Multi-A(ge)nt Graph Patrolling and Partitioning*

Yotam Elor and Alfred M. Bruckstein.
Faculty of Computer Science and the Goldstein UAV and Satellite Center.
Technion, Haifa 32000, Israel.

January 6, 2009

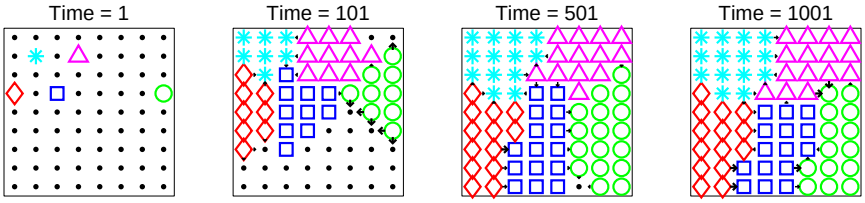
Abstract

We introduce a novel multi agent patrolling algorithm inspired by the behavior of gas filled balloons. Very low capability ant-like agents are considered with the task of patrolling an area modeled as a graph. While executing the proposed algorithm, the agents dynamically partition the graph between them using simple local interactions, every agent assuming the responsibility for patrolling his subgraph. Balanced graph partition is an emergent behavior due to the local interactions between the agents in the swarm. Extensive simulations on various graphs (environments) showed that the average time to reach a balanced partition is linear with the graph size. The simulations yielded a convincing argument for conjecturing that if the graph being patrolled contains a balanced partition, the agents will find it. However, we could not prove this. Nevertheless, we can prove that if a balanced partition is reached, the maximum time lag between two successive visits to any vertex using the proposed strategy is at most twice the optimal so the patrol quality is at least half the optimal.

1 Introduction

To patrol is to continuously travel through an area. The purpose of patrolling is to visit every point in the area as often as possible. "Informally, a good (patrol) strategy is one that minimizes the time lag between two passages to the same place and for all places" as described by [1]. It is convenient to model the area being patrolled as an undirected graph. In this model, time is discrete and will be measured in cycles. Using the undirected graph model we can loosely define the patrolling task as continuously visiting all the vertices of a graph. The *idleness* of a vertex is defined as the time since the last visit to this vertex by an agent. Several evaluation criteria for a patrol strategy such as *average idleness*, *worst idleness* and *exploration time* were defined by [1, 2].

*This research was supported by the Technion Goldstein UAV and Satellite Center.



(a) time = 0 (b) time = 100 (c) time = 500 (d) time = 1000

Figure 1: A typical run of BDFS on a grid with 5 agents. The different markers and colors represent areas controlled by different agents. After 100 time cycles the cyan asterisk agent is blocked, so he applies pressure over his neighbors and gradually increases its domain. A balanced configuration is reached within 1000 time cycles.

We present the Balloon DFS algorithm (BDFS) as a patrolling strategy. The algorithm addresses a simplified version of the problem in which all edge lengths are assumed equal. When k agents are patrolling a graph G using BDFS, they dynamically partition G into k connected subgraphs. Experiments show that in time the algorithm achieves a stable partition. We prove that the largest subgraph in any stable partition is of size at most $\frac{|G|}{k} + \frac{k}{2}$. Every agent patrols his subgraph using DFS hence the *worst idleness* is $2\frac{|G|}{k} + k$. In every time cycle every agent can visit at most one vertex so every multi-agent strategy (including the optimal) yields a *worst idleness* of at least $|G|/k$. Assuming $|G| \gg k$, the *worst idleness* of BDFS is about $2|G|/k$ which is at most twice the optimal.

Previously suggested partition based strategies work in two stages[3, 4]. In the first stage, the graph is partitioned between the agents and in the second the agents are sent to patrol the graph. The first stage complexity is high and requires some kind of "coordinator agent" who knows the graph in advance. In contrast, in our approach, the graph partitioning is executed "on the run" , there is no need to a-priory know the graph and there is no need for any preprocessing before starting the patrol. Furthermore, if the graph changes during the run, or some agents break down, the agents automatically repartition the graph without any external intervention. On the downside, BDFS cover time is large and reaching a good partition may take a long time.

2 Previous Work

When facing any multi agent task one must carefully address the assumed agents' capabilities. Obviously the strategy selected will be based on the assumed agent capabilities.

Given a graph and single agent to patrol it we can find a minimal length closed path which covers all vertices of the graph. Finding that kind of path is

the well known traveling sales person problem (TSP). An optimal single agent strategy would be to follow the best TSP circuit as suggested by [3]. Considering a multi agent scenario, we can make all agents follow the same cycle while maintaining a constant gap between them or alternately partition the graph into subgraphs, each subgraph being patrolled by an agent performing an optimal circuit in it.

