ABSTRACT

Title of dissertation: EFFICIENT ALGORITHMS FOR COORDINATED MOTION IN SHARED SPACES

Philip Dasler
Doctor of Philosophy, 2020

Dissertation directed by: Professor David M. Mount
Department of Computer Science

The steady development of autonomous systems motivates a number of interesting algorithmic problems. These systems, such as self-driving cars, must contend with far more complex and dynamic environments than factory floor robots of the past.

This dissertation seeks to identify optimization problems that are simple enough to analyze formally, yet realistic enough to contribute to the eventual design of systems extant in real-world, physical spaces. With that in mind, this work examines three problems in particular: automated vehicles and unregulated traffic crossings, a smart network for city-wide traffic flow, and an online organizational scheme for automated warehouses.

First, the *Traffic Crossing Problem* is introduced, in which a set of n axis-aligned vehicles moving monotonically in the plane must reach their goal positions within a time limit and subject to a maximum speed limit. It is shown that this problem is NP-complete and efficient algorithms for two special cases are given.

Next, motivated by a problem of computing a periodic schedule for traffic lights in an urban transportation network, the problem of *Circulation with Modular Demands* is introduced. A novel variant of the well-known minimum-cost circulation problem in directed networks, in this problem vertex demand values are taken from the integers modulo λ, for some integer $\lambda \geq 2$. This modular circulation problem is solvable in polynomial time when $\lambda = 2$, but the problem is NP-hard when $\lambda \geq 3$. For this case, a polynomial time approximation algorithm is provided.

Finally, a theoretical model for organizing portable storage units in a warehouse subject to an online sequence of access requests is proposed. Complicated by the unknown request frequencies of stored products, the warehouse must be arranged with care. Analogous to virtual-memory systems, the more popular and oft-requested an item is, the more efficient its retrieval should be. Two formulations are considered, dependent on the number of access points to which storage units must be brought, and algorithms that are $O(1)$-competitive with respect to an optimal algorithm are given. Additionally, in the case of a single access point, the solution herein is asymptotically optimal with respect to density.

EFFICIENT ALGORITHMS FOR COORDINATED MOTION IN SHARED SPACES

by

Philip Dasler

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2020

Advisory Committee:
Dr. David M. Mount, Chair/Advisor
Dr. William Gasarch
Dr. William Goldman
Dr. Dinesh Manocha
Dr. Dana S. Nau

Table of Contents

Chapter 1: Introduction

As autonomous systems become more prevalent in everyday life, there is a growing interest in maximizing their efficiency. Humans are increasingly sharing physical space with robots, automation, and various forms of ubiquitous computing, and as these systems continue to require more of our resources, we in turn require greater efficiency from them. Just as the unprecedented growth of data requires a revisiting of efficient storage and movement algorithms, so too does the growing physical presence of automated systems. No longer confined to designated areas free of the possibility of interaction, the ever-increasing number of autonomous systems working under the constraints of shared space require gains in efficiency in order to continue to do their jobs and to do them well.

This need for efficiency motivates a number of interesting algorithmic problems. Applying techniques of algorithm design and computational geometry to autonomous systems in physical spaces allows us to analyze how complex these systems are, to assess the difficulty of the tasks they are attempting to solve, and to determine how best to devise solutions to these tasks.

Take, for example, the steady development of automated vehicle technology. Groups of such autonomous agents have the potential to increase the overall efficiency of entire traffic systems through stronger and more purposeful coordination. The practical engineering of such solutions will require the consideration of myriad issues, including the physical limitations of vehicle motion, the complexities of traffic and

urban navigation, and human factors. However, underlying these practical issues are broader, more fundamental theoretical issues of working within a shared space that, when investigated, could lead to overarching insights into how best to solve these problems.

Preliminary work has shown that the efficiency of solutions to the problems of working in physical spaces can be very sensitive to the geometric layout of the system. While fundamentally this dissertation involves objects moving through space, it differs from the typical motion planning problem by focusing mainly on the constraints imposed by the geometry of the spaces and the moving vehicles. Automatic traffic management, for example, can be viewed at many scales, ranging from global issues such as vehicle routing through a city [1–4], down to low-level issues that concern themselves with vehicle kinematics and configuration spaces [5,6], but the focus of this dissertation lies somewhere in between and draws heavily from the tools and techniques of computational geometry and algorithmic analysis. Thus, the goal is to identify optimization problems that are simple enough to analyze formally, yet realistic enough to contribute to the eventual design of systems rooted in shared, physical spaces. With that in mind, three problems in particular are examined: automated vehicles and unregulated traffic crossings, a smart network for city-wide traffic flow, and an online organizational scheme for automated warehouses.

1.1 Unregulated Traffic Crossings

In urban settings, road intersections are regulated by traffic lights or stop/yield signs. Neither is an especially efficient solution. For example, a traffic light locks the entire intersection, preventing cross traffic from entering it even when there is adequate space and time to do so. Automated traffic control has the potential to realize gains in efficiency. Chapter 3 introduces two algorithmic formulations of automated traffic control in the form of the *Traffic Crossing Problem*, a problem that involves coordinating the motions of a set of vehicles moving through an unregulated traffic crossing. These simple formulations capture the essential computational challenges of coordinating crosswise motion through intersections and provide a foundation from which the work can evolve and grow. In both, vehicles move monotonically along axis-parallel lines (traffic lanes) in the plane. The objective is to plan the collision-free motion of these vehicles as they move to their goal positions, subject to deadlines and a global speed limit.

After a formal definition of the traffic-crossing problem in Section 3.2, three results are presented. First, it is shown in Section 3.3 that this problem is NP-complete. Second, in Section 3.5 a constrained version motivated by a scenario in which traffic moving in one direction (e.g., a major highway) has priority over crossing traffic is considered, and an algorithm that solves this problem in $O(kn \log n)$ time, for n vehicles and k crossing lanes, is provided. Finally, the problem is viewed in a discrete setting in Section 3.6, in which a solution is given that limits the maximum delay of any vehicle and is proven to be asymptotically optimal.

1.2 City-wide Coordination

While Chapter 3 is concerned with automating a system of vehicles passing through a single intersection, Chapter 4 considers the cascading effects of automating a system of multiple intersections.

Consider an urban transportation network consisting of a grid of horizontal and vertical roads as laid out on a map. Each pair of horizontal and vertical roads meets at a unique intersection with a controller that alternates, allowing either horizontal or vertical traffic through, such that the pattern repeats over a time interval λ. The goal is to synchronize these periodic systems so that traffic flows smoothly throughout the city with minimal delay.

Minimum (and maximum) cost network flows and circulations are fundamental computational problems in discrete optimization and are a natural avenue of attack for such a problem. Chapter 4 introduces the problem of *Circulation with Modular Demands* (or λ-*CMD*, for short), a variant of the classic circulation problem, where vertex demand values are taken from the integers modulo λ, for some integer $\lambda \geq 2$. For example, if $\lambda = 10$ a vertex with demand 6 can be satisfied by any net incoming flow of 6, 16, 26 and so on or a net outgoing flow of 4, 14, 24, and so on.

Section 4.2 discusses the relevance of the λ-CMD problem to traffic management in an urban grid, Section 4.3 presents some preliminary observations regarding this problem, and in Sections 4.4–4.6 three main results are given: (a) 2-CMD can be solved exactly in polynomial time, (b) λ-CMD is NP-hard, for $\lambda \geq 3$, and (c) there is a polynomial time $4(\lambda - 1)$-approximation to λ-CMD.

1.3 Online Warehouse Management

Online shopping has grown rapidly in recent years and, as such, the efficiency of warehouses and fulfillment centers is increasingly important in meeting customer demand at scale. Several companies have developed automated systems to help streamline operations in these warehouses, making use of standardized portable storage units and robotic retrieval systems. The robots maneuver themselves under standardized shelving units, lift them from below, and carry them to a location in the warehouse where a human waits to complete an order with items from the shelf. The frequency with which each storage unit is accessed varies, and, as access probabilities vary over time, there is a natural question of how to dynamically organize the warehouse's placement of storage units in order to guarantee efficient access at any time.

From a practical perspective there are many ways in which to model objects residing in a warehouse. To obtain meaningful theoretical results without imposing irrelevant technical details, this work proposes a very simple and general model, which encapsulates the most salient aspects of efficient self-organizing behavior. The model is based on similar models used for memory caching, but is here applied to physical storage space in the real world. Storage units, or *boxes*, are modeled as movable unit squares on a grid in the plane. In addition to the boxes, there are designated fixed points, called *access points*, where boxes are brought on demand. The problem input consists of a sequence of *access requests*, each specifying that a particular box in the system be moved to a given access point.

In Chapter 5, two versions of this problem are considered: the *attic problem*, where there is a single access point and the *warehouse problem*, where there are multiple access points. Additionally, two movement types are considered in each version: (a) lifting boxes directly and carrying them over the warehouse (as cargo containers are lifted by an overhead crane) and (b) sliding boxes along the ground, requiring clear paths in which to do so. For each problem/motion-type pair, an online algorithm is presented that is competitive with respect to an optimal solution with complete knowledge of the access sequence.

All three of the presented problems (unregulated traffic crossings, coordinated city-wide traffic flow, and online organization for automated warehouses) are examples of applying techniques from algorithm analysis and computational geometry to solve problems set in a shared physical space. Each example can be considered a case study of sorts, illustrating the efficiencies that can be gained in real-world physical systems by considering their theoretical boundaries and complexity. Additional studies of this kind could further elucidate the intersection of computational theory and physical systems, improve the efficiency of such systems, and broaden the applicability of these and similar results by continuing to generalize the models used. Chapter 6 briefly discusses remaining open problems and future avenues of exploration.

Chapter 2: Literature Review

This section provides an overview of literature relevant to the three algorithmic problems presented in their respective chapters. Prior work in these areas ranges from engineering approaches that are focused on realistic issues to algorithmic approaches that are very theoretic in nature. Related to many different topics, these works include publications covering automated traffic management, multiple-agent motion planning, kinetic algorithms in computational geometry, and dynamic flows in networks. This survey of works is meant to provide a general overview of applicable topics; references to sources that are more directly used (such as applying particular techniques introduced therein) are reserved for their respective chapters.

2.1 Motion Planning Through the Lens of Computational Geometry

As the work presented in this dissertation seeks to examine motion planning problems on a fundamental level, formally defining them and analyzing them from the perspective of their computational complexity, this review of the prior arts begins with (and primarily consists of) a survey of computational geometry techniques for dealing with moving objects, coordinated motion, and the computational complexity of motion planning.

2.1.1 Kinetic Data Structures

Introduced by Guibas [7] in 1998, Kinetic Data Structures (KDSs) are designed for solving typical computational geometry problems, but with the added complication of motion. Consider, for example, the problem of maintaining a data structure for answering visibility queries for a set of obstacles in motion. Creating a binary space partition is common for answering such queries, but algorithms to do so (and structures that store this information) assume that the objects in the space are static. Any movement of these objects may invalidate already derived structures. What is necessary, then, is a structure that can efficiently adjust as objects move.

KDSs assume that the motion of objects is known. Typically, they are created by adapting known algorithms for static problems to handle dynamic data. For example, Basch, Guibas, and Hershberger [8] focus on adapting algorithms for maintaining the convex hull and closest pair in a set of moving points. (Guibas refers to these types of problems which deal with the compactness of a set as *extent* problems.) Their technique is in large part a simple plane sweep, except with an added dimension to represent time. It is across this dimension that the plane actually sweeps.

These structures are initialized at some point in time, say t_0, with the respective positions and time-parameterized motions of the points and an initial solution to the problem (e.g., the convex hull of the points in their initial configuration). Next, a set of *certificates* is added. These certificates are a set of Boolean conditions which describe certain necessary properties for a calculated solution to be valid,

e.g., all points must lie within the interior of the convex hull. Along with these certificates is a set of update rules detailing the necessary steps for correcting a violated certificate. Next, a priority queue is created containing the first point in time at which each certificate fails. This priority queue provides a structure that reports which certificates are violated for any queried time t. If none are violated at this point, then the original solution is reported. If any certificates are violated, the solution is updated by working through the priority queue, encountering each certificate violation and applying that certificates update rule, until the solution has reached the appropriate state for time t.

While this, at first blush, requires the movement of points to be fixed (i.e., the velocity of a point does not change), Basch et al. do allow for changes in the motion of objects by way of an update to the priority queue. This queue is originally built from the known motion of objects at time t_0, but can be modified as needed by recalculating the failure times of each certificate (though in practice, this is rarely done as it is a costly operation).

These types of adapted KDS are evaluated across four criteria: responsiveness, efficiency, compactness, and locality. The *responsiveness* of a KDS is related to the computational complexity of repairing the structure after a certificate fails. The greater the complexity, the less responsive the structure is. *Efficiency* is the ratio between certificate failures and necessary changes to the solution. If few changes are necessary in the face of many certificate failures, the KDS is considered to be efficient. *Compactness* measures the size of the certificate set. Solution dependencies that are easily encapsulated in a small number of certificates lead to a compact KDS.

Finally, the *locality* of a KDS is a measure of the span of certificates. In other words, if each certificate governs a relatively small number of points in the set, then the KDS is considered to be relatively local.

The KDS approach has been successfully applied to a number of kinetic problems in computational geometry including collision detection [9–11], clustering [12–14], and closest pair calculations [15, 16].

Saltenis et al. tackle the problem of simply keeping track of the positions of a set of moving objects [17]. Their goal is to create an index that, when queried, will return all objects that lie within a given range and time interval. This range may be defined by two d-dimensional ranges, forming a $(d+1)$-dimensional trapezoid between the first and second ranges and across the time interval. The structure used to support this query is a modified R^*-tree which, at its leaves, stores both the position of the objects at time t_0 as well as a time-parameterized function describing its motion. Internal nodes store bounding boxes around points in their subtree which must also be time-paramaterized. Though no formal bounds are proven, experimental results show this new structure (deemed a time-parameterized R-tree or TPR-tree) to be capable of supporting queries efficiently and that its performance does not degrade over time.

Many of these methods may be useful for planning the motion of vehicles in the traffic crossing problem, but one major drawback of the KDS framework is that the vehicles are not treated as acting agents. Instead, their motion is assumed to be unchanging and known *a priori*. For further discussion on pathfinding involving multiple agents see Section 2.3.2.

2.1.2 Geometric Motion Planning

Motion planning, especially through obstacles, is a topic that has been well studied within the field of computational geometry.

Ó Dúnlaing, Sharir, and Yap [18] propose the use of Voronoi Diagrams as a simple motion planning solution to two special cases of the Movers' Problem. These cases involve moving either a disc or a line segment through a two-dimensional space bounded by polygonal obstacles. The solution, in essence, creates a cell structure such that the edges of the cells are as far from the obstacles as possible. This is a natural result of the Voronoi Diagram, as each edge is equidistant from the two points that define it, and, in this case, the points come from the obstacles. Given this, a motion plan can be created simply by finding the nearest edges on the diagram to the start and goal positions and then doing simple pathfinding along the diagram's edges. This method requires $O(n \log n)$ time for preprocessing and can answer path-finding queries in $O(n)$ time.

Although its primary focus is not on motion planning, the work by Abellanas et al. is instead a modification to the usual Voronoi Diagram algorithm. This modification constructs the Voronoi Diagram based on the time to travel between two points rather than based on Euclidean distances, and the authors illustrate its use by applying it to the problem of motion planning in the *Straight Line Transportation Model*. This model adds a line to the usual point set, representing a highway through a set of locations. The highway allows vehicles to travel at a greater speed than anywhere else in the space. The time-based Voronoi Diagram is then used to

determine the quickest paths between points. It seems that this time-based Voronoi Diagram method could be combined with the simple motion planning from the paper by Ó Dúnlaing, Sharir, and Yap above to create variable speed motion plans that optimize over travel time, although this is left for future work and not attempted in this dissertation.

While typical motion planning problems involve planning the paths of one or more agents from some starting configuration to some goal configuration, there are other ways to formulate them. One example of this is *the milling problem*, in which all points of a polygonal region must, at some point, be covered by a cutting head (typically circular or square). The task is to find the shortest path for the cutting head that accomplishes this without the cutting head leaving the region. In a paper by Arkin et al. [19], the milling problem is shown to be NP-hard. Several years later, the same authors published a 2.5-approximation algorithm for the problem [20].

The problem of milling has also been considered in the context where milling tools of various radii can be used [21]. While the milling problem is known to be NP-hard even when restricted to the case of a single tool, this paper presents a polynomial-time approximation algorithm for the multiple-tool milling problem.

Although milling may not directly involve the type of coordinated motion planning necessary to direct traffic, the concept of sweeping a polygon across a space while avoiding previously swept regions is connected in the sense that the traffic in a system creates collision zones that must be avoided by polygonal vehicles. Such zones are constructed and used for motion planning in Chapter 3.

Berger and Klein consider a dynamic motion-panning problem in a similar

vein to the one presented in Chapter 3, which is loosely based on the video game *Frogger* [22]. In *Frogger*, players control the eponymous frog, whose goal is to cross either a busy road or a flowing log flume. In the former, moving vehicles must be avoided lest the amphibious hero be brought to an untimely end. In the latter, the only way to cross the flowing water is by using passing logs as platforms. While both versions are quite similar to one another (one could treat the spaces between cars as though they were logs in the flume), there are some subtle differences with regard to game ending states (e.g., one can wait on a log until swept off-screen but there is much less time between vehicles). Berger and Klein actually focus on the latter version of the problem, in which moving objects in the space are considered *carriers* rather than obstacles. They show that the problem is undecidable for dimension $d \geq 8$, is NP-hard in the plane, and does not even admit a constant-factor approximation (under the assumption that P $\neq$ NP).

The work of Berger and Klein is based, at least in part, on the work of Arkin, Mitchell, and Polishchuk [23] in which a group of circular agents must cross a field of polygonal obstacles. These obstacles are dynamic, but their motion is fixed and known *a priori*. Similar to the work by Basch et al., time is treated as an additional spatial dimension, uniformly discretized. Additionally, rather than discretizing through a regular grid, each time-slice is discretized by a maximal packing of disks. Finally, each edge in the graph is assigned a capacity of one and its maximum flow is found, resulting in a pseudopolynomial-time dual-approximation algorithm. If there exists paths for K unit disks traveling at unit speed, the algorithm will find K paths for disks of radius $\Omega(1)$ traveling at speed $O(1)$.

2.1.2.1 Coordinated Motion

A subset of motion planning, coordinated motion covers movement plans for multiple agents so that, working together, they can achieve some overarching goal.

Kawamura and Kobayashi [24], for example, consider a multi-agent motion planning problem involving a set of agents patrolling a linear fence. As seen in previous papers, time is once again treated as an extra dimension, leading to a rectangular work space. As agents patrol the fence, they sweep out strips of this space in much the same way as a cutting head in the milling problem, though there is no freedom with respect to the velocity in the direction of time. The paper discusses coordinating the motion of agents so that either the entire space is covered by these strips (i.e., all points of the fence are under surveillance at all times) or the height of uncovered areas are minimized (i.e., the length of time an area of the fence is unseen is minimized).

The work by Pasqualetti et al. [25] is similar to the patrolling work above, though rather than dealing with a linear fence, a polygonal environment is patrolled. The paper does away with the environment relatively quickly in favor of a "robotic roadmap," i.e., a graph connecting a set of n viewpoints necessary for visibility to all points of the polygon. Having one robot placed at each of the n points simultaneously is necessary and sufficient for the entirety of the polygon to be covered. Thus, this is a patrolling problem across a branching graph rather than a single line.

Chiang et al. [26] study conflict detection and resolution in air traffic management (ATM). In this problem, planes enter a polygonal airspace at specified

locations and must reach one of several runways while avoiding conflicts with no-fly zones and other airplanes. The authors use the kinetic data structure for Delaunay triangulations detailed in the paper by Guibas et al. [27]. This allows for conflict detection in $O(n^3)$ in the worst case.

A subset of coordinated motion, swarm robotics deals with coordinating the motion of multiple agents under the constraints of physical systems. Take, for example, recent work by Arul et al. [28], which deals with the coordination of a swarm of quadcopters as they surveil a 3D urban environment. The generated motion plans provide for local collision avoidance with both static and dynamic obstacles in the field while taking into account coverage constraints of the space to be monitored, the dynamics constraints of the quadcopters, and the inevitable uncertainty in vehicle motion. The system scales linearly with respect to the number of agents, at least up to a few dozen, although this is demonstrated experimentally rather than through formal theoretic methods.

More theoretic in nature is the work by Demaine et al. [29] in which a swarm of robots is reconfigured, moving each robot from a starting position to a goal position, while minimizing distances travelled. Two models are given. The first, restricted to a grid, is shown to be NP-hard when looking for optimal paths, but the approximation algorithm provided yields a constant stretch factor. The second model uses disc shaped agents in the plane, with the same approximation result. While this work is quite similar to the work in Chapter 3 of this dissertation, there are some obvious differences in model definition (such as vehicle geometry) and this work optimizes distance traveled rather than time taken to reach goal positions as is

done in Chapter 3. Additionally, while this dissertation has been submitted after the work by Demaine et al., the published paper that Chapter 3 is based on predates it.

2.1.3 Complexity of Motion Planning

While the work above focuses mainly on techniques for planning motion, other works have focused more on theoretic analyses of motion planning in general. For example, Reif and Sharir [30] look at the computational complexity of what they refer to as the *dynamic mover's problem*. This problem involves planning the motion of some polygonal body (say, a couch) through a set of moving obstacles (say, a house party). Again, in this problem the movement of all obstacles is known beforehand. Reif and Sharir prove that, even with full knowledge of obstacle movement, this motion planning problem is NP-hard if there are no bounds on the velocity of the body. Added bounds push the problem into PSPACE-hardness.

Sharma and Aloimonos [31] study a problem called the *warehouseman's problem*, which involves the coordinated motion of objects within a confined space. The intuitive analogy presented is that of a warehouse full of boxes that must be rearranged within the confines of the warehouse to reach a desired configuration. The authors show that when the problem is defined as a two dimensional problem with rectangular, axis-aligned objects in a rectangular room, the problem is PSPACE-complete. Finally, they introduce some constraints on the space that allow for polynomial time algorithms.

Flake and Baum [32] perform an analysis on a similar problem, the well-known

sliding-block puzzle game, *Rush Hour*. This puzzle involves a set of rectangular vehicles on a square grid that can move either vertically or horizontally, but not both. The goal is to find a sequence of moves such that a target vehicle can exit the grid. Flake and Baum generalize this to an $n \times n$ grid and show that deciding whether or not a set of legal moves exist such that the target car can escape is PSPACE-complete.

Hearn and Demaine [33] go one step further by generalizing the results above. Through the nondeterministic constraint logic model of computation, they show that sliding-block puzzles in general are PSPACE-hard. In particular, they strengthen the conditions for PSPACE-completeness of the Warehouseman's Problem and PSPACE-completeness of *Rush Hour*.

Work by Gupta and Nau [34] discusses the complexity of planning in the blocks-world environment. Specifically, they show that planning in this context (and in several variants of the problem) is NP-hard due to deadlocks. They go on to provide a simple hill-climbing algorithm that finds an optimal solution in the absence of deadlocks and explain why deadlocks are so difficult to deal with. Many aspects of this work seem to parallel the challenges provided by the problems presented in Chapters 3 and 4. For example, in these problems solutions can be presented as sets of partial orderings, either determining the order in which blocks are handled or when vehicles cross through intersections.

2.2 Network Flows

Motion planning problems exist at a scale commensurate to the objects for whom motion is being planned, but inefficiencies at this scale can cascade to those above. If one street intersection is jammed, there is a reasonable chance it will cause delays elsewhere in a network as well. To understand how these cascades affect the network and how they can be mitigated, a new type of optimization problem based on network flows is introduced in Chapter 4.

Network flow problems, or the more general circulation problems, are a class of optimization problems in which some commodity must make its way across a network under certain constraints (e.g., edge capacity limits) and with the goal of maximizing or minimizing some utility (e.g, the sum of the weights of used edges). The standard minimum-cost circulation problem is well studied. The reader is referred to any of a number of standard sources on this topic, for example, [35–40].

Dynamic network flow problems, introduced by Ford and Fulkerson [41, 42], also analyze traffic through a network, but with the key difference that this analysis occurs over time rather than in a steady state. Edges in the network are defined by both a capacity and a travel time, and the goal is to maximize flow to the sink vertex within some amount of time T. Ford and Fulkerson provide an algorithm for solving this that identifies bottlenecks in the network (both when and where they occur) and creates a small set of actions, referred to as chain-flows, that are repeated until time T and which lead to the maximal flow. This work was extended by Edmonds and Karp [39], who point out particular situations in which the original algorithm

struggles and then provide simple modifications to avoid these situations.

Traditional work on network flows considers the problem in one of two contexts: *discrete time* in which networks are time-expanded (i.e., copied for each time step with edges added between time layers) or *continuous time* in which capacities and costs can change over time. Fleischer and Tardos [43], meld these two research tracks by extending some of the polynomial time algorithms from the discrete setting to the continuous-time setting. This work, like that of Ford and Fulkerson, allows for the storage of traffic at nodes, a luxury not afforded to a model meant to model a physical system like a vehicular road network.

As mentioned previously, much work in dynamic flows is done through the time expansion of networks, a method by which a discrete time system can be represented in a single, static graph. While this allows for the use of standard static graph algorithms, a non-trivial time range leads to a very large network. Fleisher and Skutella [44] seek to alleviate this by providing methods for condensing these time-expanded graphs. These condensed graphs discretize time at a lower resolution, are polynomial in size, and can be used to compute a solution of arbitrary precision in polynomial time.

2.3 Robotics and Artificial Intelligence

Let us now consider work in the robotics and artificial intelligence communities. In much of this work, vehicles communicate either with one another or with a local controller that allows vehicles to pass through an intersection simultaneously if it

can be ascertained (perhaps with a small adjustment in velocities) that the motion is collision-free (see, e.g., [45]). Even though such systems may be beyond the present-day automotive technology, the approach can be applied to controlling the motion of parcels and vehicles in automated warehouses [46].

The work by Fiorini and Shiller on velocity obstacles [47] considers motion coordination in a decentralized context, in which a single agent is attempting to avoid other moving objects. These obstacles have a predefined motion which is combined with the agent's possible velocities to form a *velocity obstacle*, a set of velocities for the agent that lead to an inevitable collision (this assumes that the motion of the agent does not change). To avoid a collision, a velocity outside of this set is chosen. Furthermore, the selected velocity is restricted by considering limitations on the agent itself, such as acceleration limits.

To generalize for obstacle trajectories, this same computation occurs at repeated regular intervals. By searching a tree of these feasible velocities, a near-optimal collision free path can be found for the agent from the start location to the goal location. Additionally, a heuristic search can be used to satisfy a prioritized list of objectives in an efficient manner.

The concept of velocity obstacles introduced above is extended by Van den Berg et al. [48, 49] by considering the behavior of other agents. Rather than treating them as passively moving obstacles, this formulation views them as reactive agents making similar collision-avoidance decisions. This type of dual motion planning without communication may lead to oscillations as agents continually try to predict and react to each other, but the authors show that their method guarantees safe and

oscillation-free motions for the agents. They go on to demonstrate that the concept is scalable and performs well in real-time by simulating hundreds of agents in densely populated environments containing static and dynamic obstacles.

Akella and Hutchinson's work [50] deals with creating collision free plans for robots which have overlapping workspaces. The trajectories of the robots are known beforehand (these are often constrained by the work being performed, e.g., a spray-painting robot must follow a specified velocity profile in order to appropriately paint a target). Collision zones are determined for robot motion and linear constraints are derived and the problem is then solved through the use of an Integer Programming (IP) software package. The paper suggests that the problem is NP-Complete given the use of IP, but no formal proof is given.

Given that these works focus on real-world robotics issues, it may be the case that the time necessary to create a motion plan is greater than the time one has in which to execute the resulting plan. It may be that the robot needs to act quickly, lest the information at hand become outdated, rendering any conceived motion plan invalid. To work around these constraints, Petti and Fraichard introduce the concept of a *Partial Motion Plan* (PMP) [51]. Essentially, the system will form as much of a motion plan as possible in the allotted time. Furthermore, *Inevitable Collision States* (ICS) are identified so that no PMP ends leaving the robot in a position such that the next PMP is unable to advance the robot toward its goal.

Rodriguez et al. [52] provide a framework for planning motion among moving obstacles that considers two types of obstacle: hard and soft. A hard obstacle is one with which a collision is strictly prohibited. A soft obstacle, on the other hand, is one

where some "collision" is allowed. The example they present is one of a pedestrian with a safety margin around it. Modeled as a disc, it is highly preferred that no collisions occur with this disc, but it is not strictly necessary. Motion planning is done as a two-step process. First, a roadmap is created using Probabilistic Roadmap Methods or PRM [53]. This roadmap is created around the hard obstacles only, providing the means to find a motion plan that is collision free with respect to these obstacles. The second step alters this plan to take into account limitations on the robot's movement (e.g., acceleration limits) and to attempt to avoid the soft obstacles. While the formal complexity of this method is not analyzed, experimental results are promising, in so far as computation time and soft obstacle avoidance metrics are concerned.

