UNIVERSITY OF
LIVERPOOL

Centralised and Distributed Shape Formation with a Linear-Strength Model

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by

Abdullah Abdurhman Almethen

November 2021

Abstract

This thesis is in the area of algorithmic robotic systems, where the main objective is to investigate the fundamental possibilities and limitations of different centralised and distributed algorithms related to *shape formation*, including the domain of small-scale robotics, such as programmable matter. It aims to combine current formalisms and develop a new theoretical model that is expected to promote the hardware and practical achievements while also enhancing the deployment and implementation of energy-efficient systems. It consists of three main parts.

First, we introduced a new linear-strength model of a discrete system of entities residing on a two-dimensional square grid. Each entity is modelled as a node occupying a distinct cell, and the set of all n nodes forms initially a connected shape A on the grid. Entities are equipped with a linear-strength pushing mechanism that can push a whole line of entities in parallel in a single time-step on one position in a given (one of the four possible) direction of a grid. A target connected shape B is also provided and the goal is to *transform* A into B via a sequence of line moves. Existing models of individual movements – such as rotating or sliding a single node – can be shown to be special cases of the present model, therefore their (inefficient, quadratic time) *universal transformations* carry over. We present an efficient *centralised* algorithmic framework that can universally transform any pairs of shapes in sub-quadratic worst-case transformations.

The second part focuses on designing *centralised* transformations aiming at *minimising the total number of moves* subject to the constraint of *preserving connectivity* of the shape throughout the transformation. That is, in each intermediate configuration we always guarantee that an *associated graph* induced by the occupied nodes is connected. We introduce very fast connectivity-preserving transformations for the case in which the associated graphs of A and B contain a Hamiltonian path. In particular, our transformation completes in a number of moves, which is asymptotically equal to the best known running time of connectivity-breaking transformations. Our most general result is then a connectivity-preserving universal transformation that can transform A into B (and B into A), through a sequence of sub-quadratic moves in a worst-case transformation.

The last part establishes a corresponding distributed model where nodes are simple indistinguishable devices called *agents*, which act as finite-state automata. Each gent has constant memory and can observe the states of nearby agents in a Moore neighbourhood. Our main contribution is the first distributed connectivity-preserving transformation that exploits line moves within sub-quadratic moves, which is asymptotically equivalent to that of the best-known centralised transformations. The algorithm solves the *line formation problem* that allows agents to form a final straight line from any shape, whose associated graph contains a Hamiltonian path.

Acknowledgements

By the grace and mercy of Allah (God), I have arrived at the end of my PhD journey, which has been one of the most exciting and long adventures in my life, and now I would like to express my gratitude to everyone who helped me during this journey.

First, and foremost, I would like to express my deep feelings of gratitude and appreciation to my supervisors Professor Igor Potapov and Dr Othon Michail. I am indebted to all of their invaluable support, guidance and knowledge. I am also grateful for their encouragement and advice; it has been a privilege to have worked with them over the past four years. This thesis would not be possible without them.

I would also like to thank my academic advisors Professor Prudence Wong, Dr Russell Martin and Dr Konstantinos Tsakalidis for their useful feedback and research suggestions.

I am grateful to Professor Leszek Gąsieniec and Dr Tom Friedetzky for agreeing to act as examiners of my thesis and for the time they spent on reviewing it.

My deep gratitude goes to my parents, Nora and Abdurahman, for their endless love, devotion, also for encouraging me to follow this path and supporting me along the way, and for everything they have done for me. To my siblings; their unlimited support has been invaluable, in particular Dr Mazen for his special assistance. To my grandfather, Abdullah, and uncle, Ali, for their spiritual support.

The special warmest gratitude goes to my wonderful wife, Basma, for standing by me and dealing with hardships and tribulations over a challenging time. I have no words to express my appreciation of her patience, dedication and exceptional efforts. To my kids, Judi and Abdulrahman, for bringing happiness into our life.

Last but not least, I would like to extend my appreciation to Qassim University and the Saudi Ministry of Education for sponsoring my research, as well as the Saudi Cultural Bureau in London for supporting me throughout the course of my postgraduate studies.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Overview and Motivation

In the 1980s, a seven-year-old boy was enthralled with a Japanese anime series revolving around a super robot constructed of fictional metal parts and capable of morphing into various shapes. One day in school, during science class, the boy explained the fascinating idea of this robot to his classmates, who were astonished and said together, 'This is never going to happen'. The teacher commented, smiling, that science fiction has long served as a source of inspiration for the scientific research community, paving the way for great real-life inventions that have been enormously successful and impactful. A prominent example is Jules Verne's 1870 adventure novel *Twenty Thousand Leagues Under the Seas* [115], which later served as inspiration for the invention of the *submarine*. Another example is the concept of *geostationary orbits* in communication, which was previously envisioned by Arthur C. Clarke in his 1945 paper 'Extra-Terrestrial Relays – Can Rocket Stations Give Worldwide Radio Coverage?' [38]. Thus, the possibility should not be excluded that creations of science fiction TV shows, such as the communicator in *Star Trek*, may become a tangible reality that we live in the foreseeable future.[1]

Since the invention of computers, scientists and engineers from different disciplines have made great joint efforts to build emerging innovations and sophisticated technologies to improve the quality of human life. One of their ultimate goals is to develop robotic systems consisting of machines that can efficiently change their shape by coordinating connectivity among them in order to perform complex tasks, adapt to new surroundings

[1] *It happened! The communicator inspired the development of mobile phones.*

and environments, and tolerate any damage. For instance, the components of such a robotic system might be arranged into a spanning line shape to pass through a narrow canal, bridge, pipe or corridor in a mine. They may then reconfigure into a starlike shape to navigate rough terrain or a spherical shape in which they pivot on themselves to roll swiftly over an even surface. In another domain, those system parts may need to create a fairly static object – such as a barrier, dam or shield – or perform complicated missions or heavy-duty tasks, such as coating an object in space.

A variety of sophisticated robotic systems are now within reach due to recent advances in components from large to nano scales, including sensors, controllers, electronics and electromechanical actuators with which individuals can change their positions in a medium and move relative to each other. Through a simple set of rules and local actions, the system components can carry out tasks that are well beyond an individual's capabilities. The implementation of a robotic system indicates whether the individuals are operated centrally or self-operated through local decentralised control. In *centralised* systems, there is an external program which globally controls all individuals with full knowledge of the entire system. On the other hand, *decentralised* or *distributed* systems provide each individual with enough autonomy to communicate with its neighbours and move locally. There have been an impressive number of recent developments in collective robotic systems that have demonstrated such systems' potential and feasibility, from the milli or micro [28, 78, 85, 102] down to the nano scale [62, 100].

This has opened the door for the implementation of many innovative real-life applications across a wide range of domains. A popular recent example is the newest automated warehouse of the UK online grocery company *Ocado* [2], where a collection of robots were deployed on a two-dimensional grid system, called the *hive* and demonstrated in Figure 1.1, and worked together to collect orders and pack groceries (for a demonstration of the abstract representation of the square grid system, see Figure 1.2). The robots communicate via 4G technology and use their sensors to coordinate their traffic. Each robot holds its own battery, which can be recharged when it runs out in one of the charging bays placed across the grid. Moreover, the robot exploits a self-control system with which it can follow a movement profile to decide where and when to move and stop on the grid. Once a robot's sensors encounter any issues, the robot switches to standby mode and moves to a safe spot for further investigation. *Amazon* [40] and *Alibaba* [127] – the leading American and Chinese e-commerce companies, respectively – have also adopted similar successful applications of robotic systems in their smart warehouses. Design of efficient and correct algorithmic solutions for a collection of robots is important for building safe and reliable industrial processes and production lines. Moreover, recent accident in July 2021 at Ocado's warehouse of collided robots highlighted the importance of building algorithms with a mathematical guarantee to avoid major disruption, see [5].

Figure 1.1: Inside the hive, one of Ocado's warehouses [1].

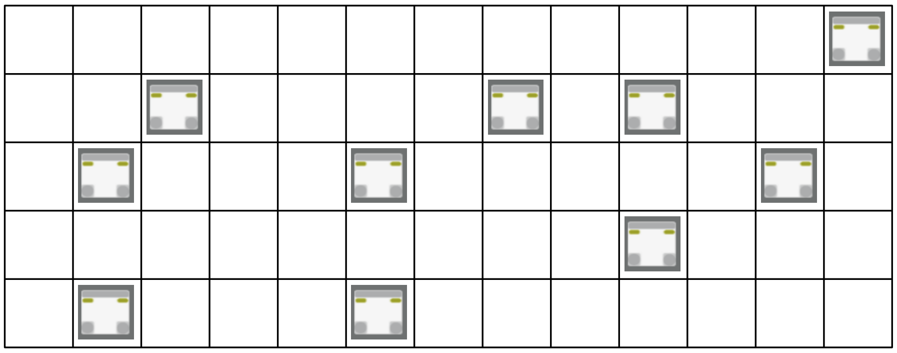


Figure 1.2: A number of robots travel atop a square grid system inside Ocado's warehouses.

Researchers from a variety of fields, including computer science, materials science, physics and biology, have recently made significant progress in developing innovative, scalable and adaptive tiny robotic-units systems. This grand vision is currently shaping the research area of *programmable matter*, which was proposed by Toffoli and Margolus [113] in 1991. The term 'programmable matter' refers to any kind of materials that can *algorithmically* change its physical properties, such as shape, colour, density and conductivity through transformations executed by an underlying program. 'Algorithmically', in this context, means that the change (or *transformation*) is the result of executing an *underlying program*. This new area has attracted growing interest from both theoretical and practical viewpoints.

If this type of matter is realised, a plethora of practical challenges can be addressed in a variety of domains. For example, in healthcare, a system of very tiny particles injected into a human body could transform into several shapes to efficiently traverse veins and capillaries and treat infected cells. In another domain, air and water contaminants could be identified and metabolised by a system of micro-particles. Another future application might be materials with the ability to verify structural integrity and repair any minor flaws in construction. Ball [22] broadly and intuitively explored this emerging concept and depicted future visions of this technology. Figure 1.3 shows one of these imaginings.

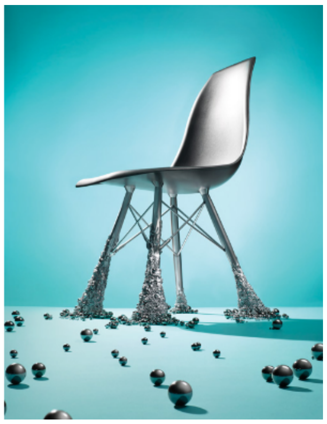


Figure 1.3: An artistic view of a future application for programmable matter [22].

Several ongoing projects are conducting research on the design of the hardware components that make up programmable matter. For example, a research group from the University of Bourgogne Franche-Comté in France [3] has been involved in the development and implementation of distributed systems consisting of a collection of smart micro–electro-mechanical units. This project is a prominent collaborative initiative in this subfield and spans a diversity of universities and institutions, including the University of Tokyo, the University of Michigan, Carnegie Mellon University and our team at the University of

Liverpool [4].

Piranda and Bourgeois [97] conducted one implementation of this project, studying the design of a milli-scale self-reconfigurable system consisting of components called *Claytronics atoms*, or '*catoms*', that are able to attach and detach from each other as well as rotate around one another. Figure 1.4 depicts the rotation mechanism. The number of challenges related to communication, motion and occupation grows at a steady pace with the number of catoms in the system. Hence, in [97], the authors offer a range of geometrical practical solutions to these issues.

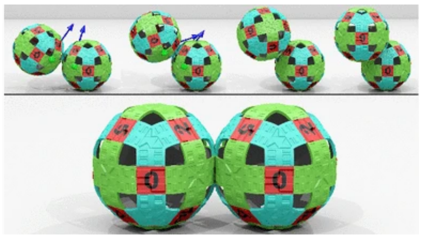


Figure 1.4: Two catoms are shown next to each other at the bottom, while the rotation movement is shown at the top [97].

This progress has motivated the parallel development of a theory of small scale robotic systems. The apparent absence of formal theoretical foundations – including modelling, possibilities and limitations, algorithms, computability and complexity – has been highlighted in, for instance, [89] and [91]. The development of a formal theory is a fundamental step for further progress in those systems, as theory can accurately predict the most promising designs and suggest new ways to optimise them by identifying crucial parameters and the interplay between them. It also provides (centralised or distributed) algorithmic solutions that are best suited to each given design and task, coupled with provable guarantees regarding their performance.

Several theoretical computer science subfields have appeared for different areas, including reconfigurable robotics [17, 30, 46, 53, 126], metamorphic systems [66, 93, 117], mobile robotics [37, 43, 45, 56, 124], passively-mobile systems [19, 20, 90, 91], DNA self-assembly [60, 101, 118, 121], and even the theory of puzzles [25, 48, 81]. A latest ongoing effort is the joining of these theoretical forces and developments within the emerging subfield of 'Algorithmic Foundations of Programmable Matter' [68]. More details and specific reviews of the relevant literature are provided later in this chapter.

The *shape formation* problem (also known as *pattern formation* or *shape transforma-*

tion) is ubiquitous in the vast variety of robotic systems. Indeed, it might be considered the most essential and natural goal in this area. Abstractly, take a step back from any specific instantiation of robotic systems and consider a system deployed on a two-dimensional square grid in which a collection of spherical entities of limited computational capabilities are typically *connected* to each other, forming an initial shape (or configuration) S_I. The shape formation problem is formulated as follows:

"*The entities move on the grid to transform S_I into a desired target shape S_F within a finite number of valid individual moves.*"

A number of models for such systems have been established and introduced in the literature. For example, Dumitrescu and Pach [64], Dumitrescu *et al.* [65, 66] and Michail *et al.* [89] considered mechanisms whereby an individual device can move over and turn around its neighbours through empty space. Transformations based on similar moves being assisted by small seeds were also considered in [8]. Each approach – or, more precisely, each model – has its own distinct beauty and has drawn attention to many interesting technical problems and open research questions, in addition to contributing several valuable insights for the development for future applications and prospective systems.

A linear-strength model: This thesis primarily focuses on boosting the theoretical underpinning of robotic systems. In particular, it investigates how the capabilities of certain mechanisms can induce more efficient shape transformations. To this end, it introduces a new *linear-strength* model for a robotic system comprised of devices residing on a two-dimensional square grid in which an entire line of consecutive devices can, in parallel in a single time-step, move by one position in a given direction (referred to as a *line move*). This model, called the *line-pushing* model, is a natural generalisation of other existing models, with a particular emphasis on exploiting the power of parallelism for fast global reconfiguration. Apart from the pure theoretical interest of exploring fast transformations on a grid, this model also provides a practical framework for efficient reconfigurations of real systems. This approach, for example, might be applied to reconfigurable robotic systems in which individual devices are equipped with linear-strength locomotion mechanisms. This might be another system of external linear-strength forces that occur naturally (e.g. gravity, water, winds) or artificially (e.g. magnetic surface, mechanical energy, a person's hands) with which the devices adhere appropriately to form a desired shape.

Section 1.2 is devoted to a thorough discussion of the relevant literature with a particular focus on related studies on the main problem of this thesis, which is the shape formation problem in Section 1.2. Next, Section 1.3 provides an overview of the overall structure of this thesis. Following that, in Section 1.4, we go over the specific problems that the thesis addresses and summarise how this study contributes to the area of robotic systems along with the publications in Section 1.5.

1.2 State of the art

We begin with a review of existing models and shape formation problems in the relevant literature. As this research study is in line with newly established subfields of robotic systems, we present a general, top-level overview of the most prominent related centralised and distributed studies. A particular focus is given to the literature most closely associated with the subject of shape formation, which is the main problem of this thesis. The overview of relevant systems and models offered in this section is based mainly on the published papers underpinning this thesis [10, 12, 13, 14, 15] and partially on the literature reviews in other recent works [46, 47, 56, 63, 89, 110]. All omitted formal details, such as definitions, conventions and proofs, can be found in the referenced literature.

A vast amount of literature on robotic systems has existed for a long time, and there are certain commonalities among these studies. Almost all share some *underlying principle* that permits the research community to establish a wide range of theoretical and practical studies. Although there is an apparent paucity of theoretical literature in the context of robotic systems (as noted in [89], among others), they contribute to some extent to the ultimate goal of the emergence of a fine-grained collective system, such as programmable matter. Here, we do not intend to provide a detailed description of each model or thoroughly explore all proposed results in depth, as doing so would be beyond the scope of this thesis. We thus intend to provide a brief survey of the most relevant systems, particularly from a theoretical point of view, such as:

- Swarm and mobile robotic systems.
- Modular self-reconfigurable robotics and metamorphic systems.
- Models for programmable matter.

The methodology that can be used to categorise robotic systems has become somewhat complicated due to the recent growth in the body of research as well as the overlapping features among these systems. In general, one can categorise robotic systems into *active* and *passive*. Entities in the passive systems have no control over their movements. Instead, they move via interactions with the environment in which they live based on their own structural characteristics. Prominent examples of research on passive systems appear in the areas of population protocols [19, 90, 91], DNA computing [6, 26], chemical reaction networks (CRNs) [61, 108], and tile self-assembly [31, 60, 96, 101, 118]. Active systems, on the other hand, 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 [66, 93, 117], swarm and mobile robotics [43, 72, 98, 102, 124], modular self-reconfigurable robotics [17, 76, 126] and recent research on programmable matter [46, 53].

While the passive models are less relevant to our approach, a number of them investigate problems similar to those of interest in this thesis, particularly tile self-assembly models. Furthermore, some models may be considered to be on the border between passive and active systems [47]. For example, population protocols can be viewed as partially passive with regard to their movements while at the same time conducting active interactions (communication and computation). Similarly, slime moulds (e.g. [27, 99, 112]) can be viewed as a hybrid system, in which individuals actively control their motion but behave passively in cooperation with their environment. Likewise, some swarm robots that communicate physically in a passive medium lie somewhere between passive and active systems (for some recent work in this area, see e.g. [36, 86, 87, 105, 122]).

On the other hand, active systems, particularly those modelling graph-based space and discrete systems, are more relevant to our algorithmic work, in which individuals interact and move actively with self-control. Most swarm robotic systems are fully active, including those of microcontrollers (see the excellent recent reviews in [23, 29, 59, 80] as well as those on the sub-area of reconfigurable modular robotic systems [17, 18, 97, 126]). There is substantial literature in this area, especially from a practical perspective, including within the closely related subfield of *metamorphic systems* [65, 66, 117].

Research on shape formation problem

We are primarily interested in relevant models that are being considered or have been used to investigate the classical shape formation problem. Starting with passive systems, most prominent approaches on shape formation appear mostly within the domain of DNA computing (e.g. [32, 55, 94]), in addition to tile-based self-assembly systems (e.g. [49, 100, 106, 109]) and population protocols (e.g. [88, 90]).

Indeed, shape formation is a well-known challenge in active robotic systems that has been studied in nearly every model. Pattern formation is another name for this problem, which appears frequently in swarm robotics. It has attracted a large amount of research attention, especially regarding its practical and physical aspects. However, there is a dearth of comprehensive surveys in the field summarising all formal and theoretical works on pattern and shape formation. We thus strive to provide the most relevant findings, which reflect typical examples of the acquired results.

1.2.1 Relevant models and results

Swarm robotic systems. Swarm robots are often composed of a group of autonomous robots, each of which is outfitted with sensors that allow it to view, communicate and manoeuvre in a given area. A wide range of problems has been considered within the context of swarm robots, including shape formations [102], graph exploration [71] and gathering problems [7]. It is an enormous field of research, with conferences dedicated specifically to swarm robotics and subjects related to the discipline, such as '*The International Conference on Swarm Robotics and Swarm Intelligence*'. Flocchini *et al.* [72] offered an excellent report previewing several theoretical studies spanning a range of approaches in the area of mobile (swarm) robots. One subfield – modular self-reconfigurable robotics – forms its own rich area of research, as discussed next.

Modular self-reconfigurable robotics. The focus of this system is largely on the inner aspects and motion features of robots, such as design, assembly, motion planning and kinematic machine control, coupled with some theoretical treatments [75]. Formal theory (e.g. [30, 82, 117]) promotes the hardware and practical achievements in a multiplicity of modular robotic systems, mainly those with more capabilities than are accessible at small scales (e.g. [67, 77, 102]). The recent Claytronics project [97], mentioned earlier, is a remarkable exception, as it carefully considers the micro-scale limitations of robots. The subclass of *metamorphic systems* (e.g. [35, 64, 65, 66]) is of particular interest, as it shares several characteristics with our approach (Chapters 2, 3 and 4). A metamorphic system may be thought of as 'a robot of robots' – that is, a collection of modular robots that are connected physically and operate collectively as a single robot on the plane. Several studies have exploited this model to investigate the shape formation [66, 89], locomotion

[33, 65] and exploration [58] problems. Hurtado *et al.* [82] presented a rigorous theoretical framework for distributed systems in modular robotics. This is one of the studies most closely connected to our distributed model (Chapter 5), as both share some characteristics such as an underlying square grid system and a focus on the movement mechanisms of robots.

Now, we devote a separate discussion to metamorphic robotic systems. The models studied in those systems [64, 65, 66, 89] consider a number of spherical devices given in the form of a (typically connected) shape A and lying on a two-dimensional square grid. The goal is to transform A into a desired target shape B via a sequence of valid movements of individual devices. The mechanisms examined in those papers were the ability to rotate and slide a device over neighbouring devices (always through empty space). Transformations based on similar moves and assisted by small seeds were also considered in [8].

Dumitrescu and Pach [64] demonstrated that, by combining rotation and sliding, they could guarantee the *feasibility* of individual movements and show that any pair of connected shapes (of the same number of robots) could *universally* transform to each other. They also proved *connectivity* – that is, all intermediate shapes were connected during transformations. Michail *et al.* [89] then proved that, if the devices are equipped only with a rotation mechanism, the decision problem of transforming A into B is in $\mathbf{P}$. They gave a constructive characterisation of the (rich) class of pairs of shapes that are transformable to each other.

Further, the authors in [66] and [89] developed distributed transformations for the shape formation problem by exploiting individual movements and presented some decidability questions, including their main universality question, which was later proved in [64]. The distributed model in [65, 66] enables a device to have a sense of the current state of the global configuration, while the model in [89] allows only for a local view. Both [66] and [89] exploited *the pipelining technique* to design distributed and centralised universal transformations, respectively, that work in linear parallel time. However, other researchers have explored the problem on alternative geometries, such as the hexagonal girds exploited in [93] and [117].

Recently, our *line-pushing model* was introduced in [10] as a new model for various robotic systems, especially those of nano-scale devices, such as programmable matter. It investigates an alternative *linear-strength* mechanism, by which a line of one or more devices can translate by one position on a two-dimensional grid in parallel in a single time-step. The main goal is to determine whether this new mechanism can *in principle* be exploited for the development of more efficient centralised and distributed algorithms, such as sub-quadratic worst-case transformations. As a first step, we naturally restricted our attention to centralised transformations. Our main result was a universal transformation of $O(n \log n)$ worst-case running time (on the number of moves) that is permitted to break *connectivity* [14] (where n denotes the number of devices of the shape and connectivity means that the *associated graph* induced by the nodes occupied by the devices is connected).

By restricting our focus to connectivity-preserving transformations, we were able to propose very fast transformations for a large class of connected shapes that takes the same asymptotic $O(n \log n)$ moves. Our most general result is then a connectivity-preserving universal transformation that can transform any pair of connected shapes through a sequence of sub-quadratic moves [12]. After examining basic concepts, we established an algorithmic decentralised framework to enable line moves and implemented the first distributed transformation that requires time asymptotically equivalent to that of the best-known centralised transformations while preserving connectivity during its course [15]. The algorithm solves the *line formation problem*, allowing agents to form a final straight line, starting from any initial connected shape, whose associated graph contains a Hamiltonian path. More details of our findings are covered in Section 1.4.

1.2.2 More peripheral related topics

On the theoretical side, a number of formal approaches on pattern formation have been established to enhance the practical studies. For example, Flocchini *et al.* [73] studied the fundamental algorithmic aspects for a set of *anonymous* robots (i.e. without *IDs*) coordinating distributively and asynchronously on a plane of two axes and able to transform into any predefined shape. Within this weak framework, they showed that such a formation could not be achieved if the robots had no prior agreement about their environment (i.e. common orientation). If, for example, the north direction is known to all robots, then any odd-sized robot can accomplish the task, while even-sized robots cannot. More generally, any set of robots can complete the transformation when two orthogonal directions are provided (i.e. common chirality).

A similar formal study [44] identified the minimal computational restrictions that *oblivious* robots (i.e. those with no memory) could impose in the setting of anonymous robotics established by [73]. It also investigated under what conditions those oblivious robots could form a set of geometric shapes starting from any arbitrary initial shape. Specifically, they studied the set of geometric shapes that anonymous and oblivious robots could form asynchronously on a system suffering a lack of global orientation. Our distributed model has a comparable setting in which the individuals are anonymous and identical. They have only visual sensory information about their local surroundings with no means of explicit communication with each other but differ in that they have constant memory.

From an engineering point of view, a recently revealed system [102] demonstrated the ability to manipulate thousands of micro-robots called *kilobots* to form complicated two-dimensional shapes. The vision behind this achievement involves creating tiny, simple kilobots of limited power that can be cheaply deployed on a two-dimensional plane. These kilobots can collectively transform into multiple shapes through local interactions in a distributed manner and are very resistant to associated global dynamics concerns, such as errors and robot failures. Although the algorithm functions locally, the number of kilobots in the system is predetermined by a global control. This reliance on the size of the system can be evaded by an application of [103]. Another model [21] considers a swarm of many simple square-shaped robots aligned on a two-dimensional grid, all of which have a generic procedure that allows them to approximate reconfiguration into any arbitrary polygonal shape, thus improving the system's ability to repair and self-heal during formation.

Further, the combinatorial game theory of one-player games (e.g. puzzles [25, 48]) may offer some insights into algorithms and complexity in robotic systems, such as decidability of several motion-planning problems [81]. In addition, the area of *cellular automata* is a well-known theoretical framework for studying a wide range of problems, particularly in bio-inspired systems [39]. Many books have discussed its theory, power and applications (e.g. [83, 107, 120, 119]). Moreover, some models have recently been built with the goal of studying the complexity and revealing the boundaries of active nano-scale entities, such as the *nubot model* [121] and the *amoebot model* [51]. Next, we discuss those models in more detail.

Models for programmable matter. We present some prominent and relevant models that have been recently introduced within the domain of programmable matter.

Woods *et al.* proposed the *nubot* model to establish a solid foundation for studying and analysing the complexities of growth and dynamic bio-inspired systems composed of active molecular entities [121]. This model is inspired by biology and based on tile-based self-assembly, molecular motors and circuits. It aims to reveal the connection between complicated molecular structures and dynamics. It is depicted as a triangular grid with entities called *monomers* that move relative to one another and can emerge and vanish from the grid. Changes in the states of individuals and their relative motions enable these monomers to expand and contract, resulting in the formation of different assemblages in the system. The nubot model differs from others in that there is always a large supply of extra monomers available to be fed into the system on demand. Further, it exploits a non-local propagation of movements across the system.

In the nubot model, all of its systematic main features permit a line formation in a logarithmic time and number of states. Their primary result is the development of a universal transformation that allows monomers to grow any target two-dimensional connected shapes within polylogarithmic time on the size of the shape. This is combined with the time it takes to simulate a Turing machine (TM) that decides whether a given pixel is in the target shape. The nubot framework is distinguished by its reliance on a key characteristic; namely, a single monomer is able to deterministically push or pull around large structures (i.e. other monomers). The question arose in [34] whether a more restrictive system could obtain similar performance for these inherently uncontrolled and unpredictable molecular movements. That study demonstrated the feasibility of forming squares in polylogarithmic time and a line in sublinear expected time under engineered molecular models. Correspondingly, this movement is analogous in spirit to the *length-strength pushing mechanism* used throughout this thesis.

Recently, Derakhshandeh *et al.* introduced the *amoebot* model, which "offers a versatile framework to model organising particles and facilitates rigorous algorithmic research in the area of programmable matter" [51]. It is an amoeba-inspired model consisting of a collection of entities (called *particles*) residing on a two-dimensional hexagonal grid that have limited computational, communication and locomotion capabilities. Since its appearance in 2014, it has been the subject of a growing body of research and has been used to study several problems, including shape formation [52, 53, 56], leader election [24, 56, 63] and coating [46]. During this time, the amoebot model has occasionally been revised and updated to address specific problems. Generally, it shares some similarities with the nubot model, such as the underlying grid and the extension and contraction movement of particles relative to each other. In his PhD thesis, Daymude [47] comprehensively discussed all updates, recent activities and developments within the framework of the amoebot model.

In the amoebot model, the first algorithm was introduced is to solve the line formation problem in [54]. Next, the transformation was extended in [52] to cover other simple connected shapes, such as a snake-like shape. Following that, a more general transformation was developed in [53]: given an initial *well-structured* configuration of particles forming a triangle in which each particle encodes its final position, then the transformation allows them to form a final target shape belonging to a set of connected shapes (regular hexagons) within a sublinear number of rounds and quadratic expected time on the number of moves. This transformation works under some assumptions, such as common agreement of orientation (i.e. chirality), randomisation for symmetry breaking (i.e. to elect a leader) and a sequential order of scheduler activation.

Most recently, Di Luna *et al.* [56] effectively abandoned those restrictions and devised a wider general transformation that works for any *feasible* pair of connected shapes, where 'feasible' refers to which pair of connected shapes can be *deterministically* transformed to each other. The final configuration is a scaled-up copy of the initial shape (i.e. is not necessarily identical). That study brought the Look-Compute-Move (LCM) paradigm into the amoebot framework to handle concurrency and avoid sequential activations. The transformation completes its course in a quadratic number of rounds and movements, though it only works for connected shapes that do not contain 'holes'. In our distributed approach, we use the LCM concept of mobile autonomous robots in the domain of line pushing.

Another relevant line of research in [43, 70, 79] studied a set of models considering a single robot that moves over a static shape consisting of tiles. The goal is for the robot to transform the shape by carrying one tile at a time. In those systems, the single robot that controls and carries out the transformation is typically modelled as a finite automaton. Those models can be viewed as partially centralised: on the one hand, they have a unique controller, but on the other, that controller operates locally and suffers from a lack of global information.

Furthermore, the pattern formation problem has also been widely researched in the context of cellular automata, a classic and well-established model of computation developed in the 1940s by Stanislaw Ulam [114] and John von Neumann [116]. Packard and Wolfram [95] presented an overview of preliminary findings. Many studies have exploited cellular automata to form and investigate various patterns in biology (see a recent book in [55]). One application can be seen in [94], where Nickson and Potapov exploited the model of *broadcasting automata* to investigate the formation of discrete shapes induced by informational waves on a square grid lattice within von Neumann and Moore neighbourhoods. This paradigm, which has linkages to many communication concepts in distributed computing, can be simulated using cellular automata. Another application is in [32], where a genetic algorithm was used to generate two- and three-dimensional target patterns through evolving cellular automata.

Therefore, our model is inheriting a number of characteristics from the above models. For example, the idea of movements corresponds to similar models of active systems in which an individual can decide when and where to move, as previously mentioned in swarm robotics and some programmable matter models. The minimum number of movements in our model can be related to block-pushing and sliding puzzles in respect to the evaluation on lower bounds. Moreover, the agent's local vision in our distributed model is inherited from Moore neighbourhoods in cellular automata. Abstractly, our distributed transformation (in Chapter 5), as well as the nubot model, can be modelled as an application of a cellular automaton to a certain degree. Our model differs in terms of how the cells in cellular automata can appear and disappear on the system during an execution. Further, modular self-reconfigurable robotics influences our objective of the shape formation problem.

1.3 Thesis outline

Nowadays, there are massive efforts underway to enhance energy efficiency and raise awareness of the need to reduce energy demand in many various domains. In particular, from the perspective of researchers and theoreticians in robotics, this highlights the importance of designing and deploying more *energy-efficient* robotic systems. The majority of the existing formal research has examined transformation efficiency in terms of the traditional complexity metrics of time and, occasionally, space, while completely neglecting the cost of energy usage. Given that actuation is a major source of energy consumption in real programmable matter and robotic systems, move minimisation is expected to contribute to the deployment and implementation of energy-efficient systems. In this thesis, we consider this critical issue in the context of our new model of line moves [14]. Our objective was to trade time for number of line moves and investigate whether the parallelism inherent in this new type of movement could be exploited for more efficient centralised and distributed transformations. In conjunction with this, our other main goal is to reveal the *underlying principle* of the proposed model.

Chapter 2 establishes a generic algorithmic framework of line moves for our discrete shape formations in centralised setting. This includes all conventions, definitions and basic facts to formalise the technical properties and algorithmic tools used throughout the thesis. Next, formal definitions are given for all shape formation problems considered in this work. The final section of this chapter discusses the first lower bounds established for this model under the constraints of two sets of transformations.

As a first step towards understanding the power of the proposed model, we naturally restrict our attention to centralised transformations in Chapters 3 and 4. This is because distributed are model-dependent (in terms of, for example, knowledge and communication), while centralised show what is possible *in principle*. Moreover, some of the ideas in centralised transformations might prove useful for their distributed counterparts. Chapter 3 explores two different shape formation problems. Throughout that chapter, we focus on a worst-case instance and then present an efficient universal algorithm that correctly solves the problem in which the transformations are allowed to drop the *connectivity-preservation* condition during their course. We next develop and analyse the first connectivity-preserving transformations for a worst-case pair of shapes, as well as a very fast connectivity-preserving transformation for a large family of shapes (Chapter 4). The main contribution of that chapter is a connectivity-preserving universal transformation.

The final technical chapter of this thesis establishes a decentralised linear-strength framework for line moves (Chapter 5). It also introduces the first distributed transformation for robotic systems implementing the new mechanism. Furthermore, it formalises all communication, interaction and vision in the tradition of *finite state automata*, allowing for more precise handling of time, synchronisation and transformations. Our distributed implementation solves the basic line formation problem and preserves 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. In Chapter 6, we conclude with a discussion of the thesis, highlighting a number of open problems and suggesting promising new future research directions.

In this thesis, the proposed transformations are categorised into unrestricted (Chapter 3), connectivity-preserving (Chapter 4) and distributed (Chapter 5), in an attempt to provide a logical step-by-step story towards the main results. This arrangement is expected to facilitate the presentation and motivate the reader to comprehend the sequence of development gradually, starting with elementary reasoning and basic operations. Figure 1.5 depicts a dependency digram that shows how transformations build on each other, progressing chronologically from a hard-special case to more general restricted and distributed transformations with increasingly better bounds. In Section 3.1.1 of Chapter 3, the hard-case partitioning approach has been used to develop better bound transformations in Section 3.1.2 and universal transformations in Section 3.2.1, which were then exploited altogether to achieve universality more efficiently in Section 3.2.2. Thus, Section 3.1.3

answers whether a uniform recursion of 3.1.2 could achieve better bound. In Chapter 4, Section 4.1.1, which preserves connectivity for the hard-case, is influenced by Section 3.1.1 and then combined with Section 3.2.1 to produce more restricted universal algorithms in Section 4.3. This also aided in the development of an alternative hard-instance strategy (Section 4.1.2), whose algorithmic tools are integrated with those of Section 3.1.2 to generate connectivity-preserving Hamiltonian transformations of the best-known bound (Section 4.2), which are then distributed in Chapter 5.

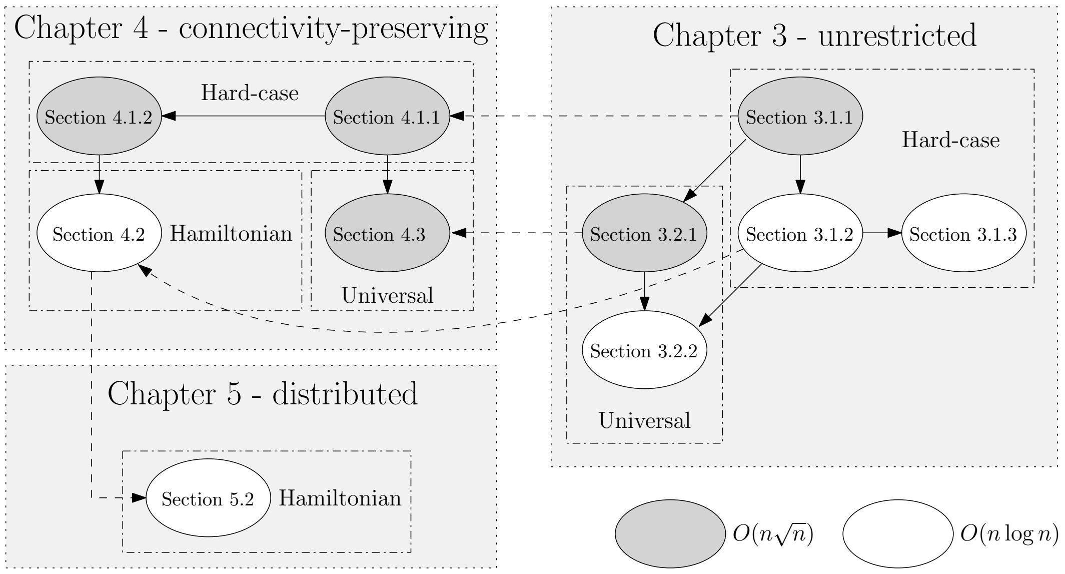


Figure 1.5: A dependency digram of transformations.

As evidenced by a number of publications (listed in Section 1.5), this thesis covers a rang of linear-strength transformations modelling, design, and implementation. First, in Chapter 2, the line-pushing model and some of its technical features were introduced in [10]. Furthermore, in Chapter 3, the $O(n\sqrt{n})$-time and $O(n \log n)$-time hard-case partitioning approaches in Sections 3.1.1 and 3.1.2, as well as an $O(n\sqrt{n})$-time and $O(n \log n)$-time universal transformations in Sections 3.2.1 and 3.2.2, were initially published in [10]. Then, they were extended and combined with the first $O(n\sqrt{n})$-time connectivity-preserving transformations (Section 4.1.1) in [14]. Following that, the $O(n \log n)$-time Hamiltonian (Section 4.2) and $O(n\sqrt{n})$-time universal transformations (Section 4.3) in Chapter 4 were presented in [12] to solve the connectivity-preservation condition. Finally, [15] makes the first $O(n \log n)$-time distributed connectivity-preserving transformation in Chapter 5.

1.4 Technical contributions

Let us consider a system deployed on a two-dimensional square grid in which a collection of spherical devices (referred to as nodes or agents from now on) are typically connected to each other, forming a shape S_I. By a finite number of valid individual moves, S_I can be transformed into a desired target shape S_F. In [64, 65, 66, 89] and related models, where in any time-step at most one node can move a single position in its local neighbourhood, it can be proved (see, for instance, [89]) that there will be pair of shapes that require $\Omega(n^2)$ moves to be transformed into each other. This follows directly from the inherent 'distance' between the two shapes and the fact that this distance can be reduced by only a constant in every time-step. An immediate question has arisen then:

"How can we come up with more efficient transformations?"

Two main alternatives have been explored in the literature in an attempt to answer this question. One is to consider parallel time, meaning that the transformation algorithm can move more than one node (up to a linear number of nodes if possible) in a single time-step. This is particularly natural and fair for distributed transformations, as it allows all nodes to have their chances to take a move in every given time-step. For example, such as transformations based on pipelining [66, 89], where essentially the shape transforms by moving nodes in parallel around its perimeter, can be shown to require $O(n)$ parallel time in the worst case. This technique has also been applied in systems (e.g. [102]).

The other approach is to consider more powerful actuation mechanisms, that have the potential to reduce the inherent distance faster than a constant per sequential time-step. These are typically mechanisms where the local actuation has strength higher than a constant. This is different from the above parallel-time transformations, in which local actuation can only move a single node one position in its local neighbourhood and the combined effect of many such moves at the same time is exploited. In contrast, in higher-strength mechanisms, it is a single actuation that has enough strength to move many nodes at the same time. Prominent examples in the literature are the linear-strength models of Aloupis *et al.* [17, 18], in which nodes are equipped with extend and contract arms, each having the strength to extend and contract the whole shape as a result of applying such an operation to one of its neighbours. Further, the prospered model of Woods *et al.* [121], in which a chain of nodes is being dragged parallel to one of the axes directions by a single rotating node (acting as a linear-strength rotating arm).

Our proposed model in Chapter 2 follows similar approach, by introducing and investigating a linear-strength model in which a node can push a line of consecutive nodes one position (towards an empty cell) in a single time-step. One of the most fundamental feature of the proposed model is *transformability*. Section 2.4 shows that it can easily simulate the combined rotation and sliding mechanisms of [64, 89] by restricting moves to lines of length 1 (i.e. individual nodes). It follows that this model is also capable of universal transformations, with a time complexity at most twice the worst-case of those models, i.e. again $O(n^2)$. Naturally, our focus is set on exploring ways to exploit the parallelism inherent in moving lines of larger length in order to speed-up transformations and, if possible, to come up with a more efficient in the worst case universal transformation.

Further, as *reversibility* of moves is still valid in our model (Section 2.4), we adopt the approach of transforming any given shape S_I into a spanning line S_L (vertical or horizontal). This is convenient, because if one shows that any shape S_I can transform fast into a line S_L, then any pair of an initial S_I and final S_F shapes (of the same size) can then be transformed fast to each other by first transforming fast S_I into S_L and then S_L into S_F by reversing the fast transformation of S_F into S_L. Given this, our focus in the thesis is to investigate whether the presented linear-strength mechanism can achieve faster transformations that transform any pair of connected shapes to each other. Next, we establish $\Omega(n \log n)$ lower bounds for two restricted sets of transformations in Section 2.5. These are the first lower bounds, under restrictions, for this model and are matching the best known $O(n \log n)$ upper bounds.

In Chapter 3, we start our investigation on a set of unrestricted transformations in which the nodes are not necessarily preserving the system connectivity (i.e. the associated graph induced by the occupied nodes is connected) during their course. Thus, we begin by identifying the diagonal shape S_D (which is considered connected in our model and is very similar to the staircase worst-case shape of [89]) as a potential worst-case initial shape to be transformed into a line S_L. This intuition is supported by the $O(n^2)$ individual node distance between the two shapes and by the initial unavailability of long line movements: the transformation may move long lines whenever available, but has to pay first a number of movements of small lines in order to construct longer lines. In this benchmark (special) case, the trivial lower and upper bounds $\Omega(n)$ and $O(n^2)$, respectively, hold. More details of this worst-case are discussed in Section 2.1.

For example, take the diagonal worst-case shape S_D (e.g. Figure 3.1 (a)) and try to convert it into a straight line S_L. Observe that any transformation requires $\Theta(n^2)$ sequential individual movements to transform S_D into S_L, see for example Figure 2.4. By exploiting linear-strength mechanism, let us now perform the following simple strategy; (1) divide S_D into several diagonal segments of length $\sqrt{n}$ each (see Figure 3.1 (a)), then (2) turn each segment into a straight line segment via individual moves, which takes linear time for each segment (Figure 3.1 (b) and (c)). Next, (3) transfer every line segment all the way down to the bottom of S_D (Figure 3.1 (d)) in which each line segment pushes a maximum of n distance. Finally, (4) change the orientation of all line segments in linear time to form the target straight line S_L (Figure 3.1 (e)). This transformation takes a total of $O(n\sqrt{n})$ time-steps. Thus, in contrast to the aforementioned models, the new mechanism allows for sub-quadratic strategies, like the one just sketched. The complete technical description of this strategy will be presnted later in Section 3.1.1.

As the $O(\sqrt{n})$ length of uniform partitioning into segments is optimal for the above type of transformation, we turn our attention into different approaches, aiming at further reducing the running time of transformations. Allowing once more to break connectivity, we develop an alternative transformation in Section 3.1.2 based on *successive doubling*. The partitioning is again uniform for individual 'phases', but different phases have different partitioning length. The transformation starts from a minimal partitioning into $n/2$ lines of length 2, then matches them to the closest neighbours via shortest paths to obtain a partitioning into $n/4$ lines of length 4, and, continuing in the same way for $\log n$ phases. It thus maintains the invariant of having $n/2^i$ individual lines in each phase i, for $1 \leq i \leq \log n$. Proving that the cost of pairwise merging through shortest paths in each phase is linear in n, it is sufficient to conclude that this approach transforms the diagonal into a line in time $O(n \log n)$, thus yielding a substantial improvement.

Observe that the problem of transforming the diagonal line shape into a straight line seems to involve solving the same problem into smaller diagonal segments (in order to transform those into corresponding line segments). Then, one may naturally wonder whether a recursive approach could be applied in order to further reduce the running time. Section 3.1.3 provides a negative answer to this, for the special case of uniform recursion and at the same time obtain an alternative $O(n \log n)$ transformation for the diagonal-to-line problem.

Then, the focus turns on attempting to generalise the ideas developed for the above benchmark case in order to come up with *equally efficient universal transformations*. In Section 3.2, we successfully managed to generalise both the $O(n\sqrt{n})$ and the $O(n \log n)$ approaches, obtaining universal transformations of worst-case running times $O(n\sqrt{n})$ and $O(n \log n)$, respectively. We achieve this by enclosing the initial shape into a square bounding box and then subdividing the box into square sub-boxes of appropriate dimension. For the $O(n\sqrt{n})$ bound, a single such partitioning into sub-boxes of dimension $\sqrt{n}$ turns out to be sufficient. For the $O(n \log n)$ bound we again employ a successive doubling approach through phases of an increasing dimension of the sub-boxes, that is, through a new partitioning in each phase. Therefore, our ultimate theorem (followed by a constructive proof, providing the claimed transformation) states that:

"In this model, when connectivity need not necessarily be preserved during the transformation, any pair of connected shapes S_I and S_F can be transformed to each other in sequential time $O(n \log n)$."