The scenario where the agents can sense the *idleness* of all the vertices of the graph lead to several *heuristic* approaches proposed by [1, 2]. Two different assumptions were made: either each agent can sense only the vertices' *idleness* relative to it own visits or each agent sense the vertices' *idleness* with respect to all agents. Note that in the second case the *idleness* of the vertices serves as a communication method. Among other strategies considered, a straightforward one was to choose the shortest route to the vertex with the highest *idleness*. A distributed sensing variation proposed by [5] is to choose the vertex with the highest *idleness* only from the adjacent vertices as inspired by ants. Another distributed sensing variation is to mark the graph edges with pheromone marks (instead of the vertices) which leads to patrolling the graph using an Euler cycle as shown by [6].

When direct communication between the agents is allowed, the patrolling task becomes an agreement task. In order to reach an agreement, a "vertex-market" was constructed by [2]**.** Each agent owns some vertices and it is his responsibility to patrol his vertices.Agents can trade vertices in the market, for example an agent can trade a far vertex with a closer one and by that decrease its route length.

In our BDFS, agents dynamically partition the graph into almost equal subgraphs. The graph partitioning problem was widely studied and a survey of the work up to 1997 can be found in [7]. The most common variant of the problem is the k-cut problem. k-cut is an NP-hard problem which consists of finding a partition of a graph into k size balanced parts so that the sum of weights of all edges cut is minimized. Note that in the k-cut problem the partition subgraphs are not required to be connected. On the other hand, in our problem, we are not required to minimize the edges cut weight.

The bubble framework described by [8] resembles our algorithm. In the bubble scheme k vertices are chosen as seeds and the seeds expand BFS-style (or "diffuse") to create a k-partition of the graph. Afterward, k new seed vertices are chosen based on the resulting partition and the process is repeated until the seeds do not change their location. For a variant of the algorithm presented by [9] convergences has been proved, however with no analytic guarantees on the partition quality.

Among the many heuristic distributed algorithms that were considered for graph partitioning the ANTS algorithm proposed by [10] is based on the multi agent scheme in which some agents are working together to color a graph in k colors such that the weight of the cut between the colors is minimal.

3 High-Level description of the Algorithm and the Physical Intuition

Our algorithm is inspired from gas filled rubber balloons. Consider a space volume V with N balloons inside it. The balloons are all filled with the same quantity of gas and they are infinitely elastic. Given enough time, the system will converge to a balanced state in which the balloons fill the whole space V and the pressure inside the balloons is equal. Since the amount of gas in every balloon is the same, the balloons will have a volume of V/N each. The algorithm we propose mimics the balloons' behavior. Every agent controls a connected part of the graph. There are some border edges i.e. edges who connect vertices belonging to different agents. The agents sense those border edges and apply some "artificial pressure" over them - similarly to the pressure the gas molecules apply on the balloon surface. When the pressure applied by an agent is higher than the pressure applied from the other side, the agent can conquer the vertex on the other side. Since agents who control relatively small area hit the border edges more often than agents who control larger areas, they apply more pressure and the process described will hopefully converge to a balanced partition of the graph.

In physical balloon systems the space is continuous hence there is always a perfectly balanced partition of the space. In our system $|G|/k$ is generally not an integer so such partition does not always exist. The most balanced partition is a partition in which some agents control subgraphs of size $\lfloor |G|/k \rfloor$ and some of size $\lceil |G|/k \rceil$. It is desired that such a partition will be stable i.e. no more conquests will happen. In order to achieve this we set the rule that an agent can conquer a vertex from another agent only if the size difference of the agents' subgraphs is (strictly) larger than 1. As a result, every partition in which the size difference between every two adjacent subgraphs is at most 1 is stable. In a stable partition every subgraph is of size between $\frac{|G|}{k} - \frac{k}{2}$ and $\frac{|G|}{k} + \frac{k}{2}$ (see Lemma 3).

4 The Algorithm

4.1 Model

Ant-like agents having limited capabilities are considered here. The agents are assumed to have little memory (a finite number of registers with $O(log_2 n)$ bits where n is an upper bound on the graph size), they all execute the same algorithm and no direct communication is allowed between them. However, the agents can mark their neighborhood with pheromone stamps which can later be sensed by every agent. These markings are used as a primitive form of distributed memory and communication. The number of markings on every vertex or edge is fixed where the size of each marking is $O(log_2 n)$ bits.

For simplicity, we use a *strong synchronous* model. We assume that agents are operating in rounds. In every round the agents have slightly different timing

Algorithm 1: Multi Level DFS

```
   /* The agent is on vertex u                                          */
1  while ∃w ∈ N(u), | σ(uw) < SearchLevel do
       /* v is the next vertex to go to chosen by
          σ(uv) = min_{w∈N(u), | σ(uw)<SearchLevel}{σ(uw)}.             */
2      σ(uv) ← t
3      if σ(v) < SearchLevel then
4          σ(vu) ← t
5          σ(v) ← t
6          go to v (return)
7  if ∃v ∈ N(u), | σ(uv) = σ(u) and σ(u) > SearchLevel then
8      go to v (return)
9  else
10     SearchLevel ← t
11     σ(v) ← t
12     ∀w ∈ N(u),  σ(uw) ← σ(uw) − 1
```

phase so they never work simultaneously. These *strong synchronous* assumptions can be relaxed later. A related issue to consider is "what an agent can do in a single time step?". Since the problem under discussion is the patrol problem we have chosen to adopt the patrolling constrains: every passage between adjacent vertices takes one time cycle. Other operations like marking or sensing marks are assumed to be instantaneous.

