Distributed Sequential Task Allocation in Foraging Swarms

Harry Goldingay
Department of Computer Science
Aston University
Aston Triangle
Birmingham, UK
Email: goldinhj@aston.ac.uk

Jort van Mourik
Non-linearity and Complexity Research Group
Aston University
Aston Triangle
Birmingham, UK

***Abstract*—When designing a practical swarm robotics system, self-organized task allocation is key to make best use of resources. Current research in this area focuses on task allocation which is either distributed (tasks must be performed at different locations) or sequential (tasks are complex and must be split into simpler sub-tasks and processed in order). In practice, however, swarms will need to deal with tasks which are both distributed and sequential. In this paper, a classic foraging problem is extended to incorporate both distributed and sequential tasks. The problem is analysed theoretically, absolute limits on performance are derived, and a set of conditions for a successful algorithm are established. It is shown empirically that an algorithm which meets these conditions, by causing emergent cooperation between robots can achieve consistently high performance under a wide range of settings without the need for communication.**

I. Introduction

Swarm robotics is an emerging field in which swarms of autonomous robots are employed to cooperatively solve tasks of which the robots are incapable individually [1]. Each robot in a swarm typically has only limited communication and local sensing capabilities, and decisions must be made based on local interactions. Hence, in a well designed system "intelligent" global behaviour is an emergent consequence of these interactions [2]. A swarm should be robust, as it is not critically dependent on any individual, and scalable, because robots' behaviour only depends on local interactions, not the size of the swarm. In such decentralised systems, *task allocation* to individuals , without centralised control, is key. As swarms are well suited for distributed problems [3], it is important that the task allocation mechanism works well in a distributed context where individuals do not have a full overview of the state of the environment [4]. This problem has been well studied in the swarm intelligence literature, and good solutions modelled on the behaviour of social insects [5], [6] or using market-like bidding mechanisms [7] have been proposed.

While the above studies show that simple tasks can be well allocated in a distributed environment, one of the difficulties in building a practical swarm robotics system is that real world tasks are inherently complex (exceeding the complexity of the robots attempting to solve them). Nouyan et al. [8] show that it is possible for swarms to solve complex problems by breaking them down into simpler sub-tasks. Such decomposition of problems is also seen is social insect colonies [9], a frequent source of inspiration for swarm robotics. Apart from complexity reduction, task decomposition may allow for a better use of resources (e.g. space [10]). However, as sub-tasks form part of a larger problem, they may have sequential dependencies; one sub-task (or set of sub-tasks) must be completed before another can be started. The speed with which the swarm can process a task is limited by its slowest sub-task. The task allocation problem has now become one of decentralised *sequential task allocation*, without centralised control and such that all sub-tasks are satisfied.

Current studies of sequential task allocation either focus on ensuring optimal usage of resources (e.g. robots, time) on each sub-task *globally* [10]–[12], or on optimal switching between sequential and non-sequential processing of tasks [9], [13] (when coordination for partitioning tasks incurs an overhead). However, a problem not addressed so far is *distributed sequential task allocation*: the optimal use of fractions of resources *locally* in a distributed problem. This is clearly an important issue as spatially distributed problems are a key application for swarm robotics [3], and poor local distribution of resources limits the performance as much as poor global distribution.

In this paper, we extend the classic "foraging" problem found in swarm robotics literature to test the capability to process distributed sequential tasks. We analyse the problem theoretically to establish bounds on performance, and identify the mechanisms for break down of cooperation between swarm members. Hence, we propose a set of policies for individuals for location selection without communication or a global knowledge. We test these policies under a range of system conditions, and show that high performance is achievable in all conditions with a single algorithm.

The paper is organized as follows. In section II we define and analyse the problem. In section III, we propose several solutions. In section IV, we present and analyse numerical results in comparison with the theoretical analysis. Finally, section V, contains the conclusions and an outlook to future work.

II. The Problem

In the literature, the sequential task allocation problem is frequently studied in the context of foraging; the swarm must gather resources in an environment and return them to a central nest. The task is split into two sub-tasks which must be

processed sequentially. At an intermediate location robots who have finished the first sub-task transfer the resource to robots waiting to start the second. In some studies, this transfer is directly between robots [10], [12], [14], whereas others make use of a cache in which resources can be deposited after completion of the first sub-task, regardless whether a robot is ready to start the second [9], [13], [15]. These studies, however, all use a single source and nest, and hence resources have a single entry-, exchange- and exit-point.

Since we are interested in a distributed setting, we extend this to a multiple source, single nest scenario. To reduce the need for exact temporal coordination between swarm members, exchanges between are indirect (via caches) and at multiple (distributed) locations. Robots are limited to local perception and can only discover the state of a location upon a visit. The swarm has no capacity for direct communication (communication incurs additional cost and complexity which may detract from swarm flexibility in some cases [9]), and is not centrally controlled. As the problem of optimal fractions of robots performing each sub-task has already been explored [10], [12], [14], this paper focusses on controlling where the sub-tasks are performed. As we concentrate on the essential characteristics of the problem environment rather than the physical representation of the robots, the members of the swarm are henceforth referred to as *agents*.

A. The Environment

The environment has a distributed set of prey sources, and the overall aim of the problem is to maximise **throughput**: the number of prey items taken from these sources back to a central nest. However, a foraging task cannot be completed in one go; it must be broken down into two sub-tasks: **harvesting** and **storing**, which must be performed sequentially by the agents. At any given time, an agent is specialized in one these sub-task types known as its **job**. Hence we can distinguish between **harvesting** and **storing** agents, but after completion they may choose another job.

Harvesting agents travel to a source of their choice (sources are assumed to always contain prey), pick up a prey item and transport it so that it can be processed by a storing agent. To facilitate the transfer of prey items from foraging to storing agents, harvesting agents deposit their items in a caches of finite capacity C, with one cache associated with each source. Storing agents travel to a cache of their choice, pick up a prey item if one is present and transport it back to the central nest which has unlimited capacity to receive prey. We describe the set of sequential tasks (i.e. harvesting from a specific source and storing prey found in its associated cache) as a **task chain**, and the problem environment consists of N_c task chains, as illustrated in figure 1.

Task chains are not homogeneous; in order to test algorithms' abilities to prioritize the most promising foraging locations, we introduce **travel times** such that harvesting and storing take different amounts of times at different task chains. In particular, each sub-task in a chain has its own travel time; when starting a task, agents take this long to reach the location that they are aiming to pick up a prey item (source and cache for harvesting and storing agents respectively) and then take the same amount of time to travel to the location to deposit it (cache and nest for harvesting and storing agents respectively).