Yu and LaValle [4] show that multi-agent path planning on graphs can actually be reduced to network flow problems. Additionally, they prove that when the problem's goals are permutation invariant (that is, agents do not have a specific goal to reach, rather each goal location must be reached by a unique agent), there is always a solution with a finish time upper-bounded by $n + V - 1$, for n agents and V vertices on the graph. An algorithm for finding such a solution in $O(nVE)$ is shown, where E is the number of edges in the graph. This permutation invariant problem is, in essence, the escape problem, in which multiple agents are attempting to escape some space (generally represented as a graph) by reaching any one of multiple exits (marked as goal vertices) without concern for which exit they reach.

2.3.1 Game Theory

Additionally, there has been some work which focuses on the game theoretic concerns which occur in motion planning problems, often times viewing the problem in the context of finding strategic equilibria. For example, imposing the need for a Nash equilibrium in any selected strategy helps to ensure that drivers will follow it, as it is within their individual best interests to do so. This not only helps to guarantee the safety of a system (as there is no incentive to "go rogue"), it also implies some level of quality in the chosen strategy.

The work by LaValle and Hutchinson uses Nash equilibria for the coordination of multiple robots in the same space [54]. Their approach can be considered somewhere between centralized planning and decoupled planning. A roadmap is computed for each independent robot (decoupled), and the paths and motions of the robots on the roadmap are coordinated (centralized). Rather than focusing on an overall optimization strategy, LaValle and Hutchinson show that there is a partial ordering among all viable strategies. This ordering yields a set of maximal strategies, that is, for all strategies not in this set, there is another strategy in this set that is clearly equal to or better than it. Given this set, further criteria may be used to select one of the maximal strategies. As an indication that these strategies provide an outcome that is acceptable to all (that is, no agent's well-being is sacrificed in the name of optimization) it is shown that the strategies in this maximal set are also Nash equilibria. In other words, it will never be in an agent's best interest to deviate from these strategies.

Nisan et al. [55] introduce a traffic-crossing-type problem in the context of game theory. In it, two vehicles have arrived at an intersection simultaneously and must decide when to cross. If both vehicles cross at the same time, they each suffer a large penalty. If only one crosses, that vehicles receives some small payout while the vehicle that waits receives nothing. The concept of a *correlated equilibrium* [56] is introduced in which a third party dictates the moves of the players with the restriction that the strategies provided by the coordinator must be stable strategies, that is, players will find that these strategies are best for them when acting in their own self interest and will have no incentive to act differently.

Koch and Skutella [57] apply game theory to network flow problems. They do so by introducing the concept of a routing game over time, in which a player determines a route from a source node to a sink node, given a network and a start time, with the goal of minimizing the cost of reaching the sink node. The authors analyze Nash equilibria in this setting, particularly by showing that they can be characterized through a series of static flows with special properties. Finally, it is shown that these Nash equilibria are optimal and can be computed in polynomial time.

Kopparty and Ravishankar [58] provide a framework for pursuit evasion games. In such games, a set of n pursuers are attempting to capture a single evader. Movement occurs either simultaneously or in turns, is restricted by a maximum velocity and the possible inclusion of obstacles, and is restricted to a space in $\mathbb{R}^n$, which may or may not be bounded. While there is not a direct connection between pursuit and evasion games and the motion and traffic problems in this work, the

framework and resultant algorithms provide insight into effective collaboration and coordinated motion.

In work by Carlino et al. [59], they present an auction based system in which each vehicle waiting at an intersection bids money so that the lead vehicle in their lane may be the next vehicle to cross through the intersection. Rather than allowing the system to be dominated by wealthy drivers, auctions are regulated by a "benevolent system agent." Empirical evidence lends mild support to the efficacy of this method, though in general it seems that the equalizing effect of the system agent renders all vehicles essentially equal in priority and so the performance of the auction system is very close to that of a first-in, first-out policy.

2.3.2 Multi-Agent Systems and Traffic

Multi-Agent Systems research focuses on multiple intelligent agents interacting with each other in a shared environment. Each of the works below take a multi-agent systems approach to *autonomous intersection management* (AIM).

In a model by Dresner and Stone [60], cars approach a 4-way intersection at roughly the same speed and must make their way to the opposite side without collision. They propose a reservation-based system in which agents announce to an intersection controller their arrival time, velocity, direction of travel, maximum velocity, acceleration limits, length, and width. The intersection itself is divided into an $n \times n$ grid of tiles. The information provided by the vehicle allows the intersection controller to determine which of these tiles will be occupied and when.

If these tiles are not already reserved at those times, the controller will confirm the vehicle's registration. It is then up to the vehicle to follow the reservation or to send a cancellation message in the event that it cannot. If the reservation is canceled by the vehicle or rejected by the controller the vehicle must continue to make reservation requests as it approaches the intersection, decelerating as it does so to avoid entering the intersection without a reservation. Using a simulator designed by the authors, it is shown that this reservation-based system leads to an average delay for vehicles that is two to three hundred times shorter than the delay incurred by traffic lights.

Dresner and Stone [61] go on to extend their previous model by allowing vehicles to accelerate through intersections as well as turn within them. Additionally, they alter the communication protocol such that the vehicles do not need to know anything about the intersection control policy. In this system, vehicles will, as before, request a reservation based on their current state (e.g., velocity, time of arrival, etc.), but the intersection controller has the option of making a counter-offer. This counter-offer is a new reservation that the vehicle must either adhere to or must request the cancellation of (if, for example, it is unable to comply). This allows the controller to coordinate the vehicles in any manner it sees fit without the vehicles needing to understand the underlying logic, generalizing the protocol to encompass the reservation system, traditional traffic lights, stop signs, etc. Greater detail is provided in [45].

Similar to the reservation system used above, work by VanMiddlesworth et al. [62], uses a reservation-based system for prioritizing vehicle motion through the intersection. However, rather than requiring a centralized intersection controller,

this work provides a peer-to-peer protocol. This protocol requires vehicles to lay claim to the routes they will use to pass through the intersection. A priority ordering is established based on the vehicles' states, so that when conflicts occur it is clear which vehicle must alter its claim. This peer-to-peer system is intended for use at intersections with low traffic density as these types of intersections make up the majority of those in the real-world and the lack of a centralized intersection controller makes this solution much cheaper to implement.

In a more theoretic vein is the work by Wang and Botea [63] which attempts to straddle the gap between a centralized and a distributed system, thereby gaining efficiency while maintaining good scalability. This work identifies classes of multi-agent path planning problems that can be solved efficiently (i.e., in polynomial time) using MAPP, a multi-agent path planning algorithm that is theoretically tractable and scales well with the size of the set of moving agents, but is meant for problems on undirected graphs rather than in continuous space.

2.4 Real-world Traffic Management

There exists a large body of work studying real-world traffic systems. Of particular note is work that focuses on intersection management, traffic light policies, and natural systems related to vehicular traffic. While many of these works concern themselves with real-world issues, they still provide insight into the behavior of systems that deal with multiple dynamic agents. Inspiration can be drawn from this, and general concepts, terminology, and existing solutions may be helpful in formally

defining, analyzing, and finding solutions in the theoretic framework put forward in this dissertation.

Sasaki and Nagatani [64] analyze the efficiency of a traffic-light-regulated intersection on a single-lane roadway. In their analysis, three separate traffic light policies are explored and their effects on the density of the roadway are analyzed (dependent on different starting densities). The first policy is a simple synchronized strategy in which all traffic lights change simultaneously. The second, known as the green wave strategy, causes each light to change with a certain delay after the previous light changes. In this way, a green light will propagate down the road. The third policy tested was a random switching strategy which would randomly select a cycle time for each light. For vehicle behavior, the authors use the *optimal velocity model* [65], a model in which vehicles will accelerate to some optimal speed based on the speed limit and the room in front of the vehicle. Through simulation, the authors discover that saturation of current (that is, the point at which there is maximum throughput of traffic) occurs at some critical density. The value of this saturation (i.e., throughput) is not directly dependent on the traffic light strategy chosen. Instead, the critical density changes as a function of traffic light strategy, with cycle time being perhaps one of the most influential factors.

Gershenson uses a multi-agent simulation in order to study the efficiency of three different self-organizing methods of steering traffic through a city to minimize congestion [66]. With a simple rule set and no communication between agents, these methods outperformed traditional methods. The first method keeps count of the vehicles approaching the red light. Once a threshold is reached, the light changes.

This threshold will naturally create platoons of vehicles as they build up at each red light. The second method is similar to the first, but with some minimum time threshold, preventing the light from oscillating between states too quickly. In the final method an extra constraint is added to help keep platoons together. If a platoon is passing through a green light, the light will not change until the platoon is safely through, regardless of what other conditions are met. Each of these three methods adapt themselves to the changing traffic patterns and, through empirical evidence, the authors have shown that they outperform standard traffic light policies. Cools, Gershenson, and D'Hooghe extend this work to a realistic setting by incorporating data from a real Brussels avenue into the simulation [67].

A paper by John *et al.* [68] draws an analogy between a unidirectional flow of ants on a trail with that of traffic. This paper reports on experimental results, pointing out differences in vehicular traffic flow and streaming ants, specifically pointing out why it is that traveling ants never seem to suffer from traffic jams the way vehicles do. First, ants tend to move together in clusters (the paper refers to them as platoons) in which the relative velocity of the ants with respect to one another is quite low. Second, ants never speed up in order to overtake another ant. Third, the speed at which ants travel appears to be independent of the density of ants on the trail, a trait that differs greatly from vehicular traffic.

Lammer and Treiber [69] propose a traffic light policy similar to that of Gershenson in the sense that it involves an adaptive system meant to minimize congestion. However, traffic lights in this system attempt to be proactive rather than reactive. In Gershenson's system, the lights change based on how many vehicles

are approaching it. In Lammer and Treiber's system, the lights will change based on how many cars are on the roadway beyond it, allowing fewer through if the next intersection is being overcrowded. One nice property of this approach is its localized nature. By not requiring global knowledge, it may be the case the overall computational complexity is relatively low.

2.5 Warehouse Management

The sources thus far have mainly dealt with movement in shared spaces, but perhaps even more fundamental is the question of how much fits into a space and how best to utilize this limited resource. There have been a number of papers devoted to the problem of organizing storage units in real-world warehouses, although much of the prior work has focused on solving the various engineering challenges involved.

For example, Amato et al. [70] study control algorithms for warehouse robots, assuming a continuous distribution of item locations throughout the warehouse and ignoring the benefits of intelligent item placement. In a similar vein, Chang et al. [71] attempt to minimize unnecessary task repetition using genetic algorithms, thus shortening robot travel times, but assume a fixed storage scheme regardless of differing access frequencies. Sarrafzadeh and Maddila [72] use a discrete grid-based model, as is done in this work, but their focus is still an engineering one, concerned primarily with robot path-finding and constructing clearings through which to move. Closer to the work herein, Pang and Chan [73] address the question of where certain items should be stored in the warehouse, proposing a data-mining

approach to determine the relationships between products and co-locating those that are often purchased together. Experimental analysis shows that their methodology outperforms a simple greedy policy, but they do not present any formal proofs on the performance of their approach.

The word "warehouse" has been used for various optimization problems. In the context of operations research, the *warehouse problem* was proposed by Cahn [74] and later refined and extended by Charnes and Cooper [75] and Wolsey and Yaman [76]. This may sound related to the work done in this dissertation, but its focus is on the logistics of managing a warehouse's stock in the face of changing demand. The word is also used in the context of coordinated motion planning under the name of the *warehouseman's problem*. As mentioned in Section 2.1.3, this is a multi-agent motion planning problem amidst obstacles. It has been shown to be PSPACE-hard [33, 77], but efficient solutions exist for restricted versions (see, e.g., [31]).

While the approach used in this dissertation is theoretic in nature, the high complexity of the warehouseman's problem is avoided by restricting the shapes of boxes (to unit squares) and the allowed layout of boxes (by introducing additional empty working space throughout to facilitate easy motion). The problems studied in Chapter 5 are less focused on motion planning and more on how to organize the warehouse's contents to ensure efficient processing of a series of access requests.

More closely related to this work, however, is the *dial-a-ride problem* [78]. In this problem, a set of users must be conveyed from source locations to specified destinations in a metric space. The goal is to plan a route (or routes, in the case of multiple vehicles or the more general k-server problem [79]) that satisfies all

transportation requests while minimizing total distance traveled. One key difference is that the source locations are fully specified by the problem input, whereas in the warehouse problem presented in Chapter 5 the location of requested boxes can be adjusted according to need, and how best to do so is central to the problem.

Additionally, while packing problems are well studied, both as static [80, 81] or online problems [82], the focus of this work is more on reducing retrieval times by organizing storage units based on their online access frequency and is similar in spirit to online algorithms for self-organizing memory structures [83, 84]. Further details on this connection can be found in Chapter 5.

Chapter 3: On the Complexity of an Unregulated Traffic Crossing

3.1 Introduction

As autonomous and semi-autonomous vehicles become more prevalent, there is an emerging interest in algorithms for controlling and coordinating their motions to improve traffic flow. The steady development of motor vehicle technology will enable cars of the near future to assume an ever-increasing role in the decision making and control of the vehicle itself. In the foreseeable future, cars will have the ability to communicate with one another in order to better coordinate their motion. This chapter considers two algorithmic formulations of a simple and fundamental geometric optimization problem involving coordinating the motion of vehicles through an intersection.

Traffic congestion is a complex and pervasive problem with significant economic ramifications, costing drivers in the United States over $305 billion in 2017 alone [85]. Practical engineering solutions will require consideration of myriad issues, including the physical limitations of vehicle motion and road conditions, the complexities and dynamics of traffic and urban navigation, external issues such as accidents and break-downs, and human factors. This chapter is focused on an algorithmic problem, called the *traffic-crossing problem*, that involves coordinating the motions of a set of vehicles moving through an intersection. In urban settings, road intersections are *regulated* by traffic lights or stop/yield signs. Like an asynchronous semaphore,

a traffic light locks the entire intersection preventing cross traffic from entering it, even when there is adequate space to do so. Some studies have proposed a less exclusive approach in which vehicles communicate either with one another or with a local controller that allows vehicles, possibly moving in different directions, to pass through the intersection simultaneously if it can be ascertained (perhaps with a small adjustment in velocities) that the motion is collision-free (see, e.g., [45]). Even though such systems may be beyond the present-day automotive technology, the approach can be applied to controlling the motion of parcels and vehicles in automated warehouses [46].

Prior work on autonomous vehicle control has generally taken a high-level view (e.g., traffic routing [1–4]) or a low-level view (e.g., control theory, kinematics, etc. [5, 6]). Between these extremes, there has been a great deal of work on decentralized models of crowd motion, including methods based on velocity obstacles [47], reciprocal collision avoidance [49], and implicit crowds [86]. Much closer to the approach presented in this chapter is the work on *autonomous intersection management* (AIM) [45, 59–62, 87, 88]. This work, however, largely focuses on the application of multi-agent techniques and deals with many real-world issues. As a consequence, formal complexity bounds are not proved.

Consider, instead, a simple problem formulation of the traffic-crossing problem, which captures the essential computational challenges of coordinating crosswise motion through an intersection. Vehicles are modeled as line segments moving monotonically along axis-parallel lines (traffic lanes) in the plane. Vehicles can alter their speed, subject to a maximum speed limit, but they cannot reverse direction

nor make turns. The objective is to plan the collision-free motion of these segments as they move to their goal positions.

After a formal definition of the traffic-crossing problem in Section 3.2, three results are presented. First, Section 3.3 shows that this problem is NP-complete. While this is a negative result, it shows that this problem is of a lower complexity class than similar PSPACE-complete motion-planning problems, like sliding-block problems [33]. Second, in Section 3.5 a constrained version is considered in which vehicles travel vertically at a fixed speed. This variant is motivated by a scenario in which traffic moving along one axis (e.g., a major highway) has priority over crossing traffic (e.g., a small road). An algorithm based on plane-sweep is presented that solves this problem in $O(n \log n)$ time. Finally, the problem is considered in a discrete setting in Section 3.6, which simplifies the description of the algorithms while still capturing many of the interesting scheduling elements of the problem. As part of this consideration, a solution to the problem is provided that limits the maximum delay of any vehicle and it is proven that this solution is asymptotically optimal.

3.2 Problem Definition

The *traffic-crossing problem* is one in which several vehicles must cross an intersection simultaneously. For a successful crossing, all vehicles must reach the opposite side of the intersection without colliding, and they must do so in a reasonable amount of time. This time-based restriction exists to encourage an improvement in

efficiency over the traffic light regulated crossing. Here, a "reasonable amount of time" is short enough that the traffic cannot simply take turns crossing the intersection (i.e., using the manner in which a traffic light regulates intersections) but instead forces some amount of simultaneity.

This problem can be posed either as one of optimization (how quickly can all the cars get across without colliding) or as a decision problem (can all vehicles cross, collision-free, within a particular time limit). Here, it is treated as a decision problem so that its parallels with other hard decision problems, such as Boolean satisfiability, can be more easily illustrated.

Formally, an instance of traffic crossing is defined as a tuple $C = (V, \delta_{max})$, comprised of a set V of n *vehicles*, which exist in $\mathbb{R}^2$, and a positive real δ_{max}, which represents a *global speed limit*. Each vehicle is modeled as a vertical or horizontal directed line segment that moves parallel to its orientation. It will simplify matters to assume that these segments are open. Like a car on a road, each vehicle moves monotonically, but its speed may vary between zero and the speed limit. Assume that vehicles do not make turns. Even with this no-turn assumption, it is possible to capture many of the complexities of managing traffic at an intersection and, as will be shown later, the problem remains computationally difficult. A vehicle's position at any time is specified by the location of its leading point (relative to its direction).

Each vehicle $v_i \in V$ is defined as a set of properties, $v_i = \{l_i, p_i^{\vdash}, p_i^{\dashv}, t_i^{\vdash}, t_i^{\dashv}\}$[1], defined as follows (see Figure 3.1(a)):

- l_i: The length of the vehicle's line segment.
- $p_i^{\vdash}$: The starting position of the vehicle, i.e., the vehicle's position prior to its start time (see below). The position is defined as a point and represents the leading edge of the vehicle.
- $p_i^{\dashv}$: The goal position of the vehicle. The vehicle is considered to have successfully reached its goal if its leading point reaches this position either on or before its deadline (see below).
- $t_i^{\vdash}$: The starting time of the vehicle. The vehicle may not move prior to this time.
- $t_i^{\dashv}$: The deadline for the vehicle. This is an absolute point in time by which the vehicle must reach its goal position.

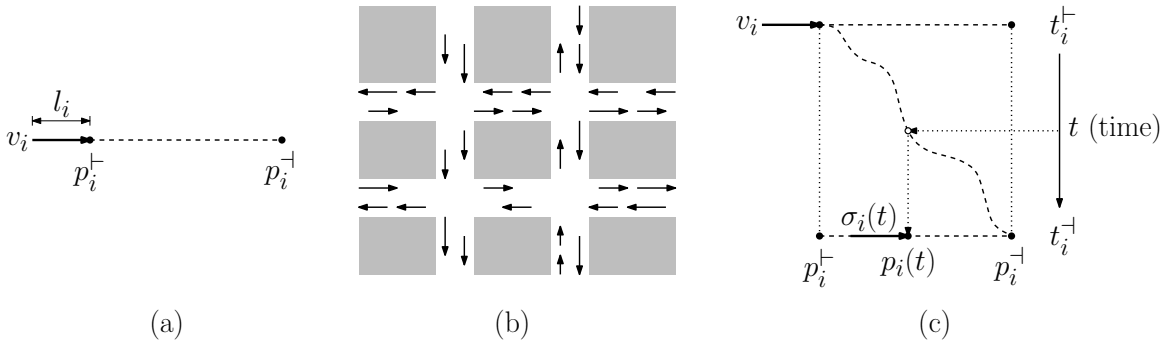


Figure 3.1: (a) Specification of a vehicle v_i, (b) modeling traffic as an instance of traffic crossing, and (c) position as a function of time.

[1]The notational use of $\vdash$ and $\dashv$ set above a variable (e.g., $\alpha^{\vdash}$) represent the beginning and end of a closed interval, respectively (e.g., start and end times).

The set V of vehicles and the global speed limit δ_{max} define the problem and remain invariant throughout. Planning the movement of a set of vehicles over a grid-based road network can be modeled as an instance of the traffic-crossing problem (see Figure 3.1(b)).

The objective is to determine whether there exists a collision-free motion of the vehicles that respects the speed limit and satisfies the goal deadlines. Such a motion is described by a set of functions, called *speed profiles*, that define the instantaneous speed of each vehicle as a function of time. This set of functions is defined as $D = \{\delta_i(t) \mid i \in [1, n], \forall t, 0 \leq \delta_i(t) \leq \delta_{max}\}$. A vehicle's direction of travel is a unit length vector d_i directed from its initial position to its goal. Given its speed profile, the position of a vehicle at time t is $p_i(t) = p_i^{\vdash} + d_i \left[\int_0^t \delta_i(x)dx\right]$, and the vehicle v_i inhabits the open line segment between $p_i(t)$ and $p_i(t) - d_i l_i$, which is denoted by $\sigma_i(t)$. The position as a function of time can be expressed as a graph (see Figure 3.1(c)). Formally, a set D of speed profiles is *valid* if it satisfies the following conditions for each vehicle $v_i \in V$:

- Stationary outside its time interval: $\forall t \notin \left[t_i^{\vdash}, t_i^{\dashv}\right],\ \delta_i(t) = 0$
- Satisfies the speed limits: $\forall t \in \left[t_i^{\vdash}, t_i^{\dashv}\right],\ \delta_i(t) \in [0, \delta_{max}]$
- Does not collide with other vehicles: $\forall t$ and $\forall v_j \neq v_i,\ \sigma_i(t) \cap \sigma_j(t) = \emptyset$
- Reaches its goal: $p_i(t_i^{\dashv}) = p_i^{\dashv}$

A traffic crossing instance C is *solvable* if there exists a valid set of speed profiles D.

For any vehicle v_i, the line segment swept out by the vehicle as its position

varies from $p_i^{\vdash}$ to $p_i^{\dashv}$ is called its *travel segment*. (Its length is $l_i + \|p_i^{\vdash} p_i^{\dashv}\|$.) The point at which two perpendicular travel segments meet is called an *intersection*. Collisions can arise in two different ways: a *T-bone collision* involves two vehicles with perpendicular orientations that collide at an intersection, and a *rear-end collision* involves two collinear vehicles with equal orientations, where the following vehicle overtakes the leading vehicle.[2]

While this definition places no restriction on the speed profile functions, it will be convenient to assume that they have a simple form in which each speed profile is piecewise constant, alternating between zero and δ_{max}. Furthermore, it may be assumed that vehicles stop for one of three possible reasons. First, they may be stopped because the current time is outside the vehicle's time interval, $\left[t_i^{\vdash}, t_i^{\dashv}\right]$. Second, the vehicle may be waiting just prior to entering an intersection. Third, it may be part of a chain of collinear vehicles, where the first vehicle in the chain is stopped for one of the two previous reasons. Such a speed profile is said to be *binary*. A solution is *binary* if all its speed profiles are binary. It is not hard to see that, through the introduction of auxiliary stopping points, there is no loss in generality in assuming that a valid solution is binary, but for the sake of completeness, a formal proof is provided in Section 3.4.

[2]There is a third possibility, namely a *head-on collision*, which involves two collinear vehicles with opposite orientations. It is easy to see, however, that this occurs if and only if the input contains two overlapping collinear travel segments of opposite orientation. Such inputs are simply forbidden, since they are not solvable.

3.3 Hardness of Traffic Crossing

This section shows that determining whether an instance of the traffic-crossing problem is solvable is NP-complete. In Section 3.3.7 it will be proven that the problem is in NP, but first hardness is established through the following reduction.

Lemma 3.3.1. *Given a Boolean formula F in 3-CNF, there exists a traffic crossing $C = (V, \delta_{max})$, computable in polynomial time, such that F is satisfiable if and only if there exists a valid set of speed profiles D for C.*

Throughout the reduction, it will be assumed that time and distances are scaled so that δ_{max} is one unit distance per unit time. The input to the reduction is an instance of 3-SAT, that is, a Boolean formula F in 3-CNF. Let $\{z_1, \ldots, z_n\}$ denote its variables and $\{c_1, \ldots, c_m\}$ denote its clauses. A key element of the reduction is the manner in which variable truth assignments are modeled by vehicle speed profiles. In light of the remarks at the end of Section 3.2, speed profiles are assumed to be binary.

Before presenting the reduction, it is helpful to begin with an explanation of the convention used to illustrate motion over time in the figures. The presented mechanisms will constrain each vehicle's motion to one of two types: moving to the goal at full speed without delay or delaying exactly one time unit and then moving at full speed to the goal. The latter form of movement will be visualized by drawing the vehicle one unit of length *behind* its starting position (see vehicle v_2 in Fig. 3.2(a)). Otherwise, the vehicle will be drawn at its natural starting position. This way, one

can visualize either situation as the vehicle moving at full speed from its natural starting time from this *adjusted starting position*.

To see why this convention is used, consider two vehicles v_1 and v_2 moving along perpendicular travel segments as shown in Fig. 3.2(b), where v_1 is moving downwards and v_2 is moving to the right. Consider a diagonal strip at a 45° angle projected from v_1 towards v_2 relative to their adjusted positions described in the previous paragraph (shown in blue in the figure). Observe that if the two vehicles start at the same time from their adjusted positions and move at the same speed, they will collide if and only if v_2 overlaps this strip. Thus, in order to create ensembles of collision-free vehicle motions, it suffices to avoid such overlaps.

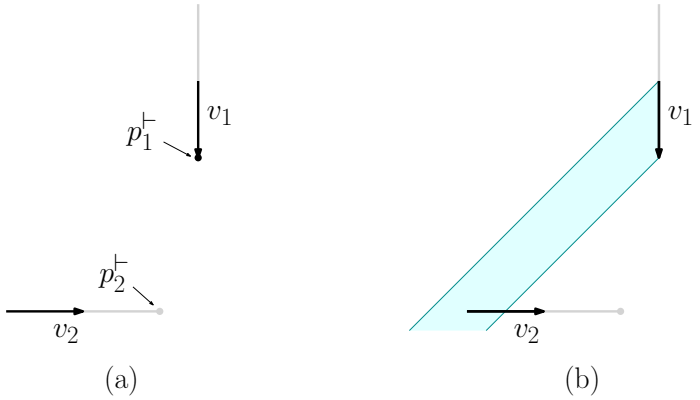


Figure 3.2: (a) This dissertation's convention of illustrating vehicles, where v_2 has been displaced behind its starting position to indicate its initial delay by one time unit. (b) Vehicle v_2 lies within the diagonal strip projected from v_1 if and only if these two vehicles will collide in the future.

3.3.1 Variable Representation

This section resumes the presentation of the reduction. Each variable z_i in the Boolean formula F is represented by a pair of vehicles whose motion encodes the

variable's truth value. The vehicles of this pair, referred to as *value vehicles*, travel downward in a coordinated manner along two vertical lines that are separated by a distance of one unit.

As mentioned above, the system is designed so that vehicles can move in essentially just two ways in any valid solution: (1) proceed at full speed directly to the final destination, arriving one time unit before its deadline or (2) delay for one time unit and then proceed at full speed to the goal. These two movement types will be referred to as *delay-last policy* and *delay-first policy*, respectively. In order to force this behavior, additional *helper vehicles* are introduced.

Because the final mechanism is a bit complex, a simplified version is first introduced here, which operates under the restriction that each vehicle delays by exactly zero or one time unit. The simplified mechanism consists of two downward-moving value vehicles v_i and v_i' and two additional right-moving helper vehicles u_i and u_i', each of unit length (see Fig. 3.3(a)). Consider which combinations of motion types are possible. There are four possible delay configurations for the value vehicles v_i and v_i'. First one of the two vehicles might delay and the other does not. In this case, the delays of the helper vehicles u_i and u_i' are staggered in a complementary manner to avoid collisions (see Fig. 3.3(b) and (c)). The first configuration where v_i delays corresponds to setting the variable z_i to **True** and the configuration where v_i' delays corresponds to setting z_i to **False**. Otherwise, both v_i and v_i' delay or neither does (see Fig. 3.3(d) and (e)). These cases are invalid, because one of the two helper vehicles cannot avoid a collision.