4.2 Multi Level DFS

Depth First Search (DFS) can be used in order to travel a graph. A variant of DFS called Multi Level DFS (MLDFS) was introduced by [11]. MLDFS is carried out by using marks on the graph edges and vertices. An important property of the MLDFS is its noise robustness. If at some point the agent loses its backtracking path he just starts a new search. A variant of MLDFS is presented here as Algorithm 1. In this variant, each vertex or edge can be marked with a time stamp $\sigma(v)$ and $\sigma(uv)$ accordingly. Each edge has two markings, one from each side i.e. $\sigma(uv)$ and $\sigma(vu)$ are not necessarily equal. A time cycle is lost every time a new search starts, hence our MLDFS cover time is $2|G| - 1$ as opposed to $2(|G| - 1)$ in a regular DFS.

4.3 The Balloon Mechanism

In BDFS every agent controls a connected subgraph and is responsible for patrolling it. We introduce the balloon mechanism which eventually balances the graph partition. The balloon mechanism requires an additional mark on each edge: the pressure mark. The *pressure* on an edge is a *vector mark* i.e. a mark

Algorithm 2: Balloon DFS

```
   /* the agent is on vertex u                                          */
 1 while exist an unvisited border edge uv do
 2     mark uv as visited
 3     if P(uv) > 0 then
 4         with probability r conquer vertex v and go to v. (return)
 5     else
 6         if AreaSize = −P(uv) − 1 then
 7             P(uv) ← 0
 8         else
 9             P(uv) ← AreaSize
10 while exist an unvisited non-border edge uv do
11     mark uv as visited
12     if v is unvisited then
13         go to v (return)
14 if there is a backtrack edge then
15     backtrack (return)
16 else
17     start new BDFS (return)
```

which gives opposite readings when being sensed from different sides of the edge. For example, consider the edge uv. Assuming $P(uv) = p$ i.e. the pressure of edge uv when sensed from vertex u is p; then $P(vu) = -p$ i.e. the pressure when sensed from vertex v is $-p$.

A *border edge* is an edge connecting two vertices belonging to different agents. Every time an agent enters a vertex for the first time in the current search, it senses the pressure of all border edges emanating from the vertex. Assuming the agent is on vertex u, let uv be a border edge and denote the sizes of the subgraphs containing u and v by $|G_u|$ and $|G_v|$ respectively. Upon sensing the pressure of edge uv the agent acts as follows:

1. $P(uv) \leq 0$ and $|G_u| \neq -P(uv) - 1$. In this case the agent announces his area size by setting $P(uv) \leftarrow |G_u|$.

2. $P(uv) \leq 0$ and $|G_u| = -P(uv) - 1$ implies with high probability that $|G_u| = |G_v| - 1$ so the agent will set $P(uv) \leftarrow 0$.

3. $P(uv) > 0$ implies with high probability that $|G_u| \leq |G_v| - 2$ so the agent will conquer vertex v with probability r.

We can show that most of the time the agent knows what his subgraph size is (see Lemma 1). A high-level pseudo code of BDFS can be found in Algorithm 2.

5 Analysis

The proposed algorithm behavior is analyzed by defining system configurations and deriving the transition probabilities between them. A full description of the system includes the areas controlled by agents, the agents locations, the agents routes and the pressure over border edges. Instead of using the full description, we will use a partial one including only the areas controlled by agents. Formally, a system configuration c is a mapping of vertices to markings, $c: v \in V \rightarrow \{id_0, ..., id_k\}$.

5.1 Observations

We first introduce some notations. The system comprised k agents patrolling undirected graph G with $|G|$ vertices. $G_u(t)$ is the subgraph controlled by agent u at time t and G_u^c is the subgraph controlled by agent u in configuration c. This notation implies that vertex $u \in G_u$. G_u is a connected subgraph of G. All the vertices of G_u are marked by the pheromone mark of agent u denoted by id_u. However it may happen during the execution of the algorithm that some vertices are marked by id_u but do not belong to G_u, after a *balloon explosion* which is discussed in section 6. Time periods are denoted by $[T_0, T_1]$ where $t \in [T_0, T_1]$ implies that $T_0 \leq t < T_1$. We will say that G_u is *stable* in a time period $[T_0, T_1]$ if for every two time cycles $t_0, t_1 \in [T_0, T_1]$, $G_u(t_0) = G_u(t_1)$. A configuration is *stable* in a time period if all the subgraphs are *stable* in that time period. It is convenient to define $\Delta u \triangleq 2|G_u| - 1$ which is agent u's route length. The following lemma describes the agent behavior when patrolling his domain.