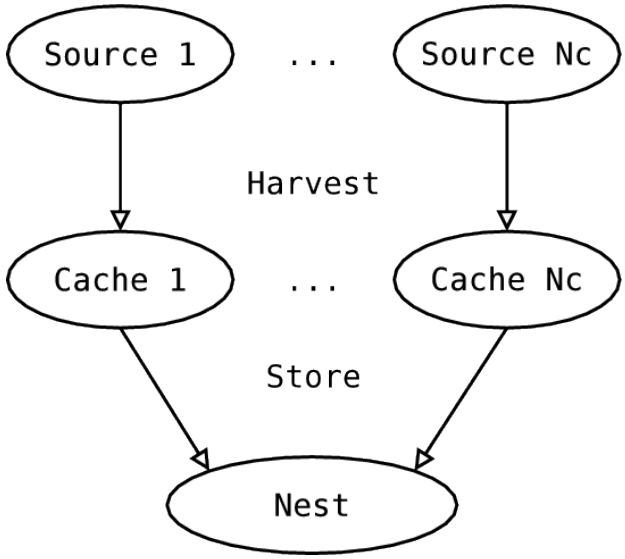


Fig. 1. An illustration of the problem environment. Agents *harvest* from sources and deposit prey in associated caches. Prey from these caches is then *stored* in a central nest.

Formally, the state of task chain c at time t is given by:

$$\mathcal{C}_c(t) = \{T_{c,H}, T_{c,S}, r_c(t)\} \tag{1}$$

where:

- $T_{c,H}$ is the travel time for the harvesting sub-task,
- $T_{c,S}$ is the travel time for the storing sub-task,
- $r_c(t)$ is the number of prey items in cache at time t (limited by the global cache capacity C).

The problem environment (i. e. the set of all N_c task chains), is denoted by: $\boldsymbol{\mathcal{C}}(t) \equiv \{\mathcal{C}_1(t), ..., \mathcal{C}_{N_c}(t)\}$.

B. The Agents

Foraging agents are homogeneous, with each agent capable of both harvesting and storing a single item of prey. These sub-tasks can only be taken on one at time, however, and, once engaged in, must be completed (either successfully or otherwise) before a new sub-task is chosen. An agent completes a sub-task only after attempting deposit prey, regardless of whether it has successfully picked prey up (this is equivalent to the need to return to a central hub to re-orient on failure, but viewing it as part of the task simplifies the description of the algorithm).

As discussed previously, to promote an efficient, scalable system, agents are not centrally controlled, nor do they communicate directly; instead, they must rely on environmental information to make good decisions about which task locations and sub-task types to select. However, this information may relate to a time prior to when the agent makes its decision (for instance, it could reasonably be based on how many items were in a cache when an agent last visited it). It may also be useful to gain a better estimate of some noise-dependent environmental factor by using its average rather than its value at a single time. In order to allow for this, we also give each agent a **memory**, denoted $\mathcal{M}_a(t)$: a set of information about the environment

upon which an agent can base its decisions. The details of this memory depend on which task-chain selection policy is being used and are discussed in section III.

Therefore, the full state of agent a at time t is given by:

$$\mathcal{A}_a(t) = \{C_a(t), J_a(t), \tau_a(t), \mathcal{M}_a(t)\} \quad (2)$$

where:

- $C_a(t)$ is the current task chain.
- $J_a(t)$ is the current job type, taken from the set $\{H(\text{arvesting}), S(\text{toring})\}$.
- $\tau_a(t)$ is the time spent undertaking its current sub-task, needed to keep track of how close it is to completion.
- $\mathcal{M}_a(t)$ is its memory

The state of the whole swarm (i.e. set of all N_a agents) is denoted by: $\boldsymbol{\mathcal{A}}(t) = \{\mathcal{A}_1(t), ..., \mathcal{A}_{N_a}(t)\}$.

C. The Simulation

The foraging problem proceeds in discrete time-steps of size 1, starting at $t = 1$. Agents are initialized with task chain and sub-task type randomly drawn from a uniform distribution of the possible task chains and sub-task types respectively and $\tau_a(1)$ is defined to be 0 so that these tasks are started at the beginning of the run. Unless explicitly changed in the algorithm below, an agent's variables are assumed to be unchanged after a time-step (e.g. $C_a(t) = C_a(t-1)$). At time t, the simulation proceeds as follows:

1) Agents engage in their current sub-task (updating $\tau_a(t) \rightarrow \tau_a(t)+1$), and become ready to collect (resp. deposit) after $T_{C_a(t),J_a(t)}$ (resp. 2 $T_{C_a(t),J_a(t)}$) time-steps.
2) Agents ready to deposit, act in a (uniform) random order, succeeding if they are:
   - A harvesting agent which is at a chain with a non-full cache ($r_{C_a(t)}(t) < C$), increasing the number of items in the cache ($r_{C_a(t)}(t) \rightarrow r_{C_a(t)}(t) + 1$).
   - A storing agent which has picked up a prey item from the cache during its current sub-task.

   After depositing, agents:
   - Change their job based on the rule from section II-D.
   - Update $\mathcal{M}_a$ and choose a new task chain based on one of the policies from section III.

   $C_a(t+1)$ and $J_a(t+1)$ are updated to reflect the newly chosen sub-task and chain. $\tau_a(t+1)$ is reset to 0.
3) Agents ready to collect act in a (uniform) random order, picking up prey if they are:
   - A harvesting agent.
   - A storing agent which is at a chain with a non-empty cache ($r_{C_a(t)}(t) > 0$), decreasing the number of items in the cache ($r_{C_a(t)}(t) \rightarrow r_{C_a(t)}(t) - 1$).

Note that in steps 2-3, we specify that agents must act in a random order. While our simulation is discrete, it approximates a continuous problem, in which agents would not arrive at exactly the same time such that they could be easily ordered according to some rule. In order to prevent an algorithm gaining a performance advantage from some unrealistic ordering of agents, we specify that they must act in a random order.

D. Sub-task Type Selection

Note that agents have two sources of freedom in their behaviour. After depositing a prey item (step 2 in the algorithm) agents must both choose a new job (i.e. sub-task type), and a new task chain to visit. Although a mechanism for taking a joint decision may be expected to optimize performance, it is beyond the scope of this paper; for simplicity, we treat these decisions as independent. As previously discussed, the problem of balancing the numbers of agents between jobs was previously explored, and we focus on the best policy for chain selection (section III). We do, however, need some method for sub-task selection. As task type selection methods in the literature are aimed at balancing global levels of agents doing each job (rather than the local levels important in a distributed setting), and depend on information not present in our setting (e.g. waiting times, either at caches or in direct handover of prey items), we opt for a single simple method.

On successful completion of the previous sub-task, agents stick to the same job, whereas on failure they switch job with probability 0.5. Although we lay no claims to optimal performance, the global number of agents for each sub-task type is balanced in our experimental setup, hence this switching probability should lead a good global distribution of agents to jobs.

E. Performance Measures

Before designing policies which aim to lead to good allocations of agents to chains, it is useful to consider what an optimal centralised allocation would be with full knowledge of the system. By devising and analysing the performance of this optimal allocation, we can establish both the conditions for a good algorithm, and derive upper limits on performance. Any practical algorithm can then be tested against these limits. Here, we show how to maximise throughput and quantify the performance of such a system. We also show that this places a limit on the number of agents that can be engaged in useful work at a given task chain.