The *connectivity-preservation* is an important assumption for many robotic systems, see for example [42]. It is a desirable property for the design of any robust transformation to enhance reliable communication among different types of networks. This condition is substantial for acquiring a computationally distributed nature, as well as for practical applications that often require energy for data transmission and the implementation of various locomotion methods. Further, maintaining connectivity could be crucial to prevent the configuration from falling apart and to keep all agents powered. Thus, any disconnection that occurs during transformations must be performed very carefully to prevent the creation of any potentially separated sections that cannot be reconnected again.

Hence, Chapter 4 restricts the attention on designing centralised transformations with the aim of minimising the total number of moves subject to the constraint of preserving connectivity of the shape throughout the course of the transformation. That is, in each intermediate configuration we always want to guarantee that the associated graphs induced by the occupied nodes are connected. First, we take the algorithmic idea of partitioning one step further, by developing two transformations building upon it, that can achieve the same time-bound while preserving connectivity throughout their course: one is based on folding segments in Section 4.1.1 and the other on extending them in Section 4.1.2.

Next, we present an $O(n \log n)$-time transformation in Section 4.2, called *Walk-Through-Path*, that works for all pair of shapes (S_I, S_F) that have the same order and belong to the family of *Hamiltonian shapes.* A *Hamiltonian shape* is any connected shape S *associated* to a graph $G(S)$ that contains a Hamiltonian path (see also Itai *et al.* [84] for the Hamiltonian paths). At the heart of our transformation is a recursive successive doubling technique, which starts from one endpoint of the Hamiltonian path and proceeds in $\log n$ phases. In every phase i, it moves a terminal line L_i of length 2^i a distance 2^i higher on the

Hamiltonian path through a *LineWalk* operation. This leaves a new terminal sub-path S_i of the Hamiltonian path, of length 2^i. Then the general procedure is recursively called on S_i to transform it into a straight line L'_i of length 2^i. Finally, the two straight lines L_i and L'_i which are perpendicular to each other are combined into a new straight line L_{i+1} of length 2^{i+1} and the next phase begins.

A core technical challenge in making the above transformation work is that Hamiltonian shapes do not necessarily provide free space for the *LineWalk* operation. Thus, moving a line has to take place through the remaining configuration of nodes while at the same time ensuring that it does not break their and its own connectivity, including keeping itself connected to the rest of the shape. We manage to overcome this by revealing a nice property of line moves, according to which a line L can *transparently* walk through *any* configuration S (independently of the latter's density) in a way that: (i) preserves connectivity of both L and S and (ii) as soon as L has gone through it, S has been restored to its original state, that is, all of its nodes are lying in their original positions. This property is formally proved in Proposition 4.

Section 4.3 demonstrates a *universal transformation*, called *compression*, that within $O(n\sqrt{n})$ moves transforms any pair of connected shapes of the same order to each other, while preserving connectivity throughout its course. Starting from the initial shape S_I computed on a spanning tree T of $G(S)$, enclose the shape into a square box of size n and divide it into sub-boxes of size $\sqrt{n}$, each of which contains at least one sub-tree of T. By moving lines in a way that does not break connectivity, we compress the nodes in a sub-box into an adjacent sub-box towards a parent sub-tree. By carefully repeating this we manage to arrive at a final configuration which is always a compressed square shape. The latter is a type of a *nice* shape (a family of connected shapes defined in 2.1.1), which can be transformed fast into a straight line in linear time. We provide an analysis of this strategy based on the number of *charging phases*, which turns out to be $\sqrt{n}$, each making at most n moves, for a total of $O(n\sqrt{n})$ moves.

Despite the fact that the preceding transformations reveals the underlying transformation complexity, they are not directly applicable to real robotic systems. Hence, the next main goal is to develop the first distributed transformations implementing this linear-strength move. If possible, it will always guarantee to 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.

The central coordinator of the previous transformations have a global control over all individual nodes in the system. Thus, it is fully aware of the number of nodes in the system and their initial/final configuration. Moreover, it is responsible for deciding which line will be pushed and who will push that line. This contains the push direction, distance and timing in a particular sequential or parallel order. Further, the central authority has the power to notify nodes on the line about their move and relocation in a single time-step, as well as to terminate the transformations. Hence, communication takes less time and may not even be an issue in the centralised setting.

However, there are considerable technical challenges that must be overcome in order to develop such a distributed solution. As will become evident, the lack of global knowledge of the individual nodes and the condition of preserving connectivity greatly complicate the transformation, even when restricted to special families of shapes. For example, nodes should deduce whether they are part of a line through local vision and communication, as well as decide who is in charge of pushing. Timing is an essential issue as the pushing node needs to know when to start and stop pushing. When moving or turning, all nodes 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 nodes do not automatically know whether they have been pushed (but it might be possible to infer this through communication and local observation).

In Chapter 5, we establish an algorithmic framework that enables individuals to exploit the linear-strength of line moves in a distributed manner: Consider a discrete system of n simple indistinguishable devices, called *agents*, forming a connected shape S_I on a two-dimensional square grid. 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 (LMC) cycles on the grid. All communication is local, and actuation is based on this local information as well as the agent's internal state.

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 nodes 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 nodes are occupying the consecutive diagonal cells on the grid $(x_1, y_1), (x_1+1, y_1+1), \dots, (x_1+n-1, y_1+n-1)$. In each round, a node $p_i = (x, y)$ moves one step down if $(x-1, y-1)$ is occupied, otherwise it stays still in its current cell. After

$O(n)$ rounds, all nodes form S_L within a total number of $1 + 2 + \ldots + n = O(n^2)$ moves, while preserving connectivity during the transformation. See Figure 2.5.

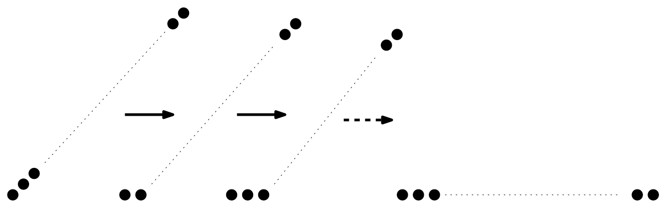


Figure 1.6: A simulation of the simple procedure. From left to right, rounds $0, 1, 2, \ldots, n$.

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 nodes, 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 robotic systems (e.g. programmable matter), moves minimisation is expected to contribute in the deployment and implementation of energy-efficient systems.

Our main contribution then is the first distributed connectivity-preserving transformation that exploits the line moves within a total of $O(n \log n)$ moves, which is asymptotically equivalent to that of the best-known centralised transformations. The algorithm solves the *line formation problem* that allows agents to form a final straight line S_L, starting from any shape S_I, whose associated graph contains a Hamiltonian path.

1.5 Author's publications

The contributions of this thesis are based on co-authored publications in peer-reviewed scientistic journals and conference proceedings. The following is a chronological list of all publications and manuscripts that are currently under submission.

- A. Almethen, O. Michail, I. Potapov. "Pushing Lines Helps: Efficient Universal Centralised Transformations for Programmable Matter". In: *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS) 2019 (pp. 41-59)* [10], also a full version available on arXiv, arXiv:1904.12777 [11].

- A. Almethen, O. Michail, I. Potapov. "Pushing Lines Helps: Efficient Universal Centralised Transformations for Programmable Matter". In: *Theoretical Computer Science. 2020; 830:43-59* [14].

- A. Almethen, O. Michail, I. Potapov. "On efficient connectivity-preserving transformations in a grid". In: *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS) 2020 (pp. 76-91)* [12].

- A. Almethen, O. Michail, I. Potapov. "On efficient connectivity-preserving transformations in a grid". To appear in: *Theoretical Computer Science* [9], also available on arXiv, arXiv:2005.08351 [13].

- A. Almethen, O. Michail, I. Potapov. "Distributed Transformations of Hamiltonian Shapes based on Line Moves". In: *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS) 2021 (pp. 1-16)* [15]. **Invited for the TCS journal special issue on ALGOSENSORS 2021**.

- A. Almethen, O. Michail, I. Potapov. "Distributed Transformations of Hamiltonian Shapes based on Line Moves". *Under journal submission*, also available on arXiv, arXiv:2108.08953 [16].

Chapter 2

Discrete Shape Formation

In this chapter, we provide a generic algorithmic framework for our discrete shape formation problems. It brings together all notations, definitions and basic facts that are used throughout the thesis.

First, we formally define the overall settings of the centralised system (the distributed version is later introduced separately in Chapter 5 due to consistency). Following that, in Section 2.1, we present a formal description of *connected shapes*, then focus particularly on a multiple set of discrete shapes addressed in the transformations. Next, we introduce our shape formation model in Section 2.2, including the linear-strength mechanism alongside other related special-case models. After that, Section 2.3 formally defines all of the shape formation problems considered in this thesis. We then provide several properties that typically facilitate the development of our transformations. The final section, Section 2.5, establishes the first lower bounds of this models for two restricted sets of transformations.

2.1 Discrete shapes

All transformations considered in this thesis operate on a two-dimensional square grid, in which each cell has a unique position of non-negative integer coordinates (x, y), where x represents columns and y denotes rows in the grid. A set of n *nodes* (or called *agents*) on the grid forms a shape S (of the order n), where every single node $u \in S$ occupies only one cell, $cell(u) = (u_x, u_y)$. Nodes and agents are used interchangeably to denote entities of centralised and distributed systems, respectively. A node u can be indicated at any given time by the coordinates (u_x, u_y) of the unique cell that it occupies at that time. A node

$v \in S$ is a *neighbour* of (or *adjacent* to) a node $u \in S$ if and only if their coordinates satisfy $u_x - 1 \leq v_x \leq u_x + 1$ and $u_y - 1 \leq v_y \leq u_y + 1$ (i.e. their cells are adjacent vertically, horizontally or diagonally). Throughout, all logarithms are to base 2.

Definition 1 (A discrete shape)**.** *A discrete shape S is a finite set of nodes on a two-dimensional square grid.*

A shape S is *associated* with a graph $G(S) = (V, E)$, where $u \in V$ iff u is a node of S and $(u, v) \in E$ iff u and v are neighbours in S. In other words, $G(S)$ is the graph induced by the occupied nodes of S. Then, a *connected shape* is defined as follows.

Definition 2 (A connected shape)**.** *A shape S is connected iff $G(S)$ is a connected graph.*

A spanning tree of $G(S)$ is denoted by $T(S)$ (or just T when clear from context). Whenever we state that such a tree is given we make use of the fact that $T(S)$ can be computed in polynomial time. In what follows, n denotes the number of nodes in a shape under consideration. The following definitions from [89] shall be useful for our transformations. To illustrate the shape, we colour black any cell occupied by a node (as in Figure 2.1).

Definition 3. *A hole H of S is a set of empty cells enclosed by non-empty cells that are occupied by nodes $u \in S$, such that any simple path of infinite length that starts from an empty cell in the hole $h \in H$ and moves, only vertically and horizontally, must pass through a black cell of a node $u \in S$.*

Definition 4. *A compact shape is a connected shape that does not contain any holes.*

Definition 5. *The perimeter (border) of S is defined as a polygon of unit length line segments, which surrounds the minimum-area of the interior of S.*

Definition 6. *The surrounding layer of a connected shape S consists of all empty cells in the grid that share at least a line segment or a corner with the S's perimeter.*

Definition 7. *The external surface of a connected shape S consists of all non-empty cells in the grid that share at least a line segment or a corner with the S's perimeter.*

Figure 2.1 depicts the perimeter, surrounding layer and external surface of S as a yellow line, grey cells and cells of black spherical nodes, respectively. The following proposition shows that the external surface of a connected shape S is connected (proved in [89]).

Proposition 1. *The surrounding layer of any connected shape S is itself a connected shape.*

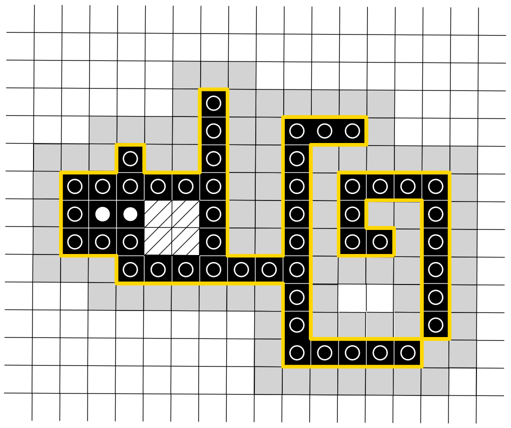


Figure 2.1: All nodes of S occupy the black cells, where black spherical nodes reside on the external surface of S. The surrounding layer's cells are coloured grey, while the yellow line depicts the perimeter of S. The dashed black cells define a hole of S.

Proof. The proof is a modification of Proposition 2 from [89] to fit our model. Assume S is connected, then the perimeter of S is connected too; and hence, it forms a cycle. Each segment of the perimeter is contributed by two cells, belonging to the external surface and the surrounding layer (recall Definitions 5, 6 and 7). Now, if one walks on the perimeter (vertically or horizontally) or turns (left or right) clockwise or anticlockwise at any segment, one of the following cases will occur:

- Pass through two adjacent vertical or horizontal cells on the surrounding layer and the external surface of S.

- Stay put at the same position (cell) on the external surface and move through three neighbouring cells connected perpendicularly on the surrounding layer of S.

- Stay put at the same position (cell) on the surrounding layer and pass through three neighbouring cells connected perpendicularly on the external surface of S.

- Stay put at the same position (cell) on the surrounding layer and:

  1. Either pass through two neighbouring cells connected diagonally on the external surface of S.
  2. Or pass through three neighbouring cells connected perpendicularly on the external surface of S.

Thus, all cases above preserve the connectivity of the surrounding layer and the external surface of S. $\square$

2.1.1 Family of shapes

Now, we define different essential shapes and introduce some classes of connected discrete shapes that are considered in this thesis.

Definition 8 (A straight line). *A straight line S_L is a connected shape of n nodes occupying a sequence of consecutive cells vertically in $(x, y), (x, y+1), \ldots, (x, y+n-1)$ or horizontally in $(x, y), (x+1, y), \ldots, (x+n-1, y)$ (not, e.g. diagonally).*

Definition 9 (A diagonal shape). *A diagonal line shape S_D is a connected shape of n nodes occupying a sequence of consecutive cells diagonally in $(x, y), (x+1, y+1), \ldots, (x+n-1, y+n-1)$.*

Figure 2.2(c) shows an example of the diagonal line shape. Now, we consider two family of more general shapes, *Central Line* shapes, to use as efficient intermediate shapes at more complex transformation and *Hamiltonian shapes* to exploit as an infinite class of connected shapes, which offers insight and reveals useful properties towards the future possible development of universal transformations.

Central Line shapes

Let us first introduce a family of Central Line shapes, which we also call $\mathscr{NICE}$ shapes. A *central line* or a *nice* shape is, informally, any connected shape that contains a particular line called the *central line* (denoted L_C). Intuitively, one may think of L_C as a supporting (say horizontal) base of the shape, where each node u not on L_C must be connected to L_C through a vertical line. Figure 2.2 shows some examples of a *nice* shape and non *nice* shapes.

Definition 10 (A nice shape). *A nice shape $S \in \mathscr{NICE}$ is a compact connected shape, which contains a central line $L_C \subseteq S$, such that every node $u \in S \setminus L_C$ is connected to L_C via a line perpendicular to L_C*

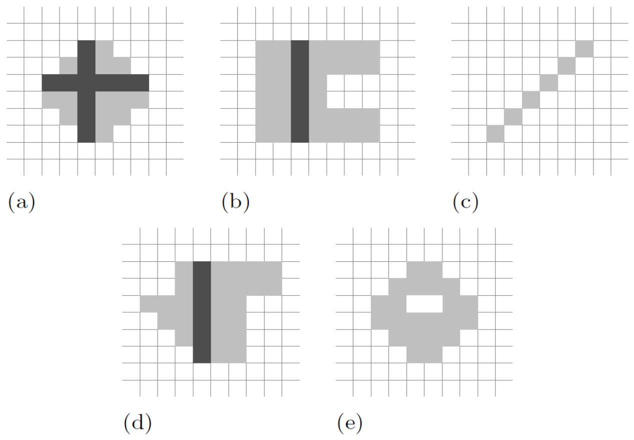


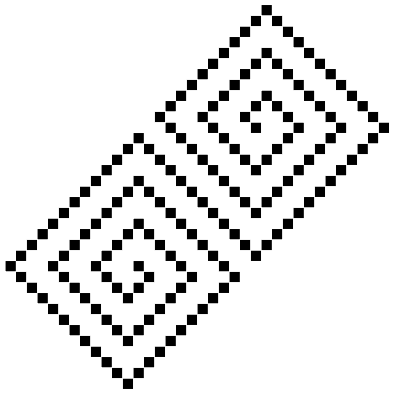
Figure 2.2: The central line L_C occupies black cells of *nice* shapes in (a), (b) and (d). The shape in (c) is not *nice* (due to the lack of L_C) as is the shape in (e), which has a hole (white cells), preventing the existence of L_C.

Hamiltonian shapes

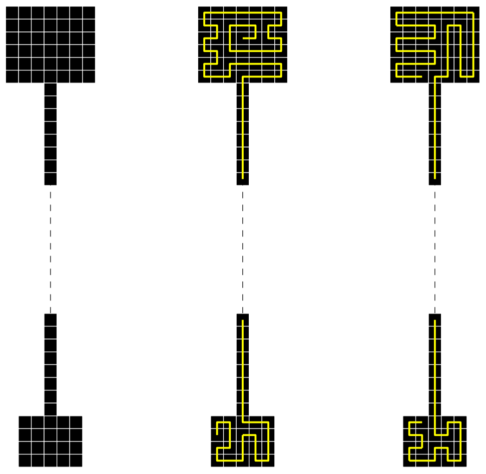
We define another family of shapes, called Hamiltonian shapes and denoted $\mathcal{H}$. Figure 2.3 presents some examples of this class of shapes.

Definition 11 (A Hamiltonian shape)**.** *A shape S is called Hamiltonian iff $G(S) = (V, E)$ contains a path starting from a node $u \in V$, visiting every node in V exactly once and ending at a node $v \in V$, where $v \neq u$.*

In this thesis, the diagonal S_D and the straight line S_L are considered to be an extreme-case for a pair of shapes that can be transformed to each other. On a two-dimensional square grid, S_L represents an extreme instance of the Central Line shapes, that is, S_L contains the longest central line (of length n) that a connected shape can have. Likewise, S_D is another hard-case shape belonging to the family of Hamiltonian shapes due to the fact that all nodes of S_D occupy n distinct rows and columns, that is, it draws n lines on the grid, each of length 1. Further, the $O(n^2)$ individual node distance between the two shapes (S_D, S_L), as well as the initial lack of long line movements, reinforce this intuition. See an illustration in Figure 2.4. A more detailed discussion of this is given later in Section 2.5.



(a) A double-spiral shape



(b) A shape of two different Hamiltonian paths in yellow.

Figure 2.3: Examples of Hamiltonian shapes denoted $\mathcal{H}$.

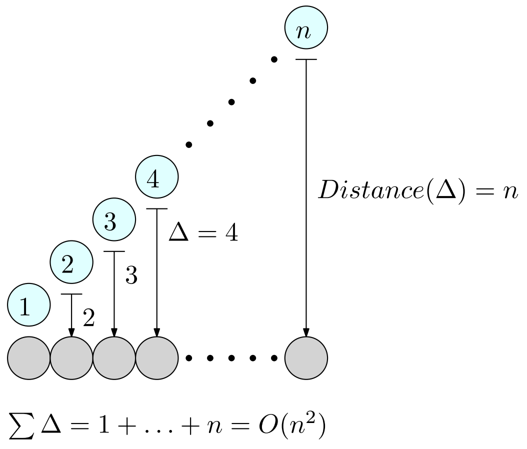


Figure 2.4: The $O(n^2)$ individual node distance between S_D (blue nodes) and S_L (grey nodes).

2.2 The shape formation model

In this thesis, we investigate a linear-strength transformation model, called the *line-pushing model*. A line L is a sequence of consecutive non-empty cells occupied by nodes, $u \in S$, vertically or horizontally (not, e.g. diagonally). A ***line move*** is an operation by which all nodes of L move together in parallel in a single move, towards an empty *cell* adjacent to one of L's endpoints. That is, one of L's endpoints pushes all nodes of L in parallel in a single time-step on one position in a given (one of the four possible) direction of a grid. A line move may also be referred to as a *step* (or *move* or *movement*) and time is discrete and measured in number of moves throughout. A move in this model is equivalent to choosing a node u and a direction $d \in \{up,\ down,\ right,\ left\}$ and moving u one position in direction d. This will additionally push by one position the whole line L of nodes in direction d, L (possibly empty) starting from a neighbour of u in d and ending at the first empty cell.

2.2.1 The linear-strength model

More formally and in slightly different terms: A line $L = (x, y), (x+1, y), \ldots, (x+k-1, y)$ of length k, where $1 \leq k \leq n$, can push all its k nodes rightwards in parallel in a single move to positions $(x+1, y), (x+2, y), \ldots, (x+k, y)$ iff there exists an empty cell to the right of L at $(x+k, y)$. The 'down', 'left', and 'up' moves are defined symmetrically, by rotating the whole system $90°, 180°$, and $270°$ clockwise, respectively.

Definition 12 (A permissible line move)**.** *Let L be a line of k nodes, where $1 \leq k \leq n$. Then, L can move as follows (depending on its original orientation, i.e. horizontal or vertical):*

1. Horizontal. *Can push all k nodes rightwards in parallel in a single move from $(x, y), (x+1, y), \ldots, (x+k-1, y)$ to positions $(x+1, y), (x+2, y), \ldots, (x+k, y)$ iff there exists an empty cell to the right of L at $(x+k, y)$. Similarly, it can push all k nodes to the left to occupy $(x-1, y), (x, y), \ldots,\ (x+k, y)$, iff there exists an empty cell at $(x-1, y)$.*

2. Vertical. *Can push all k nodes upwards in parallel in a single move from $(x, y), (x, y+1), \ldots,\ (x, y+k-1)$ into $(x, y+1), (x, y+2), \ldots, (x, y+k)$, iff there exists an empty cell above L at $(x, y+k)$. Similarly, it can push all k nodes down to occupy $(x, y-1), (x, y), \ldots, (x, y+k)$, iff there exists an empty cell below L at $(x, y-1)$.*

2.2.2 The sliding and rotation model

There are related settings in which the available moves to the nodes are rotation and sliding [64, 89]. First, the sliding operation is equivalent in all those models, that is, if a node u is located at a cell of the grid, $u = (x, y)$, then u can slide right to an empty cell at $(x+1, y+1)$

over a horizontal line of length 2, such as in Figure 2.5(a). On the other hand, the rotation models typically perform a single operation to rotate a node $u_1 = (x, y)$ around another $u_2 = (x, y-1)$ by a 90° clockwise rotation iff there exist two empty cells at $(x+1, y)$ and $(x+1, y+1)$, see Figure 2.5(b).

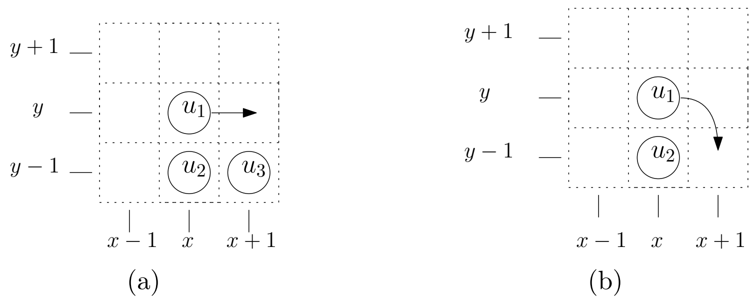


Figure 2.5: (a) An example of sliding u_1 over u_2 and u_3 to an empty cell to the right. (b) Rotate u_1 a 90° clockwise around u_2.

2.3 Shape formation problem

A number of nodes are given in the form of a (typically connected) shape S_I lying on a two-dimensional square grid, the *shape formation problem* is to transform (reconfigure) S_I into a desired target shape S_F via a sequence of valid movements of individual nodes. More formally, the shape formation problem can be defined as a tuple $M = (I, F)$, where I is the set of initial shapes and F is the set of final shapes.

A configuration of the system is defined as a mapping $C\colon \mathbb{Z} \times \mathbb{Z} \to \{0, 1\}$, where $C(x, y) = 0$ if cell(x, y) is empty or $C(x, y) = 1$ if cell(x, y) is occupied by a node. Equivalently, a configuration can be defined as a set $\{(x, y) : x, y \in \mathbb{Z} \text{ and } C(x, y) = 1\}$. Let C_0 denote the initial configuration of the system. We say that C' is *directly reachable* from C and denoted $C \to C'$, if C can be transformed into C' in one line move. Moreover, C' is *reachable* from C, denoted $C \to^* C'$, if there is a sequence of configurations $C = C_1, C_2, \ldots, C_t = C'$ such that $C_i \to C_{i+1}$ holds for all $i \in \{1, 2, \ldots, t-1\}$.

Throughout this thesis, we consider a number of the shape formation problems defined formally as follows:

DiagonalToLine. Given an initial connected diagonal line S_D and a target vertical or horizontal connected spanning line S_L of the same order, transform S_D into S_L, without necessarily preserving the connectivity during the transformation.

DiagonalToLineConnected. Restricted version of DiagonalToLine in which connectivity must be preserved during the transformation.

HamiltonianLine. Given any initial Hamiltonian shape S_I, 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.

HamiltonianConnected. Given a pair of connected Hamiltonian shapes (S_I, S_F) of the same order, where S_I is the initial shape and S_F the target shape, transform S_I into S_F while preserving connectivity throughout the transformation.

UniversalTransformation. Give a general transformation, such that, for all pairs of shapes (S_I, S_F) of the same order, where S_I is the initial shape and S_F the target shape, it will transform S_I into S_F, without necessarily preserving connectivity during its course.

UniversalConnected. Given any pair of connected shapes (S_I, S_F) of the same order, where S_I is the initial shape and S_F the target shape, transform S_I into S_F while preserving connectivity throughout the transformation.

2.4 The linear-strength transformation properties

As already mentioned, there are related settings in which any pair of connected shapes S_I and S_F of the same order can be transformed to each other[1] while preserving the connectivity throughout the course of the transformation.[2] This, for example, has been proved for the case in which the available moves to the nodes are rotation and sliding [64, 89]. We now show that the models of [64] and [89] are special cases of our model, implying that all transformations established there (with their running time at most doubled, including

[1]We also use $S_I \rightarrow S_F$ to denote that shape S_I can be transformed to shape S_F.

[2]In this thesis, whenever transforming into a target shape S_F, we allow any placement of S_F on the grid, i.e., any shape S'_F obtained from S_F through a sequence of rotations and translations.

universal transformations) are also valid transformations in the present model.

Proposition 2. *The rotation and sliding model is a special case of the line-pushing model.*

Proof. We establish a technique to prove that our model is able to simulate rotation and sliding models of [64, 89] on a two-dimension square grid system. First, the sliding operation is equivalent in all those models, that is, if a node u is located at a cell of the grid, $u = (x, y)$, then u can slide right to an empty cell at $(x+1, y+1)$ over a horizontal line of length 2, such as in Figure 2.5(a). The 'down', 'left', and 'up' moves are defined symmetrically, by rotating the whole system $90°, 180°$, and $270°$ clockwise, respectively. Now, the presented model is capable of performing the same *sliding* rule in those models, i.e., push a line of length 1 vertically or horizontally, as explained in Definition 12. For rotation, all mentioned models perform a single operation to rotate a node $u_1 = (x, y)$ around another $u_2 = (x, y - 1)$ by a $90°$ clockwise iff there exists two empty cells at $(x + 1, y)$ and $(x + 1, y + 1)$, see Figure 2.5(b). Analogously, this holds for all possible rotations by again rotating the whole system $90°, 180°$, and $270°$ clockwise, respectively. Still, the rotation mechanism is also adopted by this model following Definition 12, and actually it costs twice for a single rotation to take place, compared with others. Subsequently, it implies that all transformations established there (with its running time at most doubled), including universal transformations and preserving connectivity (recall Proposition 1), are also valid transformations in the present model. □

We are now ready to show the following lemmas which will be used multiple times during our transformations:

Lemma 1. *It is possible to turn a horizontal line of length k into a vertical line of length k or vice versa in $2k - 2$ steps.*

Proof. To simplify the argument, assume that a two-dimensional square grid contains only a straight line L_1 of k nodes at $(x, y), \dots, (x, y + k - 1)$, where $k \geq 1$, and empty cells on $(x + 1, y), \dots, (x + k - 1, y)$, as depicted in Figure 2.6. This assumption can be dropped easily when L_1 is a part of a connected shape S in the grid. Now, L_1 wants to change its direction; for the sake of simplicity, let us assume that L_1 turns from vertical to horizontal, where L_1 occupies k consecutive rows and a single column. Hence, the first bottommost node, $u_1 \in L_1$, moves one position right to occupy an empty cell on $(x + 1, y)$, which consequently creates a new empty cell at (x, y) that was occupied by u_1. Then, by linear-strength pushing mechanism, the consecutive $k - 1 \in L_1$ nodes push down one move altogether in parallel in a single-time-move towards the new empty cell (x, y) to occupy positions $(x, y), \ldots, (x, y + k - 2)$. Thus, L_1 consists now of $k - 1$ nodes occupying $k - 1$ consecutive rows, as shown in Figure 2.6(a), (b) and (c). Observe that u_1 take one time-step to move right, and all $k - 1$ nodes push down in one time-step. By repeating this at most $2k - 2$ steps, it shall completely turn L_1 to occupy a single row and k consecutive columns. Therefore, any straight line of k nodes changes its direction in a number of steps at most twice its length, $2k - 2 = O(k)$.

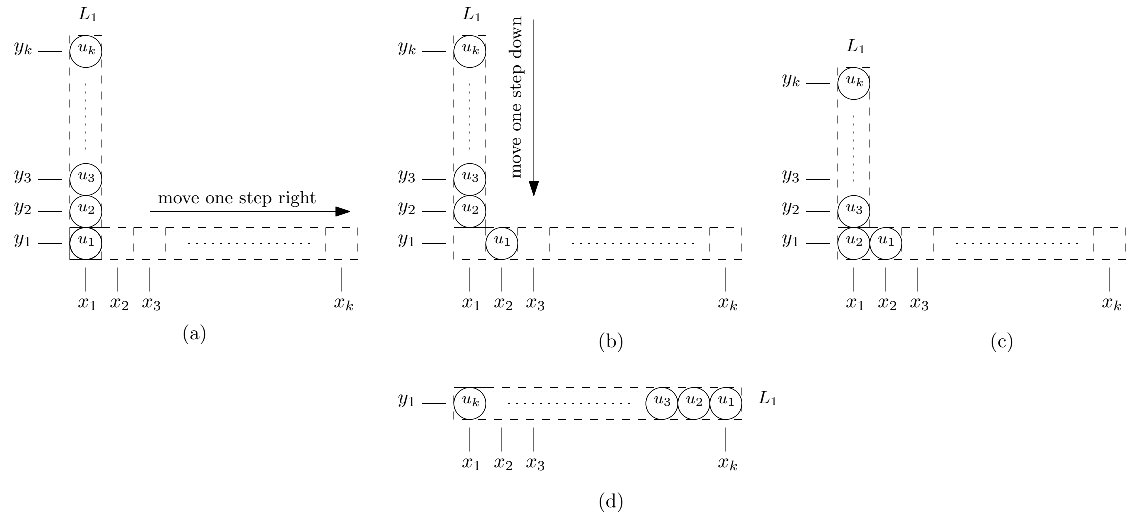


Figure 2.6: A line of k nodes changes orientation by two consecutive steps per node. (a) Move u_1 one step right. (b) and (c) All $k - 1$ nodes push down altogether in a single step. In (d), the line has finally transformed from vertical to horizontal after $2k$ steps.

□

A property that typically facilitates the development of transformations, is reversibility of moves. To this end, we next demonstrate that line moves are reversible.

Lemma 2 (Reversibility). *Let (S_I, S_F) be a pair of connected shapes of the same number of nodes n. If $S_I \to S_F$ ('$\to$' denoting 'can be transformed to via a sequence of line moves') then $S_F \to S_I$.*

Proof. First, we should prove that each single line move is reversible. Figure 2.7 left shows four nodes forming two vertical and one horizontal lines, $L_1 = \{u_1, u_2\}$, $L_2 = \{u_3, u_4\}$ and $L_3 = \{u_2, u_3\}$, respectively. Assume this configuration has no more space to the left, beyond the dashed line; therefore, L_1 moves to occupy the empty cell $(i+2, j+1)$. Now, all L_1 nodes are moving altogether to fill in positions $(i+1, j+1)$ and $(i+2, j+1)$, as depicted in Figure 2.7 right. Consequently, the previous line move creates another empty cell at $(i, j+1)$, which can be occupied reversibly by L_1. Since every line move is reversible, it implies that reversibility holds for any finite sequence of line moves. □

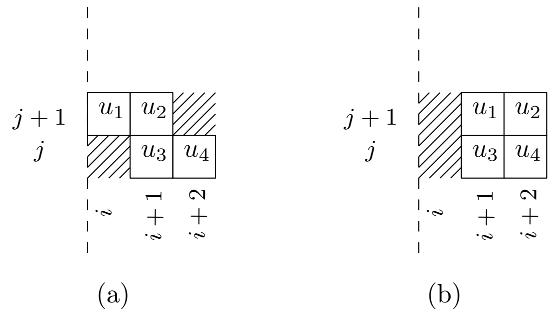


Figure 2.7: An example of a reversible line move.

By reversibility (Lemma 2), we provide the following proposition for nice shapes (Central Line shapes):

Proposition 3. *Let $(\mathscr{A}, \mathscr{B}) \in \mathscr{NICE}$ be a pair of nice shapes of the same order. Then $\mathscr{A} \to \mathscr{B}$ and $\mathscr{B} \to \mathscr{A}$ in $O(n)$ steps.*

Proof. Let $S_{\mathscr{NICE}}$ be a *nice shape* of order n that contains a central line of i nodes, for some $1 \leq i \leq n$, and assume without loss of generality, that L_C is horizontal and occupies row y_1. By Definition 10, there will be $n-i$ nodes, where all are connected to L_C via a line orthogonal to L_C. Similarly, we can say there are $L_1, L_2, \ldots, L_w$ vertical lines of total $n-i$ nodes, where $1 \leq w < n$, which are all perpendicularly connected to L_C. Now, by Lemma 1, such a vertical line would perform a number of steps twice its length to change its direction, occupy row y_1 and be an extension to L_C. Then, it follows that all those vertical lines requires a total cost of at most $2(n-i) = O(n)$ steps to eventually occupy n consecutive cells in row y_1 where a straight line of order n remains. By Lemma 2, we conclude that any pair of *nice shapes* are transformable to each other via a straight line in $O(n)$ steps. □

We define a *rectangular path* P over the set of cells as $P = [c_1, c_2, c_3, \ldots, c_k]$, where $c_i, c_{i+1} \in \mathbb{Z} \times \mathbb{Z}$ are two cells adjacent to each other either vertically or horizontally, for all

$i \in \{1, 2, \ldots, k-1\}$. Given any *rectangular path* P, by C_P we donate the configuration of P, which is the subset of C (configuration of the system) restricted to the cells of P. The following proposition proves a basic property of line moves which will be a core technical tool for our Hamiltonian shapes transformations.

Proposition 4 (Transparency of Line Moves)**.** *Let S denote any shape, $L \subseteq S$ any line and P a rectangular path starting from a position adjacent to one of L's endpoints. There is a way to move L along P, while satisfying all the following properties:*

1. No delay: *The number of steps is asymptotically equal to that of an optimum move of L along P in the case of C_P being empty (i.e., if no cells were occupied). That is, L is not delayed, independently of what C_P is.*
2. No effect: *After L's move along P, $C'_P = C_P$, i.e., the cell configuration has remained unchanged. Moreover, no occupied cell in C_P is ever emptied during L's move (but unoccupied cells may be temporarily occupied).*
3. No break: *S remains connected throughout L's move.*

Proof. Whenever L walks through an empty cell (x, y) of P, a node $u \in L$ fills in (x, y). If L pushes the node u of a non-empty cell of P, a node $v \in L$ takes its place. When L leaves a non-empty cell (x, y) that was originally occupied by node v, L restores (x, y) by leaving its endpoint $u \in L$ in (x, y). Finally, Figure 2.9 shows how to deal with the case in which L turns at a non-empty corner-cell (x, y) of P, which is only connected diagonally to a non-empty cell of S and is not adjacent to any cell occupied by L. Figure 2.8 shows an example of configuration C_P.

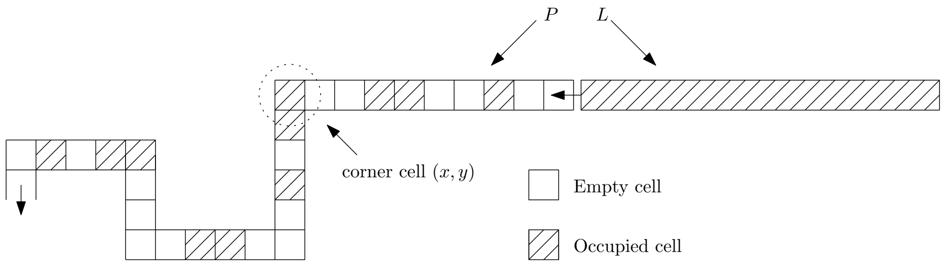


Figure 2.8: A path P of a given configuration C_P. A line L will pass along P.

Now assume that L turns at a non-empty corner cell (x, y) of P (say without loss of generality, from horizontal to vertical direction). Typically the node occupying the corner cell(x, y) moves vertically one step along P, and then L pushes one move to fill in the empty cell (x, y) by a node $u \in L$. Unless (x, y) is being only connected diagonally to a non-empty cell that is not a neighbour of any node $u \in L$. Figure 2.9 shows how to deal with the case in which L turns at a non-empty corner-cell (x, y) of P, which is only connected diagonally to a non-empty cell of S and is not adjacent to any cell occupied by L.

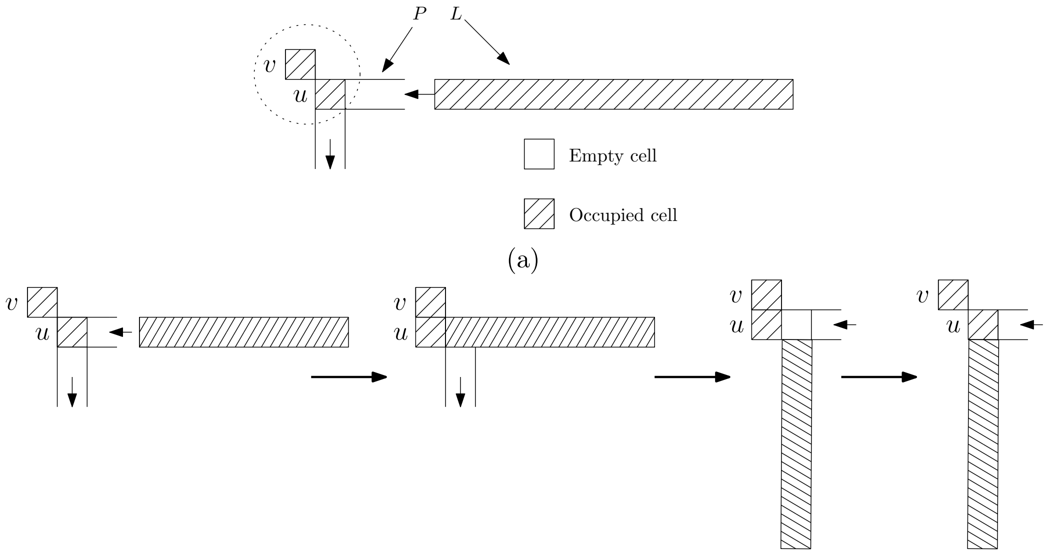


Figure 2.9: (a) A line L moving through a path P and arriving at a turning point of P. u occupies a corner cell of P, and v occupies a cell of S and is only connected diagonally to u while not being adjacent to any cell occupied by L. (b) L pushes u one position horizontally and turns all of its nodes vertically. Then u moves back to its original position in P. All other orientations are symmetric and follow by rotating the shape 90°, 180° or 270°.

Therefore, it always temporarily maintain global connectivity and restores all of those nodes to their original positions. Hence, L's move takes a number of moves to pass through any C_P equal to or even less than its optimum move in the case of empty C_P. Therefore, L can *transparently* walk through *any* configuration S (independently of the latter's density) in a way that: (i) preserves connectivity of both L and S and (ii) as soon as L has gone through it, S has been restored to its original state, that is, all of its nodes are lying in their original positions. □

2.5 Lower bounds on line moves

The question of lower bound on line moves appears intriguing, i.e. to answer whether there is an $o(n \log n)$-time transformation (e.g. linear) or whether there is an $\Omega(n \log n)$-time lower bound matching our best transformations (even when connectivity can be broken). We suspect the latter but lack sufficient facts to substantiate or verify it. As will become seen, it seems challenging to lower bound in the general case, most likely due the trade-off between the cost of inherent individual distances and the cost of parallelism inherent in line moves. Thus, this section reports our attempts on this and presents the first lower bounds established for this model restricted to two sets of transformations. It is expected to provide useful discussion that might help in proving the lower bounds on line moves.

In this section, we identify the connected shape of a diagonal line S_D (see Definition 9) as a potential worst-case initial shape to be transformed into a line S_L (see Definition 8). This intuition is supported by the $O(n^2)$ individual node distance between the two shapes and by the initial unavailability of long line movements: the transformation may move long lines whenever available, but has to pay first a number of individual and small line movements in order to construct longer lines.

In this benchmark (special) case, the trivial lower and upper bounds $\Omega(n)$ and $O(n^2)$, respectively, hold. Moreover, observe that a sequential gathering of the nodes starting from the top right and collecting the nodes one after the other into a snake-like line of increasing length is still quadratic, because, essentially, for each sub-trip from one collection to the next, the line has to make a 'turn', meaning to change both a row and a column. In models of single individual moves (e.g. rotation and sliding in [64, 89]), this costs a number of steps equal to the length of the line, that is, roughly, $1 + 2 + \ldots + (n-1) = \Theta(n^2)$ total time. In other words, each node in S_D occupies a unique row and column where $n = \#rows = \#columns$ by which a transformation pays the maximum number of moves to transform it into S_L . Further, the diagonal S_D is very similar to the staircase worst-case shape of [89]. In what follow, we establish $\Omega(n \log n)$ lower bounds for two restricted sets of transformations. These are the first lower bounds for this model and are matching the best known $O(n \log n)$ upper bounds.

Given an associated graph $G = (V, E)$ of S_D in which V is a set of nodes in S_D and E is non-negative edge weights (Manhattan distance between nodes). Then, to simplify the problem, gather all nodes on S_D at the bottom-most node. That is, every node in S_D must perform one or more hops through other nodes and end up at the bottom-most node.

When going through a node the two or more nodes can continue traveling together and exploit parallelism.

Any such solution to the problem forms a spanning tree $T \subseteq G$, where every leaf to root path corresponds to the hops of a specific node until it reached the end. The cost of each subtree is: $c(T)$ is the total sum of the distances of its edges plus the cost of nodes $c(V)$. Every edge $E(u, v)$ has a cost equal to the distance of moving u to v, where each node has a cost of paying for each internal node of a subtree the number of nodes in its subtree. The latter cost is due to not being able to exploit parallelism whenever turning, and any hop requires another turn. The cost due to distances is just:

$$c(E) = \sum_{e \in E(T)} cost(e), \tag{2.1}$$

and the cost of internal nodes is equal to:

$$c(V) = \sum_{i=1}^{d(T)} i \cdot v \in d(T)_i, \tag{2.2}$$

where $d(T)$ is the depth of tree T and $d(T)_i$ is the number of nodes at level i. The total cost in number of moves given by such a tree T is the sum of 2.1 and 2.2:

$$c(T) = c(E) + c(V). \tag{2.3}$$

Now, the two sums seem to give some trade-off. If the depth is very small, then the cost due to distances seems to increase (e.g. if all nodes travel into one hop, they all pay their distances and the cost is quadratic). This approach is similar to any sequential transformation of individual movements which pays a cost of $\Theta(n^2)$ to transform S_D into S_L. The summation of the total individual distances is, $\Sigma\Delta = 0 + 1 + 2 + \ldots + (n-1) = \Theta(n^2)$, independently of whether connectivity is preserved or not during transformations. This is because of the inherent individual distance between S_D and S_L. On the other hand, the tree T of very large depths looks as a spanning line where a lot of parallelism must be exploited. The distance in this case would cost only $c(E) = n - 1$. While the sum of turns at each node becomes quadratic, $c(V) = n^2$. Therefore, we observe that more balanced trees of logarithmic depth, such as binomial trees, manage to balance both sums and give total cost $n \log n$. Due to the trade-off, it does not seem easy to lower bound in the general case. Further, it does not seem easy to lower bound the edges-sum even by some parameters depending on the depth (so that both sums will be using similar parameters). It might not even be related to that parameter. Therefore, we tried to further simplify the problem by restricting the solutions to extremely limited depths. Below, we have successfully managed to establish some special-case lower bounds for this problem.

It is quite obvious that no uniform strategy can yield better bound than the $O(n \log n)$-time universal strategy of Section 3.2, by simply increasing the number of lines that are merging in every phase to decrease the number of phases. Hence, we have the following proposition.

Proposition 5. *Any strategy represented by a balanced tree performs at most $O(n \log n)$ steps.*

Proof. Observe that such a strategy is essentially trying to increase the degrees of the nodes of a balanced tree and decrease its depth. For example, take any merging parameter $k \geq 2$. Notice that the $O(n \log n)$-time transformation (*DL-Doubling* in Chapter 3) has k = 2, as it is merging pairs of lines and get $\log n$ phases. So, in every phase i we are going to partition the L lines into L/k groups of k consecutive lines each and merge the lines within each group into a single line.

