqrt{\frac{1}{\pi}\left(\frac{\pi r_d^2}{1+E_{spo}[r_d]*(1-G_d(r_d))} - (n-1)E_{lo}[r_d]*(1-G_d(r_d))\right)} \tag{14}$$

Then predict performance as

$$H_7(r_d,n,n_t) = minimum\left(n(1-(1-G_d(r_d^*)))^{n_t}, m_{nt}\right) \tag{15}$$

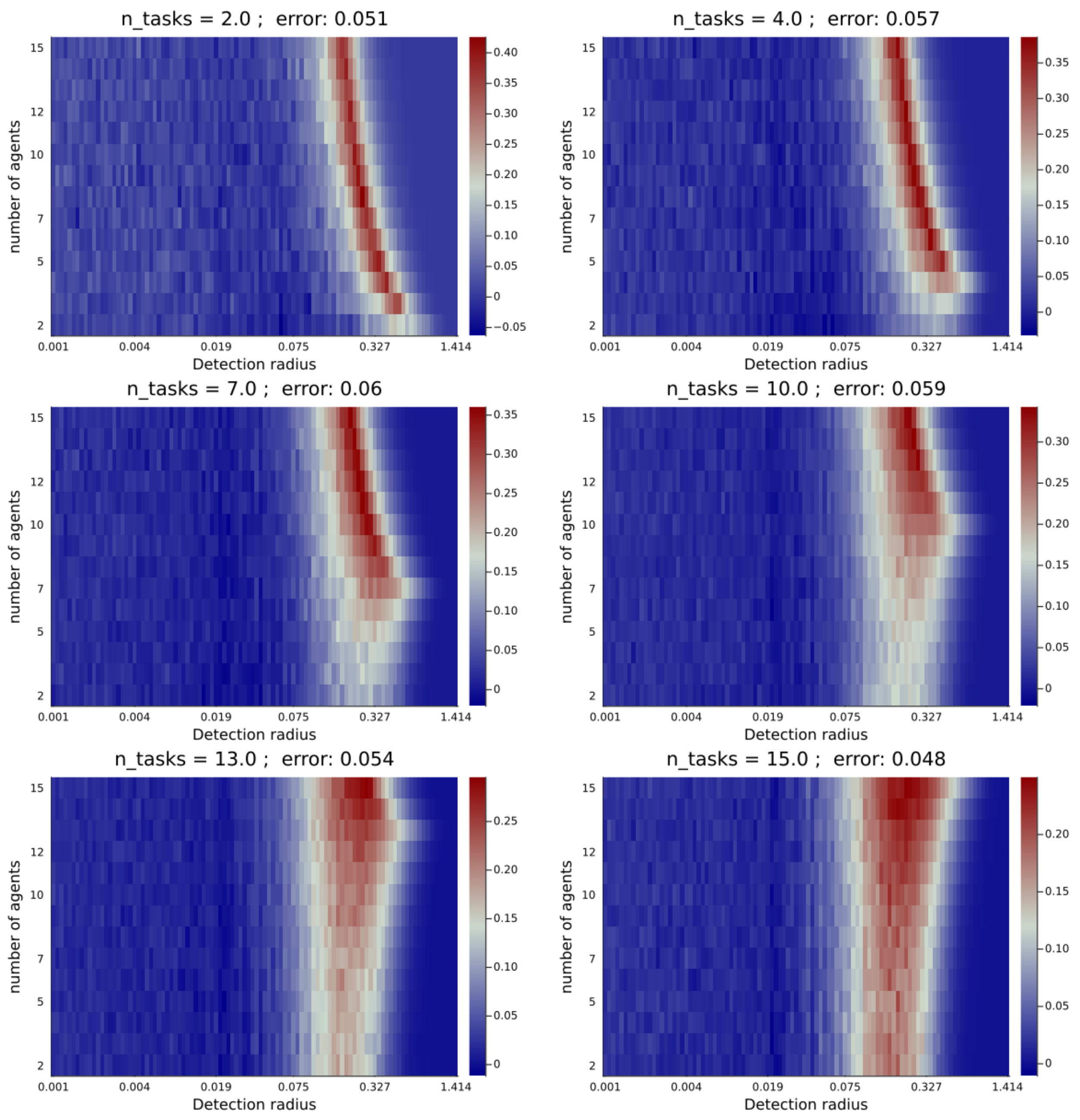


Figure 31: Relative difference between simulated and predicted (H_7, B_2)

In Figure 31 we see the average and peak error is reduced, at the cost of slightly higher under-prediction in some cases. However, the peak error is still relatively high at up to 40%.

The likely reason for this is that if a task is found before the last agent samples during an iteration, the probability of that agent finding a task is

$$1 - (1 - G_d(r_d))^{n_t - 1}$$

instead of

$$1 - (1 - G_d(r_d))^{n_t}$$

With that insight a prediction can be made as

$$k_s = min\{max\{k\}|\sum_{i=0}^{k}\frac{1}{1-(1-G_d(r_d))^{n_t-i}} \leq n, m_{nt}\}$$

Then the number of tasks solved per iteration is

$$\sum_{i=0}^{k_s}\frac{1}{1-(1-G_d(r_d))^{n_t-i}} + (n-k_s)(1-(1-G_d(r_d))^{n_t-ks}) \tag{16}$$

Then we define the prediction as

$$H_8(r_d = r_d^*, n, n_t) = \sum_{i=0}^{k_s}\frac{1}{1-(1-G_d(r_d))^{n_t-i}} + (n-k_s)(1-(1-G_d(r_d))^{n_t-ks}) \tag{17}$$

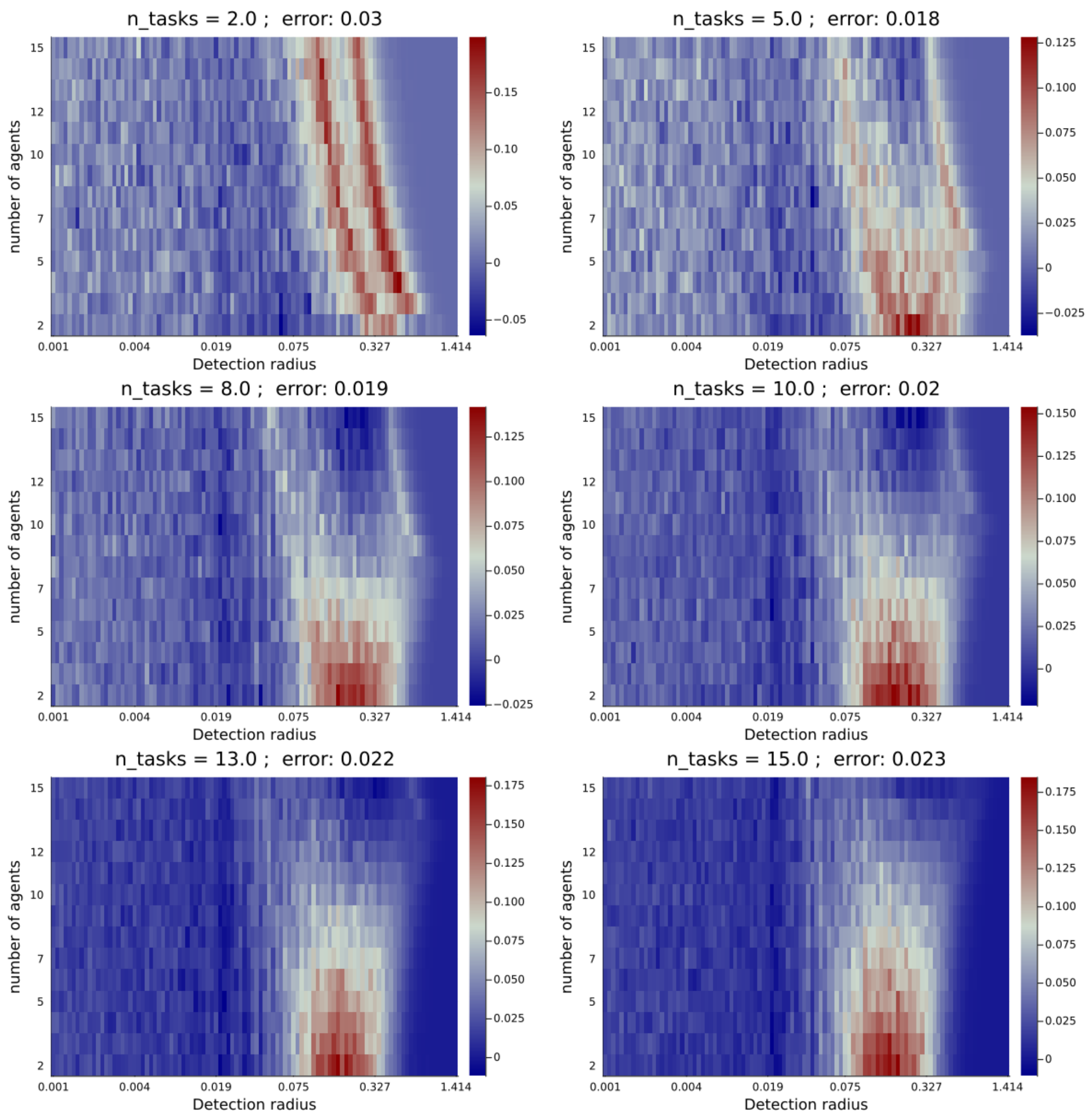


Figure 32: Relative difference between simulation and prediction by H_8

As seen in 32 errors are now much lower, with an average error between 2% and 3%. In addition, the peak error is also significantly reduced, at a maximum of about 18%.

H_8 is slightly complicated, we can estimate it by realizing that

$$\frac{k}{(1-(1-G_d(r_d))^{n_t}} \leq n \leq \frac{k}{(1-(1-G_d(r_d))^{n_t-k}}$$

Then the minimum value of k,

$$k_1 = n * (1-(1-G_d(r_d))^{n_t}$$

which is H_6 without bounds. The maximum value k_2 given by

$$\frac{k}{(1-(1-G_d(r_d))^{n_t-k}} = n$$

is a bit more complicated to calculate. Rewrite

$$(1-G_d(r_d))^{n_t-k} = P^{n_t-k}$$

then

$$\frac{k}{1-P^{n_t-k}} = n \rightarrow k = n(1-P^{n_t}P^{-k})$$

so

$$k = n - nP^{n_t}P^{-k}$$

then multiply by P^k so

$$kP^k = nP^k - nP^{n_t}$$

then multiply by P^{-n} so

$$(k-n)P^{(k-n)} = -nP^{n_t-n}$$

Then change the equation to ye^y by multiplying with $ln(P)$, so

$$ln(P)(k-n)e^{ln(P)(k-n)} = -nP^{n_t-n}ln(P)$$

The equation $ye^y = x$ can be solved with Lambert's W function at branch 0, that is $y = W_0(x)$. We then find that

$$ln(P)(k-n) = W_0(-nP^{n_t-n}ln(P)) \rightarrow k_2 = \frac{W_0(-nP^{n_t-n}ln(P)) - nln(P)}{ln(P)}$$

Then we can predict with

$$H_9(r_d = r_d^*, n_t, n) = \begin{cases} k_1 & k_1 \leq 1 \\ m_{nt} & r_d = \sqrt{2} \\ \frac{k_1+k_2}{2} & otherwise \end{cases} \tag{18}$$

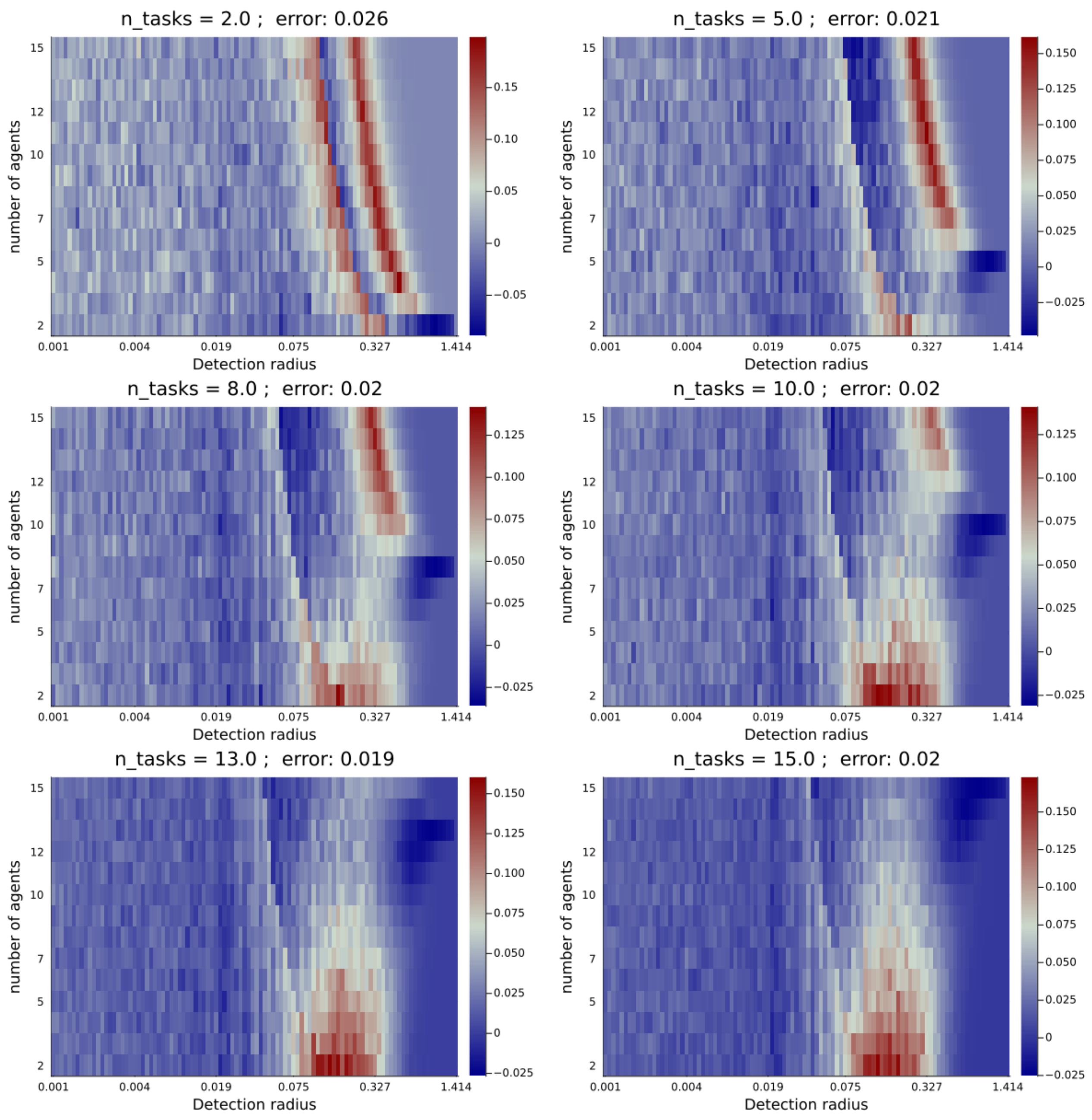


Figure 33: Relative difference between simulated and predicted result by H_9

The performance of H_9 in Figure 33 is almost identical to H_8 while being more tractable in some sense, though its worth mentioning that while Lambert's W function is well known, it cannot be expressed in terms of elementary functions.