1) Maximum Performance: At any given chain c the maximum number of tasks which can be processed is limited by the capacity C of the cache. Assuming the cache is kept full, in a perfectly coordinated system it would be possible for groups of C agents to pick up prey items from the cache every 2 time-steps [1]. As an agent needs 2 $T_{c,s}$ time-steps to deliver its collected resource to the nest and return to the cache to pick up a new resource, $T_{c,S}$ such groups of agents would be required to fully exploit a task chain with a continually full cache. For similar reasons, if we assume that a cache is constantly being

[1]Note that this factor of 2 is a function of the discrete nature of our simulation and the fact that agents take the same time to pick up a resource as they do to deliver it. An agent starting a sub-task with travel time T will finish it after $2T$ time-steps and, because all agents start the simulation at time $t = 0$, agents can only start a task on an even time-step. It would be possible to rescale the system time to remove this factor, but it would then be necessary to use half time-steps.

emptied, we would need $T_{c,H}$ groups of C harvesting agents to refill the cache every 2 time-steps.

A chain is **saturated** if it exactly meets the conditions on the numbers of visiting agents set out above. If it has too few agents, it is **under-saturated** while it is **over-saturated** if it has too many. Exactly $C(T_{c,H} + T_{c,S})$ agents are required to saturate chain c. The total travel time $T_{ct} \equiv T_{c,H} + T_{c,S}$ also determines the maximum efficiency of agents at the (saturated) chain c: an agent can on average complete $\frac{C}{2\ C\ T_{ct}} = \frac{1}{2T_{ct}}$ tasks per time-step. Using similar reasoning, we can see that this average is an upper limit on the performance of agents at under-saturated task chains, although in these cases no allocation of agents to sub-tasks that actually reaches this limit may exist. For maximum performance we do not need to consider over-saturation as adding agents to a saturated location cannot increase performance.

Since we have a finite number of agents and want to maximise the number of completed tasks, it is clear that we should aim to maximise the average number of tasks completed per-agent. As this average monotonically decreases with total travel time, maximal throughput is obtained by saturating locations with the lowest total travel time first. Then the maximum throughput of the system per time-step is closely approximated [2] by $f(N_a, 2)$ (recursively) defined as:

$$f(N,T) = \begin{cases} 0 & \text{if } |\boldsymbol{\mathcal{C}}_{T'}| = 0,\ \forall T' \geq T, \\ \frac{N_T}{2T} + f(N - N_T, T+1) & \text{if } N \geq N_T, \\ \frac{N}{2T} & \text{otherwise.} \end{cases} \tag{3}$$

where $\boldsymbol{\mathcal{C}}_T = \{\mathcal{C}_c | T_{c,H} + T_{c,S} = T\}$ is the subset of chains with total travel time T, and $N_T = T\ C\ |\boldsymbol{\mathcal{C}}_T|$ is the number of agents needed to saturate all these locations.

2) Useful Agents: Achieving maximum performance requires perfect coordination between agents which, in a practical context, is difficult to achieve. While comparing the throughput of a concrete algorithm with the theoretical maximum can indicate how well coordinated agents are, it does not tell us about how their coordination is breaking down at a given location. We measure this by defining the number of **useful agents** at a location. Let a task chain have a total of v_H harvesting agents and v_S storing agents acting (i.e. in moving to collect or deposit a resource) at it at a given time-step. If $v'_H \leq v_H$ and $v'_S \leq v_S$ are the minimum numbers of harvesting and storing agents, respectively, required to maintain the current throughput at the chain, then the number of useful agents is given by $v'_H + v'_S$. Intuitively, this is the minimum number of agents who could achieve the same throughput as the current set of visiting agents without switching sub-task.

In two situations a visiting agent can fail to contribute to throughput at a task chain. At an already saturated or over-saturated chain, any agent above the number required for saturation does not contribute to useful work. Secondly, at an under-saturated chain with an unbalanced number of harvesting and storing agents in relation to the travel times (i.e. if one sub-task is being done more slowly than another), the agents undertaking one of the sub-tasks will fail (either harvesting agents because the cache is full or storing agents because the cache is empty). In a balanced setting, we have $T_{c,S}\ v_H = T_{c,H}\ v_S$ therefore, for v_H harvesting agents at a chain, a maximum of $\frac{T_{c,S}\ v_H}{T_{c,H}}$ storing agents can be useful, and vice-versa.

Combining the limits on the number of useful agents at over- and under-saturated chains into a single rule, we obtain that at any chain with (v_H, v_S) agents, the number of useful agents is limited from above by:

$$u(c, v_H, v_S) = T_{c,H}\ \min(C, \frac{v_S}{T_{c,S}}) + T_{c,S}\ \min(C, \frac{v_H}{T_{c,H}}) \tag{4}$$

III. Task Chain Selection

When selecting a task chain, an agent must balance two factors: the inherent quality of the chain (measured in terms of travel time) and the existing population of agents at the chain (with whom it should cooperate rather than compete). From the reasoning in section II-E, we can formulate the following principles for a "good" algorithm:

1) *Agents should minimize their travel times* - Agent performance is limited by the total travel time at the chain. The direct evidence an agent has for the total travel time is the time it spends performing its own sub-task. By minimizing this, it should be able to perform tasks more quickly, thus increasing the system's overall throughput.
2) *Agents should maximise the proportion of sub-tasks in which they succeed* - The simplest measure of whether an agent is cooperating or competing with agents at its chosen task chain is whether it succeeds or fails. Frequent success implies that it has sufficient agents performing the other sub-task at the chain (cooperation), and not too many competing agents doing its own sub-task.
3) *Agents should maximise their "usefulness"* - An agent may be blocking other agents from doing work. If an agent can estimate its own usefulness, it can target work that contributes most to the system's throughput.

We propose a set of **task chain selection policies**, i.e. a collection of rules and information used by agents to select their task chain. Each policy is based on a different combination of principles 1-3, such that we can test which are most useful for maximising throughput. Task chain selection is based on agent memory which contains, at least a vector of preferred task chains $\vec{P}_a = \{P_{a\ell} : 1 \leq \ell \leq L\}$ of limited maximum size L (the value of L depends on the task selection policy used). Memory is initially empty but, when an agent completes a sub-task successfully, it can add its current task chain to the vector if it is not already present. It will do this automatically if $\vec{P}_a$ has not reached its maximum size, but may do so otherwise according to its selection policy rules.

When choosing a new task chain to visit, agents return to one of their preferred task chains in $\vec{P}_a$ with probability p_r, or select a task chain randomly from the set of all task chains

[2] Note that the only difference between the true maximum and this approximation comes is for the case $N < N_T$ with not enough agents to saturate all locations in $\boldsymbol{\mathcal{C}}_T$. In this case we may, in the worst case, end up with a single under-saturated task chain at which the agents' maximum average performance is limited above by $\frac{1}{2T}$. However, in a system with a large number of task chains (as studied in this paper), the difference between f and the true maximum is negligible and we prefer it for its simplicity.

with probability $1 - p_r$. When $\vec{P}_a$ is empty, the task chain is also randomly selected. By sometimes allowing agents to choose a random location, we prevent the situation where they are trapped with a sub-optimal set of preferences.