First, in phase 1, $L = n$, and we are partitioning into n/k groups. For each group we are paying at least k^2 asymptotically to merge the lines in it. Therefore for phase 1 we pay $(n/k)k^2 = nk$ (this is similar also to the $O(n\sqrt{n})$-time transformation in Section 3.2, but there it only did it once and gathered all the lines to the bottom and not in any further phases). Then in phase 2 $L = n/k$, we are partitioning into $L/k = n/k^2$ groups. Each group is paying at least k^3 asymptotically, because the distance between consecutive lines has now increased to k (roughly). Thus, it gives again cost at least nk, which should hold for the other phases.

Now, observe that this strategy gives $\log_k n = \log n \log k$ phases. If each is paying nk, then the total cost is $(nk)(\log n \log k) = n \log n (k \log k)$, which for all $k \geq 2$ is at least $2n \log n = \Omega(n \log n)$. This would be helpful because it excludes any attempts to get a better than the $O(n \log n)$-time transformation by simply playing with the degrees of the tree in a uniform way (which in turn decreases its depth and thus the number of phases). $\square$

2.5.1 An $\Omega(n \log n)$ lower bound for the 2-HOP tree

We start to study a special case lower bound for all solutions that represented by a tree T of minimum depths. Assume any such solution moves all nodes in only one-way via shortest paths towards their target positions. Once a node joins other nodes, they do not split after that during the transformation. Let $d(T)$ denote the depth of the tree. For $d(T) = 1$, the tree becomes a star, and the total cost is quadratic in this case, due to the summation of individual distances $c(E) = 0 + 1 + 2 + \ldots + (n-1) = \Theta(n^2)$.

Then, we investigate the tree T of depth at most 2, $d(T) = 2$. Observe that we *asymptotically* pay for each node in the tree at least the square of the number of children. The reason is that at most 2 children can be at distance 1, then at most 2 can be nodes of distance 2; hence, at most 2 of distance i in general due to the neighbouring properties of the diagonal on the two-dimensional square grid. Thus, any such tree gives a total of $c(T) = \sum_i d(u_i)^2$, where $d(u_i)$ is the number of children of u_i. That is, the squares of the degrees of all internal nodes, excluding their parent (the root u_i).

Let T of k nodes be a tree of depth 1, as shown in Figure 2.10. Then, the total asymptotic cost of the tree $c(T)$ is at least:

$$c(T) \geq \sum_{i=0}^{k} d(u_i)^2, \tag{2.4}$$

where $d(u_i)$ is the degree of node u_i.

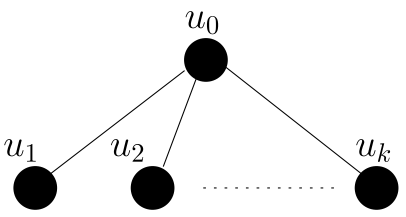


Figure 2.10: A tree T of k nodes.

Given the tree T of k we show the first case of a minimum total cost T must pay if it has a node $u_i \in T$ with degree at least $n \log n$;

Lemma 3. *If* $\exists\ d(u_i) \geq \sqrt{n \log n}$*, then* $c(T) \geq n \log n$*, for all* $0 \leq i \leq k$.

Proof. The proof is straightforward. Consider the tree T of k nodes in Figure 2.10. If $k \geq \sqrt{n \log n}$, then the tree shall have a minimum total cost of $c(T) \geq d(u_0)^2 = k^2 =$

$(\sqrt{n\log n})^2 = n\log n$. In general, if there exists a node $u_i \in T$, for all $0 \le i \le k$, such that $d(u_i) \ge \sqrt{n\log n}$, then the total cost of the tree must be at least $c(T) \ge n\log n$. □

Now, let us assume that all nodes in the tree have degrees less than $\sqrt{n\log n}$. Thus, we show the lower bound of Lemma 3 holds in this case.

Lemma 4. *Let $d(u_i) < \sqrt{n\log n}$ $\forall i$, where $0 \le i \le d(u_0) = k$. Then, $c(T) > n\log n$.*

Proof. Given a tree T of n nodes that has depth of 2, and a subtree $T' \subseteq T$ of k nodes as in Figure 2.10. So, let $d(u_i) < \sqrt{n\log n}$, for all $0 \le i \le d(u_0) = k$. Assume without loss of generality that the nodes $u_i \in T'$, for all $1 \le i \le d(u_0) = k$, are ordered in non-increasing degrees from left to right (increasing order i), that is, $d(u_1) \ge d(u_2) \ge \ldots \ge d(u_k)$. Hence, there are $n - (k+1) \in T$ nodes remaining to be assigned. As $d(u_1) \in T'$ is the maximum, it must hold that, $d(u_1) \ge \frac{n-(k+1)}{k}$, thus $\frac{n-(k+1)}{k} \le d(u_1) < \sqrt{n\log n}$. Next, there are $n - (k+1) - d(u_1) \in T$ nodes need to be allocated. As $d(u_2) \in T'$ is the maximum among the rest, it must hold that $d(u_2) \ge \frac{n-(k+1)-d(u_1)}{k-1}$, thus $\frac{n-(k+1)-d(u_1)}{k-1} \le d(u_2) < \sqrt{n\log n}$. In general, if a node $d(u_i) \in T'$ is the maximum, then the following must hold that,

$$d(u_i) \ge \frac{n - \left(\sum_{j=0}^{i-1} d(u_i)\right) - 1}{k-(i-1)}, \tag{2.5}$$

thus,

$$\frac{n - \left(\sum_{j=0}^{i-1} d(u_i)\right) - 1}{k-(i-1)} \le d(u_i) < \sqrt{n\log n}. \tag{2.6}$$

Now, plug $i = 1$ and $k = d(u_0)$ in (2.5) yields,

$$d(u_1) \ge \frac{n - d(u_0) - 1}{d(u_0)} > \frac{n - \sqrt{n\log n} - 1}{\sqrt{n\log n}}, \tag{2.7}$$

when $i = 2$, we will get,

$$d(u_2) \ge \frac{n - (d(u_0) + d(u_1)) - 1}{d(u_0)} > \frac{n - 2\sqrt{n\log n} - 1}{\sqrt{n\log n} - 1}. \tag{2.8}$$

For all $1 \leq i \leq d(u_0) = k$, we shall obtain,

$$d(u_i) \geq \frac{n - \left(\sum_{j=0}^{i-1} d(u_i)\right) - 1}{d(u_0) - (i-1)} > \frac{n - i\sqrt{n \log n} - 1}{\sqrt{n \log n} - (i-1)}. \tag{2.9}$$

Then, we plug (2.9) into (2.4), which implies,

$$\begin{aligned} c(T) &> \sum_{i=0}^{d(u_0)} \left[\frac{n - i\sqrt{n \log n} - 1}{\sqrt{n \log n} - (i-1)}\right]^2 > (\sqrt{n \log n})^{-1} \sum_{i=0}^{d(u_0)} (n - i\sqrt{n \log n} - 1)^2 \\ &= (\sqrt{n \log n})^{-1} \left[d(u_0) \cdot n^2 + n \sum_{i=0}^{d(u_0)} (i^2 \log n - 2i\sqrt{n \log n})\right]. \end{aligned} \tag{2.10}$$

We need to bound the summation of (2.10):

$$\begin{aligned} \sum_{i=0}^{d(u_0)} n(i^2 \log n - 2i\sqrt{n \log n}) &= \sum_{i=0}^{d(u_0)} i^2 \log n - \sum_{i=0}^{d(u_0)} 2i\sqrt{n \log n} \\ &= \log n\Big(d(u_0)^3 + d(u_0)^2 + d(u_0)\Big) \\ &- 2\sqrt{n \log n} \cdot d(u_0)^2. \end{aligned} \tag{2.11}$$

Now, plug (2.11) into (2.10), then it will give a total cost of the tree $c(T)$ that asymptotically bounded on:

$$\begin{aligned} c(T) &> (\sqrt{n \log n})^{-1}\Big(n^2 \cdot d(u_0) + n \log n \cdot d(u_0)^3 - \sqrt{n \log n} \cdot d(u_0)^2\Big) \\ &> \frac{n^2 \cdot d(u_0) + n \log n \cdot d(u_0)^3}{\sqrt{n \log n}} - n \log n. \end{aligned} \tag{2.12}$$

Finally, since $d(u_0) > 1$, it implies that,

$$\begin{aligned} c(T) &> \frac{n^2}{\sqrt{n \log n}} - n \log n = \left(\frac{\sqrt{n}}{\log^2 n} - 1\right) n \log n = \\ &= \Omega(n \log n). \end{aligned} \tag{2.13}$$

$\square$

As a result, both Lemmas 3 and 4 show that the total cost of any spanning tree $c(T)$

of n nodes and depth at most $d(T) \leq 2$ is always bounded by $\Omega(n \log n)$.

Theorem 1. *Any 2-HOP spanning tree of n nodes and depth at most $d(T) \leq 2$ has a total cost $c(T)$ of $\Omega(n \log n)$.*

2.5.2 A conditional $\Omega(n \log n)$ lower bound - one way transformation

Now, we present another restricted lower bound for transformations of line moves. Again, our techniques is based on one-way assumption in which all nodes move in one direction via shortest paths towards the target node (e.g. from top to bottommost node in the diagonal). Whenever a node joins other nodes, they continue travelling together and do not split thereafter. Assume a potential placement of S_L horizontally on the bottommost row y_1 or vertically at the leftmost column x_1 of the shape. Without loss of generality, assume all lines are moving down and leftwards. This is convenient as they always push a minimum distance towards their target positions on the potential placement of S_L.

Enclose each individual node of S_D into a square box of dimension $d_\Delta = 1$, this gives a total of n boxes, see the black squares boxes in Figure 2.11. Then, double the dimension $d_\Delta = 2$ to surround $n/2$ boxes of two nodes each, see the red squares boxes in Figure 2.11. Repeat doubling dimensions $\log n$ times, until arriving at a single box of $d_\Delta = n$, which contains all nodes of S_D. Assume that n is a power of 2, the total number of all boxes then shall be $n + n/2 + n/4 + \ldots + 1 = 2n - 1$ boxes, where there are n boxes of $d_\Delta = 1$, $n/2$ of $d_\Delta = 2$, $\ldots$ and 1 box of $d_\Delta = n$.

Now, observe that such a transformation at any order during its course, must pay at least n steps to push n nodes out from their black boxes of $d_\Delta = 1$. Hence, a trivial lower bound of $\Omega(n)$ holds. Further, when a line l_1 of 1 node occupying a box of $d_\Delta = 1$ is pushed one move to cross the boundary, no one will be pushed for free, see a demonstration in Figure 2.12(a). The same argument follows, when a line l_2 of length 2 contained into a 2×2 red box, pushes a distance of 2, say to the left, then no line will leave its red box for free, as depicted in Figure 2.12(b). By this observation, any transformation requires at least $d_\Delta \cdot n/d_\Delta$ moves to evacuate all lines from n/d_Δ boxes of d_Δ, where the dimension $d_\Delta = 2^k$ for all $0 \leq k \leq \log n$. Hence, no line will be pushed for free at any order during the course ot the transformation.

With this, we can then calculate the total minimum moves that must be paid to empty all $2n - 1$ boxes containing lines (of various lengths). That is, any strategy needs to pay a minimum number of $d_\Delta = 2^k$ moves to evacuate each occupied box at any order of its course.

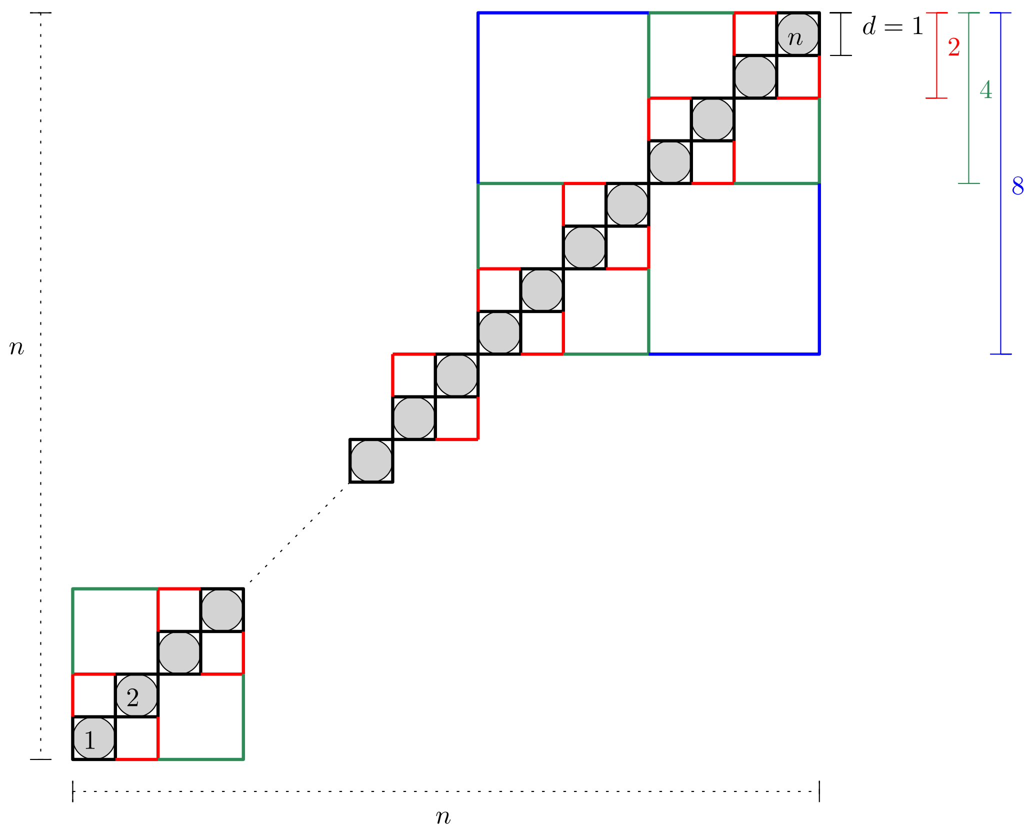


Figure 2.11: Every node of S_D is contained into $\log n$ boxes.

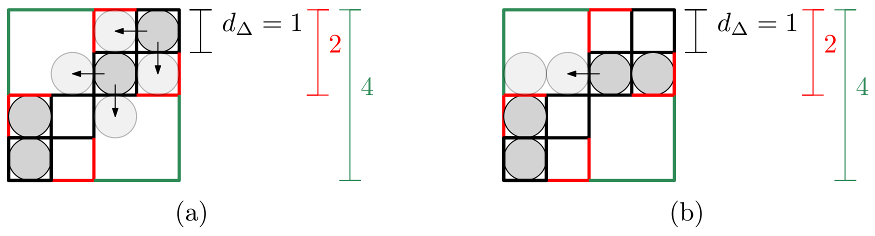


Figure 2.12: (a) If a node (line of length 1) inside a black box pushes one move, no one will be pushed for free. (b) A line of length 2 pushing a distance of 2 towards the left and getting out of its red box.

Thus, the total minimum moves can be given by $(1 \cdot n) + (2 \cdot n/2) + (4 \cdot n/4) + \ldots + (n \cdot 1) = n + n + \ldots + n$. Since we have $\log n$ different dimensions, this gives a total of $n \log n$ minimum number of moves. Hence, any transformation exploiting line moves asks for at least $\Omega(n \log n)$ moves to transfer all n nodes from their initial cells in S_D to their final cells at the potential placement of S_L, i.e. to transform S_D into S_L. Therefore, we can say that:

Proposition 6. *Any shape transformation exploiting line moves and restricted to a one-way direction requires $\Theta(n \log n)$ moves to transform S_D into S_L.*

Chapter 3

Unrestricted transformations

As a first step towards understanding the power of the proposed model, we naturally restrict our attention to centralised transformations, assuming that the connectivity needs not be necessarily preserved during the execution. This unrestricted framework provides more insight into the underlying principle of the parallelism inherent in line moves, which has the potential to be exploited for efficient, i.e. sub-quadratic worst-case, transformations. Furthermore, some of the algorithmic tools in unrestricted centralised transformations might prove useful for more restricted versions.

In this chapter, we consider a number of transformations that not necessarily preserve the connectivity of the shape throughout the execution. In Section 3.1, we first introduce the partitioning transformation of time at most $O(n\sqrt{n})$ moves for the apparent extreme-case of converting the diagonal line shape into a straight line in Section 3.1.1. As the uniform partitioning into segments is optimal for the above type of transformation, we turn our attention into different unrestricted transformations, aiming at further reducing the running time. Hence, Section 3.1.2 presents an alternative transformation based on *successive doubling*, yielding a substantial improvement of at most $O(n \log n)$ moves to transform the diagonal into a line. Then, we show in Section 3.1.3 that a *uniform recursion* approach for this special case obtains an alternative $O(n \log n)$ transformation and could not be applied to further reduce the running time for the diagonal-to-line problem.

Following that, Section 3.2 generalises the techniques created for the preceding benchmark case and introduces universal transformations with equivalent efficiency. Both the $O(n\sqrt{n})$ and the $O(n \log n)$ approaches have been successfully generalised, obtaining universal transformations of worst-case running times $O(n\sqrt{n})$ in Section 3.2.1 and $O(n \log n)$

in Section 3.2.2, respectively. This is accomplished by surrounding the initial shape in a square bounding box and then subdividing the box into square sub-boxes of appropriate size. A single such partitioning into sub-boxes of size $\sqrt{n} \times \sqrt{n}$ is found to be adequate for the first bound, $O(n\sqrt{n})$. We then use a successive doubling strategy for the $O(n \log n)$ bound, this time through phases of increasing sub-box dimension, that is, a new partitioning in each phase.

3.1 The Diagonal-To-Line transformation

As mentioned earlier, we identify the diagonal connected shape S_D of order n (see Definition 9) as an initial potential worst-case shape to be transformed into a straight line S_L. Our goal is then to transform S_D into S_L in which the shape is allowed to break its connectivity throughout transformations. We do this, because this problem seems to capture the worst-case complexity of transformations in the line-pushing model. For example, S_D can be viewed as a staircase shape of stairs of length 1, which is much harder than the staircase worst-case shape of [89] of stairs of length 2. Further, this particular diagonal shape consists of the maximum number of n lines of length 1 across all connected shapes drawn on the square grid.

Below, we demonstrate an $O(n\sqrt{n})$-time strategy, called *DL-Partitioning*, to transform the diagonal S_D into a straight line S_L without necessarily preserving the connectivity during the transformation.

3.1.1 DL-Partitioning: An $O(n\sqrt{n})$-time Transformation

This transformation solves DiagonalToLine as follows. Divide the diagonal into several segments, as in Figure 3.1(a), each segment performs a trivial (inefficient, but enough for our purposes) line formation by moving each node independently to its leftmost column (Figure 3.1(b)). Hence, all segments are transformed into lines (Figure 3.1(c)). Now, transfer each line segment all the way down to the bottommost row of the diagonal S_D (Figure 3.1(d)). Finally, turn the orientation of all line segments to form the target straight line (Figure 3.1(e)).

More formally, let S_D denote a diagonal of n nodes occupying $(x, y), (x+1, y+1), \ldots, (x+n-1, y+n-1)$, such that x and y are the leftmost column and the bottommost row of S_D, respectively. Then, S_D is divided into $\lceil\sqrt{n}\rceil$ segments, $l_1, l_2, \ldots, l_{\lceil\sqrt{n}\rceil}$, each of length $\lfloor\sqrt{n}\rfloor$.

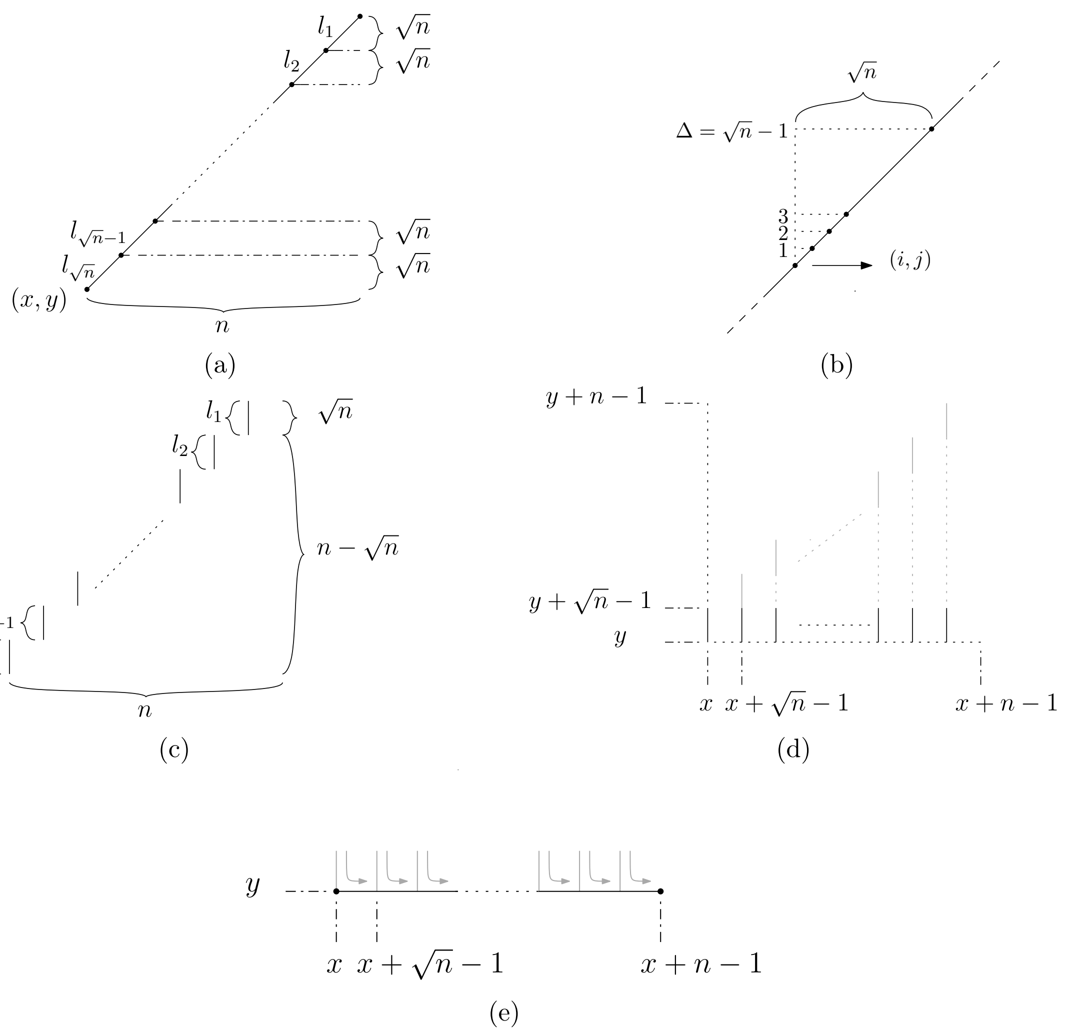


Figure 3.1: (a) Divide the diagonal into $\sqrt{n}$ segments of length $\sqrt{n}$ each. (b) A closer view of a single segment, where $1, 2, 3, \ldots, \sqrt{n} - 1$ are the distances for the nodes to form a line segment at the leftmost column. (c) Each line segment is transferred downwards to the bottommost row of the shape in (d). (e) All line segments turned to fill in the bottommost row, and hence, form the target straight line.

Figure 3.1(a) illustrates the case of integer $\sqrt{n}$. Without loss of generality, *DL-Partitioning* consists of three phases:

- **Phase 1**: Transforms each diagonal segment $l_1, l_2, \ldots, l_{\sqrt{n}}$ into a line segment. Notice that segment l_k, $1 \leq k \leq \sqrt{n}$, contains $\sqrt{n}$ nodes occupying positions $(x + h_k, y +$

$h_k), (x+h_k+1, y+h_k+1), \ldots, (x+h_k+\sqrt{n}-1, y+h_k+\sqrt{n}-1)$, for $h_k = n-k\sqrt{n}$; see Figure 3.1(b). Each of these nodes moves independently to the leftmost column of l_k, namely column $x+h_k$, and occupy $(x+h_k, y+h_k), (x+h_k, y+h_k+1), \ldots, (x+h_k, y+h_k+\sqrt{n}-1)$. By the end of Phase 1, $\sqrt{n}$ vertical line segments have been created (Figure 3.1(c)).

- **Phase 2**: Transfers all $\sqrt{n}$ line segments from Phase 1 down to the bottommost row y. Observe that line segment l_k has to move distance h_k (see Figure 3.1(d)).

- **Phase 3**: Turns all $\sqrt{n}$ line segments into the bottommost row y (Figure 3.1(e)) into positions $(x+h_k, y), (x+h_k+1, y), \ldots, (x+h_k+\sqrt{n}-1, y)$.

Now, we are ready to analyse the running time of all phases of *DL-Partitioning.*

Theorem 2. *DL-Partitioning solves the* DiagonalToLine *problem in $O(n\sqrt{n})$ moves.*

Proof. In the first phase, the cost of the trivial line formation is the run of all distances for $\lceil\sqrt{n}\rceil$ nodes to be gathered at the leftmost column of a single segment l_k. Observe that in Figure 3.1(b), the $\lceil\sqrt{n}\rceil$ nodes of l_k have to move distances of $\Delta = 0, 1, 2, \ldots, (\sqrt{n}-1)$. Therefore, the total run t_1 of all distance of other $\sqrt{n}-1$ nodes in a single segment (except the bottommost node l_k which stays still), is:

$$\begin{aligned} t_1 &= 1+2+\ldots+(\sqrt{n}-1) = \sum_{i=1}^{\sqrt{n}-1} i \\ &= \frac{\sqrt{n}(\sqrt{n}-1)}{2} = \frac{n-\sqrt{n}}{2} \\ &= O(n). \end{aligned} \tag{3.1}$$

Multiplying (3.1) by $\lceil\sqrt{n}\rceil$ implies the total distances T_1 for all segments:

$$\begin{aligned} T_1 &= t_1 \cdot \lceil\sqrt{n}\rceil \\ &= O(n\sqrt{n}). \end{aligned} \tag{3.2}$$

Now, all $\lceil\sqrt{n}\rceil$ line segments move downwards the in phase 2 (except the one already there). Thus, any line segment l_k has to transfer a distance of $n-k\sqrt{n}$ to reach the y bottommost row. The total run T_1 in the second phase is:

$$\begin{aligned}
T_2 &= \sum_{k=1}^{\sqrt{n}-1} (n - k\sqrt{n}) = (n\sqrt{n} - n) - \sum_{k=1}^{\sqrt{n}-1} k\sqrt{n} \\
&= (n\sqrt{n} - n) - \sqrt{n}\left(\frac{\sqrt{n}(\sqrt{n}-1)}{2}\right) \\
&= O(n\sqrt{n}). \qquad (3.3)
\end{aligned}$$

The last phase basically turns (re-orientates) each line of $\lceil\sqrt{n}\rceil$ nodes into the y bottommost row of length n. Now, each line segments l_k contributes a node at the bottommost row y, therefore, we have $\lceil\sqrt{n}\rceil$ nodes are sitting on row y, and the other $n - \lceil\sqrt{n}\rceil$ nodes are waiting to be pushed (*turned*) into the bottommost row of length n. By Lemmas 1 and 9, the transformation fills in the bottommost row y, and each node turns within two moves to reach y, implying a total T_3 for all $n - \lceil\sqrt{n}\rceil$ as follows:

$$\begin{aligned}
T_3 &= 2 \cdot (n - \lceil\sqrt{n}\rceil) = 2n - 2\sqrt{n} = 2(n - \sqrt{n}) \\
&= O(n). \qquad (3.4)
\end{aligned}$$

The sum of (3.2), (3.3) and (3.4) gives the total cost T in moves for all phases,

$$\begin{aligned}
T &= T_1 + T_2 + T_3 \\
&= \frac{n(\sqrt{n}-1)}{2} + \frac{n(\sqrt{n}-1)}{2} + 2(n - \sqrt{n}) \\
&= O(n\sqrt{n}).
\end{aligned}$$

□

3.1.2 DL-Doubling: An $O(n \log n)$-time Transformation

We now investigate another approach (called *DL-Doubling*) that solves DiagonalToLine more efficiently in a total of $O(n \log n)$ moves. This algorithm achieves a better bound of $O(n \log n)$ than the prior one in Section 3.1.1, which was $O(n\sqrt{n})$. The main idea is as follows. The initial configuration can be viewed as n lines of length 1. We start (in *phases*) to successively double the length of lines (while halving their number) by matching them in pairs through shortest paths, until a single straight line remains. Let the lines existing in each phase be labelled $1, 2, 3, \ldots$ from top-right to bottom-left. In each phase, we shall distinguish two types of lines, *free* and *stationary*, which correspond to the odd $(1, 3, 5, \ldots)$ and even $(2, 4, 6, \ldots)$ lines from top-right to bottom-left, respectively. In any phase, only the *free* lines move, while the *stationary* stay still.

In particular, in phase i, every free line j moves via a shortest path to merge with the next (top-right to bottom-left) stationary line $j + 1$. This operation merges two lines of length k into a new line of length $2k$ residing at the column of the stationary line. In general, at the beginning of every phase i, $1 \leq i \leq \log n$, there are $n/2^{i-1}$ lines of length 2^{i-1} each. These are interchangeably free and stationary, starting from a free top-right one, and at distance 2^{i-1} from each other. The minimum number of steps by which any free line of length k_i, $1 \leq k_i \leq n/2$ can be merged with the stationary next to it is roughly at most $4k_i = 4 \cdot 2^i$ (by two applications of turning of Lemma 1). By the end of phase i (as well as the beginning of phase $i+1$), there will be $n/2^i$ lines of length 2^i each, at distances 2^i from each other. The total cost for phase i is obtained then by multiplying $n/2^i$ free lines, each is paying at most $4 \cdot 2^i$ to merge with the next stationary, thus, a linear cost in each one of $\log n$ phases in total. See Figure 3.2 for an illustration of *DL-Doubling*.

More formally, given an initial diagonal line S_D of n nodes. Due to symmetry, it is sufficient to show transformations occurring on one orientation. Initially, the set of $n/2$ *free* nodes are on $(x_1, y_1), (x_3, y_3), \ldots, (x_{n-1}, y_{n-1})$ while the $n/2$ *stationary* occupy $(x_2, y_2), (x_4, y_4), \ldots, (x_n, y_n)$. In each phase i, $1 \leq i \leq \log n$, each line doubles and has a length of 2^i. In phase $i = 1$, all *free* lines of length 1 move a distance $\Delta = 1$ to their left to occupy $(x_2, y_1), (x_4, y_3), \ldots, (x_n, y_{n-1})$ and merge with the next following $n/2$ *stationary* lines. Hence, a total of $n/2$ *vertical* lines of length 2 are created with bottommost nodes occupying $(x_2, y_1), (x_4, y_3), \ldots, (x_n, y_{n-1})$, as depicted in the top of Figure 3.2. In the second phase, those $n/2$ *vertical* lines are arranged into two sets, $n/4$ *free* and $n/4$ *stationary* lines interchangeably at distance $\Delta = 2^{2-1} = 2$ from each other. Thus, all of the *free* move a distance $\Delta = 3$ to form $n/4$ *vertical* lines of length 4 each. *DL-Doubling* repeats this process $\log n$ to eventually end up with the final straight line S_L of order n.

Observe that each *free* line goes through several turns in order to merge with the next following *stationary* line. In particular, each $free$ line performs the following: turn its orientation, push one move to connect perpendicularly with the *stationary* and finally turns orientation again where both form a double-length line. Lemma 1 states that during any phase i, a *free* line of length k_i, where $1 \leq k_i \leq n/2$, performs at most $4k_i = 4 \cdot 2^i$ moves to merge with closest *stationary* line. Thus, we give the following lemma.

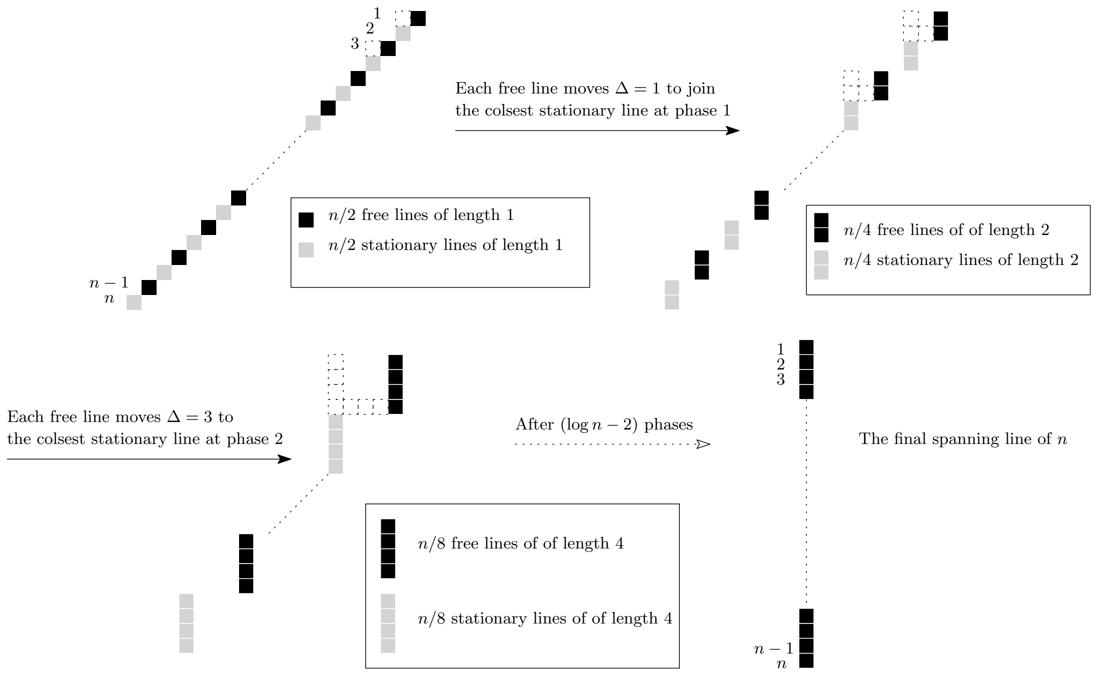

1
2
3
n − 1
n
Each free line moves Δ = 1 to join the colsest stationary line at phase 1
n/2 free lines of length 1
n/2 stationary lines of length 1
n/4 free lines of of length 2
n/4 stationary lines of length 2
Each free line moves Δ = 3 to the colsest stationary line at phase 2
After (log n − 2) phases
The final spanning line of n
n/8 free lines of of length 4
n/8 stationary lines of of length 4

Figure 3.2: The process of the $O(n \log n)$-time *DL-Doubling*. Nodes reside inside the black and grey cells.

Lemma 5. *By the end of phase i, DL-Doubling takes at most $O(n)$ moves to form $n/2^i$ lines of length 2^i each, where $1 \leq i \leq \log n$.*

Proof. Given that in phase i each free line of length 2^{i-1} pays at most $4 \cdot 2^{i-1} = 2^{i+1} - 3$ moves (applying Lemma 1 twice), then all of the $n/2^i$ lines pay a total cost t_i of at most:

$$t_i = \frac{n}{2^i} \cdot \left(2^{i+1} - 3\right) = 2n - \frac{3n}{2^i} = n\left(2 - \frac{3}{2^i}\right). \tag{3.5}$$

Thus, for each phase i, the bound of $O(n)$ moves holds trivially and inductively. □

Utilising Lemma 5, we can now formulate the following:

Theorem 3. *DL-Doubling solves the* DiagonalToLine *problem in $O(n \log n)$ moves.*

Proof. The goal line S_L of length n increases exponentially in each of the $\log n$ phases during the transformation. With that, the total running time T of this transformation is computed by summing the cost of 3.5 over all $\log n$ phases, as follows:

$$\begin{aligned} T &= \sum_{i=1}^{\log n} t_i \\ &= \sum_{i=1}^{\log n} n(2 - \frac{3}{2^i}) = 2n \log n - 3n \sum_{i=1}^{\log n} \frac{1}{2^i}. \end{aligned} \tag{3.6}$$

Let us compute the right summation of 3.6,

$$\sum_{i=1}^{\log n} \frac{1}{2^i} = \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^{\log n}} \right), \tag{3.7}$$

by multiplying 3.7 by 2 and subtracting it again by itself,

$$\begin{aligned} &\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^{\log(n-1)}}\right) - \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots + \frac{1}{2^{\log(n-1)}} + \frac{1}{2^{\log(n)}}\right) \\ &= 1 - \frac{1}{2^{\log(n)}}. \end{aligned} \tag{3.8}$$

Then, plug 3.8 into 3.6, this yields,

$$\begin{aligned} &2n \log n - 3n\left(1 - \frac{1}{2^{\log n}}\right) = 2n \log n - 3n + \frac{3n}{2^{\log n}} = n\left(2 \log n + \frac{3}{2^{\log n}} - 3\right) \\ &= O(n \log n). \end{aligned}$$

Thus, it has been proved that *DL-Doubling* transforms S_D of n nodes into S_L in a total of $O(n \log n)$ moves. □

3.1.3 An $O(n \log n)$-time Transformation Based on Recursion

An interesting observation for DiagonalToLine (i.e. without necessarily preserving connectivity), is that the problem is essentially self-reducible. This means that any transformation for the problem can be applied to smaller parts of the diagonal, resulting in small lines, and then trying to merge those lines into a single straight line. An immediate question is then whether such recursive transformations can improve upon the $O(n \log n)$ best upper bound established so far. The extreme application of this idea is to employ a full uniform recursion (call it *DL-Recursion*), where S_D is first partitioned into two diagonals of length

$n/2$ each, and each of them is being transformed into a line of length $n/2$, by recursively applying to them the same halving procedure. Finally, the top-right half has to pay in a total of at most $4(n/2) = 2n$ to merge with the bottom-left half and form a single straight line (and the same is being recursively performed by smaller lines).

More formally, consider a diagonal S_D of n nodes where the bottom-left and top right nodes occupy (x_1, y_1) and (x_n, y_n), respectively. Then, the goal is to collect all nodes at the leftmost column, say x_n. The collection can be arranged in a recursive way by creating stop points (partitions) on S_D in which each stop point always creates equal partitions of the same length. This can be parametrised by $\frac{n}{x}$ for each partition, where $2 \leq x \leq n$. For example, if $x = 2$, we have a stop point that halves S_D into 2 partitions of length $\lceil \frac{n}{2} \rceil$. As a consequence, the first node on the top will stop at the middle of S_D and wait for all nodes to its right to gather at that point (*column*) and then continue directly to the gathering (*column*) x_n.

Now, let us repeat the same precess on each of the x partitions recursively, by considering the partition as a diagonal of length roughly $\frac{n}{x} - 1$, which is divided into x sub-partitions each of length $\frac{n}{x^2}$ roughly. Every recursion shrinks the partitions by a factor of $\frac{1}{x}$. For example, in the $x = 2$ case, we halve the length of the partitions every time we subdivide, therefore, we will end the recursion when it arrives at partitions of length 1, which will happen after $\log n$ repetitions. That occurs similarly for the general x case by simply end after $\log_x n$ repetitions. For example, Figure 3.3 demonstrates this procedure.

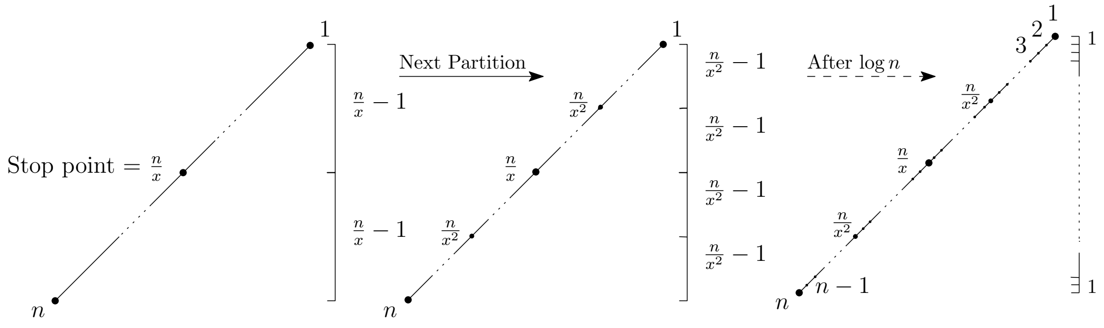


Figure 3.3: Shows all steps of subdividing the diagonal shape recursively by a factor of $\frac{1}{x}$, where $x = 2$.

Next, we draw an abstract underlying tree of the partitioning process to trace all necessary computations required to travel from the diagonal S_D into a bottommost left column x_n. Figure 3.4 presents the tree of n nodes and weighted edges indicate the minimum distances (shortest paths) Δ between them. The abstract tree has a degree and depth of $\log_2 n$, which is also the number of phases that are needed to chase the segmentation recursively. Here, node n is the root of the tree occupies the target column at x_n, and it has $\log n$ child nodes $n-1, n-2, n-4, \ldots, \frac{n}{2}$ of distances $\Delta = 1, 2, 4, \ldots, \frac{n}{2}$, respectively. Now, the distance Δ between a parent u and child node v defines two basic properties: 1) the number of sub-child nodes (siblings) of v, namely each child node v gets $\log \Delta(u, v)$ sub-child nodes, and 2) the maximum cost by which a child node v requires to merge with its parent u, and it is computed by $(2^{\Delta+1} - 3)$ (a reader may consult Lemma 5 and Theorem 3 concerning the maximum cost). For example, the $\frac{n}{2}$ child node needs $(2^{\frac{n}{2}+1} - 3)$ steps to join its parent node n in a distance of $\Delta = n - \frac{n}{2} = \frac{n}{2}$, and at the same time, this tells us that this child node is also holding $\log \frac{n}{2}$ sub-child nodes. On the contrary, $n - 1$ node got no sub-child since $\log 1 = 0$, so it is a leaf that requires $(2^{1+1} - 3) = 1$ steps to reach its parent node n. The same idea follows for other sub-trees, such as $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, 2$.

Having said that, the implementation of any transformation strategy that solves the above recursion problem can generally be reordered without affecting the cost, and each stop point takes place in the reordered version. That is because the abstract tree representation remains conveniently invariant (i.e. inherits the same cost) for any arrangement by which a transformation can exploit to collect nodes into the target point (*column* or *row*). However, this recursion proceeds in $\log n$ phases of *cost* , where in each phase i, for all $1 \leq i \leq \log n$, we upper bound a cost that the transformation A pays at most to move all nodes in each phase (even though A may move them in an entirely different order). In phase 1, A works on nodes $1, 3, 5, \ldots, n-1$ to eventually align all of them; hence, no matter in what order the nodes move, A performs at most $O(\frac{n}{2})$ steps during the first phase.

In the second phase, (*columns* or *rows*) of nodes $2, 4, 6, \ldots, (n-2)$ are occupied by 2 nodes. Those pairs of nodes principally move at some point (even together or as part of other nodes going through them) to its parent nodes (i.e. the next occupied column or row), with paying a cost of $(2^{2+1} - 3) = 5$ each. Repeat the same argument for the rest of the phases, the upper bound now is based on the following observation: whenever k nodes, where $1 \leq k < n$, all move as maximum as $(2^{k+1} - 3)$ steps to merge with the next stop point (parent node). In other words, suppose that k nodes are a line of k nodes, it walks at most $(2^{k+1} - 3)$ steps to form a line with the next occupied row or column.

Generally, we can say that at any phase i of any transformation A solves the recursion problem, where $0 \leq i \leq \log n$, there is a node v with other 2^i nodes occupying the same *column* (*row*), where the distance between v and its parent u (the next occupied *column*

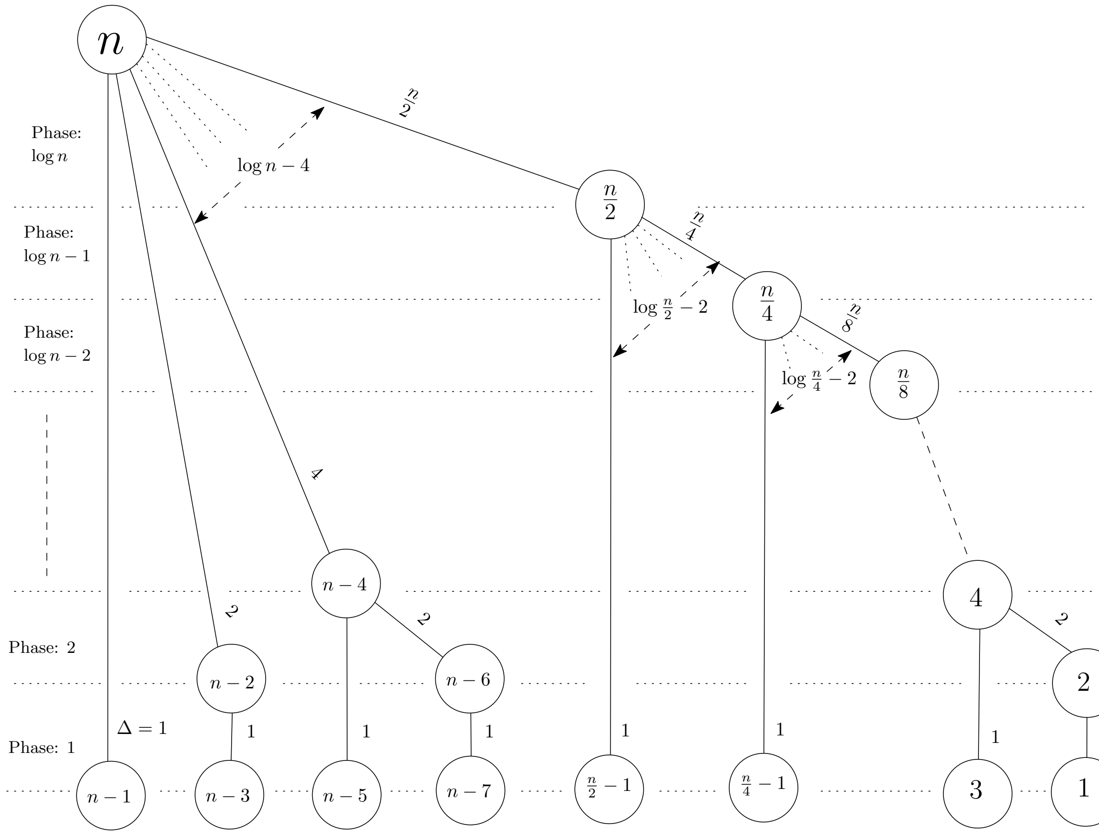


Figure 3.4: The underlying tree representation of a recursive partitioning of a diagonal. Edges are weighted by the minimum distance (Δ) between nodes. See the text for more explanation.