4.4 Modeling task allocation

To separate task allocation from search $r_d = \sqrt{2}$,tthen each agent sees the entire search space. When modeling task allocation, the goal is to understand how much time is spent solving a task and what parameters affect it. Assuming agents are distributed uniformly in the search space, one wants to find the distance from the task to the furthest agent required to solve it. The distribution

$g_d(d)$ gives a good starting point, as each sample of $g_d(d)$ is a distance from one uniform point to another. When n samples are drawn from $g_d(d)$

they will have an order

$$d_{1:n} \leq d_{2:n} \leq d_{3:n} \leq d_{4:n} ... \leq d_{n:n}$$

where $d_{1:n}$ is the smallest realization. Consider that we want to find the cumulative distribution of the smallest of 2 draws,

$$F_{1:2}(d) = P(D_{1:2} \leq d)$$

=P(at least 1 of D_1, D_2 is at most d) which equals P(1 of D_1, D_2 at most d)+P(2 of D_1, D_2 at most d)

Then

$$F_{1:2}(d) = 2F(d)(1 - F(d)) + F(d)^2 = F(d)(2(1 - F(d) + F(d)) \\ = F(d)(2 - F(d)) = 2F(d) - F(d)^2 = 1 - (1 - F(d))^2$$

.

We see that this coincides with the tail binomial distribution,since

$$F_{i:n}(d) = \textstyle\sum_i^n \text{P(exactly i of } d_{1:n}\text{... is at most d)} = \sum_i^n \begin{pmatrix} n \\ i \end{pmatrix} F(d)^i (1 - F(d)^{n-i}$$

We find the PDF of $F_{i:n}(d)$ in [6] to be

$$f_{i:n}(d) = \frac{n!}{(i-1)!(n-i)!} F(d)^{i-1}(1 - F(d))^{n-i} f(d) \tag{19}$$

Then the expected distance from one uniform point to another can be calculated by substituting $f(d)$ for $g_d(d)$ and taking the expected value, that is

$$E[d]_{i:n} = \int_0^{\sqrt{2}} d \frac{n!}{(i-1)!(n-i)!} G(d)^{i-1}(1 - G(d))^{n-i} g(d) dd \tag{20}$$

$E[d]_{i:n}$ is solved numerically by Integrals.jl [33], since there is only a few discrete values they are easy to store. The results are given below,

order\samples	1	2	3	4	5	6	7	8	9	10
1	0.5214	0.3795	0.3111	0.2692	0.2403	0.2188	0.2021	0.1887	0.1775	0.1681
2	0.0	0.6633	0.5162	0.4368	0.3849	0.3476	0.3191	0.2964	0.2779	0.2623
3	0.0	0.0	0.7369	0.5955	0.5147	0.4596	0.4188	0.387	0.3614	0.3401
4	0.0	0.0	0.0	0.7841	0.6494	0.5698	0.514	0.4718	0.4384	0.4111
5	0.0	0.0	0.0	0.0	0.8177	0.6891	0.6116	0.5562	0.5136	0.4794
6	0.0	0.0	0.0	0.0	0.0	0.8435	0.7201	0.6449	0.5903	0.5478
7	0.0	0.0	0.0	0.0	0.0	0.0	0.864	0.7452	0.6722	0.6186
8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.881	0.7661	0.6951
9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.8954	0.7838
10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9077

order\samples	11	12	13	14	15	16	17	18	19	20
1	0.16	0.1529	0.1467	0.1412	0.1362	0.1317	0.1276	0.1239	0.1205	0.1173
2	0.2491	0.2376	0.2275	0.2186	0.2106	0.2034	0.1969	0.191	0.1855	0.1805
3	0.322	0.3065	0.293	0.2811	0.2704	0.2609	0.2523	0.2444	0.2373	0.2307
4	0.3881	0.3686	0.3516	0.3367	0.3236	0.3118	0.3011	0.2915	0.2827	0.2746
5	0.4512	0.4273	0.4067	0.3888	0.373	0.3589	0.3463	0.3349	0.3245	0.315
6	0.5133	0.4846	0.4601	0.439	0.4204	0.404	0.3893	0.376	0.364	0.353
7	0.5765	0.5421	0.5131	0.4884	0.4668	0.4478	0.4309	0.4158	0.4021	0.3896
8	0.6427	0.6011	0.5669	0.5379	0.513	0.4912	0.4719	0.4547	0.4392	0.4252
9	0.7148	0.6635	0.6225	0.5886	0.5597	0.5348	0.5129	0.4934	0.476	0.4603
10	0.7992	0.7319	0.6817	0.6414	0.6078	0.5791	0.5542	0.5323	0.5128	0.4953
11	0.9186	0.8126	0.747	0.6978	0.6581	0.625	0.5966	0.5718	0.5499	0.5303
12	0.0	0.9282	0.8245	0.7604	0.7122	0.6732	0.6405	0.6123	0.5877	0.5659
13	0.0	0.0	0.9369	0.8352	0.7725	0.7252	0.6869	0.6546	0.6267	0.6023
14	0.0	0.0	0.0	0.9447	0.8448	0.7834	0.737	0.6993	0.6674	0.6399
15	0.0	0.0	0.0	0.0	0.9518	0.8536	0.7933	0.7478	0.7107	0.6792
16	0.0	0.0	0.0	0.0	0.0	0.9584	0.8617	0.8025	0.7577	0.7211
17	0.0	0.0	0.0	0.0	0.0	0.0	0.9644	0.8691	0.8108	0.7668
18	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.97	0.8759	0.8186
19	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9753	0.8823
20	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.9802

order\samples	21	22	23	24	25	26	27	28	29	30
1	0.1144	0.1117	0.1091	0.1068	0.1045	0.1024	0.1005	0.0986	0.0968	0.0951
2	0.1759	0.1716	0.1676	0.1638	0.1603	0.157	0.1539	0.151	0.1482	0.1456
3	0.2246	0.219	0.2137	0.2088	0.2042	0.1999	0.1959	0.1921	0.1885	0.185
4	0.2672	0.2603	0.2539	0.248	0.2424	0.2372	0.2323	0.2277	0.2233	0.2192
5	0.3062	0.2981	0.2907	0.2837	0.2772	0.2711	0.2654	0.26	0.2549	0.2501
6	0.343	0.3337	0.3251	0.3171	0.3097	0.3028	0.2962	0.2901	0.2844	0.2789
7	0.3782	0.3677	0.358	0.349	0.3407	0.3329	0.3255	0.3187	0.3122	0.3061
8	0.4124	0.4007	0.3898	0.3798	0.3705	0.3618	0.3537	0.3461	0.339	0.3322
9	0.446	0.433	0.4209	0.4099	0.3996	0.39	0.3811	0.3727	0.3649	0.3575
10	0.4794	0.4649	0.4516	0.4394	0.4281	0.4176	0.4079	0.3987	0.3902	0.3821
11	0.5127	0.4967	0.4821	0.4687	0.4564	0.4449	0.4343	0.4243	0.415	0.4063
12	0.5463	0.5287	0.5127	0.498	0.4845	0.472	0.4604	0.4496	0.4395	0.4301
13	0.5805	0.5611	0.5434	0.5273	0.5126	0.499	0.4865	0.4748	0.4639	0.4537
14	0.6156	0.594	0.5746	0.557	0.5409	0.5262	0.5126	0.4999	0.4882	0.4772
15	0.652	0.6279	0.6065	0.5872	0.5696	0.5536	0.5388	0.5252	0.5125	0.5007
16	0.6901	0.6632	0.6394	0.6181	0.5989	0.5814	0.5654	0.5507	0.537	0.5243
17	0.7308	0.7002	0.6736	0.65	0.6289	0.6098	0.5924	0.5765	0.5617	0.5481
18	0.7753	0.7398	0.7096	0.6833	0.66	0.639	0.6201	0.6027	0.5868	0.5722
19	0.8258	0.7832	0.7482	0.7184	0.6924	0.6693	0.6485	0.6297	0.6125	0.5966
20	0.8882	0.8326	0.7905	0.7561	0.7266	0.7009	0.678	0.6574	0.6387	0.6216
21	0.9848	0.8938	0.8389	0.7974	0.7634	0.7344	0.7089	0.6862	0.6658	0.6473
22	0.0	0.9891	0.899	0.8448	0.8039	0.7703	0.7416	0.7164	0.694	0.6738
23	0.0	0.0	0.9932	0.904	0.8504	0.81	0.7769	0.7485	0.7236	0.7014
24	0.0	0.0	0.0	0.9971	0.9086	0.8556	0.8158	0.783	0.755	0.7303
25	0.0	0.0	0.0	0.0	1.0008	0.913	0.8606	0.8212	0.7889	0.7611
26	0.0	0.0	0.0	0.0	0.0	1.0043	0.9172	0.8653	0.8264	0.7944
27	0.0	0.0	0.0	0.0	0.0	0.0	1.0076	0.9212	0.8698	0.8313
28	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0108	0.925	0.8741
29	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0139	0.9287
30	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0168

We can verify the equation the same way as before, by throwing some dice (drawing uniform numbers) and measuring their distance from each other. Then average the ordered results from n draws as described in the Algorithm 10. In the algorithm, two arrays of n points are created, then the n distances between the points are calculated term-wise, and the distance is then sorted and averaged.

Algorithm 10 order statistic verification

```
n #number of draws
N #number of samples for each draw
d_vec=zeros(n) # vector of draws
for _ in 1:N
    Pv1=rand(n,2) # array of n uniform points
    Pv2=rand(n,2) # array of n uniform points
    dists=zeros(n) # initialization of vector of distances
    #fill distance vector
    for i in 1:n
        dists[i]=distance(Pv1[i,:],Pv2[i,:])
    end
    d_vec+=sort(dists) # add the sorted distance vector term wise to d_vec
end
return d_vec/N # divide termwise by N
```

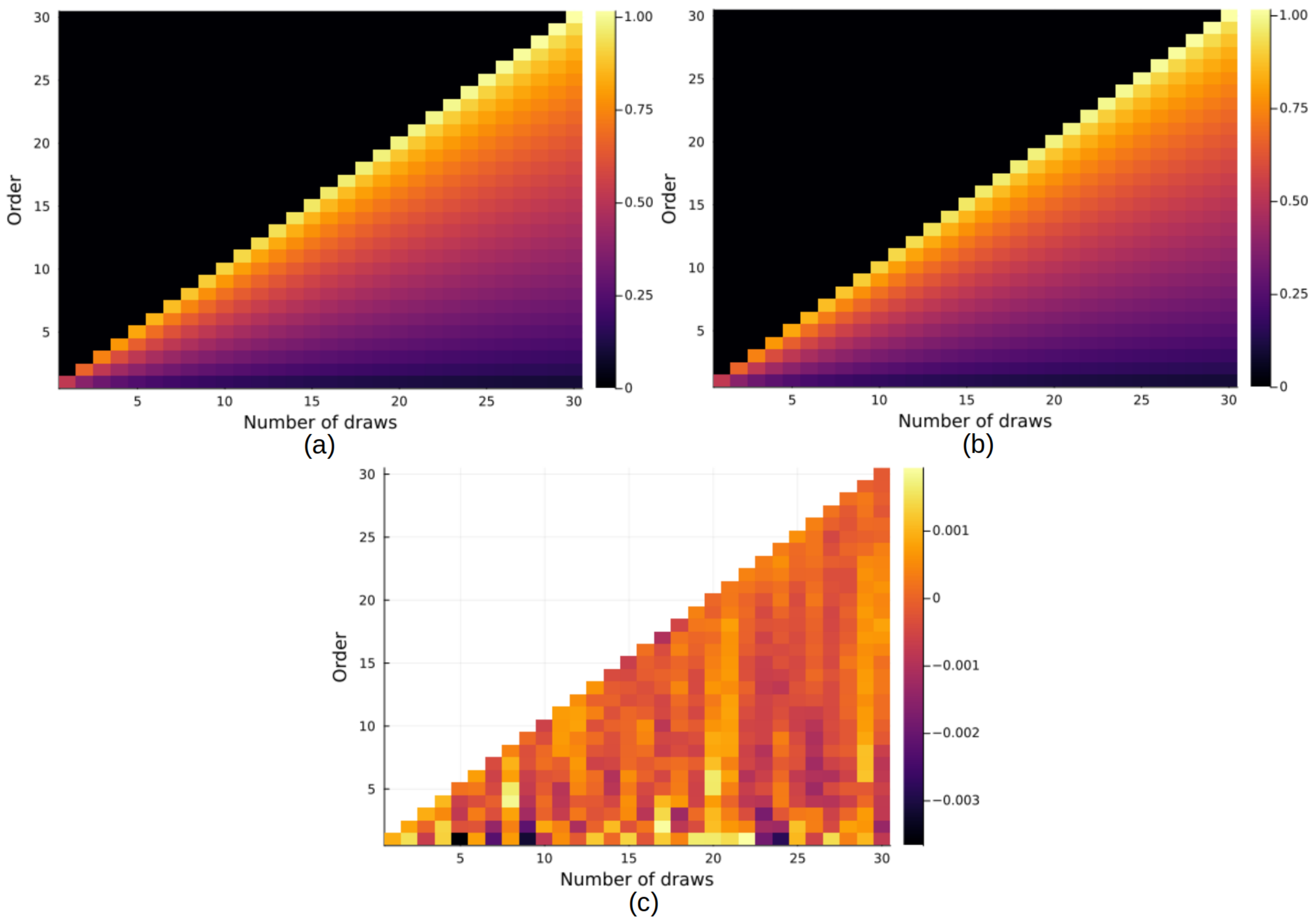


Figure 34: (a) Simulated mean distance with number of draws on the x-axis and order on the y-axis. (b) Predicted expected distance with $E[d]_{i:n}$. (c) Relative difference between (a) and (b).