A. Random Selection Policy

In order to establish a baseline, we test our policies against one in which agents choose new task chains randomly, denoted **random selection policy (RSP)**. Agent memory formally contains preferred locations ($\mathcal{M}_a = \{\vec{P}_a\}$), but we limit the number of preferred locations to $L = 0$, such that agents select a random task chain.[3]

B. Greedy (Time) Selection Policy

The **Greedy (Time) Selection Policy (GTSP)** is based on the 1^{st} principle outlined at the start of section III: agents should minimize their travel times. Agents greedily select the task chain with the lowest travel time they encounter. Agent a has a memory $\mathcal{M}_a = \{\vec{P}_a, t_{P\,a}\}$ where:

- $\vec{P}_a$ has a maximum size of $L = 1$
- $t_{P\,a}$ is the lowest sub-task travel time a has encountered so far.

Upon successful completion of a sub-task, agent a compares the time spent during that sub-task to the best time encountered so far. If $\tau_a(t) < t_{P\,a}$,the memory is reset: $\vec{P}_a = \{C_a(t)\}$ and $t_{P\,a} = \tau_a(t)$. Since memory is retained on task switch (and travel times for sub-tasks are not guaranteed to be identical), an agent always updates $t_{P\,a}$ when returning to the same location. In this way, an agent minimizes its travel time given its current knowledge. Conversely, this rule only tangentially takes success rate into account: the agent must successfully complete a sub-task for the task chain to become preferred but, once a location with low enough travel time is added, the agent will keep returning regardless of subsequent success.

C. Greedy (Cache) Selection Policy

The **Greedy (Cache) Selection Policy (GCSP)** is based on the 2^{nd} principle outlined at the start of section III: agents should maximise the proportion of sub-tasks in which they succeed. An agent's success or failure in a sub-task depends entirely on the state of the cache at the visited chain. A harvesting agent which tries to deposit a prey item in a full cache will fail, as will a storing agent when trying to pick from an empty cache. Therefore, harvesting (storing) agents greedily select the task chain which they encounter which has the lowest (highest) number of prey items in its cache at the time of visit. Agent a has a memory $\mathcal{M}_a = \{\vec{P}_a, \rho_{P\,a}\}$ where:

- $\vec{P}_a$ has a maximum size of $L = 1$
- $\rho_{P\,a}$ is number of items found in the cache of the agent's preferred task chain at last visit.

Upon successful completion of a sub-task, an agent compares the number of items encountered in the cache to $\rho_{P\,a}$. If the agent is harvesting (storing), it switches preferred task if it finds fewer (more) items in the cache, updating $\vec{P}_a$ and $\rho_{P\,a}$ to match. Agents always update $\rho_{P\,a}$ when returning to their preferred locations such that they act on the most current information. This policy should lead to high success rates and promotes cooperation; an agent is more likely to take up a sub-task if another agent is carrying out the other sub-task at the location (as evidenced by the cache). However, as travel time is not taken into account at all, agents are not driven to maximise the rate at which they can do tasks.

D. Success Time Selection Policy

The **Success Time Selection Policy (STSP)** is motivated by the fact that the two greedy policies introduced above target one of our first two principles while ignoring the other. Agents using the GTSP could be driven to fail very often, while those using the GCSP could be driven to succeed but very slowly. In order to avoid these flaws, we must combine the 1^{st} and 2^{nd} principle into a single rule by explicitly targeting *success rate* - one over the expected number of steps that it takes for an agent to succeed in its sub-task at a chain.

This cannot be estimated from the results of a single attempt at a sub-task (as an agent can only succeed or fail in a task, the rate only takes the extreme values $\frac{1}{2T}$ or 0). Therefore, an agent needs to maintain both an average (to reduce its error when estimating success rate) and more than one preferred location. When an agent completes a task at a location not present in its set of preferences, it must choose whether to discard one of the preferred locations based on the single measurement at this new location (it is impractical to maintain average results for all possible locations). If the agent were only able to retain a single preferred location, it again has the problem of being forced to base the entire content of $\vec{P}_a$ on an extreme value for success rate at a location.

To mitigate this, we give agents a memory $\mathcal{M}_a = \{\vec{P}_a, \vec{\tau}_a, \vec{\sigma}_a\}$ where:

- $\vec{P}_a = (P_{a,1}, ..., P_{a,L})$ has a maximum of size $L = 5$.
- $\vec{\tau}_a = (\tau_{a,1}, ..., \tau_{a,L})$ is a vector of total travel times, where $\tau_{a,\ell}$ stores the cumulative time that the agent has spent travelling at task chain $P_{a,\ell}$ since it was added to the agent's memory.
- $\vec{\sigma}_a = (\sigma_{a,1}, ..., \sigma_{a,L})$ is a vector of total successes, where $\sigma_{a,\ell}$ stores the number of times that an agent has successfully completed a sub-task at task chain $P_{a,\ell}$ since it was added to the agent's memory.

With this information, an agent can assign a weight $w(\tau_{a,\ell}, \sigma_{a,\ell})$ to each preferred task chain ℓ where $w(\tau, \sigma) = \frac{\sigma}{\tau}$ is the agent's observed success rate at ℓ. After successful completion of a sub-task at a chain not in $\vec{P}_a$, an agent compares the observed success rate at this new location ($w(\tau_a(t), 1)$) to the lowest success rate at the preferred locations. If the success rate at the new location is higher, it replaces this lowest weighted preferred task chain in the agent's memory. When an agent visits preferred location ℓ it updates its related variables, adding its travel time to $\tau_{a,\ell}$ and increasing $\sigma_{a,\ell}$ by 1 if was successful.

With probability p_r, an agent using STSP randomly chooses a location from $\vec{P}_a$ with probability proportional to

[3]Note that we could define a random task selection policy more naturally, but doing it this way allows for a single framework to treat all policies.

its weight. It is worth noting that agents retain all information in their memory even after switching jobs; while this may lead to the agent having an inaccurate estimate of the success rate of sub-tasks for this job, we have observed empirically that the benefit (of driving agents towards locations with low total travel times) significantly outweighs the disadvantages. This policy should avoid the flaws of the other two greedy algorithms but, not being specialized, is likely to be worse than one or other of them in situations where high selectivity in one of the criteria is crucial.

E. Global Success Time Selection Policy

A memory based on success rate is a good way for agents to maximise their own performance. However, this does not necessarily maximise the performance of the whole system if the proportion of useful work done is too low (see principle 3 at the start of section III). For example, if we have two task chains which each have capacity 1, one with travel time 1 and one with travel time 5, and two agents to serve them then the agents will maximise their own success rate by both choosing the task chain with low travel time (each expecting a success rate of $\frac{0.5}{2\times1} = 0.25$ if they compete, and an expected success rate of $\frac{1}{2\times5} = 0.1$ for an agent that switches). However, as only one agent can do useful work for this sub-task (due to a capacity and travel time of 1) this leads to a sub-optimal total success rate ($0.25+0.25 = 0.5$) compared to one agent serving the low travel time chain and the other serving the high travel time ($\frac{1}{2\times1} + \frac{1}{2\times5} = 0.6$).

The proposed policy for solving this problem, the **Global Success Time Selection Policy (STSP)**, is similar in spirit to the algorithms proposed by Pini et al. [9] in that agents base their decision to switch task chain based on their own estimates of the costs and benefits of doing so. The difference, however, is that agents assume they are part of a relatively small population at any given task chain such that their presence at a location can have a non-negligible effect on the performance of other agents. This effect is taken into account when deciding at what task chain the agent would be most useful. As with STSP, agents measure performance in terms of success rate.

