Distributed Transformations of Hamiltonian Shapes based on Line Moves☆,☆☆

Abdullah Almethen[a,1,*], Othon Michail[b], Igor Potapov[b]

[a] *Unit of Scientific Research, Applied College, Qassim University, Buraydah, Saudi Arabia*
[b] *Department of Computer Science, University of Liverpool, Liverpool, UK*

Abstract

We consider a discrete system of n simple indistinguishable devices, called *agents*, forming a *connected* shape S_I on a two-dimensional square grid. Agents are equipped with a linear-strength mechanism, called a *line move*, by which an agent can push a whole line of consecutive agents in one of the four cardinal directions in a single time-step. We study the problem of transforming an initial shape S_I into a given target shape S_F via a finite sequence of line moves in a distributed model, where each agent can observe the states of nearby agents in a Moore neighbourhood. We develop the first distributed connectivity-preserving transformation that exploits line moves. The transformation solves the *line formation problem*. That is, starting from any shape S_I whose *associated graph* contains a Hamiltonian path known to them, the agents can form a final straight line S_L. The complexity of the transformation is $O(n \log_2 n)$ moves, which is asymptotically equivalent to that of the best-known centralised transformations.

Keywords:
Line movement, Discrete transformations, Shape formation, Reconfigurable robotics, Programmable matter, Distributed algorithms

1. Introduction

The explosive growth of advanced technology over the last few decades has contributed significantly towards the development of a wide variety of distributed systems consisting of large collections of tiny robotic-units, which we here call *entities* or *agents* (also known as *modules* or *monads*). These entities are able to move and communicate with each other by being equipped with microcontrollers, actuators and sensors. However, each entity is severely restricted and has limited computational capabilities, such as a constant memory and lack of global knowledge. Further, entities are typically homogeneous, anonymous and indistinguishable from each other. Through a simple set of rules and local actions, they collectively act as a single unit and carry out several complex tasks, such as transformations and explorations.

In this context, scientists from different disciplines have made great efforts towards developing innovative, scalable and adaptive collective robotic systems. This vision has recently given rise to the area of programmable matter, first proposed by Toffoli and Margolus [45] in 1991, referring to any kind of materials that can algorithmically change their physical properties, such as shape, colour, density and conductivity

☆A preliminary version of the results in this paper has appeared in [4].

☆☆This work was supported by the University of Liverpool EEE/CS initiative NeST. The last author was also supported by the Leverhulme Trust Senior Research Fellowship (SRF\R1\201074)

*Corresponding author (Telephone number: +966 (0)16 380 0050, Postal Address: Unit of Scientific Research, Applied College, Qassim University, Buraydah, Saudi Arabia 52571, SA).

Email addresses: `Amthn@qu.edu.sa` (Abdullah Almethen), `Othon.Michail@liverpool.ac.uk` (Othon Michail), `Potapov@liverpool.ac.uk` (Igor Potapov)

[1] The research was partially conducted while the author was at the University of Liverpool.

through transformations executed by an underlying program. This newborn area has been of growing interest lately both from a theoretical and a practical viewpoint.

One can categorise programmable matter systems into *active* and *passive*. Entities in passive systems have no control over their movement. Instead, they move and interact due to the dynamics of the environment and based on their own structural characteristics. Prominent examples of research on passive systems appear in the areas of population protocols [8, 35, 36], DNA computing [1, 10] and tile self-assembly [19, 42, 48]. On the other hand, the active systems allow computational entities to act and control their movements in order to accomplish a given task, which is our primary focus in this work. The most popular examples of active systems include metamorphic systems [23, 38, 47], swarm/mobile robotics [13, 25, 40, 43, 51], modular self-reconfigurable robotics [6, 27, 53] and recent research on programmable matter [14, 16]. There is also increasing interest from the engineering research community, hence, various solutions and frameworks have been produced for systems ranging from milli/micro-scale [11, 28, 31] down to nano-scale [20, 41].

Shape transformations (sometimes called *pattern formation*) can be seen as one of the most essential goals for almost every system among the vast variety of robotic systems including programmable matter and swarm robotic systems. In this work, we focus on a system of entities operating on a two-dimensional grid. The entities are typically connected to each other, forming an initial connected shape S_I. Each entity is equipped with a linear-strength mechanism that can push an entire line of consecutive entities one position in a single time-step in a given direction of the grid. The goal is to design an algorithm that can transform an initial shape S_I into a given target shape S_F through a chain of permissible moves and without breaking connectivity. That is, we must guarantee that, in each intermediate configuration, the graph formed by the entities and the set of their horizontal, vertical and diagonal connections is a connected graph. Connectivity-preservation is an important property to aim at, as, among other benefits, it enables constant communication and energy flow and allows the system to maintain its unity by withstanding environmental forces.

1.1. Related Work

Many models of centralised or distributed coordination have been studied in the context of shape transformation problems. The assumed mechanisms in those models can significantly influence the efficiency and feasibility of shape transformations. For example, the authors of [2, 21, 22, 23, 34] consider mechanisms called sliding and rotation by which an agent can move and turn over neighbours through empty space. Under these models of individual movements, Dumitrescu and Pach [21] and Michail *et al.* [34] present universal transformations for any pair of connected shapes (S_I, S_F) of the same size. By restricting to rotation only, the authors in [34] proved that the decision problem of transformability is in $\mathbf{P}$. Recently, Connor *et al.* [12] presented a centralised connectivity-preserving transformation using rotation only. The transformation works for a class of shapes known as *nice shapes* (this class was first introduced in [3] for the line-pushing model), if triggered by a seed of as few as 4 additional entities.

The alternative less costly reconfiguration solutions can be designed by employing some parallelism,where multiple movements can occur at the same time; consult [14, 18] for theoretical studies and [43] for a more practical implementation. Moreover, it has been shown that there exists a universal transformation with rotation and sliding that converts any pair of connected shapes to each other within $O(n)$ parallel moves in the worst case [34]. Also, fast reconfiguration might be achieved by exploiting actuation mechanisms, where a single agent is now equipped with more strength to move many entities in parallel in a single time-step. A prominent example is the linear-strength model of Aloupis *et al.* [6, 7], where an entity is equipped with arms giving it the ability to extend/extract a neighbour, a set of individuals or the whole configuration in a single operation. Another elegant approach by Woods *et al.* [49] studied another linear-strength mechanism by which an entity can rotate a chain of other entities of arbitrary length, through an arm rotation operation.

A more recent study along this direction is [3], introducing the *line-pushing* model. In this model, an individual entity can push the whole line of consecutive entities one position in a given direction in a single time-step. As we shall explain, this model generalises some existing constant-strength models with a special focus on exploiting its parallel power for fast and more general transformations. Apart from the purely theoretical benefit of exploring fast reconfigurations, this model also provides a practical framework for more efficient reconfigurations in real systems. For example, self-organising robots could be reconfiguring into multiple shapes in order to pass through canals, bridges or corridors in a mine. In another domain, individual robots could be containers equipped with motors that can push an entire row to manage space in large warehouses. Another future application could be a system of very tiny particles injected into a human body and transforming into several shapes in order to efficiently traverse through the veins and capillaries and treat infected cells.

This model is capable of simulating some constant-strength models. For example, it can simulate the sliding and rotation model [21, 34] with an increase in the worst-case running time only by a factor of 2. This implies that all universality and reversibility properties of individual-move transformations still hold true in this model. A small technical difference to [21, 34] is that the line-pushing model allows diagonal connections on the grid, an assumption that we keep in this paper. Several sub-quadratic time centralised transformations have been proposed, including an $O(n\sqrt{n})$-time universal transformation that preserves the connectivity of the shape during its course [5]. By allowing transformations to disconnect the shape during their course, there exists a centralised universal transformation that completes within $O(n \log n)$ time.

Another recent related set of models studied in [13, 24, 29] consider a single robot which moves over a static shape consisting of tiles and the goal is for the robot to transform the shape by carrying one tile at a time. In those systems, the single robot which controls and carries out the transformation is typically modelled as a finite automaton. Those models can be viewed as partially centralised as on one hand they have a unique controller but on the other hand that controller is operating locally and suffering from a lack of global information.

1.2. Our Contribution

In this work, our main objective is to give the first distributed transformations for programmable matter systems implementing the linear-strength mechanism of the model of line moves. All existing transformations for this model are centralised, thus, even though they reveal the underlying transformation complexities, they are not directly applicable to real programmable matter systems. Our goal is to develop distributed transformations that, if possible, will preserve all the good properties of the corresponding centralised solutions. These include the *move complexity* (i.e. the total number of line moves) of the transformations and their ability to preserve the connectivity of the shape throughout their course.

Consider a discrete system of n simple indistinguishable devices, called *agents*, forming a connected shape S_I on a two-dimensional square grid (i.e. the graph induced by S_I is connected). Agents act as finite-state automata (i.e. they have constant memory) that can observe the states of nearby agents in a Moore neighbourhood (i.e. the eight cells surrounding an agent on the square gird). They operate in synchronised Look-Compute-Move (LCM) cycles on the grid. All communication is local and actuation is based on this local information as well as the agent's internal state.

Within this distributed setting of identical agents, breaking symmetry emerges as a fundamental issue, rendering many agreement problems in distributed computing systems impossible. In [50], this concept is formally defined as the *symmetricity* of a network. By concentrating on the shape formation problem, it is necessary for the agents to have some common agreement on a coordinate system [26, 44]. They may, for example, form any arbitrary target shape if they have a common sense of direction, unit distance and coordinate axes. Furthermore, the degree of synchronisation is another major issue in distributed computing in general, and it has a special impact on the feasibility of algorithms in graph-based robotic systems (see for example [15]). Thus, as a first attempt at distributing line moves on the two-dimensional square grid, we adopted a fully-synchronised model of agents that share a sense of orientation which will be discussed in more detail later.

Let us consider a very simple distributed transformation of a diagonal line shape S_D into a straight line S_L, $|S_D| = |S_L| = n$, in which all agents execute the same procedure in parallel synchronous rounds. In general, the diagonal appears to be a hard instance because any parallelism related to line moves that might potentially be exploited does not come for free. Initially, all agents are occupying the consecutive diagonal cells on the grid $(x_1, y_1), (x_1+1, y_1+1), \ldots, (x_1+n-1, y_1+n-1)$. In each round, an agent $p_i = (x, y)$ moves one step down if $(x-1, y-1)$ is occupied, otherwise it stays still in its current cell. After $O(n)$ rounds, all agents form S_L within a total number of $1 + 2 + \ldots + n = O(n^2)$ moves, while preserving connectivity during the transformation (throughout, connectivity includes horizontal, vertical, and diagonal adjacency). See Figure 1.

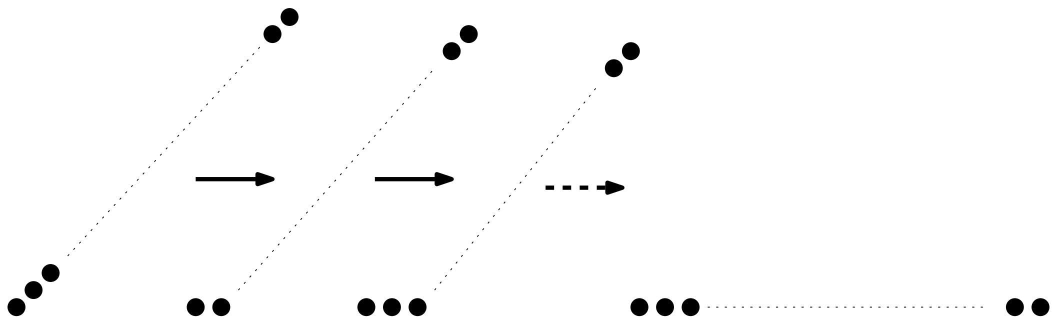


Figure 1: A simulation of the simple procedure. From left to right, rounds $0, 1, 2, \ldots, n$.

The above transformation, even though time-optimal has a move complexity asymptotically equal to the worst-case single-move distance between S_I and S_F. This is because it always moves individual agents, thus not exploiting the inherent parallelism of line moves. Our goal is to trade time for number of line moves in order to develop alternative distributed transformations which will complete within a sub-quadratic number of moves. Given that actuation is a major source of energy consumption in real programmable matter and robotic systems, moves minimisation is expected to contribute in the deployment and implementation of energy-efficient systems.

However, there are considerable technical challenges that one must deal with in order to develop such a distributed solution. As will become evident, the lack of global knowledge of the individual entities and the condition of preserving connectivity greatly complicate the transformation, even when restricted to special families of shapes. Timing is an essential issue as the line needs to know when to start/stop pushing. When moving or turning, all agents of the line must follow the same route, ensuring that no one is being pushed off. There is an additional difficulty due to the fact that agents do not automatically know whether they have been pushed (but it might be possible to infer this through communication and/or local observation).

We already know that there is a centralised $O(n \log n)$-move connectivity-preserving transformation, working for a large family of connected shapes [5]. That centralised strategy transforms a pair of connected shapes (S_I, S_F) of the same order (i.e. the number of agents) to each other, when the associated graphs of both shapes contain a Hamiltonian path (see also Itai *et al.* [30] for rectilinear Hamiltonian paths), while preserving connectivity during the transformation.

On a high level, the algorithm transforms a Hamiltonian shape into a straight line in $\log n$ phases as follows: In phase i, 2^i terminal agents forming a straight line L_i, representing the start or end of a Hamiltonian path, merge with the next 2^i agents which can be in any configuration S_i, in order to form a new straight line of length 2^{i+1}. Hence, the line length is doubled in each phase. On one end of the Hamiltonian path, 2^i agents that are forming the respective line L_i followed by S_i of 2^i agents, are identified. Afterwards a feasible transformation path is computed. The agents thereby push L_i from the prior phase (of length 2^i) to its destination at the far end point of S_i. Then, S_i is merged recursively with L_i in that phase, forming a new straight line L_{i+1} of 2^{i+1} agents. After $O(n \log n)$ moves, the algorithm forms the final straight line S_L of length n, which can be then transformed into S_F by reversing the transformation of S_F into S_L (this is possible in the centralised case), within the same asymptotic number of moves.

In this work, we introduce the first distributed transformation exploiting the linear-strength mechanism of the *line-pushing* model. It provides a solution to the line formation problem, that is, for any initial Hamiltonian shape S_I, forms a final straight line S_L of the same order. It is natural to commence with basic shape problems, e.g. the line, as a first stepping stone towards more general transformations. This is motivated by the principle that if a shape can be transformed into an intermediate shape (in our case a straight line), then any pair of shapes (with the same number of agents) can be transformed to each other (see for example [49] and [17]).

The proposed approach is essentially a distributed implementation of the centralised Hamiltonian transformation of [5] to form any Hamiltonian shape into a straight a line. We show that it preserves the asymptotic bound of $O(n \log n)$ line moves (which is still the best-known centralised bound), while keeping the whole shape connected throughout its course. This is the first step towards distributed transformations between any pair of Hamiltonian shapes. The inverse of this transformation (S_L into S_I) appears to be a much more complicated problem to solve as the agents need to somehow know an encoding of the shape to be constructed (i.e. each agent should initially be aware of its intended position on the target shape) and that in contrast to the centralised case, reversibility does not apply in a straightforward way. Hence, the reverse of this transformation (S_L into S_I) is left as a future research direction.

We restrict attention to the class of Hamiltonian shapes. There is an algorithm by Umans and Lenhart [46] for computing Hamiltonian paths and cycles in polynomial time for a large sub-class of *grid graphs*. In particular, the algorithm works for what they call *quad-quad graphs*, a class including all hole-free shapes defined on the 2D square grid. Note that the problem of computing Hamiltonian paths and cycles for general grid graphs has been shown to be NP-complete by Itai *et al.* [30]. The class of Hamiltonian shapes, apart from being a reasonable first step in the direction of distributed transformations in the given setting, might give insight to the future development of universal distributed transformations, i.e. distributed transformations working for any possible pair of initial and target shapes. Note that not only Hamiltonian paths can be computed efficiently for a large class of 2D shapes when they exist, but also connected shapes defined on a 2D grid tend to have long simple paths. For example, the length of their longest path is provably at least $\sqrt{n}$. We here focus on developing efficient distributed transformations for the extreme case in which the longest path is a Hamiltonian path. However, one might be able to apply our Hamiltonian transformation to any pair of shapes, by, for example, running a different or similar transformation along branches of the longest path and then running our transformation on the longest path. We leave how to exploit the longest path in the general case (i.e. when initial and target shapes are not necessarily Hamiltonian) as an interesting open problem.

We assume that a centralised pre-processing provides the Hamiltonian path. Note that this pre-processing is polynomial-time for all hole-free shapes (by using, e.g. the algorithm of [46]). At the same time, this provides the agents with a global sense of direction, through a labelling of their local ports (e.g. each agent maintains two local ports incident to its predecessor and successor on the path). Similar assumptions exist in the literature of systems of complex shapes that contain a vast number of self-organising and limited entities. A prominent example is [43] in which the transformation relies on an initial central phase to gain some information about the number of entities in the system.