(*row*) is $\Delta(u, v) = 2^i$. Then, our purpose is to upper bound A's cost for v to reach and merge with u; therefore, v walks at most $(2^{i+1} - 3)$ steps to form a line with u.

By analysing the running time of such a uniform recursion, we obtain that it is still $O(n \log n)$, partially suggesting that recursive transformations might not be enough to improve upon $O(n \log n)$ (also possibly because of an $\Omega(n \log n)$ matching lower bound, which is left as an open question). If we denote by T_k the total time needed to split and merge lines of length k, then the recursion starts from 1 line incurring T_n and ends up with n lines incurring T_1. In particular, we analyse the recurrence relation:

$$T_n = 2 \cdot T_{n/2} + 2n = 2(2 \cdot T_{n/4} + n) + 2n = 4 \cdot T_{n/4} + 4n = 4(2 \cdot T_{n/8} + n/2) + 4n$$

$$= 8T_{n/8} + 6n = \cdots = 2^i \cdot T_{n/2^i} + 2i \cdot n = \cdots = 2^{\log n} \cdot T_{n/2^{\log n}} + 2(\log n)n.$$

Since $T_1 = 1$, we get,

$$\begin{aligned} T_n &= n \cdot T_1 + 2n(\log n) = n + 2n(\log n) \\ &= O(n \log n). \end{aligned}$$

To prove this, let us rewrite the recurrence as,

$$T_n = aT_{n/b} + \Theta(n^d), \tag{3.9}$$

for constants $a \geq 1, b > 1$ and $d \geq 1$. Then by the master theorem [41], T_n satisfies the following three cases:

$$T_n = \begin{cases} \text{case 1: } \Theta(n^d), & \text{if } d > \log_b a; \\ \text{case 2: } \Theta(n^d \log n), & \text{if } d = \log_b a; \\ \text{case 3: } \Theta(n^{\log_b a}), & \text{if } d < \log_b a. \end{cases}$$

Recall that the recurrence runtime is,

$$T_n = 2T_{n/2} + 2n.$$

Hence, we have $a = 2$ and $b = 2$ and thus $\log_b a = \log_2 2 = 1$. Since $d = \log_b a = 1$, we can apply case 2 of the master method and conclude that $T_n = O(n \log n)$. Now, let us drop the assumption (n is a power of two) and prove that $T_n = O(n \log n)$ for all n. In the general case, the running time gets the following recurrence:

$$T_n = T_{\lceil n/2 \rceil} + T_{\lfloor n/2 \rfloor} + \Theta(n).$$

By replacing $aT_{n/b}$ in 3.9 by $a_1 T_{\lceil n/b \rceil} + a_2 T_{\lfloor n/b \rfloor}$, where $a_1, a_2 \geq 0$ and $a_1 + a_2 = a$, then the master theorem still holds, and hence the bound $T_n = O(n \log n)$ remains true for all n. Finally, we give the following theorem,

Theorem 4. *DL-Recursion transforms any diagonal S_D of order n into a line S_L in $O(n \log n)$ moves.*

3.2 Universal Transformations

In this section, we develop universal transformations that transform *any* initial connected shape S_I into *any* target shape S_F of the same order n exploiting line moves in which the connectivity of the shape can be broken during the transformation. Due to reversibility (Lemma 2), it is sufficient to show that any initial connected shape S_I can be transformed into a straight line (implying then that any pair of shapes can be transformed to each other via the line and by reversing one of the two transformations).

3.2.1 U-Box-Partitioning: An $O(n\sqrt{n})$-time Transformation

We present a universal transformation, called *U-Box-Partitioning*, that solves UNIVERSALTRANSFORMATION in $O(n\sqrt{n})$ moves. Observe that any initial connected shape S_I can be enclosed in an appropriately positioned $n \times n$ square (called a *box*). Our universal transformation is divided into three phases:

Phase A: Partition the $n \times n$ box into $\sqrt{n} \times \sqrt{n}$ sub-boxes (n in total in order to cover the whole $n \times n$ box). In each sub-box move all nodes in it down towards the bottommost row of that sub-box as follows. Start filling in the bottommost row from left to right, then if there is no more space continue to the next row from left to right and so on until all nodes in the sub-box have been exhausted (resulting in zero or more complete rows and at most one incomplete row). Moving down is done via shortest paths (where in the worst case a node has to move distance $2\sqrt{n}$). This brute-force line formation is illustrated in Figure 3.5.

Phase B: Choose one of the four length-n boundaries of the $n \times n$ box, say without loss of generality the left boundary. This is where the straight line will be formed. Then, transfer every line via a shortest path to that boundary (incurring a maximum distance of $n - \sqrt{n}$ per line).

Phase C: Turn all lines (possibly consisting of more than one line on top of each other), by a procedure similar to that of Figure 3.1(e), to end up with a straight line of n nodes on the left boundary.

Now we provide a more formal description of *U-Box-Partitioning.* First, there are two variants of $\sqrt{n} \times \sqrt{n}$ sub-boxes:

1. *Occupied sub-box*: Denoted by s and contains k nodes of S_I, where $1 \leq k \leq n$.
2. *Unoccupied sub-box*: An empty sub-box (has no nodes).

Given that, we show some basic properties of *occupied sub-boxes*. Given an *occupied sub-box* s of k nodes, where $1 \leq k \leq n$, then the maximum number of lines which can be formed inside s is at most $\lceil \frac{k}{\sqrt{n}} \rceil$. As mentioned above, those k lines can be aligned horizontally at bottommost rows or vertically at leftmost columns of the occupied sub-box by a brute-force line formation in Figure 3.5. Hence, the following lemma gives an upper bound on the number of sub-boxes that any initial connected shapeS_I may occupy.

Lemma 6. *A connected shape S_I of order n occupies $O(\sqrt{n})$ sub-boxes.*

Proof. By Corollary 1, any S_I can occupy at most $O(\frac{n}{d})$ sub-boxes of dimension d. In this case, *U-Box-Partitioning* divides the $n \times n$ square box into $n \times n$ sub-boxes of size $d = \sqrt{n}$, with S_I occupying no more than $\frac{n}{\sqrt{n}} = O(\sqrt{n})$ sub-boxes. □

Below, we prove the correctness and analyse the running time of phase A.

Lemma 7. *Starting from any connected shape S_I of order n, Phase A completes within $O(n\sqrt{n})$ moves.*

Proof. By Lemma 6, let S_I be a connected shape of n nodes occupies $\sqrt{n}$ sub-boxes of size $\sqrt{n} \times \sqrt{n}$ each, and $s \in S_I$ be any *occupied sub-box* of k nodes, where $1 \leq k \leq n$. Then, s performs a trivial line formation to collect all k nodes at its bottommost (*or leftmost*) boundary. Consider the worst case of s having a node that occupies the top-right corner and wants to gather at the bottom-left. It needs to move a distance of at most $\Delta = 2\sqrt{n}$ via shortest path to arrive at the bottom-left corner. Observe that s forms at least one *complete* line of length $\sqrt{n}$ or one *incomplete* of less than $\sqrt{n}$. Since S_I is connected, there must be at most $\sqrt{n}$ *occupied sub-boxes*, implying that a total of at most $\frac{n}{\sqrt{n}} = O(\sqrt{n})$ lines (*complete or incomplete*) living inside all of those *occupied sub-boxes*. With that, a line l of w nodes, for all $1 \leq l, w \leq \sqrt{n}$, is formed within a total moves t that given by:

$$\begin{aligned} t &= w \cdot 2\sqrt{n} = \sqrt{n} \cdot 2\sqrt{n} = 2n \\ &= O(n). \end{aligned} \tag{3.10}$$

Finally, all of the $\sqrt{n}$ lines ask for a total cost T_1 of moves to complete the brute-force line formation inside the *occupied sub-boxes*, which is computed by multiplying 3.10 by $\sqrt{n}$, as follows:

$$T_1 = \sqrt{n} \cdot t = \sqrt{n} \cdot 2n = 2n\sqrt{n}$$
$$= O(n\sqrt{n}). \tag{3.11}$$

□

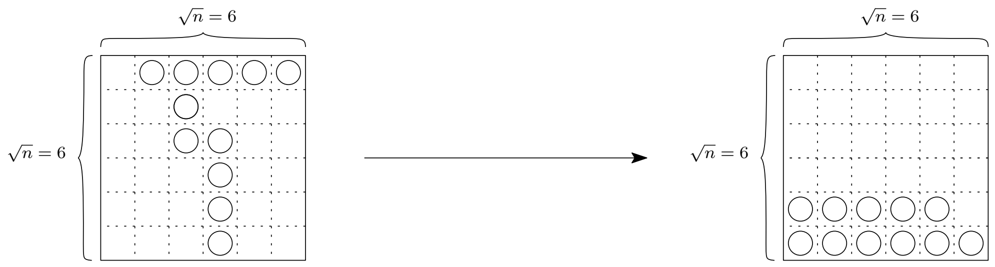


Figure 3.5: An example of a brute-force line formation to collect all k nodes at bottommost rows of a sub-box of size 6×6 containing $k = 11$ nodes.

In phase B, set any (length-n) boundary of the $n \times n$ square box as the *gathering boundary* of all lines that are formed in phase A. Then, the following lemma computes the total moves of the gathering.

Lemma 8. *Starting from any connected shape S_I of order n, Phase B completes in $O(n\sqrt{n})$ moves.*

Proof. It follows from Lemmas 6 and 7. Let S_I be a connected shape of order n enclosed by a $n \times n$ box and then partitioned into $\sqrt{n} \times \sqrt{n}$ *occupied sub-boxes* of k nodes each, where $1 \leq k \leq n$. By phase A, there are l lines, for all $1 \leq l \leq \sqrt{n}$, formed inside all $\sqrt{n}$ *occupied sub-boxes*. Without loss of generality, define the left border of the $n \times n$ box as the gathering boundary for all those lines. The distance between any line inside an *occupied sub-box* and the defined boundary is no longer than $n - \sqrt{n}$. Thus, each line moves a distance δ by at most $\delta \leq n - \sqrt{n}$ to arrive at the gathering boundary. Then, all lines l reach the left border with a total cost T_2 of at most:

$$T_2 = l \cdot \delta$$
$$= \sqrt{n} \cdot (n - \sqrt{n}) = n\sqrt{n} - n$$
$$= O(n\sqrt{n}). \tag{3.12}$$

□

By the end of phase B, all $\sqrt{n}$ lines have transferred and arrived at the length-n gathering boundary, where each contributes a node to that boundary. By Lemma 9, in phase C, all of those lines fill in the length-n border in order to form the goal straight line. Formally, we give the following Lemma.

Lemma 9. *Consider any length-n boundary and n nodes forming k lines, where $1 \leq k \leq n$, that are perpendicular to that boundary. Then, by line moves, the k lines require at most $O(n)$ moves to form a line of length n on that boundary. This implies that Phase C is completed in $O(n)$ moves.*

Proof. In Figure 3.6, the lines $\{l_1, l_2, \dots, l_k\}$ of n nodes are connected perpendicularly the dashed line which denotes the length-n gathering border, where $1 \leq k \leq \sqrt{n}$. The n nodes are sufficient to completely fill up the border of length $\Delta = n$. Now, pick the first line l_1 of k_1 nodes and start to push k_1 into the topmost point of the boundary, until either 1) k_1 are exhausted, or 2) reaching the topmost point of the boundary and still have nodes waiting to be pushed. In this case, l_1 can easily begin to push the remaining of k_1 downwards at the boundary. Repeat the same strategy for all other lines shall fill in the length-n border completely by n nodes.

Observe that each of the k lines contributes one node to length-n boundary and there is still $n - k$ nodes waiting to be pushed at that boundary. By an application of Lemma 1, each of the $n - k$ nodes requires 2 moves to be included to the border. This means that all $n - k$ nodes must conduct a total of $2(n - k)$ moves in order to completely fill up the border of length n. Thus, for all k lines of n nodes connected perpendicularly to a length-n border, *U-Box-Partitioning* pushes the k lines of n nodes into the length-n border in a total T_3 of at most,

$$\begin{aligned} T_3 &= 2(n-k) = 2(n-\sqrt{n}) \\ &= O(n). \end{aligned} \tag{3.13}$$

□

By Lemmas 6, 7 and 9, the following lemma state that any connected shape S_I can transform into a line S_L in sub-quadratic moves.

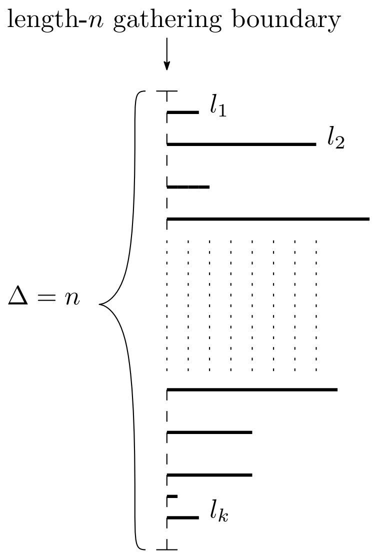


Figure 3.6: The dashed line indicates the length-n gathering boundary of the $n \times n$ box, whilst the bold black lines represent the k lines of n nodes.

Lemma 10. *U-Box-Partitioning transforms any connected shape S_I into a straight line S_L of the same order n, in $O(n\sqrt{n})$ moves.*

Proof. The sum of 3.11, 3.12 and 3.13 computes the overall moves T as follow:

$$\begin{aligned} T &= T_1 + T_2 + T_3 \\ &= O(n\sqrt{n}) + O(n\sqrt{n}) + O(n) \\ &= O(n\sqrt{n}), \end{aligned} \tag{3.14}$$

which finally provides an upper bound $O(n\sqrt{n})$ of *U-Box-Partitioning* to transform any arbitrary connected shape S_I into a single straight line S_L of the same number of nodes. □

In conclusion, putting Lemma 10 and reversibility (Lemma 2) together implies that, for any pair of connected shapes S_I and S_F of the same order, *U-Box-Partitioning* can be used to transform S_I into S_F (and S_F into S_I) in $O(n\sqrt{n})$ moves. Then, we formally state that:

Theorem 5. U-Box-Partitioning *solves* UNIVERSALTRANSFORMATION *in $O(n\sqrt{n})$ moves.*

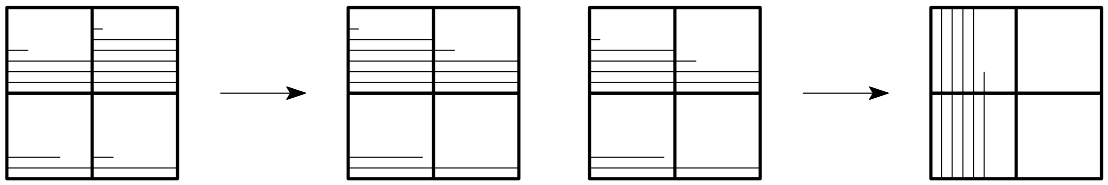
3.2.2 U-Box-Doubling: An $O(n \log n)$-time Transformation

We now introduce *U-Box-Doubling*, an alternative universal transformation that solves UniversalTransformation more efficiently in $O(n \log n)$ moves, compared to the previous one of $O(n\sqrt{n})$. This transformation has benefited from the strategies outlined in Sections 3.1.2 and 3.2.1. Refer to the dependency digram in Figure 1.5 for the development sequence of the transformations.

Given a connected shape S_I of order n, do the following. Enclose S_I into an arbitrary $n \times n$ square box that is completely containing S_I (this is always possible). For simplicity, we assume that n is a power of 2, but this assumption can be dropped. Proceed in $\log n$ phases as follows: In every phase i, where $1 \leq i \leq \log n$, partition the $n \times n$ box into $2^i \times 2^i$ sub-boxes, disjoint and completely covering the $n \times n$ box. Assume that from any phase $i-1$, any $2^{i-1} \times 2^{i-1}$ sub-box is either empty or has its k, where $0 \leq k \leq 2^{i-1}$, bottommost rows completely filled in with nodes, possibly followed by a single incomplete row on top of them containing l, where $1 \leq l < 2^{i-1}$, consecutive nodes that are left aligned on that row. This case holds trivially for phase 1 and inductively for every phase. That is, in odd phases, we assume that nodes fill in the leftmost columns of boxes in a symmetric way. Every $2^i \times 2^i$ sub-box (of phase i) consists of four $2^{i-1} \times 2^{i-1}$ sub-boxes from phase $i-1$, each of which is either empty or occupied as described above.

The operation of *Boundary-Filling* is to fill in empty cells at a boundary of the $2^i \times 2^i$ sub-box by nodes of lines that are aligned perpendicularly to that boundary. Due to symmetry, we only show the left boundary case. That is, start filling in empty cells from the leftmost column bottom-top and continuing to the right, by exploiting a linear procedure similar to that of Figure 2.6 (and of *nice* shapes). Without loss of generality, fill in the leftmost column until the row is exhausted or the column is being completely occupied, in this case, start filling in the next column to the right (see Figure 3.7(b)). If an incomplete column remains in the top left $2^{i-1} \times 2^{i-1}$ sub-box, push the nodes in it to the bottom of that column.

Consider the case where i is odd, thus, the nodes in the $2^{i-1} \times 2^{i-1}$ sub-boxes are bottom aligned. For every $2^{i-1} \times 2^{i-1}$ sub-box, move each line from the previous phase that resides in the sub-box to the left as many moves as required until that row contains a single line of consecutive nodes, starting from the left boundary of the sub-box, as shown in Figure 3.7(a). Then, perform *Boundary-Filling* on the left boundary the $2^i \times 2^i$ sub-box, as in Figure 3.7(b). The case of even i is symmetric, the only difference being that



(a) Pushing left in each $2^{i-1} \times 2^{i-1}$ sub-box of the $2^i \times 2^i$ sub-box.

(b) The operation of *Boundary-Filling* on the left boarder of the $2^i \times 2^i$ sub-box.

Figure 3.7: An example of the transformations during phase i.

the arrangement guarantee from $i-1$ is left alignment on the columns of the $2^{i-1} \times 2^{i-1}$ sub-boxes and the result will be bottom alignment on the rows of the $2^i \times 2^i$ sub-boxes of the current phase. This completes the description of the transformation. We first prove *correctness*:

Lemma 11. *Starting from any connected shape S_I of order n,* U-Box-Doubling *forms by the end of phase* $\log n$ *a line of length n.*

Proof. In phase $\log n$, the procedure partitions into a single box, which is the whole original $n \times n$ box. Independently of whether gathering will be on the leftmost column or on the bottommost row of the box, as all n nodes are contained in it, the outcome will be a single line of length n, vertical or horizontal, respectively. □

Lemma 12. *In every phase i, the 'super-shape' formed by the occupied $2^i \times 2^i$ sub-boxes is connected.*

Proof. By induction on the phase number i. For the base of the induction, observe that for $i = 0$ it holds trivially because the initial S_I is a connected shape. Assuming that it holds for phase $i-1$, we shall now prove that it must also hold for phase i. By the inductive assumption, the occupied $2^{i-1} \times 2^{i-1}$ sub-boxes form a connected super-shape. Observe that, by the way the original $n \times n$ box is being repetitively partitioned, any box contains complete sub-boxes from previous phases, that is, no sub-box is ever split into more than one box of future phases. Additionally, observe that a sub-box is occupied iff any of its own sub-boxes (of any size) had ever been occupied, because nodes cannot be transferred between $2^i \times 2^i$ sub-boxes before phase $i+1$. Assume now, for the sake of contradiction, that the super-shape formed by $2^i \times 2^i$ sub-boxes is disconnected. This means that there exists a "cut" of unoccupied $2^i \times 2^i$ sub-boxes as in Figure 3.8.

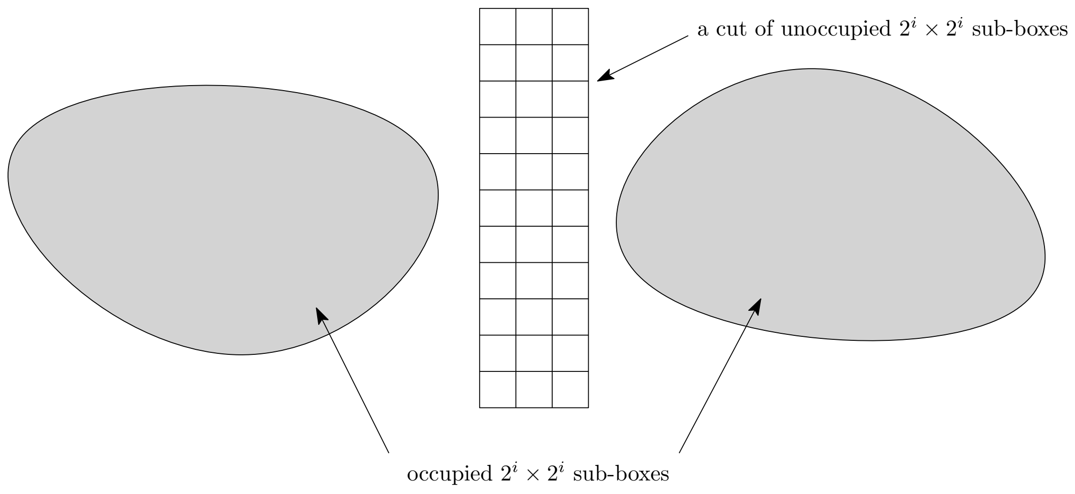

a cut of unoccupied $2^i \times 2^i$ sub-boxes
occupied $2^i \times 2^i$ sub-boxes

Figure 3.8: An example of a 'cut' of unoccupied $2^i \times 2^i$ sub-boxes.

Replacing everything by $2^{i-1} \times 2^{i-1}$ sub-boxes, yields that this must also be a cut of $2^{i-1} \times 2^{i-1}$ sub-boxes, because a node cannot have transferred between $2^i \times 2^i$ sub-boxes before phase $i+1$. But this contradicts the assumption that $2^{i-1} \times 2^{i-1}$ sub-boxes form a connected super-shape. Therefore, it must hold that the $2^i \times 2^i$ sub-boxes super-shape must have been connected. □

Next, we give an upper bound on the number of occupied sub-boxes in a phase i.

Lemma 13. *Given that U-Box-Doubling starts from a connected shape S_I of order n, the number of occupied sub-boxes in any phase i is $O(\frac{n}{2^i})$.*

Proof. First, observe that a $2^i \times 2^i$ sub-box of phase i is occupied in that phase iff S_I was originally going through that sub-box. This follows from the fact that nodes are not transferred by this transformation between $2^i \times 2^i$ sub-boxes before phase $i+1$. Therefore, the $2^i \times 2^i$ sub-boxes occupied in (any) phase i are exactly the $2^i \times 2^i$ sub-boxes that the original shape S_I would have occupied, thus, it is sufficient to upper bound the number of $2^i \times 2^i$ sub-boxes that a connected shape of order n can occupy. Or equivalently, we shall lower bound the number N_k of nodes needed to occupy k sub-boxes.

In order to simplify the argument, whenever S_I occupies another unoccupied sub-box, we will award it a constant number of additional occupations for free and only calculate the additional distance (in nodes) that the shape has to cover in order to reach another unoccupied sub-box. In particular, pick any node of S_I and consider as freely occupied that sub-box and the 8 sub-boxes surrounding it, as depicted in Figure 3.9(a). Giving sub-boxes for free can only help the shape, therefore, any lower bound established including the free sub-boxes will also hold for shapes that do not have them (thus, for the original problem).

Given that free boxes are surrounding the current node, in order for S_I to occupy another sub-box, at least one surrounding $2^i \times 2^i$ sub-box must be exited. This requires covering a distance of at least 2^i, through a connected path of nodes. Once this happens, S_I has just crossed the boundary between an occupied sub-box and an unoccupied sub-box. Subsequently, it has just crossed the boundary between an occupied sub-box and an unoccupied sub-box. Then, by giving it for free at most 5 more unoccupied sub-boxes, S_I has to pay another 2^i nodes to occupy another unoccupied sub-box; see Figure 3.9(b). We then continue applying this 5-for-free strategy until all n nodes have been used.

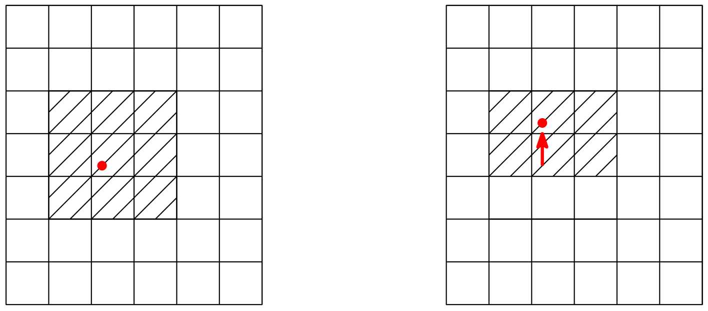


Figure 3.9: (a) A node of shape S_I in red and the occupied sub-boxes that we give for free to the shape. (b) The node just exited the sub-box with arrow entering an unoccupied sub-box. By giving the 5 horizontally dashed sub-boxes for free, a distance of at least 2^i has to be travelled in order to reach another unoccupied sub-box.

To sum up, the shape has been given 8 sub-boxes for free, and then for every sub-box covered it has to pay 2^i and gets 5 sub-boxes. Thus, to occupy $k = 8 + l \cdot 5$ sub-boxes, at least $l \cdot 2^i$ nodes are needed, that is,

$$N_k \geq l \cdot 2^i. \tag{3.15}$$

But, that leads to

$$k = 8 + l \cdot 5 \Rightarrow l = \frac{k-8}{5}. \tag{3.16}$$

Thus, from (3.15) and (3.16):

$$N_k \geq \frac{k-8}{5} \cdot 2^i. \tag{3.17}$$

But shape S_I has order n, which means that the number of nodes available is upper bounded by n, i.e. $N_k \leq n$, which gives:

$$\frac{k-8}{5} \cdot 2^i \leq N_k \leq n \Rightarrow$$
$$\frac{k-8}{5} \cdot 2^i \leq n \Rightarrow \frac{k-8}{5} \leq \frac{n}{2^i} \Rightarrow$$
$$k \leq 5\left(\frac{n}{2^i}\right) + 8.$$

We conclude that the number of $2^i \times 2^i$ sub-boxes that can be occupied by a connected shape S_I, and, thus, also the number of $2^i \times 2^i$ sub-boxes that are occupied by *U-Box-Doubling* in phase i, is at most $5(n/2^i) + 8 = O(n/2^i)$. □

As a corollary of this, we obtain:

Corollary 1. *Given a uniform partitioning of $n \times n$ square box containing a connected shape S_I of order n into $d \times d$ sub-boxes, it holds that S_I can occupy at most $O(\frac{n}{d})$ sub-boxes.*

We are now ready to analyse the running time of *U-Box-Doubling*.

Lemma 14. *Starting from any connected shape of n nodes, U-Box-Doubling performs $O(n \log n)$ moves during its course.*

Proof. We prove this by showing that in every phase i, $1 \leq i \leq \log n$, the transformation performs at most a linear number of moves. We partition the occupied $2^i \times 2^i$ sub-boxes into two disjoint sets, B_1 and B_0, where sub-boxes in B_1 have at least 1 *complete line* (from the previous phase), i.e. a line of length 2^{i-1}, and sub-boxes in B_0 have 1 to 4 *incomplete lines*, i.e. lines of length between 1 and $2^{i-1} - 1$. For B_1, we have that $|B_1| \leq n/2^{i-1}$. In each phase, we may have horizontal or vertical lines that need to be aligned to the left or bottom boundary of their $2^i \times 2^i$ sub-box, respectively, depending on the parity of i. As the two cases are symmetric, without loss of generality we only show horizontal lines which are moving to their left. Hence, for every complete line, we pay at most 2^{i-1} to transfer it left. As there are at most $n/2^{i-1}$ such complete lines in phase i, the total cost for this is at most $2^i \cdot (n/2^{i-1}) = n$.

Each sub-box in B_1 may also have at most 4 incomplete lines from the previous phase, as in Figure 3.7 left, where at most two of them may have to pay a maximum of 2^{i-1} to be transferred left (as the other two are already aligned). As there are at most $n/2^{i-1}$ sub-boxes in B_1, the total cost for this is at most $2 \cdot 2^{i-1} \cdot (n/2^{i-1}) = 2n$. Therefore, the total cost for pushing all lines towards the required border in B_1 sub-boxes is at most:

$$n + 2n = 3n. \tag{3.18}$$

For B_0, we have (by Lemma 13) that the total number of occupied sub-boxes in phase i is at most $5(n/2^i) + 8$, therefore, $|B_0| \leq 5(n/2^i) + 8$ (taking into account also the worst case where every occupied sub-box may be of type B_0). There is again a maximum of 2 incomplete lines per such sub-box that need to be transferred a distance of at most 2^{i-1}, therefore, the total cost for this to happen in every B_0 sub-box is at most:

$$2 \cdot 2^{i-1}\left(5 \cdot \frac{n}{2^i} + 8\right) = 5n + 8 \cdot 2^i \leq 13n. \tag{3.19}$$

By paying the above costs, all occupied sub-boxes have their lines aligned to the left, and the final task of the transformation for this phase is to apply a linear procedure in order to fill in the left boundary of the $2^i \times 2^i$ sub-box. This procedure costs at most $2k$ for every k nodes aligned as above, in a total of at most $(2n - 2)$ steps (see Lemma 1). As there is an additional cost of 2^{i-1} for an incomplete line to be transferred into the bottom left, as shown in Figure 3.6, the total cost for this phase is at most:

$$2n - 2 + (2^{i-1}) \leq 3n. \tag{3.20}$$

This completes the operation of *U-Box-Doubling* for phase i. Putting (3.18), (3.19), and (3.20) together, we obtain that the total cost T_i, in moves, for phase i is,

$$T_i \leq 3n + 13n + 3n = 19n.$$

As there is a total of $\log n$ phases, we conclude that the total cost T of the transformation is,

$$\begin{aligned} T &\leq 19n \cdot \log n \\ &= O(n \log n). \end{aligned}$$

□

Finally, together Lemma 11, Lemma 14, and reversibility (Lemma 2) imply that; for any pair of connected shapes S_I and S_F of the same order n, transformation *U-Box-Doubling* can be used to transform S_I into S_F (and S_F into S_I) in $O(n \log n)$ moves. Then, we formally state:

Theorem 6. *U-Box-Doubling solves* UNIVERSALTRANSFORMATION *in $O(n \log n)$ moves.*

Chapter 4

Connectivity-Preserving transformations

The *connectivity-preservation* condition is crucial in many robotic systems, see for example [42]. It is a desired feature for any resilient transformation that aims to improve the reliability of communication in different types of networks. Furthermore, maintaining connectivity among all robots is vital for practical applications that regularly require energy for data transmission and the execution of various locomotion mechanisms. Although this is often a desirable condition in distributed systems, we study it first in the context of global knowledge within our centralised framework. This is because distributed are model-dependent (in terms of, for example, knowledge and communication), whereas centralised systems demonstrate what is theoretically feasible and probably help to come up with different techniques that may be applicable to the distributed algorithms. The majority of the connectivity solutions presented in this chapter aided in the design of distributed transformations later in Chapter 5.

This chapter investigates the case where the transformations are more constrained and must preserve the shape's connectivity during its execution. To this end, we present solutions that provably ensure that the corresponding graphs of all intermediate configurations are connected throughout the transformations.

We start our investigation with the potentially hard-case of transforming the diagonal line shape into a straight line in Section 4.1. First, we solve this problem by presenting an $O(n\sqrt{n})$-time transformation called *DLC-Folding* in Section 4.1.1. This approach divides the diagonal into $\sqrt{n}$ segments of length $\sqrt{n}$ each and proceeds in $\sqrt{n}$ phases. In each phase, it folds the segments sequentially in a top-down order via two main operations, *turn* and *push*. In the final phase, the algorithm forms *a square shape*, which can be transformed fast into a line. Thus, this transformation (*folding*) preserves connectivity and takes total time $O(n\sqrt{n})$. Next, in Section 4.1.2, we provide another $O(n\sqrt{n})$-time transformation called *DLC-Extending*, which also maintains connectivity during the transformation using a different technique. This approach begins at a topmost $\sqrt{n}$-length segment of the diagonal, then extends and doubles its size through an implementation of two primitives, *turn* and *push* until arriving at the target straight line of length n.

Going a step further in Section 4.2, we manage to reduce the running time and presented very fast connectivity-preserving transformations for the case in which the associated graphs of the initial and goal shapes contains to a Hamiltonian line. Our transformations, in particular, perform $O(n \log n)$ moves, which is asymptotically equivalent to the best known running time of connectivity-breaking transformations shown in the previous chapter. It is called *Walk-Through-Path* and proceeds in $\log n$ phases. In each phase i, this procedure transforms a terminal 2^i nodes on the Hamiltonian path into a straight line whose length is doubled as follows. First, a respective line of length 2^{i-1} from the previous phase $i-1$ is identified. Afterwards, a feasible transformation path is computed via a specific operation called *LineWalk*. Once having that transformation path, the respective line (of length 2^{i-1}) pushes to its destination. This line is then merged recursively with the remaining nodes in that phase to form a new line of length 2^i.

The last section in this chapter, Section 4.3, demonstrates our most general algorithm which is a connectivity-preserving universal transformation that can transform any pair of connected shapes of the same order to each other within subquadratic time, namely, through a sequence of at most $O(n\sqrt{n})$ moves. This algorithm encloses the initial shape, which is represented by a spanning tree T, into a square box of size n and spilites it into sub-boxes of size $\sqrt{n}$, each of which contains at least one sub-tree of T. It then compresses the nodes in a sub-box into a neighbouring sub-box towards a parent sub-tree by shifting lines in a way that does not break connectivity. A final compressed square configuration is achieved by carefully repeating this technique. The latter is a type of *Central Line shapes* (or *nice shapes*), defined in Section 2.1.1, which can be fast transformed into a straight line in linear time. By the reversibility of our model, we then analyse this strategy based on the number of charging phases, which turns out to be $\sqrt{n}$ each with at most n moves, for a total of $O(n\sqrt{n})$ moves.

4.1 The Diagonal-To-Line transformation

In this section, we present two shape transformations that can reconfigure the diagonal connected shape S_D into a straight line S_L *while preserving connectivity* during transformations. Recall that we focus on this apparently hard instance due to several special properties that cannot be found in other connected shapes. For example, each single node of S_D occupies a unique x- and y- axis, forming an initial shape of n lines of length 1. This gives the $O(n^2)$ individual node distance between S_D and S_L. Below, we provide two $O(n\sqrt{n})$-time transformations, *DLC-Folding* and *DLC-Extending*, that preserve connectivity of the shape throughout the transformation.

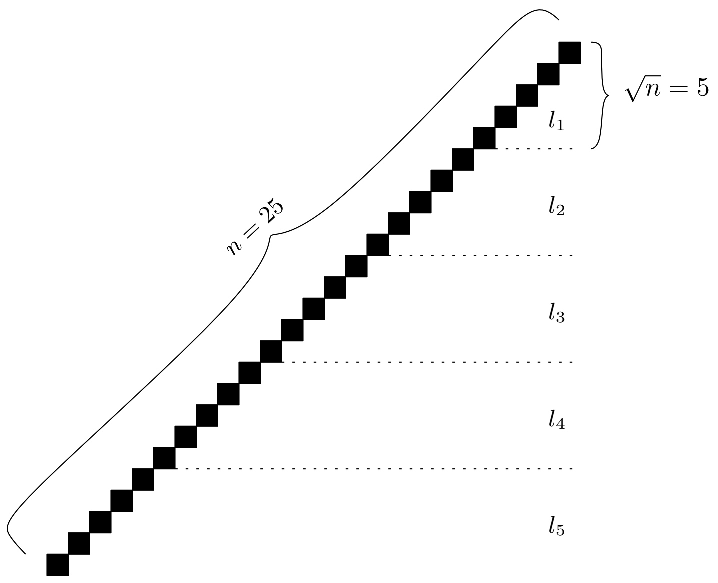


Figure 4.1: A diagonal line of 25 nodes.

4.1.1 Folding: An $O(n\sqrt{n})$-time Transformation

Consider an S_D of n nodes occupying $(x, y), (x + 1, y + 1), \ldots, (x + n - 1, y + n - 1)$, such as the diagonal line of 25 nodes depicted in Figure 4.1. Below, we give an $O(n\sqrt{n})$-time transformation to transform S_D into S_L by exploiting the line-pushing mechanism, while preserving connectivity of the shape throughout transformations.

This strategy divides S_D into $\sqrt{n}$ segments each of length $\sqrt{n}$ and proceeds in $\sqrt{n}$ phases. Informally, *DLC-Folding* performs two different operations, *turn* and *push* to fold the diagonal segments in every phase as follows: *turn* diagonals into straight lines, *push* them a distance of $\sqrt{n}$ towards the following diagonal segment and then inversely *turn* the lines again into diagonals. Figure 4.2 shows the transformations in the first phase where it folds the top segment of S_D. This transformation keeps folding segments above each other, until arriving at the bottom segment. By the end of the final phase, *DLC-Folding* forms a *Central line shape* (*nice shape*), which can then be trivially transformed into a line (see Proposition 3). Figures 4.1–4.6 demonstrate the above transformations on a diagonal of 25 nodes.

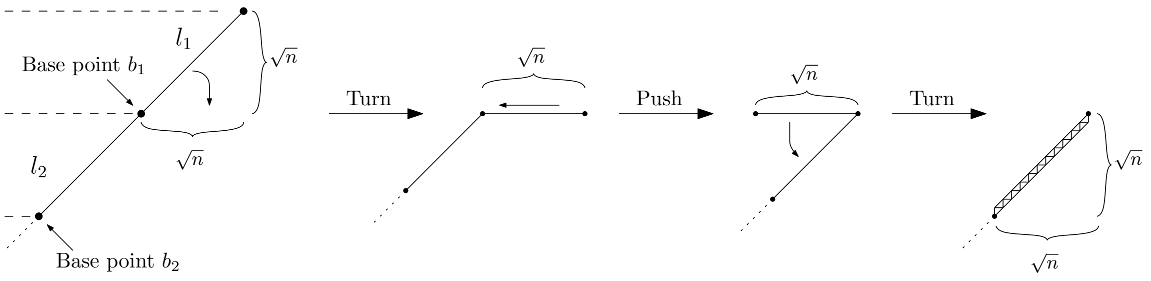


Figure 4.2: Folding the top segment of the diagonal line.

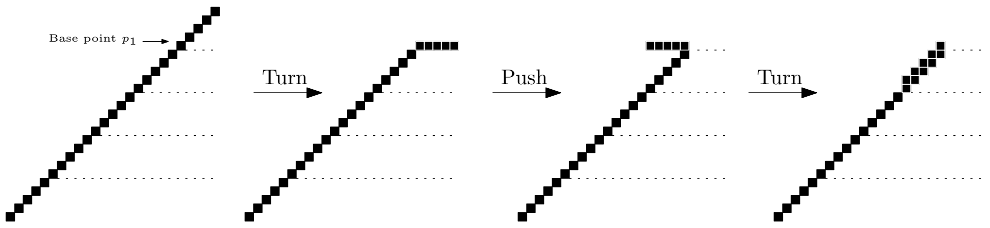


Figure 4.3: The first phase of folding.

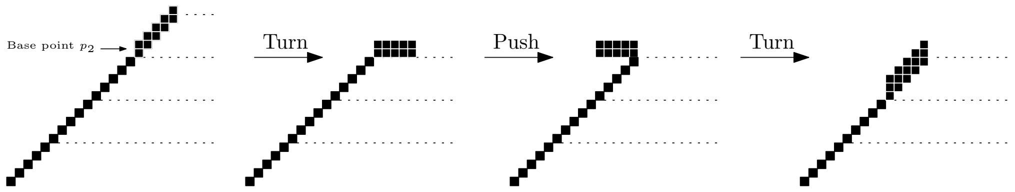


Figure 4.4: The second phase of folding.

More formally, let S_D denote a diagonal of n nodes, where (x, y) is the left bottom point of S_D. Then, S_D is divided into $\sqrt{n}$ segments, $l_1, l_2, \ldots, \sqrt{n}$, each of which has a length of

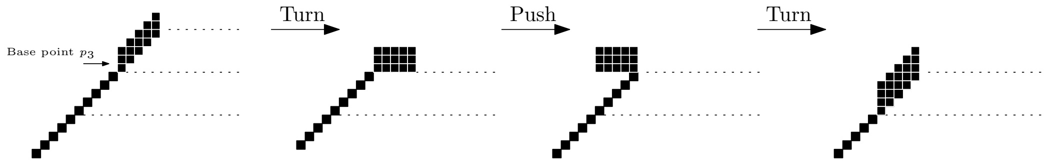


Figure 4.5: The third phase of folding.

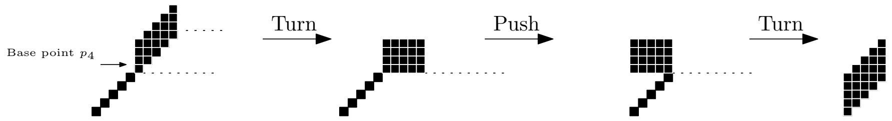


Figure 4.6: The fourth phase of folding. The resulting connected shape in the far right is a *nice* shape.

$\sqrt{n}$, where l_1 and $l_{\sqrt{n}}$ are the top and bottom segments of S_D, respectively. Each segment l_k, $1 \le k \le \sqrt{n}$, consists of $\sqrt{n}$ nodes occupying $(i,j),(i+1,j+1),\ldots,(i+\sqrt{n}-1,j+\sqrt{n}-1)$, where $i = x + h_k$ and $j = y + h_k$, for $h_k = n - k\sqrt{n}$. Here, l_k has a base point $b_k = (i,j)$, which is the bottommost node of l_k.

Due to symmetry, it is sufficient to demonstrate one orientation. The top segment l_1 folds around b_1 in the first phase as follows: (1) *Turn* all $\sqrt{n}$ nodes into the bottommost row of l_1 (brute-force line formation) from positions $(i,j),(i+1,j+1),\ldots,(i+\sqrt{n}-1,j+\sqrt{n}-1)$ into $(i,j),(i+1,j),\ldots,(i+\sqrt{n}-1,j)$. Thus, a horizontal line of length $\sqrt{n}$ is formed whose base point b_1 keeps place at (i,j). Then, (2) *Push* l_1 a distance of $\sqrt{n}$ towards the leftmost column y of S_D to occupy $(i-\sqrt{n},j),(i+1-\sqrt{n},j),\ldots,(i+\sqrt{n}-1-\sqrt{n},j)$. Next, (3) perform an inverse operation of (1) which converts the line l_1 into a diagonal again to stay atop the following diagonal segment l_2 on positions $(i-\sqrt{n},j-\sqrt{n}+1),(i+1-\sqrt{n},j-\sqrt{n}+2),\ldots,(i+\sqrt{n}-1-\sqrt{n},j)$ (except the base point b_1 which stays still in place, at $(i-\sqrt{n},j)$). By the end of this phase, two parallel diagonal segments l_1 and l_2 (shown in Figure 4.2) were formed as components of a new connected shape defined below.

Definition 13. *A Ladle is a connected shape of order n consisting of two parts, $\mathcal{D}$ and $\mathcal{S}$. For a given phase k, where $2 \le k \le \lceil\sqrt{n}\rceil$, we have $Ladle_k = \mathcal{D}_k + \mathcal{S}_k$, where both parts are connected via a base point $b_k = (i,j)$, such that:*

- $\mathcal{D}_k$, is a diagonal containing $n - k\sqrt{n} + 1$ nodes occupying $(x,y),(x+1,y+1),\ldots,(i,j)$, such that (x,y) are the left bottom point of $Ladle_k$, where $\sqrt{n} < i = j < n - \sqrt{n} + 1$. $\mathcal{D}_k$ is connected to $\mathcal{S}_k$ via (i,j).

- $\mathcal{S}_k$, is parallelogram consists of k parallel diagonal segments of size $k\sqrt{n}$ nodes formed $\sqrt{n}$ lines. $\mathcal{S}_k$ is connected to $\mathcal{D}_k$ via its bottommost node at (i, j).