Given that agents cannot directly communicate, an agent must infer its own effect from its observations of the environment. It has a memory $\mathcal{M}_a = \{\vec{P}_a, \vec{\tau}_a, \vec{\sigma}_a, \vec{v}_a, \hat{\tau}_a, \hat{\sigma}_a\}$ where

- $\vec{P}_a$, $\vec{\tau}_a$ and $\vec{\sigma}_a$ are defined as in STSP.
- $\vec{v}_a = (v_{a,1}, ..., v_{a,L})$ is a vector of total visits, where $v_{a,\ell}$ stores the number of times that an agent has attempted to perform a sub-task at task chain $P_{a,\ell}$ since it was added to the agent's memory.
- $\hat{\tau}_a$ and $\hat{\sigma}_a$ are global versions of the elements of $\vec{\tau}_a$ and $\vec{\sigma}_a$. They store, respectively, the total time which the agent has spent travelling at any task chain and the total number of sub-tasks successfully completed by the agent.

When deciding upon the task chain to visit, or whether a chain should be added to $\vec{P}_a$, the agent estimates the local decrease (or otherwise) and the global increase in success rate that it would cause by choosing a different task chain.

The following shows how an agent estimates its effect when visiting a task chain from memory (with the indices a, ℓ omitted from estimates for convenience), but calculations for a new location are the same. Given the observed success percentage, an agent estimates the size N of the local population at a task chain as the follows:

$$N_{\text{est}} = \frac{C\ v_{a,\ell}}{\sigma_{a,\ell}} \tag{5}$$

under the assumptions[4] that the task chain always has a full cache, and that its own success rate is representative for the local population.

The estimated local population size is then used to estimate the success percentage P of the remaining agents assuming that one agent leaves:

$$P_{\text{est}} = \begin{cases} \min\left(\frac{C}{N_{\text{est}}-1}, 1\right) & \text{if } N_{\text{est}} > 1, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

The increase in local task completion rate at ℓ *attributable* to agent a, is then calculated as:

$$\Delta R^{\text{local}}_{\text{est}} = \left(N_{\text{est}}\ \frac{\sigma_{a,\ell}}{v_{a,\ell}} - (N_{\text{est}} - 1)\ P_{\text{est}}\right) / \left(\frac{\tau_{a,\ell}}{v_{a,\ell}}\right) \tag{7}$$

which can be interpreted as the difference between the number of useful agents, with and without a, divided by the observed travel time at ℓ to give a completion rate.

An agent should also consider what it loses by not acting elsewhere. While ideally we should treat an agent's global contribution in the same way as its local one: taking into account its likelihood of being "useful", this is significantly more complex in the global case. While we could use the agent's global success percentage to infer information about how often it is useful globally, we would also need to know how this changes with travel time for it to be meaningful. A policy based on this information would increase the complexity of the algorithm significantly, hence we omit it. Instead, we base an agent's global contribution on its expected task completion rate assuming that it does not act at chain ℓ, given by:

$$\Delta R^{\text{global}}_{\text{est}} = \begin{cases} \frac{\hat{\sigma}_a - \sigma_{a,\ell}}{\hat{\tau}_a - \tau_{a,\ell}} & \text{if } \tau_{a,\ell} \neq \hat{\tau}_a \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

This is equivalent to the assumption that agents are useful at locations other than the one they are considering travelling to (for which they estimate their own usefulness). While this assumption does not always hold, its breakdown implies that there is redundancy in the system (because swarm is sufficiently large such that agents are frequently not doing useful work) and, as such, that the performance of individual agents is not as critical as in other circumstances.

Finally, an agent uses these two estimates to quantify its preference for each task chain:

$$p(a, \ell) = \Delta R^{\text{local}}_{\text{est}} - \Delta R^{\text{global}}_{\text{est}}\ . \tag{9}$$

[4]Although the first assumption is not always true, C normally cancels out before it is used by the agent to make a decision (except in the rare case where $C < N_{\text{est}} < C + 1$ when it only has a minor effect). The second assumption is more plausible, because agents act in random order.

TABLE I. PROPORTION OF TASK CHAINS WITH TRAVEL TIMES TAKEN FROM THE SET $\{1, 2, 5\}$ WITH RELATIVE FREQUENCIES $\{0.5, 0.3, 0.2\}$.

	$T_{c,H} = 1$	$T_{c,H} = 2$	$T_{c,H} = 5$
$T_{c,S} = 1$	0.25	0.15	0.10
$T_{c,S} = 2$	0.15	0.09	0.06
$T_{c,S} = 5$	0.10	0.06	0.04

Agents always visit the task chain with the highest preference and replace their least preferred chain with a newly encountered one if the preference for the new chain is higher. Since a newly encountered task chain (at which an agent has just successfully completed a task) has an observed success percentage of 1, travel time of $t_a(t)$ and a single visit, its preference value is given by:

$$p_{new} = \frac{1}{t_a(t)} - \frac{\hat{\sigma}_a - 1}{\hat{\tau}_a - t_a(t)} \tag{10}$$

As in STSP, agents retain their memory after switching jobs.

IV. RESULTS

In this section, we first discuss the experimental setup, and the parameters used in it. Then we measure the performance of the various task selection policies discussed in the previous section, under a wide range of conditions. Finally, we study the efficiency of agent allocation for these policies in more detail, in particular as a function of N_a for a specific representative setting of all other parameters. We identify where and why these policies fail in comparison to ideal performance.

A. Experimental setup and parameters

The most important parameters that we independently control in our simulations are the number of agents N_a, and the cache size C. Then we take the number of task chains as $N_c = 10000/C$, such that the total number of sub-tasks is constant when C is varied. Next we set the distribution of the travel times $T_{c,H}, T_{c,S}$. For a given set of travel times $\{T_i,\ i = 1, .., n_T\}$, we only fix their relative frequencies $\{f_i,\ \sum_i f_i = 1\}$, such that the number of task chains with travel times $(T_{c,H}, T_{c,S}) = (T_i, T_j)$ is given by $N_c\ f_i\ f_j$, as illustrated in table I. We set the probability p_r that a preferred task chain is chosen to the empirically determined (close to optimal) value of $p_r = 0.9$.

In all experiments the algorithms run for 5000 iterations and throughput is calculated as the total taken over the entire run. Useful/wasted visits are only measured in the final iteration such that agents have time to explore the environment and reach cooperative groups if led to by their task chain selection policies. All results are averaged over 20 runs. Standard deviation was measured, but is negligible (comparible to the line width in the graphs) and so has not been included.

B. Performance

In this section, We test the absolute performance of the task selection policies under variations of the cache capacity C, the number of agents N_a, and the distribution of travel times.

We have taken two values for C: a low one $C = 10$ and a high one $C = 100$, to test whether the increased pressure on coordination changes the relative performance of the selection policies.