Now, we are ready to sketch a high-level description of the transformation. A Hamiltonian path P in the initial shape S_I starts with a head on one endpoint labelled l_h (used as a pre-elected unique leader), which is leading the process and coordinating all the sub-procedures during the transformation. The transformation proceeds in $\log n$ phases, each consisting of six sub-phases (or sub-routines) and every sub-phase running for one or more synchronous rounds. Figure 2 gives an illustration of a phase of this transformation when applied on the diagonal line shape. Initially, the head l_h forms a trivial line of length 1. By the beginning of each phase i, $0 \leq i \leq \log n - 1$, there is a line L_i with $2^i - 2$ internal agents labelled l, running from the head l_h to the tail l_t. During phase i, both l_h and l_t are responsible for interchangeably pushing the line via the following sub-phases:

1. DefineSeg: Identify the next segment S_i of length 2^i in the Hamiltonian path.

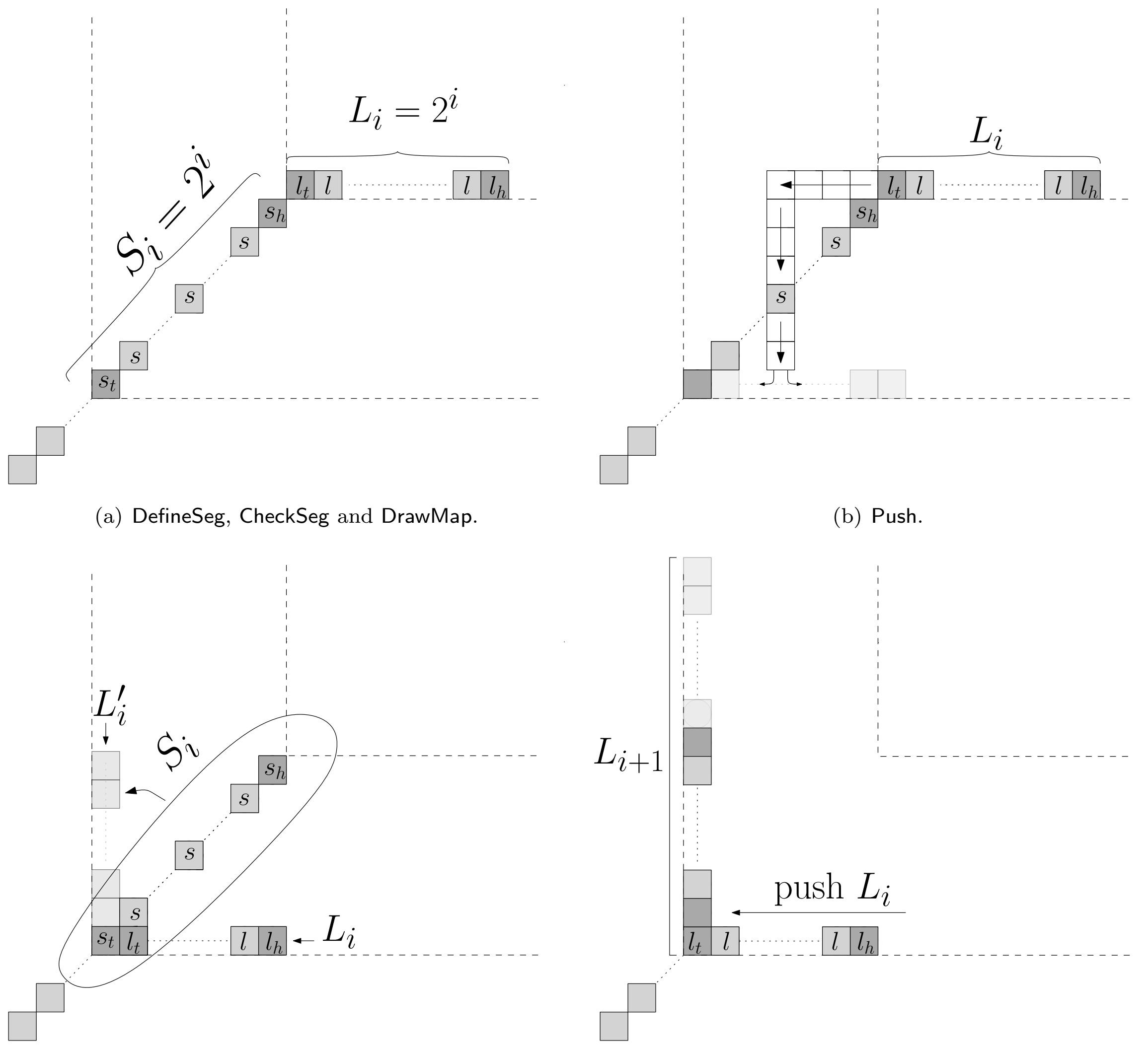
2. CheckSeg: Check whether S_i is in line or perpendicular line to L_i. Start phase $i+1$ if in line, go to (6) if perpendicular or go to (3) otherwise.

3. DrawMap: Compute a route map and store it in L_i's agents so that L_i can travel to the end of S_i.

4. Push: Move L_i along the drawn route map.

5. RecursiveCall: A recursive-call on S_i to transform it into a straight line L'_i.

6. Merge: Combine L_i and L'_i together into a straight line L_{i+1} of 2^{i+1} double length. Then, phase $i+1$ begins.



(a) DefineSeg, CheckSeg and DrawMap. (b) Push. (c) RecursiveCall. (d) Merge.

Figure 2: From [5], a snapshot of phase i of the Hamiltonian transformation on the shape of a diagonal line. Each occupied cell shows the current label state of an agent. Light grey cells show ending cells of the corresponding moves.

Section 2 formally defines the model and the problem under consideration. Section 3 presents our distributed connectivity-preserving transformation that solves the line formation problem for Hamiltonian shapes, achieving a total of $O(n \log n)$ line moves.

2. Model

We consider a system consisting of n agents forming a connected shape S on a two-dimensional square grid in which each agent $p \in S$ occupies a unique cell $cell(p) = (x, y)$, where x indicates the column index and y represents the row index. Throughout, an agent shall also be referred to by its coordinates. Each cell (x, y) is surrounded by eight adjacent cells in each cardinal and ordinal direction, (N, E, S, W, NE, *NW*, *SE*, *SW*). That is, our definitions hold for an infinite grid, yet a finite subset of it is sufficient for all transformations/properties presented. At any time, a cell (x, y) can be in one of two states, either empty or occupied. An agent $p \in S$ is a *neighbour* of (or *adjacent* to) another agent $p' \in S$, if p' occupies one of the eight adjacent cells surrounding p, that is their coordinates satisfy $p'_x - 1 \leq p_x \leq p'_x + 1$ and $p'_y - 1 \leq p_y \leq p'_y + 1$, see Figure 3. For any shape S, we associate a graph $G(S) = (V, E)$ defined as follows, where V represents agents of S and E contains all pairs of adjacent neighbours, i.e. $(p, p') \in E$ iff p and p' are neighbours in S. We say that a shape S is connected iff $G(S)$ is a connected graph. The *distance* between agents $p \in S$ and $p' \in S$ is defined as the Manhattan distance between their cells, $\Delta(p, p') = |p_x - p'_x| + |p_y - p'_y|$. A shape S is called *Hamiltonian shape* iff $G(S)$ contains a Hamiltonian path, i.e. a path starting from some $p \in S$, visiting every agent in S and ending at some $p' \in S$, where $p \neq p'$.

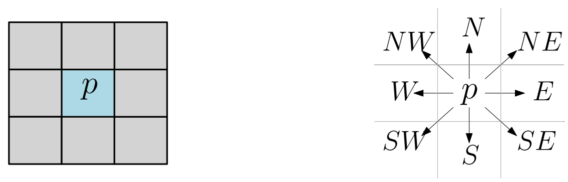


Figure 3: An agent p is a neighbour to any agent locating at one of the eight surrounding cells in grey. (For interpretation of the colours in the figures, the reader is referred to the web version of this article.)

In this work, each agent is equipped with the linear-strength mechanism introduced in [3], called the *line pushing mechanism.* A line L consists of a sequence of k agents occupying consecutive cells on the grid, say w.l.o.g, $L = (x, y), (x+1, y), \ldots, (x+k-1, y)$, where $1 \leq k \leq n$. The agent $p \in L$ occupying (x, y) is capable of performing an operation of a **line move** by which it can push all agents of L one position rightwards to positions $(x+1, y), (x+2, y), \ldots, (x+k, y)$ in parallel in a single time-step iff there exists an empty cell at $(x+k, y)$, i.e. all of the k agents are pushed one step right at the same time, and only the pushing agent is aware of this move and the other $k-1$ agents are not necessarily informed in a single time-step. The *line moves* towards the "down", "left" and "up" directions are defined symmetrically by rotating the system 90°, 180° and 270° clockwise, respectively. From now on, this operation may be referred to as *move*, *movement* or *step*. An agent can be equipped with an internal linear-strength pushing mechanism or be affected by external linear-strength forces that occur naturally (e.g. gravity) or artificially (e.g. magnetic surface) in a system. We call the number of agents in S the *size* or *order* of the shape, and throughout this work all logarithms are to the base 2.

We assume that the agents share a sense of orientation through a consistent labelling of their local ports. Agents do not know the size of S in advance, nor do they have any other prior knowledge about S. Each agent has a constant memory (of size independent of n) and a local visibility mechanism by which it observes the states of its eight neighbouring cells simultaneously. The agents act as finite automata operating in synchronous rounds consisting of LCM steps. Thus, in every discrete round, an agent observes its own state and for each of its eight adjacent cells, checks whether it is occupied or not. For each of those occupied, it also observes the state of the agent occupying that cell. Then, the agent updates its state or leaves it unchanged and performs a *line move* in one direction $d \in \{$*up, down, right, left*$\}$ or stays still. A *configuration* C of the system is a mapping from $\mathbb{Z}^2_{\geq 0}$ to $\{0\} \cup Q$, where Q is the state space of agents. We define $S(C)$ as the shape of configuration C, i.e. the set of coordinates of the cells occupied in S. Given a configuration C, the LCM steps performed by all agents in the given round, yield a new configuration C' and the next round begins. If at least one move was performed, then we say that this round has transformed $S(C)$ to $S(C')$.

Throughout this work, we assume that the initial shape S_I is Hamiltonian and the final shape is a straight line S_L, where both S_I and S_L have the same order. We also assume that a pre-elected leader is provided at one endpoint of the Hamiltonian path of S_I. It is made available to the agents in the distributed way that each agent p_i knows the local port leading to its predecessor p_{i-1} and its successor p_{i+1}, for all $1 \leq i \leq n$.

An agent $p \in S$ is defined as a 5-tuple (X, M, Q, δ, O), where Q is a finite set of states, X is the input alphabet representing the states of the eight cells that surround an agent p on the square grid, so $|X| = |Q|^8$,

$M = \{\uparrow, \downarrow, \rightarrow, \leftarrow, none\}$ is the set of moves and a transition function $\delta : Q \times X \rightarrow Q \times M$.

2.1. Problem definition

We now formally define the problem considered in this work.

HamiltonianLine. Given any initial Hamiltonian shape S_I contains a Hamiltonian path P computed by a polynomial-time centralised pre-processing (e.g. [46]) where each agent $p \in S_I$ knows its predecessor and successor on P, the agents must form a final straight line S_L of the same order from S_I via line moves while preserving connectivity throughout the transformation.

3. The Distributed Hamiltonian Transformation

In this section, we develop a distributed algorithm exploiting line moves to form a straight line S_L from an initial connected shape S_I which is associated to a graph that contains a Hamiltonian path. As we will argue, this strategy performs $O(n \log n)$ moves, i.e. it is as efficient with respect to moves as the best-known centralised transformation [5], and completes within $O(n^2 \log n)$ rounds, while keeping the whole shape connected during its course. Additional rounds are required in this case mostly for local observations and agent synchronisation.

We assume that through some pre-processing the Hamiltonian path P of the initial shape S_I has been made available to the n agents in a distributed way (i.e. we assume that P is part of the input along with S_I). P starts and ends at two agents, called the head p_1 and the tail p_n, respectively. The head p_1 is leading the process (as it can be used as a pre-elected unique leader) and is responsible for coordinating and initiating all procedures of this transformation. As mentioned earlier, the transformation proceeds in $\log n$ phases, each of which consists of six sub-phases (or sub-routines). Every sub-phase consists of one or more synchronous rounds. It starts with a trivial line of length 1 at the head's endpoint, then it gradually flattens all agents along P gradually while successively doubling its length, until arriving at the final straight line S_L of length n.

A state $q \in Q$ of an agent p will be represented by a vector with seven components $(c_1, c_2, c_3, c_4, c_5, c_6, c_7)$. The first component c_1 contains a label λ of the agent from a finite set of labels Λ (meaning, in this work, constant and independent of the size of the system), c_2 is the transmission state that holds a string of length at most three, where each symbol of the string can either be a special mark w from a finite set of marks W or an arrow direction $a \in A = \{\rightarrow, \leftarrow, \downarrow, \uparrow, \nwarrow, \nearrow, \swarrow, \searrow\}$ and c_3 is called a waiting state and will store a symbol from c_2's string, i.e. a special mark or an arrow. The local Hamiltonian direction $a \in A$ of an agent p indicating predecessor and successor (i.e. a single arrow pointing from an agent to its successor on the Hamiltonian path) is recorded in c_4, the counter state c_5 holds a bit from $\{0, 1\}$, c_6 stores an arrow $a \in A$ for map drawing (as will be explained later) and finally c_7 is holding a pushing direction $d \in M$. The "$\cdot$" mark indicates an empty component; a non-empty component is always denoted by its state. An agent p may be referred to by its label $\lambda \in \Lambda$ (i.e. by the state of its c_1 component) whenever clear from context.

We begin with a high-level explanation of the transformation, assuming that n is a power of 2 (this is only to simplify the exposition; the algorithm is designed to operate in the general case indicating when it needs to distinguish between even and odd cases), Figure 4 depicts an overview of the algorithm. First, it identifies the next 2^i agents on P. These agents are forming a segment S_i which can be in any configuration. To do that, the head emits a signal which is then forwarded by the agents along the line. Once the signal arrives at S_i, it will be used to re-label S_i so that it starts from a head in state s_h, has $2^i - 2$ internal agents in state s, and ends at a tail s_t; this completes the DefineSeg sub-phase. Then, l_h calls CheckSeg in order to check whether the newly defined segment S_i is forming a straight line that is in line with L_i (i.e. both S_i and L_i form a straight line as depicted in Figure 5 downmost) or S_i is a straight line perpendicular to L_i (e.g. see Figure 6 (a)). This can be easily achieved through a moving state initiated at L_i and checking for each agent of S_i its local directions relative to its neighbours. If S_i is in line or perpendicular to L_i, then l_h calls Merge to combine L_i and S_i into a new line L_{i+1} of length 2^{i+1} and starts a new round $i + 1$. If neither is true, l_h proceeds with the next sub-phase, DrawMap.

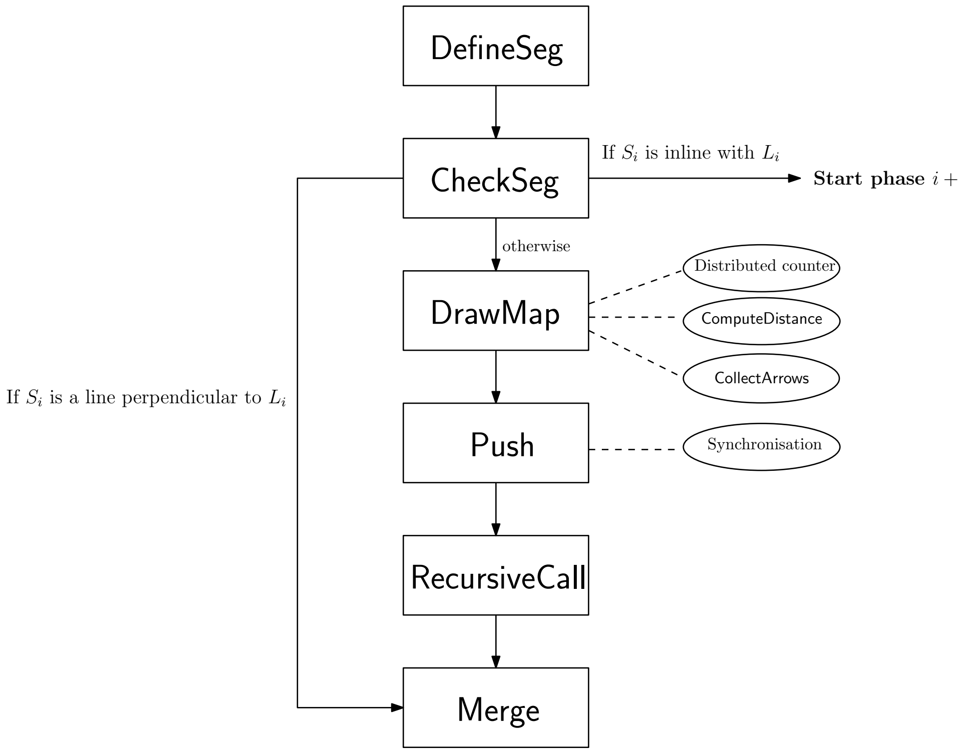


Figure 4: A schematic representation of the strategy and its sub-procedures in phase i.

In DrawMap, l_h designates a route on the grid through which L_i pushes itself towards the tail s_t of S_i. It consists of two primitives: ComputeDistance and CollectArrows. In ComputeDistance, the line agents act as a distributed counter to compute the Manhattan distance between the tails of L_i and S_i. In CollectArrows, the local directions are gathered from S_i's agents and distributed into L_i's agents, which collectively draw the route map. Once this is done, L_i becomes ready to move and l_h can start the Push sub-phase. During pushing, l_h and l_t synchronise the movements of L_i's agents as follows: (1) l_h pushes while l_t is guiding the other line agents through the computed route and (2) both are coordinating any required swapping of states with agents that are not part of L_i but reside in L_i's trajectory. Once L_i has traversed the route completely, l_h calls RecursiveCall to apply the general procedure recursively on S_i in order to transform it into a line L_i'. Figure 21 shows a graphical illustration of the core recursion on the special case of a diagonal line shape. Finally, the agents of L_i and L_i' combine into a new straight line L_{i+1} of 2^{i+1} agents through the Merge sub-procedure. Then, the head l_h of L_{i+1} begins a new phase $i+1$.

Now, we are ready to proceed with the detailed description of each sub-phase.