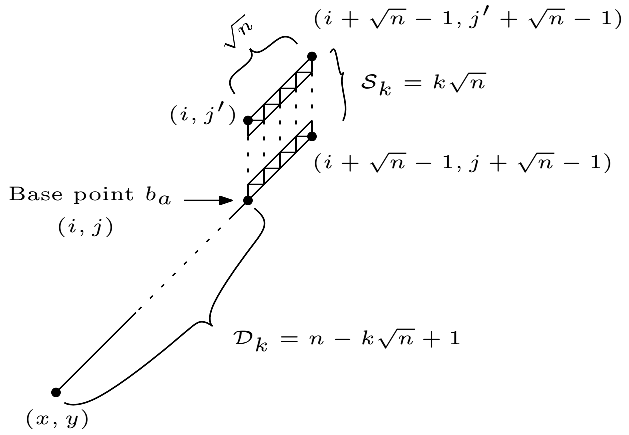

$\sqrt{n}$
$(i + \sqrt{n} - 1, j' + \sqrt{n} - 1)$
$\mathcal{S}_k = k\sqrt{n}$
(i, j')
$(i + \sqrt{n} - 1, j + \sqrt{n} - 1)$
Base point b_a
(i, j)
$\mathcal{D}_k = n - k\sqrt{n} + 1$
(x, y)

Figure 4.7: A *Ladle* shape in phase k, where $j\prime = j + k - 1$.

Figure 4.7 shows an example of *Ladle* shape in phase k. Now, the following lemmas prove *correctness* of *DLC-Folding*:

Lemma 15. *Let S_D be a diagonal of order n partitioned into $\lceil\sqrt{n}\rceil$ segments $l_1, l_2, ..., l_{\sqrt{n}}$. By the end of the first phase, DLC-Folding converts S_D into a Ladle.*

Proof. Due to symmetry, it is sufficient to show the implementation on the top segment of S_D. The first phase formed (1) a diagonal part occupies $(x, y), (x + 1, y + 2), \ldots, (x + n - \sqrt{n} - 1, y + n - \sqrt{n} - 1)$ and (2) two parallel diagonal segments of $2\sqrt{n}$ nodes where both are connected via the base point $(x + n - \sqrt{n} - 1, y + n - \sqrt{n} - 1)$. Observe that this point is the topmost node of the diagonal part and the bottommost of the two parallel diagonal segments while the rest nodes are making a single diagonal line. Thus, this new shape meets all conditions of a *Ladle* mentioned in Definition 13.

□

Lemma 16. *Given a Ladle of n nodes in phase k, where $1 < k \leq \sqrt{n}$. Then, DLC-Folding increases the size of $\mathcal{S}_k$ by $\sqrt{n}$ and decreases the length of $\mathcal{D}_k$ by $\sqrt{n}$ in phase $k + 1$.*

Proof. The size of the *Ladle* $= |n|$ must be the same each phase and at all times. In phase k, a *Ladle*$_k$ consists of two parts, $\mathcal{D}_k = |n - k\sqrt{n} + 1|$ and $\mathcal{S}_k = |k\sqrt{n}|$ connected via a common node (i, j) (see Definition 13). Now, we want to fold the $\mathcal{S}_k$ section, which has k

segments of length $\sqrt{n}$ placed diagonally on top of each other. Without loss of generality, transfer all $\sqrt{n}$ vertical lines downward to the bottommost row i of $\mathcal{S}_k$, which must form k horizontal lines by entirely filling up the k bottom rows of $\mathcal{S}_k$. As a result, the horizontal lines form a rectangle, as seen in Figure 4.8(a), which can be pushed $\sqrt{n}$ distance to the left, as shown in Figure 4.8(b). Hence, the strategy *turns* these lines inversely above the next diagonal segment (except the rightmost one), as depicted in Figure 4.8(c). By the end of phase $k+1$, a new *Ladle* has been created, consisting of $\mathcal{D}_{k+1} = |n-(k-1)\sqrt{n}+1|$ and $\mathcal{S}_{k+1} = |(k+1)\sqrt{n}|$ connected via the base point b_{k+1} at $(i-\sqrt{n}, j-\sqrt{n})$. Therefore, in phase $k+1$, the size of $\mathcal{S}_{k+1}$ increased by $\sqrt{n}$ nodes, while the length of $\mathcal{D}_{k+1}$ decreased by $\sqrt{n}$. Thus, for any phase k, where $1 < k \leq \sqrt{n}$, this holds trivially and inductively.

□

Next, we prove that *DLC-Folding* transforms S_D into a *Central Line* (*nice*) shape within $\sqrt{n}$ phases.

Lemma 17. *DLC-Folding converts S_D into a Central Line shape through $\sqrt{n}$ phases.*

Proof. From Lemma 15, by phase $k=2$, S_D converts into a *Ladle*$_2$ of $\mathcal{D}_2 = |n-2\sqrt{n}+1|$ and $\mathcal{S}_2 = |2\sqrt{n}|$. Then, according to Lemma 16, by the final phase $k=\sqrt{n}$, $\mathcal{D}_{\sqrt{n}} = \phi$ is exhausted, whereas the parallelogram part acquires all n nodes, $\mathcal{S}_{\sqrt{n}} = |n|$. Hence, the resulting new shape is a compressed into $\sqrt{n} \times \sqrt{n}$ lines, which is a *Central Line* shape. □

Let us now analyse the running time of *DLC-Folding*.

Lemma 18. *Given a diagonal S_D of order n partitioned into $\sqrt{n}$ segments, DLC-Folding folds the topmost (bottommost) segment in $O(n)$ moves.*

Proof. By a brute-force line formation of the first phase, the top (bottom) segment of S_D of length $\sqrt{n}$ becomes a line (similar of Figure 3.1(b)), which is trivially computed by:

$$1+2+...+(\sqrt{n}-1) = \frac{\sqrt{n}(\sqrt{n}-1)}{2} = \frac{n-\sqrt{n}}{2} = O(n).$$

The line then pushes at most $\sqrt{n}$ moves before turning inversely into a diagonal at the same cost of $(n-\sqrt{n})/2 = O(n)$. Therefore, the first segment folds in a total cost $t_1$of moves at most:

$$t_1 = \frac{n-\sqrt{n}}{2} + \sqrt{n} + \frac{n-\sqrt{n}}{2} = n - \sqrt{n} + \sqrt{n} = n$$

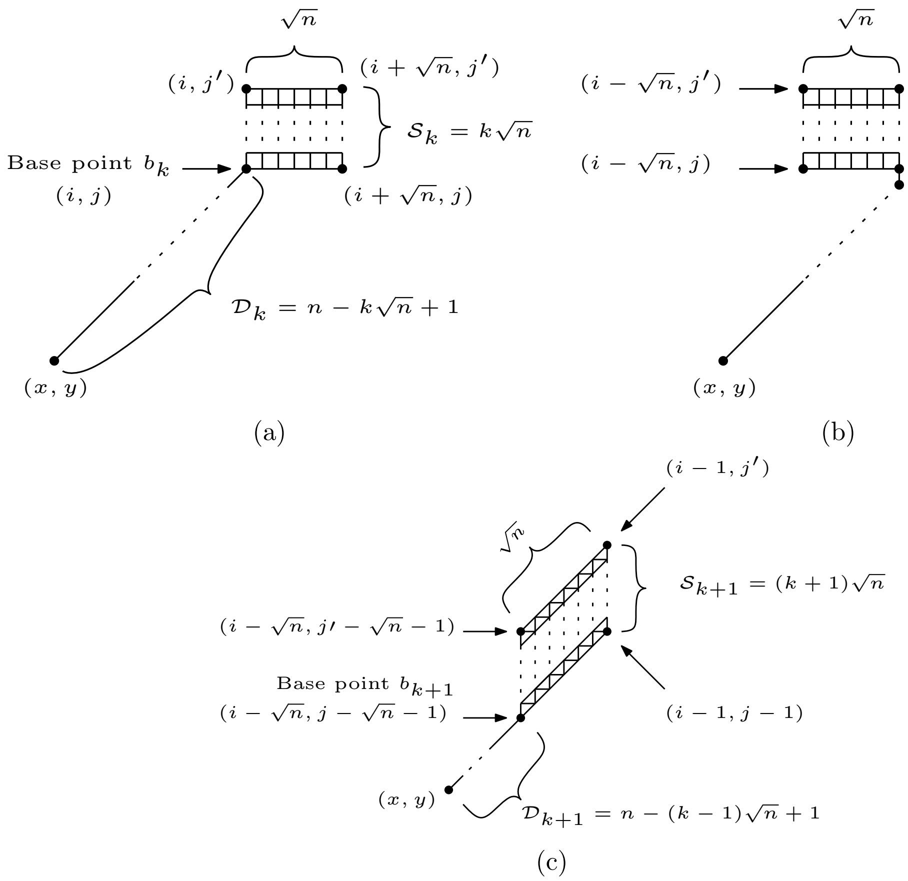


Figure 4.8: Folding a $Ladle_k$ over phase k, where $j\prime = j + k - 1$. See Lemma 16 for further explanation.

$$= O(n).$$

□

Lemma 19. *By the end of phase k, for all $1 < k \leq \sqrt{n}$, DLC-Folding takes at most $O(n)$ moves to transform $Ladle_k$ into $Ladle_{k+1}$.*

Proof. During phase k, all $\sqrt{n}$ lines of $\mathcal{S}_k = |k\sqrt{n}|$ convert into k lines within a total run of moves by at most:

$$1 + 2 + ... + (\sqrt{n} - 1) = O(n). \tag{4.1}$$

Now, those k lines push a distance of $\sqrt{n}$ in a total of at most:

$$k\sqrt{n} = O(n), \tag{4.2}$$

moves. Then, the $\sqrt{n}$ lines turn diagonally above the following segment incurring the same cost of (4.1) by at most:

$$\frac{n-\sqrt{n}}{2} = O(n). \tag{4.3}$$

With this, the total cost of phase k is given by summing (4.1), (4.2), and (4.3):

$$\begin{aligned} t_k &= \frac{n-\sqrt{n}}{2} + k\sqrt{n} + \frac{n-\sqrt{n}}{2} = n - \sqrt{n} + k\sqrt{n} \\ &= O(n). \end{aligned} \tag{4.4}$$

This holds trivially from phase 2 and inductively for every phase k, where $1 < k \leq \sqrt{n}$. □

Altogether, Proposition 3 and Lemmas 18 and 19 imply that,

Theorem 7. *DLC-Folding solves* DiagonalToLineConnected *in $O(n\sqrt{n})$ moves.*

Proof. By Lemma 18, *DLC-Folding* creates a *Ladle* in a total cost of moves by at most:

$$T_1 = \frac{n-\sqrt{n}}{2}. \tag{4.5}$$

Now, by Lemma 19, the total running time for all k phases, $1 < k \leq \sqrt{n}$, is given as follows:

$$\begin{aligned} T_2 &= \sum_{i=1}^{\sqrt{n}-1} n - \sqrt{n} + i\sqrt{n} = n\sqrt{n} - 2n - \sqrt{n} + \sqrt{n}\sum_{i=1}^{\sqrt{n}-1} i \\ &= n\sqrt{n} - 2n - \sqrt{n} + \left(\frac{n\sqrt{n}-n}{2}\right) = \frac{n\sqrt{n} - 5n - 2\sqrt{n}}{2} \\ &= O(n\sqrt{n}). \end{aligned} \tag{4.6}$$

Sum the cost of the first phase in (4.5) to the remaining phases (4.6) to obtain the total cost of moves T_3 for this transformation as follows:

$$T_3 = T_1 + T_2$$

$$= \frac{n-\sqrt{n}}{2} + \frac{n\sqrt{n}-5n-2\sqrt{n}}{2} = \frac{n\sqrt{n}-4n-3\sqrt{n}}{2}$$
$$= O(n\sqrt{n}). \tag{4.7}$$

Since the resulting shape of *DLC-Folding* is a *Central Line* (*nice*) shape, which can be transformed into a line S_L in $O(n)$ moves *(see Proposition 3)*, therefore, the overall cost T required to transform S_D into S_L, is bounded by:

$$\begin{aligned} T &= T_3 + O(n) \\ &= O(n\sqrt{n}). \end{aligned}$$

□

4.1.2 Extending: An $O(n\sqrt{n})$-time Transformation

DLC-Extending is another approach for transforming the diagonal S_D into a straight line S_L in $O(n\sqrt{n})$ moves that preserves connectivity throughout transformations. This algorithm may not have an advantage over the above, though it has influenced the development of the following Hamiltonian transformations in Section 4.2 (recall the dependency digram in Figure 1.5). Thus, we devote a separate section to present this transformation in more details.

Using a similar partitioning of S_D into $\sqrt{n}$ segments, this strategy performs two primitives, *turn* and *push* in each of the $\sqrt{n}$ phases. In each phase k, $1 \leq k \leq \sqrt{n}$, the extension is implemented as follows: (1) *Turn*: a diagonal segment l_k of S_D transforms into a line by a brute-force line formation. (2) *Push*: the line formed in (1) pushes towards the other end of the S_D, extending the k-length target straight line by $\sqrt{n}$. Repeat this in each phase shall arrive at the ultimate straight line S_L of length n. Figure 4.9 shows an implementation of *DLC-Extending* on a diagonal of 25 nodes.

More formally, consider an initial diagonal S_D of n nodes partitioned as in *DLC-Folding* (see Section 4.1.1). For the sake of simplicity, we only examine converting from the topmost segment $l_1 \in S_D$ occupying cells $(i,j),(i+1,j+1),\ldots,(i+\sqrt{n}-1,j+\sqrt{n}-1)$, where $i = x+h_k$ and $j = y+h_k$, for $h_k = n-k\sqrt{n}$. Analogously, this corresponds to the bottommost section. Initially, the diagonal l_1 performs a brute-force line formation to *turn* into a *line*, i.e. collect all nodes to occupy the leftmost columns of l_1 on $(i,j),(i,j+1),\ldots,(i,j+\sqrt{n}-1)$. Due to symmetry, the same transformation holds to collect them at the bottommost row.

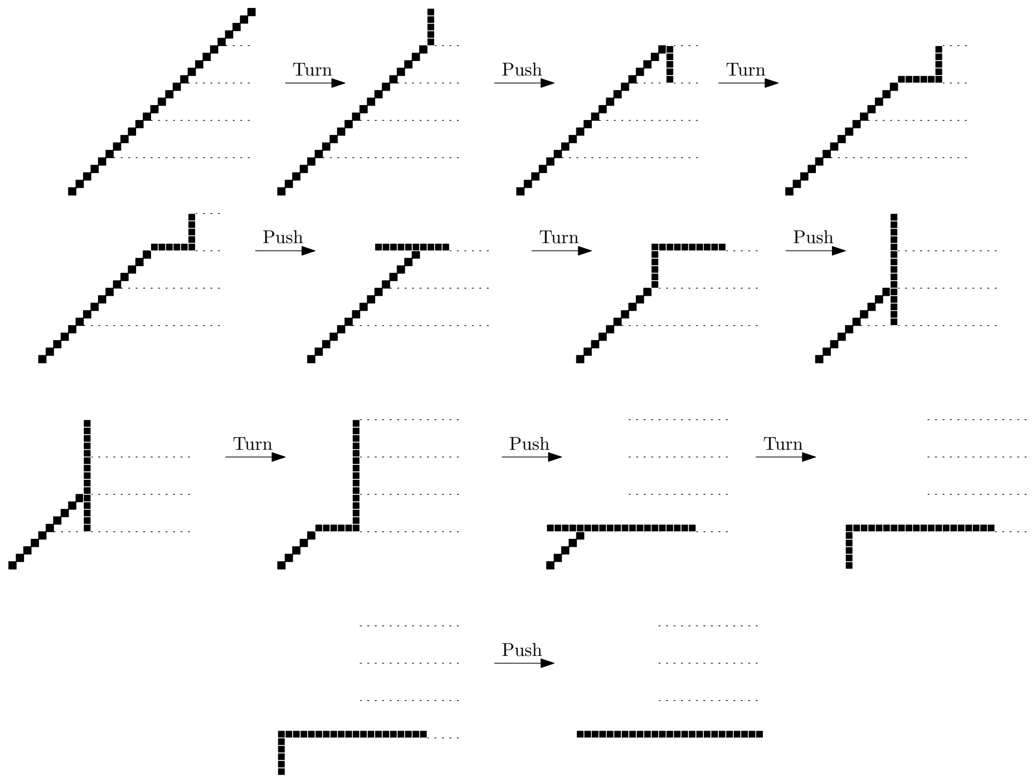


Figure 4.9: An implementation of *DLC-Extending* on a diagonal of 25 nodes.

Then, the line l_1 pushes $\sqrt{n}$ moves to positions $(i, j - \sqrt{n}), (i, j + 1 - \sqrt{n}), \ldots, (i, j + \sqrt{n} - 1 - \sqrt{n})$. Figure 4.10 demonstrates the first phase.

In the second phase, the next diagonal segment l_2 *turns* into a *line* to occupy a position where both l_1 and l_2 are connected perpendicularly via $(i, j - \sqrt{n})$, as depicted in Figure 4.11). Hence, l_1 changes its direction to be in line with l_2 and both merge into a single straight line of length $2\sqrt{n}$. This creates a specific *connected* shape, called T_{shape} and defined as follows:

Definition 14. *A T_{shape} is a connected shape of order n consisting of two parts, $\mathcal{D}$ and $\mathcal{S}$. Both are connected via a common intersection point (i, j). In phase k, $1 < k < \sqrt{n}$, $T^k_{shape} = \mathcal{D}_k + \mathcal{S}_k$ is defined as follows:*

- *$\mathcal{D}_k$ is a diagonal line containing $n - k\sqrt{n} + 1$ nodes occupying $(x, y), (x + 1, y + 1), \ldots, (i, j)$, where x and y are the leftmost column and the bottommost row of T^k_{shape},*

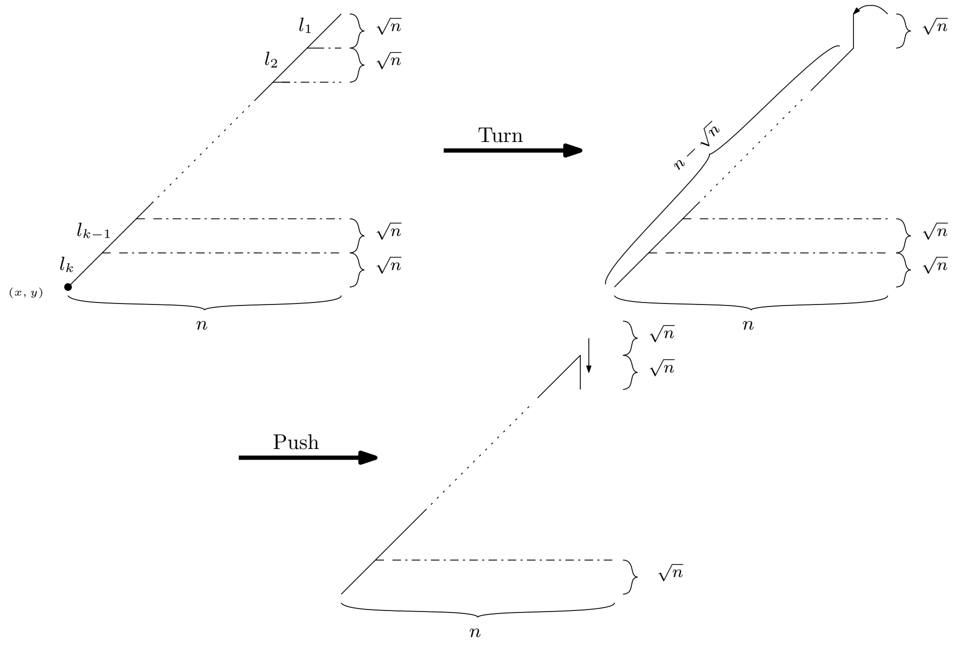


Figure 4.10: Two primitives, *turn* and *push*, during the first phase.

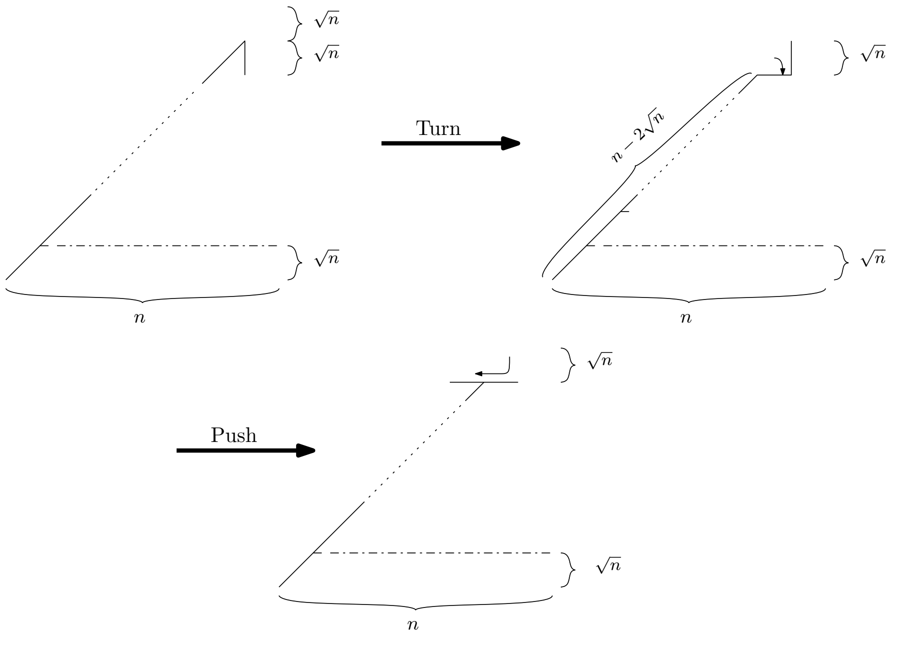


Figure 4.11: Two primitives, *turn* and *push*, during the second phase.

respectively, for all $\sqrt{n} < i = j < n - \sqrt{n} + 1$.

- $\mathcal{S}_k$ *is a horizontal or vertical line of length* $k\sqrt{n}$ *nodes occupying* $(i - \sqrt{n} - 1, j), (i + 1 - \sqrt{n} - 1, j), \ldots, (i\prime, j)$ *or* $(i, j - \sqrt{n} - 1), (i, j + 1 - \sqrt{n} - 1), \ldots, (i, j\prime)$*, respectively, where* $i\prime = i + k\sqrt{n} - 1$ *and* $j\prime = j + k\sqrt{n} - 1$

Figure 4.12 shows an example of $T^k_{shape} = \mathcal{D}_k + \mathcal{S}_k$ in phase k. Now, we prove the correctness of *DLC-Extending*. First, we show the formation of a T_{shape} from a diagonal S_D.

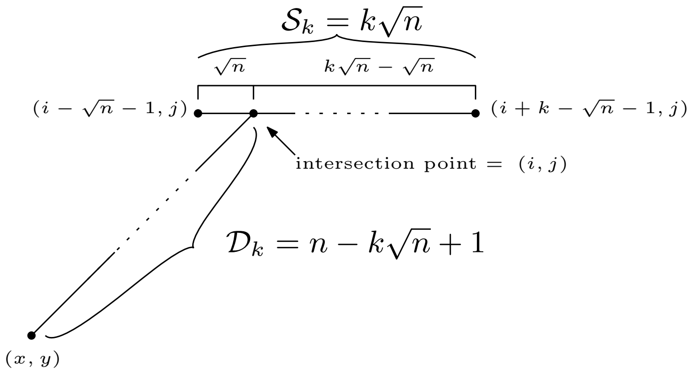

$\mathcal{S}_k = k\sqrt{n}$
$\sqrt{n}$
$k\sqrt{n} - \sqrt{n}$
$(i - \sqrt{n} - 1, j)$
$(i + k - \sqrt{n} - 1, j)$
intersection point = (i, j)
$\mathcal{D}_k = n - k\sqrt{n} + 1$
(x, y)

Figure 4.12: A T_{shape} in phase k.

Lemma 20. *Let* S_D *be a diagonal of order* n *partitioned into* $\lceil\sqrt{n}\rceil$ *segments* $l_1, l_2, ..., l_{\sqrt{n}}$. *By the end of the second phase, DLC-Extending converts* S_D *into a* T_{shape}.

Proof. Applying the two main primitives, *turn* and *push*, sequentially on the two topmost segments, l_1 and l_2, shall form two connected components: (1) A diagonal at $(x, y), (x+1, y+1), \ldots, (i, j)$, where $i = x+n-2\sqrt{n}-2$ and $j = y+n-2\sqrt{n}-2$. (2) A horizontal or vertical line of length $2\sqrt{n}$ occupying $(i-2\sqrt{n}, j), \ldots, (i+2\sqrt{n}, j)$ or $(i, j-2\sqrt{n}), \ldots, (i, j+2\sqrt{n})$. Both (1) and (2) intersect at (i, j) and meet Definition 14. □

The following lemma demonstrates that the two primitives of *DLC-Extending* hold for any phase k, for all $2 \leq k < \sqrt{n}$.

Lemma 21. *Given a* T^k_{shape} *of* n *nodes in phase* k*, for all* $2 \leq k < \sqrt{n}$. *Then, DLC-Extending transforms* T^k_{shape} *into* T^{k+1}_{shape} *in phase* $k + 1$.

Proof. By Definition 14, assume $\mathcal{S}_k$ is *horizontal* in phase k (this is sufficient as the *vertical* holds due to symmetry). First, the topmost diagonal segment of $\mathcal{D}_k$ *turns* into a line of $\sqrt{n}$ nodes from cells $(i-\sqrt{n}+1, j-\sqrt{n}+1), \dots, (i-1, j-1)$ into $(i-\sqrt{n}+1, j-\sqrt{n}+1), \dots, (i-\sqrt{n}+1, j-1)$. Thus, the shape's connectivity is still preserved, and the new constructed *line* is now perpendicular with $\mathcal{S}_k$. Next, the line segment $\mathcal{S}_k$ *pushes* towards the new line vertically by a length of $\sqrt{n}$ on positions $(i-\sqrt{n}, j-2\sqrt{n}), \dots, (i-\sqrt{n}, j+k-2\sqrt{n})$. Hence, a new *connected* shape in phase $k+1$ is now consisting of $\mathcal{D}_{k+1}$ of length $n-(k-1)\sqrt{n}$ and $\mathcal{S}_{k+1}$ of length $(k+1)\sqrt{n}$, where both intersect at a common point $(i-\sqrt{n}-1, j)$; as shown in Figure 4.13. Finally, we derive that *DLC-Extending* decreases the length of $\mathcal{D}_{k+1}$ by $\sqrt{n}$ and increases the length of $\mathcal{S}_{k+1}$ by $\sqrt{n}$ at the end of phase $k+1$. □

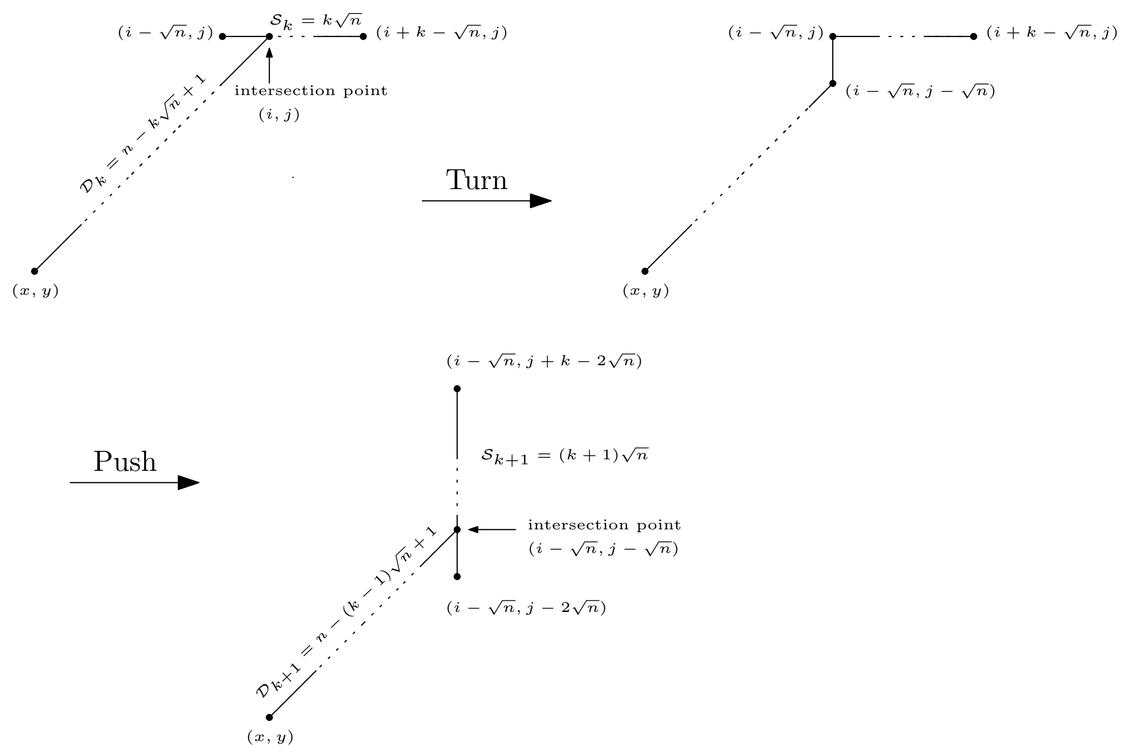

$(i-\sqrt{n}, j)$
$\mathcal{S}_k = k\sqrt{n}$
$(i+k-\sqrt{n}, j)$
intersection point
(i, j)
$\mathcal{D}_k = n - k\sqrt{n} + 1$
(x, y)
Turn
$(i-\sqrt{n}, j)$
$(i+k-\sqrt{n}, j)$
$(i-\sqrt{n}, j-\sqrt{n})$
(x, y)
Push
$(i-\sqrt{n}, j+k-2\sqrt{n})$
$\mathcal{S}_{k+1} = (k+1)\sqrt{n}$
intersection point
$(i-\sqrt{n}, j-\sqrt{n})$
$(i-\sqrt{n}, j-2\sqrt{n})$
$\mathcal{D}_{k+1} = n-(k-1)\sqrt{n}+1$
(x, y)

Figure 4.13: An implementation of *DLC-Extending* on T^k_{shape} during phase k.

By the end of the final phase $\sqrt{n}$, the following lemma shows that *DLC-Extending* forms a straight line S_L of n nodes.

Lemma 22. *DLC-Extending transforms S_D into a straight line S_L by the end of phase $\sqrt{n}$.*

Proof. By Lemmas 20 and 21, the end of phase $\sqrt{n}-1$ shall form:

$$\begin{aligned} T_{shape}^{\sqrt{n}-1} &= \mathcal{D}_{\sqrt{n}-1} + \mathcal{S}_{\sqrt{n}-1} \\ &= \left[n - (\sqrt{n}-1)\sqrt{n}\right] + \left[(\sqrt{n}-1)\sqrt{n}\right] \\ &= \left[n - (n-\sqrt{n})\right] + \left[(n-\sqrt{n})\right] \\ &= \left[\sqrt{n}\right] + \left[(n-\sqrt{n})\right], \end{aligned}$$

where $|\mathcal{D}_{\sqrt{n}-1}| = \sqrt{n}$ and $|\mathcal{S}_{\sqrt{n}-1}| = n-\sqrt{n}$. Therefore, *DLC-Extending* forms the straight line S_L, via *turn* and *push*, by the end of the finial phase $\sqrt{n}$. □

To sum up, the following theorem states that, given any diagonal of n nodes, this transformation solves the DiagonalToLineConnected problem within $O(n\sqrt{n})$ moves.

Theorem 8. *DLC-Extending solves the* DiagonalToLineConnected *problem in* $O(n\sqrt{n})$ *moves.*

Proof. By Lemmas 20, 21 and 22, *DLC-Extending* performs transformations using the two main primitives, *turn* and *push*, throughout a total of $\sqrt{n}$ phases. The brute-force line formation of '*turn*' on a single diagonal segment costs a number of moves t_1 given by:

$$\begin{aligned} t_1 &= 1+2+\ldots+(\sqrt{n}-1) = \sum_{i=1}^{\sqrt{n}-1} i = \frac{\sqrt{n}(\sqrt{n}-1)}{2} = \frac{n-\sqrt{n}}{2} \\ &= O(n). \end{aligned} \tag{4.8}$$

With that, multiply 4.8 by $\lceil\sqrt{n}\rceil$ to obtain the cost T_1 of total moves required for all $\lceil\sqrt{n}\rceil$ phases (diagonal segments):

$$\begin{aligned} T_1 &= t_1 \cdot \lceil\sqrt{n}\rceil = \frac{n-\sqrt{n}}{2} \cdot \lceil\sqrt{n}\rceil = \frac{n\sqrt{n}-n}{2} \\ &= O(n\sqrt{n}). \end{aligned} \tag{4.9}$$

Then, the cost of the *push* primitive increases gradually every phase k by a factor of $\lceil\sqrt{n}\rceil$, for all $1 \le k \le \sqrt{n}$. Thus, in phase k, it costs a total number of moves t_2 that asymptotically bounded by (see Lemma 1):

$$t_2 = 2k. \tag{4.10}$$

The total moves T_2 of '*push*' is the sum of 4.10 plus the cost of pushing the last segment of length $\sqrt{n}$:

$$\begin{aligned}
T_2 &= \sum_{i=1}^{\sqrt{n}-1} t_2 + 2\sqrt{n} \\
&= \sum_{i=1}^{\sqrt{n}-1} 2i + 2\sqrt{n} = 2\sum_{i=1}^{\sqrt{n}-1} i + 2\sqrt{n} = 2 \cdot \frac{n-\sqrt{n}}{2} + 2\sqrt{n} = n - \sqrt{n} + 2\sqrt{n} \\
&= O(n). \qquad (4.11)
\end{aligned}$$

Finally, putting 4.9 and 4.11 together, we conclude that *DLC-Extending* takes a total cost of moves T given by,

$$\begin{aligned}
T &= T_1 + T_2 \\
&= \frac{n\sqrt{n} - n}{2} + n + \sqrt{n} \\
&= O(n\sqrt{n}).
\end{aligned}$$

□

4.2 Walk-Through-Path: An $O(n \log n)$-time Hamiltonian shapes transformation

In this section, we build upon the above strategies aiming to design a very efficient and general transformation that are additionally able to keep the shape connected throughout their course. We present *Walk-Through-Path*, an $O(n \log n)$-time transformation for the HamiltonianConnected problem that works for any pairings shapes $(S_I, S_F) \in \mathcal{H}$ of the same order and belong to the family of *Hamiltonian shapes* denoted $\mathcal{H}$ (see Definition 11). Recall that a *Hamiltonian shape* is any connected shape S whose *associated graph* $G(S)$ contains a Hamiltonian path (consult also [84] for Hamiltonian paths). In the continuous setting, a natural analogue of Hamiltonian paths is space-filling curves [104] (e.g. Hilbert curve), referring to a curve which covers the entire two-dimensional square (or n-dimensional in general).

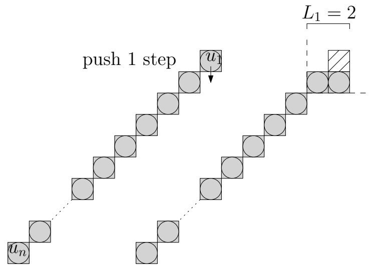
Our transformation starts from one endpoint of the Hamiltonian path of S_I and applies a recursive successive doubling technique to transform S_I into a straight line S_L in $O(n \log n)$ time. By replacing S_I with S_F in *Walk-Through-Path* and reversing the resulting transformation, one can then go from S_I to S_F in the same asymptotic time.

4.2.1 Transforming diagonal shape into line shape

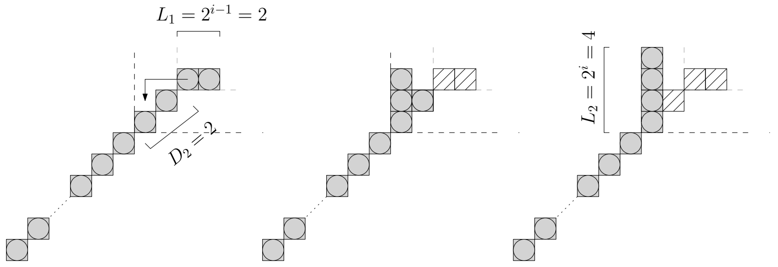
We first demonstrate the core recursive technique of this strategy in a special case which is sufficiently sparse to allow local reconfigurations without the risk of affecting the connectivity of the rest of the shape. In this special case, S_I is a diagonal of any order and observe that $S_I, S_F \in \mathcal{H}$ holds for this case. We then generalise this recursive technique to work for any $S_I \in \mathcal{H}$ and add to it the necessary sub-procedures that can perform local reconfiguration in *any* area (independently of how dense it is), while ensuring that global connectivity is always preserved.

Let S_I be a diagonal of n nodes $u_n, u_{n-1}, \ldots, u_1$, occupying cells $(x, y), (x+1, y+1), \ldots, (x+n-1, y+n-1)$, respectively. Assume for simplicity of exposition that n is a power of 2; this can be dropped later. As argued above, it is sufficient to show how S_I can be transformed into a straight line S_L. In phase $i = 0$, the top node u_1 moves one position to align with u_2 and form a line L_1 of length 2, as depicted in Figure 4.14(a). Next phase, L_1 moves and turns to align with u_4, then repeat whatever done in phase $i = 0$ again on nodes u_3 and u_4 (where both form a diagonal segment D_2 to create a line L'_1, and then combine the two perpendicular lines L_1 and L'_1 into a line L_2 of length 4, as shown in Figure 4.14(b).

In any phase i, for all $1 \leq i \leq \log n$, a line L_i occupies 2^i consecutive cells in a terminal subset of S_I (see an example in Figure 4.15(a)). L_i moves through a shortest path towards the far endpoint of the next diagonal segment D_i of length 2^i (Figure 4.15(b)). Note that for general shapes, this move shall be replaced by a more general *Line-Walk* operation (defined in the sequel). By a recursive call on D_i, D_i transforms into a line L'_i (Figure 4.15(c)). Finally, the two perpendicular lines L_i and L'_i are combined in linear time into a straight line L_{i+1} of length 2^{i+1} (Figure 4.15(d)). Observe that connectivity might be broken as L'_i moving up and L_i pushing left in Figure 4.15(d); hence, this case can be resolved in many ways, such as Figure 2.9 in Proposition 4. By the end of phase $\log n$, a straight line S_L of order n has been formed.



(a) First phase.



(b) Second phase.

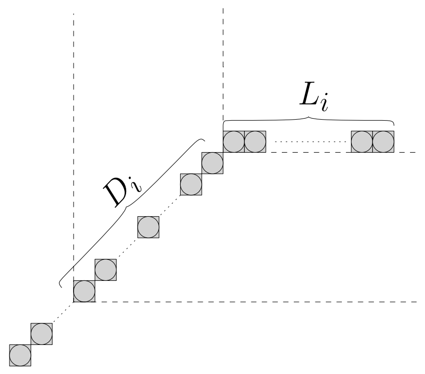
Figure 4.14: First and second phase of *Walk-Through-Path* transformation on the diagonal shape.

4.2.2 Transforming Hamiltonian shapes into a straight line

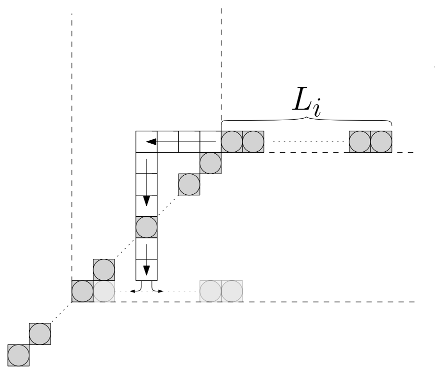
A core technical challenge in making the above transformation work in the general case, is that Hamiltonian shapes do not necessarily provide free space, thus, moving a line has to take place through the remaining configuration of nodes while at the same time ensuring that it does not break their and its own connectivity. In the more general *LineWalk* operation that we now describe, we manage to overcome this by exploiting *transparency* of line moves, according to which a line L can *transparently* walk through any configuration S (independently of the latter's density); see Proposition 4.

LineWalk. At the beginning of any phase i, there is a terminal straight line L_i of length 2^i containing the nodes $v_1, \ldots, v_{2^i}$, which is connected to an $S_i \subseteq S_I$, such that S_i consists of the 2^i subsequent nodes, that is $v_{2^i+1}, \ldots, v_{2^{i+1}}$. Observe that S_i is the next terminal sub-path of the remaining Hamiltonian path of S_I. We distinguish the following cases:

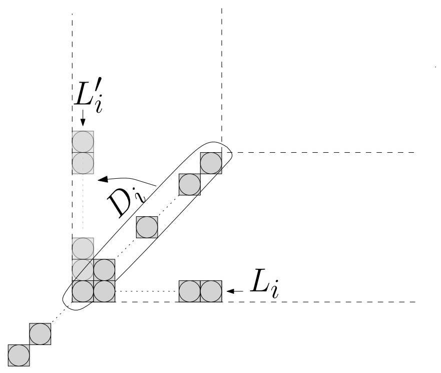
(1) If L_i and S_i are already forming a straight line, then go to phase $i + 1$.



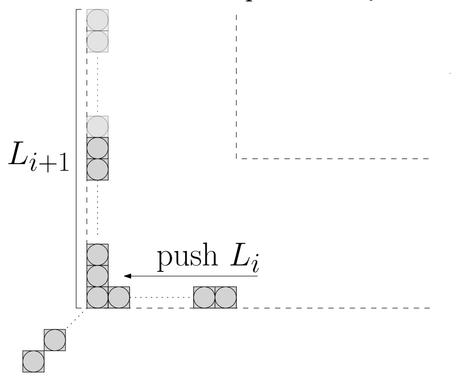
(a) A line L_i and a diagonal segment D_i both of length 2^i



(b) L_i moves through a shortest path towards the far endpoint of D_i.



(c) D_i recursively transforms into a line L'_i.



(d) A line L_{i+1} of length 2^{i+1} formed by combining L_i and L'_i.

Figure 4.15: A snapshot of phase i of HamiltonianToLine transformation applied on a diagonal line shape. Light grey cells represent the ending positions of the corresponding moves depicted in each sub-figure.

(2) If S_i is a line perpendicular to L_i, then combine them into a straight line by pushing L_i to extend S_i and go to phase $i+1$.

(3) Otherwise, check if the (Manhattan) distance between v_{2^i} and $v_{2^{i+1}}$ is $\delta(v_{2^i}, v_{2^{i+1}}) \leq 2^i$, then L_i moves from $v_{2^i} = (x, y)$ vertically or horizontally towards either node (x, y') or (x', y) in which L_i turns and keeps moving to $v_{2^{i+1}} = (x', y')$ on the other side of S_i.

(4) If not, L_i must first pass through a middle node of S_i at $v_{2^i+2^{i-1}} = (x'', y'')$, therefore L_i repeats (3) twice, from v_{2^i} to $v_{2^i+2^{i-1}}$ and then towards $v_{2^{i+1}}$.

Note that cases (3) and (4) ensure that L_i is not disconnected from the rest of the shape. Moreover, moving L_i must be performed in a way that respects transparency (Proposition 4), so that connectivity of the remaining shape is always preserved and its configuration is restored to its original state. These details are described later in this section.

Algorithm 1, HamiltonianToLine, gives a general strategy to transform any Hamiltonian shape $S_I \in \mathcal{H}$ into a straight line in $O(n \log n)$ moves. In every phase i, it moves a terminal line L_i of length 2^i a distance 2^i higher on the Hamiltonian path through a *LineWalk* operation. This leaves a new terminal sub-path S_i of the Hamiltonian path, of length 2^i. Then the general procedure is recursively called on S_i to transform it into a straight line L'_i of length 2^i. Finally, the two straight lines L_i and L'_i which are perpendicular to each other are combined into a new straight line L_{i+1} of length 2^{i+1} and the next phase begins. The output of HamiltonianToLine is a straight line S_L of order n.

Algorithm 1: HamiltonianToLine(S)

```
$S = (u_0, u_1, ..., u_{|S|-1})$ is a Hamiltonian shape
Initial conditions: $S \leftarrow S_I$ and $L_0 \leftarrow \{u_0\}$
for $i = 0, \dots, \log |S|$ do
    LineWalk($L_i$)
    $S_i \leftarrow$ select($2^i$) // select the next terminal subset of $2^i$ consecutive
        nodes of $S$
    $L'_i \leftarrow$ HamiltonianToLine($S_i$) // recursive call on $S_i$
    $L_{i+1} \leftarrow$ combine($L_i, L'_i$) // combines $L_i$ and $L'_i$ into a new straight line
        $L_{i+1}$
end
Output: a straight line $S_L$
```

4.2.3 Correctness and runtime analysis

Now, we are ready to show correctness of HamiltonianToLine transformation, which is capable of transforming any Hamiltonian shape $S \in \mathcal{H}$ into a line shape of the same order, while preserving connectivity.