As seen in figure 34 the difference between simulated values by 10 and predicted values is practically nonexistent.

4.4.1 Single task

To allow agents to spread tasks are given a respawn timer t_{rt},if tasks respawn instantly, the agents will move to the next task as a group. While we can model the artifact behavior of group formations, it is not of interest just yet. As in most practical applications of search and task allocation systems, a task is unlikely to be found right after another task is solved.

The average time between tasks solutions needs to be found to measure the performance of task allocation prediction. While simulating, the cumulative number of tasks solved is recorded per iteration, creating a vector V. Define the sequence

$$j_i = min\{j | j > j_{i-1}, V_{j_{i-1}} - V_j \neq 0\} \text{ with } j_0 = 1$$

then knowing the total number of tasks solved V_N the sum

$$\tfrac{1}{V_N} \textstyle\sum_{i=1}^{V_N} dt * j_i - t_{rt}$$

is the average time between solved tasks.

$r_d = \sqrt{2}$ so the agents see the task from anywhere within the search space, this means all the agents required to solve the task need to travel to it. With low detection radius one agent would already be at the task when it is found. Instead each agent needs to travel some distance $d - d_m$ where d_m is the maximum distance at which a task can be solved. To predict the time taken to solve a task we use

$$H = \frac{E[r_d]_{i:n} - (d_m - dt)}{dt * v} \tag{21}$$

along with

$$B = \frac{1}{V_N} \sum_{i=1}^{V_N} dt * j_i - t_{rt} \tag{22}$$

it seems that subtracting dt from d_m increased predictor performance, it is not obvious why it would be better. The difference between

$$H = \tfrac{E[r_d]_{i:n} - d_m}{v} \text{ and}$$

$$H = \tfrac{E[r_d]_{i:n} - (d_m - dt)}{v},$$

equals $\frac{dt}{v}$ so as simulation time resolution increases the difference decreases.

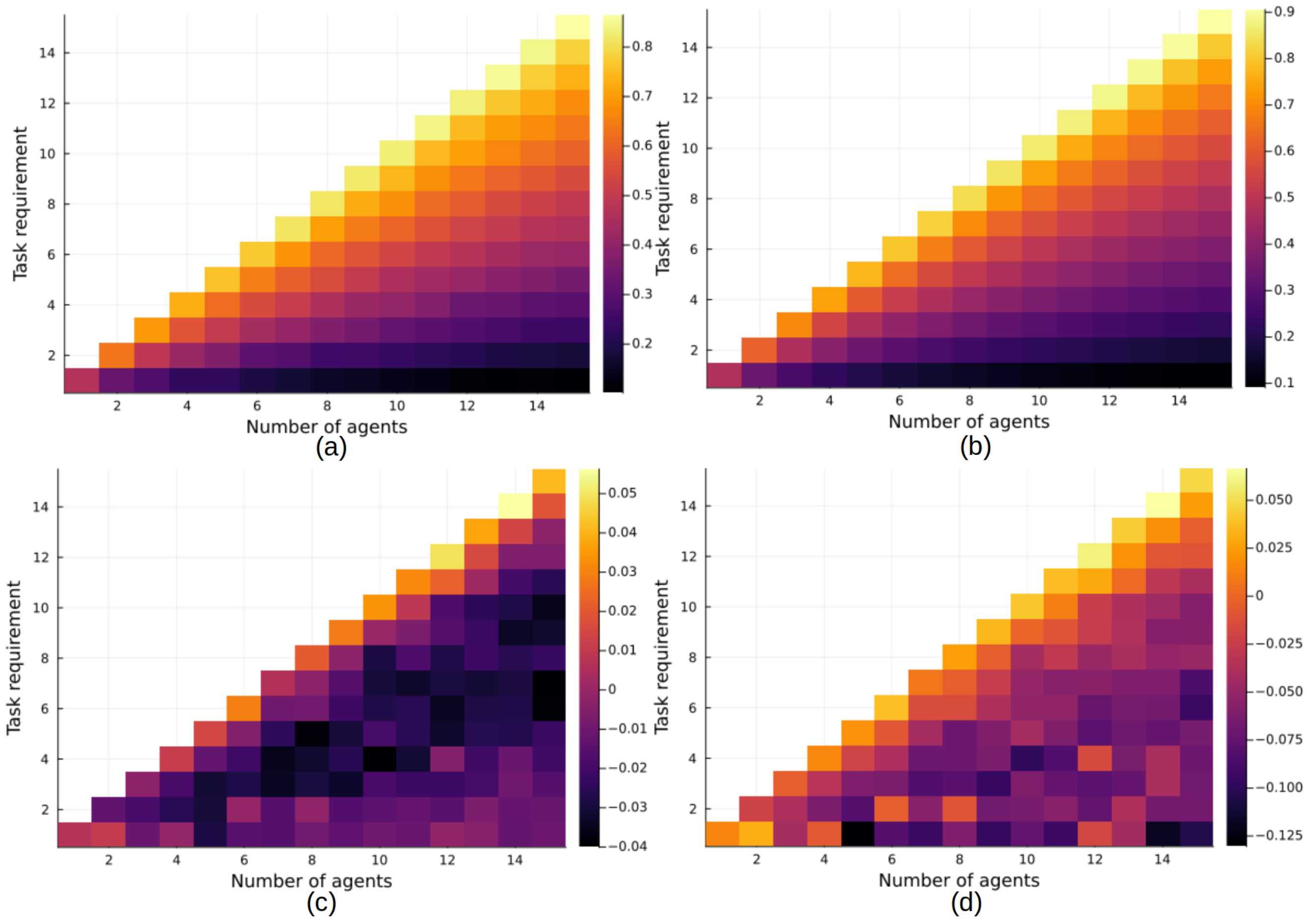


Figure 35: (a) simulated time to solve tasks with number of agents on x-axis and task requirement on y-axis. (b) Predicted time to solve task with H. (c) Difference between (a) and (b). (d) relative difference between (a) and (b)

In Figure 35the average error is ≈0.049. There are some peak lows at -12.5% for low task requirements. It might be that the distribution does not fit perfectly because when a task is found, it can be viewed as a fixed point. Then one needs to find the distribution of distance from that single point to t_r other points. Our distribution still assumes that task location is uniform, and arises from the distance between $2t_r$ points. Interestingly if one considers

$H = \frac{E[r_d]_{i:n} - c}{v}$

and optimizes for c to minimizes error one finds that when $c = 0.031$ the average error is ≈ 0.03.

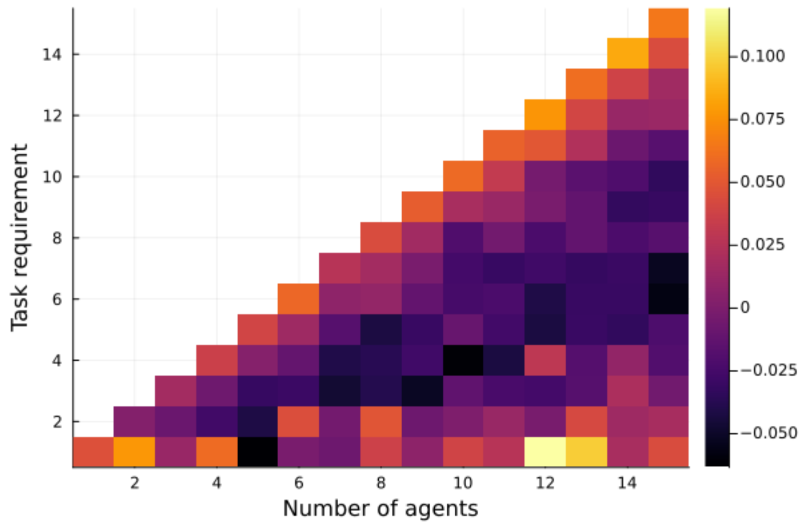


Figure 36: Relative difference between prediction and performance with optimized offset

As seen in 36 the peak errors are also reduced, except for a quite high error with 12 agents and $t_r = 1$.

Other task allocation configurations, such as multi-task, are hard to separate from the search process. Therefore, they will be modeled in combined search and task allocation.

4.5 Modeling search & task allocation

4.5.1 Single task multi agent

$t_{rt} = 20$ to let agents spread after solving a task.

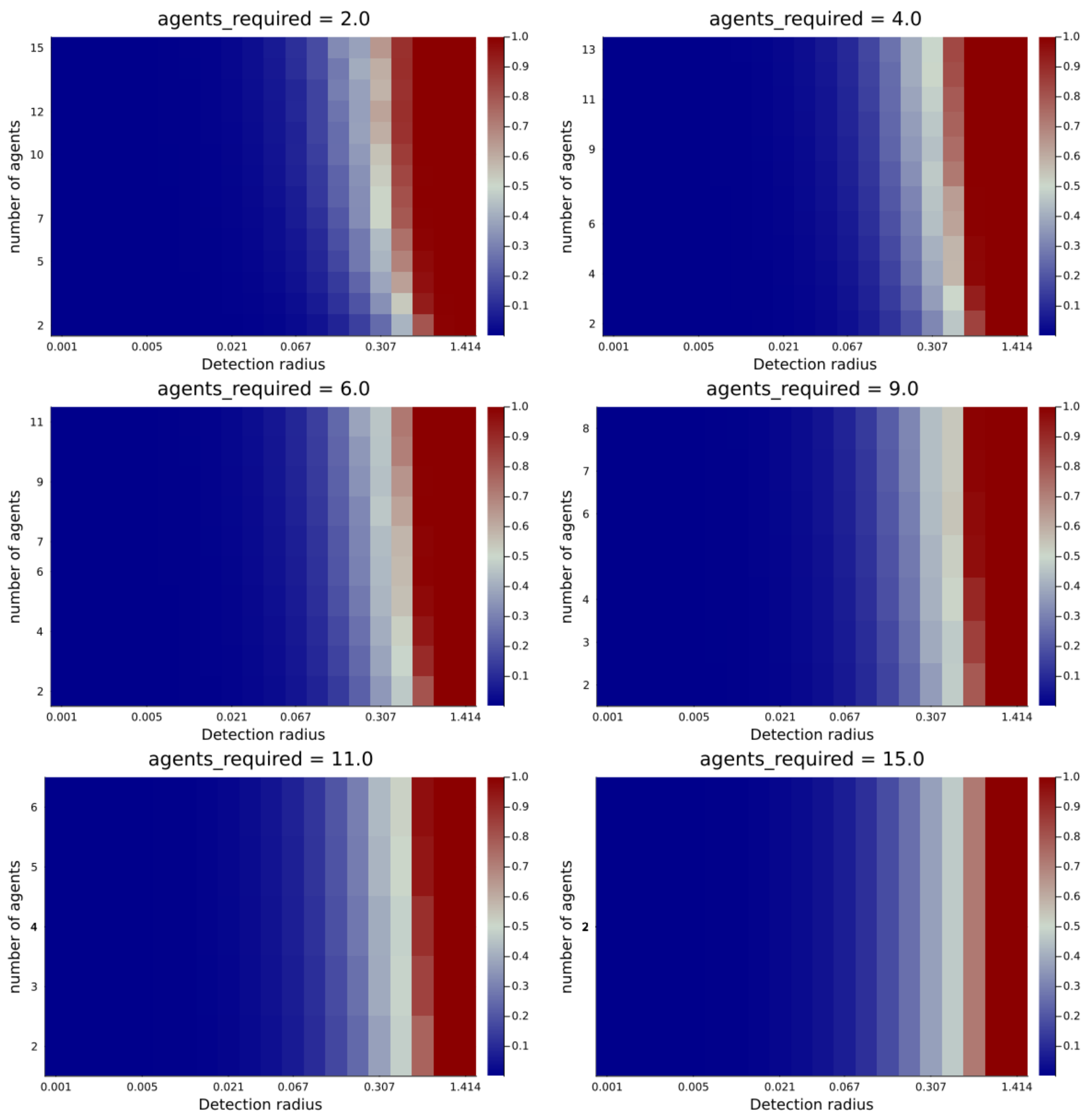


Figure 37: System performance for different number of agents required to solve the task, since task requirement=1,agents required=t_r. the Number of agents is given on the y-axis and detection radius on x-axis.