Lemma 1. *When agent u patrols G_u which is* stable *in $[T_0, T_1]$:*

1. *Agent u will start a new DFS search at time $t_0 \in [T_0, T_0 + \Delta u]$.*
2. *In $[t_0, T_1]$, agent u will perform sequential DFS searches using exactly the same route according to the MLDFS algorithm.*

Proof. Assume on the contrary that in the time period $[T_0, T_0 + \Delta u]$ agent u does not start a new DFS search. However, the current DFS search can take at most Δu time steps - a contradiction (proving 1). From the beginning of this DFS search G_u is stable, according to the algorithm the agent will perform exactly the same route in every search as long as G_u is stable (proving 2). □

In order to calculate the transition probability between configurations, we derive the conquest probability over a border edge. We will abbreviate the sentence "agent u applies pressure on all border edges emanating from vertex u" to "u-press". According to the algorithm, after a u-press $P(uv) \geq 0$ and after a v-press $P(uv) \leq 0$. A conquest is possible when upon pressing a *border edge* the pressure is strictly positive. Consider a border edge uv. Upon a u-press, if the last press of edge uv was a v-press then $P(uv) \leq 0$ and agent u can not

conquer vertex v. Denoting the event that agent u presses an edge uv and the last press of the edge was a u-press by "double u press". So a double press is a necessary condition for a conquest. However it is not sufficient as the next lemma shows.

Lemma 2. *Given a system in configuration c which is stable in $[T_0, T_1]$ and a border edge uv where $|G_u^c| \leq |G_v^c|$ then the probabilities of a conquest in any time period $[T, T + \Delta u \Delta v] \subseteq [T_0 + \Delta v, T_1]$ are:*

$$p\left(\textit{agent } u \textit{ conquers vertex } v\right) = \tag{1}$$
$$\begin{cases} \left(\Delta v - \Delta u\right) r & |G_u| \leq |G_v| - 2 \\ 0 & else \end{cases}$$
$$p\left(\textit{agent } v \textit{ conquers vertex } u\right) = 0 \tag{2}$$

Proof. Observe any time period T' of length $\Delta u \Delta v$ i.e. $T' = [T, T + \Delta u \Delta v] \subseteq [T + \Delta v, T_1]$. Consider any vertex w, the agent who controls vertex w applies pressure over all border edges emanating from w every first visit to w in each DFS search. G_u and G_v are *stable* in T' so according to Lemma 1, there will be Δv u-presses and Δu v-presses in T'. Hence there are $\Delta v - \Delta u$ double u-presses and there are not any double v-presses in T'. Since a double v-press is a necessary condition for a conquest by agent v, equation 2 is proved.

In order to prove equation 1 we are considering three cases:

1. If $|G_u| = |G_v|$ there are no double presses so the conquest probability is zero.

2. If $|G_u| = |G_v| - 1$ there are $\Delta v - \Delta u$ double u-presses. Consider a double u-press composed of the first and second u-presses. The press on edge uv preceding the first u-press is a v-press. So right before the first u-press $P\left(uv\right) = -\left|G_v\right|$ and as a result $|G_u| = -P\left(uv\right) - 1$. According to the algorithm, agent u will set $P\left(uv\right) \leftarrow 0$ so at the second u-press the pressure is not strictly positive and the conquest probability is zero.

3. If $|G_u| \leq |G_v| - 2$ there are $\Delta v - \Delta u$ double u-presses. Consider a double u-press composed of the first and second u-presses. Whether the last press on edge uv preceding the first u-press is a v-press or a u-press, right before the first u-press $|G_u| < -P\left(uv\right) - 1$ so agent u will set $P\left(uv\right) \leftarrow |G_u|$. According to the algorithm, in the second u-press $P\left(uv\right) > 0$ so the conquest probability is r. In the time period T' there are $\Delta v - \Delta u$ double u presses hence the total conquest probability in this time period is $\left(\Delta v - \Delta u\right) r$ for small r.

□

The time period $T_0' = [T_0, T_0 + \Delta v]$ is right after G_u or G_v have changed so it is possible that one of the agents has not yet found a stable DFS route. However, there is at most one v-press in that time period and if G_u and G_v are

about the same size there are at most two u-presses. So the conquest probability in this time period can be bounded from above by:

$$\begin{aligned} p\,(\text{agent u conquers vertex v in} T_0') &\leq \quad 2r \\ p\,(\text{agent v conquers vertex u in} T_0') &\leq \quad r \end{aligned} \tag{3}$$

Since r is small we will neglect the probability of a conquest in this time period. We will later refer to a conquest in this time period as an unjustified conquest since it can happen whether $|G_u| \leq |G_v| - 2$ or not.

5.2 Balanced Configurations

A balanced configuration is a configuration in which for all border edges uv, $||G_u| - |G_v|| \leq 1$ and all vertices are "controlled" by agents. The next lemma shows that the size of any subgraph in a balanced configuration is bounded. Both bounds are proved by induction on the number of agents.