Unfortunately this simple construction is not correct when general vehicle

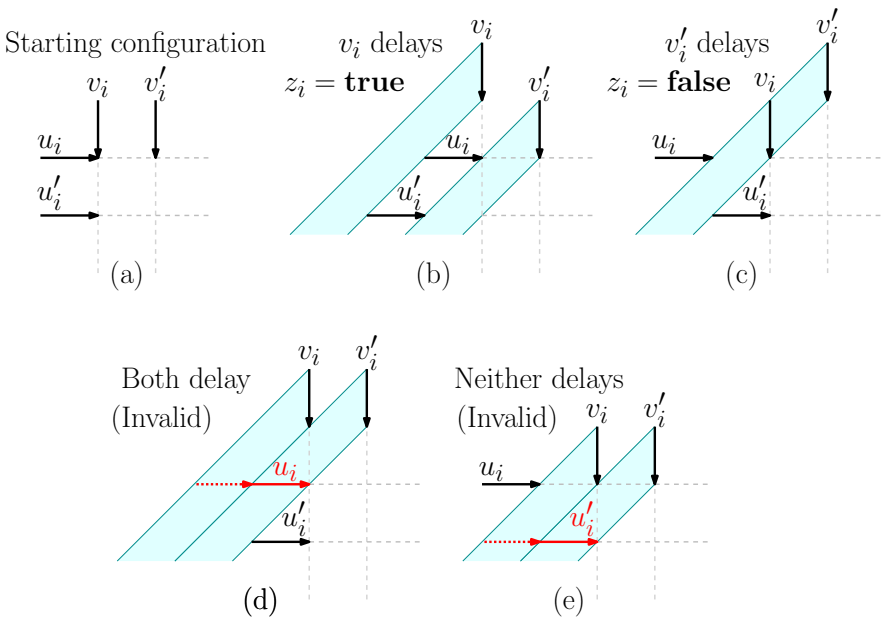


Figure 3.3: The simplified variable-setting mechanism.

motion is allowed. For example in the scenario where neither vehicle delays, it is possible for v_i' to enter the intersection and remain in the gap between u_i and u_i' for one time unit, which allows u_i' to avoid a collision.[3] To remedy the problem, a more complex mechanism can be employed that avoids this error. The full mechanism and its correctness is presented in Section 3.3.2. For now, it suffices to describe the remainder of the construction using the simple mechanism.

To represent all of the variables in $\{z_1, \ldots, z_n\}$ multiple instances of the mechanism described above are created, one for each variable, stacked vertically atop each other to form a single *variable stream* (see Fig. 3.4). The value vehicles' positions are initialized so that each member in a pair is collinear with the respective members of all other pairs of value vehicles. Additionally, the starting positions are spaced a distance of $s \geq 7$ units apart. This padding is to allow for the later insertion of

[3]The author is indebted to Nil Mamano and Michael Goodrich for pointing this out.

additional mechanisms used in the reduction.

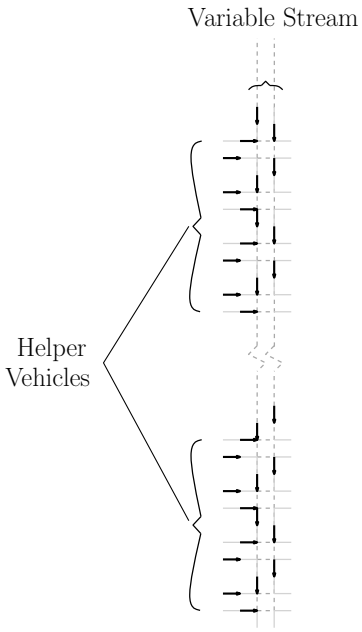


Figure 3.4: An example of value vehicles arranged into a variable stream representing four variables.

The variable stream is conceptually divided into blocks of length $s|V|$, long enough to accommodate all of the value vehicles and their requisite spacing. Every clause in F is associated with two of these blocks (one for the positive literals and one for the negative literals), requiring $2|C|$ such blocks (see Fig. 3.5). Two extra blocks are added, one at either end of the variable stream, to accommodate the initialization of the value vehicles with the helper vehicles. Truth values for the appropriate literals will be copied and transferred out of each block to a mechanism which adjusts their relative timing. This adjustment prepares the vehicles for a final mechanism that validates the satisfaction of the associated clause.

So, given a formula F with $|C|$ clauses and $|V|$ variables, each variable z_i is

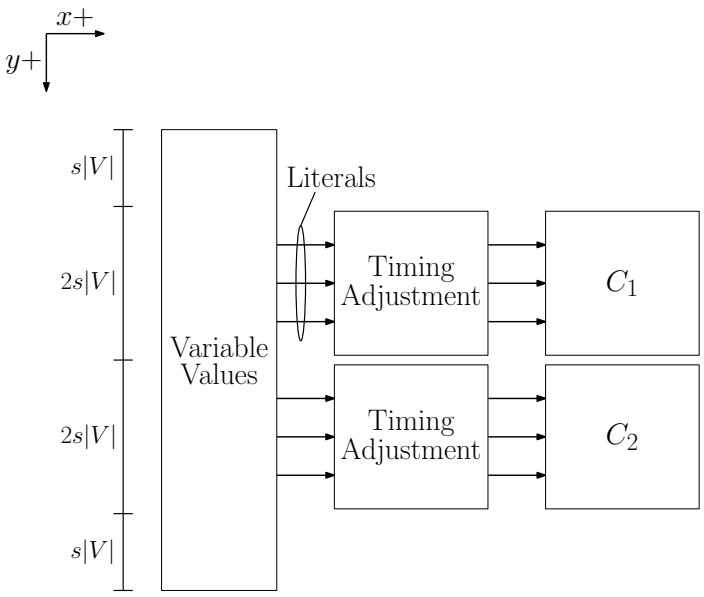


Figure 3.5: An overview of a reduction from 3-SAT to an instance of the traffic-crossing problem.

represented by a vehicle v_i with the following parameters:

$$p_i^{\vdash} = (0, si) \tag{3.1a}$$

$$p_i^{\dashv} = (0, 2s|V|(|C|+1) + si) \tag{3.1b}$$

$$t_i^{\vdash} = 0 \tag{3.1c}$$

$$t_i^{\dashv} = 2s|V|(|C|+1) + 1 \tag{3.1d}$$

In addition, the vehicle v_i' is created with similar parameters, but shifted one unit to the right.

3.3.2 Final Mechanism for Variable Representation

This section presents the detailed mechanism for representing each variable of the formula. Recall from the simplified mechanism given in Section 3.3.1 that

each variable z_i is represented by a pair of value vehicles, which travel downward in a coordinated manner along two vertical lines that are separated by a distance of one unit, and each can endure a delay in the interval $[0, 1]$. The movement of each pair of value vehicles is constrained by a pair of helper vehicles, which travel together horizontally, are separated vertically by the unit distance, and are placed so that they intersect the value vehicles' paths. Their goal positions, start times, and end times are all set so that they must interact with the value vehicles. All of these vehicles are of length $1 - \varepsilon$, for a constant $0 < \varepsilon < \frac{1}{2}$ to be specified later.

To correct the error mentioned in Section 3.3.1, for each value/helper vehicle group, an additional set of smaller vehicles is created whose purpose is to prevent other vehicles from stopping in the intersection. These vehicles are of length ε and are allowed no delay (by setting $t_k^{\dashv} - t_k^{\vdash} = \|p_k^{\dashv} - p_k^{\vdash}\|/\delta_{max}$). Thus, their motion is constrained to travel at full speed until reaching their goal positions. There is one such *ε-vehicle* for each of the vehicles (value and helper) described above.

More formally, one can define the requisite vehicles for representing a generic variable by first defining a reference point (x, y), and positive axes to the right and down for x and y, respectively. Let $(x, y - \varepsilon)$ and $(x + 1, y - \varepsilon)$ denote the positions of the leading points of a value vehicle pair (v_1, v_1') at time t (see Fig. 3.6). Next, define a value Δ as a function of the number of variables and clauses in the original Boolean formula and set the value vehicles' goal positions to $(x, y - \varepsilon + \Delta)$ and $(x + 1, y - \varepsilon + \Delta)$, respectively. The value of Δ is set so that the value vehicles travel far enough to accommodate the necessary gadgets for each clause.

Place a helper vehicle pair (u_1', u_1) at (x, y) and $(x, y + 1)$, respectively, set their

goal positions to $(x+2, y)$ and $(x+2, y+1)$, their start times to t, and their deadlines to $t+3$. Similarly, place another helper vehicle pair (w_1', w_1) at $(x, y+\Delta-1)$ and $(x, y+\Delta)$, respectively, set their goal positions to $(x+2, y+\Delta-1)$ and $(x+2, y+\Delta)$, their start times to $t+\Delta-1$, and their deadlines to $t+2+\Delta$ (here, Δ is added to account for the arrival time of the value vehicles).

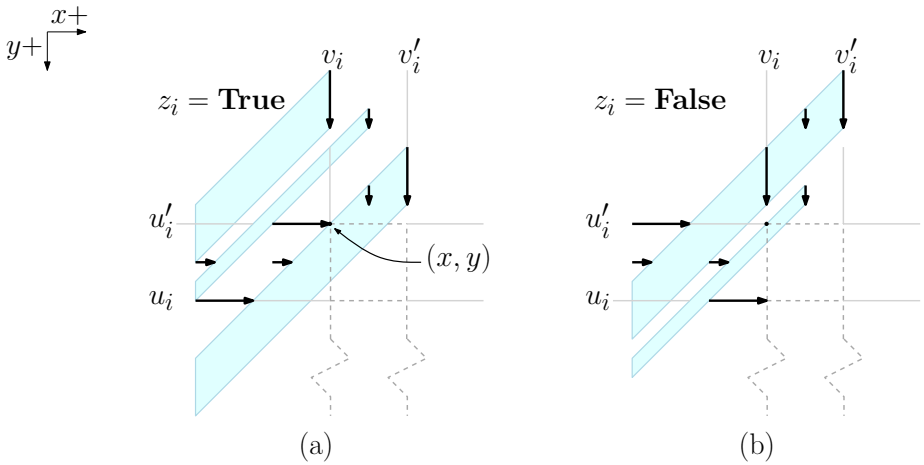


Figure 3.6: (a/b) Value vehicles taking on opposing values, allowing for valid paths for the helper vehicles. (See the remarks at the start of Section 3.3.1 on figure layouts.)

Finally, for each intersection between pairs of vehicles, four vehicles of length ε are created, one for each vehicle crossing the intersection. These ε-vehicles will prevent their matching vehicle (either value or helper) from delaying inside the intersection (see Fig. 3.7). For the value vehicles, place two ε-vehicles at $(x+0.5, y-0.5+\varepsilon)$ and $(x+0.5, y-1.5+\varepsilon)$, set their goal positions to $(x+0.5, y+2.5+\varepsilon+\Delta)$ and $(x+0.5, y+1.5+\varepsilon+\Delta)$, their start times to $t+\Delta$, and their deadlines to $t+3+\Delta$. Notice these deadlines allow for no delay in the motion of the ε-vehicles.

Similarly, for the helper vehicles (u_1', u_1), place two ε-vehicles at $(x-0.5, y+0.5)$ and $(x-1.5, y+0.5)$, set their goal positions to $(x+2.5, y+0.5)$ and $(x+1.5, y+0.5)$,

their start times to t, and their deadlines $t+3$. Two similar ε-vehicles are created for the helper vehicles (w'_1, w_1), only Δ units lower so that they lie between the vehicle pair.

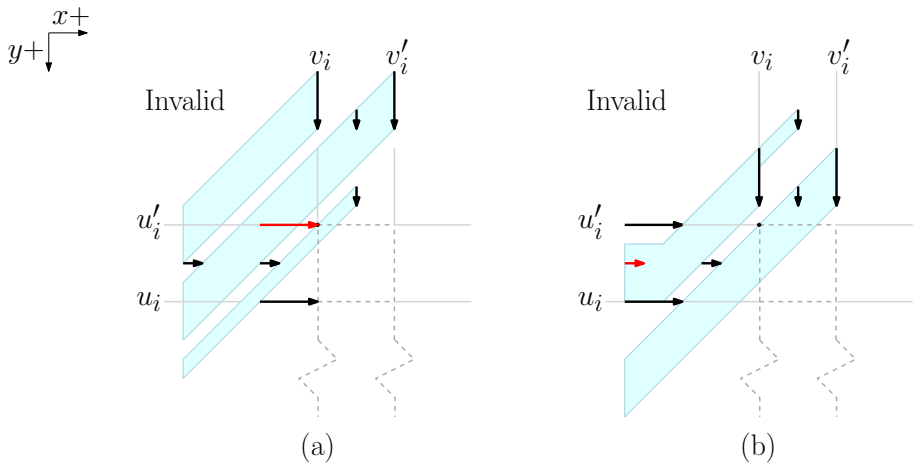


Figure 3.7: If both value vehicles select the same delay policy, then there is no valid speed profile for one of the helper vehicles, as in figure (a), or a collision occurs, as in figure (b) when v_i must delay to avoid u_i and collides with an ε-vehicle.

Lemma 3.3.2. *Given the pairs (v_1, v'_1), (u'_1, u_1), (w'_1, w_1), and the ε-vehicles as defined above, the value vehicles v_1 and v'_1 must each adopt one of the following two movement policies:*

(a) delay for exactly one unit of time and then move beyond the paths of (u'_1, u_1) at speed δ_{max} (i.e., delay-first); or

(b) move beyond the paths of (w'_1, w_1) at δ_{max} without delay (i.e., delay-last).

Additionally, v_1 and v'_1 may not select the same policy.

Proof. First, notice that for a value vehicle there are two perpendicular ε-vehicles with which it will interact. These vehicles, set between the helper vehicles, have no freedom of movement and so force the value vehicle to either delay for one time

unit before encountering them or to pass by them at full speed without delay (see Fig. 3.8). A corollary to this is the fact that the spacing between vehicles is tight, so that even a small delay before encountering the ε-vehicles requires the value vehicle to stop until its accumulated delay is exactly one time unit before it can successfully pass.

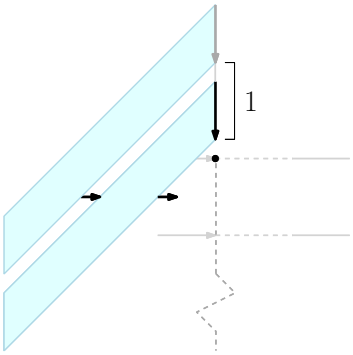


Figure 3.8: A value vehicle is limited to one of two movement options by the fully constrained ε-vehicles.

Second, if the value vehicle is to delay before reaching the ε-vehicles, it must actually delay before reaching the helper vehicles, as there is not enough room between the epsilon and helper vehicles in which to wait. Next, because v_1 and u'_1 begin within a vehicle's length of the point at which their paths cross, they will collide if neither one delays. Instead, they must choose different movement profiles so that one delays first, allowing the other to pass.

Finally, notice that a delay of v'_1 necessitates a similar delay of u'_1. This is because it takes one time unit for u'_1 to reach the point at which their paths intersect. If v'_1 was to delay one time unit yet u'_1 was to leave immediately, they would reach this point together and collide.

Given that u'_1 must delay if v'_1 does, and v_1 cannot enact the same movement

policy that u'_1 does, it must be the case that both value vehicles cannot choose to delay for one time unit at this point. A similar dependency exists between the value vehicles and u_1, though this dependency prevents v_1 and v'_1 from both leaving immediately and continuing forward at constant speed.

The logic above also holds for the second helper pair, (w'_1, w_1), constraining the value vehicles to opposing movement policies until they have moved beyond the paths of these helper vehicles. This also prevents the value vehicles from swapping movement policies, i.e., altering their Boolean values. To do so would require the lagging vehicle (i.e., the vehicle that adopted the delay-first policy) to speed up while the lead vehicle slows down. However, given the constraints placed on the vehicles, they are already traveling at the speed limit δ_{max}, so the lagging vehicle may not go any faster. □

3.3.3 Value Transmission and Timing

Henceforth, the full variable mechanism from Section 3.3.2 will be assumed, even though the figures will be based on the simple mechanism. For each clause, the three literal values will need to be carried to the appropriate clause mechanisms so that they arrive in the correct place at the correct time. This requires the introduction of two new mechanisms: one that copies truth values, and one that can adjust the timing of when a value reaches a particular location.

The first mechanism uses a pair of vehicles whose movement is constrained by a perpendicular pair of vehicles in the same way as the helper vehicles do. The

second mechanism uses a snaking path to induce a delay by increasing the distance traveled.

3.3.3.1 Value Duplication

In order to perform clause verification the variable values must be able to be transmitted freely around the space. To do so, a new pair of parallel vehicles is created, separated by a distance of one unit, whose purpose is to copy these values from the variable stream and carry them elsewhere. This pair is placed so that its starting position lies on the leftmost side of the variable stream, traveling to the right, and its start time $t_i^{\vdash}$ is the time at which the leading edge of the appropriate value pair reaches the vertical position of the uppermost vehicle (see Fig. 3.9). Just like the helper vehicles, each of these copy vehicles has their deadlines set so that they may delay for one time unit at most, and because of this, the vehicles become a negative copy of the original value vehicles, with the negation on top and the original variable value on the bottom. These values can continue to be copied in order to carry them through the traffic space, taking orthogonal turns each time a copy is made. Any copies along this path that travel vertically will carry the variable's value on the left and the negation on the right. Any horizontal copy carries the negation on top and the original value below.

Each of a clause's positive literal values will be copied off of the variable stream simultaneously. The negative literals are copied similarly. By chaining vehicle copies across the space, the literal values can be routed to any location as necessary.

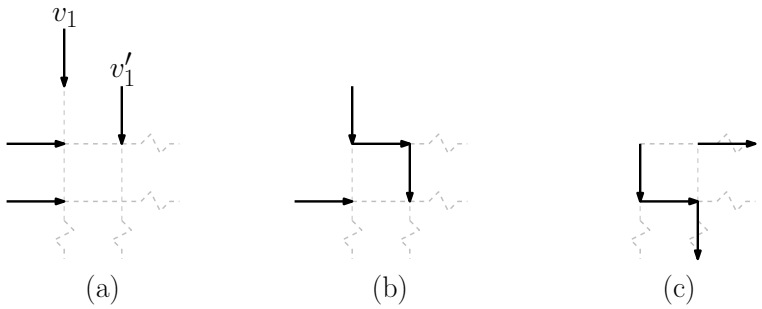


Figure 3.9: (a) An example of transferring a truth value at start time $t_i^{\vdash}$ for the copying vehicles (For the sake of legibility, ε-vehicles are excluded from this and future images). In this example, the variable z_1 is **True**, making v_1 and v_1' **True** and **False**, respectively. (b) At time $t_i^{\vdash} + 1$, notice that in the orthogonal copy the upper vehicle will take on the value of the negation while the lower vehicle takes the original value. (c) Time $t_i^{\vdash} + 2$.

3.3.3.2 Timing and Delays

The routing of values may require that they travel different distances to reach certain points. By the structure of the presented reduction, except when stopped, all vehicles travel at the same speed. Because of this, any difference in path length will cause a difference in timing that may need to be corrected. This is done through the introduction of a delay mechanism. This mechanism is inserted into the path of every copy coming off of the variable stream and can be configured to delay a vehicle pair's leading edge by an arbitrary amount. This delay does not affect the values carried by the vehicles. Essentially, the value is routed through an S shape in the mechanism, doubling back on itself (see Fig. 3.10). The size of this S determines the extra distance that must be traveled and thus the total amount of delay. A parameter d represents the extra distance added to the S in order to tune the mechanism, leading to a delay of $2d$ (as described below). Vehicle pairs are arranged in the mechanism as follows (see the heavy arrowed lines in Fig. 3.10), with the first and last referred

to as the *incoming pair* and *outgoing pair*, respectively:

- Eastbound at (x, y) and $(x, y+1)$ at time t,
- Southbound at $(x+2+d, y)$ and $(x+3+d, y)$, with a start time of $t+2+d$,
- Westbound at $(x+3+d, y+2)$ and $(x+3+d, y+3)$, with a start time of $t+4+d$,
- Southbound at $(x, y+2)$ and $(x+1, y+2)$, with a start time of $t+6+2d$,
- Eastbound at $(x, y+4)$ and $(x, y+5)$, with a start time of $t+8+2d$,
- Northbound at $(x+5+d, y+5)$ and $(x+6+d, y+5)$, with a start time of $t+13+3d$,
- Eastbound at $(x+5+d, y)$ and $(x+5+d, y+1)$, with a start time of $t+17+3d$.

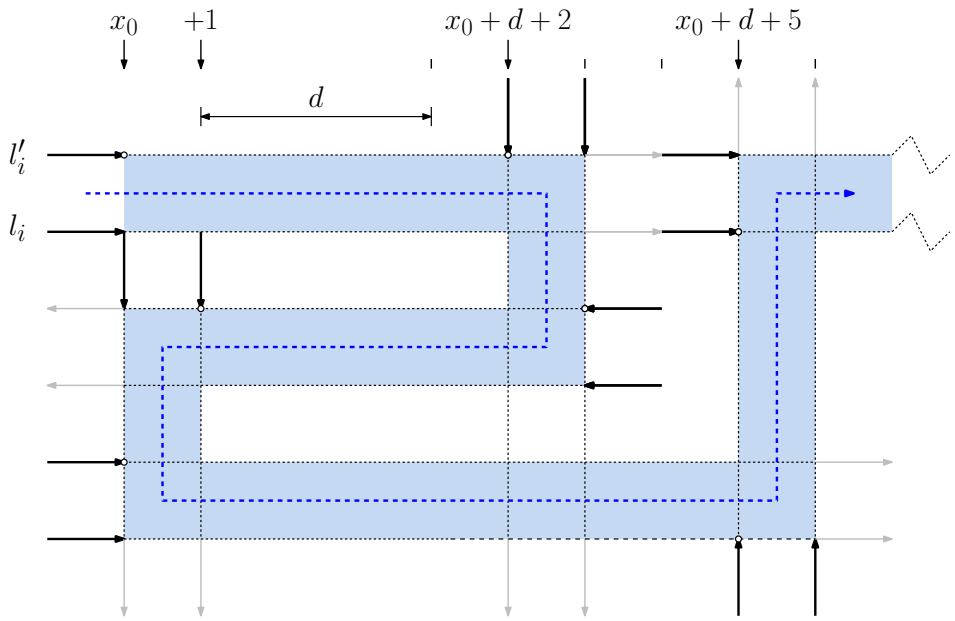


Figure 3.10: A diagram tracing the path a single pair of truth values (z'_i, z_i) take through the delay mechanism. The dotted vertical line represents where the mechanism can be expanded, separating the vehicles on either side by a distance of d and thus increasing the induced delay. Note: for clarity, ε-vehicles are not shown.

The distance between the incoming vehicle pair and the outgoing vehicle pair

is $5+d$, so, if the incoming pair were to continue on, both pairs would be in the same position at $t+5+d$. Since the outgoing pair starts at time $t+17+3d$, the mechanism induces a delay in the transmission of the incoming pair of $12+2d$. Adding the delay mechanism to all copies made from the variable stream enables the adjustment of the relative timing of each vehicle pair by adjusting the value of d in each delay mechanism.

3.3.4 Clause Satisfaction

This section demonstrates how clauses are modeled. For each clause $c_i \in F$, a mechanism is created that forces a collision if, and only if, all three literals are **False**. This mechanism checks the positive and negative literals separately, then combines the results in order to determine whether the clause is satisfied. The mechanism consists of two parts: one for the positive literals and one for the negative literals. Each part contains vehicle pairs representing the literals and their negations, blocking vehicles to appropriately constrain movement, and a verifying vehicle. If the set of literals do not satisfy the clause, each verifying vehicle is constrained to a single speed profile and they will collide. The following analysis will first look at the half that verifies the negative literals.

Define a point $r = (x, y)$ to be a reference point from which all other positions will be defined at a reference time t (see Fig. 3.11). Next, assume three pairs of incoming vehicles $(l'_1, l_1), (l'_2, l_2)$, and (l'_3, l_3), each a copy of the appropriate variables. These pairs travel horizontally, one unit apart vertically, with their leading edges 4

units behind the previous pair. Thus, the leading edges of the pairs are (x, y) and $(x, y+1)$, $(x-4, y+2)$ and $(x-4, y+3)$, and $(x-8, y+4)$ and $(x-8, y+5)$. Note that each pair of vehicles has a pair of ε-vehicles between them as defined previously.

Next, place two blocking vehicles, each of length 1, at $(x-2.5, y+1.5)$ and $(x-6.5, y+3.5)$. These vehicles have a start time of t, travel horizontally to the right, and have their deadlines set so that they must travel at δ_{max} with no delays. Finally, place a verifying vehicle v at $(x, y-\epsilon)$ with a start time of t and traveling downward. The deadline for the verifying vehicle is set so that it can delay up to 5 time units.

Lemma 3.3.3. *Given the vehicle pairs, blocking vehicles, and verifying vehicle defined above, the verifier must delay for* 5 *time units if all of the literals are* **False** *but may delay for less if at least one is* **True**.

Proof. First, notice that every horizontal vehicle in the mechanism is on a possible collision course with the verifying vehicle. Thus, if the slope of the line between one of these vehicles and the verifying vehicle has a magnitude of 1 (or if their positions are equal), the vehicle will collide with the verifying vehicle v if both continue without delay.

If each variable z_i is **True** then its negative copy l'_i is **False**, taking a delay-last movement policy. This places l'_1 at (x, y) and l_1 at $(x-1, y-1)$ at time t, which would lead to a collision with v. While l'_1 could still delay for one unit, l_1 no longer has this freedom as it has adopted the delay-first policy. Thus, to avoid a collision, v must delay for at least one time unit.

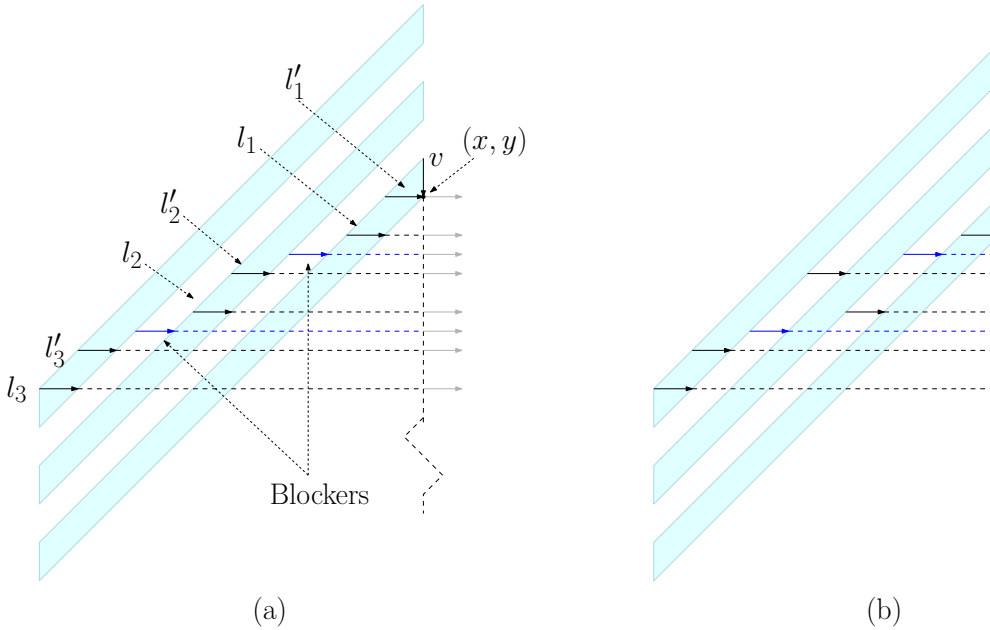


Figure 3.11: (a) The initialization of the negative half of a clause verifier for the clause $(\neg z_1 \vee \neg z_2 \vee \neg z_3)$ and with each variable $z_i =$ **True**. (b) The verifier with $z_2 =$ **False** and $z_1 = z_3 =$ **True**. For the sake of clarity, ε-vehicles have been omitted.

- At time $t+1$, the first blocking vehicle has moved to $(x-1.5, y+1.5)$. The blocking vehicles' deadlines allow for no delay, so again v must delay.

- At time $t+2$, l'_2 has moved to $(x-2, y+2)$ and l_2 has moved to $(x-3, y+3)$. Just as with l_1, v is forced to delay to avoid a collision.

- At time $t+3$, the second blocking vehicle is at $(x-3.5, y+3.5)$, forcing another delay of v.

- Finally, at time $t+4$, l'_3 has moved to $(x-4, y+4)$ and l_3 has moved to $(x-5, y+5)$, forcing one last delay of v.

Thus, if all of the variables z_i are **True**, making the negative literals l'_i all **False**, the verifying vehicle v must delay for 5 units of time in order to avoid a collision.

If any of the variables are **False**, their resultant copies l'_i and l_i will have shifted horizontal positions, no longer lying on the line of collision with v (i.e., their slopes are no longer of magnitude 1), allowing v to delay for less than 5 units and slip between them. □

The positive half of the mechanism works in the same manner, with slight changes to the incoming literal vehicles and some added vehicles to account for these changes. First, the incoming literal pairs are not staggered with respect to each other but instead arrive with collinear leading edges and one unit apart (see Fig. 3.12(a)). Next, a copy of each literal pair is made, traveling downward. The first copy pair is placed at $(x + 5, y - \epsilon)$ and $(x + 6, y - \epsilon)$ and has a start time of $t + 5$. The next pair is placed at $(x + 3, y + 2 - \epsilon)$ and $(x + 4, y + 2 - \epsilon)$ with a start time of $t + 3$. The third pair is placed at $(x + 1, y + 4 - \epsilon)$ and $(x + 2, y + 4 - \epsilon)$ and has a start time of $t + 1$.