TABLE II. PERFORMANCE AS A FRACTION OF THE MAXIMUM FOR TRAVEL TIMES $\in \{1, 2, 5\}$ VARYING BOTH N_a AND C

C	10			100		
N_a	5000	23500	42000	5000	23500	42000
RSP	0.431	0.667	0.809	0.471	0.744	0.938
GTSP	**0.799**	0.506	0.524	**0.868**	0.521	0.548
GCSP	0.435	0.700	0.871	0.475	0.750	**0.962**
STSP	0.700	0.788	0.793	0.798	0.852	0.828
GSTSP	0.686	**0.837**	**0.892**	0.835	**0.938**	0.957

TABLE III. PERFORMANCE AS A FRACTION OF THE MAXIMUM FOR TRAVEL TIMES $\in \{1, 10\}$ VARYING BOTH N_a AND C

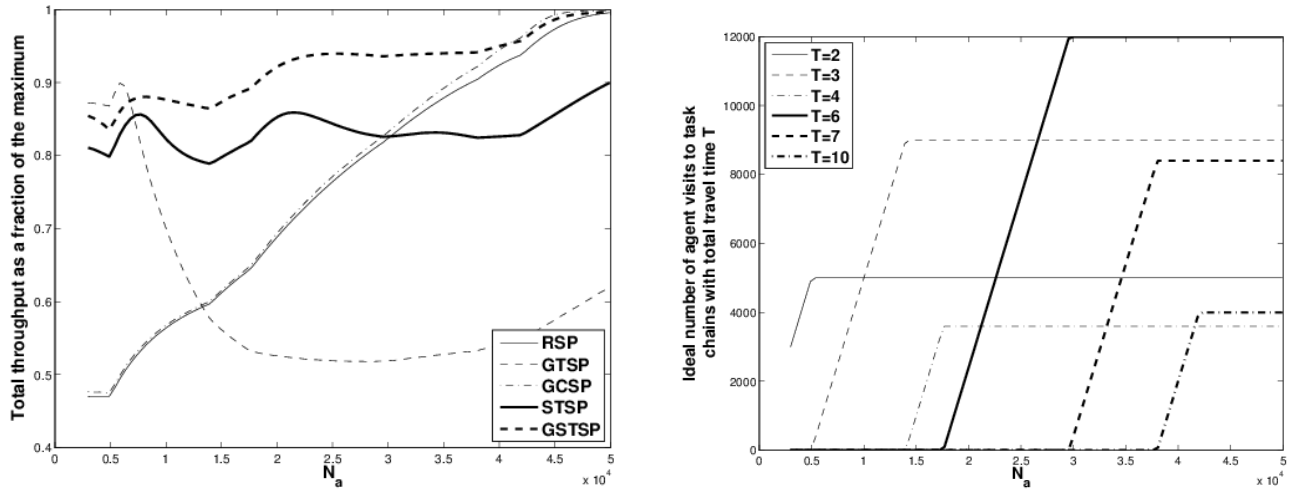
C	10			100		
N_a	5000	57500	110000	5000	57500	110000
RSP	0.164	0.630	0.801	0.179	0.699	0.926
GTSP	**0.565**	0.682	0.746	**0.609**	0.713	0.793
GCSP	0.152	0.645	0.860	0.185	0.726	**0.950**
STSP	0.448	0.792	0.799	0.530	**0.841**	0.832
GSTSP	0.480	**0.803**	**0.882**	0.555	0.830	0.947

A low number of agents N_a is used to test whether algorithms can efficiently specialize to the best locations, while high N_a is used to test whether they can efficiently distribute resources between all locations, and finally intermediate values of N_a are used to test how they adapt. For the results tables, we have set the low N_a value to the exact number of agents needed to saturate all task chains with $T = 2$, the high N_a value to the exact number of agents needed to saturate all task chains, and we the intermediate value is the midpoint between these values.

Travel times are either taken from the set $\{1, 2, 5\}$ with relative frequencies $(0.5, 0.3, 0.2)$, or from the set $\{1, 10\}$ with relative frequencies $(0.5, 0.5)$. These are chosen to test whether the relative performance of selection policies changes with the viability of sub-optimal solutions increases or decrease. Note that the difference in total travel time between the best and second best task chains is 1 for $\{1, 2, 5\}$ $((1, 1) \leftrightarrow (1, 2))$, while it is 9 for $\{1, 10\}$ $((1, 1) \leftrightarrow (1, 10))$.

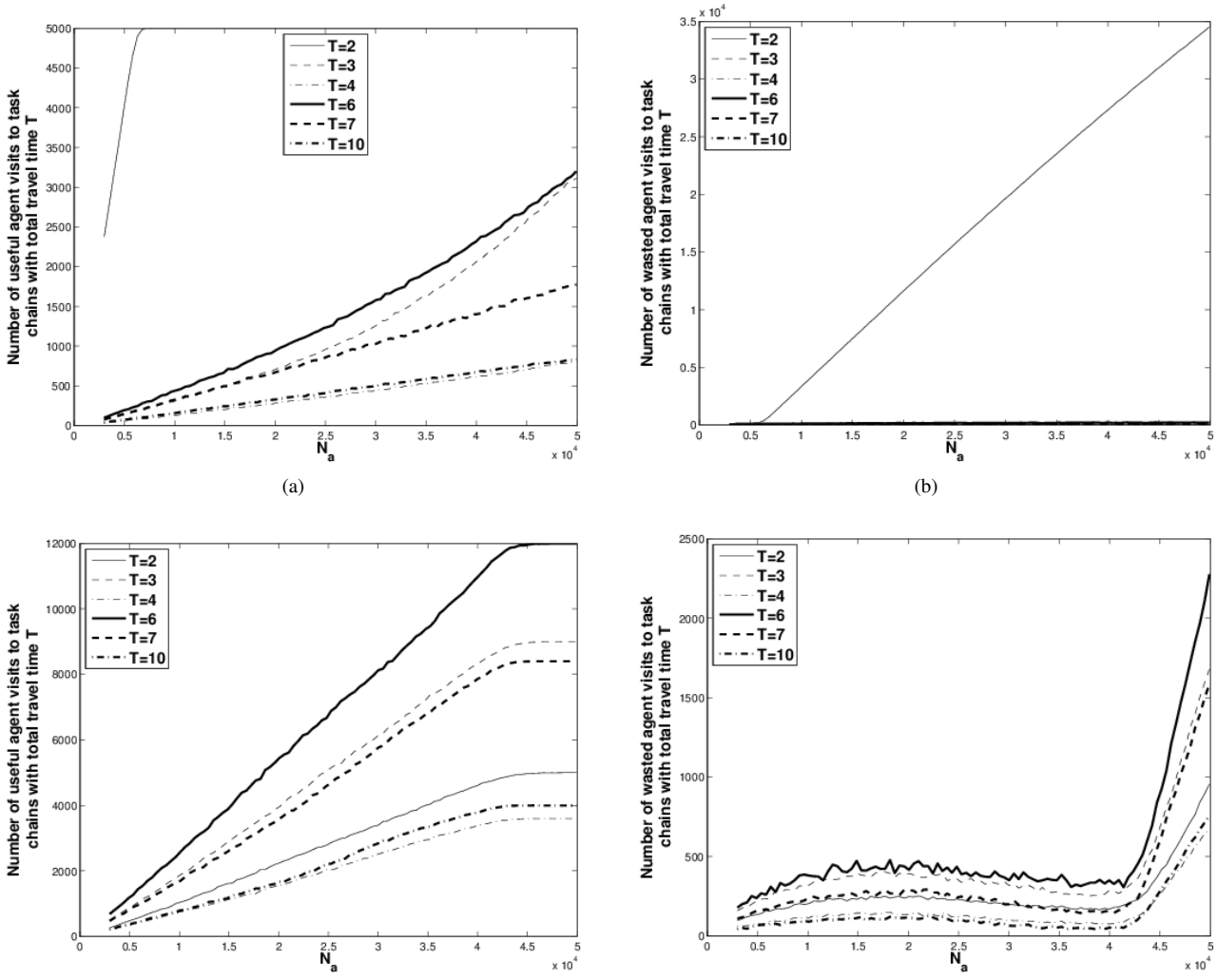
In tables II and III respectively, we show the performance (i.e. the throughput as a fraction of the maximum obtainable one) for the two different choices of travel time distributions respectively, for the various task selection policies and for the chosen combinations of C and N_a values. The best performance, in each environment is indicated in bold.