3.1. Defining the next segment S_i

This is the first sub-phase of this algorithm, and it is charge of identifying the next segment S_i on the Hamiltonian path P in such a way that each agent p_l in a terminal straight line L_i of length 2^i activates its corresponding agent p_s in S_i(i.e the first agent in L_i defines the first agent in S_i and so on), as depicted in Figure 5. Given L_i on P, this sub-phase defines S_i and activates its 2^i agents as follows: The line head l_h transmits a special mark "Ⓗ" to go through all active agents in the Hamiltonian path P. It updates its transmission component c_2 as follows: $\delta(l_h, \cdot, \cdot, a \in A, \cdot, \cdot, \cdot) = (l_h, Ⓗ, \cdot, a \in A, \cdot, \cdot, \cdot)$. This is propagated by active agents by always moving from a predecessor p_i to a successor p_{i+1}, until it arrives at the first inactive agent with label k, which then becomes active and the head of its segment by updating its label as $\delta(k, Ⓗ, \cdot, a \in A, \cdot, \cdot, \cdot) = (s_h, \cdot, \cdot, a \in A, \cdot, \cdot, \cdot)$. Similarly, once a line agent p_i passes "Ⓗ" to p_{i+1}, it also initiates and propagates its own mark "Ⓛ" to its successor p_{i+1} in order to activate a corresponding segment agent s. The line tail l_t emits "Ⓣ" to activate the segment tail s_t, which in turn bounces off a special end mark "⊗" announcing the end of DefineSeg. By that time, the next segment S_i consisting of 2^i agents, starting from a head labelled s_h, ending at a tail s_t and having $2^i - 2$ internal agents with label s, has been defined. The "⊗" mark is propagated back to the head l_h along the active agents, by always moving from p_{i+1} to p_i. Now, Lemma 1 proves correctness of DefineSeg.

Lemma 1. *DefineSeg correctly activates all agents of S_i in $O(n)$ rounds.*

Proof. When an active agent p_i of label l or l_t observes the head mark "Ⓗ" on the state of its predecessor p_{i-1}, it then updates transmission state c_2 to "Ⓗ" and initiates a special mark on its waiting state c_3. This can be either inline "Ⓛ" or tail "Ⓣ" mark. Once an inactive agent notices predecessor with "Ⓛ" or "Ⓣ" mark, it activates and changes its label c_1 to the corresponding state, "s" or "s_t", respectively. Immediately after activating the tail s_t, it bounces off a special end mark "⊗" transmitted along all active agents back to the head l_h of the line to indicate the end of this sub-phase. That is, the tail s_t sets "⊗" in transmission state, so when agent p_i observes successor p_{i+1} showing "⊗", it updates its transmission state to $c_2 \leftarrow \otimes$. When witnessing predecessor and successor with an empty transmission state, an agent resets c_2 to "$\cdot$". Once the head l_t detects the "⊗" mark, it then calls the next sub-routine, CheckSeg. Because the transformation always doubles the length of the straight line, the line L_i cannot be of an odd length, unless the initial line of one agent labelled l_h is adjacent to an inactive neighbour on the path P. In this case, the adjacent agent activates when it observes the head mark, updates label to s_h and reflects an end special mark "⊗" back to l_h. □

Figure 5 depicts an implementation of DefineSeg on a straight line of four agents, in which the next segment S_i is represented as a line for clarity, but it can be of any configuration. All transitions of this sub-routine is given in Algorithm 1, excluding all that have no effect. In the following, Lemma 2 discusses the runtime of DefineSeg.

Lemma 2. *DefineSeg requires at most $O(n)$ rounds to define S_i.*

Proof. By Lemma 1, the head mark "Ⓗ" shall traverse all agents of the line L_i of length $|L_i|$ until it arrives at the first inactive agent, and this takes at most $O(|L_i|)$ rounds. Thus, all other agents on the line propagate marks that take $O(|L_i|)$ parallel rounds to reach their corresponding agents on the next segment. In the worst case, the line can be of length $n/2$, which requires at most $O(n)$ rounds of communication to identity the next segment S_i of length $n/2$. □

3.2. Checking the next segment S_i

This sub-phase checks the geometrical configuration of the newly defined segment S_i, determining if it is in line with L_i, perpendicular to L_i or contains one turn (L-shape). It aims to save energy in the system, surpassing one or more of the subsequent sub-phases. Through some local communications, the head l_h will recognise if: (1) S_i of more than one turn, then stat the next sub-phase DrawMap. (2) S_i is forming a line perpendicular to L_i (see Figure 6(a)) or of a single L-shaped turn (see Figure 6(b)), where it can simply turn at its corner to create L_{i+1} and save the cost of DrawMap, Push and RecursiveCall. (3) S_i and L_i already form a single straight line L_{i+1} of double length (as illustrated in Figure 5), therefore the next phase $i+1$ begins.

When l_h observes "⊗", it propagates its own local direction stored in component $c_4 = a \in A$ by updating $c_2 \leftarrow c_4$. Then, all active agents on the path forward a from p_i to p_{i+1} via their transmission components. Whenever a p_i with a local direction $c_4 = a' \in A$ notices $a' \neq a$, it combines a with its local direction a' and changes its transmission component to $c_2 \leftarrow aa'$. After that, if a p'_i having $c_4 = a'' \in A$ observes $a'' \neq a'$, it updates its transmission component into a negative mark, $c_2 \leftarrow \neg$. All signals are to be reflected by the segment tail s_t back to l_h, which acts accordingly as follows: starts the next sub-phase DrawMap if it observes "¬", calls Merge to combine the two perpendicular lines if it observes aa' or begins a new phase $i+1$ if it receives back its local direction a. Algorithm 2 shows the pseudocode of this sub-routine. Now, Lemma 3 establishes the correctness of CheckSeg.

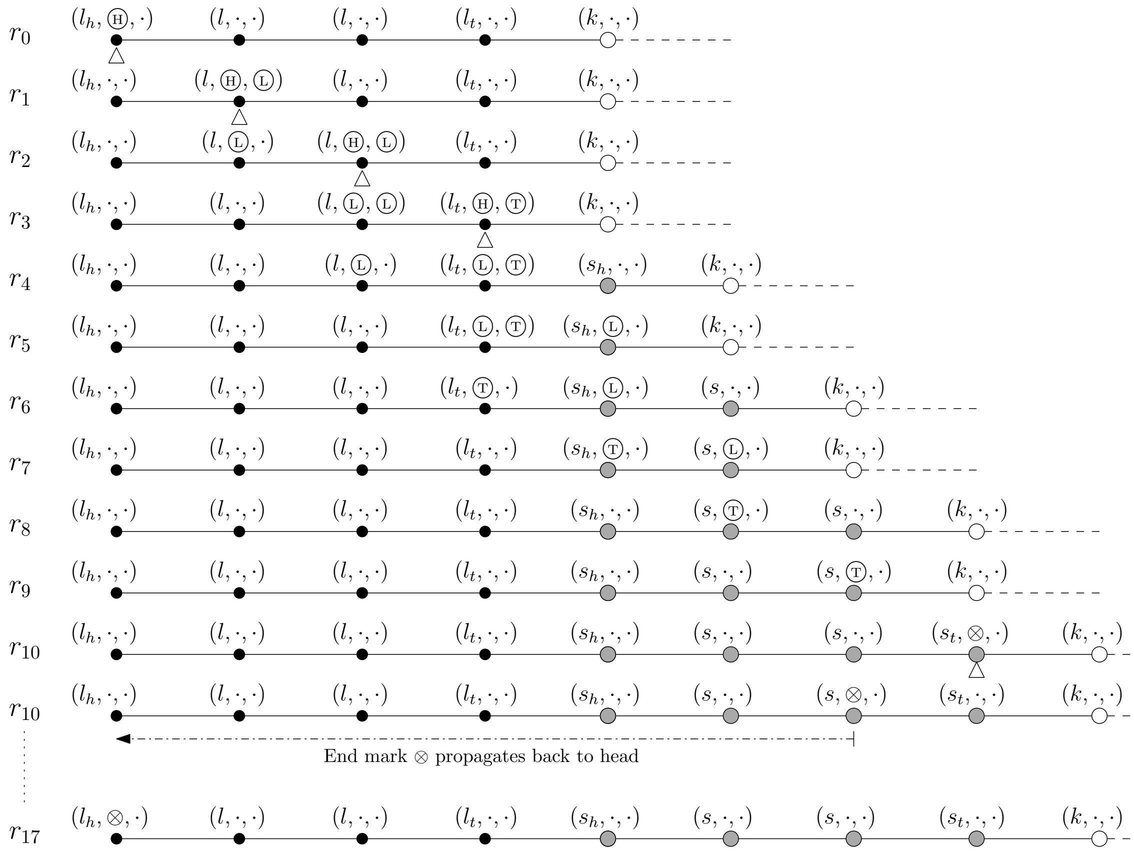


Figure 5: An implementation of DefineSeg on a line L_i of four agents depicted as black dots. Each agent uses only its 3 state components (c_1, c_2, c_3), where c_1 is the label state, c_2 the transmission state and c_3 the waiting mark state. In round $r = 0$, L_i is labelled correctly, starting from a head l_h and ends at a tail l_t with internal agents l. Inactive agents with circles are labelled k. First, l_h sets $c_2 \leftarrow$ Ⓗ. Thereafter, when an active agent p_i notices predecessor p_{i-1} showing "Ⓗ" , it updates to $c_2 \leftarrow$ Ⓗ (a small triangle indicates this initialisation in rounds r_0, r_1, r_2, r_3, and r_{10}). Agents of label l and l_t propagate "Ⓛ" and "Ⓣ", respectively. Whenever an inactive agent (white dot) sees predecessor presenting a mark, it activates (grey dot) and updates label to corresponding state. Once activating the segment tail s_t, it propagates an end mark "⊗" back to the head to start CheckSeg.

Lemma 3. *CheckSeg correctly checks which one of the following mutually exclusive properties is satisfied by the configuration of S_i: (i) S_i is in line with L_i, (ii) S_i forms a straight line perpendicular to L_i, (iii) S_i forms an L-shape, (iv) S_i contains more than one turn.*

Proof. This sub-routine starts as soon as the head l_h observes the end mark "⊗" of DefineSeg, which means that all agents of the segment S_i are active and labelled correctly. Given that, the input configuration of CheckSeg is a Hamiltonian path terminates at straight line L_i followed by S_i, both are composed of 2^i active agents. All other inactive agents in the rest of the configuration are labelled k. During this sub-phase, the active agents use their local path directions stored in state c_4 by which a p_i knows each port incident to predecessor p_{i-1} and successor p_{i+1}.

Now, l_h updates its transmission state to $c_2 \leftarrow c_4$ where it emits its local direction held in c_4. Assume without loss of generality, c_4 holds a local direction pointing to the east neighbour "→", then l_h performs this state transition: $\delta(l_h, \otimes, \cdot, \rightarrow) = (l_h, \rightarrow, \cdot, \rightarrow)$. This arrow "→" propagates through transmission states to all active agents of L_i and S_i. When a p_{i-1} displays an empty transmission state, each agent p_i updates state c_2 to "·". If "→" matches a local direction stored on c_4 of p_i, then p_i transmits the same arrow "→" from p_{i-1} to p_{i+1}. If p_i stores a turning arrow (e.g. "↓" or "↑") on c_4, then it updates state c_2 with a special L-shape mark, "→L", which is then passed to p_{i+1}. Whenever p_j stores a turning arrow and observes p_{j-1}

Algorithm 1: DefineSeg

$S = (p_1, \dots, p_{|S|})$ is a Hamiltonian shape
Initial configuration: $S \leftarrow S_i$, a line $L \subset S$ of length $k = 1, \dots, \log|S|$, labelled as in Figure 5 topmost

```
p1.c2 ← (H) // head sets a mark in transmission state
repeat
    // each agent acts based on its current label state
    Head l_h:
    if (p1.c2 = (H)) then p_i.c2 ← · // reset transmission state
    if (p_{i+1}.c2 = ⊗) then p1.c2 ← ⊗ // observe end mark; end this sub-phase

    Active:
    if (p_{i-1}.c2 = (H)) // observe predecessor with head mark
     then
        p_i.c2 ← (H)
        if (inline p_i.c1 = l) then p_i.c3 ← (L)
        if (tail p_i.c1 = l_t) then p_i.c3 ← (T)
    end
    if p_{i-1}.c2 = (L) then p_i.c2 ← (L) // predecessor shows inline mark
    if p_{i-1}.c2 = (T) then p_i.c2 ← (T) // predecessor shows tail mark
    if ((p_i.c2 = (H) ∨ (L) ∨ (T)) ∧ p_{i-1}.c2 = ·) then
    p_i.c2 ← p_i.c3 // transmit marks
    p_i.c3 ← · // reset marks
    if p_{i+1}.c2 = ⊗ then p_i.c2 ← ⊗ // successor shows end mark
    if p_i.c2 = ⊗ then p_i.c2 ← · // reset transmission state

    Inactive:
    if (p_{i-1}.c2 = (H)) then p_i.c1 ← s_h // activate to segment head s_h
    if (p_{i-1}.c2 = (L)) then p_i.c1 ← s // activate to insegment s
    if (p_{i-1}.c2 = (T)) then
    p_i.c1 ← s_t // activate to segment tail s_t
    p_i.c2 ← ⊗ // initiate end mark
until (p1.c2 = ⊗)
CheckSeg
```

showing "→L" , p_j initiates a negative mark $c_2 \leftarrow \neg$ and relays it back to l_h, calling out for DrawMap (this proves S_i *contains more than one turn*). Once s_t observes "→L", it bounces off the mark "L" back towards l_h to start Push, which proves that S_i *forms an L-shape.* Otherwise, s_t propagates a special check-mark "✓" backwards, alerting l_h there is no turns on S_i. When l_h detects "✓", it will either call Merge if its predecessor holds a different direction (proving S_i *forms a straight line perpendicular to* L_i) or begin a new phase (proving S_i *is in line with* L_i). □

Now, Lemma 4 provides analysis of CheckSeg.

Lemma 4. *An execution of CheckSeg requires at most $O(n)$ rounds of communication.*

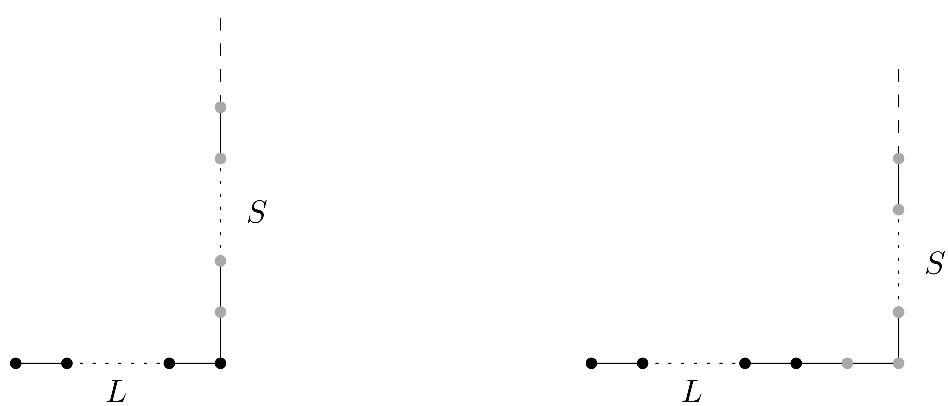
Proof. Consider the worst-case in which the direction mark traverses a n-length path and a special mark "✓" bounces off the other end of the path and returns to the head. This journey takes at most $2n - 2$ rounds, during which an agent p_i, $1 \leq i \leq n$, emits the direction mark to p_{i+1} and "✓" to p_{i-1}, excluding the two endpoints of the path. □

Algorithm 2: CheckSeg

$S = (p_1, \ldots, p_{|S|})$ is a Hamiltonian shape
Initial configuration: $S \leftarrow S_i$, a line $L \subset S$ of length $k = 1, \ldots, \log |S|$, labelled correctly as in Figure 5 bottommost

$p_1.c_2 \leftarrow p_1.c_4$ `// head emits its direction`
repeat
  `// each agent acts based on its current label state`
  Head l_h**:**
  if $(p_1.c_2 = c_4)$ **then** $p_i.c_2 \leftarrow \cdot$ `// reset transmission state`
  if $(p_{i+1}.c_2 = \neg)$ **then** $p_1.c_2 \leftarrow \neg$ `// end this sub-phase`
  if $(p_{i+1}.c_2 = \checkmark)$ **then** start phase $i+1$ `// a new phase begins`
  if $(p_{i+1}.c_2 = \mathsf{L})$ **then** PushLine(L) `//` S_i `has one turn`

  Active:
  if $(p_{i-1}.c_2 = c_4)$ **then** $p_i.c_2 \leftarrow c_4$ `// observe same direction`
  if $p_{i-1}.c_2 \neq c_4)$ **then** $p_i.c_2 \leftarrow c_4\mathsf{L}$ `// show a turn`
  if $(p_{i-1}.c_2 = c_4\mathsf{L})$ **then** $p_i.c_2 \leftarrow \neg$ `// show another turn`
  if $(p_{i+1}.c_2 = \neg \vee \checkmark \vee \mathsf{L})$ **then** $p_i.c_2 \leftarrow p_{i+1}.c_2$ `// transmit marks backwards`
  if $(p_{|2L|-1}.c_2 = c_4)$ **then** $p_{|2L|.c2} \leftarrow \checkmark$`//` s_i `transmits mark backwards`
  if $(p_{|2L|-1}.c_2 = c_4\mathsf{L})$ **then** $p_{|2L|.c2} \leftarrow \mathsf{L}$`//` s_i `transmits mark backwards`
  if $(p_{i-1}.c_2 \neq \cdot)$ **then** $p_i.c_2 \leftarrow \cdot$ `// reset transmission state`
until $p_1.c_2 = \neg$
DrawMap



(a) L_i is perpendicular to S_i.

(b) S_i has a single turn.

Figure 6: Two configurations of a Hamiltonian path terminates at a straight line L_i (in black dots) followed by a segment S_i (in grey dots) on the path.

3.3. Drawing a route map

This local technique creates a map of minimum turns, with the goal of achieving the lowest cost. By a map, we mean a set of directions $\bar{A} \subseteq A$ representing the shortest path (route) on the grid from the line tail l_t to the segment tail s_t. On the square grid, there are several finite routes between l_t and s_t whose length (Manhattan distance) is equal. The most cost-effective route to draw is one with a single turn, such an L-shaped route, because a turn costs $2|L_i|$ whereas a single vertical or horizontal movement only costs one line move. For the purpose of connectivity preservation, it can be demonstrated that there exists some worst-case routes in which the line L_i may disconnect while travelling towards the tail of S_i. In Figure 7, for example, the Manhattan distance between l_t and s_t is $\Delta(l_t, s_t) > |L_i|$ in which L_i will break the connectivity if it moves from (x, y) vertically or horizontally to either (x', y) or (x, y') and keeps moving to s_t on (x', y'). In this case, L_i must first pass through a middle agent of S_i to preserve the whole connectivity of the shape.