Lemma 3. *For any balanced configuration b, the size of every subgraph G_i^b is bounded by $\frac{|G|}{k} - \frac{k}{2} \leq |G_i^b| \leq \frac{|G|}{k} + \frac{k}{2}$.*

Proof. We prove the upper bound by induction on k. The induction base is $k = 1$, $|G_1^b| = |G| \leq \frac{|G|}{1} + \frac{1}{2}$. Assuming the upper bound holds when the number of agents is at most k, we will show that it holds for $k+1$ agents. Consider a system composed of the graph G and $k+1$ agents patrolling it. Assume on the contrary that in a stable configuration b there is a subgraph with more than $\frac{|G|}{k+1} + \frac{k+1}{2}$ vertices. Without losing generality assume that $|G_1^b| > \frac{|G|}{k+1} + \frac{k+1}{2}$. The graph $G' \triangleq G - G_1^b$ is patrolled by k agents and is of size $|G'| = |G| - |G_1^b| < |G| - \frac{|G|}{k+1} - \frac{k+1}{2}$. Denote by g the component of G' for which the ratio between the size of the component to the number of agents patrolling it is the smallest. Let the number of agents patrolling g be k'. g is patrolled by $k' \leq k$ agents so by the induction hypothesis, in a stable configuration the size of each of the k' subgraphs of g is bounded by $|g_i| \leq \frac{|g|}{k'} + \frac{k'}{2}$. We have chosen g such as the ratio $|g|/k'$ is the smallest so we can write $|g_i| \leq \frac{|g|}{k'} + \frac{k'}{2} \leq \frac{|G'|}{k} + \frac{k}{2}$. Since G is connected there is a border edge connecting G_1^b to a subgraph of g. According to the stable configuration definition $|g_i| \geq |G_1^b| - 1$. Using the above inequalities we get a contradiction.

$$\begin{aligned} |g_i| &\geq |G_1^b| - 1 > \frac{|G|}{k+1} + \frac{k+1}{2} - 1 \\ &= \frac{2|G|k + k^3 - k}{2k(k+1)} \geq \frac{2|G|k + k^3 - 2k - 1}{2k(k+1)} \\ &= \frac{|G| - \frac{|G|}{k+1} - \frac{k+1}{2}}{k} + \frac{k}{2} > \frac{|G'|}{k} + \frac{k}{2} \geq |g_i| \end{aligned}$$

The lower bound is proved using similar induction.

□

After defining a balanced configuration we must ask "which graphs enable a balanced configurations?"[1]. To the best of our knowledge this variation of the graph partition problem have never been asked. A simple sufficient condition is the existence of a Hamilton path i.e. every graph that contains a Hamilton path contains a balanced configuration for any k. This simple condition implies that grids contain a balanced configuration.

5.3 The Convergence Conjecture

We believe that if balanced configurations exist, the system will eventually converge to one of them. However, we could neither prove this nor find a counter example. One way to prove convergence is to show the following: from every configuration there is a probability $p \geq \epsilon > 0$ that the system will reach a balanced configuration within a finite time (Conjecture 1) and after reaching a balanced configuration the system will remain in the balanced configuration indefinitely (Lemma 4). While the later assertion can be proved, we could not prove the first. Conjecture 1 is hard to prove because system transitions that lead to balanced configurations may be highly complex as shown in the next section.

Conjecture 1. *If a balanced configuration exists, from every configuration there is a probability $p \geq \epsilon > 0$ that the system will reach a balanced configuration within a finite time.*

Lemma 4. *When the system is in a balanced configuration there is a probability $p \geq \epsilon > 0$ that the system will remain in that configuration indefinitely.*

Proof. Consider a system in a balanced configuration at time $t = 0$. Let $\Delta \triangleq max_{u \in G} \{\Delta u\}$. There is a strictly positive probability that there will be no conquest in the time period $[0, \Delta]$ (see Equation 3). Assuming that in $[0, \Delta]$ no conquests occur, in the time period $[\Delta, \infty]$ all conquest probabilities are zero (by Lemma 2). So the system will remain in the balanced configuration indefinitely. □

6 Discussion and Simulations

The quality of the partition can be measured using an "entropy" defined as:

$$E(c) = \begin{cases} -\sum |G_i^c| \cdot log|G_i^c| & \bigcup G_i^c = G \\ -\infty & else \end{cases}$$

The entropy is finite only if the subgraphs controlled by the agents cover the graph. In our case, a more balanced partition yields higher entropy. The algorithm can be classified as a local search algorithm. Using vertex conquests

[1] The exact question we ask is "which graphs can be partitioned into k connected subgraphs such as the size difference between any two adjacent subgraphs is at most 1?"

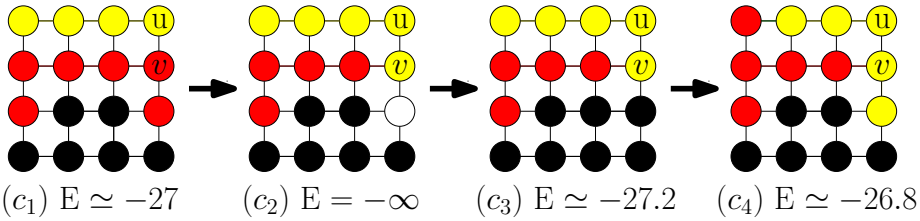


Figure 2: An example of a deadlock on a 4×4 grid resolved by a *balloon explosion*.

the agents continuously change the graph partition toward configurations with higher entropy. Under the heavy restrictions of the lack of global information and the low communication and memory abilities of the agents a local search algorithm is a natural choice.