We note the following patterns as a function of N_a. Although at high N_a, RSP performs reasonably well (which can be understood because a random assignment of agents to task chains yields a number of agents proportional to the travel time, ideal at high levels), it doesn't quite reach the of performance of GCSP or indeed GSTSP. This is due to the fact that the agents at a given chain tend not to be balanced (collisions). At low N_a, GTSP tends to be the best, while STSP is at least competitive at low to intermediate values of N_a, but less so at high values because it does not take into account whether its work is useful. Finally, we note that GSTSP competitive (and often best) at all levels.



(a) (b)

Fig. 2. 2a Performance as a fraction of the maximum for each of the task selection policies as a function of N_a. 2b Optimum number of agents visiting task chains with total travel time T as a function of N_a



(a) (b)

(c) (d)

Fig. 3. The number of useful (3a and 3c) and failed (3b and 3d) visits to task chains with total travel time T as a function of N_a, for the task selection policies GTSP (3a and 3b), and GCSP (3c and 3d)

When varying the values of C we note that there is an obvious and easily understood overall performance increase for all policies with increasing C. At low values of C, GSTSP performs slightly better compared to the other policies. In general, however, the influence of C on the relative performance of the various policies is limited.

When varying the distribution of travel times, we observe that at low values of N_a the overall performance is better for $\{1, 2, 5\}$. this can be understood by the fact that not being at an ideal task chain is less of a dis-advantage. At other levels of N_a, performance is generally similar for the $\{1, 2, 5\}$ and $\{1, 10\}$ cases, with the only exception that GTSP does comparatively better for $\{1, 10\}$.

C. Problem Analysis

Since the behaviour at all parameter choices is qualitatively reasonably similar, we now take one setting as an example to investigate in depth the factors that influence performance for the various policies, at different values of N_a. We set $C = 100$, travel times are taken from $\{1, 2, 5\}$ with relative frequencies $(0.5, 0.3, 0.2)$ and we vary N_a starting from a slightly lower value (3000) than that in the tables to a slightly higher value (50000), to see whether the trends from the previous section continue.

For each algorithm, we measure three quantities: the **performance** as a fraction of the maximum, the number of **useful** task chain visits, and the number of **failed** task chain visits. We note that the number of useful and failed task chain visits are both needed as there are two main mechanisms for preformance loss. The first is to assign agents to chains with higher T before those with lower T are saturated, resulting in lower success rates than is optimal. The second mechanism is to over-saturate low T-chains, resulting in too many failed visits. Since we know the ideal distribution of agents to task chains, this is presented in figure 2b for reference, which is a snapshot of the system state at the end of the run. Any deviation from this (because agents go to chains with higher T too early) by practical algorithms is sub-optimal.

In Figure 2a, we note that the two greedy algorithms do well at the extremes where they are expected to do well, but quickly drop off away from that extreme. For intermediate values of N_a, STSP outperforms all other algorithms, except GSTSP which is significantly better and gives the best or at least competitive performance at all levels. We also note that the performance as fraction of the maximum is non-monotonic for STSP and GSTSP. When they are trying to saturate the same type of chain as the maximum, the maximum does so more efficiently and increases faster. However, when it has saturated all task chains of this type, it needs to switch to the next (higher T) while the other algorithms still have unsaturated low T-task chains, and can therefore temporarily improve faster.

From Figure 3, it is clear why GTSP fails for anything but low values of N_a as it only saturates $T = 2$ task chains. Higher T-chains remain under-saturated (figure 3a compared to the ideal in figure 2b), while $T = 2$ is massively over-saturated (see figure 3b). It also becomes clear why GCSP fails for low to intermediate values of N_a, as it does not have preference for low T-chains (the number of visits to each type of chain is proportional to its frequency multiplied by its travel time). At high N_a, however, this is ideal and high performance is obtained. Note that for GCSP significant over-saturation only happens for $N_a > 42000$, the minimum number of agents needed to saturate all chains.

In Figure 4 we observe that for low values of N_a STSP and GSTSP show similar behaviour to 2b (therefore good early performance), but they then start to deviate. STSP mainly loses performance due to over-saturation of low T-chains, as it does not consider the performance of other agents. GSTSP on the other hand, under-saturates intermediate T-chains (e.g. $T = 6$ and $T = 7$). We hypothesize that it has difficulty differentiating between them given its limited perspective. As was the case for GCSP, there is no significant over-saturation of chains until it becomes inevitable (i.e. $N_a > 42000$).

V. Conclusions

In this paper, we have extended the foraging problem (used in the literature to study sequential task allocation) to a *distributed* setting, such that it can be used to study *distributed* sequential task allocation. We have analysed the maximum performance for this problem theoretically, and have used this analysis to devise various policies for location selection by agents. We have tested the performance of these policies under a range of conditions. One particular policy, GSTSP, in which agents use local information to estimate both their own performance and their effect on the rest of the swarm, is shown to achieve good performance (compared to the theoretical maximum) under all conditions.

While our study considers the case of two sub-tasks with sequential dependencies, it is possible to extend this to include more complex tasks, with higher numbers of sub-tasks and both parallel and sequential dependencies. From a practical point of view, it would be useful to test whether the performance of our task selection policies (or similar policies following the guidelines from section III) would remain high under those conditions. In addition, it has been assumed that agents cannot communicate (in order to ensure that the algorithm is applicable to very limited agents) and that sources always have prey to harvest. Relaxing either of these assumptions could lead to interesting extensions of the problem. Finally, the main aim of this study was to maximise the performance using task chain selection; combining this with job selection could lead to further performance benefits.

References

[1] M. Dorigo and E. Şahin, "Guest editorial," *Autonomous Robots*, vol. 17, no. 2-3, pp. 111–113, 2004.

[2] G. Beni, "From swarm intelligence to swarm robotics," in *Swarm Robotics*, ser. Lecture Notes in Computer Science, E. Şahin and W. Spears, Eds. Springer Berlin Heidelberg, 2005, vol. 3342, pp. 1–9.

[3] E. ahin, "Swarm robotics: From sources of inspiration to domains of application," in *Swarm Robotics*, ser. Lecture Notes in Computer Science, E. Şahin and W. Spears, Eds. Springer Berlin Heidelberg, 2005, vol. 3342, pp. 10–20.