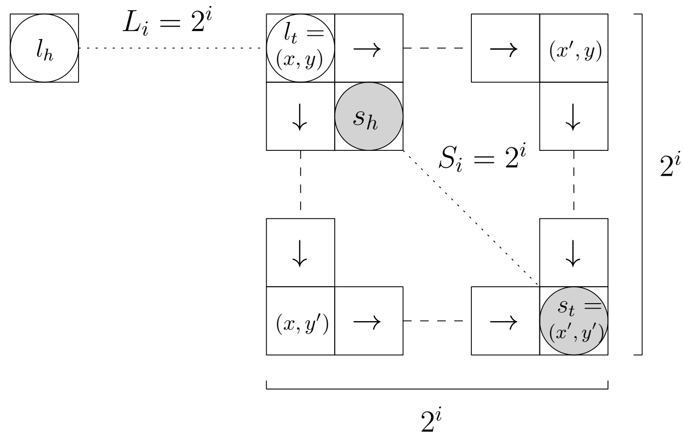


Figure 7: An example of a worst-case route.

Thus, this distance $\Delta(l_t, s_t)$ is important in determining whether to take an L-shaped route directly to s_t or to go through an intermediary agent of S_i, passing through two L-shaped routes. From our distributed perspective, the Manhattan distance $\Delta(l_t, s_t)$ cannot be computed in a straightforward manner due to several challenges, such as individuals with constant local memory and limited computational power. Below, DrawMap addresses these challenges by using L_i agents as (1) a distributed binary counter for calculating the distance and (2) a distributed memory for storing local directions of agents, which collectively draw the route map.

This sub-phase computes the Manhattan distance $\Delta(l_t, s_t)$ between the line tail l_t and the segment tail s_t, by exploiting ComputeDistance in which the line agents implement a distributed binary counter. First, the head l_h broadcasts "Ⓒ" to all active agents, asking them to commence the calculation of the distance. Once a segment agent p_i observes "Ⓒ", it emits one increment mark "$\oplus$" if its local direction is cardinal or two sequential increment marks if it is diagonal. The "$\oplus$" mark is forwarded from p_i to p_{i-1} back to the head l_h. Correspondingly, the line agents are arranged to collectively act as a distributed binary counter, which increases by 1 bit per increment mark, starting from the least significant at l_t.

When a line agent observes the last "$\oplus$" mark, it sends a special mark "①" if $\Delta(l_t, s_t) \leq |L_i|$ or "②" if $\Delta(l_t, s_t) > |L_i|$ back to l_h. As soon as l_h receives "①" or "②", it calls CollectArrows to draw a route that can be either heading directly to s_t or passing through the middle of S_i towards s_t. In CollectArrows, l_h emits "$\leftrightarrows$" to announce the collection of local directions (arrows) from S_i. When "$\leftrightarrows$" arrives at a segment agent, it then propagates its local direction stored in c_4 back towards l_h. Then, the line agents distribute and rearrange S_i's local directions via several primitives, such as cancelling out pairs of opposite directions, priority collection and pipelined transmission. Finally, the remaining arrows cooperatively draw a route map for L_i (see Definition 1). Below, we give more details of DrawMap.

Definition 1 (A route)**.** *A route is a set of cells $R = [c_1, \ldots, c_{|R|}]$ on $\mathbb{Z}^2$, where c_i and c_{i+1} are two cells adjacent vertically or horizontally, for all $1 \leq i \leq |R| - 1$. For a system configuration C, C_R denotes the configuration of R where $C_R \subset C$ is defined by $[c_1, \ldots, c_{|R|}]$.*

3.3.1. Distributed binary counter

Due to the limitations of this model, individual agents cannot calculate and keep non-constant numbers in their state. Alternatively, the line L_i of 2^i agents can be utilised as a distributed binary counter (similar to a Turing machine tape) which is capable to store up to $2^i - 1$ unsigned values. This i-bit binary counter supports increment which is the only operation we need in this procedure. Thus, in principle and practice, the counter cannot overflow because the maximum distance between l_t and s_t can be at most $\Delta(l_t, s_t) = 2i$ where $2^i - 1 >= 2i$, for all $i >= 0$. Each agent's counter state c_5 is initially "$\cdot$" and can then hold a bit from $\{0, 1\}$. The line tail l_t denotes the least significant bit of the counter. An increment operation is performed as follows: Whenever a line agent p_i detects p_{i+1} showing an increment mark "$\oplus$", p_i switches counter component c_5 from "$\cdot$" or 0 to 1 and destroys the "$\oplus$" mark. If p_i holds 1 in c_5, it flips 1 to 0 and redirects the increment mark "$\oplus$" to p_{i-1} (i.e. update the transmission state c_2 to "$\oplus$"). All increment marks triggered by segment agents, including the last one initiated by the segment tail, are transmitted back to the counter. When a line agent detects the last mark, it notifies the line head, which is in charge of resetting the counter. See an implementation of this counter in Figure 8.

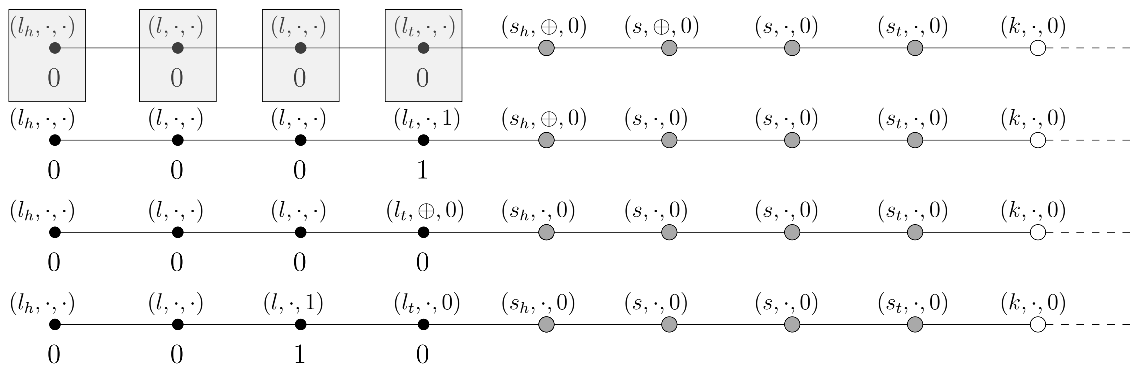


Figure 8: A 4-bit line counter L_i. Agents of L_i and S_i are depicted by black and grey dots, respectively. The state of an agent is (c_1, c_2, c_5) denoting c_1 the label, c_2 transmission and c_5 counter components, omitting others with no effect. Each shaded area shows a corresponding binary number, with the line tail l_t representing the counter's least significant bit. Top: the counter represents a decimal value of 0. 2nd: an increment of 1. 3rd: l_t flips state c_5 from 1 to 0 and updates c_2 with "$\oplus$". Bottom: the counter increased by 1 corresponding to a decimal value 2.

3.3.2. ComputeDistance procedure

Initially, the head l_h emits a special mark "Ⓒ" to all active agents, preparing them the for the calculation of the Manhattan distance $\Delta(l_t, s_t)$ between the line tail l_t and the segment tail s_t. The segment agents just pass "Ⓒ" along until it reaches s_t, then the counting effectively happens later. Whenever a segment agent p_i (of label s_h, s or s_t) observes p_{i-1} with "Ⓒ", it performs one of two transitions: (1) It updates transmission state to $c_2 \leftarrow \oplus$ if its local direction stored in c_4 is cardinal (horizontal or vertical) from $\{\rightarrow, \leftarrow, \uparrow, \downarrow\}$. (2) If c_4 holds a diagonal direction from $\{\nwarrow, \nearrow, \swarrow, \searrow\}$, it receptively updates the transmission and waiting states, c_2 and c_3, to "$\oplus$". Eventually, the segment head s_h produces the last special increment mark "$\oplus'$". In principle, any diagonal direction between two cells in a square grid can increase the distance by two (in the Manhattan distance), whereas horizontal and vertical directions always increase it by one.

As a result, all increment marks initiated by segment agents are transmitted backwards to the counter L_i, similar to the propagation of end mark described in DefineSeg. Hence, the binary counter increases by 1 bit each time it detects "$\oplus$", starting from the least significant bit stored in l_t. Because of transmission parallelism, the binary counter may increase by more than one bit in a single round. When a line agent p_i sees predecessor with the last increment mark "$\oplus'$", p_i passes "①" towards the line head l_h. This mark "①" is altered to "②" on its way to l_h only if it passes a line agent of a counter state $c_5 = 1$, otherwise it is left unchanged. Eventually, the head l_h observes either "①", by which it calls CollectArrows procedure to draw a route map directly to the tail s_t of S_i, or "②", by which it calls CollectArrows to push via a middle agent s towards s_t. We provide Algorithm 3 of the ComputeDistance procedure below.

Let $\Delta(l_t, s_t)$ denote the Manhattan distance between the line tail l_t and the segment tail s_t. Lemma 5 below demonstrates that this technique calculates $\Delta(l_t, s_t)$ in linear time.

Algorithm 3: ComputeDistance(L_i, S_i)

$S = (p_1, \ldots, p_{|S|})$ is a Hamiltonian shape
Initial configuration: a straight line L_i and a segment S_i labelled as in Figure 8

1. The line head l_h propagates counting mark Ⓒ along L_i and S_i
2. Once Ⓒ arrives at the segment tail s_t, a segment agent acts as follows:
3. s_t sends one increment $\oplus$ back to l_h if its direction is cardinal or two $\oplus$ if diagonal
`// pipelined transmission`
4a. s observes $\oplus$, sends one increment $\oplus$ back to l_h if its direction is cardinal or two $\oplus$ if diagonal
4b. s_l observes $\oplus$, sends one increment $\oplus'$ back to l_h if its direction is cardinal or two $\oplus'$ if diagonal
5. The distributed counter L_i increases by 1 bit each time it receives $\oplus$
6. A line agent observes the last $\oplus'$ coming to L_i, sends a mark ① back to l_h
7a. Each line agent observes ① and has 1 bit, passes ② towards l_h
7b. Each line agent observes ① and has 0 bit, passes ① towards l_h
7c. Each line agent observes ②, passes ② towards l_h
`// distance` $\Delta(l_t, s_t) \leq i$
8a. If l_h sees ①, it calls CollectArrows to draw one L-shaped route
`// distance` $\Delta(l_t, s_t) > i$
8b. If l_h sees ②, it calls CollectArrows to draw two L-shaped route
8c. If l_h sees either "$\oplus$" or "$\oplus'$",it calls CollectArrows to draw two L-shaped route
`// reset`
9. l_h resets the counter

Lemma 5. *ComputeDistance requires $O(|L_i|)$ rounds to compute $\Delta(l_t, s_t)$.*

Proof. Consider an input configuration labelled $(\overbrace{l_h, \ldots, l, \ldots, l_t}^{L_i}, \overbrace{s_h, \ldots, s, \ldots, s_t}^{S_i}, k, \ldots, k)$, starting at a line head p_1 of label l_h, where $|L_i| = |S_i|$. We only show affected states in this proof. Initially, l_h emits a counting mark "Ⓒ" by updating transmission state to $p_1.c_2 \leftarrow$ Ⓒ, then l_h resets transmission state to $c_2 \leftarrow \cdot$ in subsequent rounds. Once an active agent p_i in round r_{j-1} (where $j \leq 2|L_i|$) detects predecessor showing state $p_{i-1}.c_2 =$ Ⓒ, it updates transmission state to $p_i.c_2 \leftarrow$ Ⓒ in r_j and then resets $p_i.c_2 \leftarrow \cdot$ in r_{j+1}. Upon arrival of "Ⓒ" at s_t, its predecessor changes transmission state to $c_2 \leftarrow \oplus$ and puts another increment mark in waiting state $c_3 \leftarrow \oplus$ if it stores a diagonal arrow in its local direction c_4.

Due to the goal of counting, the direction of s_t is dropped. Each segment agent p_i of label s_h and s observes a successor presenting state $p_{i+1}.c_2 = \oplus$ in round r_{j-1}, then the following transitions apply in r_j: (1) $p_i.c_2 \leftarrow \oplus$ if $p_{i+1}.c_2 = w\oplus$, (2) $p_i.c_2 \leftarrow \oplus$ if $p_{i+1}.c_2 = \cdot$ and $p_i.c_3 = \oplus$, (3) the head of segment s_h sets $p_i.c_2 \leftarrow \oplus'$ if $p_{i+1}.c_2 = \cdot$ and $p_i.c_3 = \oplus$ and (4) $p_i.c_2 \leftarrow \cdot$ if $p_{i+1}.c_2 = \cdot$ and $p_i.c_3 = \cdot$.

Correspondingly, the line agents (of labels l_h, l and l_t) behave as a binary counter described above and illustrated in Figure 8. When a line agent p_i detects "$\oplus$" in the state of p_{i+1} in round r_{j-1}, it updates state based on one of these transitions in round r_{j-1}: (1) $p_i.c_5 \leftarrow 1$ if $p_i.c_5 = \cdot$ or $p_i.c_5 = 0$ or (2) $p_i.c_5 \leftarrow 0$ and $p_i.c_2 \leftarrow \oplus$ if $p_i.c_5 = 1$. In the case where the last increment mark "$\oplus'$" detected by p_i in round r_{j-1}, then p_i updates state to $p_i.c_2 \leftarrow$ ① in r_j. When p_{i-1} observes ①, then it updates states to either (1) $p_{i-1}.c_2 \leftarrow$ ① if $p_{i-1}.c_5 = 0$ or (2) $p_{i-1}.c_2 \leftarrow$ ② if $p_{i-1}.c_5 = 1$. Thus, the head l_h sees "①", "②", "$\oplus$" or the last increment mark "$\oplus'$", it acts appropriately (calls CollectArrows procedure and reset the counter). As a result, the counter size is sufficient to ensure that the distance $\Delta(l_t, s_t)$ does not exceed the line length $|L_i|$.

Now, we analyse the cost of communication of this procedure in a number of rounds. First, the counter mark "Ⓒ" goes on a journey that takes $t_1 = 2|L_i| = O(|L_i|)$ rounds. That is, the pipelined transmission of increment marks requires at most $t_2 = O(|L_i|)$ parallel rounds of communication. Moreover, the marks "①" or "②" travel to the head l_h within at most $t_3 = O(|L_i|)$. Finally, l_h requires an additional $t_4 = O(|L_i|)$ rounds to reset the counter. Altogether, the total running time is bounded by $t = t_1 + t_2 + t_3 + t_4 = O(|L_i|)$ parallel rounds. □

3.3.3. CollectArrows procedure

Informally, the distance obtained from the ComputeDistance procedure can be (1) equal or less than the line length $|L_i|$ (l_h observes this mark "①") or (2) greater than $|L_i|$ (l_h observes "②"). In case (1), it propagates a special collection mark "⇆" through all active agents until it reaches the segment tail s_t. When "⇆" arrives, s_t broadcasts its local arrow in c_4 back to l_h via active agent transmission states. This journey accomplishes the following: (a) Gathers arrows similar to s_t and puts them in priority transmission. (b) Eliminates pairs of opposite arrows and replaces them with a hash mark "#". (c) Arranges the arrows on L_i's distributed memory. In case (2), l_h emits a special mark "Ⓜ" to s_h, defining a midpoint on S_i through which the line L_i passes towards s_t.

Now, s_h propagates two marks down s_t, a fast mark "(m1)" is transmitted every round and a slow mark moves three rounds slower "(m2)". The fast mark "(m1)" bounces off s_t, where both "(m1)" and "(m2)" meet in a S_i middle agent p_j, which changes label to s'_t and a successor p_{j+1} switches to s'_h. This temporarily divides S_i into two segments, $S_i^1 = s_h, \ldots, s'_t$ and $S_i^2 = s'_h, \ldots, s_t$. The middle agent s'_t propagates "Ⓜ" to tell l_h that a midpoint has been identified. Case (1) is then repeated twice to collect arrows from S_i^1 and S_i^2 and distribute them into the line agents (distributed memory). After that, Push(S) begins. Algorithm 4 presents the pseudocode that briefly formulates this procedure.

Algorithm 4: CollectArrows(L_i, S_i)

Input: a straight line L_i and a segment S_i

`// priority and pipelined transmission, see text for details`

(A) Line head l_h observes ①

1. l_h propagates collection mark ⇆
2. Each active agent p_i emits ⇆ to p_{i+1}
3. s_t observes ① and propagates its direction d in c_4, $c_2 \leftarrow c_4$
4. Each segment agent p_i passes a direction to p_{i-1}
5. Distribute directions into the line agents
6. Rearrangement of directions
7. Push(S) begins

(B) Line head l_h observes ②

1. l_h propagates a midpoint mark Ⓜ
2. Each line agent p_i broadcasts Ⓜ to p_{i+1}
3. s_h sees Ⓜ, then emits fast (m1) and slow (m2) waves down to s_t
4. (m1) bounces off s_t and meets (m2) at middle agent p_j with label changed to s'_t
5. s'_t propagates Ⓜ to l_h
6. Once l_h sees Ⓜ again, it goes to (A)

Next, Lemma 6 proves the correctness and analysis of CollectArrows.

Lemma 6. *The CollectArrows procedure completes within $O(|L_i|)$ rounds.*

Proof. Given an initial configuration defined in Lemma 5. Assume the Manhattan distance $\delta(l_t, s_t) \leq |L_i|$. For simplicity, we prove (A) in algorithm 4 showing only affected states. Once l_h observes ①, it emits a collection mark "⇆", which then transfers forwardly among active agents until it reaches s_t, similar to counting mark transmission described previously in Lemma 5. When s_t detects "⇆", it updates transmission state c_2 with its local direction held in $c_4 = d$; recall that d is an arrow that locally shows where the Hamiltonian path comes in and out, $d \in \{\rightarrow, \leftarrow, \downarrow, \uparrow, \nwarrow, \nearrow, \swarrow, \searrow\}$, i.e. a single arrow pointing from an agent (the *predecessor*) to the next agent (its *successor*) on the Hamiltonian path.