To predict this behavior we need to combine search and task allocation. We use

$$H_6(r_d, n) = 1 - (1 - G_d(r_d^*))^n$$

with r_d^* corrected for self pathing overlap as in single task multi agent search. From task allocation we have $E[d]_{i:n}$, in this case we assume that an agent is already on the task so the expected distance from the task to the agent furthest away is

$$E[d]_{tr-1:n-1}$$

As in To use $E[d]_{i:n}$ for prediction we first need to consider what

$$1-\left(1-G\left(\frac{r}{l}\right)\right)^{n_t}$$

represents and convert it in a way so that we can combine it with the expected distance.

$$H=\frac{T_s}{tdT_s+t_{rt}+t_s}$$

Where

$$\frac{1}{1-\left(1-G\left(\frac{r}{l}\right)\right)^{n_t}}$$

is the expected number of iterations before a task is found, so when it is solved instantly it would represent the time required to solve a task on average.

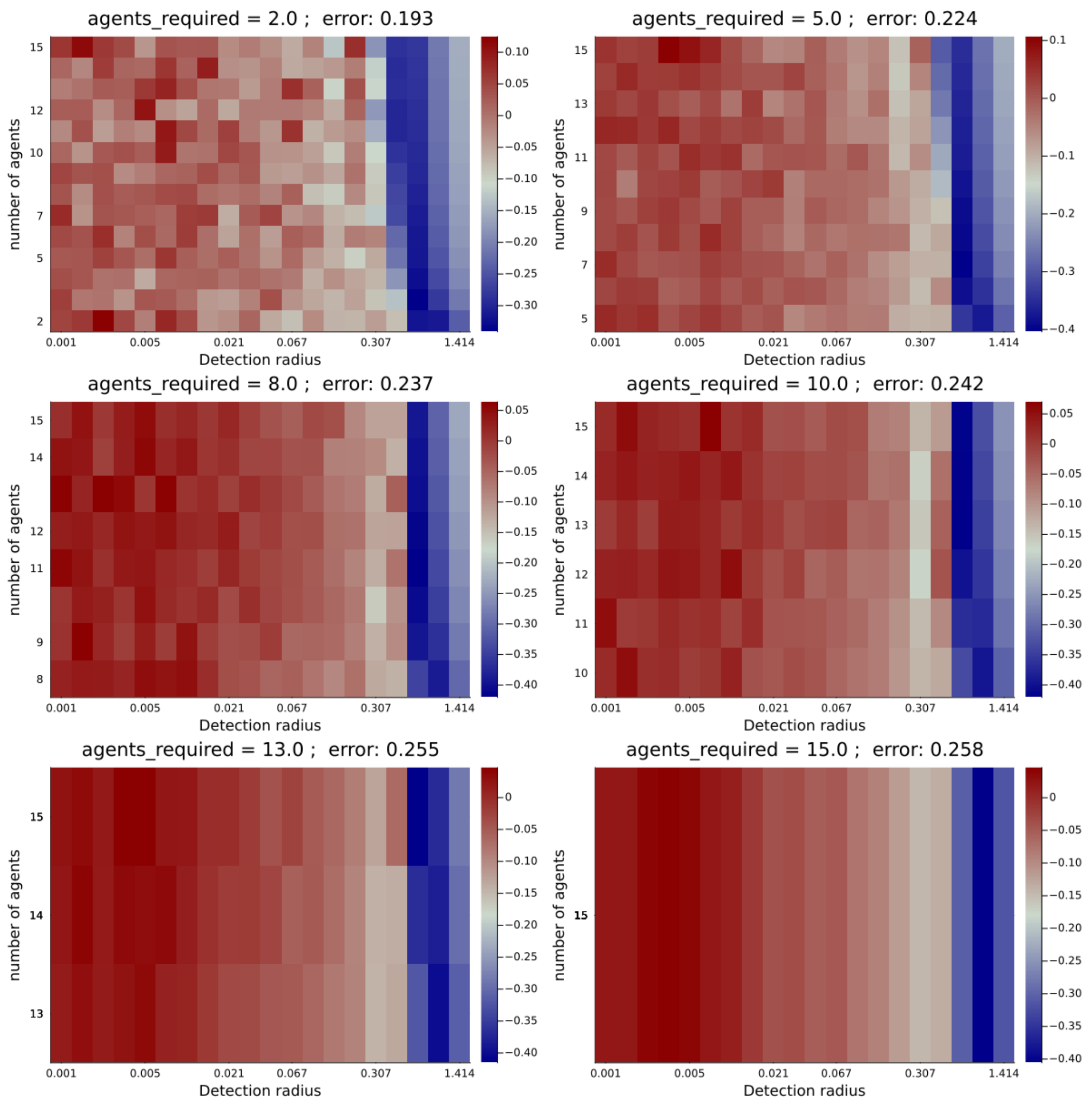


Figure 38: Relative difference between predicted and simulated value for a different number of agents required to solve the task.

We can create a better model by further studying the process that is occurring in the system.

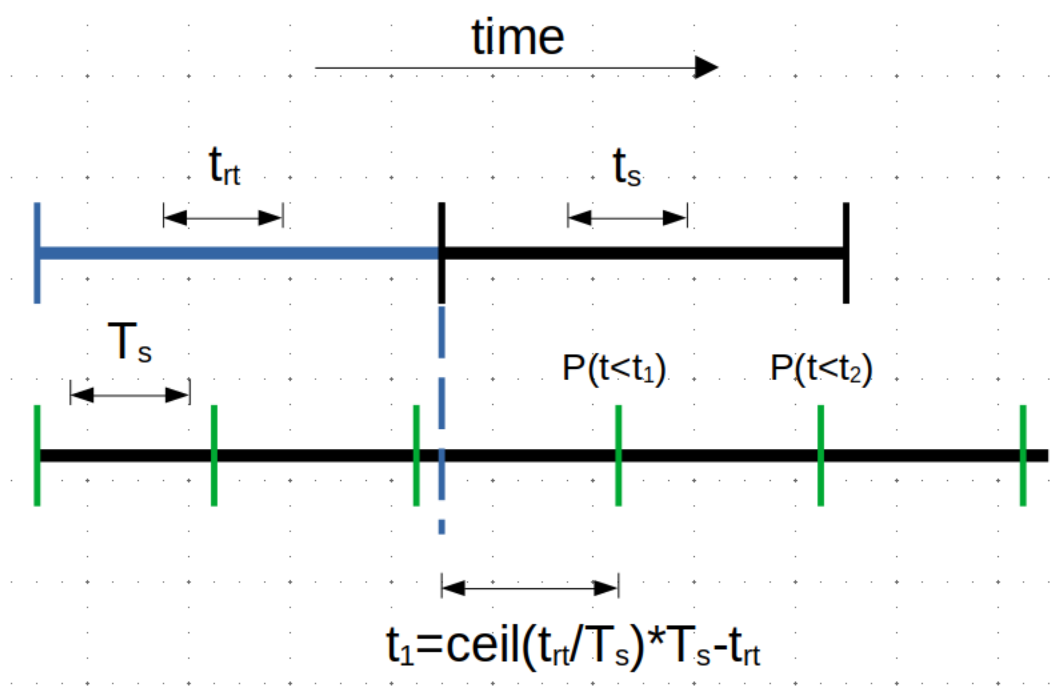


Figure 39: Illustration of system processes, a task is found at time 0. The time it takes to solve a task is at least t_{rt} - the task respawn time. T_s - the sampling period can be viewed as the fundamental time interval of our system. Since each task starts with a sampling, system performance equals the expected number of sampling periods per task.

After t_{rt} the time until the next sample is

$$t_i = ceil(\frac{t_{rt}}{T_s})T_s - t_{rt}$$

then there is a probability equal to $P(t \leq t_i)$ that the task can be solved at time $t_i + t_{rt}$ after the previous task was found.

With the equation

$$t = \frac{l * d - d_m}{v * dt}$$

where d follows the cumulative order statistic $G_{i:n}(d)$, it follows that

$$P(t \leq t_i) = P(t \leq \frac{l * d - d_m}{v * dt}) = P(d \leq \frac{t * v * dt}{l} - d_m) = G_{i:n}(\frac{t_i * v * dt}{l} - d_m)$$

Where we further define

$$t_i = (ceil(\frac{t_{rt}}{T_s}) + i)T_s - t_{rt}$$

Combined with the probability

$$1 - (1 - G_d(r_d^*))^n$$

that a task is found, there is a probability

$$G_{i:n}(\frac{t_i * v * dt}{l} - d_m)(1 - (1 - G_d(r_d^*))^n)$$

that a task takes

$$T_s(ceil(\frac{t_{rt}}{T_s}) + i)$$

time to solve. This information can be combined into a geometric distribution to find the expected time it takes to solve a task.

$$E[t] = \sum_{i=0}^{m}(t_i + trt)G_{t_r^*:n^*}(\frac{t_i * v * dt}{l} - d_m)(1 - (1 - G_d(r_d^*))^n) \prod_{j=1}^{i-1}\left(1 - G_{t_r^*:n^*}(\frac{t_j * v * dt}{l} - d_m)(1 - (1 - G_d(r_d^*))^n)\right) \quad (23)$$

Where

$$t_r^* = t_r - 1$$

and $n^* = n - 1$. $E[t]$ converges quickly in most cases but when r_d is very small it can be quite costly to compute. The maximum expected time to solve a task is

$$t_{rt} + t_s + T_s * t_d$$

Where

$$t_d = \frac{1}{(1 - G_d(r_d^*))^n}$$

the expected number of samples for detection. When

$$t_d > t_s + t_{rt}$$

most the time is spent on detection and

$$t = T_s t_d + t_s + t_{rt}$$

is a good predictor of the time spent on each task. Additionally when r_d is very large there is a probability that the expected traveling time as predicted by

$$G_{t_r-1:n-1}(d)$$

is actually smaller than the time it takes the detecting agent itself to travel to the task.

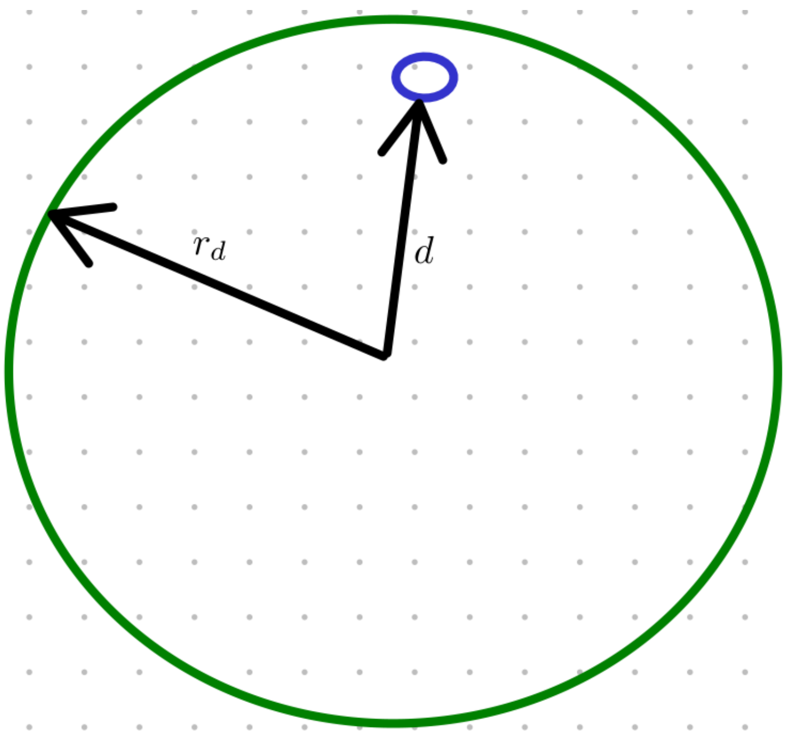


Figure 40: Agent detection boundary in green defined by r_d and a task in blue within the detection radius at a distance d from the agent.

The distance between a point and the center of a circle follow a different distribution than $g(d)$, we can easily simulated this distribution with rejection sampling.

Algorithm 11 Circle distance distribution

```
N #number of repetitions
res # resolution
d_vec=zeros(res) #intialization of distribution vector
c=0 #counter
for _ in 1:N
    $d = \sqrt{(2(U[0,1]-0.5))^2 + (2(U[0,1]-0.5))^2}$
    if $d \leq 1$
        ind=floor(d*res)+1 # index of distance d
        d_vec[ind]+=1 # increment d_vec
        c+=1 #increment counter
    end
end
d_vec=d_vec/c #normalize d_vec
D_vec=[sum(d_vec[1:i]) for i in 1:res] #cumulative distribution
```

Given the cumulative distribution $D_c(d)$ from Algorithm 11, as the detecting agent and the agents must arrive at the task before it can be solved, this cumulative probability can be multiplied into geometric distribution. To simplify the algebraic clutter, the following definition is applied:

$$P = (1 - (1 - G_d(r_d^*))^n)$$

and

$$G_t(i) = G_{t_r^*:n^*}(\frac{t_i * v * dt}{l} - d_m)$$

Then we can multiply by $D_c(d)$ with $d = \frac{t_i v}{rl}$

$$E[t] = \sum_{i=0}^{m} (t_i + trt) G_t(i) P D_c(\frac{t_i v}{rl}) \prod_{j=1}^{i-1} \left(1 - G_t(j) P D_c(\frac{t_i v}{rl})\right) \tag{24}$$

Finally we have that

$$H = \begin{cases} \frac{T_s}{t_{rt} + t_s + T_s * t_d} & ; t_d < t_{rt} + t_s \\ \frac{T_s}{E[t]} & ; otherwise \end{cases} \tag{25}$$