[4] K. Lerman, C. Jones, A. Galstyan, and J. Maja, "Analysis of dynamic task allocation in multi-robot systems," *International Journal of Robotics Research*, vol. 25, pp. 225–242, 2006.

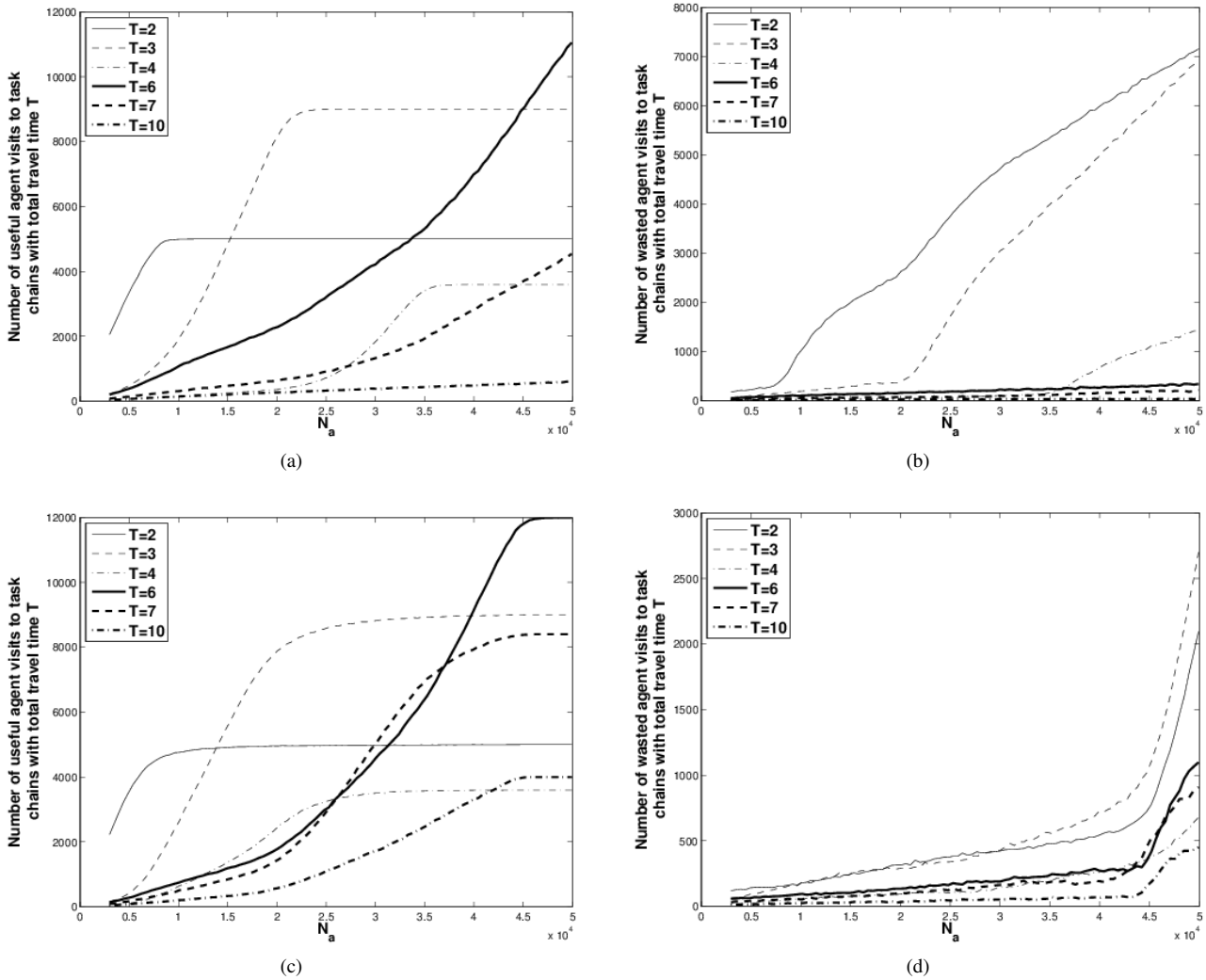


Fig. 4. Number of useful (4a and 4c) and failed (4b and 4d) visits to chains with total travel time T as a function of N_a, for task selection policies STSP (4a and 4b), and GSTSP (4c and 4d)

[5] E. Bonabeau, G. Theraulaz, and J.-L. Deneubourg, "Fixed response thresholds and the regulation of division of labor in insect societies," *Bulletin of Mathematical Biology*, vol. 60, no. 4, pp. 753–807, July 1998.

[6] G. Theraulaz, E. Bonabeau, and J.-L. Deneubourg, "Response threshold reinforcements and division of labour in insect societies," *Proceedings of the Royal Society B: Biological Sciences*, vol. 265, pp. 327–332, 1998.

[7] M. B. Dias, R. Zlot, N. Kalra, and A. Stentz, "Market-based multirobot coordination: A survey and analysis," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1257–1270, August 2006.

[8] S. Nouyan, R. Groß, M. Bonani, F. Mondada, and M. Dorigo, "Teamwork in self-organized robot colonies," *IEEE Transactions on Evolutionary Computation*, vol. 13, no. 4, pp. 695–711, 2009.

[9] G. Pini, M. Gagliolo, A. Brutschy, M. Dorigo, and M. Birattari, "Task Partitioning in a Robot Swarm : a Study on the Effect of Communication," *Swarm Intelligence*, no. February, 2013.

[10] G. Pini, A. Brutschy, M. Birattari, and M. Dorigo, "Task partitioning in swarms of robots: reducing performance losses due to interference at shared resources," *Informatics in Control Automation ...*, pp. 217–228, 2011.

[11] C. Parker and H. Zhang, "Collective unary decision-making by decentralized multiple-robot systems applied to the task-sequencing problem," *Swarm Intelligence*, vol. 4, no. 3, pp. 199–220, 2010.

[12] A. Brutschy, G. Pini, C. Pinciroli, M. Birattari, and M. Dorigo, "Self-organized task allocation to sequentially interdependent tasks in swarm robotics," Tech. Rep. TR/IRIDIA/2012-008, 2012.

[13] M. Frison, N.-L. Tran, N. Baiboun, A. Brutschy, G. Pini, A. Roli, M. Dorigo, and M. Birattari, "Self-organized task partitioning in a swarm of robots," in *Swarm Intelligence*, ser. Lecture Notes in Computer Science, M. Dorigo, M. Birattari, G. Caro, R. Doursat, A. Engelbrecht, D. Floreano, L. Gambardella, R. Gro, E. Şahin, H. Sayama, and T. Sttzle, Eds. Springer Berlin Heidelberg, 2010, vol. 6234, pp. 287–298.

[14] A. Brutschy, "Task allocation in swarm robotics. Towards a method for self-organized allocation to complex tasks," Tech. Rep. TR/IRIDIA/2009-007, 2009.

[15] G. Pini, A. Brutschy, M. Frison, A. Roli, M. Dorigo, and M. Birattari, "Task partitioning in swarms of robots: an adaptive method for strategy selection," *Swarm Intelligence*, vol. 5, no. 3-4, pp. 283–304, 2011.