In what follows, we distinguish between cardinal $\{\rightarrow, \leftarrow, \downarrow, \uparrow\}$ and diagonal directions $\{\nwarrow, \nearrow, \swarrow, \searrow\}$. Figure 9 shows how local arrows are assigned to agents according to the Hamiltonian path. For a cardinal local direction, s_t updates transmission state to $c_2 \leftarrow d$ and marks local direction state with a star $c_4 \leftarrow d^\star$,

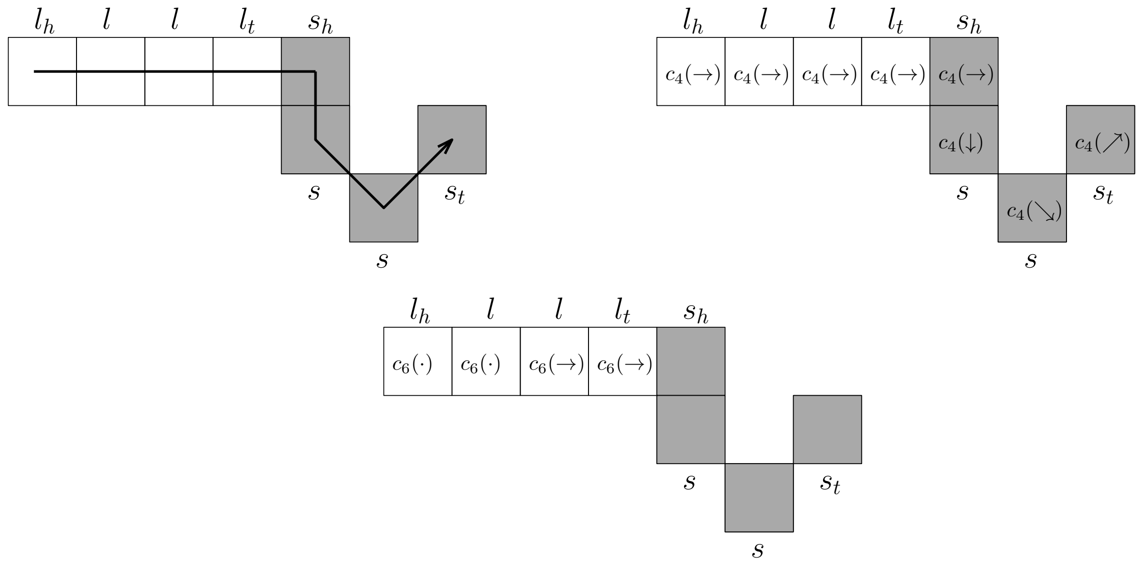


Figure 9: Drawing a map: from top-left a path across occupied cells and corresponding local arrows stored on state c_4 in top-tight, where the diagonal directions, "$\searrow$" and "$\nearrow$", are interpreted locally as, "$\downarrow\rightarrow$" and "$\uparrow\rightarrow$". The bottom shows a route map drawn locally on state c_6 of each line agent.

indicating that d has been collected. A diagonal local direction between any two neighbouring cells on the two-dimensional square grid is made up of two cardinal arrows, such as $\nwarrow$ is composed of $\uparrow$ and $\leftarrow$. In other words, an agent needs to move two steps to occupy an adjacent diagonal cell. For example, if s_t stores a diagonal direction in c_4, it puts d^1 on transmission $c_2 \leftarrow d^1$, d^2 on waiting state $c_3 \leftarrow d^2$, and marks it with a star, $c_4 \leftarrow d^\star$. Next round, the transmission state of s_t resets $c_2 \leftarrow \cdot$ if c_3 is empty or sets $c_2 \leftarrow c_3$ if c_3 contains an arrow.

We now show the priority and pipelined collection of local arrows of S_i (in Algorithm 4). Assume a direction (arrow) d^+ transmits from the segment tail s_t, travelling through transmission states via an active agent p_{i+1} to p_i. When p_{i+1} encounters an opposite arrow d^- recorded in transmission state of p_i, both directions are erased and replaced by the hash sign "#". If p_{i+1} and p_i hold the same direction d^+, then both directions take priority in c_2 (i.e. both are assigned to the transmission state c_2 in order to transmit before any recoded marks in the waiting state c_3). If p_{i+1} observes p_i with a perpendicular arrow $\perp d$, then d^+ is placed in c_2 and $\perp d$ in waiting state c_3. Therefore, any directions kept in waiting states will eventually be sent via transmission states towards the line agents.

For example, eight agents make up the S_i configuration shown in Figure 10, and their arrows are collected in Figure 11 where L_i (white vertices) and S_i (grey vertices) are shown as a tab for better visibility. In the topmost shape, local directions c_4 are inside vertices, label c_1 and transmission c_2 above vertices, waiting c_3 (only for segment agents) and map state c_6 (only for line agents) below vertices. The process starts from round r_j downwards. The associated transitions for each active agent are detailed below, though they may be complicated, therefore we supplement Figure 11 to make this sub-phase easier to understand.

Let p_i be a segment agent of label s or s_t in round r_{j-1}, where $j \leq 2|L_i|$. Then we show how p_i acts when the direction is either cardinal or diagonal. In the first case, consider p_i of an uncollected cardinal direction d_i observes p_{i+1} showing a direction d_{i+1}, two directions $d_{i+1}(d^1_{i+1}d^2_{i+1})$ or # in transmission component c_2. Then, p_i updates its state in r_j as follows: (1) Set d_{i+1} or d^1_{i+1} in transmission $p_i.c_2 \leftarrow p_{i+1}.c_2$, put d_i in waiting $p_i.c_3 \leftarrow p_i.c_4$ and mark it $p_i.c_4 \leftarrow d^\star_i$ if d_i is perpendicular to d_{i+1}, such as $\rightarrow$ and $\uparrow$. (2) Set $p_i.c_2 \leftarrow \#$, put d_i in waiting $p_i.c_3 \leftarrow p_i.c_4$ and mark its local direction $p_i.c_4 \leftarrow d^\star_i$ if d_i and d_{i+1} are a pair of opposite arrows, such as $\uparrow$ and $\downarrow$. (3) Set both directions d_{i+1} and d_i in transmission $p_i.c_2 \leftarrow d_{i+1}d_i$, resets $c_3 \leftarrow \cdot$ and mark d_i with a star $p_i.c_4 \leftarrow d^\star_i$ if d_i and d_{i+1} are a pair of same arrows, such as $\uparrow$ and $\uparrow$. When a cardinal direction is already collected $d^\star_i$, p_i sets d_{i+1} or d^1_{i+1} in transmission $p_i.c_2 \leftarrow p_{i+1}.c_2$. If d_{i+1} (or

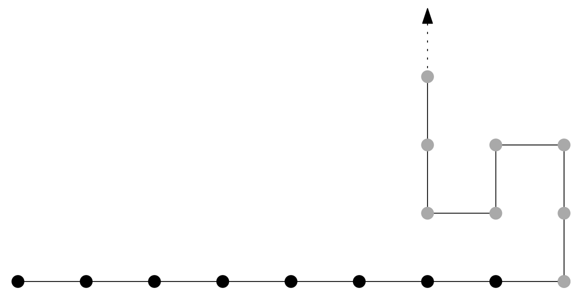


Figure 10: A configuration of L_i (black dots) and S_i (grey dots).

d^1_{i+1}) and $c_3 = d_i$ are having the same direction, then p_i sets $p_i.c_2 \leftarrow d_{i+1}d_i$ (or $p_i.c_2 \leftarrow d^1_{i+1}d_i$) and resets $p_i.c_3 \leftarrow \cdot$. If $p_{i+1}.c_2$ is empty, then p_i puts waiting direction in transmission $p_i.c_2 \leftarrow p_i.c_4$ or resets $p_i.c_2 \leftarrow \cdot$, otherwise.

In the second case, p_i holds an uncollected diagonal arrow $d_i(d^1_i d^2_i)$ in r_{j-1}, so it performs one of the following in r_j: (1) Set d_{i+1} and d^1_i in transmission $p_i.c_2 \leftarrow d_{i+1}d^1_i$, put d^2_i in waiting $p_i.c_3 \leftarrow d^2_i$ and mark d_i with a star $p_i.c_4 \leftarrow d^\star_i$ if d_{i+1} and d^1_i (or d^2_i) are similar, such as $\uparrow$ and $\nwarrow = (\uparrow\leftarrow)$. (2) Set $p_i.c_2 \leftarrow \#$, put d^2_i in waiting $p_i.c_3 \leftarrow d^2_i$ and mark the direction $d^\star_i$ if d_{i+1} is opposites to either d^1_i or d^2_i, such as $\uparrow$ and $\swarrow = (\downarrow\leftarrow)$. If a diagonal arrow has been already collected $d^\star_i$, then p_i sets d_{i+1} or d^1_{i+1} in transmission $p_i.c_2 \leftarrow p_{i+1}.c_2$. If d_{i+1} (or d^1_{i+1}) and waiting direction $c_3 = d_i$ are the same, then p_i updates to $p_i.c_2 \leftarrow d_{i+1}d_i$ (or $p_i.c_2 \leftarrow d^1_{i+1}d_i$) and resets $p_i.c_3 \leftarrow \cdot$. If $p_{i+1}.c_2$ is empty then, p_i puts waiting direction in transmission $p_i.c_2 \leftarrow p_i.c_4$ or resets $p_i.c_2 \leftarrow \cdot$, otherwise.

Meanwhile, the line agents receive the collected arrows and divide them among respective states as follows. Let p_i denote a line agent, holding a map state $p_i.c_6 = \cdot$, observes p_{i+1} showing a direction d_{i+1} or a hash sign "#". Then, p_i acts accordingly: (1) $p_i.c_6 \leftarrow d_{i+1}$, (2) if p_i is l_h or sees p_{i-1} with a map state $c_6 = \#$, then $p_i.c_6 \leftarrow \#$. Whenever $p_i.c_6 \neq \cdot$ detects $p_{i+1}.c_2 = d_{i+1}$ or $p_{i+1}.c_2 = \#$, then p_i updates state to d_{i+1} or "#" if $p_{i-1}.c_6 = \cdot$. Once the line tail l_t of a non-empty map component detects $p_{i+1}.c_2 = \cdot$, it propagates a special mark "$\leftrightarrows\checkmark$" via line agents towards l_h, announcing the completion of arrows collection.

Now, let us discuss (B) in algorithm 4 in which l_h observes "②", indicating the Manhattan distance $\delta(l_t, s_t) > |L_i|$. In reaction to this, l_h emits the midpoint mark "Ⓜ" forwardly down the line agents towards s_h. Once s_h detects "Ⓜ", it emits two waves via the segment, fast "$\textcircled{m1}$" and slow "$\textcircled{m2}$". The fast wave "$\textcircled{m1}$" moves from p_i to p_{i+1} every round, while the slow wave "$\textcircled{m2}$" passes every three rounds. In this way, the fast wave "$\textcircled{m1}$" bounces off s_t and meets "$\textcircled{m2}$" at a middle agent p'_i of S_i which updates label to s'_t, and $p_{i+1'}$ changes label to s'_h as well. See a demonstration in Figure 12. Consequently, S_i is temporarily divided into two halves S^1_i and S^2_i labelled:

$$(\ldots, \overbrace{s_h, \ldots, s, \ldots, s'_t}^{S^1_i}, \overbrace{s'_h, \ldots, s, \ldots, s_t}^{S^2_i}, \ldots).$$

Now, s'_t emits the "Ⓜ" mark back to l_h via transmission states, from p_i to p_{i-1}. Upon arrival of "Ⓜ", l_h invokes the sub-procedure (A) to begin collection on the first half S^1_i and $\mathsf{Push}(S)$ to move towards s'_t, after which l_h calls (A) again to travel into s_t.

We argue that the line L_i always has sufficient memory to store all collected arrows. The Manhattan distance will always be $\delta(l_t, s_t) > |L_i|$ if the segment S_i has at least one diagonal connection. Consider the worst-case scenario of a diagonal segment in which each agent p_i gains a local diagonal direction at a cost of two cardinal arrows. Recall that each agent can store two arrows in its state, in c_6 and c_7. Given that, in the worst-case the segment contains a total of $2|S_i|$ local arrows. Thus, by applying (A) twice in each half of S_i, each single arrow of S_i will find a room in L_i.

We now calculate the running time of the $\mathsf{CollectArrows}(L_i, S_i)$ procedure on a number of rounds. Starting from steps 1 and 2 of (A), the "$\leftrightarrows$" mark takes a journey from l_h to s_t requiring at most $t_1 = |L_i| + |S_i| = O(|L_i|)$ rounds. Then, the pipelined collection and rearrangement of arrows in steps 3-6, require at most a

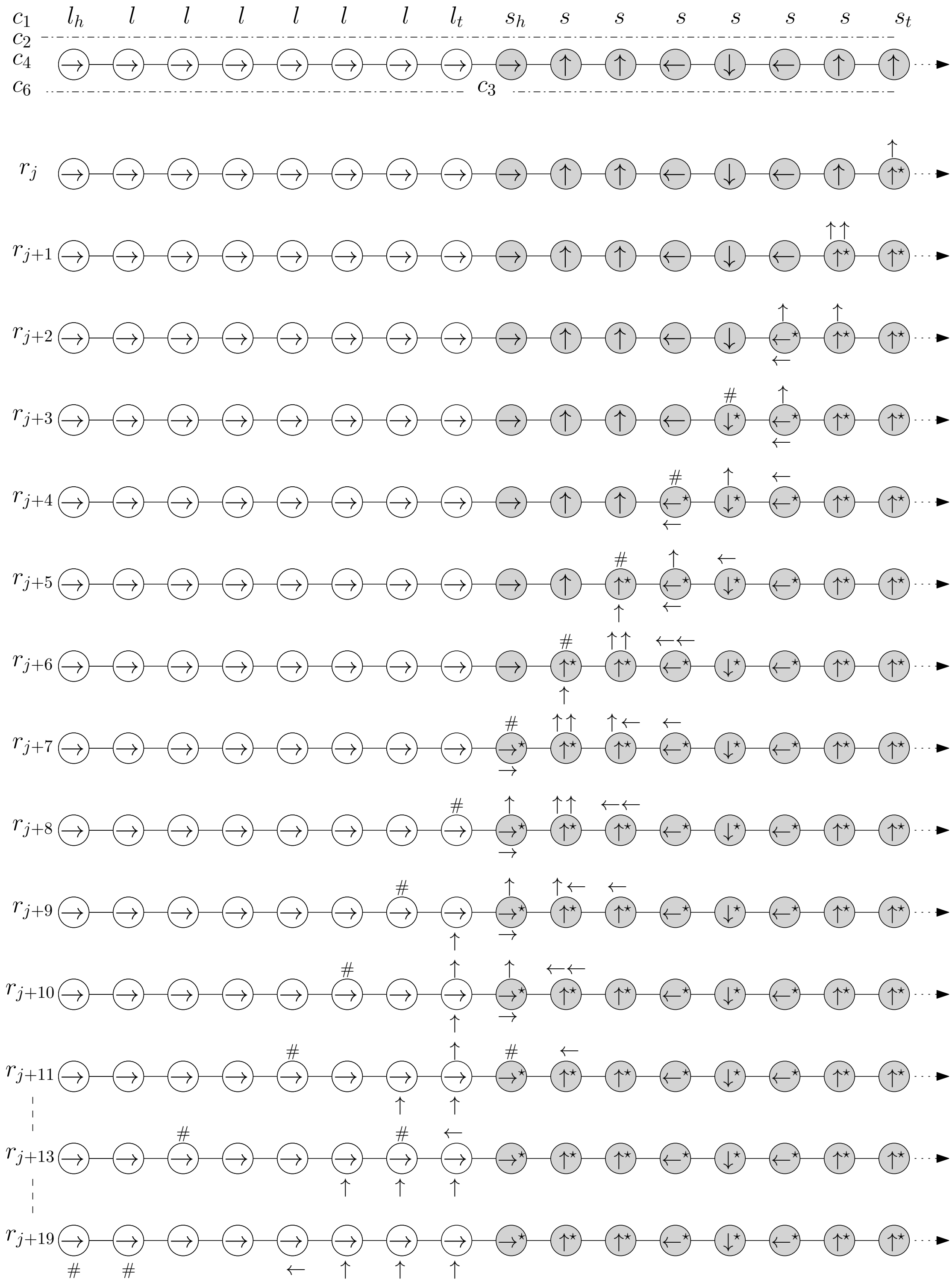


Figure 11: An implementation of the arrows collection on the shape in Figure 10, see text for explanation.