Let uv be a border edge in configuration c where $u \in G_u^c$, $v \in G_v^c$, $\bigcup G_i^c = G$ and $|G_u^c| \leq |G_v^c| - 2$. Assuming agent u conquers vertex v, let c' be the resulting configuration with $G_u^{c'}$ and $G_v^{c'}$ as the resulting subgraphs. If v is not a cut vertex of G_v^c then the conquest yields a better partition and higher entropy. However, if v is a cut vertex of G_v^c then $G_v^{c'}$ is the component of $G_v^c \backslash v$ the agent happens to be in during the conquest. We call this event a *balloon explosion*. Right after a *balloon explosion* there are some vertices controlled by no one so $E(c') = -\infty$. These empty vertices will shortly be conquered and the entropy will be finite again. However, the resulting entropy might be lower then the entropy prior to the uv conquest. Note that *Balloon explosions* are due to the discrete space we work in and would not occur in the physical balloons model described in section 3.

Since any conquest which is a not a *balloon explosion* yields a better partition, *balloon explosions* might be considered destructive. It would be possible to change the algorithm so no *balloon explosions* will occur. However, *balloon explosions* are essential in order to escape deadlocks. An example for such a deadlock can be found in Figure 2. Several sequential configurations of a system in which three agents patrol a 4×4 grid are described in the figure. The different colors of vertices represent subgraphs controlled by different agents. In configuration c_1, the red and the black agents are controlling equal size subgraphs hence the conquest probability between them is zero. Any conquest of the yellow agent will result in a *balloon explosion*. In case we limit the agents to perform only conquests which does not cause *balloon explosions* the system is in a deadlock and will never reach an optimal partition. A possible scenario leading to a better partition, when *balloon explosions* are allowed, is described in the figure. Note that the entropy of configuration c_3, which is after the vertices left empty by the *balloon explosion* were recaptured, is lower than the entropy prior to the *balloon explosion*. So c_3 is less balanced than c_1, however c_3 is better configuration since its closer to a stable one as can be seen in c_4. So sometimes less balanced configurations are actually better than more balanced

ones. That phenomena makes proving conjecture 1 hard since we could not "measure" which configurations are closer to being balanced than others.

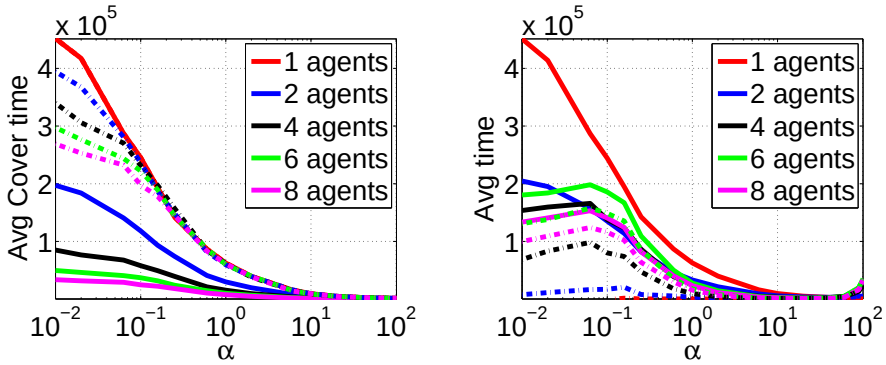
In the experiments we have measured the algorithm convergence time and cover time. The convergence time is the time to reach a balanced configuration and the cover time is the time needed to cover the graph i.e. to reach a finite entropy for the first time. The experiments were done over grids and random Hamiltonian graphs. As mentioned before, according to Conjecture 1, the algorithm always converges to a balanced configuration on such graphs. A total of 42 grids were tested with size ranges between 4×5 to 15×16. A random Hamiltonian graph $H_n(p)$ is created by first constructing an n-cycle $v_1 v_2 \dots v_n$ and then drawing additional edges at random with probability p for each possible edge. n is the number of vertices and the expected number of edges is $n + p\frac{n(n-3)}{2}$. The algorithm convergence time is influenced by the number of chords in the Hamiltonian cycle. By setting $p \triangleq \alpha/n$ the number of chords is $\simeq \alpha n/2$ i.e. proportional to n and α. In order to avoid graphs where n/k is an integer, a primal number of vertices was chosen. The cover and convergence time of every experiment configuration was averaged over 500 runs. The algorithm converged in all our experiments - supporting the convergence conjecture. A typical run on a grid is presented in Figure 1.

Before performing simulations, the value of r (the conquest probability) must be set. r should be small enough to keep the probability of unjustified conquests low (see equation 3). On the other hand, choosing r too small will increase the convergence time because conquests will become rare. In our experiments $r = 0.01$ worked well.