Next, two blocking vehicles, each of length one and traveling downward, are added at $(x + 2.5, y + 9.5)$ and $(x + 4.5, y + 5.5)$, both with a start time of $t + 9$.

Finally, a verifying vehicle traveling to the right is added at $(x + 1, y + 12)$, with a start time of $t + 9$ and deadline allowing for a delay of up to 5 time units. As before, the vehicle will be forced to delay for 5 time units if the clause is not satisfied by any of the positive literals.

A clause will never have more than three literals, so it will never be the case that both the positive and negative halves of the clause verifier will have three literals. Blocking vehicles are added to take the place of missing literals in each half and

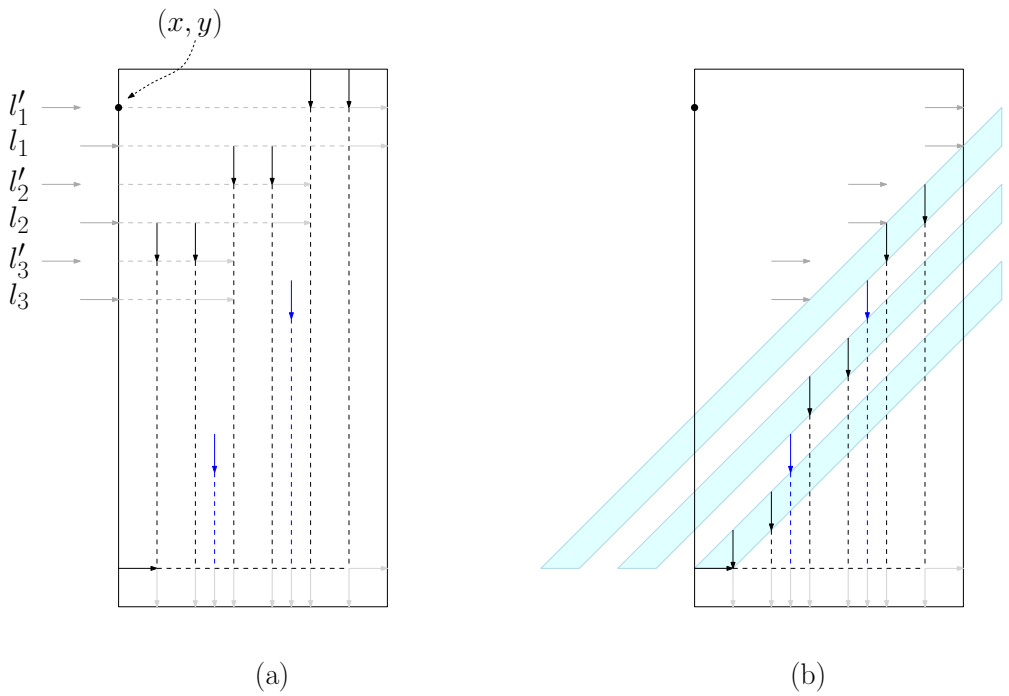


Figure 3.12: (a) The initialization of the positive half of a clause verifier for the clause $(z_1 \vee z_2 \vee z_3)$ and with each variable $z_i =$ **False**. (b) The verifier at time $t + 9$.

their deadlines are set so that no delay is allowed. In this way, the verifying vehicles are still forced to delay for 5 units when their associated set of literals do not satisfy the clause.

The positive and negative halves of the mechanism are placed so that the paths of the verifying vehicles intersect. However, the time at which each half processes its literals may differ, dependent on which variables are being evaluated and the distance their values must travel to reach the mechanism. This can be compensated for in the delay mechanisms so that the verifying vehicles will collide with one another if both delay for 5 time units. In this way, if a clause is not satisfiable, a collision is inevitable, rendering the traffic crossing unsolvable. If the clause is satisfiable, one or both of the verifying vehicles will have at least two movement options, allowing them to avoid a collision.

3.3.5 Complete System Example

In the complete system, all of the variables are stacked on top of one another to form a variable stream. The appropriate literals are extracted, passed through a delay mechanism, and routed to their clause verifier halves. These mechanisms output a vehicle that will have delayed for 5 time units if the variable assignments do not satisfy their respective clauses. The verifier vehicles from each clause will collide if neither set of literals satisfies them. An example of a 3-SAT reduction for the formula $(\neg z_1 \vee z_2 \vee \neg z_3)$ can be seen in Fig. 3.13.

3.3.6 Analysis of Reduction Complexity

Every variable in the formula F requires $12n$ vehicles: one for the variable, one for its negation, the two helper pairs, and an ε-vehicle for each of these. Next, when considering each of the m clauses, the greatest number of vehicles is necessary when all of the literals are positive. 27 are needed for the positive verifier, 15 for the negative verifier, 28 for each of the two delay mechanisms, and 24 for routing, for a total of at most 94 vehicles per clause. The complexity of translation is then $12n + 94m$ and is therefore on the order of $O(n + m)$.

As described above, the constructed mechanisms will only allow for a valid set of speed profiles if the formula F is satisfiable. Given this and the polynomial time needed to create the reduction, the traffic-crossing problem is NP-hard.

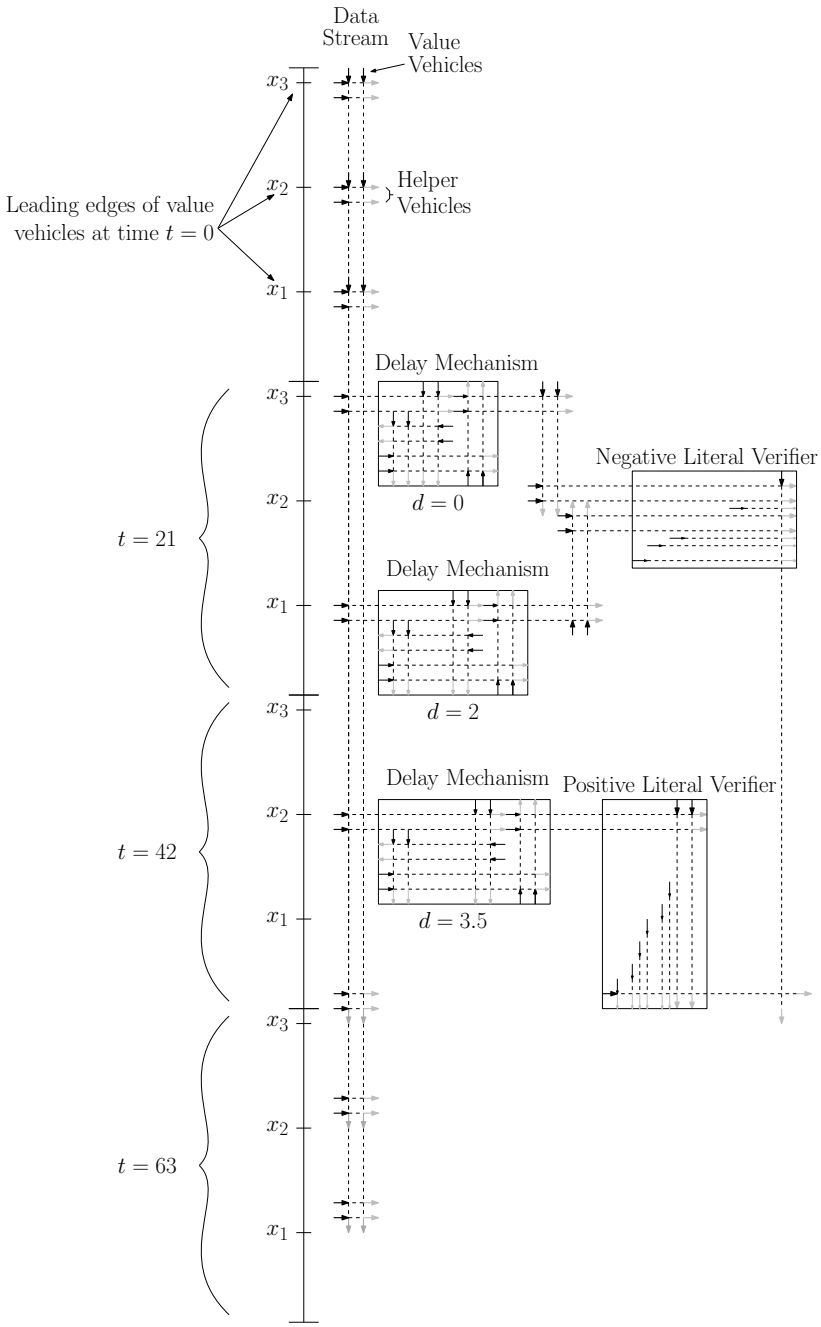


Figure 3.13: An example of a 3-SAT problem with $F = (\neg z_1 \vee z_2 \vee \neg z_3)$, expressed as a traffic crossing. As before, the ε-vehicles are not shown.

3.3.7 Membership in NP

Having shown the reduction from satisfiability, this section establishes NP-completeness by showing that Traffic Crossing is in NP. Given an input instance of the traffic-crossing problem $C = (V, \delta_{max})$, consisting of n vehicles in V, where all numeric values are given with b bits of precision, this section demonstrates a certificate of size $O(n^2)$ from which it is possible to validate a solution in $O(n^4(b + \log n))$ time.

Certificate : For each pair of orthogonal vehicles, v_i, v_j, their paths cross at a single intersection. The certificate provides a priority for each such pair, specifying which vehicle crosses through the intersection first. This requires $O(n^2)$ bits.

Let D be any valid set of speed profiles. Let $P(D)$ denote the associated certificate providing vehicle priorities. To validate D, consider a specific instance called the *full-speed profiles* as follows:

- Each vehicle v moves at full speed (i.e., δ_{max}) until either (1) arriving at an intersection, (2) it is about to collide with the rear end of a stopped vehicle in the same lane, or (3) it has reached its destination.

- If arriving at an intersection, the vehicle waits until all vehicles which have priority over it, according to P, have passed through the intersection. The vehicle will proceed through the intersection at full speed once the last of these vehicles has passed.

- If the vehicle has stopped in order to avoid a collision with the vehicle in front of it, it will proceed at full speed once the blocking vehicle has as well.

- If the vehicle has reached its destination it will stop as, of course, there is no more to be done.

This instance of D is referred to as D_{full}. To establish correctness, it suffices to show (1) D_{full} is valid if D is valid, and (2) D_{full} can be simulated in $O(poly(n, b))$ time to determine its validity. This is shown in the following two lemmas.

Lemma 3.3.4. *If D is a valid set of speed profiles then D_{full} is also valid.*

Proof. Define a *significant event* (for either profile) to be the time at which some vehicle v_i enters and leaves some intersection χ_j. These events will be referred to as $t^-(i,j)$ and $t^+(i,j)$, respectively, for the original set of speed profiles D. The significant events for the full-speed profile will be denoted as $t^-_{full}(i,j)$ and $t^+_{full}(i,j)$. To establish correctness, it suffices to show the following:

(i) There are no collisions in D_{full}.

(ii) For all i and j, $t^{\pm}_{full}(i,j) \leq t^{\pm}(i,j)$ (that is D_{full} moves vehicles through intersections as early as possible).

(iii) With D_{full}, the arrival times at destinations are earlier than or equal to those in D.

To establish (i), observe that no rear-end collision can occur by definition of the D_{full} policy. Also, no T-bone collisions can occur between crossing vehicles because (by priority) one is required to wait for the other.

Assertion (ii) is established by induction over the times of significant events. Initially, both profiles are in the same configuration, as given by the problem definition

C. Suppose toward contradiction that there exists a significant event concerning vehicle v_i and intersection χ_j where $t^-_{full}(i,j) > t^-(i,j)$. Consider the first such event. There are two possible reasons why v_i did not enter intersection χ_j at time $t^-(i,j)$ in profile D_{full}:

(a) It is waiting for some crossing vehicle v_k to exit the intersection (see Fig. 3.14(a)). By definition, v_k must have a higher priority in D (i.e., it passes prior to v_i in D), but v_k must have exited the intersection prior to $t^-(i,j)$. This contradicts the induction hypothesis that $t^-_{full}(i,j) > t^-(i,j)$.

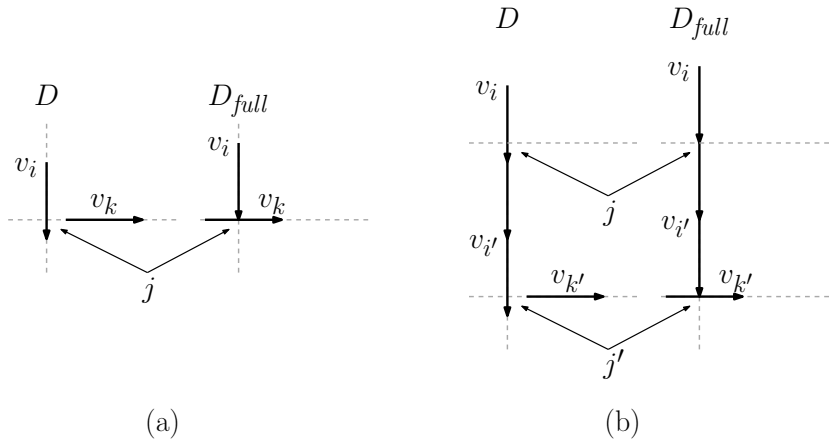


Figure 3.14: A figure comparing relative event times illustrating the cases in which (a) $t^-_{full}(i,j) > t^-(i,j)$ and (b) a traffic jam occurs.

(b) Vehicle v_i cannot proceed because it would rear-end the previous stopped vehicle (see Fig. 3.14(b)). In this situation, there is a chain of one or more vehicles stopped in front of v_i, where the first vehicle in the chain $v_{i'}$ is waiting at some intersection $\chi_{j'}$ for some vehicle $v_{k'}$ with priority to pass. In this case, the argument above can be applied to $v_{i'}$, $\chi_{j'}$, and $v_{k'}$.

Finally, the same holds for each t^+ value as they are each equal to the t^- values

offset by the constant length of the vehicles. In other words, for vehicle v_i:

$$t^+_{full}(i,j) \;=\; t^-_{full}(i,j) + length(v_i) \quad \text{and} \quad t^+(i,j) \;=\; t^-(i,j) + length(v_i).$$

To establish (iii), from (ii) it follows that $t^+_{full}(i,j) \leq t^+(i,j)$. Given this, adding the constant distance from the last intersection to the destination to each value does not change this relationship. Therefor, vehicles following D_{full} will arrive at a time equal to or earlier than vehicles following D. □

Lemma 3.3.5. *Given an instance of the traffic-crossing problem of size n, b and certificate P, D_{full} can be simulated in $O(n^4(b + \log n))$ time.*

Proof. Assume without loss of generality that the maximum speed is one unit per second. Recall the notion of significant events from Lemma 3.3.4. Simulation of the system using P is a simple discrete time event simulation in which time advances from one significant event to the next. Observe that given suitable data structures, each significant event can be processed in $O(n^2)$ time (ignoring numeric issues). The issue that remains is the number of bits of precision needed to represent the times at which each significant event occurs. Significant event times can be computed as follows:

$$t^+_{full}(i,j) = t^-_{full}(i,j) + length(v_i)$$

Let $\chi_{j'}$ denote the next intersection along the lane in which vehicle v_i is moving.

The time at which v_i hits intersection $\chi_{j'}$ is:

$$t^+_{full}(i,j) + dist(\chi_j, \chi_{j'})$$

At this time, the vehicle will either continue directly through the intersection, thus implying that the value of $t^-_{full}(i,j')$ is equal to the value above, or it will be forced to wait for some other significant event before it can move. Observe, then, that each significant event time is the sum of the vehicle length and the distances between consecutive intersections. If all coordinates are b-bits precise, then each time involves an $O(n)$-fold sum of b-bit numbers, or $O(b+\log n)$ bits, for a total of $O(n^2(b+\log n))$ bit operations. Finally, each vehicle can pass through at most $O(n)$ intersections for a total of $O(n^2)$ significant events. Thus, overall, the number of bit operations is less than or equal to the number of significant events, times the processing time for each, times the number of bits for each or $O(n^2 n^2(b+\log n) = O(n^4(b+\log n))$. □

In summary it follows that:

Lemma 3.3.6. *The traffic-crossing problem is in NP.*

By combining this with Lemma 3.3.1, the main result of this section is therefore:

Theorem 3.3.1. *The traffic-crossing problem is NP-complete.*

3.4 Sufficiency of Binary Speed Profiles

Recall that a speed profile is *binary* if vehicles move in the following restricted manner: first, they alternate between being stationary and moving at the speed

limit, δ_{max}, and second, vehicles stop only for one of three possible reasons: (a) the current time is outside the vehicle's time interval, (b) the vehicle is waiting to enter an intersection, and (c) the vehicle is part of end-to-end chain of collinear vehicles, where the chain's leader is stopped due to either (a) or (b). In this section it is shown that it suffices to assume that the speed profiles have this structure.

Let D be any solution to the given traffic-crossing problem. It will now be shown how to convert the speed profile for each vehicle into binary form. The approach presented here is to move each vehicle forward as rapidly as possible subject to two constraints: (1) the vehicle avoids rear-end collisions with the vehicle directly in front of it, and (2) if the vehicle's travel segment crosses a perpendicular travel segment, the time interval that the vehicle will spend in this intersection in the new solution will be a subinterval of that of the original solution. Constraint (1) implies that there are no rear-end collisions and constraint (2) implies that there are no T-bone collisions.

Begin by subdividing the vehicles into "traffic lanes." Given two vehicles v_i and v_j, v_i is said to *follow* v_j if

- the travel segments for these vehicles are collinear and overlap each other,
- v_j's starting position lies ahead of v_i's (relative to v_i's direction of travel), and
- there is no collinear vehicle whose starting position is between them.

As observed above, one may assume that v_i and v_j are traveling in the same direction, since otherwise a head-on collision is unavoidable. It is easy to see that each vehicle

can follow at most one other vehicle, and therefore this "following relation" partitions V into chains of one or more vehicles all moving collinearly in the same direction.

Consider any such chain $V' = \langle v_1, \ldots, v_k \rangle$, where v_i follows v_{i-1}. For the sake of concreteness, assume that the vehicles of V' are moving horizontally to the right (see Figure 3.15(a)). The construction is by induction on k. For any vehicle v_i, for $1 \leq i \leq k$, assume inductively that the speed profiles of all of the preceding vehicles (v_1 through v_{i-1}) have already been converted into binary form. Consider the vertical travel segments that cross the line segment from $p_i^{\vdash}$ to $p_i^{\dashv}$ sorted from left to right. Let $\langle a_1, \ldots, a_{m_i} \rangle$ denote the positions along v_i's segment where these crossings occur (see Figure 3.15(b)). For $1 \leq j \leq m_i$, let t_j be the time according to the original solution D where v_i first intersects a_j. (Because vehicles are open line segments, t_j is actually the infimum of the set of times at which v_i intersects this point.)

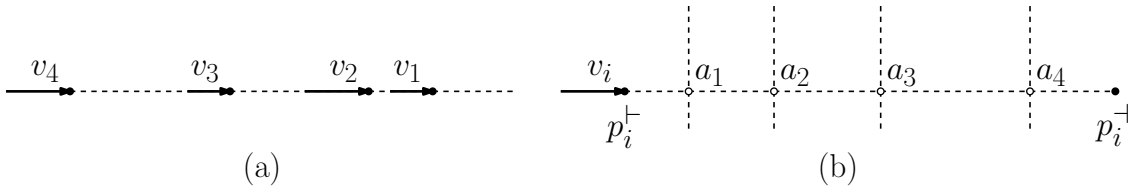


Figure 3.15: Proof of the sufficiency of binary profiles.

The new speed profile is defined as follows. First, the speed is limited throughout to avoid a rear-end collision with v_{i-1}. Since the speed profile of this preceding vehicle has already been converted to binary form, this imposes a binary upper bound on v_i's speed profile. Subject to this restriction, starting at time $t = 0$ v_i moves forward at the maximum speed δ_{max} until its right endpoint coincides with a_1. By the assumption that vehicles are open line segments, no collision can arise with vehicles traveling vertically through a_1. Because v_i travels at maximum speed, the

original solution cannot arrive at this point any earlier than v_i has. The vehicle waits here until time t_1, at which time v_i again travels at maximum speed until reaching a_2. The vehicle continues in this manner (recalling the constraint of avoiding rear-end collisions with v_{i-1}) until arriving at the goal position $p_i^{\dashv}$. Let D' denote the resulting set of binary speed profiles.

Because vehicle v_i enters a_j at the same time as in solution D, and it moves as rapidly as possible, its position in the presented solution will never lag behind its position in the original solution. (There is an implicit induction here. Vehicle v_i may be delayed to avoid rear-ending v_{i-1}, but by induction v_{i-1} cannot lag.) It follows that the time interval that each vehicle spends within a given intersection for solution D' is a subinterval of the corresponding time interval for the same vehicle-intersection pair in D. Combined with the rear-end constraint, it follows that D' is collision-free. Finally, because vehicles move as fast as possible and D satisfies all the vehicle deadlines, D' also satisfies these deadlines. In conclusion, D' is a valid binary solution, as desired.

It is apparent from the above proof that the number of breakpoints in the resulting binary solution can be bounded above by a function of the number of vehicles, and in fact it is quadratic in the number of vehicles. To see why, observe that the first vehicle of each chain stops at each of the intersections with perpendicular travel segments, of which there are at most n. The second vehicle of each chain also stops at each of its intersections with perpendicular segments, but it may also stop to avoid a rear-end collision each time the first vehicle stops, for a total of at most $2n$ stops. An easy induction argument implies that the last vehicle in a chain of

length k stops at most $kn = O(n^2)$ times. Therefore, any solvable instance with n vehicles has a binary solution of combinatorial complexity $O(n^3)$.

3.5 A Solution to the One-Sided Problem

While the generalized traffic-crossing problem is NP-complete, it is possible to solve a constrained version of the problem more efficiently. The complexity of the generalized traffic-crossing problem arises from the interplay between horizontal and vertical vehicles, which results in a complex cascade of constraints. To break this interdependency, the vertically traveling vehicles are given priority, allowing them to continue through the intersection at a fixed speed. In this variant, called the *one-sided problem*, the horizontal vehicles can plan their motion with complete information and without fear of complex constraint chains.

First, the assumption is made that the vertically traveling vehicles are invariant and are all traveling at the same speed, s_n. With vertical vehicle motion now fixed, there is no way for horizontal vehicles to affect one another and movement profiles for each can be found in isolation. Finally, all vehicles are assumed to be of length l and in general position.

For the purpose of illustration, a simplified version of the problem is presented first, and then, over the course of three cases, restrictions are relaxed until what is left is a solution to the original problem under the fixed, one-sided policy described above. These three cases are:

Intersection Between One-Way Highways

- Vertical vehicles approach from the North only.
- Horizontal vehicles approach from the West only.
- Each vehicle is in its own lane (i.e., no two vehicles are collinear).

Intersection Between a One-Way Street and a Two-Way Highway

- Vertical vehicles approach from the North and the South.
- Horizontal vehicles approach from the West only.
- There is a single horizontal lane (i.e., all horizontal vehicles are collinear) and one or more vertical lanes.

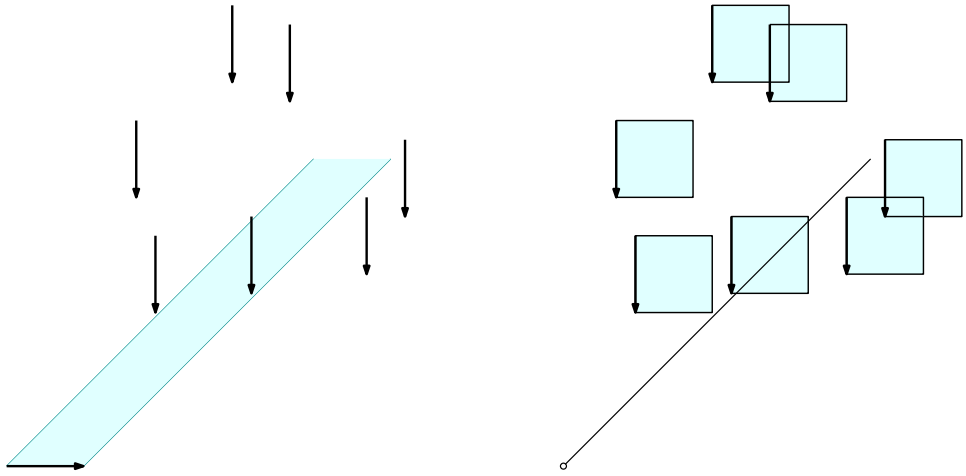
Intersection Between Two-Way Highways

- Vertical vehicles approach from the North and the South.
- Horizontal vehicles approach from the West and the East.
- There are k horizontal lanes, one or more vertical lanes, and vehicles may be collinear.

3.5.1 Intersection Between One-Way Highways

Formally, vehicles from the North are in the subset $N \subset V$ and their direction of travel is $d_n = (0, -1)$, whereas vehicles from the West are in the subset $W \subset V$ with a direction of travel of $d_w = (1, 0)$. Again, the only task is to find valid speed profiles for vehicles coming from the West.

To begin, the problem space is transformed so that the vehicles in W are represented as points rather than line segments. This makes movement planning simpler while maintaining the geometric properties of the original space. Every vehicle in W is contracted from left to right, until it is reduced to its leading point. In response, the vehicles in N are expanded, transforming each into a square obstacle with sides of length l (see Fig. 3.16) and with their left edges coincident with the original line segments.



(a) (b)

Figure 3.16: (a) A random traffic-crossing problem as viewed from a single active vehicle. (b) The resulting space after the point transformation.

Given the global speed limit δ_{max}, there are regions in front of each obstacle in which a collision is inevitable (this concept is similar to the obstacle avoidance work

done in [51]). These triangular zones (referred to as *collision zones*) are based on the speed constraints of the vehicles and are formed by a downward extension of the leading edge of each obstacle. The leftmost point of this edge is extended vertically and the rightmost point is extended at a slope derived from the ratio between δ_{max} and the obstacle speed. As one last concession to clarity, the axes of the problem space are scaled so that this ratio becomes 1. Formally, a collision zone Z_O for the obstacle O is the set of all points p, such that there is no path originating at p with a piecewise slope in the interval $[1, \infty]$ that does not intersect O.

Expanding the vehicles in N into rectangular obstacles may cause some to overlap, producing larger obstacles and, consequently, larger collision zones. This merging and generation of collision zones is done through a standard sweep line algorithm and occurs in $O(n \log n)$ steps, where n is the number of obstacles, as described below.

3.5.1.1 Merging Obstacles and Growing Collision Zones

This process is done using a horizontal sweep line moving from top to bottom. While the following is a relatively standard application of a sweep line algorithm, it is included for the sake of completeness. First, the event list is populated with the horizontal edges of every obstacle, in top-to-bottom order, requiring $O(n \log n)$ time for $O(n)$ obstacles. The sweep line status stores a set of intervals representing the interiors of disallowed regions (e.g., the inside of an obstacle or collision zone). Each interval holds three pieces of information: the location of its left edge, a sorted list

of the right edges of any obstacles within the interval, and the slopes of these right edges. These slopes will be either infinite (i.e., the edges are vertical) or will have a slope of 1.

In addition to horizontal edge positions, the event list must keep track of three other events which deal with the termination of the sloped edges of the collision zones. These edges begin at the bottom right edge of an obstacle and terminate in one of three ways: against the top of another obstacle, against the right edge of another obstacle, or by reaching the left edge of an interval. The first case is already in the event list as the top edges were added at the start of this process. The remaining two cases are added as the sweep line progresses through the obstacles.

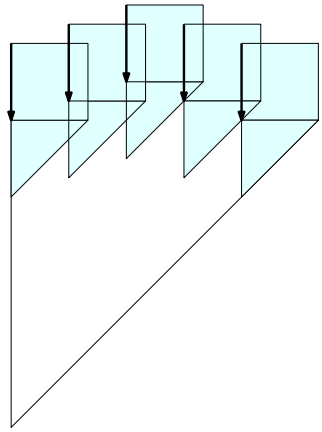


Figure 3.17: An example illustrating the need to redefine collision zones when obstacles overlap. Here, the collision zones for each individual obstacle (represented as shaded triangles) are insufficient as the merger creates a larger area that vehicles must avoid (seen here as the unfilled triangle).

So, when merging obstacles, the sweep line must handle the following events:

Top Edge Encounter :

When the sweep line encounters the top edge of an obstacle it must either create a new interval or add this obstacle to an existing interval. The creation

of a new interval is straightforward as the endpoints of the edge are all that need to be added (see Fig. 3.18(a)).