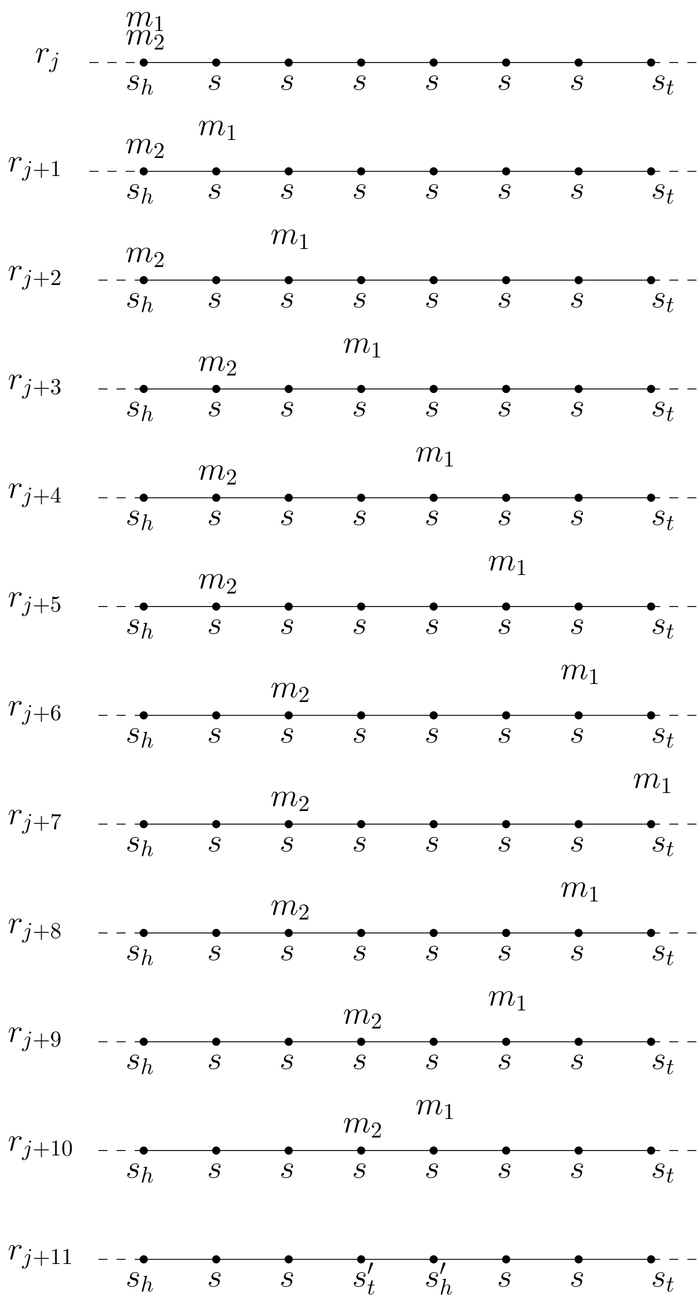


Figure 12: A fast “(m1)” and slow “(m2)” wave meeting at the middle of S_i of 8 agents. Observe that “(m1)” moves every round, while “(m2)” is three rounds slower.

number of parallel rounds equal asymptotically to the length of $|S_i| + |L_i|$, namely $t_2 = O(|L_i|)$. Moreover, the cost of “⇆✓” transmission takes time $t_3 = |L_i|$ rounds. In (B), the propagation of “Ⓜ” costs $t_4 = |L_i|$, another cost $t_5 = 3|S_i|$ is preserved for (1) and (2), which is the communication of fast “(m1)” and slow “(m2)” and the return of “Ⓜ” to the head, respectively. Hence, (A) costs at most $t_A = t_1 + t_2 + t_3 = O(|L_i|)$ parallel rounds of communication, whereas (B) requires at most $t_B = t_4 + t_5 = O(|L_i|)$. The same bound holds in the worst-case by applying (A) twice. Therefore, this procedure requires a total number of at most $T = 2t_A + t_B = O(|L_i|)$ parallel rounds to draw a route map. $\square$

Bereg *et al.* [9] provide a nice discussion on traversing Hamiltonian routes of a minimum number of turns, and Kranakis *et al.* [32] show this in a more general case. By exploiting their outcomes and Lemma 6, we now show that a route produced by the CollectArrows procedure can contain no more than 3 turns.

Lemma 7. *The CollectArrows procedure generates a route of at most 3 turns.*

Proof. Pick any two cells in the grid (e.g. l_t and s_t in Figure 7), then the length of the route with the shortest distance between these two cells is defined as the minimum number of straight line segments that make up a route that connects these two cells (consult [9] and [32] for full details). Instead, we can say it is the number of turns a line L_i must make when traversing from l_t to s_t via a minimum-turn-route linking the two cells. By the arrows collection of Lemma 6, L_i needs to take one turn (e.g, w.l.o.g, at either (x', y) or (x, y') in Figure 7) with at most two more turns at l_t and s_t for re-orientation, resulting in a three-turn route. □

Finally, ComputeDistance and CollectArrows procedures completes the DrawMap sub-phase. By Lemmas 5, 6, and 7, Lemma 8 concludes the following:

Lemma 8. *DrawMap completes within $O(|L_i|)$ rounds.*

3.4. Pushing the next segment S_i

Unlike all previous sub-phases, the transformation now allows individuals to perform line movements on the grid, taking advantage of their linear-strength pushing mechanism. That is, a straight line L_i of 2^i agents occupying a column or row of 2^i consecutive cells on the square grid can be pushed in a single step depending on its orientation in parallel vertically or horizontally in a single-time step. The line head and tail are responsible for pushing the line interchangeably during the transformation. Furthermore, L_i has the ability to change direction or turn from vertical to horizontal and vice versa.

A variety of challenges must be overcome in order to distribute the global coordination of line moves into a system of identical agents capable of only local vision and communication. One of the most essential challenges is timing: an individual agent moving the line must know when to start and stop pushing. Otherwise, it may disconnect the shape and break the connectivity-preservation requirement. Further, the line may change direction and turn around while pushing; hence, it must have some kind of local synchronisation over its agents to ensure that everyone follows the same route and no one is pushed off. Failure to do so may result in a loss of connectivity, communication, or the displacement of other agents in the configuration. Moreover, pushing a line does not necessarily traverse through free space of a Hamiltonian shape; consequently, a line may walk along the remaining configuration of agents while ensuring global connectivity at the same time. However, we were able to address all of these concerns in Push, which will be detailed below.

After some communication, l_h observes that L_i is ready to move and can start Push now. It synchronises with l_t to guide line agents during pushing. To achieve this, it propagates fast "Ⓟ1" and slow "Ⓟ2" marks along the line, "Ⓟ1" is transmitted every round and "Ⓟ2" is three rounds slower (shown earlier in DrawMap). The "Ⓟ1" mark reflects at l_t and meets "Ⓟ2" at a middle agent p_i, which in turn propagates two pushing signals "Ⓟ" in either directions, one towards l_h and the other heading to l_t. This synchronisation liaises l_h with l_t throughout the pushing process, which starts immediately after "Ⓟ" reaches both ends of the line at the same time. It is developed to handle the case where L_i has an even or odd number of agents, as illustrated in Figures 13 and 14, respectively. This is discussed in detail in Lemma 9. Recall the route map has been drawn starting from l_t, and hence, l_t moves simultaneously with l_h according to a local map direction $\hat{a} \in A$ stored in its map component c_6.

Through this synchronisation, l_t checks the next cell (x, y) that L_i pushes towards and tells l_h, whether it is empty or occupied by an agent $p \notin L_i$ in the rest of the configuration. If (x, y) is empty, then l_h pushes L_i one step towards (x, y), and all line agents shift their map arrows in c_6 forwardly towards l_t. If (x, y) is occupied by $p \notin L_i$, then l_t swaps states with p and tells l_h to push one step. Similarly, in each round of pushing a line agent p_i swaps states with p until the line completely traverses the drawn route map and restores it to its original state. Figure 17 shows an example of pushing L_i through a route of empty and occupied cells. In this way, the line agents can transparently push through a route of any configuration and leave it unchanged. Once L_i has traversed completely through the route and lined up with s_t, then RecursiveCall begins. Algorithm 5 provides a general procedure of Push.

Algorithm 5: Push

```
Input: a straight line L_i and a segment S_i

The line head l_h observes the completion of DrawMap
repeat
    l_h emits a mark to l_t to start pushing // l_t sees empty or non-empty cell
    if c_6 = d_{l_t} point to empty cell // local arrow of l_t points to empty cell
     then
        l_h syncs L_i: update states and push one step
    end
    if c_6 = d_{l_t} point to non-empty cell k then
        l_t activates k
        l_h syncs L_i // swap and update states as described in text
        L_i pushes one step
    end
until l_h swaps labels with s_h
RecursiveCall begins
```

Agents synchronisation.

Many agent behaviours, including state swapping and line movements (parallel pushing), are realised to be very efficient in the centralised systems of a global coordinator. In contrast, the constraints in this model make these simple tasks difficult, as individuals with limited knowledge cannot keep track of others during the transformation. This may result in the disconnection of the whole shape, a modification in the rest of the configuration or even the loss of a chain of actions that halts the transformation process. However, the synchronisation of agents can assist to tackle such an issue where individuals can organise themselves to eventually arrive at a state in which all of them conduct tasks concurrently. This concept is similar to a well-known problem in cellular automata known as the firing squad synchronisation problem, which was proposed by Myhill in 1957. McCarthy and Minsky provided a first solution to this problem [37]. In the following, Lemma 9 demonstrates how their solution can be translated to our model to synchronise agents contained in the same segment to perform concurrent actions in linear time.

Lemma 9 (Agents synchronisation)**.** *Let P denote a Hamiltonian path of n agents on the square grid, starting from a head p_1 and ending at a tail p_n, where $p_1 \neq p_n$. Then, all agents of P can be synchronised in at most $O(n)$ rounds.*

Proof. We show how to adopt the synchronisation process of [37] generally to the case when n is not a power of two. The strategy consists of two cases, even and odd number of agents. For the even case, the head p_1 emits fast mark "ⓜ1" and slow mark "ⓜ2" towards the tail p_n. The "ⓜ1" mark is communicated from p_i to p_{i+1} via transmission components in each round, while is transmitted from p_i to p_{i+1} every three rounds. When "ⓜ1" reaches the other end of the path p_n, it returns to p_1. Thus, the two marks collide exactly in the middle (see an example in Figure 12). Now, the two agents who witness the collision update to a special state, which will effectively split P into two sub-paths. Both agents repeat the same procedure in each half of length $n/2$ in either direction of P. Repeat this halving until all agents reach a special state (collision witness) in which they all perform an action simultaneously. An implementation of this synchronisation is depicted in Figure 13.

For the odd case, p_1 emits "ⓟ1" and "ⓟ2" where both marks meet in a slightly different way, at an exact single middle agent p_i on P. This agent p_i observes a predecessor p_{i-1} showing "ⓜ2" and successor p_{i+1} showing "ⓜ1" in transmission state and responds by switching into another special state that allows it to

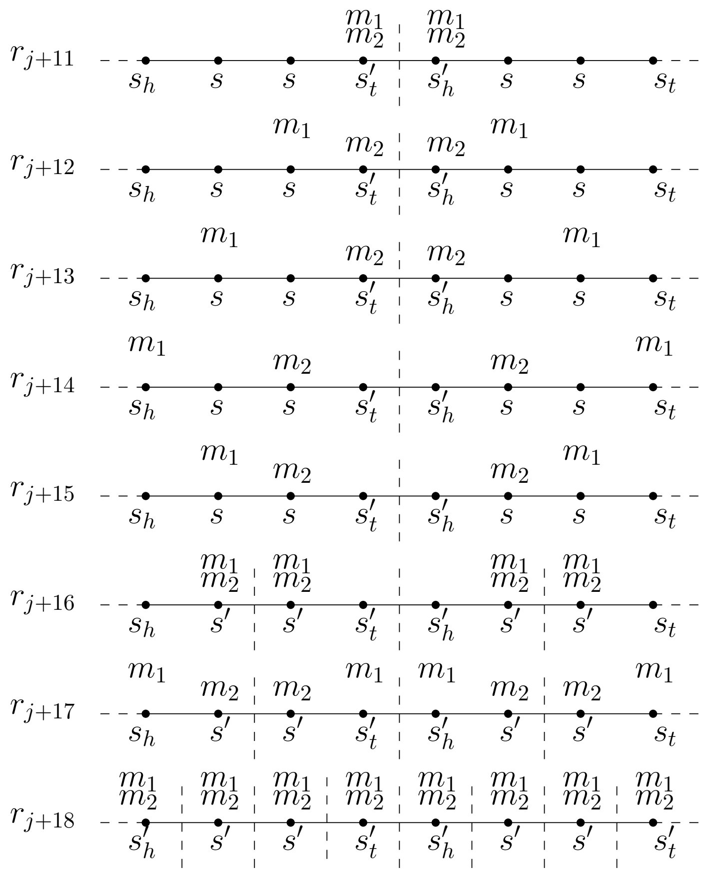


Figure 13: Synchronising 8 agents that were started in Figure 12 where the halving procedure repeats until all agents reach a synchronised state.

play two roles. That is, it emits "$\textcircled{p1}$" and "$\textcircled{p2}$" to both directions of P, this effectively splits P into two sub-paths of length $n/2 - 1$ each. Now, repeat the process in each half until the two marks intersect in the middle, at which point two agents notice the collision and change to a special state. In the same way, divide until all agents have updated to a synchronised state. Figure 14 depicts the synchronisation in the odd case.

Now, we are ready to describe the state transitions. In the first round r_j, p_1 updates to $p_1.c_2 \leftarrow \textcircled{m1}$ and combines "$\textcircled{m2}$" with "w" in waiting state, $p_1.c_3 \leftarrow \textcircled{m2}w$. Next round r_{j+1}, p_1 updates state to $p_1.c_3 \leftarrow \textcircled{m2}$ and $p_1.c_2 \leftarrow \cdot$. In the third round r_{j+2}, p_1 updates transmission state to $p_1.c_2 \leftarrow \textcircled{m2}$. Whenever p_i notices: (1) p_{i-1} (or p_{i+1}) showing "$\textcircled{m1}$", p_i shifts transmission to $p_i.c_2 \leftarrow \textcircled{m1}$ and p_{i-1} o(r p_{i+1}) resets their transmission next round. (2) p_{i-1} (or p_{i+1}) showing "$\textcircled{m2}$", p_i updates waiting state to $p_i.c_3 \leftarrow \textcircled{m2}w$ and p_{i-1} (or p_{i+1}) resets their transmission next round. (3) p_{i+1} showing "$\textcircled{m1}$" and p_{i-1} presenting "$\textcircled{m2}$" (or vice versa), p_i updates to another special state and repeats (1). When both p_i and p_{i+1} are presenting "$\textcircled{m1}$" and "$\textcircled{m2}$", they update into a special state and repeat the procedure of p_1 in either directions. Repeat until all agents and their neighbours reach a special state where all are synchronised.

Let us now analyse the runtime of this synchronisation in a number of rounds. The fast mark "$\textcircled{m1}$" moves along P taking n rounds plus $n/2$ to walks back to the centre in a total of at most $3n/2$ rounds. The same bound applies to the slow mark "$\textcircled{m2}$" arriving and meeting "$\textcircled{m1}$" in the middle. The whole procedure is now repeated on the two halves of length $n/2$, each takes $3n/4$ rounds. This adds up to a total $\sum_{i=1}^{n} 3n/2^i = 3n/2 + 3n/4 + \ldots + 3n/2^n = 3n(1/2 + 1/4 + \ldots + 1/2^n) = 3n(1) = 3n$. Therefore, this synchronisation requires at most $O(n)$ rounds of communication. $\square$

Now, Lemma 10 shows that under this model, a number of consecutive agents forming a straight line

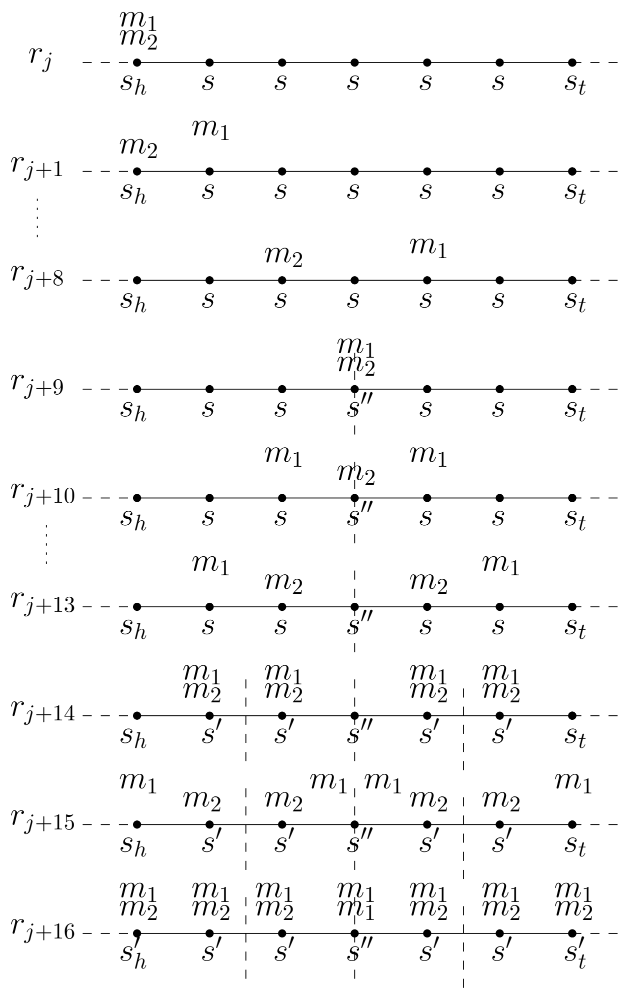


Figure 14: An example of synchronising 7 agents - odd case.

L_i can traverse through a route R of cells on the grid of any configuration C_R using their local knowledge, without breaking connectivity.

Lemma 10. *Let L_i denote a terminal straight line and R be a route of any configuration C_R, starting from a cell adjacent to the tail of L_i, where $R \leq 2|L_i| - 1$. Then, there exists a distributed way to push L_i along R without breaking connectivity.*

Proof. In Algorithm 5, the line head l_h observes the collection mark "⇆✓" indicating the completion of DrawMap(S), which draws a route R (see Definition 1). As a result, l_h emits the question mark "?" to l_t, which will broadcast via line agent transmission states from p_i to p_{i+1}. Once "?" arrives there, l_t checks whether its map arrow d_{l_t} points to an empty or occupied cell, and if the cell is empty, it emits a special mark "ⓨ" back to l_h indicating that a route is free to push. By an application of Lemma 9, l_h synchronises all line agents to reach a concurrent state in which the following occurs: (1) If the pushes are perpendicular, l_h pushes one round slower than l_t, i.e. l_t pushes one position perpendicular to L_i, then l_h pushes one position towards l_t. In this case, l_t pushes based on its map arrow c_6, updates state to $c_4 \leftarrow c_6$ and tells predecessor to turn next pushing. Otherwise, (2) l_h performs one inline push for L_i towards l_t, based on its local direction on push state c_7.