The convergence process can be divided into two time periods: The cover period starts when the algorithm starts and ends when the graph is covered. The stabilization period starts when the graph is covered and ends when a stable configuration is reached.

Simulation results on random Hamiltonian graphs are presented in Figures 3 and 4. Observe Figure 3(a) for the average cover time as a function of the graph density. The solid lines represent the cover time and the dashed lines are the normalized cover time defined as the cover time multiplied by the number of agents. The cover time is decreasing with edge density. It is clear since the denser the graph there are more border edges and more conquest opportunities. For dense enough graphs the expansion rate of the area controlled by the agents is proportional to the number of agents. So we would expect the cover time to be proportional to $1/k$. The simulations revealed that for $\alpha > 10^{-0.8}$ our expectation is correct as the normalized cover times overlap. For sparse graphs, the cover efficiency of a single agent decreases when more agents are added because the agents are stepping on each other toes. So for $\alpha < 10^{-0.8}$ the normalized cover time is smaller when there are more agents. However, even for sparse graphs, the actual cover time decreases when more agents are added to the system.

The average convergence and stabilization times as a function of the graph density are presented in Figure 3(b). We expected the convergence time to



(a) The solid lines represent the cover time and the dashed lines are the normalized cover time.

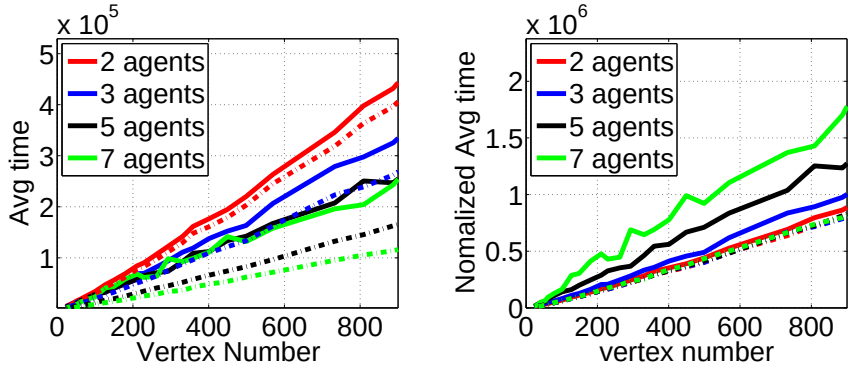
(b) The solid lines represent the convergence time and the dashed lines are the stabilization time.

Figure 3: Simulation results on random Hamiltonian graphs with varying edge-density and 101 vertices.

decrease when more edges are added to the graph since there will be more conquest opportunities and more balanced configurations. Simulations revealed that our expectation holds but for the two extremes of very sparse ($\alpha < 0.1$) and very dense ($\alpha > 50$) graphs. In the sparse extreme, the convergence time of graphs with $\alpha \simeq 0.1$ is larger than the convergence time of sparser graphs even though the cover time is smaller. The graphs with $\alpha < 0.1$ are rings with few chords so after they are covered there will be almost no *balloon explosions* delaying the stabilization. When α is increased more chords are added to the graph. The chords help to decrease the cover time but on the other hand they allow more *balloon explosions* during the stabilization phase resulting in longer stabilization period and higher convergence time. On the other extreme, in very dense graphs there are many border edges so the probabilities of unjustified conquests can not be neglected. Hence unjustified conquests occur, and have a negative effect on the stabilization time.

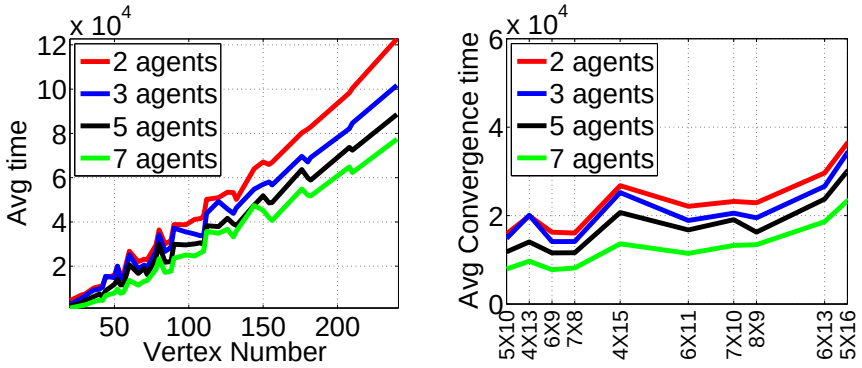
Observe Figure 4(a) for the average convergence and cover times on random Hamiltonian graphs with $\alpha = 1$. The convergence and cover times are linear with the graph size. As expected, they are decreasing when more agents are added to the system. The convergence and cover time divided by the number of agents are presented in Figure 4(b). The normalized cover times overlap so the average cover time is $\Theta(n/k)$. However, the convergence time does not decrease linearly with k. When more agents are added the system, the stabilization period is getting longer hence enlarging the convergence time. Formally, the convergence time is $\Theta(n/f(k))$ where f is a monotonic non-decreasing function.