If the top edge intersects an existing interval, however, there is a little more work to be done. First, if the leftmost point of the edge does not lie within the interval then it becomes the new leftmost edge of the interval (see Fig. 3.18(b)). If the sloped edge of a collision zone has already formed for this contiguous block of obstacles (see **Bottom Edge Encounter** for a description of how these form), then the termination point of the sloped edge may need to be updated to account for a shift in the leftmost edge. Second, the rightmost point of the encountered edge is inserted into the list of right edges in left-right order. The new edge may become the new rightmost edge and if the previous rightmost edge was sloped then it is removed from the edge list. For example, in Fig. 3.18(d) this has just occurred within the set of obstacles on the left. If the newly added right edge does not replace the sloped edge and the sloped edge intersects the newly added edge, the point at which they intersect is added to the list of events to be processed (this occurs in Fig. 3.18(c) on the right side). If there is an existing event in the event list for the sloped edge's intersection with another obstacle, it must be deleted as the addition of the newest obstacle will truncate the edge before it reaches that event.

Bottom Edge Encounter :

When the bottom edge of an obstacle is encountered, the obstacle's right edge is found in the interval's edge list. If it is not the rightmost, it is removed from

the edge list (this occurs in Fig. 3.18(e) on the left, denoted by the grey slope arrow). If the edge to be removed is the rightmost edge in the list, rather than removing it, its slope is changed to that of the ratio between the vehicles' speed limit and the speed of the vehicles, $\frac{\delta_{max}}{s_n}$. Next, the termination point for this sloped edge is added to the event list. This is the point at which the leftmost edge of the interval and the sloped edge meet. This point is illustrated in Fig. 3.18(e), though it was added when the previous bottom edge was processed. As noted above, this event may need to be updated if a top edge is encountered that moves the leftmost edge of this interval.

Sloped Edge Termination :

When the sloped edge terminates against a right edge, it is deleted from the edge list. This makes the edge with which it collided the new rightmost edge.

Interval Termination :

In this case, the sloped edge of the collision zone has met the leftmost edge of the interval. When this is the case, the interval has finally closed and can be removed from the sweep line status (see Fig. 3.18(f)).

The initial population of the event list occurs in $O(n \log n)$. As the sweep line progresses through the obstacle space, it adds and removes the right edges of obstacles to the appropriate intervals. These lists of edges are built incrementally in sorted order, requiring only $O(\log n)$ time. Finally, as there is a constant number of possible events per obstacle (a single top edge, a single bottom edge, and a single termination of its sloped edge), there are at most $O(n)$ events to be processed. Thus,

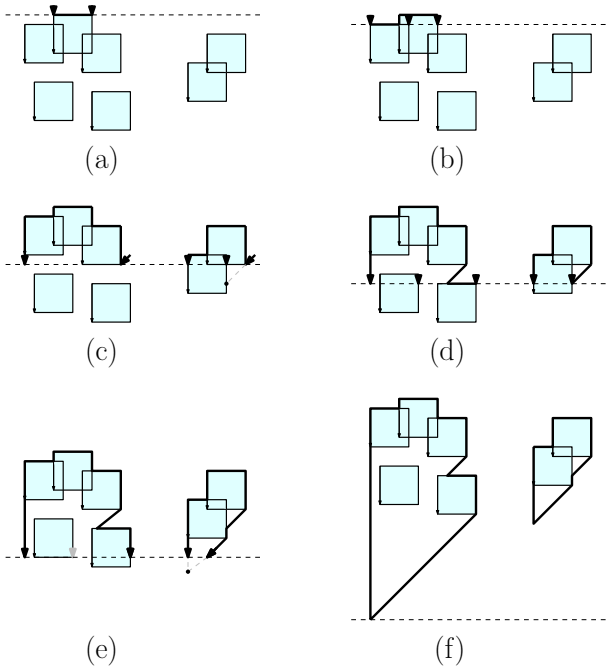


Figure 3.18: A sweep line merging obstacles and creating collision zones. Note: these illustrations do not show every step in the sweep line process. Some are skipped in order to save space. (a) Encountering the first top edge and adding an interval to the sweep line status. (b) Encountering the next top edge, which increases the interval size. (c) Encountering bottom edges changes the rightmost slope of the collision zone. Notice on the right that an internal right edge is stored in the status. (d) Sloped edges encounter the top of an unprocessed obstacle and the rightmost edge of an obstacle in an interval. (e) Encountering the bottom edge of an internal obstacle. Its rightmost edge is deleted from the sweep line status. (f) Reaching the point of convergence for a collision zone. The interval is deleted from the sweep line status.

the sweep line processes the obstacle space in $O(n \log n)$ time.

3.5.1.2 Movement Planning

Once the obstacles have been merged and grown appropriately, speed profiles allowing each vehicle to safely cross the intersection need to be found. This is done with the same obstacle-filled space that has been used thus far, though with a small change in perspective. Currently, vehicles are only allowed horizontal movement and obstacles only move vertically. Instead, the obstacles are treated as static objects and a vertical velocity component is added to the vehicles equal to the obstacles' speed. So, for example, a vehicle moving at the maximum speed will actually follow a path with a slope of $\frac{s_n}{\delta_{max}}$ whereas a stationary vehicle will travel vertically. Again, the axes have been scaled so that this ratio is 1, imposing on the vehicle monotonic movement with a slope in the interval $[1, \infty]$. With this understanding, a path through the obstacles can now easily be found while obeying the speed constraints of the vehicles.

The vehicle will travel at its minimum slope (equivalent to its maximum speed) until it either reaches its goal position or encounters an obstacle. If an obstacle is encountered, the vehicle travels vertically until it is no longer blocked (this vertical motion corresponds to stopping and waiting for the obstacle to pass). Once this occurs, the vehicle continues on its way at its maximum speed until it has covered the distance to its goal (measured horizontally, as vertical movement no longer represents spatial translation).

The path created by the above behavior can be efficiently found through the use of another line sweep. First, notice that every edge that is locally to the left of an obstacle (referred to as a *left edge*) is a vertical line segment. Since the vehicles move monotonically, they will only ever encounter an obstacle at one of these left edges. So, to find a path for each vehicle traveling at speed δ_{max}, a sweep line perpendicular to the vehicles' trajectories is created and swept from the upper-right to the lower-left (see Fig. 3.19(a)). This perpendicular line's status will maintain a list of obstacle occlusions with respect to the vehicles' direction of travel by adding an interval for each obstacle as it is encountered during the sweep. More specifically, it stores the point where the sweep line first encountered the obstacle's left edge, the horizontal position of the left edge, and the point where the sweep line last encountered the edge.

During the sweep, a tree is built representing a set of all paths through the obstacle field that encounter an obstacle. Vehicles will either encounter an obstacle in the tree or are free to travel at full speed without collision until their goal is reached. Each obstacle is a vertex in the tree and edges represent the path taken after encountering this obstacle. The edge will either lead to an encounter with another obstacle or will lead to the root. The root is the only vertex which does not represent an obstacle but instead signifies an open path to the goal.

The event list for the sweep line is populated with the upper and lower ends of each left edge. Whenever an upper end is encountered, it is inserted into the list of intervals in the sweep line status and the obstacle is inserted into the path tree. If the insertion point does not lie within an existing interval, then an edge between the

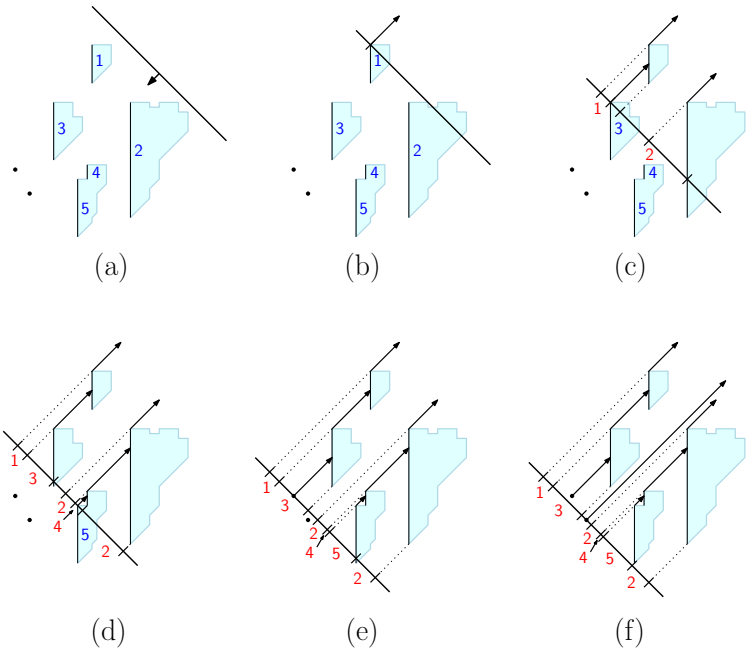


Figure 3.19: (a) A sweep line for path finding, traveling perpendicular to the direction of travel of a vehicle moving at speed δ_{max}. (b) The sweep line encountering vertical edge 1. As there is no interval on the sweep line where it occurs, this line's path goes directly to the goal at speed δ_{max}. Edges to the root of the path tree are represented by arrows going off to infinity. (c) The sweep line encountering edge 3. This encounter lies in the interval for edge 1. (d) Encountering edge 5, creating a path from it to edge 4. (e) Encountering the first vehicle, which lies in the interval for edge 3. Thus, the final path for the vehicle is to travel at maximum speed until it reaches edge 3, wait for the edge to pass, travel to 1, wait, and finally travel to the goal position. (f) The sweep line encountering the second vehicle at an open interval. Thus, this vehicle can travel at speed δ_{max} until it reaches its goal position.

obstacle and the root is created (see Fig. 3.19(b)). If the insertion point lies within an interval, that interval is split by the inserted point and an edge between the new obstacle and the interval's obstacle is added (see Fig. 3.19(c)).

Whenever the lower end point of an obstacle's left edge is encountered, the interval ending for that obstacle is added to the list. If an event occurs before an interval has completed, the interval's intermediate size can be determined using the position of the sweep line, the start point of the interval, and the position of the obstacle's left edge (this occurs in Fig. 3.19 between (c) and (d)). Finally, when a vehicle is encountered, its position along the sweep line determines its path. If it is in an interval, then its path begins by traveling to the associated obstacle and, using the path tree, travels to that obstacle's parent obstacle, repeating this process until it has reached its goal position.

So, in the example in Fig. 3.19(e), the upper vehicle encounters obstacle 3, waits for it to pass (i.e., travels vertically till the end is reached), moves at the maximum speed until it encounters obstacle 1, then continues on until it reaches its goal position. The lower vehicle, having been inserted into the interval list in between intervals, is free to travel at the maximum speed until its goal position is reached (see Fig. 3.19(f)).

3.5.2 A One-Way Street and a Two-Way Highway Intersection

In this case, vertical vehicles approach from the North and the South while horizontal vehicles travel in a single lane.

To account for the bidirectional vertical vehicles the space is folded along the horizontal lane. This rotates the northbound traffic to an equivalent southbound set of vehicles (see Fig. 3.20). This only requires an $O(n)$ transformation. Using the plane sweep algorithm above yields a combined obstacle space.

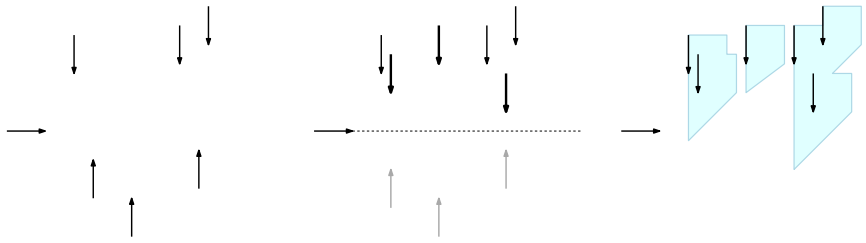


Figure 3.20: (a) An example of bidirectional cross-traffic. (b) To account for how these vehicles interact when they reach a horizontal lane, the space can be folded along the lane, rotating one set of vehicles about the fold. (c) Then, the same space transformation and obstacle merger detailed above is performed.

Finally, vehicles must be prevented from rear-ending each other. Once the lead vehicle has found a motion plan through the obstacles, it creates a new set of constraints for the vehicles behind it. The monotonic path of the lead vehicle is stored in a binary search tree, allowing for easy collision queries.

To begin with, the lead vehicle's path needs to be added to the tree. However, because the lead vehicle is represented by a single point coincident with the front of the vehicle, the path being stored needs to be shifted leftward by an amount equal to the vehicle's length (see Fig. 3.21(b)). The next vehicle in the lane must not collide with this newly created boundary.

In the simplest case, the trailing vehicle can simply adopt the same movement policy as prescribed by the algorithm in Section 3.5.1.2. However, this solution needs to be modified in the following two cases: (i) when the lead vehicle's shifted path

pierces an obstacle, closing off any space through which the trailing vehicle could follow or (ii) when the trailing vehicle would collide with the rear of the lead vehicle.

In case (i), the concern is the creation of overhangs in the obstacle space. In Section 3.5, overhangs were eliminated through the merging of obstacles. In this case, however, vertical portions of a lead vehicle's path may pierce an obstacle, recreating such overhangs. While the merging algorithm could be used to eliminate them once again, doing so for each vehicle is too costly.

Instead, for each obstacle, it is noted which other obstacle, if any, lies directly below its left edge (this can be found during the obstacle merge plane sweep or through ray shooting). If an obstacle is pierced by the leading vehicle's path then the space below it can no longer be part of a viable path, as any vehicle entering this space will become trapped. If another obstacle lies directly below the first, it will need to close off the space below itself as well. It is possible for this operation to cascade down through multiple obstacles, but once an obstacle has closed off the space below it, it will never need to make this update again. Thus, this update only requires $O(n)$ time. This cascading path is then added to the boundary created above, pruning a portion of the search tree and replacing it with the new path (see Fig. 3.21(c)).

For case (ii), the trailing vehicle must check for a collision with the boundary when traveling at full speed (i.e., any time it leaves the upper corner of an obstacle and travels at slope 1). Given the binary search tree in which this boundary is stored, one can query for collisions in $O(\log n)$ time for each obstacle the trailing vehicle encounters. If a collision occurs, the trailing vehicle simply follows the boundary

from that point forward. Because overhangs were eliminated above, this leads to the fastest collision free motion plan for the trailing vehicle. As each obstacle will be subsumed by the boundary once the trailing vehicle has determined its motion plan, this query will only ever occur once for each obstacle, leading to a time complexity of $O(n \log n)$.

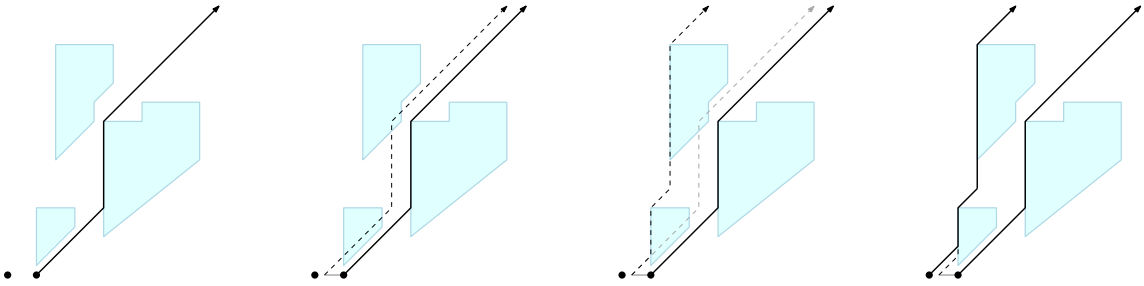


Figure 3.21: An example of two vehicles in the same lane planning their motion through a set of obstacles. (a) As before, the lead vehicle projects its path and queries the first encountered obstacle for the remainder of the motion plan. (b) The path taken by the lead vehicle's rear end is used to create a boundary for the vehicles behind it. (c) This boundary creates overhangs which are eliminated by bending it around the obstacle(s) that create the overhang. (d) The motion planning algorithm used in Section 3.5.1.2 is used here, taking the new boundary into account.

In the end, one can still account for shared lanes without a running time greater than $O(n \log n)$.

3.5.3 Intersection Between Two-Way Highways

Finally, this case combines the two above, allowing for bidirectional movement horizontally and vertically, with multiple lanes along each axis.

The vehicles approaching from the East are independent of those approaching from the West, presenting a symmetric problem that can be solved with the techniques discussed above. The addition of horizontal lanes, however, impacts the running

time of the algorithm. Previously, the bidirectional vertical traffic was accounted for by folding the obstacle space along a single horizontal lane, but in this case, because the position of the vertical vehicles relative to each other is different at any given intersection with a horizontal lane, the folding must occur individually for each lane. Thus, the algorithm runs in $O(kn \log n)$, for k horizontal lanes. In general, k is assumed to be a relatively small constant.

3.6 Traffic Crossing in the Discrete Setting

This section considers the problem in a simple discrete setting where vehicles move along the points of a grid. While this formulation is less realistic than the continuous one, it avoids some of the cumbersome elements of the continuous formulation. This admits a much clearer view of the sources of computational complexity while still capturing the most salient elements of the original traffic-crossing problem.

It is assumed that each vehicle occupies a point on the integer grid in the plane, $\mathbb{Z}^2$. Time advances discretely in unit increments, and at each time step a vehicle may either advance to the next grid point or remain where it is. A collision occurs if two vehicles occupy the same grid point.

The *discrete traffic-crossing problem* is defined in much the same manner as in the continuous case. The problem is presented as a set V of n vehicles on the integer grid. Each vehicle v_i is represented by its initial and goal positions $p_i^{\vdash}$ and $p_i^{\dashv}$, respectively, both in $\mathbb{Z}^2$. Also given are a starting time $t_i^{\vdash}$ and deadline $t_i^{\dashv}$, both

in $\mathbb{Z}^+$ (where $\mathbb{Z}^+$ denotes the set of nonnegative integers). A vehicle's direction d_i is a unit length vector directed from its initial position to its goal, which is either horizontal or vertical. Time proceeds in unit increments starting at zero. The motion of v_i is specified as a function of time, $\delta_i(t) \in \{0, 1\}$. Setting $\delta_i(t) = 0$ means that at time t vehicle i remains stationary, and $\delta_i(t) = 1$ means that it moves one unit in direction d_i. Thus, v_i's position at time $t \geq 0$ is $p_i(t) = p_i^{\vdash} + d_i \sum_{x=0}^{t} \delta_i(x)$.

Generalizing the problem definition from Section 3.2, the objective is to compute a speed profile $D = \langle \delta_1, \ldots, \delta_n \rangle$ involving all the vehicles that specifies a collision-free motion of the vehicles in such a manner that each vehicle starts at its initial position and moves monotonically towards its goal, arriving there at or before its given deadline. Similar to road networks, the assumption is made that along any horizontal or vertical grid line, the vehicle direction vectors are all the same.

In this context a natural optimization problem can be considered, namely, scheduling traffic to minimize the maximum delay experienced by any vehicle in the discrete setting. For each vehicle, consider only its initial and goal positions. It is assumed that all vehicles share the same starting time at $t = 0$. A vehicle v_i experiences a *delay* at time t if it does not move at this time ($\delta_i(t) = 0$). Otherwise, it is assumed to move one unit forward ($\delta_i(t) = 1$). The *total delay* experienced by a vehicle is the total number of time instances where it experiences a delay until the end of the motion simulation. The *maximum delay* of the system is the maximum total delay experienced by any vehicle.

While a formal proof is omitted, it is not hard to demonstrate that the NP-hardness reduction of Section 3.3 can be transformed to one showing that it is

NP-hard to minimize maximum delay in the discrete setting. Intuitively, the reason is that the reduction involves purely discrete quantities: integer vehicle coordinates and starting times, vehicles of unit length, and unit speed limit. The system described in the reduction is feasible if and only if the maximum delay is at most five time units.

3.6.1 The Unit-Delay Problem

The above hardness result for the 5-unit delay problem raises the question of whether it is possible to efficiently answer the question for *any* positive delay value smaller than five. This section will show that (subject to some assumptions on the starting configuration) it is possible to determine efficiently whether it is possible to schedule traffic in order to achieve a delay of at most one unit.

It is helpful to introduce a useful concept first, before stating the result. A sequence of vehicles lying on the same road is said to form a *caravan* if they occupy consecutive positions on the grid (similar to the platoons introduced by Besa Vial et al. [89]). Caravans are significant in the discrete setting because once any vehicle of the caravan suffers a delay, all subsequent vehicles of the caravan immediately suffer the same delay to avoid read-ending each other. Observe that in the unit-delay setting this effect is limited to a single caravan, since if the last vehicle from one caravan delays and assumes a position at the front of a caravan that is following one unit behind, it cannot delay again, and hence it cannot affect the motion of vehicles in the following caravan.

Very long caravans cause problems. Suppose that a horizontal caravan is so long that it spans two or more intersections on the same road. If a vehicle near the front of the caravan delays, then this delay will propagate back to previous intersections, impeding traffic on the crossing vertical road. If this causes a delay in a very long vertical caravan, the delay may propagate to another horizontal road, and so on. It is easy to see that this could result in a cycle of delays, resulting in a gridlocked state in which no vehicle can advance.

In order to avoid a lengthy digression into how to handle cyclic dependencies (which is left as an open problem), the problem is overcome via a simple assumption. The input is said to satisfy the *short-caravan assumption* if the length of any caravan in the input configuration along any road is strictly smaller than the distance between two intersections along this road. Intuitively, this assumption implies that delays suffered by vehicles passing through one intersection cannot "back up" into earlier intersections. This allows for the independent processing of intersections.

Theorem 3.6.1. *Subject to the short-caravan assumption, there exists an algorithm that solves the 1-unit maximum delay traffic-crossing problem in the discrete setting. The algorithm runs in time $O(nm)$, where n is the total number of vehicles and m is the maximum number of intersections crossed by each vehicle.*

The rest of this section is devoted to proving this theorem by showing that the problem can be reduced to an instance of 2-SAT in $O(nm)$ time and space. The result follows from the fact that 2-SAT can be solved in linear time [90]. We use the easy observations that clauses of the form $(x \Rightarrow y)$ (implies) and $(x \oplus y)$ (exclusive-or)

can both be expressed in 2-SAT form.

Since the maximum delay is one unit, throughout the motion process each vehicle may be in one of two states, either having not experienced any delay up to that point or having experienced a single delay. For each vehicle v_i and each intersection k it passes through, create a Boolean variable $x_{i,k}$, whose value will be **True** to signify that this vehicle has experienced a delay on entry to intersection k, and otherwise its value is **False**.

A 2-SAT instance is generated with the following clauses, which together enforce the conditions of a solution with a maximum delay of one unit:

(i) For each vehicle v_i and each pair of consecutive intersections k and k' that v_i passes through, if v_i is delayed on entering k, then it is still delayed on entering k'. Add the clause $(x_{i,k} \Rightarrow x_{i,k'})$.

(ii) For each intersection k, if two vehicles v_i and v_j, one horizontal and one vertical, pass through k, and their starting positions are equidistant from k, they cannot both be in the same state when arriving at this intersection, for otherwise they would collide. Add the clause $(x_{i,k} \oplus x_{j,k})$.

(iii) For each intersection k, if two vehicles v_i and v_j, one horizontal and one vertical, pass through k, and v_i is one unit closer to k than v_j, if v_i delays on entering k, then v_j must delay as well to avoid a collision. Add the clause $(x_{i,k} \Rightarrow x_{j,k})$.

(iv) For each pair of vehicles v_i and v_j in the same lane such that v_i's initial position is one unit before v_j's initial position, if v_i delays then v_j must delay as well to

avoid rear-ending v_i. For all intersections k through which these vehicles pass, add the clause $(x_{i,k} \Rightarrow x_{j,k})$.

Note that when long caravans are present, rule (iv) is not complete. It operates on only a single intersection, and hence it does not consider the effect of how a delay suffered by one vehicle within a caravan could propagate backwards to induce a delay on a vehicle following within the same caravan at a prior intersection. The short-caravan assumption saves us, because it implies that such propagations cannot occur.

Because only one-unit delays are tolerated, all instances of potentially colliding vehicles are handled by cases (ii), (iii), and (iv). It is easy to see that if there is a unit-delay solution to the given Traffic Crossing instance, then one can assign truth values to satisfy the 2-SAT formula.

To complete the proof, the converse is shown, namely that if the above formula is satisfiable, then there exists a unit-delay solution to the given Traffic Crossing instance. Consider any vehicle v_i. Let c_i denote the number of vehicles before it within its starting caravan, that is, let c_i be the largest integer such that at time $t = 0$, the c_i positions in front of $p_i^{\vdash}$ are occupied by other vehicles. (If v_i is not part of a caravan, one can think of it as being at the head of a trivial caravan of length one, and $c_i = 0$.) The speed profile of v_i is defined as follows. Let k be the first intersection such that $x_{i,k}$ is **True**. If no such k exists, then v_i moves without delay to its goal position. Otherwise, let $\ell_{i,k}$ denote the distance between v_i's starting position and intersection k. This vehicle moves at full speed until it is c_i units from

k, then delays one time unit, and then continues without delay to its destination. (That is, $d_i(t) = 1$ except at $t = \ell_{i,k} - c_i$, where $d_i(t) = 0$.) This motion profile is consistent with rule (i), since once a vehicle is delayed at intersection k, it is delayed at all subsequent intersections k'.

Clearly, each vehicle suffers at most a single time unit of delay under the resulting speed profile. It will be shown that, by the satisfiability of the 2-SAT formula, no collisions occur. By rules (ii) and (iii) in combination with the short-caravan assumption, which allows dependencies between intersections to be ignored, vehicle v_i cannot collide with any vehicle that is traveling perpendicular to it. All that remains to show is that v_i is not "rear-ended" by the vehicle immediately following it on the same road. Let v_j be this vehicle. If v_i starts at least two units ahead of v_j, then even if v_i delays and v_j does not, they are still at least one unit apart and so no collision can occur. (Observe, for example, that v_i may be the last vehicle of a caravan and v_j is the first vehicle of an immediately following caravan with a gap of one unit between them. It is possible that v_i delays and v_j does not, thus causing these two caravans to effectively merge. Since v_i cannot delay again, there is no possibility for a collision between these caravans.) Otherwise, v_i is directly ahead of v_j at the start. It follows that $c_j = c_i + 1$ and $\ell_{j,k} = \ell_{i,k} + 1$, and therefore $\ell_{j,k} - c_j = \ell_{i,k} - c_i$. If these vehicles are going to collide, it would occur at the instant that v_i first delays (and v_j has not yet delayed). By rule (iv) and the fact that $x_{i,k} = \textbf{True}$, it is the case that $x_{j,k} = \textbf{True}$. Therefore at time $t = \ell_{i,k} - c_i = \ell_{j,k} - c_j$, vehicle j has also delayed, and hence the "rear-ending" event cannot occur. (Vehicle v_j delays exactly at this time instant if this is the first k such that $x_{j,k} = \textbf{True}$, and

otherwise it delayed at a prior intersection and so is still delayed.)

The formula clearly involves $O(nm)$ variables and $O(nm)$ clauses and therefore the reduction runs in $O(nm)$ time. This establishes Theorem 3.6.1.

3.6.2 The Parity Heuristic

In the discrete setting it is possible to describe a simple common-sense heuristic. Intuitively, each intersection will alternate in allowing horizontal and vertical traffic to pass. Such a strategy might be far from optimal because each time a vehicle arrives at an intersection, it might suffer one more unit of delay. To address this, whenever a delay is imminent, a vehicle is chosen to delay in a manner that will avoid cross traffic at all future intersections. Define the *parity* of a grid point $p = (p_x, p_y)$ to be $(p_x + p_y)$ mod 2. Given a horizontally moving vehicle v_i and a time t, say that v_i is *on-parity* at t if the parity of its position at time t equals t mod 2. Otherwise, it is *off-parity*. Vertically moving vehicles are just the opposite, being *on-parity* if the parity of their position is *not* equal to t mod 2. Observe that if two vehicles arrive at an intersection at the same time, one moving vertically and one horizontally, exactly one of them is on-parity. This vehicle is given the right of way, as summarized below.

Parity Heuristic: If two vehicles are about to arrive at the same intersection at the same time t, the vehicle that is on-parity proceeds, and the other vehicle waits one time unit (after which it will be on-parity, and will proceed).

The parity heuristic has a number of appealing properties. First, once all the vehicles in the system are on-parity, every vehicle may proceed at full speed

without the possibility of further collisions. Second, the heuristic is not (locally) wasteful in the sense that it does not introduce a delay into the system unless a collision is imminent. Finally, the rule is scalable to large traffic systems, since a traffic controller at an intersection need only know the current time and the vehicles that are about to enter the intersection. No global information need be maintained. While this idealized setting is admittedly limited, these properties would be desirable for more realistic traffic management systems.

3.6.3 Steady-State Analysis of The Parity Heuristic

Delays may be much larger than a single time unit under the parity heuristic. (For example, a sequence of k consecutive vehicles traveling horizontally that encounters a similar sequence of k vertical vehicles will result in a cascade of delays, spreading each into an alternating sequence of length $2k$. It is easy to see that no matter how they are scheduled, at least one vehicle will suffer a delay of k.) This is not surprising given the very simple nature of the heuristic. It is not difficult to construct counterexamples in which the maximum delay of the parity heuristic is arbitrarily large relative to an optimal solution. It will be shown, however, that the parity heuristic is asymptotically optimal in a uniform, steady-state scenario (to be made precise below).