Lemma 23. *Starting from an initial Hamiltonian shape $S_I \in \mathcal{H}$ of order n, HamiltonianToLine forms a straight line $S_L \in \mathcal{H}$ of length n.*

Proof. By the beginning of the final phase, the shape configuration consists of two parts, a straight line L of length $2^{\log n - 1}$ and a shape S of $2^{\log n - 1}$ nodes. During this phase, L

performs a *LineWalk* operation, S transforms recursively into L' and then L combines with L' into a straight line S_L of length $2^{logn} = n$. Consequently, S_L shall occupy n consecutive cells on the grid, either vertically or horizontally. □

Lemma 24. *The operation of Line-Walk preserves the whole connectivity of the shape during phase i, where $1 \leq i \leq \log n$.*

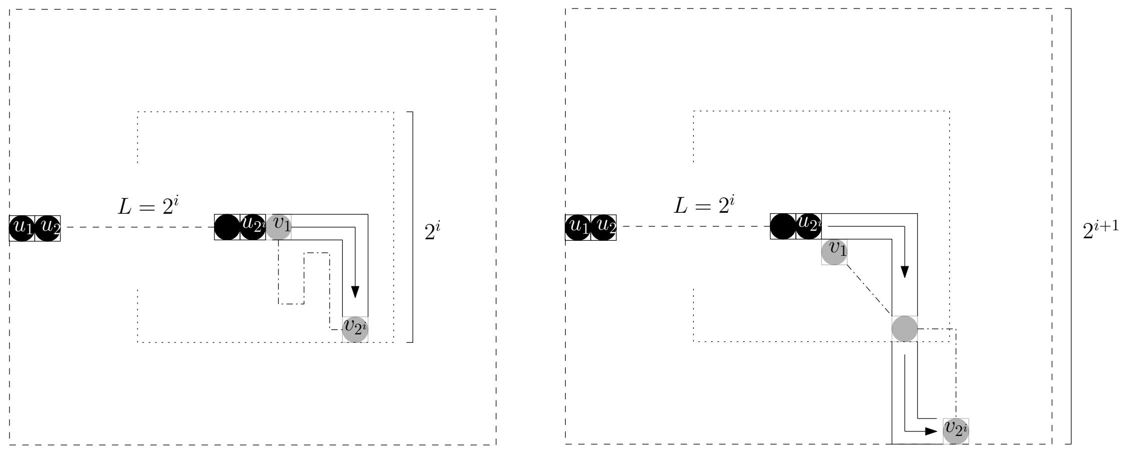
Proof. Let $S_I \in \mathcal{H}$ be a Hamiltonian shape of order n in phase i, which terminates at a straight line L_i of length 2^i nodes, starting from v_1 to v_{2^i}. During phase i, this transformation doubles the size of L_i by merging with the subsequent 2^i nodes, which starts from v_{2^i+1} to $v_{2^{i+1}}$ on the Hamiltonian path.

We now show case (1) and (2) of the *Line-Walk* operation on a horizontal L_i as the other cases are symmetric by rotating the shape 90°, 180° or 270° clockwise. In case 1, L_i and S_i are already forming a straight line L_{i+1} of length 2^{i+1}, and hence, the whole configuration of the shape remains unchanged. In case (2), L_i and S_i are forming two perpendicular straight lines in which L_i can easily push into S_i and extend it by 2^i. By exploiting *transparency* of line moves in Proposition 4, as L_i pushes and S_i extends to form L_{i+1}, they are replacing and restoring any occupied cell along their way through *any* configuration (independently of how density is). As a result, the *Line-Walk* operation preserves connectivity of L_i, S_i and the whole shape.

Now, let L_i and S_i be of the same configuration of case (3) or (4) described above, where L_i has length of 2^i and S_i consists of 2^i nodes $v_{2^i+1}, \ldots, v_{2^{i+1}}$, occupying multiple rows and columns. Due to symmetry, assume L_i is horizontal on $(x, y), (x + 1, y), \ldots, (x + 2^i, y)$ and S_i is the next terminal sub-path of the remaining Hamiltonian path. The Manhattan distance between $v_{2^i} = u$ and $v_{2^{i+1}} = v$ (given by $\delta(u, v) = |u_x - v_x| + |u_y - v_y|$) computes the path that the line L_i will go through to reach the far endpoint of S_i. There are two possible paths of one corner from u to v: (1) starts horizontally from cell (u_x, u_y) then turns at (v_x, u_y) continuing vertically towards (v_x, v_y), or (2) starts from (u_x, u_y) then turns at (u_x, v_y) continuing horizontally towards (v_x, v_y).

In case (3), the distance between v_{2^i} and $v_{2^{i+1}}$ is $\delta(v_{2^i}, v_{2^{i+1}}) \leq 2^i$, thus L_i moves horizontally from $v_{2^i} = (x, y)$ through (x', y) at which L_i changes its direction towards $v_{2^{i+1}} = (x', y')$. A path may consist of at least 2^i empty cells L_i goes through to reach the destination cell (x', y') in a worst-case scenario. Recall that L_i already consists of 2^i nodes, which guarantees connectivity all the way until arriving at (x', y'). Once L_i has arrived there, it can safely change its direction to line up with $v_{2^{i+1}}$ and occupy the column

x', while preserving connectivity. Further, any non-empty cells of the path are eventually restored due to the *transparency* of line moves shown in Proposition 4. Finally, the same argument holds for (4) by applying (3) twice. Figure 4.16 shows an example of case (3) and (4). Thus, *Line-Walk* always keeps the whole shape connected during any phase i of the transformation. □



(a) The case when $\delta(v_{2^i}, v_{2^{i+1}}) \leq 2^i$. (b) The case when $\delta(v_{2^i}, v_{2^{i+1}}) > 2^i$.

Figure 4.16: Two cases of *Line-Walk* operation.

Now, we are ready to analyse the running time of HamiltonianToLine transformation.

Lemma 25. *Given an initial Hamiltonian shape $S_I \in \mathcal{H}$ of order n, HamiltonianToLine transforms S_I into a straight line S_L in $O(n \log n)$ moves, while preserving connectivity during the course of the transformation.*

Proof. The bound $O(n)$ trivially holds for case (1) and (2). We then analyse a worst-case of (3) and (4) in which the transformation matches its maximum running time. Let T_i denote the total number of moves from phase 1 up to i for all $0 \leq i \leq \log n$. In phase i, a straight line L_i of length 2^i traverses along a path of at most $2 \cdot (2^i - 2) = 2^{i+1} - 4$ cells in which L_i changes its direction twice paying a cost of at most $2^{i+2} - 4$ moves. There is an additive factor of 2 for a special-case in which L_i turns at a non-empty corner as depicted in Figure 2.9. With that, the operation of *Line-Walk* takes total moves k'_i of at most:

$$k'_i = (2^{i+1} - 4) + (2^{i+2} - 4) + 2 = 6(2^i - 1).$$

By the end of phase i, L_i and L'_i combine together into a straight line L_{i+1} of length 2^{i+1}, in a total cost k''_i of at most:

$$k''_i = 2(2^i - 1),$$

moves. Hence, the operation of *Line-Walk* and combination of L_i and L'_i require at most k_i in phase i given by:

$$\begin{aligned} k_i &= k'_i + k''_i \\ &= 6(2^i - 1) + 2(2^i - 1) = 8(2^i - 1) \\ &= O(2^i). \end{aligned}$$

Now, let T_{i-1} denote a total number of moves for a recursive call of HamiltonianToLine transformation on S_i (of 2^i nodes) to transform it into a straight line L'_i, then the transformation in phase i requires at most:

$$T_i = 2 \cdot T_{i-1} + k_i,$$

moves. Given that the first phase i costs $T_1 = 1$, we compute the recursion as follows:

$$\begin{aligned} T_i &= 2 \cdot T_{i-1} + k_i = 2T_{i-1} + 2^i = 2(2T_{i-2} + 2^{i-1}) + 2^i = \dots \\ &= 2^i \cdot T_1 + i \cdot n = 2^i + i \cdot n \end{aligned}$$

Thus, we claim that in the final phase $i = \log n$, HamiltonianToLine transformation has a total cost $T_{\log n}$ of at most:

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

By induction, the basic case holds trivially. Let us assume it is true for phase i, and we show that this must also be valid for phase $i + 1$:

$$T_{i+1} = 2^{i+1} + (i + 1)n = 2(2^i) + i \cdot n + n$$

As a result, in the final phase $i = \log n$, we conclude that HamiltonianToLine transformation performs a total cost $T_{\log n}$ of at most $O(n \log n)$ moves. □

By Lemmas 24 and 25, HamiltonianToLine transformation can transform any Hamiltonian shape $S \in \mathcal{H}$ into a straight line $S_L \in \mathcal{H}$ of the same order within $O(n \log n)$ moves, while preserving connectivity. By reversibility of line moves (consult Lemma 2), any pair of Hamiltonian shapes $S_I, S_F \in \mathcal{H}$ of the same order can be transformed to each other by first transforming S_I into S_L and then reversing the transformation of S_F into S_L, within the same asymptotic time of $O(n \log n)$ moves. Thus, we have arrived at the following theorem:

Theorem 9. *For any pair of Hamiltonian shapes $S_I, S_F \in \mathcal{H}$ of the same order n, S_I can be transformed into S_F (and S_F into S_I) in $O(n \log n)$ moves, while preserving connectivity of the shape during the course of the transformation.*

4.3 Compression: An $O(n\sqrt{n})$-time universal transformation

This section demonstrates a universal shape transformation that solves the Universal-Connected problem in $O(n\sqrt{n})$ moves. It is called Compress and transforms any pair of connected shapes (S_I, S_F) of the same order to each other, while preserving connectivity during its course. The following is a high-level description of this strategy.

Starting from the initial shape S_I of order n with an associated graph $G(S_I)$, compute a spanning tree T of $G(S_I)$. Then, enclose the shape into an $n \times n$ square box and divide it into $\sqrt{n} \times \sqrt{n}$ square sub-boxes. Each occupied sub-box contains one or more maximal sub-trees of T. Each such sub-tree corresponds to a sub-shape of S_I, which from now on we call a *component*. Pick a leaf sub-tree T_l, let C_l be the component with which it is associated, and B_l their sub-box. Let also B_p be the sub-box adjacent to B_l containing the unique parent sub-tree T_p of T_l. Then compress all nodes of C_l into B_p through line moves, while keeping the nodes of C_p (the component of T_p) within B_p. Once compression is completed and C_p and C_l have been *combined* into a single component C'_p, compute a new sub-tree T'_p spanning $G(C'_p)$. Repeat until the whole shape is compressed into a $\sqrt{n} \times \sqrt{n}$ square. The latter belongs to the family of *Central Line* (or *nice*) shapes (see Definition 10) and can, thus, be transformed into a straight line in linear time.

Given that, the main technical challenges in making this strategy work universally is that a connected shape might have many different configurations inside the sub-boxes it occupies, while the shape needs to remain connected during the transformation. In the following, we describe the *compression* operation, which successfully tackles all of these issues by exploiting the linear-strength of line moves.

4.3.1 Universal transformation by compression approach

Let $C_l \subseteq S_I$ be a leaf component containing nodes $v_1, \ldots, v_k$ inside a sub-box B_l of size $\sqrt{n} \times \sqrt{n}$, where $1 \leq k \leq n$, and $C_p \subseteq S_I$ the unique parent component of C_l occupying an adjacent sub-box B_p. If the direction of connectivity between B_l and B_p is vertical or horizontal (see an example in Figure 4.17), push all lines of C_l one move towards B_p sequentially one after the other, starting from the line furthest from B_p. Repeat the same procedure to first align all lines perpendicularly to the boundary between B_l and B_p (Figure 4.17(d)) and then to transfer them completely into B_p (Figures 4.17(e)). Hence, C_l and C_p are combined into C'_p, and the next round begins.

The above moves are carried out in such a way that all lines (in C_l or C_p) pushed by this operation do not exceed the border of B_p. While C_l compresses vertically or horizontally, it may collide with a component $C_r \subseteq S_I$ inside B_l. In this case, C_l stops compressing and combines with C_r into C'_r, and the next round begins. If C_l compresses diagonally towards C_p (vertically then horizontally or vice versa) via an intermediate adjacent sub-box B_m and collides with $C_m \subseteq S_I$ inside B_m, then C_l completes compression into B_m and combines with C_m into C'_m. Figure 4.18 shows how to compress a leaf component into its parent component occupying a diagonal adjacent sub-box.

Algorithm 2, Compress, provides a universal procedure to transform an initial connected shape S_I of any order into a compressed square shape of the same order. It takes two arguments: S_I and the spanning tree T of the *associated graph* $G(S_I)$. In any round: Pick a leaf sub-tree of T_l corresponding to C_l inside a sub-box B_l. Compress C_l into an adjacent sub-box B_p towards its parent component C_p associated with parent sub-tree T_p. If C_l compressed with no collision, perform combine(C_p, C_l) which combines C_l with C_p into one component C'_p. If C_l collides with another component C_r inside B_l, then perform combine(C_r, C_l) into C'_r. If not, as in the diagonal compression in which C_l collides with C_m in an intermediate sub-box B_m, then C_l compresses completely into B_m and performs combine(C_m, C_l) into C'_m. Once compression is completed, update(T) computes a new sub-tree and removes any cycles. The algorithm terminates when T matches a single component of n nodes compressed into a single sub-box.

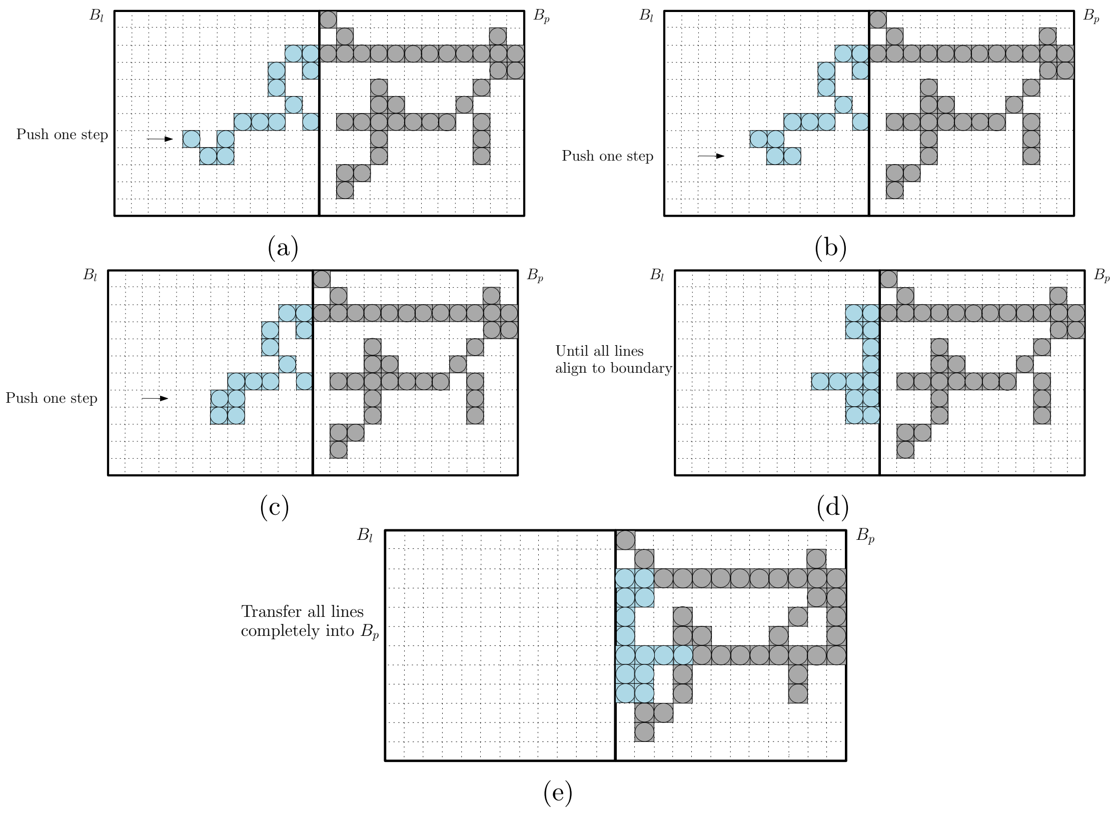


Figure 4.17: A leaf component C_l in blue compressing from the left sub-box B_l towards its parent component C_p in black inside a horizontal adjacent right sub-box B_p. From the top-left, C_l pushes all lines to align them perpendicularly to the boundary between B_l and B_p then compresses them into B_p. All other orientations are symmetric by rotating the shape 90°, 180° or 270° clockwise.

4.3.2 Correctness and runtime analysis

In this section, we prove correctness of Compress transformation. First, we show that it can transform any pair of connected shapes (S_I, S_F) of the same order to each other, while preserving connectivity throughout. We then discuss its running time, which takes a total cost of at most $O(n\sqrt{n})$ moves to compress any connected shape.

Given an initial connected shape S_I holding n nodes enclosed into an $n \times n$ square that is divided into $\sqrt{n} \times \sqrt{n}$ square sub-boxes, we provide the following definitions that are used in the rest of this chapter.

Definition 15 (Connectivity of sub-boxes)**.** *By the above partitioning, two occupied sub-*

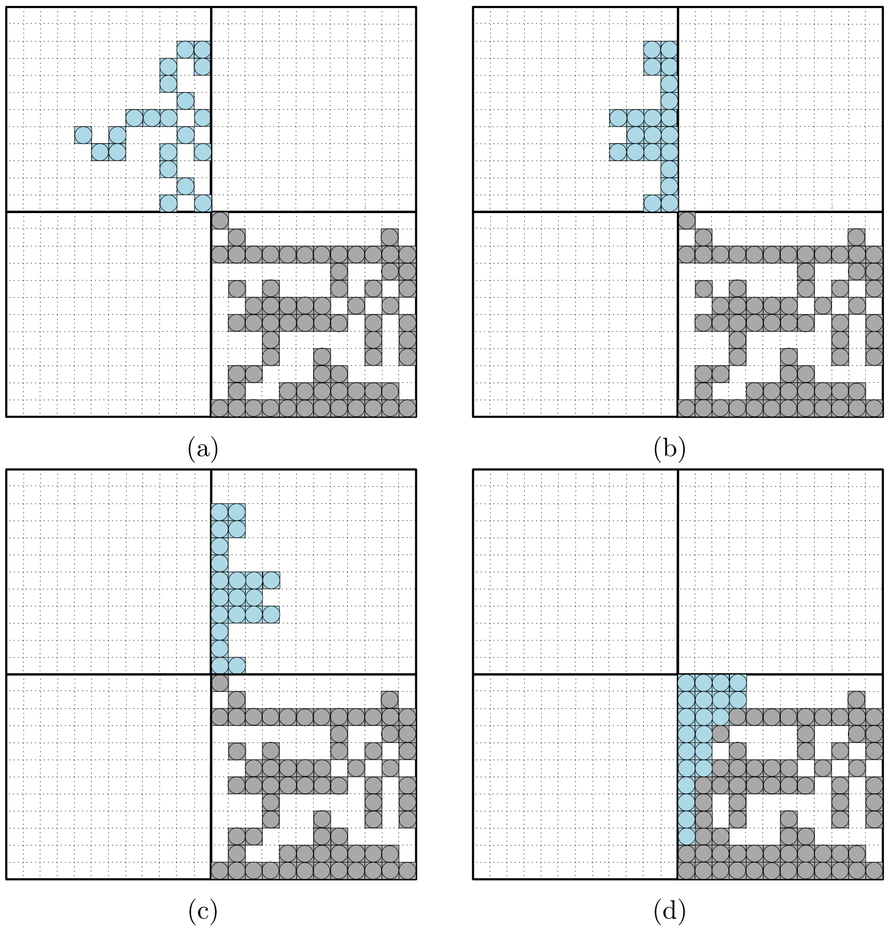


Figure 4.18: A leaf component C_l in blue compressing from the top-left sub-box towards its parent component C_p in black inside a diagonal adjacent bottom-right sub-box. From the top-left, C_l compresses first horizontally towards an intermediate top-right sub-box, then vertically into the bottom-right. All other orientations are symmetric by rotating the shape 90°, 180° or 270° clockwise.

boxes, B_1 and B_2, are connected iff there are two distinct nodes $u, v \in S_I$, such that u occupies B_1 and v occupies B_2 where u and v are two adjacent neighbours connected vertically, horizontally or diagonally.

Definition 16 (Connectivity of components)**.** *By the above partitioning, two connected components, C_1 and C_2, are connected iff there are two distinct elements $u \in C_1$ and $v \in C_2$, such that u and v are two adjacent neighbours connected vertically, horizontally or*

```
Algorithm 2: Compress($S$)
  $S = (u_1, u_2, ..., u_{|S|})$ is a connected shape, $T$ is a spanning tree of $G(S)$
  repeat
      $C_l \leftarrow$ pick($T_l$) // select a leaf component associated with a leaf
          sub-tree
      Compress($C_l$) // start compressing the leaf component
      if $C_l$ collides then
          $C'_r \leftarrow$ combine($C_r, C_l$) or $C'_m \leftarrow$ combine($C_m, C_l$) // as described in
              text
      else
          $C'_p \leftarrow$ combine($C_p, C_l$) // combine $C_l$ with a parent component
      end
      update($T$) // update sub-trees and remove cycles after compression
  until the whole shape is compressed into a $\sqrt{n} \times \sqrt{n}$ square
  Output: a square shape $S_C$
```

diagonally.

Then, we show that each sub-box holds at most $2\sqrt{n}$ components.

Lemma 26. *Any $\sqrt{n} \times \sqrt{n}$ square box can contain at most $2\sqrt{n}$ components.*

Proof. Any component $C_l \subseteq S_I$ inside a sub-box B_l must be connected through a path to one of $\sqrt{n}/2$ non-adjacent cells at a length-$\sqrt{n}$ boundary of B_l, resulting in $2\sqrt{n}$ for the four boundaries, as shown in Figure 4.19. Hence, any a sub-box can contain at most $2\sqrt{n}$ disconnected components. □

Then, the following lemma proves that any connected shape S of n nodes can be compressed into a square box of dimension $\sqrt{n}$.

Lemma 27. *Let S be a connected shape of order n occupies $\sqrt{n}$ sub-boxes of size $\sqrt{n} \times \sqrt{n}$ each. Then, it is always possible to compress all n nodes into a single sub-box.*

Proof. The number of cells inside any sub-box is $\sqrt{n} \times \sqrt{n} = n$, then it is sufficient to be filled by at most n nodes. □

Next, the following lemma shows that Compress transformation eventually forms a *Central Line* (*nice*) shape, which can be transformed fast into a straight line in linear time.

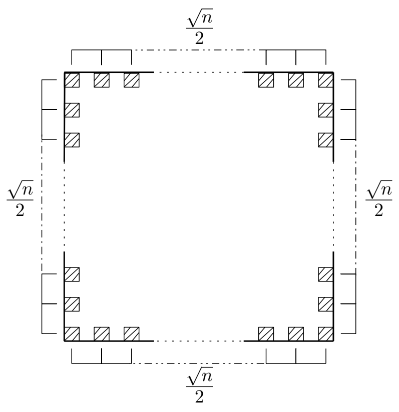


Figure 4.19: A square box of four length-$\sqrt{n}$ boundaries, each of $\sqrt{n}/2$ non-adjacent cells.

Lemma 28. *Starting from an initial connected shape S_I of order n, Compress transformation eventually forms a nice shape $S_{\mathscr{NICE}}$ of order n.*

Proof. Regardless of which sub-box the shape will compress into, the resulting final shape will form a square of size $\sqrt{n}$, which satisfies all conditions of nice shapes in Definition 10. □

Given an initial connected shape S_I of n associated with a graph $G(S_I)$, compute a spanning tree T of $G(S_I)$. Then enclose S_I into an $n \times n$ square box and divide it into $\sqrt{n} \times \sqrt{n}$ square sub-boxes. Each occupied sub-box contains one or more maximal sub-trees of T. Each such sub-tree corresponds to a component. Pick a leaf sub-tree T_l, let C_l be the component with which it is associated, and B_l their sub-box. Let also B_p be the sub-box adjacent to B_l containing the unique parent sub-tree T_p of T_l. We then provide the following lemma:

Lemma 29. *Compress transformation compresses a leaf component $C_l \subseteq S_I$ of $k \geq 1$ nodes, while preserving the global connectivity of the shape.*

Proof. Due to symmetry, we present only one direction as all other directions hold by rotating the shape 90°, 180° and 270° clockwise. Assume B_l is a left sub-box horizontally adjacent to right sub-box B_p. All k horizontal lines (rows) of C_l move sequentially towards B_p, starting from the line furthest from the boundary between B_l and B_p. Given a line $l \in k$ with length i, $1 \leq i \leq \sqrt{n}$. Then, l must be in one of the following cases in order to push one move horizontally from B_l into B_p:

- **Case 1.** Starting from the left to right boundary of B_l, a line l of length $\sqrt{n}$ occupies cells $(x, y), \ldots, (x + \sqrt{n}, y)$, and thus l moves one step to the right, occupying $(x + 1, y), \ldots, (x + \sqrt{n} + 1, y)$ in a such way similar to a simple position permutation of l's nodes to their right neighbour positions. Regardless of the shape configuration, l creates an empty cell at (x, y) while remaining connected to any other nodes occupying cells $(x, y \pm 1), \ldots, (x + \sqrt{n}, y \pm 1)$. See an example of this case in Figure 4.20(a) and (b).

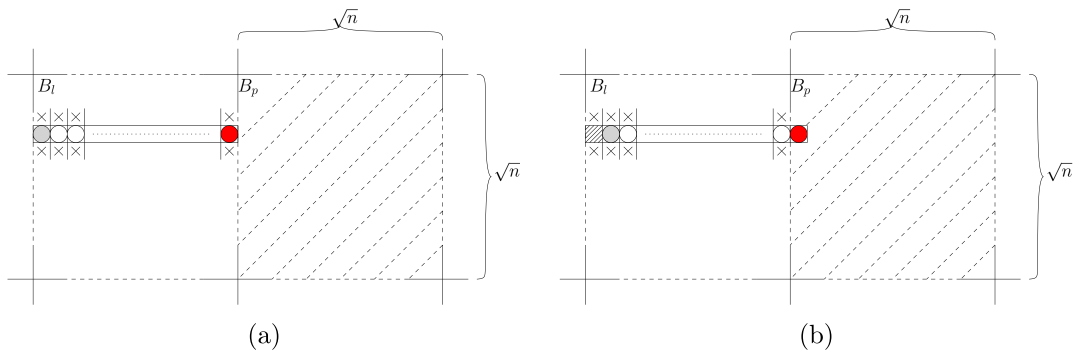


Figure 4.20: A line l of length $\sqrt{n}$ occupies the whole dimension of a sub-box in (a), is pushing one step right in (b).

- **Case 2.** A line l of length smaller than $\sqrt{n}$. It is similar of **Case 1** in which l moves one position to the right. Figure 4.21 shows one example of this case.

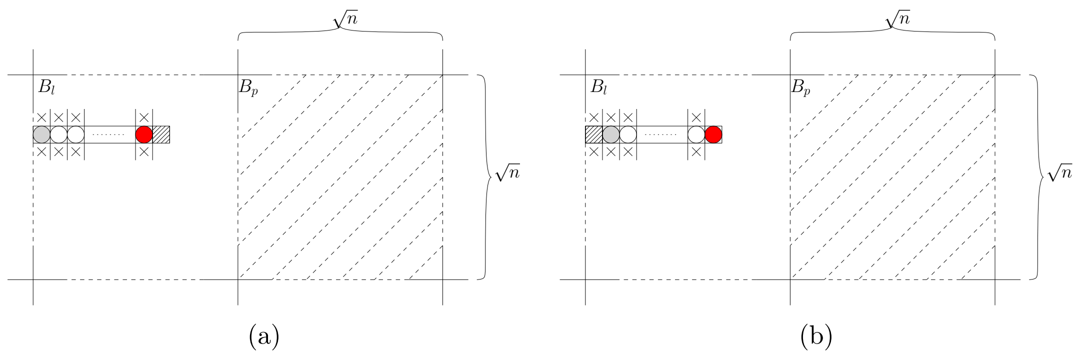


Figure 4.21: A line l of length $i < \sqrt{n}$ in (a), is pushing one step right in (b).

- **Case 3.** Two horizontal lines l and l', both smaller than $\sqrt{n}$ in length, occupy the same row and are separated by one empty cell. Say l starts from the leftmost column x and ends at $x + i$, with an empty cell $(x + i + 1, y)$ and l' occupying $(x + i + 2, y), \ldots, (x + \sqrt{n}, y)$. This is similar to **Case 2** in which l moves one step towards cell $(x + i + 1, y)$, then both lines merge into one, creating a new empty cell at (x, y), as shown in Figure 4.22.

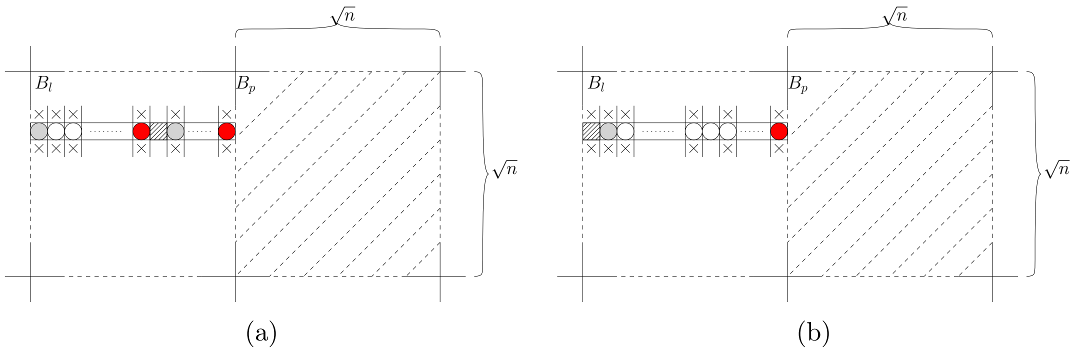


Figure 4.22: (a) Two lines occupying a row are separated by one empty cell inside B_l, both of length less than $\sqrt{n}$. The left line moves one step to the right, then both combined into a single line in (b).

As mentioned earlier, when a leaf component C_l occupying B_l compresses towards its parent C_p occupying B_p, we now demonstrate that no line crosses the border of B_p.

- **Case 4.** A line l of length $i < \sqrt{n}$, beginning at the left boundary and terminating at cell $x + i$ of B_p, is adjacent to an empty cell to the right at $(x + i + 1, y)$. Once l is pushed one move to the right, it fills in the empty cell and occupies positions $(x + 1, y), \ldots, (x + i + 1, y)$ with length $i + 1$. This is similar of Case 2 in Figure 4.21.

- **Case 5.** A line l of length $\sqrt{n}$ starts from the left to right boundary of B_p. Once l is pushed one move towards the right, it turns to fill in empty cells at the right boundary of B_p, starting from the rightmost column to the left. Figures 4.23 and 4.24 depicts two different examples of filling the boundary of B_p.

In all of the above cases, the horizontal line l pushes one move towards the right while maintaining the whole connectivity of the configuration. As an immediate observation: whenever a line $l \subset S_I$ inside a sub-box B_l pushes one move towards the boundary between (B_l, B_p), the global connectivity of the whole shape is preserved. This holds also for all k

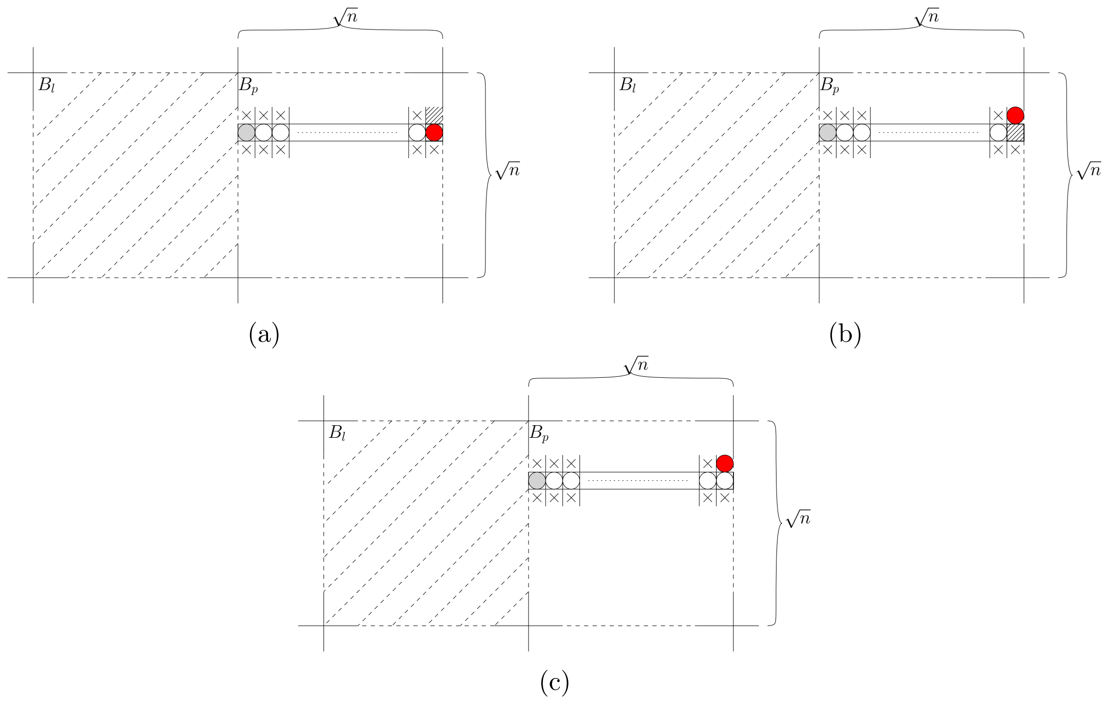


Figure 4.23: (a) A line l of length $\sqrt{n}$ takes up a whole dimension of a parent sub-box B_p with an empty cell above its right end node. As l pushes to the right, the end node moves up towards that empty cell (b), and l moves one step to right in (c).

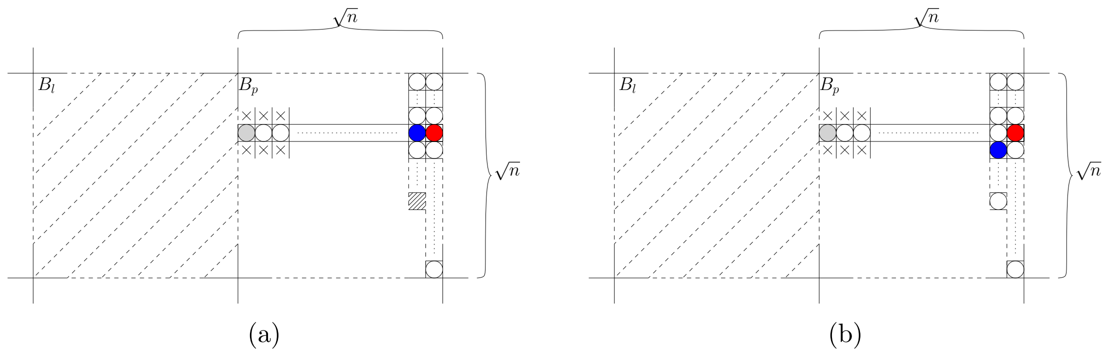


Figure 4.24: (a) A line l of length $\sqrt{n}$ occupies the entire dimension of B_p, with rightmost column completely filled. l fills up the nearest empty cell in the second column from the right in this case (b).

lines that move one step from B_l to B_p, sequentially one after another at any order, starting from the furthest-to-nearest line from B_p. Thus, this must hold for any finite number of line moves required to compress a leaf component C_l towards its parent C_p. □

Let us now analyse the running time of this transformation. In some cases, the compression cost can be as little as one move or as high as linear movements. For that, we divide the overall cost into charging phases to facilitate the analysis. We then manage to upper bound the cost of each charging phase independently of the sequential order of compression. Below we provide a rough upper bound for all possible shape configurations. We first show the total cost required to compress a leaf component C_l inside B_l into a parent C_p occupying an adjacent sub-box B_p.

Lemma 30. *Given a pair of components C_l and C_p of k_l and k_p nodes, $1 \leq k_l + k_p \leq n$, occupying two adjacent $\sqrt{n} \times \sqrt{n}$ sub-boxes B_l and B_p, receptively. Then, C_l compresses into C_p inside B_p within at most $O(n)$ moves while preserving connectivity during compression.*

Proof. Assume a worst-case configuration when B_l and B_p are connected diagonally. In this case, the compression needs to go through an intermediate sub-box B_m first then carry on towards B_p. Each line $k_l \in C_l$ moves at most distance $\sqrt{n}$ to cross the boundary between B_l and B_p; hence, all k_l lines need to pay at most n moves to completely transfer into B_m and another n moves to compress into B_p. Further, we give an extra $2n$ moves for filling in a boundary at B_p (as depicted in Figure 4.24). Thus, the transformation pays at most t moves to compress C_l into C_p,

$$\begin{aligned} t &= n + n + 2n = 4n \\ &= O(n). \end{aligned}$$

□

Given some partitionings, a family of connected shapes can be separated into n connected components deployed on $\sqrt{n}$ sub-boxes. For a variety of reasons, this family looks to approach a worst-case complexity where the Compress transformation achieves its maximum cost. To begin, such partitioning separates a shape from this family into as many components as possible up to n. Furthermore, the shape's diameter, defined as the distance between its two furthest nodes, has a maximum length of n. Unlike other densely connected shapes with smaller diameters, the compression cost of these shapes can be rather high due to the lack of long lines, which necessitates extra effort for individuals and short lines. Thus, we provide this observation which closely follows Corollary 1.

Observation 1. *There is a set of connected shapes denoted $\mathcal{C}$, such that an instance $S \in \mathcal{C}$ can have n components occupying the $\sqrt{n}$ sub-boxes.*

Figures 4.25 and 4.26 illustrate partitioning instances of two shapes from $\mathcal{C}$. Since $S \in \mathcal{C}$ is connected, each occupied sub-box has at most $O(\sqrt{n})$ connected components of size 1. Apart from the sequential sequence of the transformation, our goal now is to upper bound the cost that any connected shape S pays at maximum to compress based on the number of occupied sub-boxes.

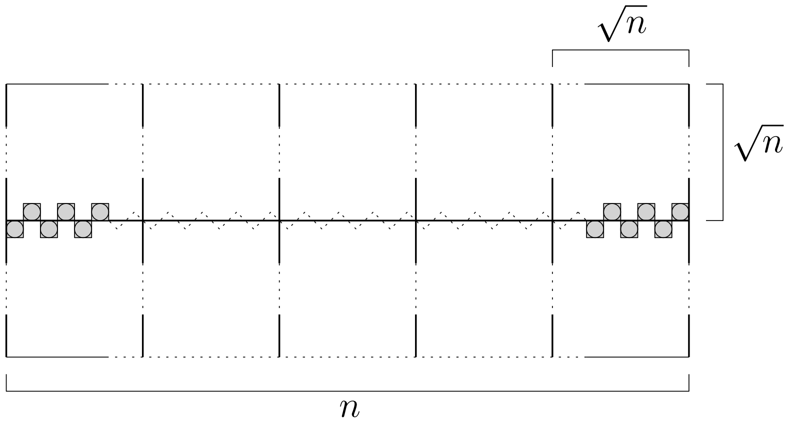


Figure 4.25: A shape of zigzag line with a partitioning positioned to cross the middle through every two nodes.

Lemma 31. *The* Compress *transformation compresses any connected shape S of order n into a $\sqrt{n} \times \sqrt{n}$ square shape in $O(n\sqrt{n})$ moves while preserving connectivity during its course.*

Proof. Let us analyse the compression cost of these shapes based a worst-case scenario. To simplify the analysis, we divide the total cost T into $\sqrt{n}$ charging phases $t_1, \ldots t_{\sqrt{n}}$, each corresponds to an occupied sub-box. Then we upper bound the cost in each phase independently of the compression order. In any charging phase t_i, for all $1 \leq i \leq \sqrt{n}$, the strategy compresses at most $O(\sqrt{n})$ lines distance of $O(\sqrt{n})$, incurring a cost of n moves, while preserving connectivity. In the worst case of Lemma 30, the compression may traverse through diagonal sub-boxes, incurring at most $2n$ moves and an additional cost of $2n$ moves for boundary rearrangements. Thus, the cost phase t_i is bounded by:

$$t_i = 4n,$$

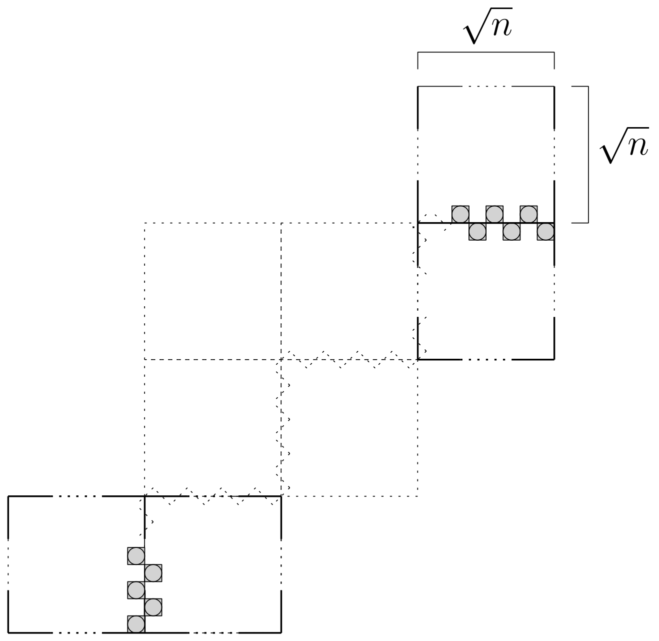


Figure 4.26: A diagonal zigzag line with a partitioning positioned to cross the middle through every two nodes.

moves, which is mostly enough for an occupied sub-box to be cleared of its lines. Then, the cost of all $\sqrt{n}$ charging phases is sufficient to empty all occupied sub-boxes of all lines inside them over the transformation, for a maximum total cost T:

$$\begin{aligned} T &= 4n \cdot \sqrt{n} = 4(n\sqrt{n}) \\ &\leq O(n\sqrt{n}), \end{aligned}$$

moves. □

By Lemmas 29 and 31, any connected shape S of order n can be transformed into a *Central Line* (*nice*) shape, denoted $S_{\mathscr{NICE}}$, within $O(n\sqrt{n})$ moves while preserving connectivity. By reversibility, any pair of connected shapes S_I and S_F of the same order can be transformed to each other within the same asymptotic time of $O(n\sqrt{n})$ moves by first transforming S_I into $S_{\mathscr{NICE}}$ and then reversing the transformation from S_F into $S_{\mathscr{NICE}}$. Finally, the following theorem implies that:

Theorem 10. *Compress solves* UNIVERSALCONNECTED *within $O(n\sqrt{n})$ moves.*

Chapter 5

Distributed transformations

In the previous chapters, we studied and revealed the underlying transformation principles of line moves – including modelling, feasibility and complexity – as well as presenting several algorithmic solutions that are best suited for each given design and task, coupled with provable guarantees on their performance. All of those transformations, however, are centralised and cannot be directly applicable to real distributed robotic systems. Thus, our current goal is to provide distributed transformations that, if possible, retain 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 during their course.

In this chapter, we establish an algorithmic framework that allows individuals to utilise their linear-strength mechanism in a distributed manner. We develop the first distributed connectivity-preserving transformation that exploits line moves and matches the bound of the best-known centralised transformations. In Section 5.1, we define a discrete system of n simple indistinguishable devices, called *agents*, forming a connected shape S_I on a two-dimensional square grid. 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. For example, there is no deterministic algorithm to elect a leader for a set of agents sharing an identical local view. In [123], 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 [74, 111]. 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 [50]). Thus, as a first attempt at distributing line moves on the two-dimensional square grid, we adopted a full-synchronised model of agents that share a sense of orientation which will be discussed in more detail later.

Recall our $O(n \log n)$-move connectivity-preserving centralised strategy presented in Section 4.2 that solves the HamiltonianLine problem. Section 5.2, similarly, presents a solution to the line formation problem, that is, for any initial Hamiltonian shape S_I, form a final straight line S_L of the same order. It is quite common for newly proposed models in distributed systems to commence with the basic shape formation problems, e.g. the line, as a first stepping stone towards more general transformations, see for example the nubot model [121] and the amoebot model [54]. This is confirmed by the fact that if every shape can be transformed into a special (specific) intermediate shape (in our case a straight line), any pair of shapes (with the same number of agents) can be transformed to each other.

In Section 5.2, we present the transformation in depth, including all of its algorithmic aspects and technical tools. We assume that a pre-processing phase provides the Hamiltonian path, that is, a global sense of direction is made available to the agents 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 [102] in which the transformation relies on an initial central phase to gain some information about the number of entities in the system.

On a high-level, the algorithm transforms a Hamiltonian shape into a straight line in $\log n$ phases (each consisting of six sub-phases) as follows: In phase i, 2^{i-1} terminal agents on the path forming a straight line merge with the next 2^{i-1} agents of any configuration, in order to form a new straight line of length 2^i; hence, the line length is doubled in each phase. On one end of the Hamiltonian path, a pre-elected leader (by a central control) leads this process. By transmitting and swapping states along the Hamiltonian path in Sections 5.2.1 and 5.2.2, agents that participate in forming the respective line, are identified. Afterwards a feasible transformation path is computed in Section 5.2.3. This is done by exploiting the agent's knowledge about their respective successor in the Hamiltonian path, and thus their respective local *Hamiltonian direction*. In Section 5.2.4, the agents thereby act as a distributed binary counter. Once having that computed path, the leader pushes the line from the prior phase (of length 2^{i-1}) to its destination. This line, in Sections 5.2.5 and 5.2.6, is then merged recursively with the active agents in that phase.

5.1 The distributed model of line moves

Below, we define our distributed transformation model that make use of line moves. Although some settings were discussed before in Chapter 2, we need to define them again here for consistency and to put everything in place, as a new separate model, and to avoid any confusion with the centralised 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 $\mathit{cell}(p) = (x, y)$, where x indicates columns and y represents rows. 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).

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 5.1. 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 $p \neq p'$.

Each agent is equipped with the linear-strength mechanism defined in Section 2.2. Recall that, given a line L consisting of a sequence of k agents occupying consecutive cells on the grid, say without loss of generality $L = (x, y), (x + 1, y), \ldots, (x + k - 1, y)$, where $1 \leq k \leq n$. Then, 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 (i.e. all of the k agents is

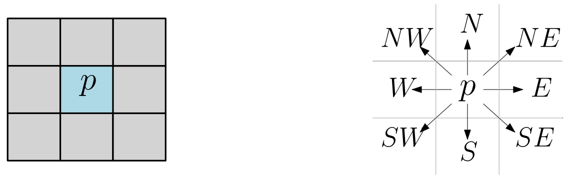


Figure 5.1: An agent p is a neighbour to any agent locating at one of the eight surrounding cells in grey.

pushed one step right at the same time, while only the pushing agent is aware of this move and the other $k-1$ agent 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. This linear-strength pushing mechanism can be equipped internally to an agent or in a system of external forces that occur naturally (e.g. gravity) or artificially (e.g. magnetic surface). We call the number of agents in S the *size* or *order* of the shape.