In general, (3) If p_i turns, it updates local direction $c_4 \leftarrow c_6$, and p_{i-1} updates push component $p_{i-1}.c_7 \leftarrow p_i.c_6$. (4) p_i of a present push component c_7 moves one step in the direction held in c_7, which then resets to $p_i.c_7 \leftarrow \cdot$. (5) All line agents shift local map direction forwardly towards l_t, $p_i.c_6 \leftarrow p_{i-1}.c_6$. Repeat these transitions until l_t encounters the segment tail s_t on the route through which l_t tells l_h to sync and push again, while l_t and s_t swaps their states. Hence, any p_i meets s_t, they swap states and resets their c_6. Eventually, l_h stops pushing once it meets and swaps states with s_t. An example is shown in Figure 15. During pushing through an L-shape route R, L_i may turn one or at most three times. In the following, we

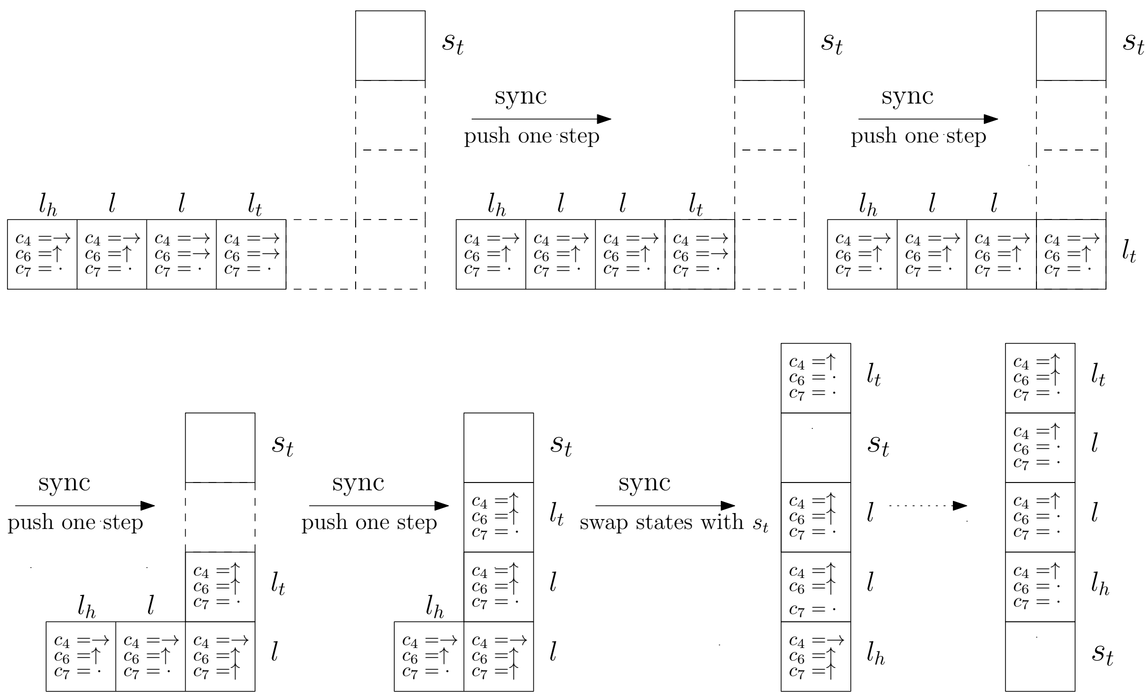


Figure 15: A line L_i of four agents inside grey cells pushing through a route of empty cells. All affected states (c_4, c_5 and c_7) are shown inside each occupied cell.

show that the number of turns depends on the orientation of both L_i and R. Without loss of generality, assume a horizontal L_i turning at a corner towards s_t, such as Figure 15 where L_i will temporarily divide into two perpendicular sub-lines while traversing to s_t. By an application of Lemma 9, both can be synchronised and organised to perform two parallel pushing where l_h liaises with l_t and push the two perpendicular sub-lines concurrently. Now, assume s_t is placed two cells above the middle of L_i, resulting in a route R of three turns along which L_i temporarily transforms into three perpendicular sub-lines. Three agents drive all other agents of the line to advance one step ahead on R. Therefore, the line can be synchronised to perform three sequential pushing operations that are asymptotically equivalent to the cost of one pushing, without breaking connectivity. The following are basic properties of line moves, called transparency properties and proven in [5], which are used to show that a line L_i can traverse through any route R of empty or/and occupied cells without breaking the configuration's overall connectivity, taking the same amount of moves asymptotically.

- No delay: L_i traverses R of any configuration C_R within the same asymptotic number of moves, regardless of how dense is C_R (density intuitively denotes a low-perimeter configuration).
- No effect: L_i restores all occupied cells to their original state and keeps C_R unchanged after traversing R.
- No break: L_i preserves connectivity while traversing along R.

We now provide transitions that show how L_i moves through R while satisfying all of the transparency properties of [5]. Assume L_i moves through a route R of non-empty cells in the configuration that are not on S_i and are occupied by other agents (denoted by k). When L_i walks through R and encounters k on R, l_t tells l_h to stop pushing. The agent k now acts as a tail and updates to a temporary state labelled $k_l t$. Based on the map arrow of l_t, it causes one of the following states to be triggered: (1) If the arrow points to an empty cell (x, y) in the same direction as L_i, then $k_l t$ emits a mark back to l_h to sync and push L_i one step further. Accordingly, $k_l t$ changes state to k_l, and each synchronised agent p_i shifts map arrow to p_{i+1}, see Figure 16. (2) If the arrow points to an empty cell (x, y) orthogonally to L_i, then $k_l t$ pushes one step perpendicularly and tells l_h to push L_i one step further. Hence, $k_l t$ changes state to k_c while each synchronised agent p_i shifts map arrow to p_{i+1}, as shown in Figure 17. During pushing, k_l swaps states with its predecessor and ensures that it remains in the same position until it meets l_h, at which k_l can update to its original state k.

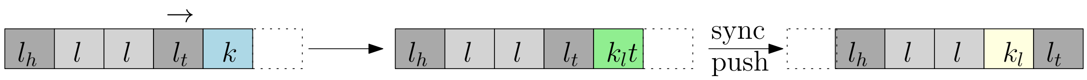


Figure 16: A line L_i of agents within grey cells (of labels l_h, l and l_t) pushing through a non-empty cell in blue (of label k) with a right map direction above l_t. In the first push, $k_l t$ changes to state k_l (yellow cell) in the first push, then k_l swaps states with its predecessor in each push until it meets l_h.

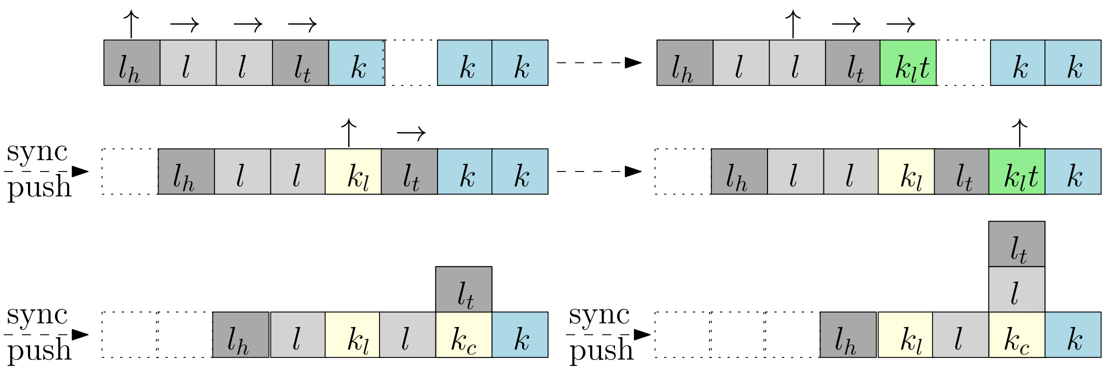


Figure 17: A line L_i of agents inside grey cells (of labels l_h, l and l_t), with map directions above, pushing and turning through empty and non-empty cells in blue (of label k). The green (labelled $k_l t$) and yellow cells (labelled k_l and k_c) show state swapping.

When L_i moves through a series of non-empty cells, it guarantees that they are neither separated nor disconnected while pushing. To achieve this, when l_t or $k_l t$ calls for synchronisation, any line agent p_i labelled l whose successor shows a label with star ($k_l^\star$ or $k_l^\star c$), both swap their states. It continues to swap states forwardly via consecutive non-empty cells until reaches the tail l_t or a line agent l. Though, when L_i traverses entirely through R and reaches the segment tail s_t, it may find another non-empty cell after swapping states with s_t. Hence, the same argument above still holds in this case. Figure 18 shows this case.

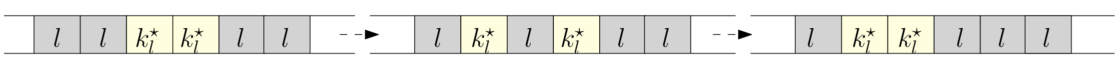


Figure 18: Four line agents of label l swap states inside grey cells swap states with others occupying consecutive yellow cells labelled $k_l^\star$.

Whenever L_i pushes into an empty cell (x, y), it fills (x, y) with an agent $p \in L_i$. During pushing, L_i always keeps the original position of a non-empty cell and restores it to its initial state (via state swapping) when leaving. However, there exists some cases that may break the connectivity. Consider a line L_i pushing along R and turning at a corner agent labelled $k_l c$, which has two diagonal neighbours where both are not adjacent to any line agent, as depicted in Figure 19 top. In this case, when $k_l c$ moves down, it will break connectivity with its upper diagonal neighbour. Hence, the transformation resolves this issue locally depending on the agents' local view. When $k_l c$ observes a pushing agent and has one or two diagonal neighbours, it temporarily switches to a state that allows it to move one step further while l_t updates into a turning agent. This also permits all line agents to turn sequentially until they reach the head l_h, which turns and waits for $k_l c$ to return to its initial cell. Figure 19 depicts how to handle this situation. Other orientations follow symmetrically by rotating the system $90°, 180°$ or $270°$ clockwise and counter-clockwise.

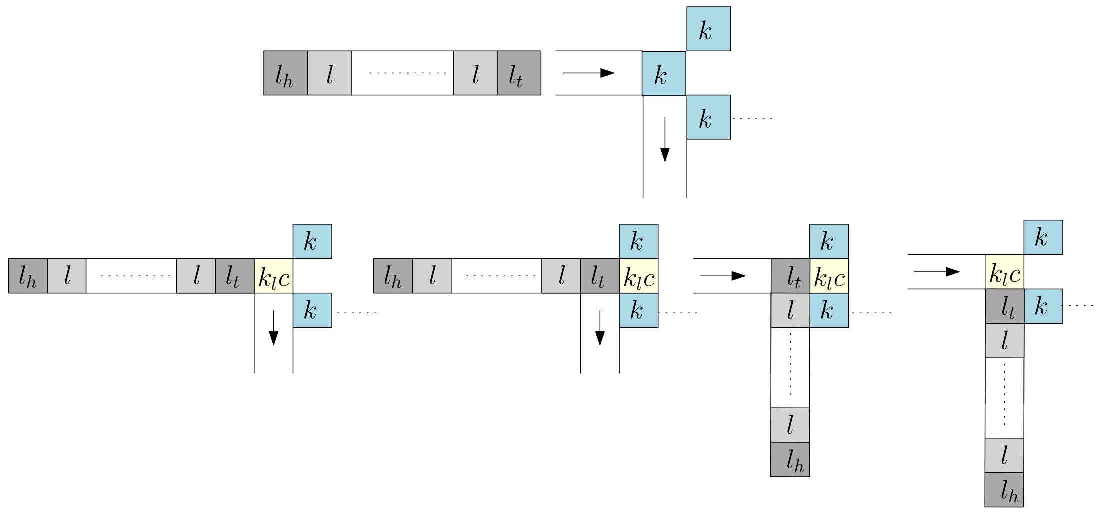
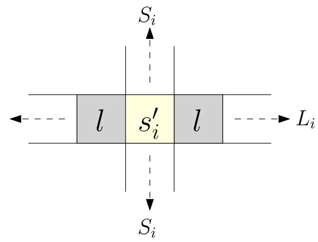


Figure 19: A line L_i (of labels l_h, l and l_t) pushes through a route R and turns at a corner agent labelled $k_l c$ (in yellow) that has two diagonal neighbours labelled k (in blue), neither of which is adjacent to any line agent.

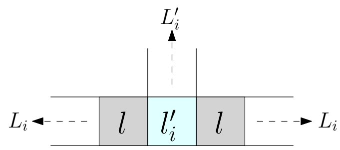
After traversing R, L_i is pushing towards its target position of $|L_i|$ consecutive cells adjacent to s_t. This position can be fully or partially occupied by other agents from the rest of the configuration or/and S_i, allowing us to distinguish between two distinct cases. First, if the agents labelled k are part of the rest of the configuration, L_i pushes them to where it replaces each labelled k agent with a line agent. All agents involved in this push are now in temporary states (i.e. they perform several tasks such as synchronization, activation, state swapping, and map arrow forwarding), implying that their original information is stored in the agents' states until a further call. When L_i leaves, it restores the agents again to their original cells/states at that position. This procedure is the same as that described above and illustrated in Figures 16 and 18. Observe that in the this case, the agents labelled k stay still until a later call for L_i to push is issued.

Second, we show what happens when some agents belong to S_i with label s_i occupy some cells in that target position. In this case, the agents labelled s_i will receive a recursive call later and therefore push ahead of L_i, potentially breaking connectivity. Hence, the transformation manages to resolve this depending on the local view of the agents. For ease of explanation, assume L_i pushes an agent of label s_i which shifts into temporary state s'_i, as mentioned above (e.g. Figure 20 (a)). Observe that s'_i cannot push until it eventually becomes a head of another line L'_i to which S_i is transformed, and s'_i switches to l'_h (e.g. Figure 20 (b)). When l'_h leaves, it observers (i.e. by its local view) that it has two diagonal neighbours of line agents labelled l, and hence stops pushing and calls that L_i be reconnected (e.g. Figure 20 (c)), similar to what we described previously in Figure 19. The head l_h then moves one step forward to rejoin L_i (e.g. Figure 20 (d)). When l'_h notices the reconnection (i.e. observes the three adjacent line agents), it can freely push L'_i, as depicted in Figure 20 (e).

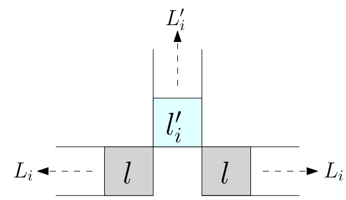
Thus, all agents involved in the push, whether they belong to L_i, S_i or the rest of configuration, are labelled and organised in such a way that can push through a route R of any configuration C_R towards an empty or occupied target position while preserving the connectivity. It implies that L_i remains connected when travelling as well as the whole configuration. Further, the original state of C_R and the target position have been restored and all of its occupied cells (if any) have been left unchanged. As a result, the algorithm meets all of the transparency criteria of line moves in [5]. □



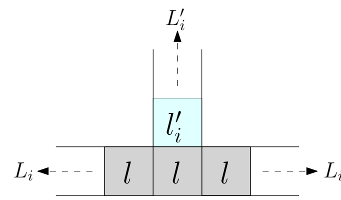
(a) s_i shifts into s'_i.



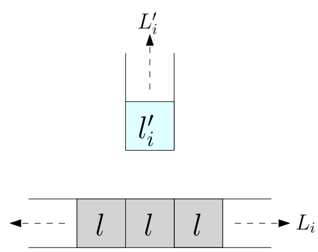
(b) s'_i becomes a head l'_h.



(c) l'_h stops pushing and calls a reconnection to L_i.



(d) l'_h has three neighbours of line agents.



(e) l'_h freely pushes L'_i.

Figure 20: A line pushing through an occupied target position. Other orientations follow symmetrically by rotating the system $90°, 180°$ or $270°$ clockwise and counter-clockwise.

By Lemmas 9 and 10, Lemma 11 shows the complexity of Push based on the number of line moves and communication rounds.

Lemma 11. *A straight line L_i requires at most $O(|L_i|)$ line moves and $O(|L_i| \cdot |R|)$ rounds to pass through a route R of any configuration C_R.*

Proof. The bound of moves depends on three factors, the number of empty cells on R, the length of L_i and the number of turns on R. Say that R is free of agents (fully empty) and has at most 3 turns, then L_i requires at most $|L_i|+3|L_i|+|L_i| = 5|L_i| = O(|L_i|)$ moves (proved in Lemma 10) to push through R. However, if R is partially or fully occupied, the communication cost may be quite high, as individuals need to perform many functions such as synchronisation, activation, state swapping, and map arrow forwarding. Those actions can be carried either sequentially or concurrently during the transformation and can be analysed independently of each other. We establish an upper bound on the extreme case below, so this should be more efficient for any other configurations of R, whether empty or partially occupied.

Assume that R is completely occupied by other agents in the shape (in a worst-case), from the cell adjacent to the line tail l_t to the cell adjacent to s_h. Then, l_t needs to traverse over at most $|R|$ agents in order to arrive at s_h, which costs $t_1^c = |R|$ rounds. Further, l_t requires a number of synchronisations equal to $|L_i|$ to move all line agents along R at a cost of no more than $t_2^c = |L_i| \cdot |R|$ rounds. In each synchronisation, a line agent swaps its state with $|R|$ agents and forwards its map direction over line agents to l_t within at most $t_3^c = |L_i| + |R|$ rounds. Thus, this sub-phase results in a maximum number of communications $T^c = t_1^c + t_2^c + t_3^c = |R| + (|L_i| \cdot |R|) + (|L_i| + |R|) = O(L_i| \cdot |R|)$ rounds. This bound holds when other agents occupy $|L_i|$ consecutive horizontal and vertical cells beyond s_h. □

3.5. Recursive call on the segment S_i into a line L'_i

This sub-phase, RecursiveCall, is the heart of this transformation and is recursively called on the next segment S_i, which eventually transforms into another straight line L'_i of 2^i agents.

When a segment tail s_t swaps states with l_h, it accordingly acts as follows: (1) propagates a special mark transmitted along all segment agents towards the head s_h, (2) deactivates itself by updating label to $c_1 \leftarrow k$, (3) resets all of its components, except local direction in c_4. Similarly, once a segment agent p_i observes this special mark, it propagates it to its successor p_{i+1}, deactivates itself, and keeps its local direction in c_4 while resetting all other components. When the segment head s_h notices this special mark, it changes to a line head state ($c_1 \leftarrow l_h$) and then recursively repeats the whole transformation from round 1 to $i-1$. Figure 21 presents a graphical illustration of RecursiveCall applied on a diagonal line shape.