Consider a traffic crossing pattern on the grid. Let m_x and m_y denote the numbers of vertical and horizontal lanes, respectively. Each lane is assigned a direction arbitrarily (up or down for vertical lanes and left or right for horizontal).

Let R denote a $W \times W$ square region of the grid containing all the intersections (see Fig. 3.22(a)). In order to study the behavior of the system in steady-state, imagine that R is embedded on a torus, so that vehicles that leave R on one side reappear instantly in the same lane on the other side (see Fig. 3.22(b)). Equivalently, one can think of this as a system of infinite size by tiling the plane with identical copies (see Fig. 3.22(c)). Assume that W is even.

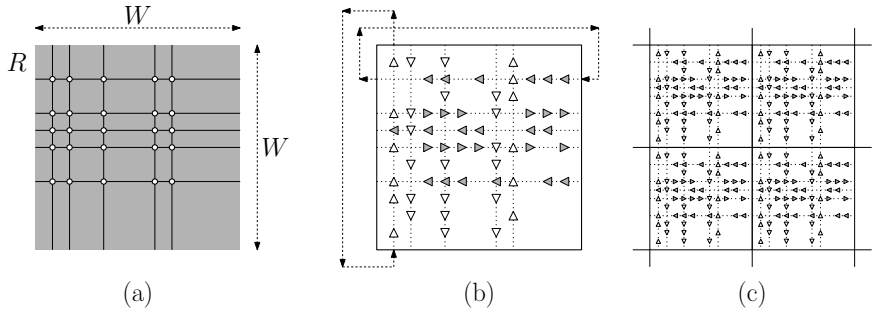


Figure 3.22: Analysis of the Parity Heuristic.

If the system is sufficiently dense, the maximum delay of the system will generally grow as a function of time. Given a scheduling algorithm and a discrete traffic crossing, define its *delay rate* to be the maximum delay after t time units divided by t. Define the *asymptotic delay rate* to be the limit supremum of the delay rate for $t \to \infty$. The objective is to show that, given a suitably uniform traffic crossing instance on the torus, the asymptotic delay rate of the parity algorithm is optimal.

A traffic crossing on the torus is said to be *uniform* if every lane (within the square R) has an equal number of vehicles traveling on this lane. Letting n' denote this quantity, the total number of vehicles in the system is $n = n'(m_x + m_y)$. (The total number of positions possible is $W(m_x + m_y) - m_x m_y$, and so

$n' \leq W - m_x m_y/(m_x + m_y)$.) The initial positions of the vehicles within each of the lanes is arbitrary. Let $p = n'/W$ denote the density of vehicles within each lane. Let $\rho_\infty^{\text{par}} = \rho_\infty^{\text{par}}(W, p, m_x, m_y)$ denote the worst-case asymptotic delay rate of the parity heuristic on any uniform discrete traffic crossing instance of the form described above, and let $\rho_\infty^{\text{opt}} = \rho_\infty^{\text{opt}}(W, p, m_x, m_y)$ denote the worst-case asymptotic delay for an optimum scheduler.

The approach will be to relate the asymptotic performance of parity and the optimum to a parameter that describes the inherent denseness of the system. Define $\chi = \max(0, 2p - 1)$ to be the *congestion* of the system. Observe that $0 \leq \chi \leq 1$, where $\chi = 0$ means that the density is at most $1/2$ and $\chi = 1$ corresponds to placing vehicles at every available point on every lane (which is not really possible given that $n' < W$). To demonstrate that the parity heuristic is asymptotically optimal in this setting, it can be shown that $\rho_\infty^{\text{par}} \leq \chi/(1 + \chi) \leq \rho_\infty^{\text{opt}}$. This is a consequence of the following two lemmas, whose proofs are given below.

Lemma 3.6.1. *Given any uniform traffic crossing instance on the torus with congestion χ, $\rho_\infty^{\text{par}} \leq \chi/(1 + \chi)$.*

Proof. Consider intersection q in the system and any lane passing through q. By unwrapping the torus within the plane, one can think of the vehicles moving towards q as an infinite sequence of blocks that repeats with period W. For any $k \geq 1$, consider the vehicles in this lane whose initial positions are within distance kW of q and are directed towards q. There are $kn' = kWp$ such vehicles, organized into identical blocks of length W. For $1 \leq j \leq kn'$, let x_j denote the distance from the

initial position of the jth vehicle to q.

To obtain an upper bound on the delay, allow each of these vehicles to pass through q only if it is on-parity. (The parity algorithm may allow off-parity vehicles to pass through if there is no imminent collision, so this assumption results in the highest possible delay.) Let t_j denote the time at which vehicle j passes through q according to the parity heuristic.

The following bound on t_j is established first:

$$t_j \leq \max_{1 \leq i \leq j}(x_i + 1 + 2(j - i)).$$

This follows by a simple induction argument. If $j = 1$, then $t_j \leq x_j + 1$, which matches the time for this vehicle to reach q together with one optional unit of delay if it is out of parity. Otherwise, observe that if the delays prior to vehicle j do not affect it, then $t_j \leq x_j + 1$. If they do, then vehicle j will pass through q two units after t_{j-1}, thus yielding

$$\begin{aligned} t_j &\leq \max(x_j + 1, t_{j-1} + 2) \\ &\leq \max(x_j + 1, \max_{1 \leq i \leq j-1}(x_i + 1 + 2(j - 1 - i)) + 2) \\ &= \max(x_j + 1, \max_{1 \leq i \leq j-1}(x_i + 1 + 2(j - i))) = \max_{1 \leq i \leq j}(x_i + 1 + 2(j - i)). \end{aligned}$$

While the maximum is taken over j choices of i, one can simplify this due to the periodic nature of the system. Suppose that i is the index that achieves the maximum in the definition of t_j. Let i' denote the index of the corresponding vehicle in a block

that is closer to q. That is, $i = i' + k'n'$ and $x_i = x_{i'} + k'W$, for some $k' \geq 1$. Thus

$$\begin{aligned} x_i + 1 + 2(j-i) &= (x_{i'} + k'W) + 1 + 2(j - (i' + k'n')) \\ &= x_{i'} + 1 + 2(j - i') - k'(2n' - W) \\ &= (x_{i'} + 1 + 2(j - i')) - k'W(2p - 1). \end{aligned}$$

Therefore, if $\chi = 0$ (meaning that $2p - 1 \leq 0$), one may assume that i achieves its maximum value among the n' vehicles immediately preceding j, that is $j - i \leq n'$. (Since using values of i' from earlier blocks can only decrease the value in the max.) Conversely, if $\chi > 0$ (meaning that $2p - 1 > 0$), i achieves its maximum among the n' vehicles that are closest to q. Let $k' = \lceil j/n' \rceil$ denote the index of the block that contains x_j. One can conclude that if $\chi = 0$, then since $x_i \leq x_j$,

$$t_j \;\leq\; x_j + 1 + 2n' \;\leq\; k'W + 1 + 2n' \;\leq\; k'W + 2W.$$

On the other hand, if $\chi > 0$, then $x_i \leq W$, $j - i \leq (k' - 1)n'$ implying that

$$t_j \;\leq\; W + 1 + 2k'n' \;\leq\; k'(2n') + 2W \;=\; k'W(2p) + 2W.$$

Note that $2p > 1$ if and only if $\chi > 0$. Therefore, one can conclude that

$$t_j \;\leq\; k'W \cdot \max(1, 2p) + 3W \;\leq\; k'W(1 + \chi) + 3W.$$

Because vehicle j is at distance x_j from q and every time unit beyond x_j contributes

to the delay, the delay is $t_j - x_j$. The delay rate is $(t_j - x_j)/t_j = 1 - x_j/t_j$. Since $x_j \geq k'W$, the delay rate for each vehicle x_j is

$$1 - \frac{x_j}{t_j} \;\leq\; 1 - \frac{k'W}{k'W(1+\chi) + 3W} \;=\; 1 - \frac{1}{(1+\chi) + 3/k'}.$$

For the asymptotic delay rate, take the limit as $k' \to \infty$, which is at most

$$1 - \frac{1}{1+\chi} \;=\; \frac{\chi}{1+\chi},$$

as desired. □

Lemma 3.6.2. *Given any uniform traffic crossing instance on the torus with congestion χ, $\rho_\infty^{\text{opt}} \geq \chi/(1+\chi)$.*

Proof. Let q be any intersection of the system. By unwrapping the torus within the plane, one can think of the vehicles moving towards q as an infinite sequence of blocks that repeats with period W. For any $k \geq 1$, consider the vehicles on the two lanes incident to q whose initial positions are within distance kW. There are $kn' = kWp$ vehicles in each lane. At most one vehicle can pass through q at any time, therefore the total time T for all the vehicles to travel through q is at least $2kn' = 2kWp$.

As observed in the proof of Lemma 3.6.1, the delay experienced by any vehicle is the difference of its time to arrive at q minus its distance from q. Therefore, at least one vehicle among this set (in particular, the last vehicle in one of the two lanes), experiences a delay of at least $\max(0, T - x)$, where x is its distance from q.

Since $x \leq kW$, the maximum delay is at least

$$\max(0, 2kWp - kW) \;=\; \max(0, kW(2p-1)) \;=\; kW\chi.$$

By Lemma 3.6.1 (where j plays the role of the last vehicle in block k to make it through q) one has $T \leq kW(1+\chi) + 3W$ for the parity algorithm, and therefore it can be no higher for the optimum. The delay rate is the delay divided by the time, which is at least

$$\frac{kW\chi}{kW(1+\chi)+3W} \;=\; \frac{\chi}{1+\chi+3/k}.$$

To obtain the asymptotic delay rate, consider the limit as k tends to infinity, which yields a lower bound on the asymptotic delay rate of at least $\chi/(1+\chi)$, as desired. □

While the proofs are somewhat technical, the intuition behind them is relatively straightforward. If $\chi = 0$, then while local delays may occur, there is sufficient capacity in the system for them to dissipate over time, and hence the asymptotic delay rate tends to zero as well. On the other hand, if $\chi > 0$, then due to uniformity and the cyclic nature of the system, delays will and must grow at a predictable rate. The following main result of this section is an immediate consequence of the above lemmas.

Theorem 3.6.2. *Given a uniform traffic crossing instance on the torus, the asymptotic delay rate of the parity heuristic is optimal.*

Chapter 4: Coordinating City-Wide Traffic with Modular Circulation

4.1 Introduction

Minimum (and maximum) cost network flows and the related concept of circulations are fundamental computational problems in discrete optimization. This chapter introduces a variant of the circulation problem, where vertex demand values are taken from the integers modulo λ, for some integer $\lambda \geq 2$. For example, if $\lambda = 10$ a vertex with demand 6 can be satisfied by any net incoming flow of 6, 16, 26 and so on or a net outgoing flow of 4, 14, 24, and so on. The motivation in studying this problem stems from an application in synchronizing the traffic lights of an urban transportation system.

Throughout, let $G = (V, E)$ denote a directed graph, and let $\lambda \geq 2$ be an integer. Each edge $(u, v) \in E$ is associated with a non-negative integer *weight*, $wt(u, v)$, and each vertex $u \in V$ is associated with a *demand*, $d(u)$, which is an integer drawn from $\mathbb{Z}_\lambda$, the integers modulo λ. Let f be an assignment of values from $\mathbb{Z}_\lambda$ to the edges of G. For each vertex $v \in V$, define

$$f_{\text{in}}(v) \;=\; \sum_{(u,v)\in E} f(u,v) \qquad \text{and} \qquad f_{\text{out}}(v) \;=\; \sum_{(v,w)\in E} f(v,w),$$

and define the *net flow* into a vertex v to be $f_{\text{in}}(v) - f_{\text{out}}(v)$. Define f to be a *circulation with λ-modular demands*, or *λ-CMD* for short, if it satisfies the *modular*

flow-balance constraints, which state that for each $v \in V$,

$$f_{\text{in}}(v) - f_{\text{out}}(v) \equiv d(v) \pmod{\lambda}.$$

Observe that a demand of $d(v)$ is equivalent to the modular "supply" requirement that the net flow out of this vertex modulo λ is $\lambda - d(v)$.

Define the *cost* of a circulation f to be the weighted sum of the flow values on all the edges, that is,

$$\text{cost}(f) = \sum_{(u,v)\in E} wt(u,v) \cdot f(u,v).$$

Given a directed graph G and the vertex demands d, the *λ-CMD problem* is that of computing a λ-CMD of minimum cost. (Observe that there is no loss in generality in restricting the flow value on each edge to $\mathbb{Z}_\lambda$, since the cost could be reduced by subtracting λ from this value without affecting the flow's validity.)

The standard minimum-cost circulation problem (without the modular aspect) is well studied and is discussed in any of a number of standard sources on this topic, for example, [35–40]. In contrast, λ-CMD is complicated by the "wrap-around" effect due to the modular nature of the demand constraints. A vertex's demand of $d(u)$ units can be satisfied in the traditional manner by having a net incoming flow of $d(u)$, but it could also be met by generating a net outgoing flow of $\lambda - d(u)$ (not to mention all variants thereof that involve adding multiples of λ).

The main results of this chapter are:

- 2-CMD can be solved exactly in polynomial time (see Section 4.4).
- λ-CMD is NP-hard, for $\lambda \geq 3$ (see Section 4.5).
- There is a polynomial time $4(\lambda - 1)$-approximation to λ-CMD (see Section 4.6).

In Section 4.2 the relevance of the λ-CMD problem to traffic-management is discussed, Section 4.3 presents some preliminary observations regarding this problem, and finally, each of the three main results are given in Sections 4.4–4.6.

4.2 Application to Traffic Management

The motivation in studying the λ-CMD problem arises from an application in traffic management. In urban settings, intersections are the shared common resource between vehicles traveling in different directions, and their control is essential to maximizing the utilization of a transportation network [91]. There are numerous approaches to modeling traffic flow and diverse computational approaches to solve and analyze the associated traffic management problems. Dresner and Stone [45] use a multi-agent systems approach, in which vehicles request a reservation from the intersection, which in turn allocates each vehicle a time slot. Vehicles must then alter their speed to cross the intersection during their reserved time. Yu and LaValle [4] draw a connection between multi-agent path planning on collision-free unit-distance graphs and network flows. Despite the popular interest in automated traffic systems, there has been relatively little work on this problem from the perspective of algorithm

design.

Chapter 3 considered the problem of scheduling the movements of a collection of vehicles through an unregulated crossing. The approach was based on the idealized assumption that the motion of individual vehicles in the system is controlled by a central server. A more practical approach is based on aggregating vehicles into groups, or *platoons*, and planning the motion of these groups [89, 92].

The problem is viewed here in this aggregated form, but from a periodic perspective. Consider an urban transportation network consisting of a grid of horizontal and vertical roads as laid out on a map. Each pair of horizontal and vertical roads meets at a unique intersection controlled by a traffic light that alternates between horizontal and vertical traffic, such that the pattern repeats over a time interval λ. It is assumed throughout that λ has been discretized to a reasonably small integer value, say in terms of seconds or tens of seconds.

More formally, a traffic-light schedule is λ-*periodic* if it repeats every λ time units. Throughout, consider a traffic management system of the foreseeable future where the traffic light schedule is transmitted to the vehicles, which in turn may adjust their speeds to avoid excessive waiting at intersections. While vehicles may turn at intersections, the schedule is designed to minimize the delay of straight-moving traffic.

To motivate the connection with modular circulations, consider a four-sided city block (see Fig. 4.1). Let a, b, c, and d denote the intersections, and let t_{ab}, t_{bc}, t_{cd}, t_{ad} denote the travel times between successive intersections along each road segment. (Practical issues such as acceleration are ignored.) If the road segment is

oriented counterclockwise around the block (as shown in the example), these travel times are positive, and otherwise they are negative. Suppose that the traffic-light schedule is λ-periodic, and that at time $t = t_a$ the light at intersection a transitions so that the eastbound traffic can move horizontally through the intersection (see Fig. 4.1(a)). In order for these vehicles to proceed without delay through intersection b, this light must transition from vertical to horizontal at time $t_b = t_a + t_{ab}$ (see Fig. 4.1(b)).

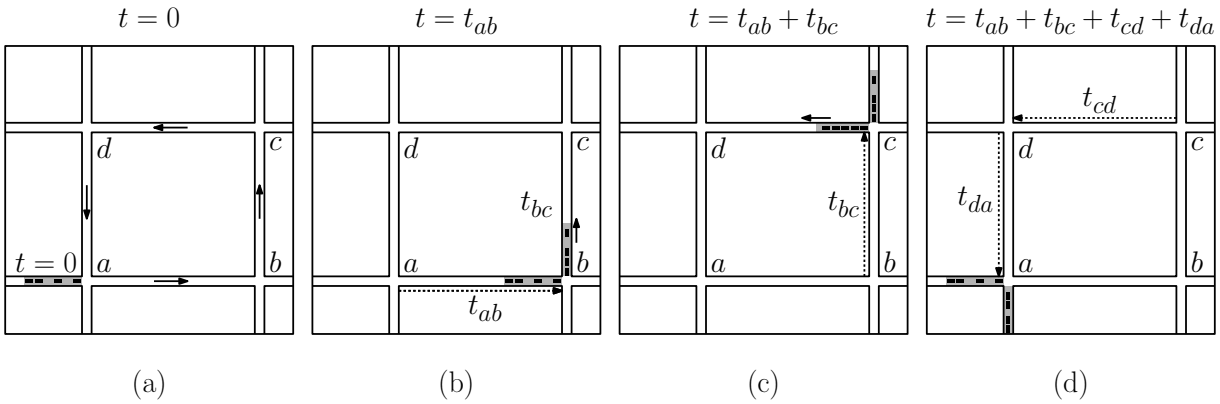


Figure 4.1: Periodic constraints from a delay-free traffic-light schedule.

Reasoning analogously for the other intersections, it follows that the vertical-to-horizontal transition times for intersections c and d are $t_c = t_b + t_{bc}$ and $t_d = t_c + t_{cd}$, respectively (see Fig. 4.1(c)). On returning to a (see Fig. 4.1(d)), it can be seen that

$$t_a \equiv t_a + (t_{ab} + t_{bc} + t_{cd} + t_{da}) \pmod{\lambda}.$$

Thus, in order to achieve delay-free flow around the intersection in a λ-periodic

context, one must satisfy the constraint

$$t_{ab} + t_{bc} + t_{cd} + t_{da} \equiv 0 \pmod{\lambda}.$$

While traffic-light scheduling is essentially cyclic in nature, it should be noted that the model ignores many practical issues that arise in real traffic-management systems. Nonetheless, it is noteworthy that even distilled to its simplest elements, this problem is far from trivial. A natural question involves the choice of the integer parameter λ. This models the length of the time period shared by the synchronized traffic lights, but in what units is it expressed? Ideally, a traffic manager would like to have the greatest degree of flexibility in choosing λ, since delays must be rounded to integer multiples of λ. For example, if the traffic lights cycle every minute, then to achieve a resolution of seconds in the delay values, set $\lambda = 60$. Because the approximation bounds degrade as λ increases, it may be better to decrease the minimum time interval to, say, 15 seconds. Then one could set $\lambda = 60/15 = 4$, resulting in more accurate approximation bounds. The price paid is that delays would now be rounded up to multiples of 15-second intervals.

Chapter 3 covers what could be considered the special case where $\lambda = 2$ and opposite sides of the block have equal travel times, and shows that a simple alternating strategy is optimal. It is not reasonable, however, to assume that all the blocks in the city will satisfy this requirement. Furthermore, the assumption that $\lambda = 2$ does not admit a more nuanced timing of the lights. In order to handle the more general case, an (ideally small) nonnegative delay $\delta_{ij} \geq 0$ is introduced along

each road segment ij. This yields the new constraint

$$(t_{ab} + \delta_{ab}) + (t_{bc} + \delta_{bc}) + (t_{cd} + \delta_{cd}) + (t_{da} + \delta_{da}) \equiv 0 \pmod{\lambda},$$

or equivalently, if one defines $T = t_{ab} + t_{bc} + t_{cd} + t_{da}$ to be the sum of (signed) travel times of the road segments around this block, it yields

$$\delta_{ab} + \delta_{bc} + \delta_{cd} + \delta_{da} \equiv -T \pmod{\lambda}. \tag{4.1}$$

The upshot is that if vehicles travel at a reduced speed so that the transit time along each of the road segments includes the associated delay, then the straight-line vehicular traffic along each road need never pause or wait at any traffic signal. The objective is to minimize the sum of delay values over all the road segments in the network, referred to as the *total delay*.

More formally, the transportation network is modeled as a set of horizontal and vertical roads. This defines a directed grid graph whose vertices are the intersections, whose edges are the *road segments*, and whose (bounded) faces are the *blocks* of the city. For each pair of adjacent intersections i and j, let t_{ij} denote the delay-free travel time along this road segment. For each block u, define the *total signed travel time* about u to be the sum of the travel times for each of the road segments bounding u, where the travel time is counted positively if the segment is oriented counterclockwise about u and negatively otherwise. Let $T(u)$ denote this value modulo λ. A λ-*periodic traffic-light schedule* assigns a *delay* to each road segment so that for each block,

these delays satisfy Eq. (4.1). The objective is to minimize the *total delay*, which is defined to be the sum of delays over all the segments in the network.

To express this in the form of an instance of λ-CMD, let $G = (V, E)$ denote the directed dual of the graph, by which it is meant that the vertex set V consists of the city blocks, and there is a directed edge $(u, v) \in E$ if the two blocks are incident to a common road segment, and the direction of the road segment is counterclockwise about u (and hence, clockwise about v). The demand of each vertex u, denoted $d(u)$, is set to $T(u)$, and the weight of each edge is set to unity.

There remains one impediment to linking the λ-periodic traffic-light schedule and the λ-CMD problems. The issue is that the delay associated with any road segment (which may be as large as $\lambda - 1$) can be significantly larger than the time to traverse the road segment. If so, the capacity of the road segment to hold the vehicles that are waiting for the next signal may spill backwards and block the preceding intersection. In order to deal with this issue without complicating the model, the assumption that λ is smaller than the time to traverse any road segment is added. The link between the two problems is presented in the following lemma.

Lemma 4.2.1. *Given a transportation network and integer $\lambda \geq 2$, let G be the associated directed graph with vertex demands and edge weights as described above.*

(*i*) *If there exists a λ-periodic traffic-light schedule with total delay Δ, then there exists a λ-CMD for G of cost Δ.*

(*ii*) *If there exists a λ-CMD for G of cost Δ and for all road segments ij, $t_{ij} \geq \lambda$, then there exists a λ-periodic traffic-light schedule with total delay Δ.*

Proof. To prove (i), consider any λ-periodic traffic-light schedule. Define a flow on G where the edge (u, v) carries flow equal to the delay on the associated road. The net flow into a vertex u of G is equal to the sum of delays for the road segments oriented clockwise about u and the sum of the negated delays for the road segments oriented counterclockwise. Given any block with (counterclockwise) intersections a, b, c, and d, the net flow into the associated node is $-(\delta_{ab} + \delta_{bc} + \delta_{cd} + \delta_{da})$. By the same reasoning behind Eq. (4.1), this sum is equivalent to $T(u)$ modulo λ. It follows directly that f is a λ-CMD. In both cases the cost is the sum of delays.

To prove (ii), let f denote any λ-CMD for the directed dual graph G. Fix any intersection and set its transition time to 0 modulo λ. The transition times for the remaining intersections can be set by a simple propagation process. In particular, if the transition time of an intersection i is known, then the transition time of an adjacent intersection j is delayed relative to i (modulo λ) by δ_{ij}, which is the flow along the associated dual edge. By Eq. (4.1) it follows that whenever this propagation loops back to an intersection whose delay was set, the propagated transition time is equivalent (modulo λ) to the original transition time. By the assumption that $t_{ij} \geq \lambda$, the vehicles that entered this road segment during the last traffic-light cycle have sufficient space[1] that they can effectively "park" themselves within the road segment for δ_{ij} time units until moving through the next intersection. □

[1]The term "space" is being used abstractly. To relate space and time, introduce a maximum speed limit, call it σ. Assuming vehicles move at their maximum speed, the total length of a vehicle platoon that can pass through any intersection in time λ is $\sigma\lambda$. To move these vehicles to the next intersection in time t_{ij} at full speed, the road segment between these intersections is of length σt_{ij}. Since $t_{ij} \geq \lambda$, then $\sigma t_{ij} \geq \sigma\lambda$, and, irrespective of the choice of σ, it follows that there is sufficient space to park these vehicles, whatever their actual sizes may be.

4.3 Preliminaries

This section presents a few definitions and observations that will be used throughout the rest of the chapter. Given an instance $G = (V, E)$ of the λ-CMD problem, consider any subset $V' \subseteq V$. Let $G' = (V', E')$ be the associated induced subgraph of G, and let $d(V')$ denote the sum of demands of all the nodes in V'. The edges in E' are referred to as the *internal edges* of this subgraph, and the edges of G that cross the cut $(V', V \setminus V')$ are referred to as the *interface*. Given such a subgraph and any flow f on G, define its *internal flow* to be only the flow on the internal edges, and define the *internal cost* to be the cost of the flow restricted to these edges. Define the *interface flow* and *interface cost* analogously for the interface edges. Define $f_{\text{in}}(V')$ to be the sum of flow values on the interface edges that are directed into V', and define $f_{\text{out}}(V')$ analogously for outward directed edges. The following lemma provides necessary and sufficient conditions for the existence of a λ-CMD.

Lemma 4.3.1. *Given an instance $G = (V, E)$ of the λ-CMD problem:*

(*i*) *For any induced subgraph $G' = (V', E')$ and any λ-CMD f,*

$$f_{\text{in}}(V') - f_{\text{out}}(V') \equiv d(V') \pmod{\lambda}.$$

(*ii*) *If G is weakly connected, then a λ-CMD exists for G if and only if $d(V) \equiv 0 \pmod{\lambda}$.*

Proof. Assertion (i) follows by applying standard results on circulations to the modular context. The "only if" part of assertion (ii) is a special case of (i), where $G' = G$. To see the "if" part of assertion (ii), consider any path (ignoring the edge directions) between two vertices u and v of nonzero demand in G. For any x, $0 \leq x < \lambda$, one can push x units of flow from u to v by pushing x units of flow along each forward-directed edge of the path and $\lambda - x$ units of flow along each backward-directed edge. (Each intermediate vertex has a net incoming flow of either 0, λ or $-\lambda$, depending on the orientation of the incident edges.) By repeating this, the demands of all the vertices in the graph can be satisfied. □

It follows from this lemma that the λ-CMD instance associated with any traffic-light scheduling problem has a solution. The reason is that each edge (u, v) contributes its travel time t_{uv} positively to $d(u)$ and negatively to $d(v)$, and therefore the sum of demands over all the vertices of the network is zero, irrespective of the travel times.

4.4 Polynomial Time Solution to 2-CMD

This section shows that 2-CMD, referred to here also as *binary CMD*, can be solved in polynomial time by a reduction to minimum-cost matching in general graphs. Intuitively, the binary case is simpler because the edge directions are not significant. If a vertex is incident to an even number of flow-carrying edges (whether directed into or out of this vertex), then the net flow into this vertex modulo λ is zero, and otherwise it is one. Thus, solving the problem reduces to computing a

minimum-cost set of paths that connect each pair of vertices of nonzero demand, which is essentially a minimum-cost perfect matching in a complete graph whose vertex set consists of the subset of vertices of nonzero demand and whose edge weights are the distances between vertices, ignoring edge directions. The remainder of this section is devoted to providing a formal justification of this intuition.

Recall that $G = (V, E)$ is a directed graph, and $d(v)$ denotes the demand of vertex v. Since $\lambda = 2$, for each $v \in V$, it is the case that $d(v) \in \{0, 1\}$. Let $G' = (V, E')$ denote the graph on the same vertices as G but with directions removed from all the edges. One may assume that G' is connected, for otherwise it suffices to solve the problem independently on each connected component of G' and combine the results. The weight of each edge of G' is set to the weight of the corresponding edge of G. If there are two oppositely directed edges joining the same pair of vertices, the weight is set to the minimum of the two.

Let $U = U(G)$ denote the subset of vertices of V whose demand values are equal to 1. By Lemma 4.3.1(ii), it can be assumed that $d(V) \equiv 0 \pmod{\lambda}$. Therefore, $d(V)$ is even, which implies that $|U|$ is also even. For each $u, v \in U$, let $\pi(u, v)$ denote the shortest weight path between them in G', and let $wt(\pi(u, v))$ denote this weight. Define $\widehat{G} = (U, \widehat{E})$ to be a complete, undirected graph on the vertex set U, where for each $u, v \in U$, the weight of this edge is $wt(\pi(u, v))$. (This is well defined by the assumption that G' is connected.) Since $\widehat{G}$ is complete and has an even number of vertices, it has a perfect matching. The reduction of 2-CMD to the minimum-cost perfect matching problem is implied by the following lemma.