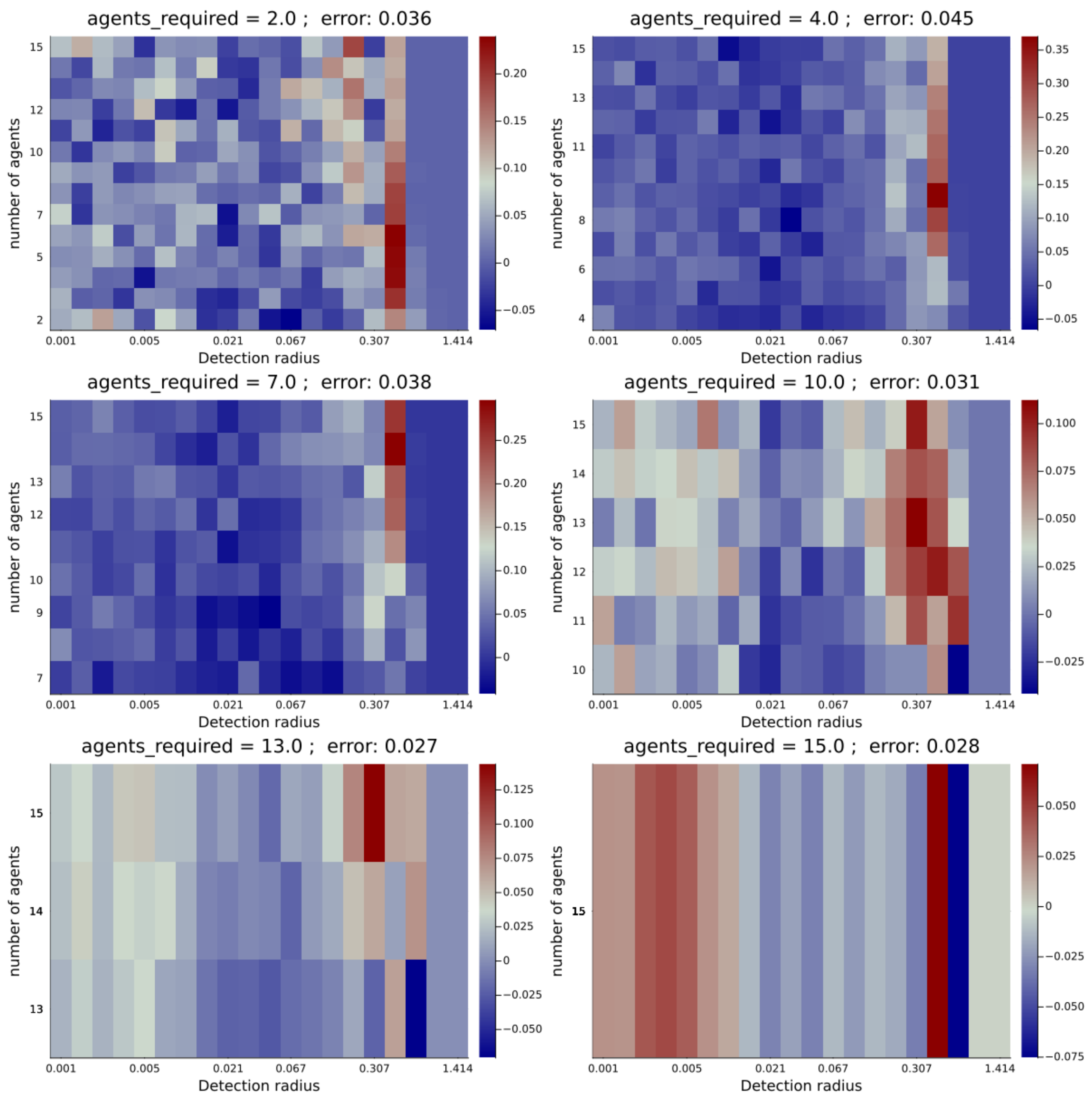


Figure 41: Relative difference between predicted and simulated performance for different number of agents required with linear error in the titles.

As seen in Figure 41 the average error is greatly reduced. There are some high error peaks, especially with high detection radii. For low and maximum radii, the predictive performance is relatively good. But as the equation is complicated, a simpler approximation is

$$H = \frac{1}{t_d + ceil(\frac{t_{rt}+t_s}{T_s}) - 1} \tag{26}$$

.

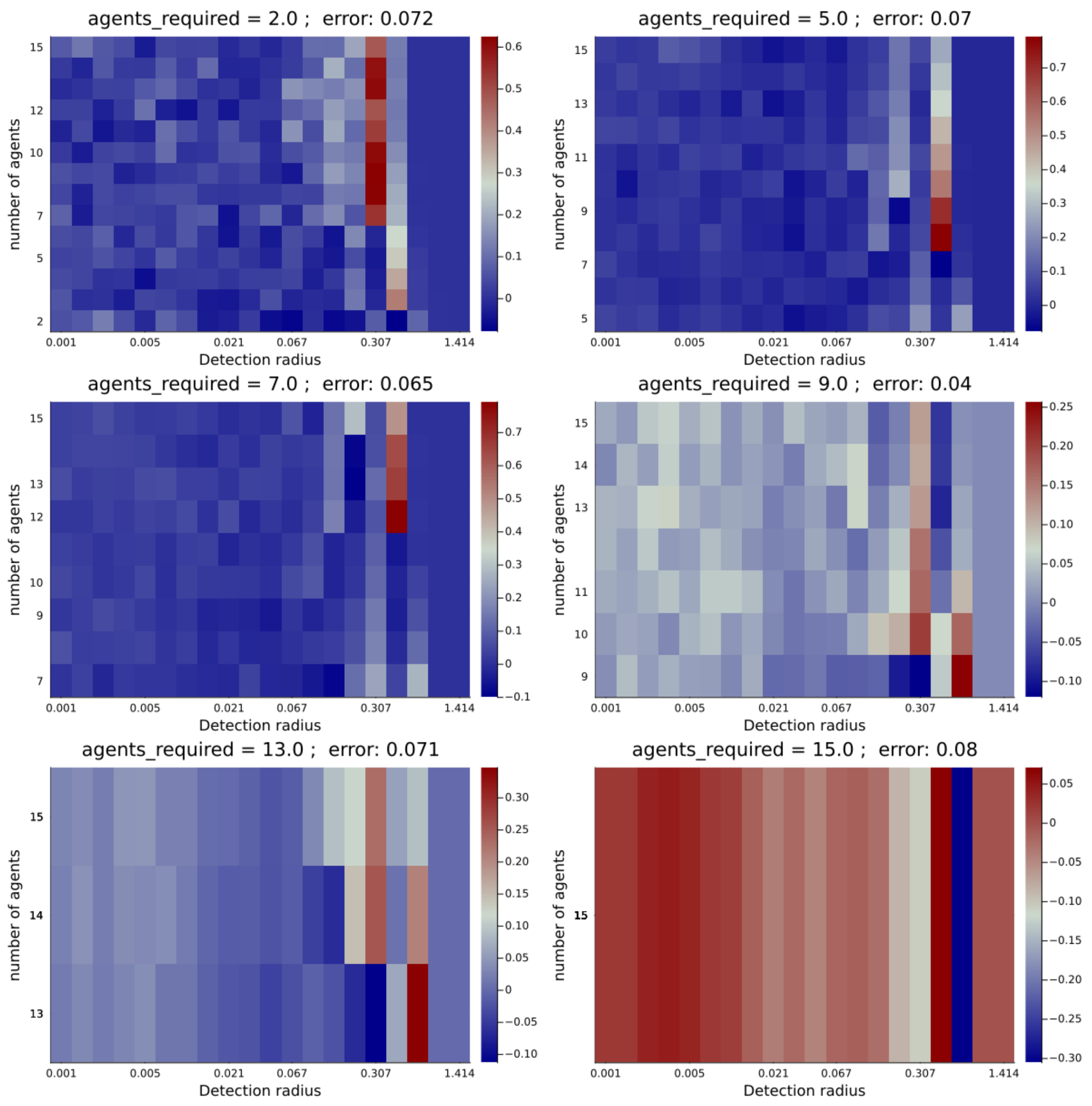


Figure 42:

4.5.2 Multi task Multi agent

$t_{rt} = 0$

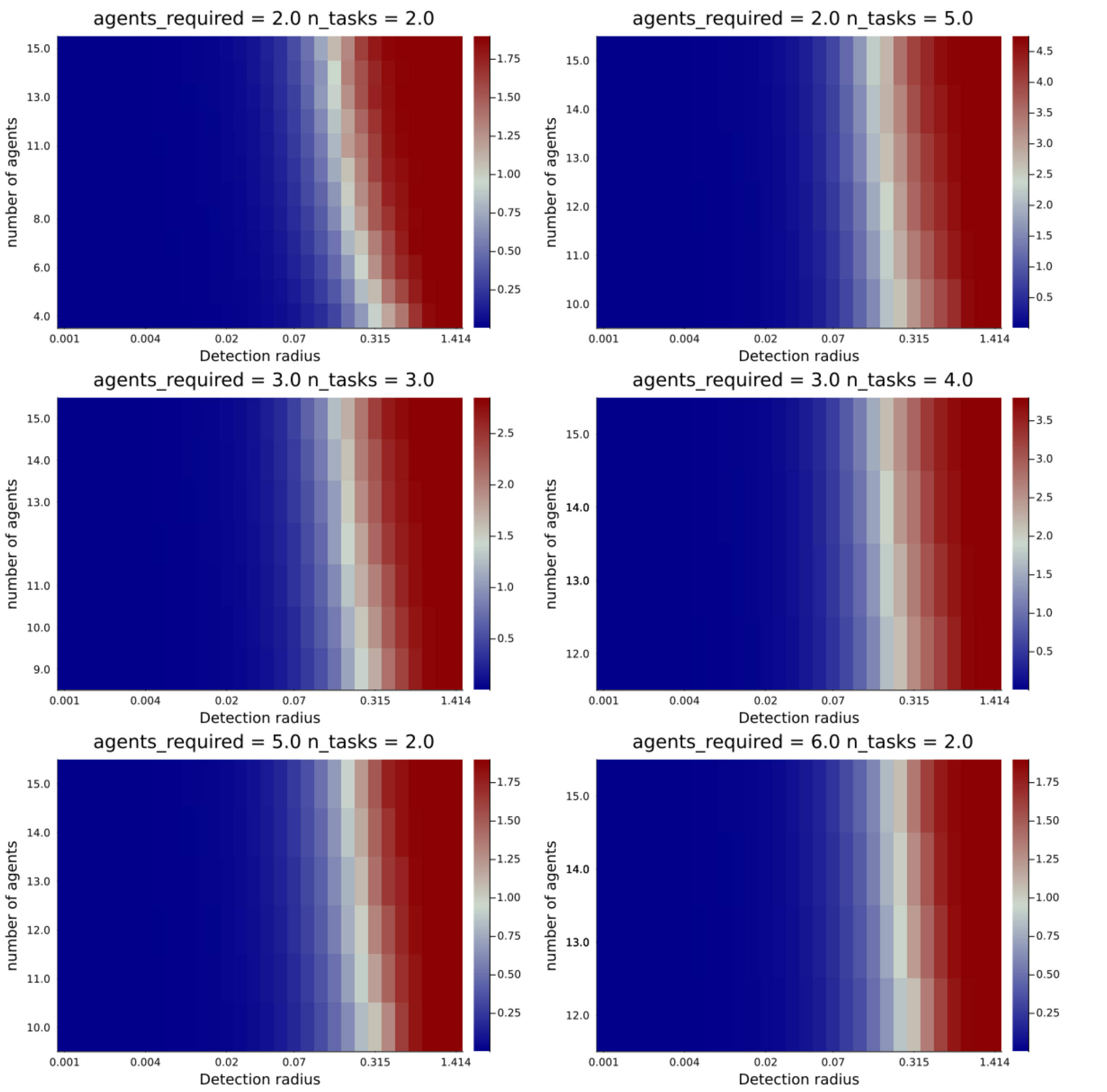


Figure 43: Simulated performance of multi agent multi task

The simple model $H = \frac{1}{t_d + ceil(\frac{t_{rt}+t_s}{T_s})-1}$ can be augmented with $m_t = min(n_t, \frac{t_c n}{t_r})$ to make a quite good predictor for its simplicity $H = \frac{m_t}{t_d + ceil(\frac{t_s}{T_s})-1}$.

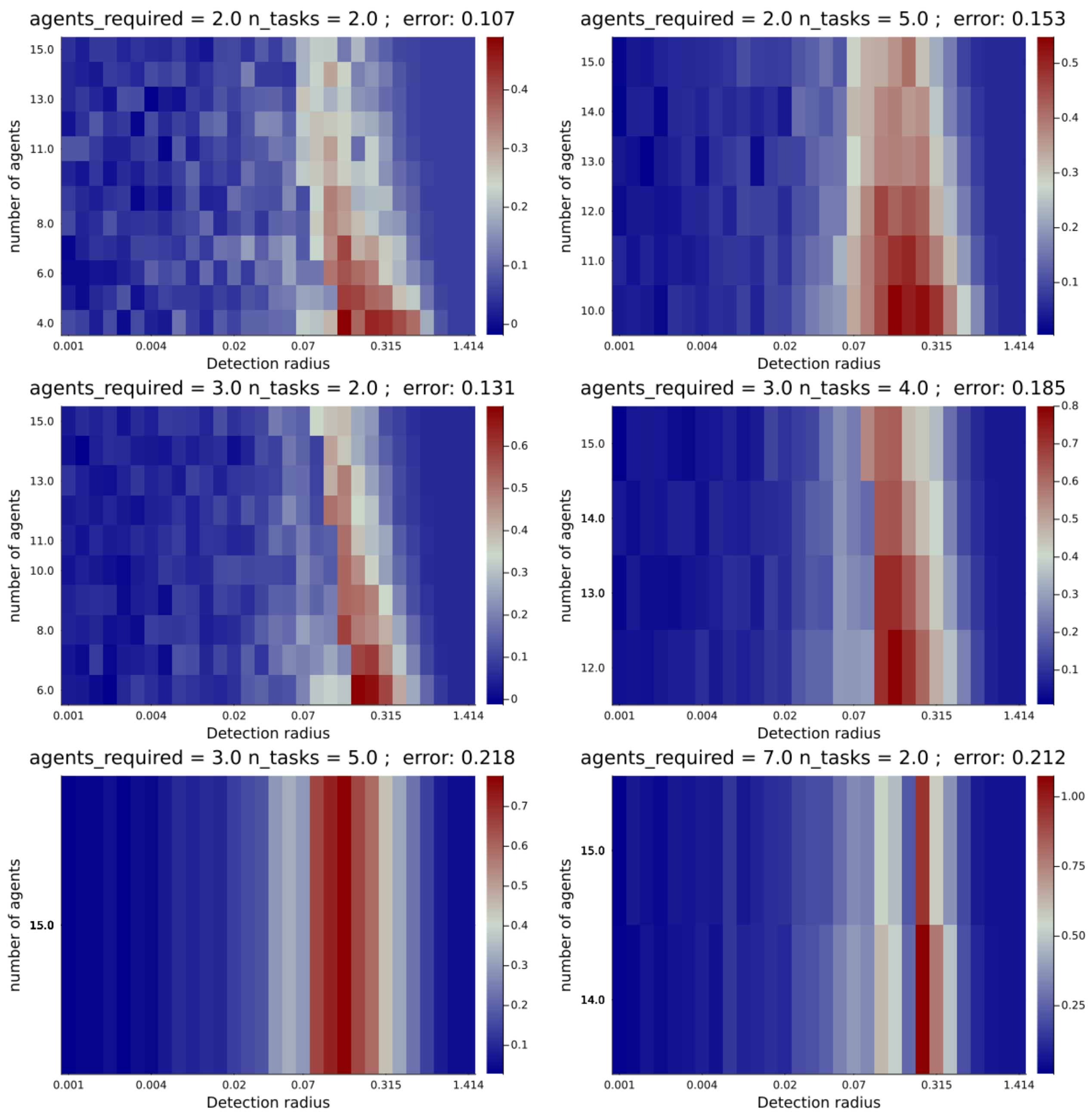


Figure 44:

4.5.3 Generalization by proportional DTMC models

In this section, a DTMC model is built up gradually and compared to the simulator. DTMC models allow for the combining observed behaviors of the MASTA system. One can simplify these models by not attempting to build a direct model but one that is proportional. Given the model output as a matrix $M \in R^{nxm}$ and simulator output as a matrix $D \in R^{nxm}$. Define the index ind then

$$\forall_{i,j}, \frac{M[j,i]\epsilon_{j,i}}{M[ind,i]\epsilon_{ind,i}} = \frac{D[j,i]}{D[ind,i]}$$

Where ϵ is the proportional error s.t.

$$\forall_{i,j}, M[j,i]\epsilon_{j,i} = D[j,i]$$

Now if

$$\forall_{i,j}, \frac{M[j,i]}{M[ind,i]} \approx \frac{D[j,i]}{D[ind,i]}$$

them $\forall_{i,j}, \frac{\epsilon_{j,i}}{\epsilon_{ind,i}} \approx 1$ which implies the proportional error along j is constant for all i. Then

$$\forall_{i,j} M[i,j]\epsilon_i = D[i,j]$$

So M is a proportional model for the system along the variable j. This can be repeated with $M[j, ind]$ to show that M is a proportional model along the variable i. If the proportional error is constant for all j and i, then M is a proportional model to D.

The simplest search and task allocation model considers the probability of finding a task and some time to solve it. Assuming tasks do not spawn within the detection radii of agents, they can only be found by the agent's newly discovered area each iteration. Discovered area

$$A_d = \pi r_d^2 - r_d^2 acos\left(\frac{v * dt}{2r_d}\right) + \frac{dt * v}{2}\sqrt{4r_d^2 - (dt * v)^2}$$

Then we find the radius of a circle with area A_d so

$$r_d^* = \sqrt{\frac{A_d}{\pi}}$$

Now $G(r_d^*)$ can be used to calculate the probability of a task being in the discovered area. Further define the detection probability as

$$P_d = 1 - (1 - G(r_d^*))^{n*n_t}$$

since DTMC's cant take values greater than 1. If a task is found at the edge of the detection radius then the traveling time t_t to that task is

$$t_t = \frac{r_d - d_{ms}}{dt * v}$$

Additionally the wait time to get help solving a task is

$$t_s = \frac{E[d]_{t_r - 1:n-1} - d_{ms}}{v * dt}$$

Then define

$$t = max(t_s, t_t, 1)$$

We can now construct the first DTMC

$$M_1 = \begin{bmatrix} 1-P & P & 0 \\ 0 & 1-\frac{1}{t} & \frac{1}{t} \\ 1 & 0 & 0 \end{bmatrix}$$

So the first state is search, the second is solving task, and the third is simplifying evaluation. The proportional time spent in state 3 equals the proportional amount of time a task is being solved. So the steady state proportions π can be calculated and π_3 is then the predicted system performance. The proportional error of M_1 is calculated as

$$\frac{M[i,j]}{M[ind,j]}\frac{D[ind,j]}{D[i,j]}$$

and can be seen in figure (here)

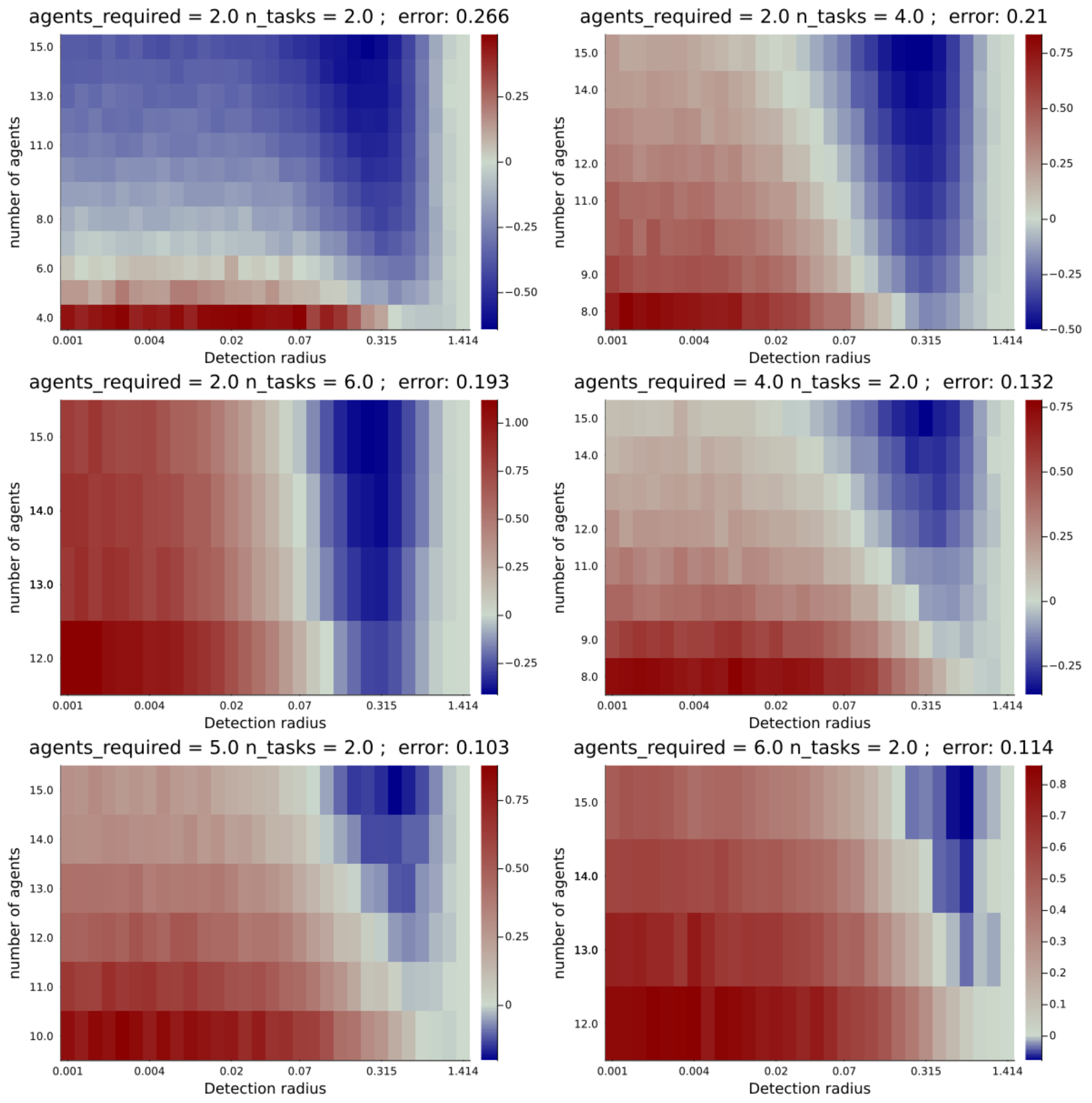


Figure 45: DTMC model performance normalized by r_d max

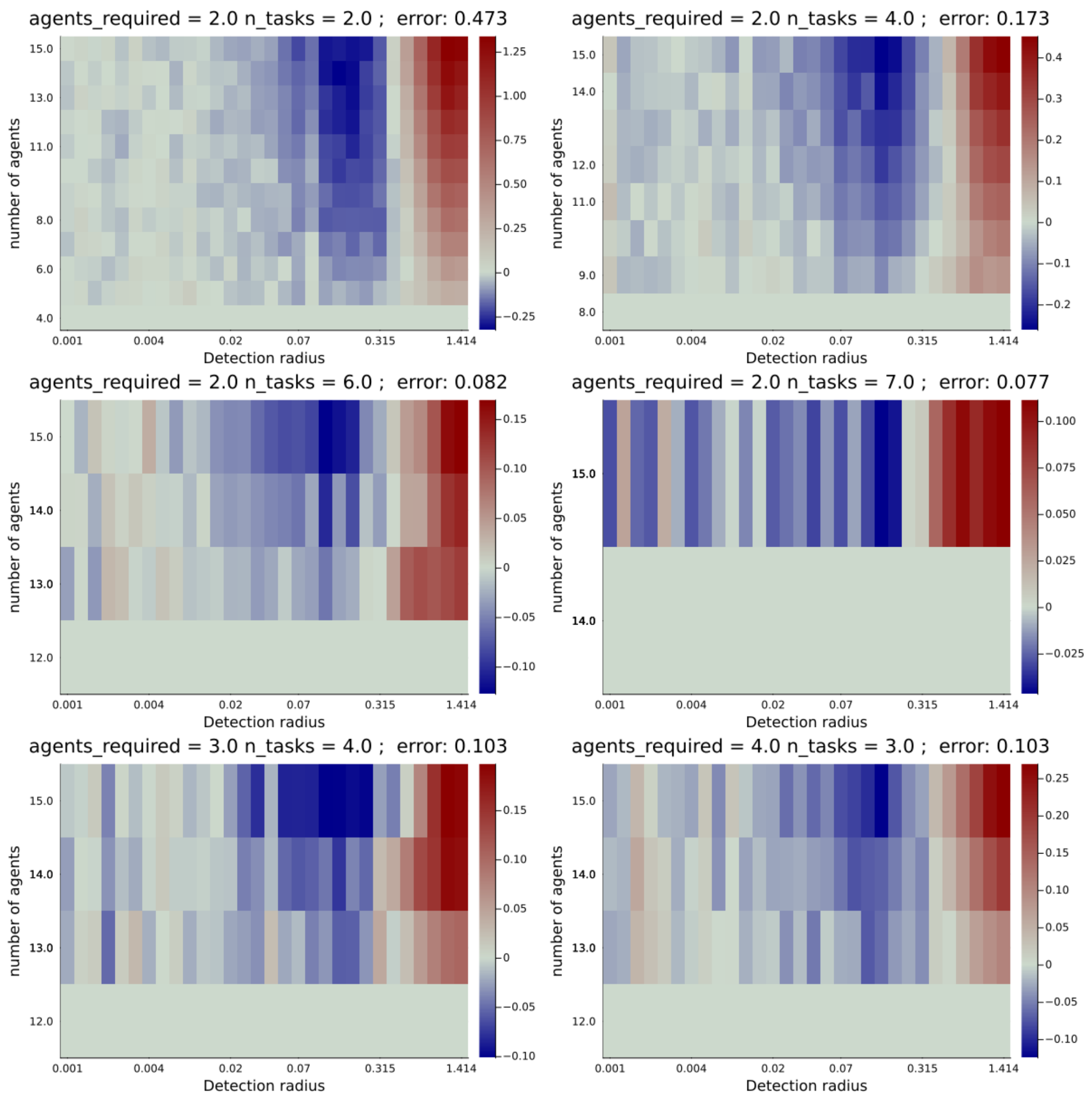


Figure 46: DTMC model performance normalized by minimum number of agents

Since tasks spawn uniformly in the search space, they can presumably spawn within an agent's detection radius. So after a task is solved, the probability that it will spawn within some agents detection radius is

$$(1 - (1 - G_d(r_d))^n)$$

assuming the agents are uniformly spread. But in this data set, the tasks spawn instantaneously; therefore, they can spawn within the detection radius of agents that just solved tasks.