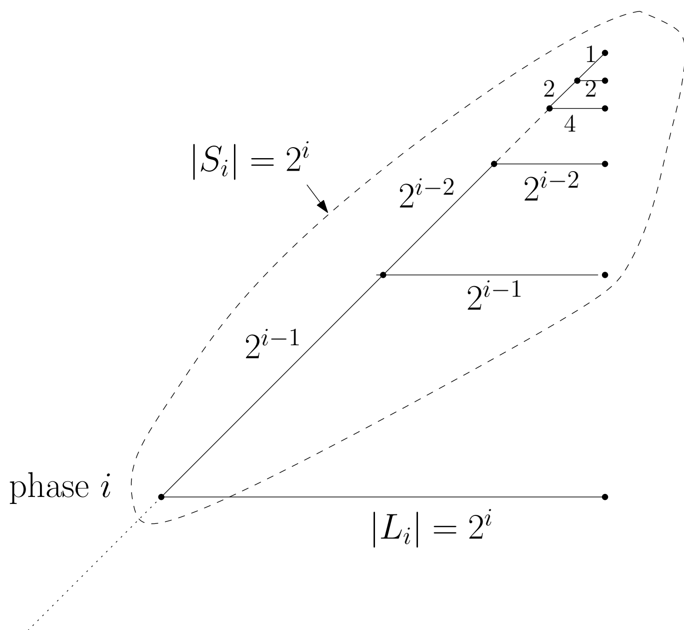


Figure 21: A zoomed-in picture of the core recursive technique RecursiveCall in Figure 2(c).

3.6. Merging the two lines L_i and L'_i

The final sub-phase of this transformation is Merge, which combines two straight lines into a single double-sized line, described as follows. The previous sub-phase, RecursiveCall, transforms the segment S_i into a straight line L'_i, starts from a head l'_h and ends at a tail l'_t (note that these labels are just for ease of explanation). Now, L_i's tail detects L'_i's tail, recognising that L'_i is the successor of L_i, and sends a signal to l_h, to start merging. Without loss of generality, say L'_i's tail occupies cell (x, y) and L_i is on positions $(x+1, y), \ldots, (x+|L_i|-1, y)$. Now, L'_i is either (1) perpendicular on $(x, y), \ldots, (x, y+|L_i|-1)$ or (2) in line with L_i occupying $(x, y), \ldots, (x-|L_i|-1, y)$. In (1), l_h emits a mark to the other head l'_h, asking to change the direction and merge with L_i to form a straight line L_{i+1} of double length, defining one head and tail for L_{i+1}. In (2), L'_i and L_i have already formed L_{i+1}; all that remains is to update labels for L_{i+1}.

Now, it is sufficient to upper bound this sub-phase by analysing only a worst-case of (1). Obviously, the straight line L'_i pushes and turns via a route R' within a distance equal to its length in order to line up with L_i. Observe that this push may go through empty or/and occupied cells. Thus, with each push, L'_i must sync its agents to perform additional actions, e.g. activation and labels exchange, as well as to alternate the pushes between the head and tail of L'_i (i.e. there is at most only one push happening at each round).

By that, the total number of moves is at most $O(|L'_i|)$, while the communication costs by at most $O(|L'_i|^2)$ rounds required for synchronisations.

Therefore, all agents in Merge communicate in linear time, and then Lemma 12 formally states:

Lemma 12. *An execution of Merge requires at most $O(|L_i|)$ line moves and $O(|L_i|^2)$ rounds of communication.*

Finally, we analyse the recursion in a worst-case shape in which individuals consume their maximum energy to communicate and move. The runtime is based on the analysis of the centralised version that has been proved in [5]. Let T_i^c and T_i^m denote the total number of communication rounds and moves in phase i, respectively, for all $i \in 1, \dots, \log n$. Apart from RecursiveCall, the 2^i agents forming a straight line L_i in phase i go through DefineSeg, CheckSeg, DrawMap, Push and Merge sub-phases that take total parallel rounds of communication t_i^c at most:

$$\begin{aligned} t_i^c &= (3 \cdot |L_i|) + (|L'_i| \cdot |L'_i|) + (|L_i| \cdot |R|) \\ &= O(|L_i| \cdot |L_i|). \end{aligned}$$

Then, in Push and Merge sub-phases, the line L_i traverses along a route of total movements t_i^m in at most:

$$t_i^m = |L_i| + |L'_i| = O(|L_i|).$$

Now, let T_{i-1}^c denote a total number of parallel rounds required for RecursiveCall on 2^i agents of the segment S_i, which transforms into another straight line L'_i. Given $|L_i| = 2^i$, this recursion in phase i costs a total rounds bounded by:

$$\begin{aligned} &T_i^c \leq i \cdot (|L_i| \cdot |L_i|) \leq i \cdot (2^i)^2 \\ &T_i^c O(\leq i \cdot n^2). \end{aligned}$$

Thus, we conclude that the call of RecursiveCall in the final phase $i = \log n$ requires a total rounds $T_{\log n}^c$:

$$\begin{aligned} T_{\log n}^c &\leq n^2 \cdot \log n \\ &= O(n^2 \log n). \end{aligned}$$

The same argument follows on the total number of movements T_{i-1}^c for a recursive call of RecursiveCall, which costs at most:

$$\begin{aligned} &T_i^m \leq i \cdot |L_i| \leq i \cdot (2^i) \\ &T_i^m \leq O(\leq i \cdot n). \end{aligned}$$

Finally, by the final phase $i = \log n$, all agents in the system pushes a total number of moves $T_{\log n}^m$ that bounded by:

$$\begin{aligned} T_{\log n}^m &\leq n \cdot \log n \\ &= O(n \log n). \end{aligned}$$

Overall, given a Hamiltonian path in an initial connected shape S_I of individuals of limited knowledge and permissible line moves, the following lemma states that S_I can be transformed into a straight line S_L in a number of moves that match the optimal centralised transformation satisfying the condition of preserving connectivity.

By Lemmas 1 through 12, it is implied that:

Theorem 1. *The* HamiltonianLine *problem can be solved within at most $O(n \log_2 n)$ line moves and $O(n^2 \log_2 n)$ rounds.*

4. Conclusions

In this work, we presented a distributed algorithmic framework for line moves on a two-dimensional square grid. In this model, the system consists of computationally limited individuals, each of which has constant memory can only observe the states of nearby agents in a Moore neighbourhood. Those individuals perform the LCM cycles through a set of rules and interactions, similar to finite state automata. Our major contribution, building upon our algorithmic investigations of centralised transformations [5], is then the first distributed connectivity-preserving transformation that exploits line moves and can work for all connected shapes that belong to the family of Hamiltonian shapes. This algorithm solves the line formation problem within a total of at most $O(n \log n)$ moves, which is asymptotically equivalent to that of the best-known centralised transformations.

The proposed approach opened a number of interesting research problems. There is still a chance to distribute the inverse transformation, i.e. transform a line into any Hamiltonian shape while preserving connectivity throughout the transformation. If achieved, it is expected to contribute to the development of more general transformations, hopefully within the same asymptotic bound of $O(n \log n)$ line moves. However, the inverse transformation (based on the current distributed setting) appears to be harder than it is in the centralised case (in the centralised it immediately follows by reversibility), and the agents need to somehow know an encoding of the shape to be constructed. Thus, it may be required to make some changes to the model in order to develop distributed counterparts. Further, a number of alternative types of grids have been considered in the relevant literature, e.g. triangular and hexagonal, and it would be interesting to investigate how our results translate there. Another direction is to extend the transformations to work on a three-dimensional grid (e.g. some of the ideas in [52] might prove useful for this extension). Finally, in this paper we assume that agents share a sense of orientation. Another challenge both in centralised and distributed models would be to consider agents which are operating on non-orientated grid, see [33, 39].

Acknowledgements

We would like to thank the anonymous reviewers of this work and its preliminary versions for their thorough reading and comments, which helped us significantly improve our work.

References

[1] L. M. Adleman, Molecular computation of solutions to combinatorial problems, Science 266 (5187) (1994) 1021–1024.

[2] H. A. Akitaya, E. M. Arkin, M. Damian, E. D. Demaine, V. Dujmović, R. Flatland, M. Korman, B. Palop, I. Parada, A. van Renssen, et al., Universal reconfiguration of facet-connected modular robots by pivots: the $O(1)$ musketeers, Algorithmica 83 (5) (2021) 1316–1351.

[3] A. Almethen, O. Michail, I. Potapov, Pushing lines helps: Efficient universal centralised transformations for programmable matter, Theoretical Computer Science 830-831 (2020) 43 – 59.

[4] A. Almethen, O. Michail, I. Potapov, Distributed transformations of Hamiltonian shapes based on line moves, in: Algorithms for Sensor Systems - 17th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS, Vol. 12961, Springer, 2021, pp. 1–16.

[5] A. Almethen, O. Michail, I. Potapov, On efficient connectivity-preserving transformations in a grid, Theoretical Computer Science, 2021.

[6] G. Aloupis, N. Benbernou, M. Damian, E. Demaine, R. Flatland, J. Iacono, S. Wuhrer, Efficient reconfiguration of lattice-based modular robots, Computational geometry 46 (8) (2013) 917–928.

[7] G. Aloupis, S. Collette, E. Demaine, S. Langerman, V. Sacristán, S. Wuhrer, Reconfiguration of cube-style modular robots using O(logn) parallel moves, in: International Symposium on Algorithms and Computation, Springer, 2008, pp. 342–353.

[8] D. Angluin, J. Aspnes, Z. Diamadi, M. Fischer, R. Peralta, Computation in networks of passively mobile finite-state sensors, Distributed Computing 18 (4) (2006) 235–253.

[9] S. Bereg, P. Bose, A. Dumitrescu, F. Hurtado, P. Valtr, Traversing a set of points with a minimum number of turns, Discrete & Computational Geometry 41 (4) (2009) 513–532.

[10] D. Boneh, C. Dunworth, R. J. Lipton, J. Sgall, On the computational power of DNA, Discrete Applied Mathematics 71 (1-3) (1996) 79–94.

[11] J. Bourgeois, S. Goldstein, Distributed intelligent MEMS: progresses and perspective, IEEE Systems Journal 9 (3) (2015) 1057–1068.

[12] M. Connor, O. Michail, I. Potapov, Centralised connectivity-preserving transformations for programmable matter: a minimal seed approach, in: International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS), Springer, 2021, pp. 45–60.

[13] J. Czyzowicz, D. Dereniowski, A. Pelc, Building a nest by an automaton, Algorithmica 83 (1) (2021) 144–176.

[14] J. Daymude, Z. Derakhshandeh, R. Gmyr, A. Porter, A. Richa, C. Scheideler, T. Strothmann, On the runtime of universal coating for programmable matter, Natural Computing 17 (1) (2018) 81–96.

[15] M. D'Emidio, G. Di Stefano, D. Frigioni, A. Navarra, Characterizing the computational power of mobile robots on graphs and implications for the euclidean plane, Information and Computation 263 (2018) 57–74.

[16] Z. Derakhshandeh, R. Gmyr, A. Richa, C. Scheideler, T. Strothmann, Universal shape formation for programmable matter, in: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2016, pp. 289–299.

[17] Z. Derakhshandeh, R. Gmyr, T. Strothmann, R. Bazzi, A. W. Richa, C. Scheideler, Leader election and shape formation with self-organizing programmable matter, in: International Workshop on DNA-Based Computers, Springer, 2015, pp. 117–132.

[18] G. A. Di Luna, P. Flocchini, N. Santoro, G. Viglietta, Y. Yamauchi, Shape formation by programmable particles, Distributed Computing.

[19] D. Doty, Theory of algorithmic self-assembly, Communications of the ACM 55 (2012) 78–88.

[20] S. Douglas, H. Dietz, T. Liedl, B. Högberg, F. Graf, W. Shih, Self-assembly of DNA into nanoscale three-dimensional shapes, Nature 459 (7245) (2009) 414.

[21] A. Dumitrescu, J. Pach, Pushing squares around, in: Proceedings of the twentieth annual symposium on Computational geometry, ACM, 2004, pp. 116–123.

[22] A. Dumitrescu, I. Suzuki, M. Yamashita, Formations for fast locomotion of metamorphic robotic systems, The International Journal of Robotics Research 23 (6) (2004) 583–593.

[23] A. Dumitrescu, I. Suzuki, M. Yamashita, Motion planning for metamorphic systems: Feasibility, decidability, and distributed reconfiguration, IEEE Transactions on Robotics and Automation 20 (3) (2004) 409–418.

[24] S. P. Fekete, R. Gmyr, S. Hugo, P. Keldenich, C. Scheffer, A. Schmidt, Cadbots: Algorithmic aspects of manipulating programmable matter with finite automata, Algorithmica 83 (1) (2021) 387–412.

[25] P. Flocchini, G. Prencipe, N. Santoro, Distributed computing by mobile entities, Current Research in Moving and Computing 11340.

[26] N. Fujinaga, Y. Yamauchi, H. Ono, S. Kijima, M. Yamashita, Pattern formation by oblivious asynchronous mobile robots, SIAM Journal on Computing 44 (3) (2015) 740–785.

[27] T. Fukuda, Self organizing robots based on cell structures-cebot, in: Proc. IEEE Int. Workshop on Intelligent Robots and Systems (IROS'88), 1988, pp. 145–150.

[28] K. Gilpin, A. Knaian, D. Rus, Robot pebbles: One centimeter modules for programmable matter through self-disassembly, in: Robotics and Automation (ICRA), 2010 IEEE International Conference on, IEEE, 2010, pp. 2485–2492.

[29] R. Gmyr, K. Hinnenthal, I. Kostitsyna, F. Kuhn, D. Rudolph, C. Scheideler, T. Strothmann, Forming tile shapes with simple robots, Natural Computing 19 (2020) 375–390.

[30] A. Itai, C. Papadimitriou, J. Szwarcfiter, Hamilton paths in grid graphs, SIAM Journal on Computing 11 (4) (1982) 676–686.

[31] A. Knaian, K. Cheung, M. Lobovsky, A. Oines, P. Schmidt-Neilsen, N. Gershenfeld, The milli-motein: A self-folding chain of programmable matter with a one centimeter module pitch, in: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, 2012, pp. 1447–1453.

[32] E. Kranakis, D. Krizanc, L. Meertens, Link length of rectilinear hamiltonian tours in grids, Ars Combinatoria 38 (1994) 177–192.

[33] R. Martin, T. Nickson, I. Potapov, Geometric computations by broadcasting automata, Nat. Comput. 11 (4) (2012) 623–635.

[34] O. Michail, G. Skretas, P. Spirakis, On the transformation capability of feasible mechanisms for programmable matter, Journal of Computer and System Sciences 102 (2019) 18–39.

[35] O. Michail, P. Spirakis, Simple and efficient local codes for distributed stable network construction, Distributed Computing 29 (3) (2016) 207–237.

[36] O. Michail, P. Spirakis, Elements of the theory of dynamic networks, Commun. ACM 61 (2) (2018) 72–81.

[37] M. L. Minsky, Computation, Prentice-Hall Englewood Cliffs, 1967.

[38] A. Nguyen, L. Guibas, M. Yim, Controlled module density helps reconfiguration planning, in: Proc. of 4th International Workshop on Algorithmic Foundations of Robotics, 2000, pp. 23–36.

[39] T. Nickson, I. Potapov, Broadcasting Automata and Patterns on $\mathbb{Z}^2$, Springer International Publishing, Cham, 2015, pp. 297–340.

[40] V. P. Prakash, C. Patvardhan, A. Srivastav, Effective heuristics for the bi-objective euclidean bounded diameter minimum spanning tree problem, in: International Conference on Next Generation Computing Technologies, Springer, 2017, pp. 580–589.

[41] P. Rothemund, Folding DNA to create nanoscale shapes and patterns, Nature 440 (7082) (2006) 297–302.

[42] P. Rothemund, E. Winfree, The program-size complexity of self-assembled squares, in: Proceedings of the 32nd annual ACM symposium on Theory of computing (STOC), ACM, 2000, pp. 459–468.

[43] M. Rubenstein, A. Cornejo, R. Nagpal, Programmable self-assembly in a thousand-robot swarm, Science 345 (6198) (2014) 795–799.

[44] I. Suzuki, M. Yamashita, Distributed anonymous mobile robots: Formation of geometric patterns, SIAM Journal on Computing 28 (4) (1999) 1347–1363.

[45] T. Toffoli, N. Margolus, Programmable matter: concepts and realization, Physica. D, Nonlinear phenomena 47 (1-2) (1991) 263–272.

[46] C. Umans, W. Lenhart, Hamiltonian cycles in solid grid graphs, in: Proceedings 38th Annual Symposium on Foundations of Computer Science, IEEE, 1997, pp. 496–505.

[47] J. Walter, J. Welch, N. Amato, Distributed reconfiguration of metamorphic robot chains, Distributed Computing 17 (2) (2004) 171–189.

[48] E. Winfree, Algorithmic self-assembly of DNA, Ph.D. thesis, California Institute of Technology (1998).

[49] D. Woods, H. Chen, S. Goodfriend, N. Dabby, E. Winfree, P. Yin, Active self-assembly of algorithmic shapes and patterns in polylogarithmic time, in: Proceedings of the 4th conference on Innovations in Theoretical Computer Science, ACM, 2013, pp. 353–354.

[50] M. Yamashita, T. Kameda, Computing on anonymous networks. I. characterizing the solvable cases, IEEE Transactions on parallel and distributed systems 7 (1) (1996) 69–89.

[51] M. Yamashita, I. Suzuki, Characterizing geometric patterns formable by oblivious anonymous mobile robots, Theoretical Computer Science 411 (26-28) (2010) 2433–2453.

[52] Y. Yamauchi, Symmetry of anonymous robots, in: Distributed Computing by Mobile Entities, Springer, 2019, pp. 109–133.

[53] M. Yim, W. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, G. Chirikjian, Modular self-reconfigurable robot systems [grand challenges of robotics], IEEE Robotics & Automation Magazine 14 (1) (2007) 43–52.