Lemma 4.4.1. *Given an instance $G = (V, E)$ of the 2-CMD problem, the minimum cost of any 2-CMD for G is equal to the minimum cost of a perfect matching in $\widehat{G}$.*

Proof. Begin with the assertion that any 2-CMD f for G can be mapped to a collection of paths in the undirected graph G' corresponding to edges of f that carry nonzero flow, such that each $u \in U$ is incident to at least one such path. This is well known when dealing with traditional flows, but it does not generally hold for CMDs when $\lambda > 2$. It will be shown that it holds in the binary case.

To establish the assertion, let f denote a 2-CMD for G. Each edge of G carries either a flow of 0 or 1. Observe that the net flow into each vertex of U (respectively, $V \setminus U$) is odd (respectively, even). Consider the following process, which will output a collection of paths while gradually reducing f to an empty flow. Start with any vertex v that is incident to an odd number of flow-carrying edges. Compute any maximal path in G' by following edges that carry nonzero flow. (The edge directions are not important, because any incident edge that carries flow toggles the net flow's parity between even and odd.) Clearly, such a path must terminate at a different vertex u that is also incident to an odd number of flow-carrying edges, and hence is also in U. Output this path $\pi(u, v)$ from v to u and modify f by setting the flow values of the edges along this path to 0.

Observe that the resulting set Π of paths is incident to every vertex of U, and their total cost is equal to the cost of f. Also, the sum of edge weights along each path between vertices u and v is at most the cost of the shortest path between them, that is, $wt(\pi(u, v))$.

To establish the lemma, the above path-reduction procedure is applied. Each path between two vertices u and v corresponds to an edge of $\widehat{G}$. After reducing the flow values along the path between such a pair of vertices, these two vertices are now incident to an even number of edges carrying nonzero flow. Thus, each vertex of U will appear in exactly one of the output paths, implying that the final set of paths defines a perfect matching in $\widehat{G}$ (whose vertex set is U). This yields $\text{cost}(f) = \sum_{\pi(u,v)\in\Pi} wt(\pi(u,v))$, which is the cost of the perfect matching.

Conversely, given any perfect matching in $\widehat{G}$, one can generate a 2-CMD by pushing a single unit of flow along the edges of the shortest path between u and v, for each pair (u,v) of the matching. This is a valid 2-CMD, since each vertex of U is incident to an odd number of edges carrying one unit of flow, and each vertex of $V \setminus U$ is incident to an even number of edges carrying one unit of flow. Note that if paths share common edges, then the total flow value along this edge may exceed 1, but if so it can be reduced to either 0 or 1. It follows that the cost of the 2-CMD is no larger than the cost of the matching. □

This approach is similar to the classical solution by Edmonds and Johnson [93] to the Chinese-Postman Problem. Vertices of odd demand here play the same role as vertices of odd degree in their work. Since $\widehat{G}$ is dense, a minimum-cost perfect matching can be constructed in $O(|U|^3)$ time. The graph can be computed in $O(n^3)$ time, where $n = |V|$, by applying the Floyd-Warshall algorithm for computing shortest paths [36]. Thus, the overall running time is $O(n^3)$, yielding the following result:

Theorem 4.4.1. *It is possible to solve the 2-CMD problem in $O(n^3)$ time on any instance $G = (V, E)$, where $n = |V|$.*

4.5 Hardness of λ-CMD

This section presents the following hardness result for λ-CMD:

Theorem 4.5.1. *For $\lambda \geq 3$, the λ-CMD problem is NP-hard.*

The reduction is from positive 1-in-3-SAT [94]. For the sake of brevity and simplicity, the proof for the case $\lambda = 3$ is shown first, but the method easily generalizes, as shown in Section 4.5.4.

To begin, let F denote a boolean formula in 3-CNF, where each literal is in positive form. Throughout, for $\alpha \in \{0, 1, \ldots, \lambda - 1\}$, the term α-vertex is used to denote a vertex whose demand is α. The reduction involves two principal components, a *variable gadget* which associates truth values with the variables of F and a *clause gadget* which enforces the condition that exactly one variable in each clause is assigned the value **True**.

4.5.1 Variable Gadget

Before discussing the general gadget, a fundamental building block from which all variables will be constructed is described. The block consists of six vertices, three 1-vertices and three $(\lambda - 1)$-vertices (i.e., 2-vertices), connected together with edges as shown in Figure 4.2(a). Edges connecting 1-vertices have weight $wt(u, v) = 1.5$, while all other edges are of weight $wt(u, v) = 1$.

If a flow of 1 is sent from each 2-vertex to its connected 1-vertex, the 2-vertices overflow and all demands are satisfied with $cost(f) = 3$ (see Figure 4.2(b)). This flow, in which there is no flow across the interface edges, represents a logical value of **False**.

If instead a flow of 1 is sent across the interface edges, then the demands of the 2-vertices are satisfied. A flow of 1 across each edge originating at the central 1-vertex will cause it to overflow and will satisfy each of the connected 1-vertices. This flow, in which each interface edge carries a flow of 1, represents a logical value of **True** and again has $cost(f) = 3$ (see Figure 4.2(c)).

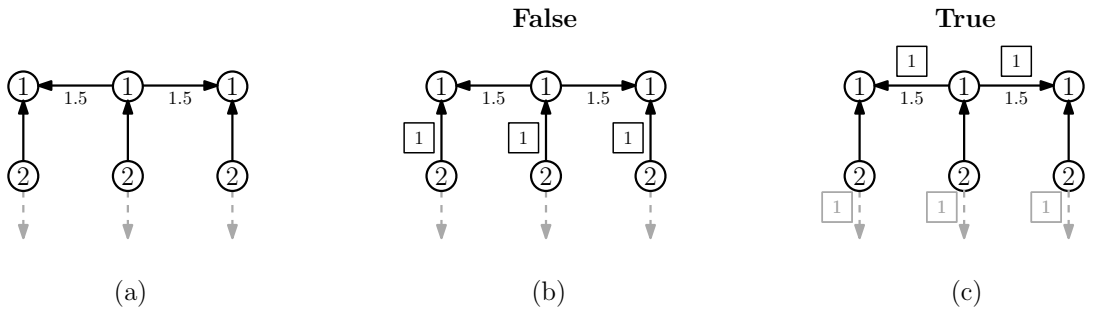


Figure 4.2: (a) The fundamental building block used to build variable gadgets. Interface edges are dashed gray segments. (b,c) CMDs representing the assignment of **False** and **True** values, respectively, with the flow values in boxes.

If every interface edge of a variable gadget carries the same flow and that flow is either 0 or 1, that variable is said to be *interface-consistent*. If all variables are consistent for a given flow, then that flow is said to be *variable-consistent*. Notice that both logical values above are realized via interface-consistent flows.

Lemma 4.5.1. *Given a fundamental block, a satisfying flow has cost* ≤ 3 *if and only if that flow is interface-consistent.*

Proof. Each 2-vertex can only be satisfied by: (1) sending a flow of 1 across one of

its edges or (2) sending a flow of 2 across both of its edges (in both cases the vertex's demand overflows).

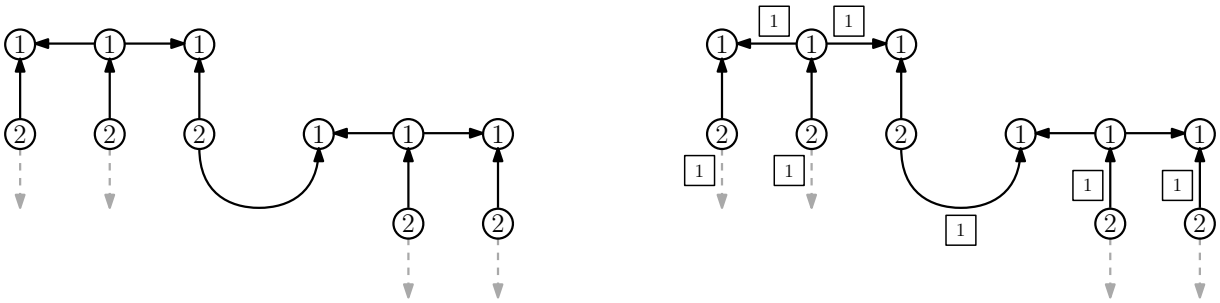
In the second case, the 2-vertex sends a flow of 2 to its neighboring 1-vertex. As per Lemma 4.3.1(i), that vertex now requires a flow of 2 across its other edge in order to have its demand satisfied. Together these flows come at a cost of 5 (one of these edges has a weight of 1.5), therefore no 2-vertex may be satisfied by a flow greater than 1 without a cost greater than 3.

There exists a satisfying flow for a fundamental block if and only if the total flow across its interface edges is equivalent to 0 (mod λ) (see Lemma 4.3.1). Given this and the fact that no single interface flow may equal 2, the flows across the interface edges must either all be 0 or all be 1, i.e., the overall flow must be interface-consistent for it to be a satisfying flow. □

As variables may appear in multiple clauses, a mechanism by which existing variables can be expanded is required. For this, an expansion module is created, any number of which can be added to a variable so that there are three interface edges for each clause in which that variable appears.

To understand how this module functions, first look at the case in which two fundamental blocks are connected together. This connection occurs through a shared 2-vertex, so that what was an interface edge for one block becomes the connection between a 2-vertex and 1-vertex in the other block (see Figure 4.3). Recall that the value assignment of a variable is determined by the direction of flow from the 2-vertices, with flow along the interface representing **True** and internal flow (i.e.,

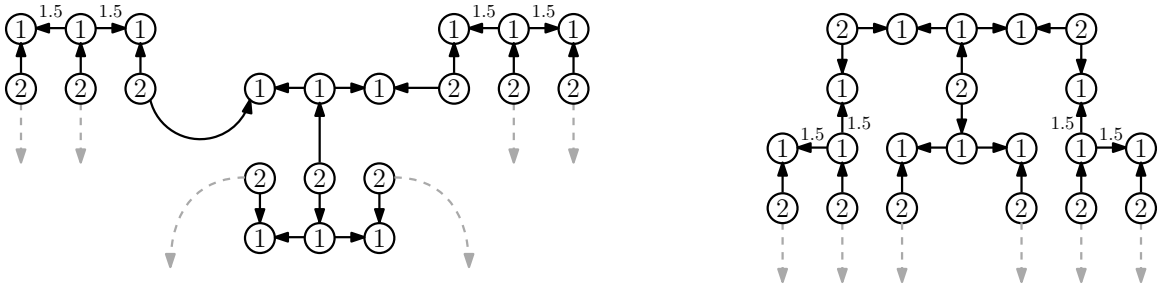
flow to the connected 1-vertices) representing **False**. Because the outgoing edges of the shared 2-vertex are simultaneously an interface edge of one block and an internal edge of the other, pushing a flow across either edge will assign opposing values to the blocks.



(a) (b)

Figure 4.3: Two fundamental blocks connected via a shared 2-vertex, with figure (b) showing a flow that is satisfying but not interface-consistent.

Knowing this, the module is constructed as a double-negative, ensuring that it is assigned the same value as the variable it extends. A fundamental block is used as a hub, and to this hub is attached two more fundamental blocks (see Figure 4.4). When attached to a variable, this module creates four new interface edges and consumes one, thus extending the variable by three interface edges.



(a) (b)

Figure 4.4: (a) A fundamental block with a single expansion module attached. (b) The same structure rearranged to emphasize the two clause outputs, each with three interface edges.

While the structure of the fundamental blocks is as described above, the weighting must be adjusted to maintain equal costs between the **True** and **False** states. Rather than each fundamental block having two edges of weight 1.5, only the rightmost block in the expansion module has such edges; all others are of weight 1. In this way, the minimal cost of a consistent satisfying flow across the module is 8, regardless of the value assigned to the variable. This is easily verified by assigning a truth value to the gadget (fixing an interface-consistent flow on the interface edges of 0 or 1) and then traversing the structure, satisfying the demand in each vertex by assigning flow to its unused edge.

Given this, Lemma 4.5.1, and the fact that the expansion module is constructed from fundamental blocks, yields the following:

Lemma 4.5.2. *Given an expansion module, there exists an interface-consistent flow of internal cost* 8 *(in either the* **True** *or* **False** *cases), and any other satisfying flow has a strictly larger internal cost.*

To construct a gadget for a variable v_i, appearing in $c(v_i)$ clauses, begin with a fundamental block and connect $c(v_i) - 1$ expansion modules to it (see Figure 4.5). Doing so provides $c(v_i)\lambda$ interface edges and yields the following result:

Lemma 4.5.3. *Given a variable gadget, there exists an interface-consistent flow of internal cost* $3 + 8[c(v_i) - 1]$ *(in either the* **True** *or* **False** *cases), and any other satisfying flow has a strictly larger internal cost.*

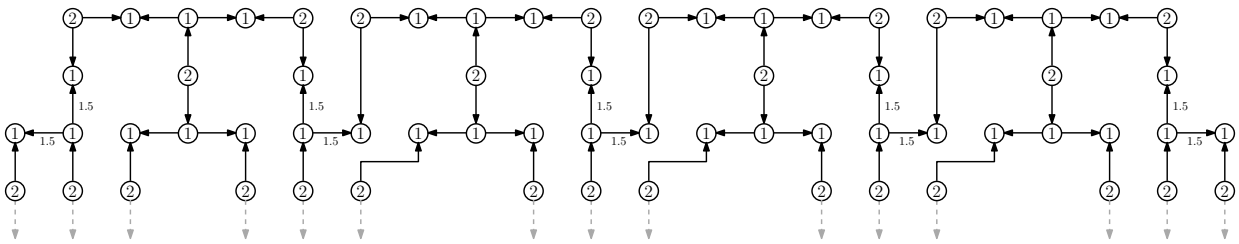


Figure 4.5: An example of a large variable gadget with five clause outputs.

4.5.2 Clause Gadget

The basis for the clause gadget is a single 1-vertex with three incoming edges, one for each literal. These edges have a weight $wt(u, v) = \gamma$ and are connected to the appropriate variables as their outgoing interface edges. If a single literal is **True**, one of these edges will carry a flow of 1, satisfying the demand of the clause vertex. If more literals are **True**, the demand underflows and the vertex is left unsatisfied. It is possible to satisfy the vertex by creating flows on these edges greater than 1, but such flows can be made cost-prohibitive by setting the edge weights γ sufficiently high.

Recall that each variable gadget produces λ copies of its respective variable (three interface edges in this example) per clause in which it appears. Because of this, the clause gadget must also be created in triplicate. Every clause consists of three 1-vertices, each with an incoming edge from its three literals (see Figure 4.6). Their weighting and behavior are as described above. Since there are no internal edges in the clause gadgets, they do not contribute to the cost of the flow (but their interface edges will).

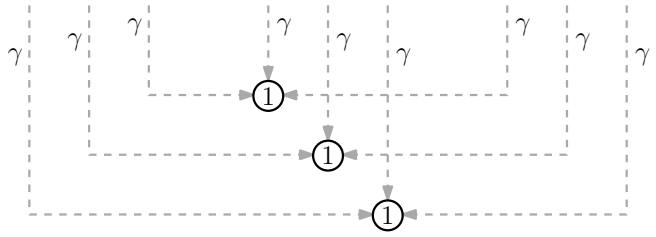


Figure 4.6: A full clause gadget, with three inputs from each of three literals.

4.5.3 Final Construction

Each variable in F is represented by a fundamental block connected to $c(v_i) - 1$ expansion modules, creating $c(v_i)\lambda$ outputs. Thus, λ outputs are linked to each of the appropriate clause gadgets (see Figure 4.7). The size of the variable gadget is a linear function of the number of clauses in which that variable appears and can thus be constructed in polynomial time.

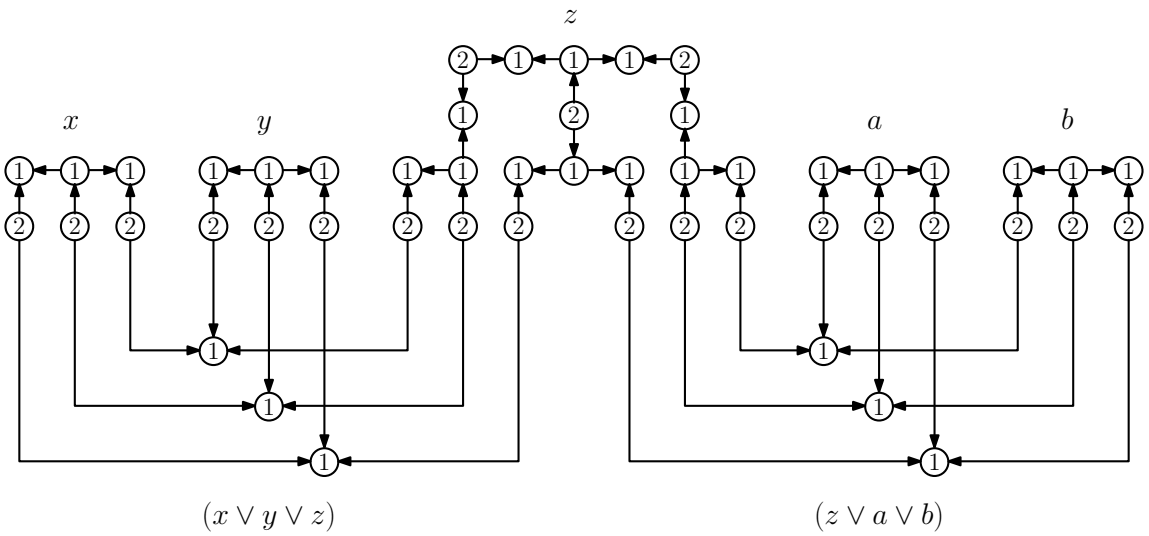


Figure 4.7: An example of a reduction from the formula $F = (x \vee y \vee z) \wedge (z \vee a \vee b)$. Note that the weights have been removed for legibility.

If F is satisfiable, then a 3-CMD exists that is variable-consistent. In this case, each fundamental block incurs a cost of 3, and each expansion module incurs an

additional cost of 8 for a flow representing a consistent truth value across its interface and the interfaces of the modules/fundamental block to which it is attached, as per Lemma 4.5.2.

For each clause, create λ 1-vertices, each connected to the clause's three literals by incoming edges. As there are no edges between these vertices, there is no flow possible within the clause gadget, resulting in an internal cost of 0. The size of the clause gadget is constant.

Finally, the flow on the edges between the variable gadgets and clause gadgets has yet to be counted as they are interface edges for both gadgets. Each clause gadget contains λ 1-vertices, with each receiving a flow of 1 across edges of weight γ. Thus, these add a cost of $3|C|\gamma$, where $|C|$ is the number of clauses in F.

If F is not satisfiable, then some set of variables must have inconsistent outputs in order to create a valid CMD. As shown in Lemma 4.5.1, these inconsistencies will always lead to a strictly greater cost. Thus:

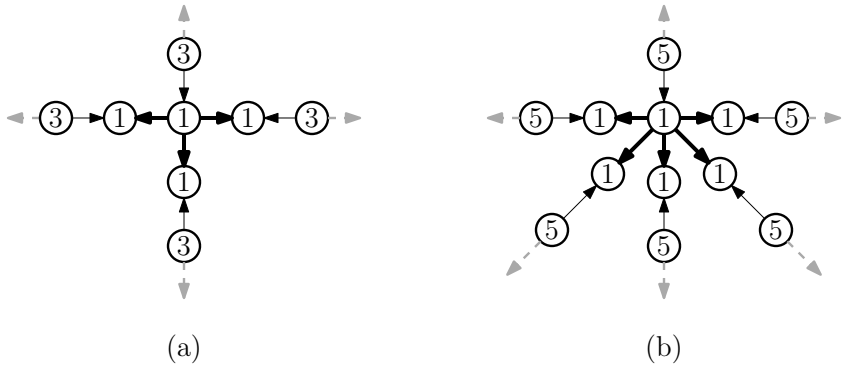
Lemma 4.5.4. *Given a positive boolean formula F in 3-CNF, in polynomial time it is possible to construct an instance of 3-CMD that has a satisfying flow with*

$$cost(f) \leq \sum_{v_i \in V} (3 + 8[c(v_i) - 1] + 3|C|\gamma)$$

if and only if F is 1-in-3 satisfiable.

4.5.4 Generalizing for $\lambda > 3$

Fundamental blocks are constructed in the same manner as described in the $\lambda = 3$ example in Section 4.5.1, but for completeness, they are described here in their general form. A fundamental block begins with λ 1-vertices, connected as a star graph (i.e., a central vertex connected to each of the remaining vertices) with edges directed outward. Each 1-vertex is also connected to a $(\lambda - 1)$-vertex via an incoming edge. Finally, these $(\lambda - 1)$-vertices each have an outgoing interface edge (Fig. 4.8).



(a) (b)

Figure 4.8: Fundamental blocks with (a) $\lambda = 4$ and (b) $\lambda = 6$.

The expansion module, too, remains largely unchanged. It consists of a central fundamental block with λ interface edges. A fundamental block is then connected to all but one of these edges, with the remaining edge reserved for connecting to the existing variable gadget. For each expansion module added to a variable gadget, $(\lambda-1)^2-1$ interface edges are added (one interface edge is consumed when connecting to the variable gadget, hence the -1 in the equation). See Figure 4.9. Just as before, the weights of a small subset of edges must be adjusted so that **True** and **False** assignments incur the same cost. To do so, arbitrarily select a fundamental block

that is not the hub (i.e., not the fundamental block which connects to the existing variable gadget) and set the outgoing edges from its central 1-vertex to a weight of $\frac{2\lambda-3}{\lambda-1}$.

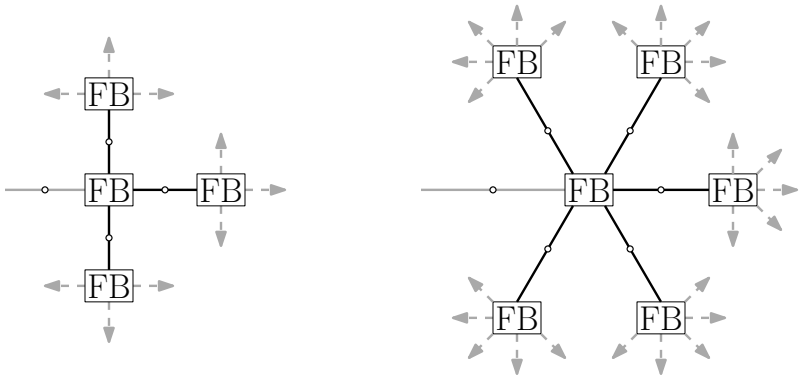


Figure 4.9: Expansion modules built from fundamental blocks (FB) with (a) $\lambda = 4$ and (b) $\lambda = 6$. Edges between fundamental blocks are represented with a disc and no direction as a reminder that the blocks are connected via a shared $(\lambda - 1)$-vertex.

Recall that the variable gadget described in Section 4.5.1 creates $c(v_i)\lambda$ interface edges. In truth, the fundamental block creates λ interface edges while each of the $c(v_i) - 1$ expansion modules add $(\lambda - 1)^2 - 1$ interface edges. When $\lambda = 3$, these formulae are equivalent, each variable has $c(v_i)\lambda$ interface edges, and each of the $c(v_i)$ clause gadgets will connect to λ interface edges.

When $\lambda > 3$, the variable gadgets will have $\lambda + [c(v_i) - 1]\,[(\lambda - 1)^2 - 1]$ interface edges. As this is not evenly divisible by $c(v_i)$, connecting to the clause gadgets becomes problematic. To correct for this, a modified expansion module, called a *seed module*, is used to start each variable gadget rather than a fundamental block. Rather than connecting to an existing variable gadget, the edge that was previously reserved for this connection is instead attached to one of the existing interface edges

(recall that in actuality a $(\lambda - 1)$-vertex is being shared, not that two edges are being connected directly). Doing so creates a seed module that has $(\lambda - 1)^2 - 1$ interface edges (Fig. 4.10) and, when used in conjunction with unaltered expansion modules, creates a variable gadget with $c(v_i)\,[(\lambda - 1)^2 - 1]$ interface edges.

Clause gadgets are constructed as described in Section 4.5.2, with the caveat that they require $(\lambda - 1)^2 - 1$ vertices rather than λ. Each vertex will still only have three incoming edges, as each clause will only ever have three literals, and a demand of 1.

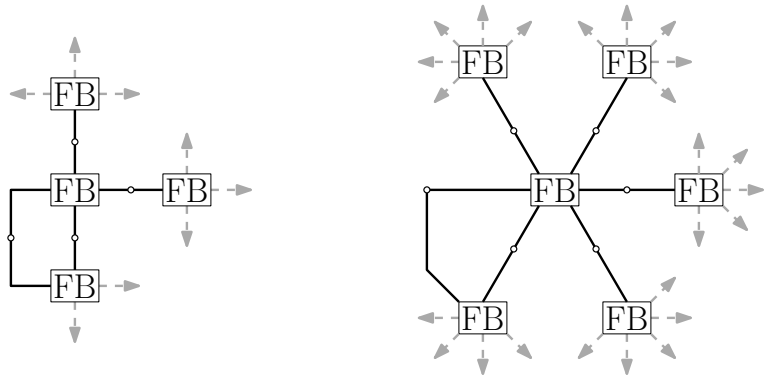


Figure 4.10: Seed modules for creating variable gadgets with (a) $\lambda = 4$ and (b) $\lambda = 6$.

Despite a small increase in the size of each gadget, the reduction can still be done in polynomial time.

Finally, the cost threshold in Lemma 4.5.4 must be generalized. In the general form, if F is satisfiable there will be a satisfying flow with

$$\text{cost}(f) \;\leq\; \sum_{v_i \in V} \left[(\lambda^2 - 1)c(v_i) + \lambda|C|\gamma\right]$$

4.6 Approximation Algorithm

This section presents a $4(\lambda-1)$-factor approximation to the λ-CMD problem for $\lambda \geq 2$. Before presenting the algorithm, some terminology is introduced. Consider an instance $G = (V, E)$ of the λ-CMD problem, with vertex demands d. Let $G' = (V, E')$ be (as defined in Section 4.4) the undirected version of G. Assume that G' is connected. Let $U = U(G)$ denote the subset of vertices of V whose demand values are nonzero. Define $SMT(U)$ to be a *Steiner minimal tree* in G' whose terminal set is U (that is, a connected subgraph of G' of minimum weight that contains all the vertices of U).

As in Section 4.4, define $\widehat{G} = (U, \widehat{E})$ to be the complete, undirected graph over the vertex set U, where for each $u, v \in U$, the weight of this edge is the weight of a minimum weight path between u and v in G'. Given any $U \subseteq V$, let $MST(U)$ denote any minimum spanning tree on the subgraph of G' induced on U. Standard results on Steiner and minimum spanning trees yields the following:

Lemma 4.6.1. *For any $U \subseteq V$, $wt(MST(U)) \leq 2 \cdot wt(SMT(U))$.*

Define a *balanced partition* to be a partition $\{U_1, \ldots, U_k\}$ of U such that for $1 \leq i \leq k$, the total demand within U_i (that is, $d(U_i)$) is equivalent to zero modulo λ. By Lemma 4.3.1(ii), one may assume that $d(V) \equiv 0 \pmod{\lambda}$, and so there is always a trivial partition, namely $\{V\}$ itself. Define $\text{cost}(U_i)$ to be $\text{cost}(SMT(U_i))$, and define the *cost* of a balanced partition to be the sum of costs over its components. A *minimum balanced partition* for G is a balanced partition of minimum cost. The

following lemma establishes the connection between balanced partitions and minimum modular circulations.

Lemma 4.6.2. *Consider an instance $G = (V, E)$ of λ-CMD. Let $\Psi = (U_1, \ldots, U_k)$ denote a minimum balanced partition of G, as defined above, and let f denote any minimum cost λ-CMD for G. Then* $\text{cost}(\Psi) \leq |f| \leq (\lambda - 1) \cdot \text{cost}(\Psi)$.

Proof. Given any λ-CMD f for G, let $E'_f \subseteq E'$ denote the corresponding edges of the undirected graph G' that carry nonzero flow. This set of edges induces a set of connected components such that each vertex of U lies within one of these components. These components define a partition of U. By Lemma 4.3.1(i), the sum of demands within each connected component is a multiple of λ, which implies that these connected components define a balanced partition for G; call it Ψ_f. Since each edge that carries flow carries at least one unit of flow, it follows that $|f| \geq \text{cost}(\Psi_f)$. This establishes the lower bound on $|f|$.

In order to prove the upper bound, the mapping of Ψ into a λ-CMD that satisfies the desired bound will be shown. The construction builds a separate flow f_i for each subset U_i, and the final flow is simply the sum of all these flows. Let $T_i = SMT(U_i)$. It will simplify matters to first describe the construction in terms of the undirected graph G', and then modify it to apply to G.