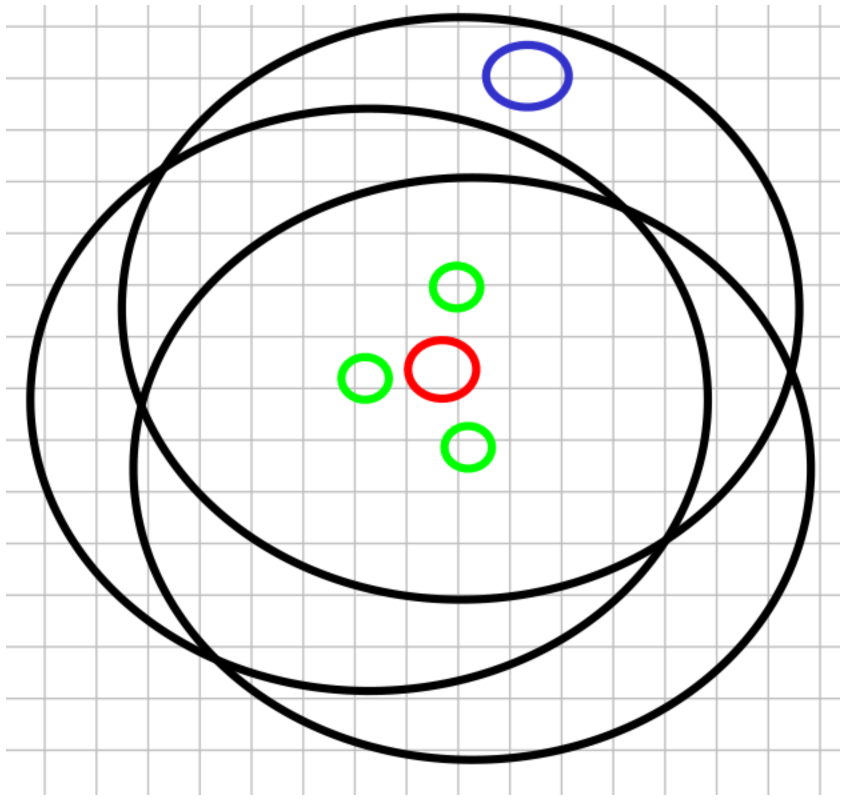


Figure 47: agents in green with associated detection radius in black. Solved task in red and newly spawned task in blue.

So there is three possibilities for the newly spawned task, either it spawns on a group of agents that just solved a task, on a singular agent that is search for a task or not within any agents detection radius. But to abuse this knowledge we need some estimate for the number of groups and free agents. Using

$$n_{ct} = \frac{W_0(-nP^{n_t-n}ln(P)) - nln(P)}{ln(P)}$$

as an estimate for the number of tasks continuously being worked on we find that

$$n_f = n - n_{ct} * t_r$$

is the number of free agents.

$$n_{tf} = n_t - n_{ct}$$

Is the number of free tasks, the number of points including groups and free agents is

$$n_p = n + n_{ct}(1 - t_r)$$

The probability that an agent is detected instantly after it is spawned can now be estimated by

$$P_i = (1 - (1 - G_d(r_d))^{(n_{tf}+1)(n_f+1)}$$

If a group detects the task after it spawns we can estimate the time it takes to solve it by

$$t_i = \frac{minimum(l * E[d]_{1:c}, 0.672 * r_d) - d_m}{dt * v}$$

where

$$c = round(\frac{n_{ct}}{t} + n_{tf} + 1)$$

$0.672r_d$ is the expected distance to a uniform point in a circle from the center of that circle. As the detection radius gets large this distance fails as good estimator because the tasks are bound by the search space not the detection radius. So smallest expected distance of c draws serves mostly very large agents, which have a high probability of already seeing all the free tasks in addition to the one spawning. There is also a probability that a free agent detects the spawning task and therefor has to wait for other agents to come and help it. In this scenario the solution time is different, since one needs to account for the probability that a group is called a simple algorithm 3 computes the expected distance. This algorithm could probably be reduced to an algebraic expression.

Algorithm 12 Expected distance with clusters

```
n number of agents
$n_{ar}$ number of agents required to solve task
N number of iterations
l side length of search area
$n_p = n - n_{ar} * n_{ct} + n_{ct}$
$n^*_{ar} = min([n_p, n_{ar}])$
$p_{cac} = \frac{n_{ct}}{n_p - 1}$
E_d=0 expected distance
for _ in 1:N

    current_pos=rand(2,1)*l
    agents_pos=rand(2,$n_p$-1)*l
    dists=d(current_pos,agents_pos)
    dists=sort(dists)
    E_d+=$(1 - pcac) * dists[n_ar - 1] + p_{cac}\frac{1}{n^*_{ar}-2}\sum_{i=1}^{n^*_{ar}-2} dists[i]$

end
return $\frac{E\_d}{N}$
```

Then we calculate t_{iw} as $t_{iw} = \frac{l*E_d - d_m}{dt*v}$. Lastly all timers are divided by n_{ct} to scale them by how many tasks the system works with at once. We can then construct the new model M_2.

$$M_2 = \begin{bmatrix} 1-P & P & 0 & 0 & 0 \\ 0 & 1-\frac{1}{max(t_t,t_w,1)} & \frac{1}{max(t_t,t_w,1)} & 0 & 0 \\ 1-P_i & 0 & 0 & \frac{n_{ct}}{n_p}P_i & \frac{n_p - n_{ct}}{n_p}P_i \\ 0 & 0 & \frac{1}{max(t_i,1)} & 1-\frac{1}{max(t_i,1)} & 0 \\ 0 & 0 & \frac{1}{max(t_{iw},1)} & 0 & 1-\frac{1}{max(t_{iw},1)} \end{bmatrix}$$

State 1 represents “normal” search where no new tasks are spawning and agents detect the tasks by new area discovered each step. State 2 has the time that a task is solved assuming all agents are uniformly distributed. State 3, also the measuring state, simply transport us with a either state 1 if there is no instantaneous detection. State 4 if a group detects a task and state 5 if a free agent detects a task.

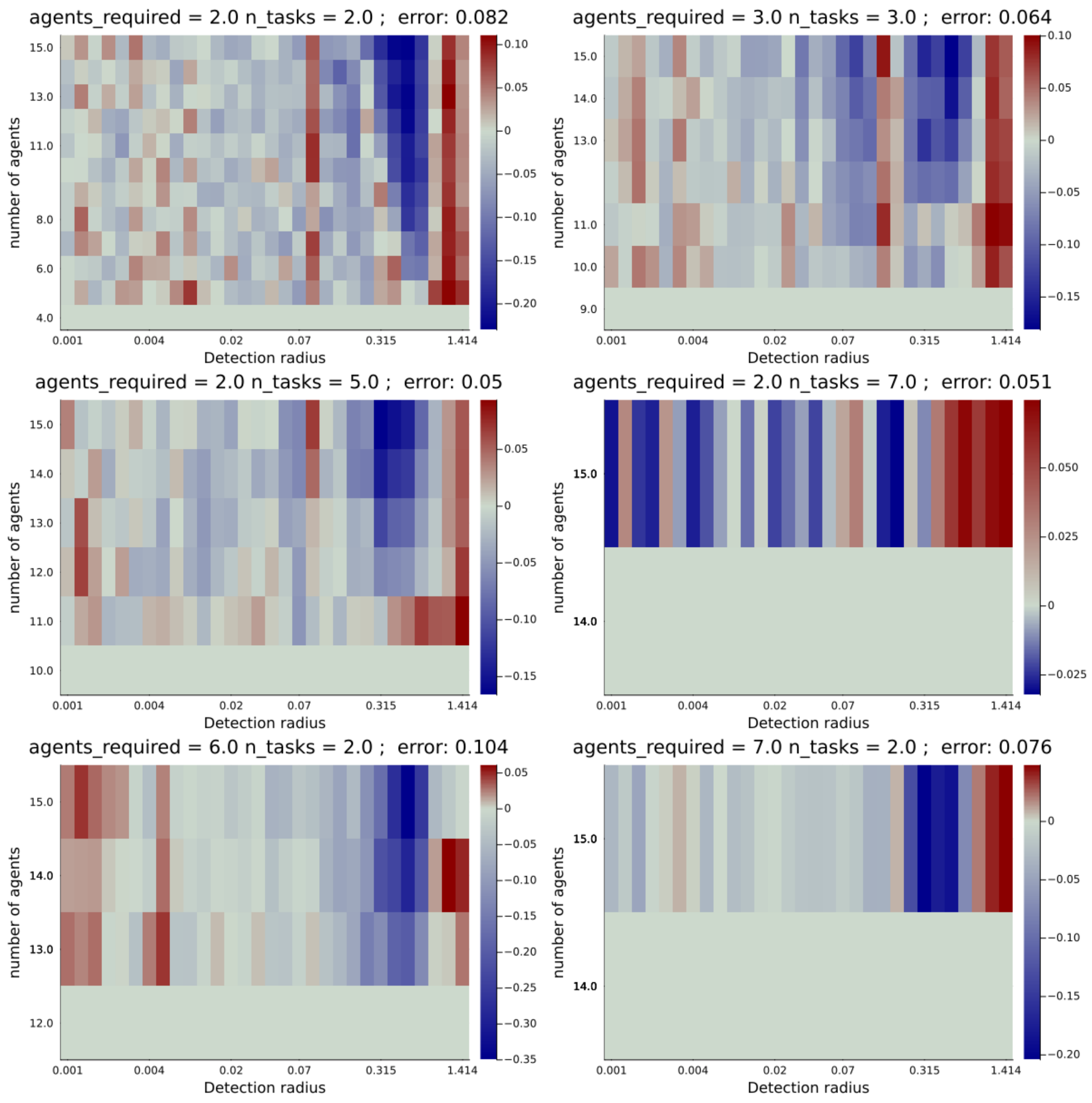


Figure 48: DTMC model 2 performance normalized by minimum number of agents

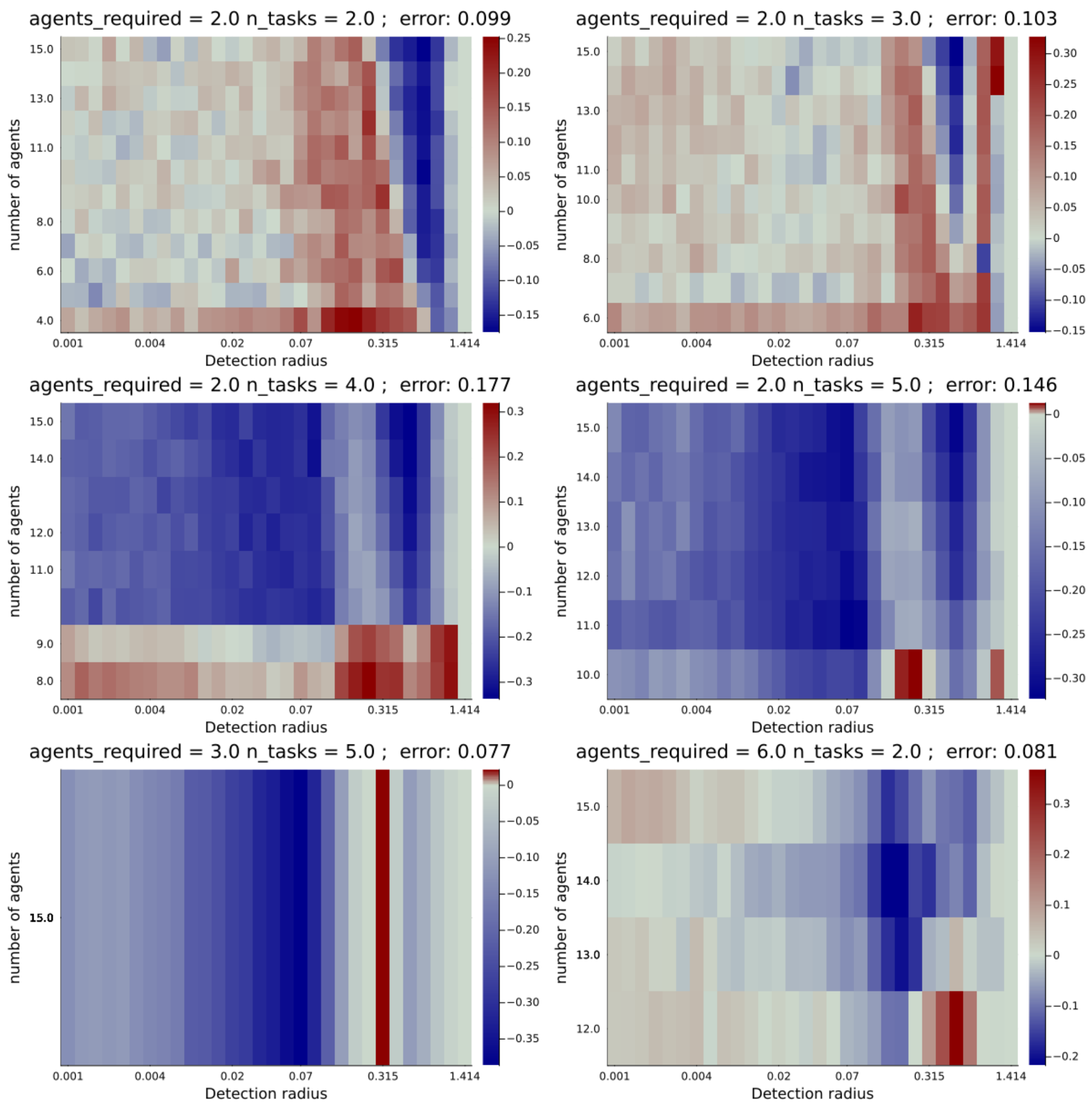


Figure 49: DTMC model 2 performance normalized by r_d max

While the results aren't perfect they have improved. The average error stays mostly under 10 %, DTMC models like this can be built upon further to increase performance. DTMC models allows for combining different behaviors that occur in different parts of parameter space. If detection capability is low then $P_i \approx 0$ then

$$M_2 = \begin{bmatrix} 1-P & P & 0 & 0 & 0 \\ 0 & 1-\frac{1}{max(t_t,t_w,1)} & \frac{1}{max(t_t,t_w,1)} & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{max(t_i,1)} & 1-\frac{1}{max(t_i,1)} & 0 \\ 0 & 0 & \frac{1}{max(t_{iw},1)} & 0 & 1-\frac{1}{max(t_{iw},1)} \end{bmatrix}$$

And so a class is created of states 1,2 and 3. On the other hand if $P_i \approx 1$

$$M_2 = \begin{bmatrix} 1-P & P & 0 & 0 & 0 \\ 0 & 1-\frac{1}{max(t_t,t_w,1)} & \frac{1}{max(t_t,t_w,1)} & 0 & 0 \\ 0 & 0 & 0 & \frac{n_{ct}}{n_p} & \frac{n_p-n_{ct}}{n_p} \\ 0 & 0 & \frac{1}{max(t_i,1)} & 1-\frac{1}{max(t_i,1)} & 0 \\ 0 & 0 & \frac{1}{max(t_{iw},1)} & 0 & 1-\frac{1}{max(t_{iw},1)} \end{bmatrix}$$

creating a class of states 3,4 and 5. So after constructing a DTMC model one can study the different classes that arise in different parameter space and evaluate their effect on performance.