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 neither they have any other 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 Look-Compute-Move (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 $S(C')$.

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 output alphabet corresponding to the set of moves, a transition function $\delta : Q \times X \to Q \times M$ setting a new state and action by receiving an input and being in a particular state and the output function $O : \delta \times X \to M$ specifying the produced action for a specified input.

5.2 The distributed Hamiltonian transformation

In this section, we develop a distributed algorithm that uses line movements to form a straight line S_L from an initial connected shape S_I that is associated with a graph that has a Hamiltonian path. As we will argue, this strategy performs $O(n \log n)$ moves, that is, it is as efficient with respect to moves as the best-known centralised transformation, and completes within $O(n^2 \log n)$ rounds, while keeping the whole shape connected during its course. As these are the initial attempts to distribute line moves, let us start with some prerequisites:

- 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. That is, each agent p_i knows the local ports incident to its predecessor p_{i-1} and its successor p_{i+1}, for all $1 \leq i \leq n$.
- The head p_1 and the tail p_n, respectively, are where P begins and finishes. They are also responsible for pushing the line on an interchangeable basis during the transformation.
- 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.

In order to simplify the exposition, we assume that n is a power of 2; this can be easily dropped later. The transformation proceeds in $\log n$ phases, each of which contains six sub-phases (or sub-routines). Each sub-phase consists of one or more synchronous rounds. The transformation begins with a simple line of length 1, then gradually flattens all agents along P while doubling its length, until it reaches the final straight line S_L of length n. The high-level description of this strategy is provided below.

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, 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 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 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. For simplicity, we shall only mention the affected components of the state of the agents.

By the beginning of phase i, $0 \leq i \leq \log n - 1$, there exists a terminal straight line L_i of 2^i active agents occupying a single row or column on the grid, starting with a head labelled l_h and ending at a tail labelled l_t, while internal agents have label l. All agents in the rest of the configuration are inactive and labelled k. During phase i, the head l_h leads the execution of six 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. Go to (6) if perpendicular or start phase $i + 1$ otherwise.

(3) DrawMap: Compute a route map that takes L_i 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.

Figure 5.2 gives an illustration of a phase of this transformation when applied on the diagonal line shape. 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 line defined by S_i is in line or perpendicular to L_i. 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 the check returns true, then l_h starts a

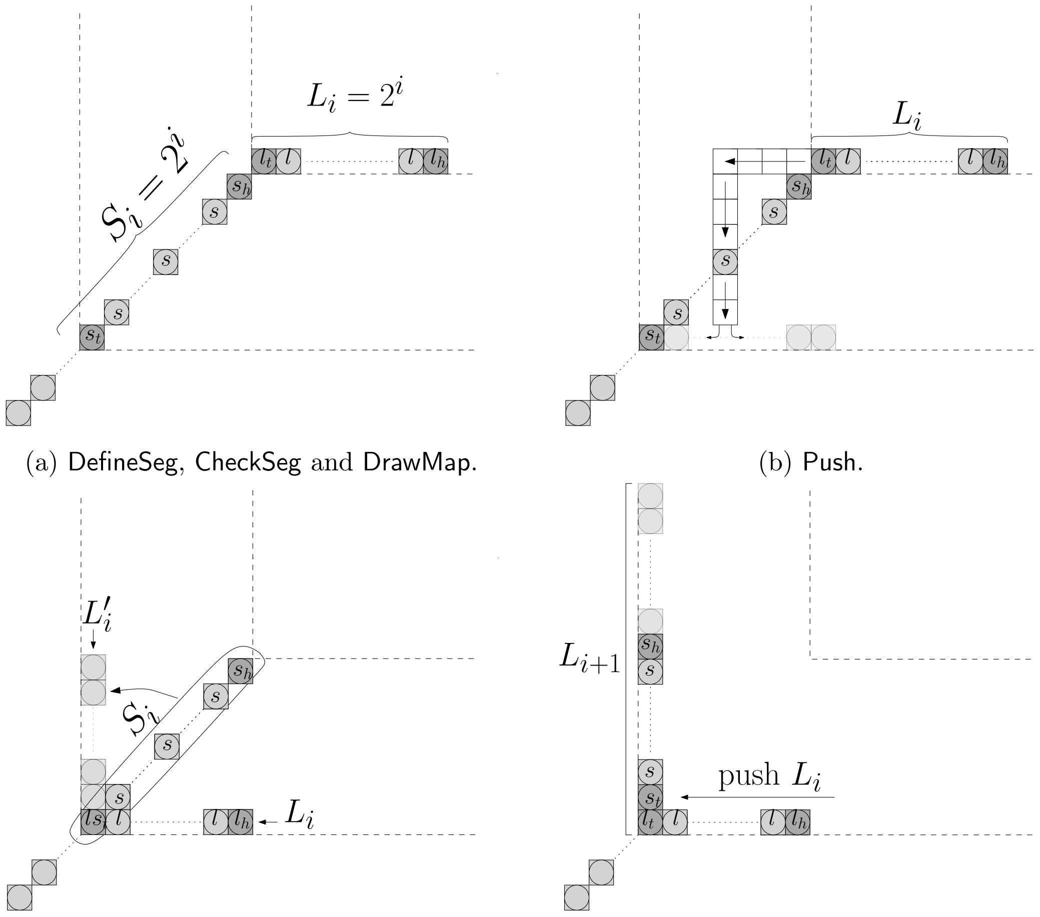


Figure 5.2: Similar to Figure 4.15, a snapshot of phase i of the Hamiltonian transformation on the shape of a diagonal line . Each occupied cell shows the current label state of an agent. Light grey cells show ending cells of the corresponding moves.

new round $i + 1$ and calls Merge to combine L_i and S_i into a new line L_{i+1} of length 2^{i+1}. Otherwise, l_h proceeds with the next sub-phase, DrawMap.

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.

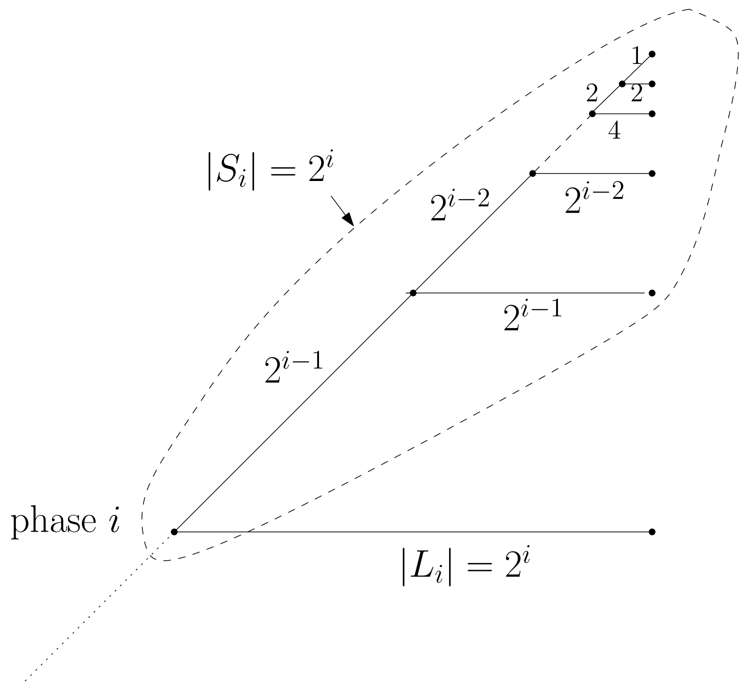


Figure 5.3: A zoomed-in picture of the core recursive technique RecursiveCall shown in Figure 5.2(c).

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 5.3 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.

5.2.1 Defining the next segment S_i

Given a terminal straight line L_i on the Hamiltonian path P, this sub-phase identifies the next segment S_i and activates its 2^i agents. The algorithm works as follows: The line head l_h transmits a special mark "$\textcircled{H}$" to go through all active agents in P. It updates its transmission component c_2 as follows: $\delta(l_h, \cdot, \cdot, a \in A, \cdot, \cdot, \cdot) = (l_h, \textcircled{H}, \cdot, a \in A, \cdot, \cdot, \cdot)$. 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 $\delta(k, \textcircled{H}, \cdot, a \in A, \cdot, \cdot, \cdot) = (s_h, \cdot, \cdot, a \in A, \cdot, \cdot, \cdot)$. Similarly, once a line agent p_i passes "$\textcircled{H}$" to p_{i+1}, it also initiates and propagates its own mark "$\textcircled{I}$" to activate a corresponding segment agent s. The line tail l_t emits "$\textcircled{T}$" to activate the segment tail s_t, which in turn bounces off a special end mark "$\otimes$" 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 "$\otimes$" mark is propagated back to the head l_h along the active agents, by always moving from p_{i+1} to p_i.

Lemma 32. *DefineSeg correctly activates all agents of S_i in $O(n)$ rounds.*

Proof. When an active agent p_i with label inline l or tail l_t observes the head mark "$\textcircled{H}$" on the state of its predecessor p_{i-1}, it then updates transmission state c_2 to "$\textcircled{H}$" and initiates a special mark on its waiting state c_4. This can be either inline "$\textcircled{L}$" or tail "$\textcircled{T}$" mark. Once an inactive agent notices predecessor with "$\textcircled{L}$" or "$\textcircled{T}$" 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 "$\otimes$" 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 "$\otimes$" in transmission state, so when agent p_i observes successor p_{i+1} showing "$\otimes$", 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 "$\otimes$" 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 odd length, unless the initial line of one agent labelled l_h and 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 "$\otimes$" back to l_h. □

Figure 5.4 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 3, excluding all that have no effect.

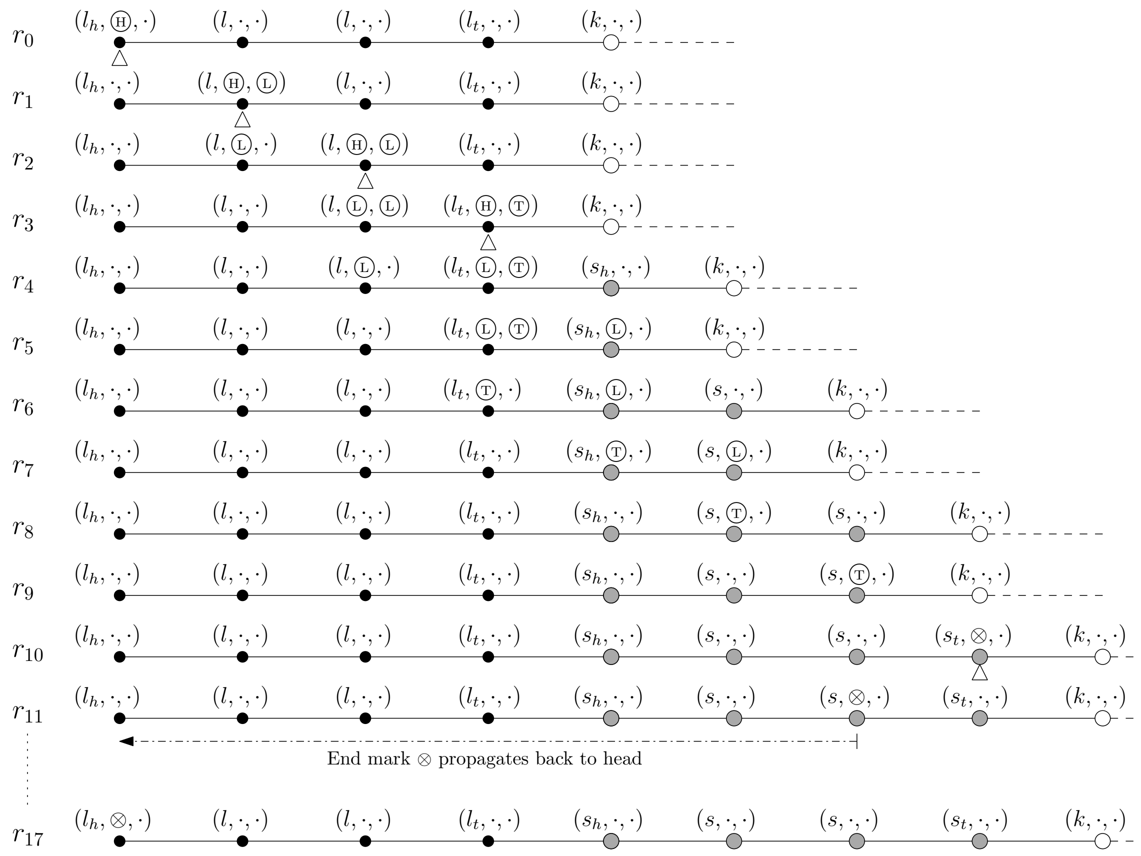


Figure 5.4: An implantation of DefineSeg on a line L_i of four agents depicted as black dots. Each agent uses only its 3 state components (c_1, c_2, c_3), where c_1 is the label state, c_2 the transmission state and c_3 the waiting mark state. In round $r = 0$, L_i is labelled correctly, starting from 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 sees predecessor presenting a mark, it activates (grey dots) 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 33. *DefineSeg requires at most $O(n)$ rounds to define S_i.*

Proof. The head mark "Ⓗ" shall traverse all agents of the line L_i of length $|L_i|$ until it arrives at the first inactive agent, taking $O(|L_i|)$ rounds. Thus, all other agents on the line propagate marks that take $O(|L_i|)$ parallel rounds to reach their corresponding agents on

Algorithm 3: 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 5.4 topmost

$\textcircled{H} \leftarrow p_1.c_2$ // head sets a mark in transmission state
repeat
    // each agent acts based on its current label state
    Head $l_h$:
    if $(p_1.c_2 = \textcircled{H})$ then $\cdot \leftarrow p_i.c_2$ // reset transmission state
    if $(p_{i+1}.c_2 = \otimes)$ then $\otimes \leftarrow p_1.c_2$ // observe end mark; end this sub-phase

    Active:
    if $(p_{i-1}.c_2 = \textcircled{H})$ // observe predecessor with head mark
     then
        $\textcircled{H} \leftarrow p_i.c_2$
        if (inline $p_i.c_1 = l$) then $\textcircled{L} \leftarrow p_i.c_3$
        if (tail $p_i.c_1 = l_t$) then $\textcircled{T} \leftarrow p_i.c_3$
    end
    if $p_{i-1}.c_2 = \textcircled{L}$ then $\textcircled{L} \leftarrow p_i.c_2$ // predecessor shows inline mark
    if $p_{i-1}.c_2 = \textcircled{T}$ then $\textcircled{T} \leftarrow p_i.c_2$ // predecessor shows tail mark
    if $\big((p_i.c_2 = \textcircled{H} \vee \textcircled{L} \vee \textcircled{T}) \wedge p_{i-1}.c_2 = \cdot\big)$ then
    $p_i.c_2 \leftarrow p_i.c_3$ // transmit marks
    $\cdot \leftarrow p_i.c_3$ // rest marks
    if $p_{i+1}.c_2 = \otimes$ then $\otimes \leftarrow p_i.c_2$ // successor shows end mark
    if $p_i.c_2 = \otimes$ then $\cdot \leftarrow p_i.c_2$ // rest transmission state

    Inactive:
    if $(p_{i-1}.c_2 = \textcircled{H})$ then $s_h \leftarrow p_i.c_1$ // activate to segment head $s_h$
    if $(p_{i-1}.c_2 = \textcircled{L})$ then $s \leftarrow p_i.c_1$ // activate to insegment $s$
    if $(p_{i-1}.c_2 = \textcircled{T})$ then
    $s_t \leftarrow p_i.c_1$ // activate to segment tail $s_t$
    $\otimes \leftarrow p_i.c_2$ // initiate end mark
until $(p_1.c_2 = \otimes)$
CheckSeg
```

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 $n/2$. □

5.2.2 Checking the next segment S_i

This sub-phase checks the geometrical configuration of the new 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. First, when S_i is in line with L_i (as illustrated in Figure 5.4), both S_i and L_i already form a single straight line L_{i+1} of double length, so the next phase $i+1$ begins. This reduces the cost of DrawMap, Push, RecursiveCall and Merge. Second, if S_i is forming a line perpendicular to L_i (see Figure 5.5(a)), L_i only needs to reverse direction and line up with S_i to generate L_{i+1}, bypassing the extra cost of DrawMap and proceeding directly to Push. Lastly, S_i has a single turn (looks like L-shape) (see Figure 5.5(b)), where it can simply turn at its corner and align with L_i, create L_{i+1} and save the cost of DrawMap, Push and RecursiveCall. A high-level explanation is provided below.

When l_h observes "$\otimes$", 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: (1) starts the next sub-phase DrawMap if it observes "$\neg$", (2) calls Merge to combine the two perpendicular lines if it observes aa' or (3) begins a new phase $i+1$ if it receives back its local direction a. Algorithm 4 shows the pseudocode of this sub-routine.

Lemma 34. *CheckSeg correctly checks the configuration of S_i to be one of the following:*

- *S_i is in line with L_i.*
- *S_i forms a straight line perpendicular to L_i.*
- *S_i forms an L-shape.*
- *S_i contains more than one turn.*

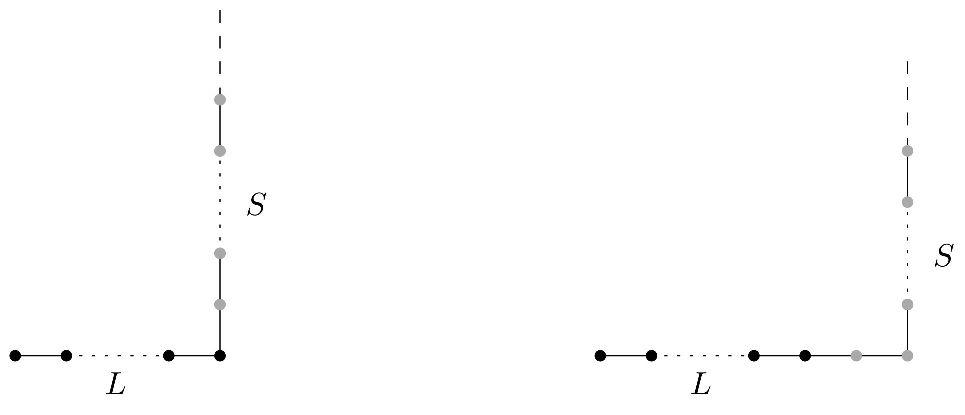
Proof. This sub-routine starts as soon as the head l_h observes the end mark "$\otimes$" of Define-Seg, which means that all agents of the segment S_i are active and labelled correctly. Given that, the input configuration of CheckSeg is a Hamiltonian path terminates at straight

```
Algorithm 4: 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 5.4 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 $\cdot \leftarrow p_i.c_2$ // reset transmission state
      if $(p_{i+1}.c_2 = \neg)$ then $\neg \leftarrow p_1.c_2$ // 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 $c_4 \leftarrow p_i.c_2$ // observe same direction
      if $p_{i-1}.c_2 \neq c_4)$ then $c_4\mathsf{L} \leftarrow p_i.c_2$ // show a turn
      if $(p_{i-1}.c_2 = c_4\mathsf{L})$ then $\neg \leftarrow p_i.c_2$ // 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 $\checkmark \leftarrow p_{|2L|.c2}$// $s_i$ transmits mark backwards
      if $(p_{|2L|-1}.c_2 = c_4\mathsf{L})$ then $\mathsf{L} \leftarrow p_{|2L|.c2}$ // $s_i$ transmits mark backwards
      if $(p_{i-1}.c_2 \neq \cdot)$ then $\cdot \leftarrow p_i.c_2$ // reset transmission state
  until $p_1.c_2 = \neg$
  DrawMap
```

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 ports 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 "$\rightarrow$", then l_h performs this state transition: $\delta(l_h, \otimes, \cdot, \rightarrow) = (l_h, \rightarrow, \cdot, \rightarrow)$. This arrow "$\rightarrow$" 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 "$\cdot$". If "$\rightarrow$" matches a local direction stored on c_4 of p_i, then p_i transmits the same arrow "$\rightarrow$" from



(a) L_i is perpendicularly to S_i. (b) S_i has a single turn.

Figure 5.5: 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.

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} showing "→L" , p_j initiates a negative mark $c_2 \leftarrow \neg$, which is relayed back to l_h, calling out for DrawMap. Once s_t observes "→L", it bounces off the mark "L" back towards l_h to start Push. Otherwise, s_t propagates a special check-mark "✓" backwards, alerting l_h that both L_i and S_i already form a straight line. □

Now, we provide analysis of this procedure.

Lemma 35. *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. □

5.2.3 Drawing a route map

This local technique creates a map and calculates the shortest route of minimum turns, with the goal of achieving the lowest cost. On the square grid, the most efficient way to accomplish this is to draw a route of a single turn, such as L-shaped routs. For the purpose of connectivity preservation, it can be demonstrated that there exist some worst-case routes in which the line L_i may disconnect while travelling towards the tail of S_i. Essentially, this can be seen in a route where the Manhattan distance between the the line tail l_t and the segment tail s_t is $\Delta(l_t, s_t) \geq |L_i|$, for additional information, see Section 4.2.

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 17). Below, we give more details of DrawMap.

Definition 17 (A route). *A route is a rectangular path R consisting of 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 $1 \leq i \leq |R| - 1$. Let C be a system configuration, C_R denotes the configuration of R where $C_R \subset C$ defined by $[c_1, \ldots, c_{|R|}]$.*

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 $k = 2^i$ agents can be utilised as a distributed binary counter (similar to a Turing machine tape) which is capable to store up to $2^k - 1$ unsigned values. This k-bit binary counter supports increment which is the only operation we need in this procedure. 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 "$\oplus$"). See an implantation of this counter in Figure 5.6.

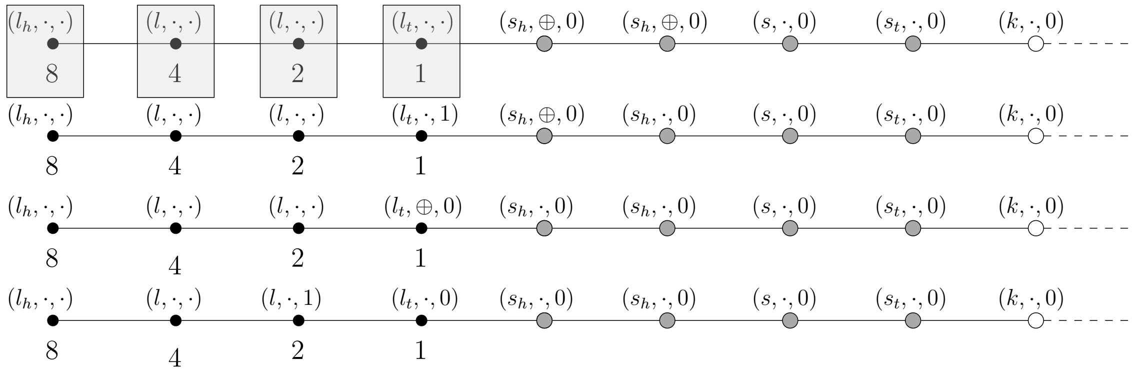


Figure 5.6: A 4-bit line counter L_i. Agents of L_i and S_i are depicted by black grey dots, respectively. The state of an agent is (c_1, c_2, c_5) denoting c_1 the label, c_2 transmission and c_5 counter components, omitting others with no effect. Each shaded area shows a corresponding decimal number. Top: the counter has a decimal value of 0. 2nd: an increment of 1. 3rd: the line tail 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.

ComputeDistance procedure

Initially, the head l_h emits a special mark "Ⓒ" to all active agents, asking them to commence the calculation of the Manhattan distance $\Delta(l_t, s_t)$ between the line tail l_t and the segment tail s_t. 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 5 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. The following lemma shows that this procedure calculates $\Delta(l_t, s_t)$ in linear time.

Lemma 36. *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 $p_{i+1}.c_2 \leftarrow \oplus$, (2) if $p_i.c_2 \leftarrow \oplus$ if

Algorithm 5: ComputeDistance(L_i, S_i)

```
$S = (p_1, \dots, p_{|S|})$ is a Hamiltonian shape
Initial configuration: a straight line $L_i$ and a segment $S_i$ labelled as in Figure 5.6