Observe Figure 5(a) for simulation results on grids. The convergence time was found to be linear with the number of vertices. However some grids are more challenging than others. Enlargement of the graph for grids of sizes $50-80$



(a) (b)

Figure 4: Simulation results on random Hamiltonian graphs with varying size. The solid lines represent the convergence time and the dashed lines are the cover time.



(a) (b) The labels are the grids dimensions.

Figure 5: Average convergence time on Grids.

vertices is presented in Figure 5(b). Note that the convergence time for a 4×15 grid is larger than for a 6×11 grid even though the second grid contains more vertices. The reason is that the second grid is more "square" so the algorithm is less limited by edge effects.

7 Conclusion

A novel graph patrolling algorithm was introduced. A group of k agents performing the algorithm partition the graph between them using simple local interactions. Graph partition into connected and size balanced components is an emergent behavior of the agent swarm. We have shown that in a balanced partition every subgraph is of size less than $\frac{|G|}{k} + \frac{k}{2}$. As a result, the time lag between two successive visits to any vertex is at most twice the optimal so the patrol quality is at least half the optimal. Simulations performed over grids and random Hamiltonian graphs showed that the average cover time is $\Theta\left(|G|/k\right)$. The average time to reach a balanced partition was experimentally found to be $\Theta\left(|G|/f\left(k\right)\right)$ where f is a monotonic non-decreasing function.

References

[1] Aydano Machado, Geber Ramalho, Jean-Daniel Zucker, and Alexis Drogoul. Multi-agent patrolling: An empirical analysis of alternative architectures. In *MABS*, pages 155–170, 2002.

[2] Alessandro Almeida, Geber Ramalho, Hugo Santana, Patricia Azevedo Tedesco, Talita Menezes, Vincent Corruble, and Yann Chevaleyre. Recent advances on multi-agent patrolling. In *SBIA*, pages 474–483, 2004.

[3] Yann Chevaleyre, Francois Sempe, and Geber Ramalho. A theoretical analysis of multi-agent patrolling strategies. In *AAMAS*, pages 1524–1525, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Tiago Sak, Jacques Wainer, and Siome Klein Goldenstein. Probabilistic multiagent patrolling. In *SBIA*, pages 124–133, 2008.

[5] Israel A. Wagner, Michael Lindenbaum, and Alfred M. Bruckstein. Efficiently searching a graph by a smell-oriented vertex process. *Annals of Mathematics and Artificial Intelligence*, 24(1-4):211–223, 1998.

[6] Vladimir Yanovski, Israel A. Wagner, and Alfred M. Bruckstein. A distributed ant algorithm for efficiently patrolling a network. *Algorithmica*, 37(3):165–186, 2003.

[7] Ulrich Elsner. Graph partitioning - a survey, 1997.

[8] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive fem. *Parallel Comput.*, 26(12):1555–1581, 2000.

[9] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. In *IPDPS*. IEEE Computer Society, 2008.

[10] Francesc Comellas and Emili Sapena. A multiagent algorithm for graph partitioning. In *EvoWorkshops*, volume 3907 of *Lecture Notes in Computer Science*, pages 279–285. Springer, 2006.

[11] I.A. Wagner, M. Lindenbaum, and A.M. Bruckstein. Distributed covering by ant-robots using evaporating traces. *Robotics and Automation, IEEE Transactions on*, 15(5):918–933, Oct 1999.

Algorithm 3: BDFS - detailed

```
   // the agent is on vertex u
 1 if u is not marked with AgentId then
 2     go to random neighbor vertex marked by AgentId. (return)
explore border edges
 3 while ∃v ∈ N(u), | σ(u,v) < SearchLevel and C(v) ≠ AgentId do
 4     σ(u,v) ← t
 5     if P(u,v) > 0 then
conquer vertex
 6         if x < r then
               // x is a random variable chosen uniformly from [0,1]
 7             ∀w ∈ N(v), σ(v,w) ← t+1, P(v,w) ← 0
 8             σ(v,u) ← t
 9             σ(v) ← t
10             C(v) ← AgentId
11             go to v (return)
12     else
apply pressure
13         if P(u,v) + AreaSize = −1 then
14             P(u,v) ← 0
15         else
16             P(u,v) ← AreaSize
explore non-border edges
17 while ∃v ∈ N(u), | σ(u,v) < SearchLevel and C(v) = AgentId do
       // v is the next vertex to go to chosen by
           σ(u,v) = min_{w∈N(u), | σ(u,w)<SearchLevel}{σ(u,w)}.
18     σ(u,v) ← t
19     if σ(v) < SearchLevel then
go to new vertex
20         AreaSize ← (t − σ(v) + 1)/2
21         σ(v,u) ← t
22         σ(v) ← t
23         go to v (return)
24 if ∃v ∈ N(u), | σ(u,v) = σ(u) and C(v) = AgentId and
   σ(u) > SearchLevel then
backtrack
25     go to v (return)
26 else
Start new BDFS
27     SearchLevel ← t
28     σ(v) ← t
29     ∀w ∈ N(u),  σ(u,w) ← σ(u,w) − 1
30     return
```