5 Discussion

Through this thesis, it has been shown that you definitely can model MASTA systems mathematically. Some parameter domains are more challenging than others. Simple models can predict performance accurately when the search is the dominating process. Task allocation with uniform agents can also be modeled relatively easily. Modeling challenges occur when there is a breakdown of spatial uniformity. Many different agent behaviors can be recognized in certain regions of parameter space. For example, observed group formations, where the groups detect tasks after solving one, the exchange of agents between groups and agents waiting for other tasks to be solved. These behaviors affect task allocation and search, and while possible to model, they pose a challenge. DTMC is a good way to tackle these behaviors since the properties of different behaviors can be strung together.

The models provide insight into MASTA systems in terms of performance. Where the models do not fit well, it indicates that more complicated behavior is occurring. This might be artifact behavior indicating behavior that can be optimized or something substandard about the simulator. So the models allow a two-pronged attack, improving the simulator by pointing out artifact behavior and optimizing unforeseen but necessary behavior. In terms of computational performance, it depends on the parameter space. The models can reduce computational requirements by many orders of magnitude for low detection capability. In some parts of the parameter space that requires more complicated models, such as the sum presented in 4.5.1, there probably is not much to gain. But systems with low detection capability are computationally expensive because they require many steps and runs for the mean to stabilize. So one can still reduce computation time greatly by using analytical models where they fit well and simulating where they do not.

While the research questions are answered, they are broad. The presented model might not provide insight or reduce computational demand for parts of parameter space that have not been addressed. There are many choices to make when implementing a simulator; agent activation, timed tasks, and varying speeds. In addition, when designing a real system, one must consider the physical aspects; in this simulator, no physics is implemented. Much more work must be done to make models like these reliable and easily applicable to any MASTA system, but we can say it is most likely possible.

6 Conclusion and future work

In this thesis, a MASTA model has been implemented in software, allowing for the simulation of single-task robots with instantaneous assignments. A number of multi-agent search configurations have been simulated and modeled. An approach to modeling task allocation was developed, combined with previous search models to model combined search and task allocation. Many models have been presented, motivated by observed system behavior and deviation from predictions. Finally, a general approach with DTMCs was presented as a possible approach to modeling over large parts of parameter space.

6.1 Future work

Future work could focus on modeling more communication mechanisms, such as call-outs. The model constructed in this thesis is not directly applicable to other multi-agent systems that have different operating principles.

The vast number of possible choices when implementing MASTA M&S creates challenges for general models. One approach would be to build a library of different models for different preferences. For example, other real systems might move in various patterns, such as MUGs, causing different losses. Having different models like these collected allows engineers and scientists to quickly construct the model that fits the most to their system.

By developing other modeling approaches, one can see many connections to existing physical models. One example would be source and sink models in fluid dynamics, where tasks that need solving act as sinks and solved tasks as sources. Stochastic partial differential equations should work in the context of modeling MASTA. Many constants and symmetries in MASTA systems can be exploited for this approach. Partial differential equations can also allow for complex behavior without explicitly defining it. Further, the performance of MASTA is, in essence, a counting problem, therefor renewal theory should apply; as with MASTA, tasks are solved at intervals that follow some probability distribution, equivalent to holding times in renewal theory. Further first-hitting-time models might be used to understand the distribution breakdown in MASTA. For example, all agents in an area are called to a task - a first-hitting-time model might be able to predict when an agent will arrive in this now-empty area.

Other frameworks, such as those for parallel computing, might be applied to MASTA systems. Surprisingly Amhdals law can be fitted almost precisely to the distance distribution of uniform points. MASTA systems often have multiple processes going in parallel, with multiple agents searching and solving tasks. Interestingly, when analyzing the movement of individual agents, one sees that the largest components of the Fourier transform of their position are very similar. Given some further insight, it might be possible to create a method that automatically models the difference between an arbitrary MASTA system and a simple uniform one. If such a method exists, it would allow for models to be adopted automatically to different MASTA systems.

Further, one might be able to develop controllers for MASTA given a model. For example, say agents are out searching for tasks in some area. Then, knowing the time it takes to solve a task, one can determine the density of tasks. If the density is low, it is possible to move agents to another area. On the other hand, if tasks take too long to solve, agents can be told to stick closer together.

References

[1] Ontology, epistemology, and teleology for modeling and simulation : Philosophical foundations for intelligent ms applications, 2013.

[2] Sameera Abar, Georgios K. Theodoropoulos, Pierre Lemarinier, and Gregory M.P. OâHare. Agent based modelling and simulation tools: A review of the state-of-art software. *Computer science review*, 24:13–33, 2017.

[3] Agents.jl. Abm framework comparison, 2022. [Online; accessed 26-October-2022].

[4] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner HÃ€hnle, Peter H Schmitt, and Mattias Ulbrich. *Deductive Software Verification – the KeY Book: From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer International Publishing AG, Cham, 2016.

[5] Carlos A Alfaro, Sergio L Perez, Carlos E Valencia, and Marcos C Vargas. The equivalence between two classic algorithms for the assignment problem. 2018.

[6] Barry C Arnold. *A first course in order statistics*, volume 54 of *Classics in applied mathematics ;*. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), Philadelphia, Pa., 2008.

[7] Osman Balci. Validation, verification, and testing techniques throughout the life cycle of a simulation study. *Annals of operations research*, 53(1):121–173, 1994.

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[9] Soon-Jo Chung, Aditya Avinash Paranjape, Philip Dames, Shaojie Shen, and Vijay Kumar. A Survey on Aerial Swarm Robotics. *IEEE Transactions on Robotics*, 34(4):837–855, Aug 2018.

[10] George Datseris, Jonas Isensee, Sebastian Pech, and TamÃ¡s GÃ¡l. Drwatson: the perfect sidekick for your scientific inquiries. *Journal of Open Source Software*, 5(54):2673, 2020.

[11] George Datseris, Ali R. Vahdati, and Timothy C. DuBois. Agents.jl: a performant and feature-full agent-based modeling software of minimal code complexity. *SIMULATION*, 0(0):003754972110688, January 2022.

[12] Gregory Dudek, MichaelR.M Jenkin, Evangelos Milios, and David Wilkes. A taxonomy for multi-agent robotics. *Autonomous robots*, 3(4):375, 1996.

[13] Erin Marie Fischell, Anne R. Kroo, and Brendan W. O'Neill. Single-Hydrophone Low-Cost Underwater Vehicle Swarming. *IEEE Robotics and Automation Letters*, 5(2):354–361, Dec 2019.

[14] Jose Manuel Galan, Luis R Izquierdo, Segismundo S Izquierdo, Jose Ignacio Santos, Ricardo del Olmo, Adolfo Lopez-Paredes, and Bruce Edmonds. Errors and artefacts in agent-based modelling. *Journal of artificial societies and social simulation*, 12(1), 2008.

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

[16] Jose Guerrero and Gabriel Oliver. Multi-Robot Task Allocation Strategies Using Auction-Like Mechanisms. page 12.

[17] Ã GÃŒrcan, O Dikenelli, and C Bernon. A generic testing framework for agent-based simulation models. *Journal of simulation : JOS*, 7(3):183–201, 2013.

[18] Matthew A. Joordens and Mo Jamshidi. Consensus Control for a System of Underwater Swarm Robots. *IEEE Systems Journal*, 4(1):65–73, Feb 2010.

[19] Krishnanand N. Kaipa and Debasish Ghose. *Glowworm Swarm Optimization*. Springer International Publishing, Cham, Switzerland, 2017.

[20] Alaa Khamis, Ahmed Hussein, and Ahmed Elmogy. Multi-robot Task Allocation: A Review of the State-of-the-Art. In *Cooperative Robots and Sensor Networks 2015*, pages 31–51. Springer, Cham, Switzerland, May 2015.

[21] G. Ayorkor Korsah, Anthony Stentz, and M. Bernardine Dias. A comprehensive taxonomy for multi-robot task allocation. *International Journal of Robotics Research*, 32(12):1495–1512, Oct 2013.

[22] Janusz Laski and William Stanley. *Software Verification and Analysis*. Springer-Verlag, London, England, UK, 2009.

[23] BenoÃ®t Legat, Robin Deits, Marcelo Forets, Oliver Evans, Gustavo Goretkin, Daisuke Oyama, Joey Huchette, Twan Koolen, Guillaume Dalle, chachaleo, bzinberg, Chase Coleman, Christian Schilling, Elliot Saba, Henrique Ferrolho, Hugo Trentesaux, Julia TagBot, Robert Schwarz, and Guillaume Berger. Juliapolyhedra/polyhedra.jl: v0.7.6, February 2023.

[24] Naomi E. Leonard, Derek A. Paley, Russ E. Davis, David M. Fratantoni, Francois Lekien, and Fumin Zhang. Coordinated control of an underwater glider fleet in an adaptive ocean sampling field experiment in Monterey Bay. *Journal of Field Robotics*, 27(6):718–740, Nov 2010.

[25] Qi Lu, G. Matthew Fricke, John C. Ericksen, and Melanie E. Moses. Swarm Foraging Review: Closing the Gap Between Proof and Practice. *Current Robotics Reports*, 1(4):215–225, Dec 2020.

[26] Mathias Minos-Stensrud. Exploring information-sharing in multi-robot systems. Master's thesis, UNIVERSITY OF OSLO, 2020.

[27] Ingunn Nilssen, Ãyvind ÃdegÃ¥rd, Asgeir J. SÃžrensen, Geir Johnsen, Mark A. Moline, and JÃžrgen Berge. Integrated environmental mapping and monitoring, a methodological approach to optimise knowledge gathering and sampling strategy. *Marine Pollution Bulletin*, 96(1):374–383, 2015.

[28] Ernesto Nunes, Marie Manner, Hakim Mitiche, and Maria Gini. A taxonomy for task allocation problems with temporal and ordering constraints. *Robotics and Autonomous Systems*, 90:55–70, Apr 2017.

[29] William L Oberkampf. Verification and validation in scientific computing, 2010.

[30] Shyam Parekh. Continuous time markov chains, fall 2020. "Lecture notes".

[31] Johan Philip. The probability distribution of the distance between two random. In *Points in a Box., TRITA MAT 07 MA 10, ISSN*, pages 1401–2278, 2007.

[32] Jorge Pena Queralta, Jussi Taipalmaa, Bilge Can Pullinen, Victor Kathan Sarker, Tuan Nguyen Gia, Hannu Tenhunen, Moncef Gabbouj, Jenni Raitoharju, and Tomi Westerlund. Collaborative multi-robot search and rescue: Planning, coordination, perception, and active vision. *IEEE Access*, 8:191617–191643, 2020.

[33] Christopher Rackauckas and Qing Nie. Differentialequations.jl â a performant and feature-rich ecosystem for solving differential equations in julia. *The Journal of Open Research Software*, 5(1), 2017. Exported from https://app.dimensions.ai on 2019/05/05.

[34] Brooks Reed and Franz Hover. Oceanographic pursuit: Networked control of multiple vehicles tracking dynamic ocean features. *Methods in Oceanography*, 10:21–43, Sep 2014.

[35] Sheldon M. Ross. *Introduction to Probability Models*. Elsevier, 11 edition, 2014.

[36] Kunal Shah, Grant Ballard, Annie Schmidt, and Mac Schwager. Multidrone aerial surveys of penguin colonies in Antarctica. *Science Robotics*, 5(47):eabc3000, Oct 2020.

[37] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591, 1965.

[38] Onn Shehory and Sarit Kraus. Methods for task allocation via agent coalition formation. *Artificial intelligence*, 101(1):165–200, 1998.

[39] Hongwei Tang, Anping Lin, Wei Sun, and Shuqi Shi. An Improved SOM-Based Method for Multi-Robot Task Assignment and Cooperative Search in Unknown Dynamic Environments. *Energies*, 13(12):3296, Jun 2020.

[40] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version x.y.z)*, YYYY. `https://www.sagemath.org`.

[41] Changyun Wei, Koen V. Hindriks, and Catholijn M. Jonker. Dynamic task allocation for multi-robot search and retrieval tasks. *Applied Intelligence*, 45(2):383–401, Sep 2016.

[42] Wikipedia contributors. Markov chain — Wikipedia, the free encyclopedia, 2022. [Online; accessed 10-January-2022].

[43] Guang-Zhong Yang, Jim Bellingham, Pierre E. Dupont, Peer Fischer, Luciano Floridi, Robert Full, Neil Jacobstein, Vijay Kumar, Marcia McNutt, Robert Merrifield, Bradley J. Nelson, Brian Scassellati, Mariarosaria Taddeo, Russell Taylor, Manuela Veloso, Zhong Lin Wang, and Robert Wood. The grand challenges of Science Robotics. *Science Robotics*, 3(14):eaar7650, Jan 2018.

[44] Yanwu Zhang, Michael A. Godin, James G. Bellingham, and John P. Ryan. Using an Autonomous Underwater Vehicle to Track a Coastal Upwelling Front. *IEEE Journal of Oceanic Engineering*, 37(3):338–347, Jun 2012.

[45] Haitao Zhao, Hai Liu, Yiu-Wing Leung, and Xiaowen Chu. Self-Adaptive Collective Motion of Swarm Robots. *IEEE Transactions on Automation Science and Engineering*, 15(4):1533–1545, Jun 2018.

[46] Robert Michael Zlot. *An Auction-Based Approach to Complex Task Allocation for Multirobot Teams*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, 9 2006.