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 $l_h$
// Manhattan distance $\Delta \leq i$
8a. When $l_h$ sees ①, it calls CollectArrows to draw one L-shaped route
// Manhattan distance $\Delta > i$
8b. Otherwise, $l_h$ sees ② and calls CollectArrows to draw two L-shaped route
```

$p_{i+1}.c_2 \leftarrow \cdot$ and $p_i.c_3 \leftarrow \oplus$, (3) the head of segment s_h sets $p_i.c_2 \leftarrow \oplus'$ if $p_{i+1}.c_2 \leftarrow \cdot$ and $p_i.c_3 \leftarrow \oplus$ and (4) $p_i.c_2 \leftarrow \cdot$ if $p_{i+1}.c_2 \leftarrow \cdot$ and $p_i.c_3 \leftarrow \cdot$.

Correspondingly, the line agents (of labels l_h, l and l_t) behave as a binary counter described above and illustrated in Figure 5.6. 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 theses transitions in round r_{j-1}: (1) $p_i.c_5 \leftarrow 1$ if $p_i.c_5 \leftarrow \cdot$ or $p_i.c_5 \leftarrow 0$ or (2) $p_i.c_5 \leftarrow 0$ and $p_i.c_2 \leftarrow \oplus$ if $p_i.c_5 \leftarrow 1$. 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 mark "②" is sent back to the head l_h, which finally sees either '①" or "②" and acts appropriately (calls CollectArrows procedure). The counter size is sufficient to calculate $\Delta(l_t, s_t)$ since the worst-case distance is $|L_i| - 2$.

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 $t_3 = O(|L_i|)$. Altogether, the total running time is bounded by $t = t_1+t_2+t_3 = O(|L_i|)$ parallel rounds. □

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 "$\leftrightarrows$" through all active agents until it reaches the segment tail s_t. When "$\leftrightarrows$" 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 "ⓜ1" is transmitted every round and a slow mark moves three rounds slower "ⓜ2". The fast mark "ⓜ1" bounces off s_t, where both "ⓜ1" and "ⓜ2" 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 temporally 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 6 presents the pseudocode that briefly formulates this procedure.

The following lemma proves the correctness and analysis of CollectArrows.

Lemma 37. *The CollectArrows procedure completes within $O(|L_i|)$ rounds.*

Proof. Given an initial configuration defined in Lemma 36. Assume the Manhattan distance $\delta(l_t, s_t) \leq |L_i|$. For simplicity, we prove (A) in algorithm 6 showing only affected states. Once l_h observes ①, it emits a collection mark "$\leftrightarrows$", which then transfers forwardly among active agents until it reaches s_t, similar to counting mark transmission described previously in Lemma 36. When s_t detects "$\leftrightarrows$", 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, $d \in \{\rightarrow, \leftarrow, \downarrow, \uparrow, \nwarrow, \nearrow, \swarrow, \searrow\}$.

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

In what follows, we distinguish between cardinal $\{\rightarrow, \leftarrow, \downarrow, \uparrow\}$ and diagonal directions $\{\nwarrow, \nearrow, \swarrow, \searrow\}$. Figure 5.7 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 $c_4 \leftarrow d^\star$, 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 6). 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 d^+ encounters an opposite arrow d^- recorded in transmission state $p_i.c_2$, both are erased and replaced by the hash sign "#". If d^+ and d^+ are similar, both take priority in c_2. If d^+ observes a perpendicular arrow $\perp d$, d^+ is placed in c_2 and $\perp d$ in waiting state c_3.

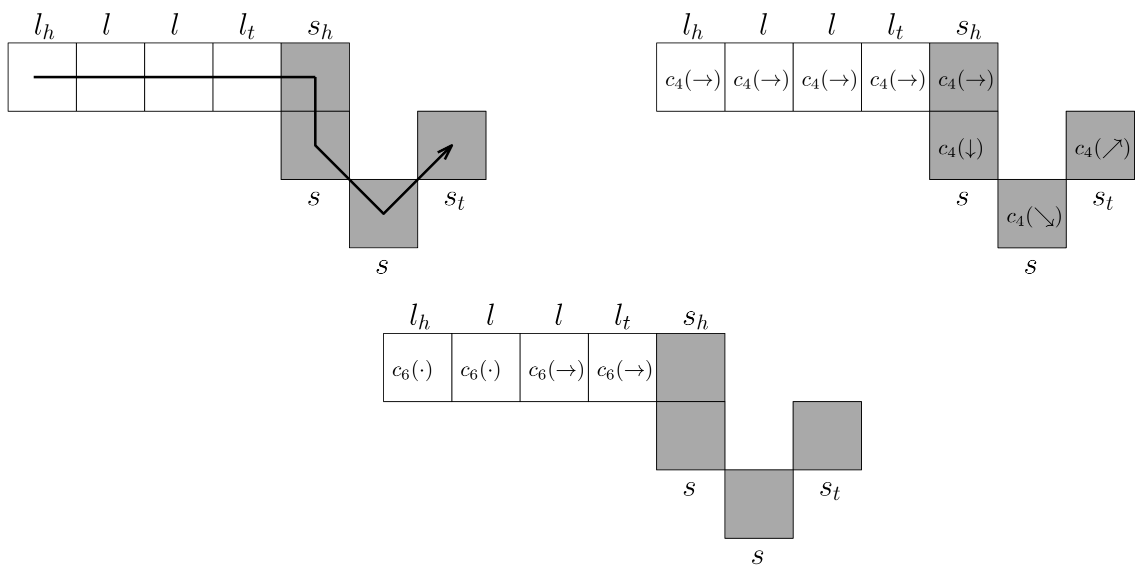


Figure 5.7: Drawing a map: from top-left a path across occupied cells and corresponding local arrows stored on state c_4 in top-tight, where the diagonal directions, "$\searrow$" and "$\nearrow$", are interpreted locally as, "$\downarrow\rightarrow$" and "$\uparrow\rightarrow$". The bottom shows a route map drawn locally on state c_6 of each line agent.

Figure 5.8 depicts a configuration of S_i consisting of eight agents, which their arrows are collected in Figure 5.9 where we represent L_i (white nodes) and S_i (grey nodes) as a tab for better visibility. In the topmost shape, local directions c_4 are inside nodes, label c_1 and transmission c_2 above nodes, waiting c_3 (only for segment agents) and map state c_6 (only for line agents) below nodes. 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 5.9 to make this sub phase easier to understand.

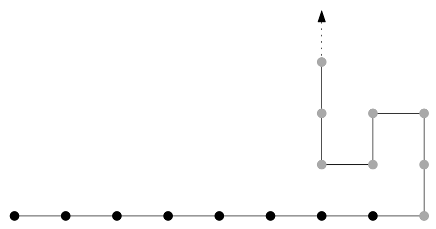


Figure 5.8: A configuration of L_i (black dots) and S_i (grey dots).

Given a segment agent p_i of label s and 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 d^1_{i+1}) and $c_3 = d_i$ are similar, 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 rests $p_i.c_2 \leftarrow \cdot$, 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 rests $p_i.c_2 \leftarrow \cdot$, 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 6 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 "$\textcircled{\scriptsize m1}$" and slow "$\textcircled{\scriptsize m2}$". The fast wave "$\textcircled{\scriptsize m1}$" moves from p_i to p_{i+1} every round, while the slow wave "$\textcircled{\scriptsize m2}$" passes every three rounds. In this way, the fast wave "$\textcircled{\scriptsize m1}$" bounces off s_t and meets "$\textcircled{\scriptsize m2}$" at a middle agent p'_i of S_i which updates label to s'_t, and $p_{i+1'}$ changes label

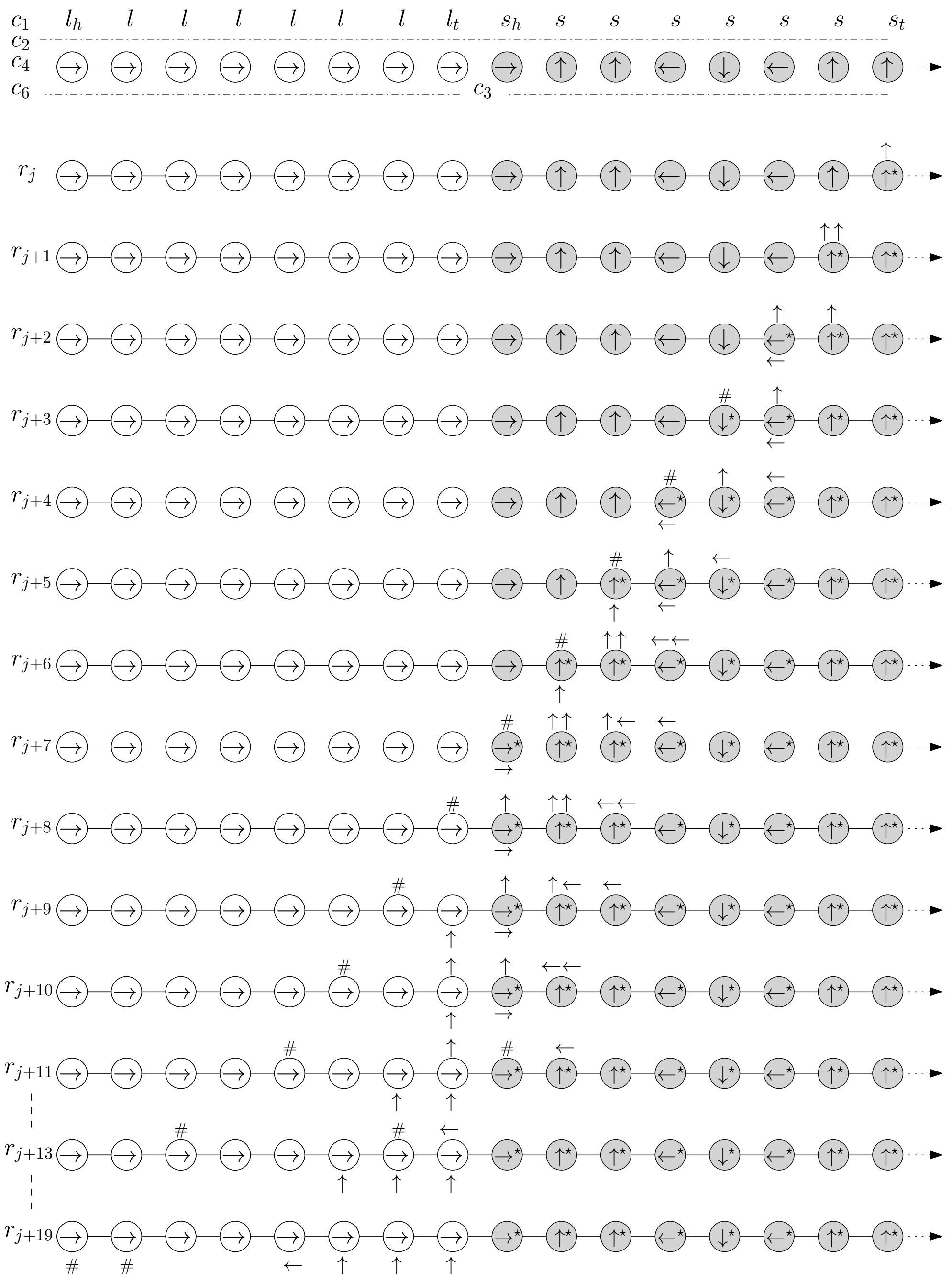


Figure 5.9: An implementation of the arrows collection on the shape in Figure 5.8, see text for explanation.

to s'_h as well. See a demonstration in Figure 5.10. Consequently, S_i is temporarily divided into two halves S_i^1 and S_i^2 labelled:

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

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_i^1 and 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 CollectArrows(L_i, S_i) procedure on a number of rounds. Starting from steps 1 and 2 of (A), the "⇆" 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 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. □

Finally, ComputeDistance and CollectArrows procedures completes the DrawMap sub-phase. By Lemmas 36 and 37, we conclude that:

Lemma 38. *DrawMap draws a map within $O(|L_i|)$ rounds.*

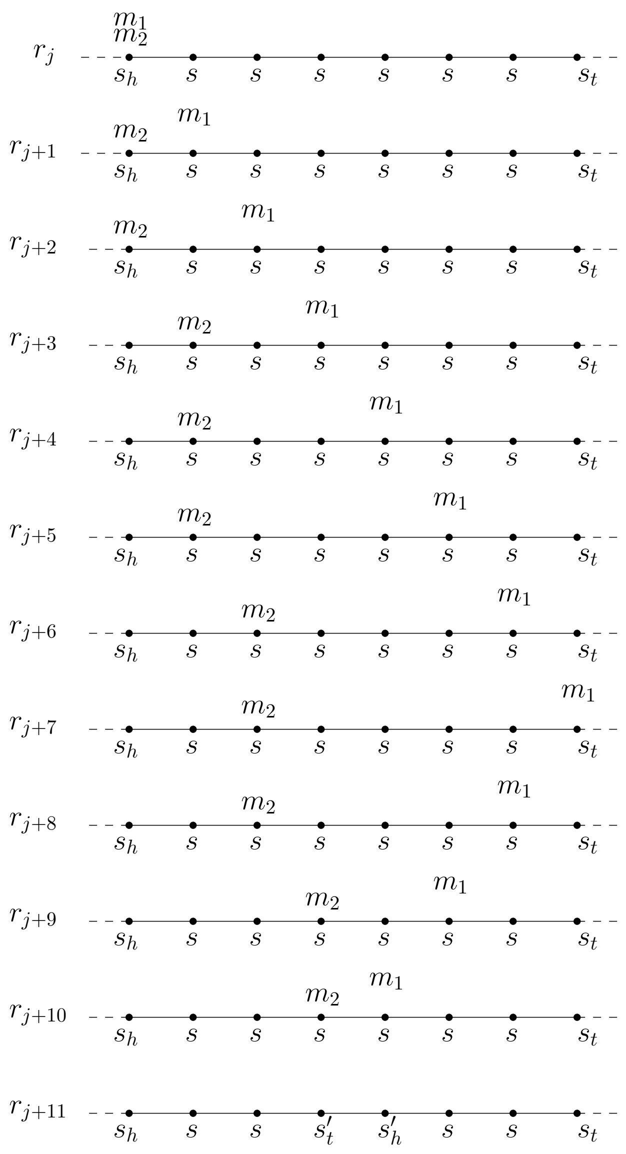


Figure 5.10: A fast “$\textcircled{m1}$” and slow “$\textcircled{m2}$” wave meeting at the middle of S_i of 8 agents. Observe that “$\textcircled{m1}$” moves every round, while “$\textcircled{m2}$” is three rounds slower.

5.2.4 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 obstacles must be overcome in order to translate the global coordinator of line moves into a system of homogeneous 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 “$\textcircled{p1}$” and slow “$\textcircled{p2}$” marks along the line, “$\textcircled{p1}$” is transmitted every round and “$\textcircled{p2}$” is three rounds slower (shown early in DrawMap). The “$\textcircled{p1}$” mark reflects at l_t and meets “$\textcircled{p2}$” at a middle agent p_i, which in turn propagates two pushing signals “$\textcircled{P}$” 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 “$\textcircled{P}$” reaches both ends of the line at the same time. 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 5.15 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 7 provides a general procedure of Push.

Algorithm 7: 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 [92]. The following lemma demonstrates how their solution can be translated to our model in order to coincide a Hamiltonian path of n agents in such a way that they can perform concurrent actions in linear time.

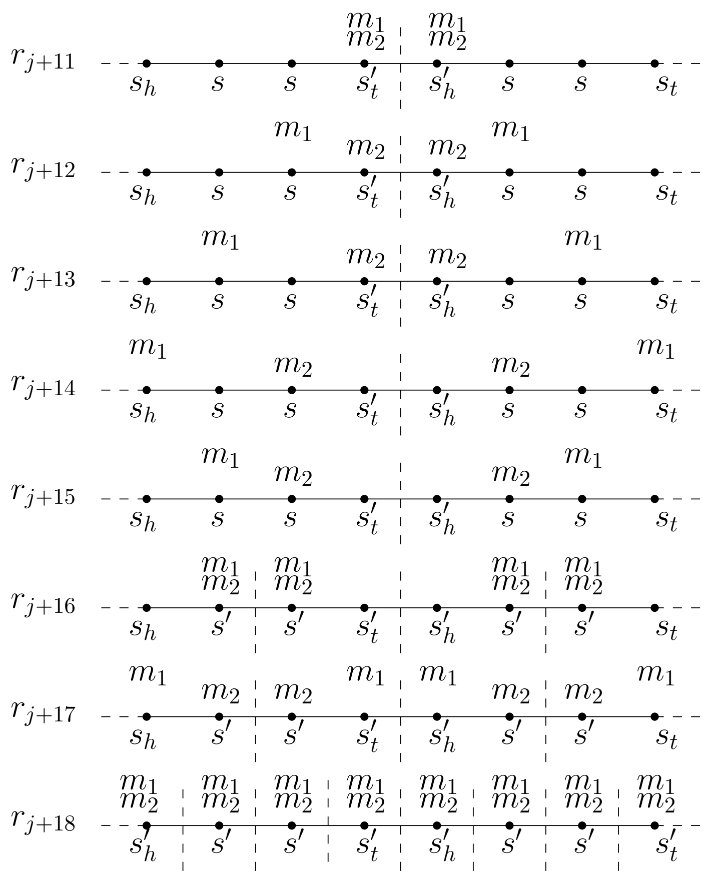


Figure 5.11: Synchronising 8 agents that were started in Figure 5.10 where the halving procedure repeats until all agents reach a synchronised state.

Lemma 39 (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. From [92], the strategy consists of two cases, even and odd number of agents. First, the head p_1 emits fast mark “$\textcircled{m1}$” and slow mark “$\textcircled{m2}$” towards the tail p_n. The “$\textcircled{m1}$” 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 “$\textcircled{m1}$” 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 5.10). 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 5.11.

Assume a path P of n odd agents in which p_1 emits, "$\textcircled{p1}$" and "$\textcircled{p2}$" along P. In this case, the two 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 "$\textcircled{m2}$" and successor p_{i+1} showing "$\textcircled{m1}$" in transmission state and responds by switching into another special state that allows it to 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 5.12 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}) rests 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}) rests 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 $\sum_{i=1}^{n} 3n/2^i = 3n/2 + 3n/4 + \ldots + 0 = 3n(1/2 + 1/4 + \ldots + 0) = 3n(1) = 3n$. Therefore, this synchronisation requires at most $O(n)$ rounds of communication. □

Now, we show that under this model, a number of consecutive agents forming a straight line L_i can traverse transparently through a route R of cells on the grid of any configuration C_R using their local knowledge, without breaking connectivity.

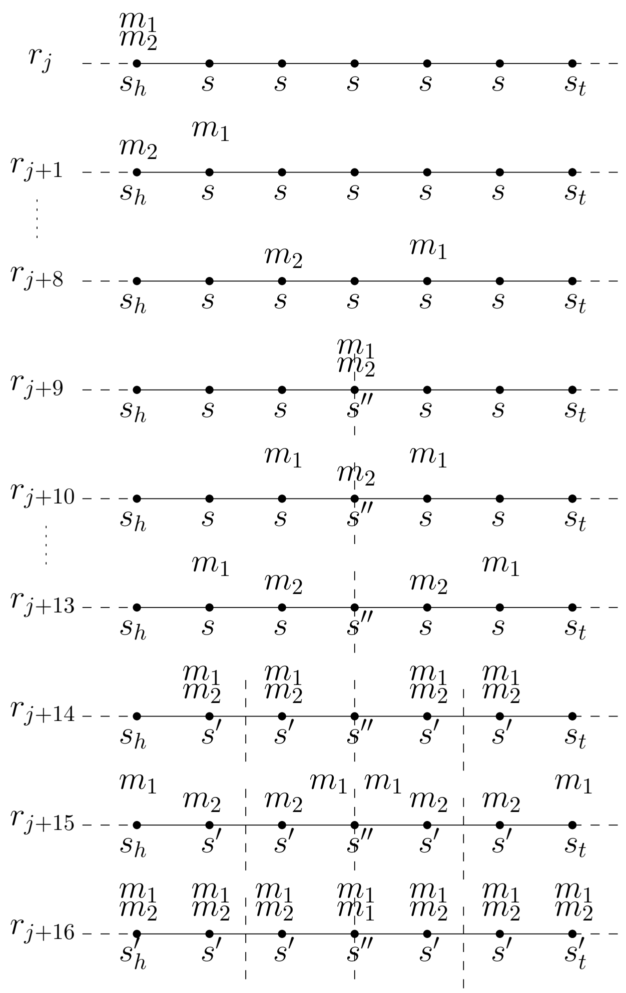


Figure 5.12: An example of synchronising 7 agents - odd case.

Lemma 40. *Let L_i denote a terminal straight line and R be a rectangular path 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 7, the line head l_h observes the collection mark "$\leftrightarrows\checkmark$" indicating the completion of $\mathsf{DrawMap}(S)$, which draws a route R (see Definition 17). 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 so, it emits a special mark "Ⓨ" back to l_h indicating that a route is free to push. By an application of Lemma 39, l_h synchronises all line agents to reach a concurrent state in which the following actions occur concurrently: (1) l_h pushes L_i one position towards l_t, based on its local direction on push state c_7. (2) l_t pushes one position based on its map arrow c_6 either in line direction or perpendicular to L_i. In the latter, l_t updates state to $c_4 \leftarrow c_6$ and tells predecessor to turn next pushing. 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 rests 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 rest their c_6. Eventually, l_h stops pushing once it meets and swaps states with s_t. An example is shown in Figure 5.13.

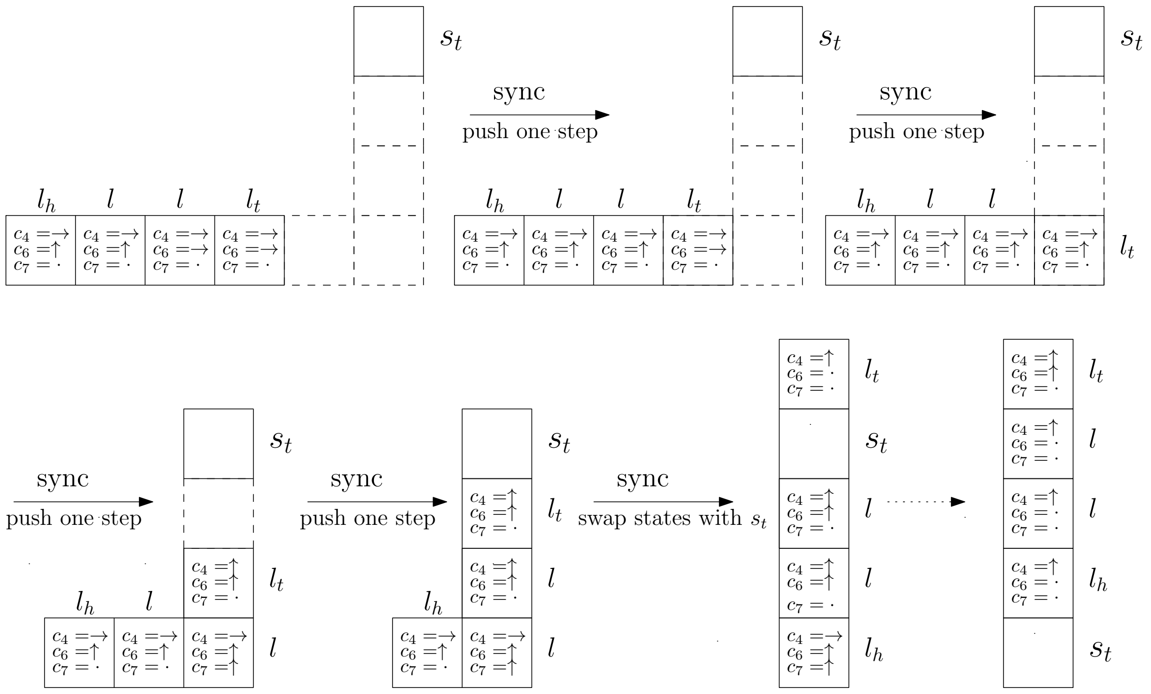


Figure 5.13: A line L_i of four agents pushing through a route of empty cells towards s_t. All affected states (c_4, c_5 and c_7) are shown inside each occupied cell.

During pushing through an L-shape route R, L_i may turn one or at most three times. In the following, we 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 5.13 where L_i will temporally divide into two perpendicular sub-lines while traversing to s_t. By a careful application of Lemma 39, 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 temporally transforms into three perpendicular sub-lines. Three agents simultaneously drive everyone to advance one step ahead on R. Therefore, the line can be synchronised to perform at most three parallel pushing operations that are asymptotically equivalent to the cost of one pushing, without breaking connectivity. Below are transitions that demonstrate how L_i pushes along R while satisfying all of the transparency properties of line moves in Proposition 4:

– 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.

– No effect: L_i restores all occupied cell to their original state and keeps C_R unchanged after traversing R.

– No break: L_i preserves connectivity while traversing along R.

Now, assume L_i walks over a route R of non-empty cells occupied by other agents (denoted by k) in the configuration that are not on S_i. Whenever L_i walks through R and l_t meets k on R, l_t tells l_h to stop pushing. The agent k now updates to a temporary state labelled k_{l_t} if it has a similar arrow of l_t or $k_{l_t}c$ if it has a turn. Based on the map arrow of l_t, k_{l_t} acts as a tail and checks whether the next cell (x, y) on R is empty, which accordingly triggers to one of the following states : (1) (x, y) is empty, then k_{l_t} emits a mark back to l_h to sync and push L_i one step further. During this, k_{l_t} changes state to k_l and each synchronised agent p_i shifts map arrow to p_{i+1}. During pushing, k_l swaps states with its predecessor and ensures that it remains in the same position (see Figure 5.14) until it meets l_h, at which k_l can update to its original state k. (2) (x, y) is occupied by another agent labelled k, then (a) k_{l_t} changes to $k^\star_{l_t}$ (or $k^\star_{l_t}c$ if the map arrow is a turn) and k into k_{l_t}, and (b) k_{l_t} emits a special mark to the line agents asking for the next map arrow $d_{k_{l_t}}$. Repeat this process as long as $d_{k_{l_t}}$ indicates an occupied cell. Once k_{l_t} observes an empty cell (x', y'), it performs (1) and updates $k^\star_{l_t}$. See a demonstration in Figure 5.15.

When L_i moves through a series of non-empty cells, it guarantees that they are neither separated or 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 ($k^\star_l$ or $k^\star_l c$),

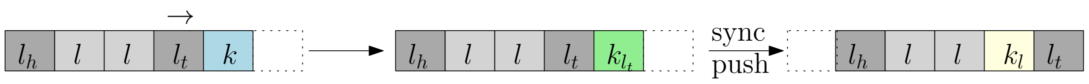


Figure 5.14: A line L_i of agents within grey cells pushing through a non-empty cell in blue with a right map direction above l_t.

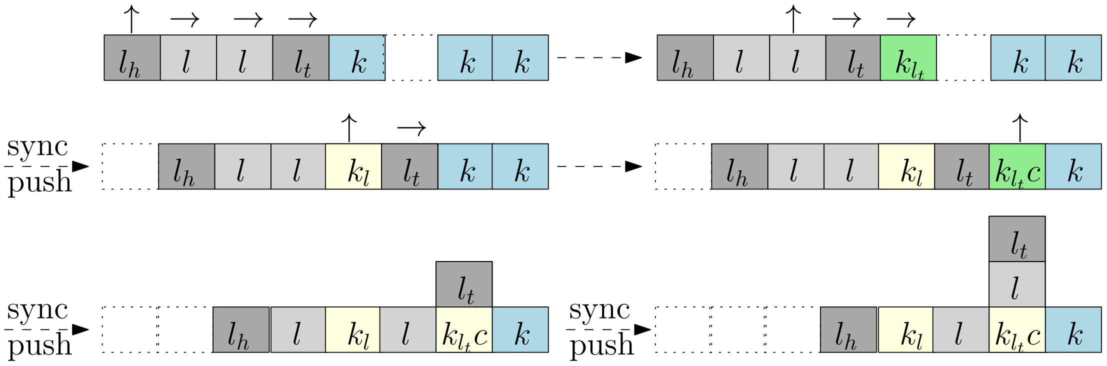

sync push
sync push
sync push
l_h l l l_t k k k
l_h l l l_t k_{l_t} k k
l_h l l k_l l_t k k
l_h l l k_l l_t $k_{l_t}c$ k
l_h l k_l l l_t $k_{l_t}c$ k
l_h k_l l $k_{l_t}c$ l l_t k

Figure 5.15: A line L_i of agents inside grey cells, with map directions above, pushing and turning through empty and non-empty cells in blue (of label k). The green and yellow cells show state swapping.

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 5.16 shows this case.

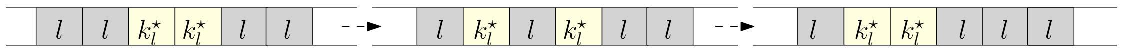

l l $k_l^\star$ $k_l^\star$ l l
l $k_l^\star$ l $k_l^\star$ l l
l $k_l^\star$ $k_l^\star$ l l l

Figure 5.16: Four agents in a line inside grey cells swap states with others occupying consecutive yellow cells.

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). However, there exists a case that may break the connectivity. Consider a line L_i pushing along R and turning at a corner agent labelled $k_{l_t}c$, which has two diagonal neighbours where both are not adjacent to any line agent, as depicted in Figure 5.17 top. In this case, when $k_{l_t}c$ 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 $k_{l_t}c$ 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 $k_{l_t}c$ to return to its initial cell. Figure 5.17 depicts how to handle this situation. Other orientations follow symmetrically by rotating the system $90°, 180°$ or $270°$ clockwise and counter-clockwise.

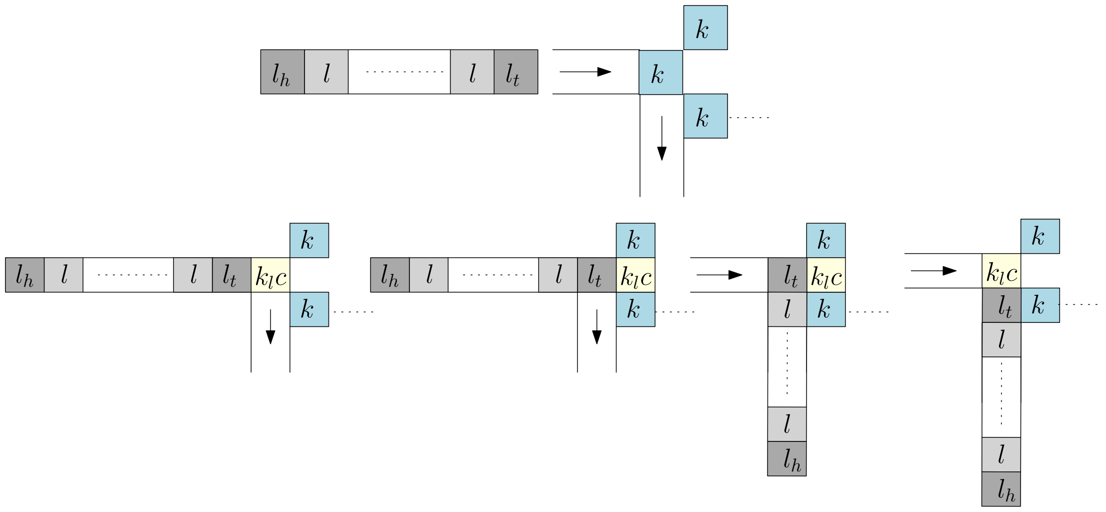


Figure 5.17: A line L_i pushing through a route R turns at a corner agent labelled $k_{l_t}c$ that has two diagonal neighbours, neither of which is adjacent to any line agent.

Thus, all agents of L_i are labelled and organised in such a way that can transparently push through a route R of any configuration C_R, whether it is being empty or partially/fully occupied. It implies that L_i remains connected when travelling as well as the whole configuration. Further, the original state of C_R has been restored and all of its occupied cells (if any) have been left unchanged. As a result, L_i meets all of the transparency criteria of line moves in Proposition 4. $\square$

The complexity of Push is provided in the following lemma based on the number of line moves and communication rounds.

Lemma 41. *A straight line L_i traverses through a route R of any configuration C_R, taking at most $O(|L_i|)$ line moves within $O(|L_i| \cdot |R|)$ rounds.*

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 40) to push through R. On the other hand, the communication cost of this sub-phase could be very high in the case of a fully occupied route R when individuals 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. In this case, we set an upper bound on the most dominating work.

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. □

5.2.5 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 5.3 presents a graphical illustration of RecursiveCall applied on a diagonal line shape.

5.2.6 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. Currently, the tail of L'_i occupies a cell adjacent to the head of L_i. Hence, l_h can simply check if L'_i is in line or perpendicular to L_i exploiting the previous procedure of CheckSeg. Without loss of generality, say that the tail agent of L'_i occupies cell (x, y) and L_i occupies cells $(x+1, y), \ldots, (x+|L_i|-1, y)$. Then, L'_i could be either (1) perpendicular with agents occupying $(x, y), \ldots, (x, y+|L_i|-1)$ or (2) in line on cells $(x, y), \ldots, (x-|L_i|-1, y)$. In (1), l_h emits a mark that travels via agents of L'_i until it reaches the other head, where it asks to change the direction of L'_i, allowing L'_i and L_i to combine into a single straight line L_{i+1} of double length and designate 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 switch and update labels to assign a head l_h, tail l_t and $2^{i+1}-2$ line agents l in between.

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 within a distance equal to its length in order to line up with L_i. It is worth noting that the agents of L'_i do not require full synchronisation for each push. Instead, they simply need to sync the head and tail of L'_i where both perform pushing at the same time. When an agent $p_i \in L'_i$ turns, it tells its predecessor $p_{i-1} \in L'_i$ to turn too. Hence, the total number of moves is at most $O(|L'_i|)$. The communication cost splits into: (1) A special mark from l_h traverses across L'_i in $O(|L'_i|)$ rounds. (2) All agents of L'_i synchronise in $O(|L'_i|)$ rounds. (3) Label swapping costs at most $|L'_i| + |L_i| = O(|L'_i|)$. Therefore, all agents in Merge communicate in linear time, and then we can say:

Lemma 42. *An execution of Merge requires at most $O(|L_i|)$ line moves and $O(|L_i|)$ 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 Section 4.2. Let T_i^c and T_i^m denote the total number of communication rounds and moves in phase i, respectively, for all $i \in 1, \ldots, \log n$. 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:

$$t_i^c = (4 \cdot |L_i|) + (|L_i| \cdot |R|) \approx O(|L_i| \cdot |L_i|).$$

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 achieving the connectivity-preserving condition.

Lemma 43. *Given an initial Hamiltonian shape S_I of n agents, this strategy transforms S_I into a straight line S_L of the same order in $O(n \log n)$ line moves and $O(n^2 \log n)$ rounds, while preserving connectivity during transformation.*

Thus, we can finally provide the following theorem:

Theorem 11. *The above distributed transformation solves* HamiltonianLine *and takes at most $O(n \log_2 n)$ line moves and $O(n^2 \log_2 n)$ rounds.*

5.3 Further discussion

Overall, this distributed approach solves a number of the shape formation problems defined in this thesis, including the DiagonalToLine and DiagonalToLineConnected problems. Thus, there is still a chance to distribute other centralised transformations of line moves, hopefully within the same asymptotic bound of $O(n \log n)$ line moves and $O(n^2 \log n)$ rounds. The HamiltonianConnected problem, for example, is an immediate candidate which can transform any pairs of Hamiltonian shapes to each other while preserving connectivity throughout the transformation. If achieved, it is expected to contribute to the development of more general transformations, such as the UniversalConnected problem. However, those appear to be a much more complicated problem to solve based on the current distributed setting. For example, in contrast to the centralised scenario, reversibility does not apply in a straightforward way because the agents need to somehow know an encoding of the shape to be constructed. Hence, it may be essential to make some changes to the model in order to develop distributed counterparts.

Chapter 6

Conclusions

In this chapter, we summarise technical contributions of the thesis highlighting algorithmic challenges for the shape formation problems. We also show an example and explore how this work may be applied to other fields of theoretical computer science. Furthermore, we present our view on the future research directions opening new perspectives for further development in the area of algorithmic programmable matter.

6.1 Final discussion

Throughout this thesis, we introduced and investigated a new linear-strength transformation of line moves that was used for the problem of shape formation. In this new paradigm, individuals can travel in parallel on a two-dimensional grid system by translating a line of any length one position vertically or horizontally in a single time-step. Our main goal is to examine the power of this mechanism and demonstrate what is theoretically feasible, with the goal of designing universal transformations that can convert any pair of connected shapes to each other. We were able to demonstrate some interesting facts about line moves and produce more efficient and general transformations that work on both centralised and distributed systems by utilising our linear-strength model. Below is a summary of our findings, organised chronologically by chapter.

In Chapter 2, we assembled all of the definitions and basic facts that are required for our formation algorithms. We started by introducing several families of discreet shapes, followed by a formal definition of our centralised algorithmic framework. Next, a number of key aspects of line moves were illustrated in order to establish effective technical tools that are used in the proposed transformations. For instance, it can be shown that this model can simulate the individual rotation and sliding movements, that is, it generalises the models of [64, 89] and also adopts all their transformability results (including universal transformations). The remainder of this chapter is devoted to a consideration of lower bounds, with a focus on some restricted variants of our line move creation problem where we established the first lower bounds for this model that are matching the best known $O(n \log n)$ upper bounds.

Chapter 3 examines whether pushing lines could assist to achieve a significant performance boost (compared to the $\Theta(n^2)$-time of individual moves). Even though it can be immediately observed that there are instances in which this is the case (e.g., initial shapes in which there are many long lines, thus, much initial parallelism to be exploited), it was not obvious that this holds also for the worst case. By identifying the diagonal as a potentially worst-case shape (essentially, because in it any parallelism to be exploited does not come for free), we managed to first develop an $O(n\sqrt{n})$-time unrestricted transformation for transforming the diagonal into a line. Going a step further, we developed a universal transformation that can transform any pair of connected shapes to each other within $O(n\sqrt{n})$ moves, while connectivity of the shape is not necessarily preserved during its course.

Our attention was restricted in Chapter 4 on developing a set of efficient transformations that can additionally preserve connectivity throughout their course. Here, we were interested in keeping the associated graphs of all configurations connected during the transformation. Again, by focusing on the apparently hard instance of transforming a diagonal into a straight line, we build upon the algorithmic idea of partitioning the diagonal into segments in Chapter 3 to obtain two transformations of time $O(n\sqrt{n})$ that preserve the connectivity of the shape during transformations: one is based on *folding* segments and the other on *extending* them. Next, we further improved and gave very fast connectivity-preserving transformations that work on the family of all Hamiltonian shapes – that is, the associated graph of the shape contains a Hamiltonian line – is and matches the running time of the best known $O(n \log n)$-time transformation while additionally managing to keep the shape connected during transformations. Our most general result is then a first universal connectivity preserving transformation for this model that can transform any pairs of connected shapes within a sequence of $O(n\sqrt{n})$ moves and works on any pair of connected shapes of the same order.

Finally, in Chapter 5, we presented a distributed algorithmic framework for the linear-strength of line moves. 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, is then the first distributed connectivity-preserving transformation that exploits line moves and can work for all pairs of connected shape 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.

Overall, this research may offer some insights into other areas of theoretical computer science. For example, the presented transformations and their underlying principle may be applied to the combinatorial game theory of one-player games (e.g. puzzles). Their logic resembles applying a global force to all entities in a swarm, which may be beneficial in investigating complexity and decidability problems.

For example, a maze of several balls that are manoeuvring into target positions in a board is a typical instance of a dexterity puzzle. This is a one-player game that raises the question of how to find the minimum number of moves or decide whether a given game is solvable. The recent work of Becker *et al.* [25] and others (e.g. [48, 81]) provides some answers regarding complexity and highlights the underlying fundamental connections between puzzles and other robotic systems. This is supported by a useful discussion of some well-known puzzles, including sliding-block and block-pushing puzzles. Many puzzles are placed on a two-dimensional square board (e.g. Nine Men's Morris and the Fifteen Puzzle) and hence may represent some special-case transformations. Moreover, several of the technical tools described in this study may be adopted there for analysing complexity, such as proving lower bounds.

Furthermore, Michail and Spirakis [90] recently introduced the *network constructors* model, which is inspired by population protocols. It refers to a system containing a population of finite automata capable of stably forming several spanning shapes (e.g. line, ring, star) via pairwise interactions of a simple set of rules that are guided by an *adversary scheduler*. The formations were considered to converge in that study, and a *uniform random scheduler* was used to scale their efficiency (convergence time). They also provided a general protocol capable of forming a large set of shapes that could be represented by a Turing machine (TM). Thus, some of our study's algorithmic tools may be useful for further investigating those protocols under a more realistic model of physical and geometrical restrictions in order to achieve terminating procedures that can construct two- or three-dimensional shapes.

6.2 Future research directions

This thesis offered a number of interesting problems and research directions. The obvious first target (and apparently intriguing) is to answer whether there is an $o(n \log n)$-time transformation (e.g. linear) or whether there is an $\Omega(n \log n)$-time lower bound matching our best transformations (even when connectivity can be broken). We suspect the latter, but do not have enough evidence to support or prove it. As a first step, it might be easier to develop lower bounds for the connectivity-preserving case. The tree representation of the problem that we discuss in Section 3.1.3 (see, for example Figure 3.4), might help in this direction. Potential ways to establish the lower bound could be by partitioning executions into 'charging phases' that no transformation can avoid (e.g. all nodes are initially in cells and have to get out of them at some point in the execution, similar to what we have discussed in Section 2.5.2).

The proposed approach in Section 5.2 is essentially a distributed implementation of the centralised Hamiltonian transformation in Section 4.2. 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 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.

Universal transformations. We restrict attention to the class of Hamiltonian shapes. This class, 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, that is, distributed transformations working for any possible pair of initial and target shapes. This is because geometric shapes tend to have long simple paths, 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.

Optimal transformations. There are also a number of interesting variants of the present model. One is a centralised parallel version in which more than one line can be moved concurrently in a single time-step. Thus, there are variants of our transformations (or alternative ones) that further reduce the running time? In other words, are there parallelisable transformations in this model? In particular, it would be interesting to investigate whether the present model permits an $O(\log n)$ parallel time (universal) transformation – that is, matching the best transformation in the model of Aloupis *et al.* [18].

Extension to other regular grids. It would also be worth studying in more depth the case in which connectivity has to be preserved during the transformations. For example, a direct direction of developing an $O(n \log n)$-time centralised universal transformation that can preserve connectivity and work for any pairs of connected shapes with the same order. In the relevant literature, a number of alternative types of grids have been considered, like triangular (e.g. in [51]) and hexagonal (e.g. in [117]), 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 [125] might prove useful for this extension). Moreover, an immediate next goal is to attempt to develop a more general distributed algorithmic framework of line moves that could function in more restricted systems of huge number of simple devices.

Fault-tolerant transformations. Furthermore, the transformations considered in this thesis are assumed to operate flawlessly at all times throughout their courses. Thus, it will be intriguing to bring them closer to a realistic setting where failures are possible. Hence, a *fault tolerance* is another desirable requirement through which a transformation can correctly handle any potential failures in the system. This, of course, includes the two main faults: *crash faults* – one or a set of faulty robots suddenly stop working in the system (e.g. collision, dead) or *Byzantine faults* – more general and harder in which faulty robots behave unusually or incorrectly (e.g. corruption, malicious). Then, one might think about designing a solution that allows non-faulty robots to carry out the transformation without the need of any faulty parts, see for example a recent line recovery in [57] for programmable matter.

Non-uniform environment. Another possible extension of the proposed model is to deal with the presence of obstacles. Assume a collection of robots placed on a grid with fixed obstacles, can we design such a solution that brings all robots to their final positions without a global control that reaches all of them uniformly? Fekete [69] mentioned that this question is still under development, though emphasised that some configurations in this problem tend to be computationally difficult. On the positive side, it is still possible to rearrange a collection of robots moving on two-dimensional artificial obstacles. Moreover, he also provided a good overview of the studies conducted on the limitations and universality of the computational internal capability of entities transforming in a robotic system.

Finally, the above are not exclusive, as looking at the concept from a wide perspective reveals a vast spectrum of new algorithmic challenges that must be addressed in order to bring those innovative robotic systems, e.g. programmable matter, closer to reality. Moreover, the ongoing physical advancements in robotic systems do have an influence on their theoretical foundations and vice versa. Thus, this emphasises the importance of continuous progress in establishing solid mathematical solutions, motivating new versatile frameworks, and providing provable guarantee for the practical implementations of programmable matter.

Bibliography

[1] Ocado shops its way to a robotics platform for groceries and beyond. https://venturebeat.com/2020/11/13/ocado-shops-its-way-to-a-robotics-platform-for-groceries-and-beyond/. Accessed: 11 August 2021.

[2] The Ocado smart platform (OSP). https://www.ocadogroup.com/our-solutions/what-is-osp. Accessed: 22 July 2021.

[3] The programmable matter project. https://www.programmable-matter.com. Accessed: 23 July 2021.

[4] Teams of the programmable matter project. https://www.programmable-matter.com/consortium. Accessed: 23 July 2021.

[5] Thousands of orders cancelled after Ocado robot fire. https://www.bbc.co.uk/news/business-57883332. Accessed: 11 August 2021.

[6] Leonard M Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[7] Chrysovalandis Agathangelou, Chryssis Georgiou, and Marios Mavronicolas. A distributed algorithm for gathering many fat mobile robots in the plane. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 250–259, 2013.

[8] Hugo A Akitaya, Esther M Arkin, Mirela Damian, Erik D Demaine, Vida Dujmović, Robin Flatland, Matias Korman, Belen Palop, Irene Parada, André van Renssen, et al. Universal reconfiguration of facet-connected modular robots by pivots: the $O(1)$ musketeers. *Algorithmica*, 83(5):1316–1351, 2021.

[9] Abdullah Almethen, Othon Michail, and Igor Potapov. On efficient connectivity-preserving transformations in a grid. *To appear in: Theoretical Computer Science.*

[10] Abdullah Almethen, Othon Michail, and Igor Potapov. Pushing lines helps: Efficient universal centralised transformations for programmable matter. In *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, ALGOSENSORS*, pages 41–59. Springer, 2019.

[11] Abdullah Almethen, Othon Michail, and Igor Potapov. Pushing lines helps: Efficient universal centralised transformations for programmable matter. arXiv preprint arXiv:1904.12777, 2019.

[12] Abdullah Almethen, Othon Michail, and Igor Potapov. On efficient connectivity-preserving transformations in a grid. In *Algorithms for Sensor Systems - 16th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS*, volume 12503, pages 76–91, 2020.

[13] Abdullah Almethen, Othon Michail, and Igor Potapov. On efficient connectivity-preserving transformations in a grid. arXiv preprint arXiv:2005.08351, 2020.

[14] Abdullah Almethen, Othon Michail, and Igor Potapov. Pushing lines helps: Efficient universal centralised transformations for programmable matter. *Theoretical Computer Science*, 830-831:43 – 59, 2020.

[15] Abdullah Almethen, Othon Michail, and Igor 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*, volume 12961, pages 1–16. Springer, 2021.

[16] Abdullah Almethen, Othon Michail, and Igor Potapov. Distributed transformations of hamiltonian shapes based on line moves. arXiv preprint arXiv:2108.08953, 2021.

[17] Greg Aloupis, Nadia Benbernou, Mirela Damian, Erik Demaine, Robin Flatland, John Iacono, and Stefanie Wuhrer. Efficient reconfiguration of lattice-based modular robots. *Computational geometry*, 46(8):917–928, 2013.

[18] Greg Aloupis, Sébastien Collette, Erik Demaine, Stefan Langerman, Vera Sacristán, and Stefanie Wuhrer. Reconfiguration of cube-style modular robots using O(logn)

parallel moves. In *International Symposium on Algorithms and Computation*, pages 342–353. Springer, 2008.

[19] Dana Angluin, James Aspnes, Zoë Diamadi, Michael Fischer, and René Peralta. Computation in networks of passively mobile finite-state sensors. *Distributed Computing*, 18(4):235–253, 2006.

[20] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, November 2007.

[21] DJ Arbuckle and Aristides AG Requicha. Self-assembly and self-repair of arbitrary shapes by a swarm of reactive robots: algorithms and simulations. *Autonomous Robots*, 28(2):197–211, 2010.

[22] Philip Ball. Make your own world with programmable matter. *IEEE Spectrum*, 27, 2014.

[23] Levent Bayındır. A review of swarm robotics tasks. *Neurocomputing*, 172:292–321, 2016.

[24] Rida A Bazzi and Joseph L Briones. Stationary and deterministic leader election in self-organizing particle systems. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 22–37. Springer, 2019.

[25] Aaron T Becker, Erik D Demaine, Sándor Fekete, Jarrett Lonsford, and Rose Morris-Wright. Particle computation: complexity, algorithms, and logic. *Natural Computing*, 18(1):181–201, 2019.

[26] Dan Boneh, Christopher Dunworth, Richard J Lipton, and Jiri Sgall. On the computational power of DNA. *Discrete Applied Mathematics*, 71(1-3):79–94, 1996.

[27] Vincenzo Bonifaci, Kurt Mehlhorn, and Girish Varma. Physarum can compute shortest paths. *Journal of theoretical biology*, 309:121–133, 2012.

[28] Julien Bourgeois and Seth Goldstein. Distributed intelligent MEMS: progresses and perspective. *IEEE Systems Journal*, 9(3):1057–1068, 2015.

[29] Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.

[30] Zack Butler, Keith Kotay, Daniela Rus, and Kohji Tomita. Generic decentralized control for lattice-based self-reconfigurable robots. *The International Journal of Robotics Research*, 23(9):919–937, 2004.

[31] Cameron Chalk, Austin Luchsinger, Eric Martinez, Robert Schweller, Andrew Winslow, and Tim Wylie. Freezing simulates non-freezing tile automata. In *International Conference on DNA Computing and Molecular Programming*, pages 155–172. Springer, 2018.

[32] Arturo Chavoya and Yves Duthen. Using a genetic algorithm to evolve cellular automata for 2D/3D computational development. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 231–232, 2006.

[33] Fengqi Chen, Yukiko Yamauchi, Shuji Kijima, and Masafumi Yamashita. Locomotion of metamorphic robotic system based on local information. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, pages 40–45. IEEE, 2014.

[34] Ho-Lin Chen, David Doty, Dhiraj Holden, Chris Thachuk, Damien Woods, and Chun-Tao Yang. Fast algorithmic self-assembly of simple shapes using random agitation. In *International Workshop on DNA-Based Computers*, pages 20–36. Springer, 2014.

[35] G.S. Chirikjian. Kinematics of a metamorphic robotic system. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 449–455 vol.1, 1994.

[36] Pavel Chvykov, Thomas A Berrueta, Akash Vardhan, William Savoie, Alexander Samland, Todd D Murphey, Kurt Wiesenfeld, Daniel I Goldman, and Jeremy L England. Low rattling: A predictive principle for self-organization in active collectives. *Science*, 371(6524):90–95, 2021.

[37] Mark Cieliebak, Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Distributed computing by mobile robots: Gathering. *SIAM Journal on Computing*, 41(4):829–879, 2012.

[38] Arthur C Clarke. Extra-terrestrial relays. *Wireless World*, page 305–308, 1945.

[39] Edgar F Codd. *Cellular automata*. Academic press, 2014.

[40] Carlos Hernández Corbato, Mukunda Bharatheesha, Jeff Van Egmond, Jihong Ju, and Martijn Wisse. Integrating different levels of automation: Lessons from winning the Amazon robotics challenge 2016. *IEEE Transactions on Industrial Informatics*, 14(11):4916–4926, 2018.

[41] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[42] Alejandro Cornejo, Fabian Kuhn, Ruy Ley-Wild, and Nancy Lynch. Keeping mobile robot swarms connected. In *Proceedings of the 23rd international conference on Distributed computing*, DISC'09, pages 496–511, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] Jurek Czyzowicz, Dariusz Dereniowski, and Andrzej Pelc. Building a nest by an automaton. *Algorithmica*, 83(1):144–176, 2021.

[44] Shantanu Das, Paola Flocchini, Nicola Santoro, and Masafumi Yamashita. On the computational power of oblivious robots: forming a series of geometric patterns. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 267–276, 2010.

[45] Shantanu Das, Paola Flocchini, Nicola Santoro, and Masafumi Yamashita. Forming sequences of geometric patterns with oblivious mobile robots. *Distributed Computing*, 28(2):131–145, April 2015.

[46] Joshua Daymude, Zahra Derakhshandeh, Robert Gmyr, Alexandra Porter, Andréa Richa, Christian Scheideler, and Thim Strothmann. On the runtime of universal coating for programmable matter. *Natural Computing*, 17(1):81–96, 2018.

[47] Joshua J Daymude. *Collaborating in Motion: Distributed and Stochastic Algorithms for Emergent Behavior in Programmable Matter*. PhD thesis, Arizona State University, 2021.

[48] Erik D Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *International Symposium on Mathematical Foundations of Computer Science*, pages 18–33. Springer, 2001.

[49] Erik D Demaine, Matthew J Patitz, Robert T Schweller, and Scott M Summers. Self-assembly of arbitrary shapes using rnase enzymes: Meeting the kolmogorov bound with small scale factor. In *28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2011.

[50] Mattia D'Emidio, Gabriele Di Stefano, Daniele Frigioni, and Alfredo Navarra. Characterizing the computational power of mobile robots on graphs and implications for the euclidean plane. *Information and Computation*, 263:57–74, 2018.

[51] Zahra Derakhshandeh, Shlomi Dolev, Robert Gmyr, Andréa W Richa, Christian Scheideler, and Thim Strothmann. Brief announcement: amoebot–a new model for programmable matter. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 220–222. ACM, 2014.

[52] Zahra Derakhshandeh, Robert Gmyr, Andréa Richa, Christian Scheideler, and Thim Strothmann. An algorithmic framework for shape formation problems in self-organizing particle systems. In *Proceedings of the Second Annual International Conference on Nanoscale Computing and Communication*, page 21. ACM, 2015.

[53] Zahra Derakhshandeh, Robert Gmyr, Andréa Richa, Christian Scheideler, and Thim Strothmann. Universal shape formation for programmable matter. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 289–299. ACM, 2016.

[54] Zahra Derakhshandeh, Robert Gmyr, Thim Strothmann, Rida Bazzi, Andréa W Richa, and Christian Scheideler. Leader election and shape formation with self-organizing programmable matter. In *International Workshop on DNA-Based Computers*, pages 117–132. Springer, 2015.

[55] A. Deutsch and S. Dormann. *Cellular Automaton Modeling of Biological Pattern Formation: Characterization, Examples, and Analysis*. Modeling and Simulation in Science, Engineering and Technology. Birkhäuser Boston, 2018.

[56] Giuseppe A. Di Luna, Paola Flocchini, Nicola Santoro, Giovanni Viglietta, and Yukiko Yamauchi. Shape formation by programmable particles. *Distributed Computing*, Mar 2019.

[57] Giuseppe Antonio Di Luna, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Giovanni Viglietta. Line recovery by programmable particles. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ICDCN'18, pages 4:1–4:10, New York, NY, USA, 2018. ACM.

[58] Keisuke Doi, Yukiko Yamauchi, Shuji Kijima, and Masafumi Yamashita. Exploration of finite 2D square grid by a metamorphic robotic system. In *International Symposium on Stabilizing, Safety, and Security of Distributed Systems*, pages 96–110. Springer, 2018.

[59] Marco Dorigo, Guy Theraulaz, and Vito Trianni. Reflections on the future of swarm robotics. *Science Robotics*, 5(49), 2020.

[60] David Doty. Theory of algorithmic self-assembly. *Communications of the ACM*, 55:78–88, 2012.

[61] David Doty. Timing in chemical reaction networks. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 772–784. SIAM, 2014.

[62] Shawn Douglas, Hendrik Dietz, Tim Liedl, Björn Högberg, Franziska Graf, and William Shih. Self-assembly of DNAinto nanoscale three-dimensional shapes. *Nature*, 459(7245):414, 2009.

[63] Fabien Dufoulon, Shay Kutten, and William K. Moses Jr. Efficient deterministic leader election for programmable matter. PODC'21, page 103–113, New York, NY, USA, 2021. Association for Computing Machinery.

[64] Adrian Dumitrescu and János Pach. Pushing squares around. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 116–123. ACM, 2004.

[65] Adrian Dumitrescu, Ichiro Suzuki, and Masafumi Yamashita. Formations for fast locomotion of metamorphic robotic systems. *The International Journal of Robotics Research*, 23(6):583–593, 2004.

[66] Adrian Dumitrescu, Ichiro Suzuki, and Masafumi Yamashita. Motion planning for metamorphic systems: Feasibility, decidability, and distributed reconfiguration. *IEEE Transactions on Robotics and Automation*, 20(3):409–418, 2004.

[67] Karthik Elamvazhuthi and Spring Berman. Mean-field models in swarm robotics: A survey. *Bioinspiration & Biomimetics*, 15(1):015001, 2019.

[68] Sándor Fekete, Andréa Richa, Kay Römer, and Christian Scheideler. Algorithmic foundations of programmable matter (Dagstuhl Seminar 16271). 6(7), 2016. Also in *ACM SIGACT News*, 48.2:87-94, 2017.

[69] Sándor P Fekete. Geometric aspects of robot navigation: From individual robots to massive particle swarms. In *Distributed Computing by Mobile Entities*, pages 587–614. Springer, 2019.

[70] Sándor P Fekete, Robert Gmyr, Sabrina Hugo, Phillip Keldenich, Christian Scheffer, and Arne Schmidt. Cadbots: Algorithmic aspects of manipulating programmable matter with finite automata. *Algorithmica*, 83(1):387–412, 2021.

[71] Paola Flocchini, David Ilcinkas, Andrzej Pelc, and Nicola Santoro. Computing without communicating: Ring exploration by asynchronous oblivious robots. *Algorithmica*, 65(3):562–583, 2013.

[72] Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. Distributed computing by mobile entities. *Current Research in Moving and Computing*, 11340, 2019.

[73] Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Peter Widmayer. Arbitrary pattern formation by asynchronous, anonymous, oblivious robots. *Theoretical Computer Science*, 407(1-3):412–447, 2008.

[74] Nao Fujinaga, Yukiko Yamauchi, Hirotaka Ono, Shuji Kijima, and Masafumi Yamashita. Pattern formation by oblivious asynchronous mobile robots. *SIAM Journal on Computing*, 44(3):740–785, 2015.

[75] T. Fukuda, S. Nakagawa, Y. Kawauchi, and M. Buss. Self organizing robots based on cell structures - ckbot. In *IEEE International Workshop on Intelligent Robots*, pages 145–150, 1988.

[76] Toshio Fukuda. Self organizing robots based on cell structures-cebot. In *Proc. IEEE Int. Workshop on Intelligent Robots and Systems (IROS'88)*, pages 145–150, 1988.

[77] Melvin Gauci, Jianing Chen, Wei Li, Tony J Dodd, and Roderich Groß. Self-organized aggregation without computation. *The International Journal of Robotics Research*, 33(8):1145–1161, 2014.

[78] Kyle Gilpin, Ara Knaian, and Daniela Rus. Robot pebbles: One centimeter modules for programmable matter through self-disassembly. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2485–2492. IEEE, 2010.

[79] Robert Gmyr, Kristian Hinnenthal, Irina Kostitsyna, Fabian Kuhn, Dorian Rudolph, Christian Scheideler, and Thim Strothmann. Forming tile shapes with simple robots. *Natural Computing*, 19:375–390, 2020.

[80] Heiko Hamann. *Swarm robotics: A formal approach.* Springer, 2018.

[81] Robert A Hearn and Erik D Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, 2005.

[82] Ferran Hurtado, Enrique Molina, Suneeta Ramaswami, and Vera Sacristán. Distributed reconfiguration of 2D lattice-based modular robotic systems. *Autonomous Robots*, 38(4):383–413, 2015.

[83] Andrew Ilachinski. *Cellular automata: a discrete universe.* World Scientific Publishing Company, 2001.

[84] Alon Itai, Christos Papadimitriou, and Jayme Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, 1982.

[85] Ara Knaian, Kenneth Cheung, Maxim Lobovsky, Asa Oines, Peter Schmidt-Neilsen, and Neil 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*, pages 1447–1453. IEEE, 2012.

[86] Sam Kriegman, Douglas Blackiston, Michael Levin, and Josh Bongard. A scalable pipeline for designing reconfigurable organisms. *Proceedings of the National Academy of Sciences*, 117(4):1853–1859, 2020.

[87] Shuguang Li, Richa Batra, David Brown, Hyun-Dong Chang, Nikhil Ranganathan, Chuck Hoberman, Daniela Rus, and Hod Lipson. Particle robotics based on statistical mechanics of loosely coupled components. *Nature*, 567(7748):361–365, 2019.

[88] Othon Michail. Terminating distributed construction of shapes and patterns in a fair solution of automata. *Distributed Computing*, 31(5):343–365, 2018.

[89] Othon Michail, George Skretas, and Paul Spirakis. On the transformation capability of feasible mechanisms for programmable matter. *Journal of Computer and System Sciences*, 102:18–39, 2019.

[90] Othon Michail and Paul Spirakis. Simple and efficient local codes for distributed stable network construction. *Distributed Computing*, 29(3):207–237, 2016.

[91] Othon Michail and Paul Spirakis. Elements of the theory of dynamic networks. *Commun. ACM*, 61(2):72–81, 2018.

[92] Marvin Lee Minsky. *Computation.* Prentice-Hall Englewood Cliffs, 1967.

[93] An Nguyen, Leonidas J Guibas, and Mark Yim. Controlled module density helps reconfiguration planning. *New Directions in Algorithmic and Computational Robotics*, pages 23–36, 2001.

[94] Thomas Nickson and Igor Potapov. Broadcasting automata and patterns on $\mathbb{Z}^2$. In *Automata, Universality, Computation*, pages 297–340. Springer, 2015.

[95] Norman H Packard and Stephen Wolfram. Two-dimensional cellular automata. *Journal of Statistical physics*, 38(5):901–946, 1985.

[96] Matthew J Patitz. An introduction to tile-based self-assembly and a survey of recent results. *Natural Computing*, 13(2):195–224, 2014.

[97] Benoit Piranda and Julien Bourgeois. Designing a quasi-spherical module for a huge modular robot to create programmable matter. *Autonomous Robots*, 42(8):1619–1633, 2018.

[98] V Prem Prakash, C Patvardhan, and Anand Srivastav. Effective heuristics for the bi-objective euclidean bounded diameter minimum spanning tree problem. In *International Conference on Next Generation Computing Technologies*, pages 580–589. Springer, 2017.

[99] Chris R Reid and Tanya Latty. Collective behaviour and swarm intelligence in slime moulds. *FEMS microbiology reviews*, 40(6):798–806, 2016.

[100] Paul Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

[101] Paul Rothemund and Erik Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the 32nd annual ACM symposium on Theory of computing (STOC)*, pages 459–468. ACM, 2000.

[102] Michael Rubenstein, Alejandro Cornejo, and Radhika Nagpal. Programmable self-assembly in a thousand-robot swarm. *Science*, 345(6198):795–799, 2014.

[103] Michael Rubenstein and Wei-Min Shen. Automatic scalable size selection for the shape of a distributed robotic collective. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 508–513. IEEE, 2010.

[104] Hans Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.

[105] William Savoie, Thomas A Berrueta, Zachary Jackson, Ana Pervan, Ross Warkentin, Shengkai Li, Todd D Murphey, Kurt Wiesenfeld, and Daniel I Goldman. A robot made of robots: Emergent transport and control of a smarticle ensemble. *Science Robotics*, 4(34), 2019.

[106] Nicholas Schiefer and Erik Winfree. Universal computation and optimal construction in the chemical reaction network-controlled tile assembly model. In *International Workshop on DNA-Based Computers*, pages 34–54. Springer, 2015.

[107] Joel L Schiff. *Cellular automata: a discrete view of the world*, volume 45. John Wiley & Sons, 2011.

[108] David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *natural computing*, 7(4):615–633, 2008.

[109] David Soloveichik and Erik Winfree. Complexity of self-assembled shapes. *SIAM Journal on Computing*, 36(6):1544–1569, 2007.

[110] Thim Frederik Strothmann. *Self-* algorithms for distributed systems: programmable matter & overlay networks*. PhD thesis, Universität Paderborn, 2017.

[111] Ichiro Suzuki and Masafumi Yamashita. Distributed anonymous mobile robots: Formation of geometric patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

[112] Atsushi Tero, Ryo Kobayashi, and Toshiyuki Nakagaki. A mathematical model for adaptive transport network in path finding by true slime mold. *Journal of theoretical biology*, 244(4):553–564, 2007.

[113] Tommaso Toffoli and Norman Margolus. Programmable matter: concepts and realization. *Physica. D, Nonlinear phenomena*, 47(1-2):263–272, 1991.

[114] Stanislaw Ulam et al. Random processes and transformations. In *Proceedings of the International Congress on Mathematics*, volume 2, pages 264–275. Citeseer, 1952.

[115] Jules Verne. *Vingt mille lieues sous les mers*. Pierre-Jules Hetzel, 1870.

[116] John Von Neumann. The general and logical theory of automata. *Cerebral mechanisms in behavior*, 1:41, 1951.

[117] Jennifer Walter, Jennifer Welch, and Nancy Amato. Distributed reconfiguration of metamorphic robot chains. *Distributed Computing*, 17(2):171–189, 2004.

[118] Erik Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, 1998.

[119] Stephen Wolfram. *A new kind of science*, volume 5. Wolfram media Champaign, IL, 2002.

[120] Stephen Wolfram. *Cellular automata and complexity: collected papers*. CRC Press, 2018.

[121] Damien Woods, H Chen, Scott Goodfriend, Nadine Dabby, Erik Winfree, and Peng Yin. Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 353–354. ACM, 2013.

[122] Hui Xie, Mengmeng Sun, Xinjian Fan, Zhihua Lin, Weinan Chen, Lei Wang, Lixin Dong, and Qiang He. Reconfigurable magnetic microrobot swarm: Multimode transformation, locomotion, and manipulation. *Science robotics*, 4(28), 2019.

[123] Masafumi Yamashita and Tsunehiko Kameda. Computing on anonymous networks. i. characterizing the solvable cases. *IEEE Transactions on parallel and distributed systems*, 7(1):69–89, 1996.

[124] Masafumi Yamashita and Ichiro Suzuki. Characterizing geometric patterns formable by oblivious anonymous mobile robots. *Theoretical Computer Science*, 411(26-28):2433–2453, 2010.

[125] Yukiko Yamauchi. Symmetry of anonymous robots. In *Distributed Computing by Mobile Entities*, pages 109–133. Springer, 2019.

[126] Mark Yim, WeiMin Shen, Behnam Salemi, Daniela Rus, Mark Moll, Hod Lipson, Eric Klavins, and Gregory Chirikjian. Modular self-reconfigurable robot systems [grand challenges of robotics]. *IEEE Robotics & Automation Magazine*, 14(1):43–52, 2007.

[127] Dan Zhang, LG Pee, and Lili Cui. Artificial intelligence in E-commerce fulfillment: A case study of resource orchestration at Alibaba's smart warehouse. *International Journal of Information Management*, 57:102304, 2021.