Begin by rooting T_i at any one of the vertices of U_i. The construction operates in a bottom-up manner by performing a post-order traversal of T_i. For each vertex u visited, let $f(u)$ denote the net flow into u from its children (or 0 if u is a leaf). If $d(u) - f(u) < 0$ (this is a net-supply vertex), direct $(d(u) - f(u)) \bmod \lambda$ units

of flow from u to its parent, and if $d(u) - f(u) > 0$ (this is a net-demand vertex), direct $(d(u) - f(u)) \bmod \lambda$ units of flow from its parent to u. Because the sum of demands within this component is a multiple of λ, the net flow into the root r must equal $d(r)$ modulo λ. It is easy to see that the net flow into every vertex of T_i is equivalent to zero modulo λ, and thus it is a valid λ-CMD. It follows directly that by applying the process to every component of Ψ one obtains a λ-CMD for G', which is denoted by f'_Ψ. Because each edge carries a flow of at most $\lambda - 1$, it can be seen that $|f'_\Psi| \leq (\lambda - 1) \cdot \text{cost}(\Psi)$.

One can convert f'_Ψ to a λ-CMD for the directed graph G as follows. Consider any edge (u, v) of any subtree T_i where u is the child and v is the parent. If the direction of this edge in G matches the direction of the flow, then there is no change. Otherwise, replace the flow value of x on this edge with the flow value of $\lambda - x$, essentially negating the value of the flow. It is easy to see that the resulting flow satisfies the modular balance constraints at each vertex, and therefore the result is a λ-CMD for G. By the same reasoning as above, $|f_\Psi| \leq (\lambda - 1) \cdot \text{cost}(\Psi)$. □

By the above lemma, it suffices to compute a balanced partition for G of low cost. Next, a simple approximation algorithm that outputs a balanced partition whose cost is within a factor of 4 of the optimum is presented.

The construction begins with the metric closure $\widehat{G}$ defined above. In a manner similar to Kruskal's algorithm, sort the edges of $\widehat{G}$ in increasing order, and start with each vertex of $\widehat{G}$ in a separate component. All these components are labeled as *active*. Process the edges one by one, letting (u, v) denote the next edge being processed. If

u and v are in distinct components, and both components are active, merge these components into a single component. If the sum of the demands of the vertices within this component is equivalent to zero modulo λ, label the resulting component as *finished*, and output its set of vertices. Because the total sum of demands of all the nodes is equivalent to zero modulo λ, it follows that every vertex is placed within a finished component, and therefore the algorithm produces a balanced partition of $\widehat{G}$ (and by extension, a balanced partition of G).

This algorithm has the same running time as Kruskal's algorithm. (Observe that one can associate each component with its sum of demands, thus enabling the ability to determine the sum of merged components in constant time.) The following lemma establishes the approximation factor for this construction.

Lemma 4.6.3. *Let Ψ' denote the balanced partition generated by the above algorithm, and let Ψ denote the optimum balanced partition. Then* $\text{cost}(\Psi') \leq 4 \cdot \text{cost}(\Psi)$.

Proof. The proof begins by modifying the optimum balanced partition Ψ into a form that will be easier to analyze. Let $\{U_1, \ldots, U_k\}$ denote the subsets of Ψ, and for $1 \leq i \leq k$, let T_i denote the minimum spanning trees of the induced subgraph of U_i within the metric closure $\widehat{G}$. By standard results on Steiner trees, it follows that $\text{cost}(T_i) \leq 2 \cdot \text{cost}(U_i)$. Henceforth, costs are measured in terms of the minimum spanning trees over the components. In particular, define $\text{cost}'(\Psi) = \sum_i \text{cost}(T_i)$, and so $\text{cost}'(\Psi) \leq 2 \cdot \text{cost}(\Psi)$. Because the cost of the minimum Steiner tree does not exceed the cost of the minimum spanning tree, $\text{cost}(\Psi') \leq \text{cost}'(\Psi')$.

For each pair of components of $U_i, U_j \in \Psi$, let w_i and w_j denote the maximum

weights of any edges of T_i and T_j, respectively. If there exists an edge of $\widehat{G}$ that connects two vertices, one in U_i and one in U_j, such that the weight of this edge is not greater than $\min(w_i, w_j)$, then merge the vertices of U_i and U_j into a single component. Repeat this process until there is no inter-component edge that satisfies this property. Clearly, the resulting set of components, denoted by Ψ'', is a balanced partition.

The claim, then, is that $\text{cost}'(\Psi'') \leq 2 \cdot \text{cost}'(\Psi)$. This follows from a simple charging argument. Consider the smallest weight edge that satisfies the merging condition. Each of the two components U_i and U_j that are connected by this edge each have edges with weights w_i and w_j, respectively, both of which are as large as the connecting edge. Add this connecting edge and charge it to the smaller of w_i and w_j. The other edge remains to be charged for future merges. In this manner, every time two components are merged, an edge from one of the components pays for the newly added connecting edge, and the other remains for future charges. Therefore, only the original edges are charged, and no edge is charged more than once. It follows that $\text{cost}'(\Psi'') \leq 2 \cdot \text{cost}'(\Psi)$, as desired.

Consider the following assertion: Ψ' is a refinement of Ψ'', meaning that each set U_i' of Ψ' is a (not necessarily proper) subset of some set U_j'' of Ψ''. This is proven by contradiction. Suppose that there was an edge (u, v) such that u and v are in the same subset Ψ', but they are in different components of Ψ''. Assume that (u, v) is a minimum weight edge with this property. By definition of Ψ'', the weight of (u, v) is strictly smaller than the weights of all the edges of either U_i'' or U_j'' (possibly both). Assume without loss of generality that U_i'' satisfies this property. It follows that at

the time that edge (u, v) is considered by this algorithm, all the edges of $MST(U_i'')$ have been considered, which implies that the algorithm has discovered all the edges of this spanning tree. Since (u, v) is the smallest inter-component edge, there can be no other vertices that are part of this component. Since Ψ'' is a balanced partition, the sum of the vertices in this component sum to zero modulo λ, which implies that this component would have been labeled as *finished* by the algorithm, contradicting the hypothesis that the edge (u, v) was considered for insertion by the algorithm.

By the above assertion, each component of Ψ' is a subset of some component of Ψ'', implying that $\text{cost}'(\Psi') \leq \text{cost}'(\Psi'')$. In conclusion,

$$\text{cost}(\Psi') \;\leq\; \text{cost}'(\Psi') \;\leq\; \text{cost}'(\Psi'') \;\leq\; 2 \cdot \text{cost}'(\Psi) \;\leq\; 4 \cdot \text{cost}(\Psi),$$

as desired. □

Combining Lemmas 4.6.2 and 4.6.3(ii), it follows that the presented algorithm achieves an approximation factor of $4(\lambda - 1)$. While obtaining the best running time has not been a focus of this work, it is easy to see that this procedure runs in polynomial time. Let $n = |V|$. The graph $\widehat{G}$ can be computed in $O(n^3)$ time by the Floyd-Warshall algorithm [36]. The Kruskal-like algorithm for computing the balanced partition can be performed in $O(n^2 \log n)$ time, as can the algorithm of Lemma 4.6.2. Thus, the overall running time is $O(n^3)$.

Theorem 4.6.1. *Given an instance $G = (V, E)$ of the λ-CMD problem for $\lambda \geq 2$, it is possible to compute a $4(\lambda - 1)$-approximation in time $O(n^3)$, where $n = |V|$.*

Chapter 5: Online Algorithms for Warehouse Management

5.1 Introduction

Online shopping has grown rapidly in recent years and, as such, the efficiency of the warehouses and fulfillment centers that support it plays an increasingly important role. Several companies have developed automated systems to help streamline operations in these warehouses, drive down the costs of order fulfillment, and increase overall efficiency. The introduction of automation comes with the opportunity for new theoretical models and computational problems with which to better understand and optimize such systems.

These systems often maintain a warehouse of standardized portable storage units, which are stored and retrieved by robots [95, 96]. For example, Amazon's Kiva robots and Alibaba's Quicktron robots help to streamline the order-fulfillment process. The Amazon robots are 16 inches tall, weigh almost 145 kilograms, can travel at 5 mph, and carry a payload weighing up to 317 kilograms. These robots maneuver themselves under standardized shelving units, lift them from below, and carry them to a location in the warehouse where a human waits to complete an order with items from the shelf.

The frequency with which each storage unit is accessed varies, and so, intuitively, units that are accessed more often should be placed closer to the access points than those that are less frequently accessed. As access probabilities vary over time, there

is a natural question of how to dynamically organize the warehouse's placement of storage units in order to guarantee efficient access at any time. The work presented here develops simple computational models for a "self-organizing warehouse", and subsequently presents online algorithms for solving them. It will be demonstrated that these algorithms are competitive with optimal algorithms. This work can be viewed as a geometric variant of online algorithms for self-organizing lists and virtual memory management systems [83, 84].

From a practical perspective there are many ways in which to model objects residing in a warehouse. In order to obtain meaningful theoretical results without imposing irrelevant technical details, what is proposed here is a very simple and general model, which encapsulates the most salient aspects of efficient self-organizing behavior. Storage units, or *boxes*, are modeled as movable unit squares on a grid in the plane. In addition to the boxes, there are designated fixed points, called *access points*, where boxes are brought on demand. The input consists of a sequence of *access requests*, each specifying that a particular box in the system be moved to a given access point.

There are two natural ways in which to move boxes in a planar setting, picking them up (as an overhead crane might) and sliding them along the ground (like the aforementioned robotic systems). The former is simpler to describe and analyze. The latter is more realistic and is consistent with other motion-planning models [33, 77]. Another issue is the geometrical configuration of the warehouse and the locations of the access points. This work presents clean and simple models based on infinite and semi-infinite grids and shows how to generalize these solutions to rectangular

warehouses.

Two versions of the problem are considered: the *attic problem*, where there is a single access point, and the *warehouse problem*, where there are multiple access points. In each version and for each motion type, an online algorithm is presented that is competitive with respect to an optimal solution that has knowledge of the entire access sequence. Details of the problem formulations and results are given in the following section.

5.1.1 Model and Results

In this work, a warehouse is modeled as a rectangular subset Ω of $\mathbb{Z}^2$, the square grid in the plane. Throughout, distances are measured in the ℓ_1 metric (the sum of absolute differences in x and y coordinates). As well, there is a finite set $A = \{a_1, \ldots, a_m\} \subseteq \Omega$ of stationary *access points* and a (significantly larger) finite set $B = \{b_1, \ldots, b_n\}$ of portable storage units, called *boxes*. Each box is a unit square. At any time, a box's lower left corner coincides with a grid point in Ω, called its *location*. A point of Ω that contains a box is said to be *occupied*, and otherwise it is *unoccupied*. No two boxes may occupy the same location at the same time.

The initial layout of the boxes is specified in the input. This is followed by a sequence of *access requests*, each being a pair (b, a), which involves moving box $b \in B$ from its current location to access point $a \in A$. Access requests are processed *sequentially*, meaning that each request is completed before the next one is started. Since the access point may already be occupied, it will be necessary to reorganize

box locations. This reorganization should be performed with care, keeping frequently accessed boxes near the access point and moving less frequently accessed boxes to the periphery. The challenge is that one does not know the future access sequence, and yet one wishes to be competitive with an optimal algorithm that does.

In general, the reorganization following each access request will involve a sequence of box movements. The box at the access point is displaced to a nearby location, the box at this location is then displaced to a new location, and so on. This chain of box movements continues until the last box in the chain arrives at an unoccupied square of the grid, possibly the original location of the requested box. More formally, let p_0 denote the original location of b, and let p_1 denote the location of a. If a is not occupied, b is simply moved here, and the sequence is complete. Otherwise, the algorithm determines a chain $p_2, \ldots, p_k$ of locations, where $p_2, \ldots, p_{k-1}$ are occupied and p_k is unoccupied (see Fig. 5.1(a)). (Note that p_0 is considered to be unoccupied, because its box has been moved to the access point.) Call this a *reorganization chain*. If $p_k \neq p_0$, this is an *open chain* (see Fig. 5.1(b)), and otherwise it is a *closed chain* (see Fig. 5.1(c)).

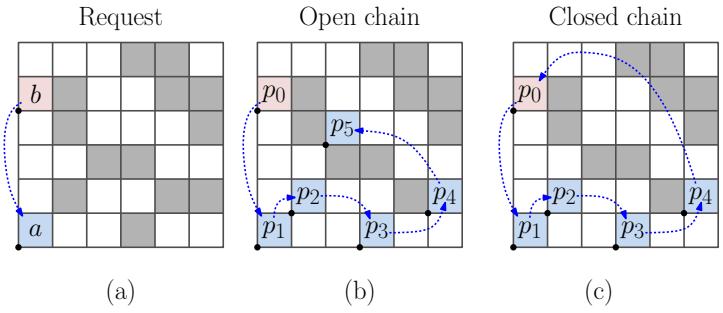


Figure 5.1: Processing a request.

For the sake of presenting the devised algorithms, it will be useful to describe

the relocation process in terms of a sequence of *motion primitives*. In the case where boxes can be picked up (as by an overhead crane), the primitive operation is a *swap*, which exchanges the contents of two grid squares. The *cost* of the operation is the ℓ_1 distance between the two locations. The aforementioned reorganization involving a chain $\langle p_0, \ldots, p_k \rangle$ (whether open or closed) can be executed by swapping boxes in reverse order along the chain, that is, $p_k \leftrightarrow p_{k-1} \leftrightarrow \cdots \leftrightarrow p_0$ (see Fig. 5.2(a)).

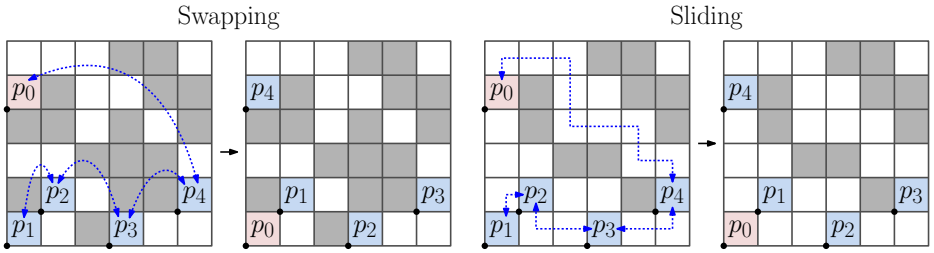


Figure 5.2: Motion primitives.

Alternatively, when boxes are moved along the ground the associated primitive operation is called *sliding*. As with swapping, the contents of two grid locations are exchanged, but the boxes are moved along a rectilinear path of unoccupied grid locations (see Fig. 5.2(b)). The *cost* of the operation is the ℓ_1 length of the path, which may generally be much higher than the ℓ_1 distance between the two locations.

Sliding motion is more relevant in contexts where the boxes are being moved by robots, but it is complicated by the need to create empty space in which to move boxes. This work will first present a simple swapping-based solution and then show how to adapt this to sliding motion without significantly increasing the cost. These two primitives provide a conceptually clear and simple model of motion costs. Of course, in practice, many other realistic issues would need to be considered.

The following problem formulations involve a *problem instance*, which consists of a specification of the domain Ω and the locations of the m access points A. An input to a given instance consists of the initial locations of the n boxes followed by a sequence S of access requests. For each access request, the output consists of the sequence $\langle p_0, \ldots, p_k \rangle$ along which motion primitives are applied (either swapping or sliding, depending on the model). Since the current focus is on reorganization strategies, a number of issues needed for a complete model are being ignored, such as how to coordinate the movement of multiple robots. This work focuses on two versions of the problem depending on the number of access points (see Fig. 5.3):

Attic Problem: Ω is an axis-aligned rectangle containing a single access point.

Warehouse Problem: Ω is an axis-aligned rectangle with access points along its bottom side.

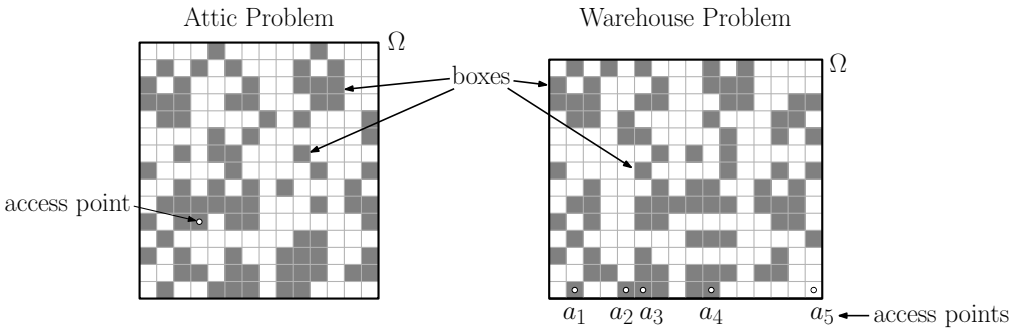


Figure 5.3: Problem versions.

The above problems are being considered in an *online* setting, which means that each access request is processed without knowledge of future requests. In contrast, in an *offline* setting the entire access sequence is known in advance. An online algorithm is said to be *c-competitive* for a constant $c \geq 1$, called the *competitive*

ratio, if for all sufficiently long access sequences S, the total cost of this algorithm is at most a factor c larger than the cost of an optimal offline solution for the same sequence. An algorithm is *competitive* if it is c-competitive for some constant c, independent of m, n, the size of the domain, and the length of the access sequence. (The competitive ratios that result from these analyses are relatively high, and are likely far from tight. Reducing them would involve establishing better lower bounds on the optimum algorithm, and this seems to be quite challenging.) The notion of "sufficiently long access sequence" allows start-up issues to be ignored, such as the initial locations of the boxes.

The main results are competitive online algorithms for these two problems in both the swapping- and sliding-motion models (presented in Theorems 5.2.1, 5.2.2, 5.3.1, and 5.3.2). The result for the attic problem has the additional feature of being asymptotically optimal with respect to box density. (The precise definition will be given in Section 5.2.3.) These online algorithms exploit an intriguing connection between the presented warehouse problems and the task of maintaining hierarchical memory systems [83]. Hierarchical memory systems are linear in nature, and the geometric context of the problems studied in this dissertation introduces novel challenges, since the reorganization must take into account the 2-dimensional locations of the boxes. Also, when sliding is involved, it is necessary to manage the set of unoccupied squares to guarantee short access paths.

5.2 Online Solution to the Attic Problem

This section presents an online algorithm for the attic problem (single access point). It will be shown that the resulting scheme is competitive with respect to an optimal algorithm. As mentioned above, this is achieved by utilizing ideas from hierarchical memory systems. In such systems, memory consists of objects called *pages*, which are organized into blocks, called *caches*. Successive caches have higher storage capacity but higher access times. A common method for organizing such memory structures involves a block-based version of the least-recently used (LRU) policy, called *Block-LRU* by Aggarwal et al. [83]. In this policy, whenever a page is accessed it is brought to the lowest level cache, and the page that has resided in this cache for the longest time is evicted to the next higher level cache. The process is repeated until reaching the lowest cache that has space to hold this page, possibly the cache that contained the originally requested page. The following section describes how the Block-LRU algorithm can be adapted to the geometric setting herein.

5.2.1 Hierarchical Model

In hierarchical memory systems, the cost of accessing an object is purely a function of each cache's speed. In the presented geometric context, the cost depends on the total cost of the motion primitives, which depends on the ℓ_1 distances between the locations of the boxes in the reorganization chain. The principal challenge is adapting the cache-based cost to the geometric setting. The presented approach to the attic problem is based on surrounding the access point by collections of nested

regions, called *containers*. Analogous to caches in the hierarchical memory systems, containers that are closer to the access point provide faster access but have lower storage capacity compared with those farther out.

It will simplify matters to describe the solution first for the infinite grid. The following *hierarchical model* is defined: an infinite sequence of nested *containers*, $C_0, C_1, \ldots$, where C_0 consists only of the origin (the access point), and for $k \geq 1$, C_k consists of the points of $\mathbb{Z}^2$ whose ℓ_1 distance from the origin varies from $2^{k-1}+1$ to 2^k (see Fig. 5.4 below). Whenever a box b is requested, it is first moved to the access point, and then a series of *evictions* takes place, where, for $k = 0, 1, \ldots$, a box from container C_k is moved to container C_{k+1}. The precise manner in which this is done for swapping and sliding motions is explained in Sections 5.2.2 and 5.2.3, respectively.

5.2.2 Online Algorithm for Swapping Motion

This section presents an online algorithm solving the attic problem in the case of swapping motion, called Block-LRU$_A$. Consider a request for a box b. If the access point is unoccupied, the box is simply moved there. Otherwise, in order to make space for b, the least-recently accessed box from C_0, C_1, and so on is evicted until one encounters the first container C_k that has at least one unoccupied location (including possibly b's location at the time of the request). More formally, let p_b denote b's location, let p_0 denote the access point (origin), and let $p_1, \ldots, p_{k-1}$ denote the locations of the least-recently used boxes of containers C_1 through C_{k-1}, respectively.

Finally, let $p_k \in C_k$ denote the final unoccupied location (possibly the former location of b). As described in Section 5.1.1, this is achieved by performing swaps in reverse order $p_k \leftrightarrow p_{k-1} \leftrightarrow \cdots \leftrightarrow p_0 \leftrightarrow p_b$ (see Fig. 5.4(a)). The cost is the sum of the ℓ_1 distances between consecutive pairs.

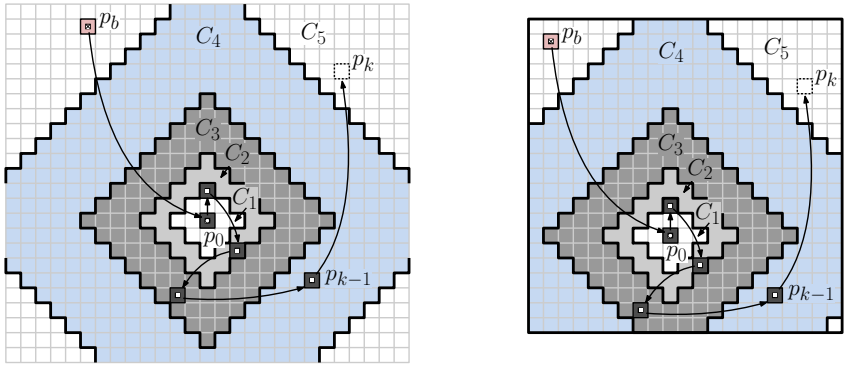


Figure 5.4: (a) Nested containers for the attic problem and (b) restriction to a rectangular domain.

In order to apply this to a rectangular domain Ω, the boundary of the containers is simply clipped at the limits of Ω (see, e.g., Fig. 5.4(b)). Next, this will be shown to be competitive.

Theorem 5.2.1. *For any instance of the attic problem and any sufficiently long access sequence S, the cost of Block-LRU$_A(S)$ is within a constant factor of the cost of an optimal solution, assuming swapping motion.*

Proof. Consider an input S consisting of the initial box placements and a sequence of access requests. Let $T_{\text{opt}}(S)$ and $T_{\text{lru}}(S)$ denote the total cost of the optimum and Block-LRU$_A$ solutions, respectively, on this input. It will be shown that there exists a constant c and quantity $f(S)$ that does not grow with the length of the access sequence, such that $T_{\text{lru}}(S) \leq cT_{\text{opt}}(S) + f(S)$. Since $f(S)$ does not grow with the

length of the access sequence, for all sufficiently long access sequences its impact on the total cost will be negligible compared to $T_{\text{opt}}(S)$.

The following analysis will be based on an auxiliary statistic. Given any container C_k, define an *eviction* to be an event in which a box lying within this container is moved to a location in an enclosing container $C_{k'}$, for $k' > k$. For the given access request sequence S, define $E_{\text{lru}}(S, k)$ to be the total number of evictions from container C_k performed by Block-LRU$_A$. Let $W_{\text{lru}}(S) = \sum_{k \geq 0} 2^k E_{\text{lru}}(S, k)$ denote the weighted cost of these evictions. It will be shown that there exist constants c_1 and c_2 and quantities $f_1(S)$ and $f_2(S)$ that do not grow with the length of the access sequence, such that the following two inequalities hold:

$$(1)\ T_{\text{lru}}(S) \ \leq \ c_1 W_{\text{lru}}(S) + f_1(S) \quad \text{and} \quad (2)\ W_{\text{lru}}(S) \ \leq \ c_2 T_{\text{opt}}(S) + f_2(S).$$

First, inequality (1) is proven. Observe that the cost of processing a request involving a box b in Block-LRU$_A$ consists of two parts, the cost of moving b to the access point (i.e., the ℓ_1 distance between b and the access point) plus the cost of performing the evictions caused by this move. It suffices to bound only the latter quantity. To see why, consider two consecutive requests to b. Just after the first request, b is located at the access point. When the second request occurs, if b is not at the access point, it has been moved away due to various evictions involving b that have occurred due to intervening access requests. By the triangle inequality, the sum of the costs of these evictions involving b is at least as large as the ℓ_1 distance of b from the access point at the time of the second request. Thus, the cost of moving b to the access point for the second request is not greater than the cost of

evictions involving b due to intervening requests. All the requests for b except the first can thereby be accounted for. Define $f_1(S)$ to be the sum of the ℓ_1 of every box's initial location to the access point. Clearly, $f_1(S)$ depends only on the initial box placements.

It remains to bound the cost needed to process the evictions. Each time Block-LRU$_A$ evicts a box from some container C_k to the enclosing container C_{k+1}, the cost is bounded above by the maximum distance between any point of C_k to any point in C_{k+1}. Clearly, this is not greater than the diameter of C_{k+1}, which is 2^{k+2}. Summing over all accesses and all containers, it follows that the total cost of Block-LRU$_A$ evictions is at most $\sum_{k\geq 0} 2^{k+2} E_{\text{lru}}(S,k) = 4W_{\text{lru}}(S)$. By the earlier observation that the cost of bringing boxes back to the access point is bounded above by the sum of $f_1(S)$ and the total eviction cost, it follows that $T_{\text{lru}}(S) \leq c_1 W_{\text{lru}}(S) + f_1(S)$, where $c_1 = 2 \cdot 4 = 8$, thus establishing (1).

To prove inequality (2), a technique will be applied similar to one given by Sleator and Tarjan [84] and Aggarwal et al. [83] for hierarchical memory systems. For any $k \geq 0$, define $\overline{C}_k = \bigcup_{j\leq k} C_j$ (that is, the set of points within distance 2^k of the origin). Also define $m_k = |C_k|$ and $\overline{m}_k = |\overline{C}_k|$, denoting the total capacities of these sets. For each $k \geq 2$, the weighted eviction cost of Block-LRU$_A$ on container C_k with respect to the cost of box movements will be related to the optimal solution within container C_k. The overall analysis comes about by summing over all container levels.

Fix any $k \geq 2$. Partition the access request sequence into contiguous *segments*, such that within any segment (except possibly the last), Block-LRU$_A$ performs $\overline{m}_k$

evictions from container C_k. (The last segment will not be analyzed, but since there is only one such segment for each k from which an eviction was performed, it follows that for all sufficiently long access segments, the impact on the overall cost of these segments is negligible. See Sleator and Tarjan [84] for more details.) Consider any complete segment. The contribution of the evictions of this segment from C_k to the weighted eviction cost $W_{\text{lru}}(S)$ is $2^k\overline{m}_k$. In Block-LRU$_A$ every container C_j for $j \leq k$ evicts the least recently accessed box, and this implies that any box evicted from container C_k is the least recently accessed box not only from C_k, but from $\overline{C}_k$ as well. It is asserted here that during this segment, the number of distinct boxes accessed must be at least $\overline{m}_k$. To see why, observe that either all of the boxes evicted during this segment are distinct, or some box was evicted twice during the sequence. If there are $\overline{m}_k$ distinct evictions, then there are at least $\overline{m}_k$ distinct boxes requested. On the other hand, if a box is evicted twice, then by the nature of Block-LRU$_A$, between these two evictions, every one of the $\overline{m}_k$ boxes in $\overline{C}_k$ must have been accessed in order for this box to transition from the most recent to the least recent.

Now consider how the optimum algorithm deals with the $\overline{m}_k$ distinct box requests that have occurred during this segment. Intuitively, because of the exponential increase in container sizes, most of the $\overline{m}_k$ distinct accessed boxes cannot fit within $\overline{C}_{k-1}$, and hence they must spill out into the surrounding region. The work needed for the spillover will be charged for but be limited to C_k (to avoid double counting).

It will simplify matters to ignore boundary issues for now and consider the unbounded case where $\Omega = \mathbb{Z}^2$. Define $\widehat{C}_k$ to be the set of points of the infinite grid that lie within ℓ_1 distance $(3/4)2^k$ of the access point. Since $k \geq 2$, therefore

$\overline{C}_{k-1} \subset \widehat{C}_k \subset \overline{C}_k$. Let $\widehat{m}_k = |\widehat{C}_k|$, yielding $\widehat{m}_k \leq c'\overline{m}_k$, where $c' \approx (3/4)^2 \leq 2/3$. Thus, a fraction of $1-c'$ or roughly one-third of the $\overline{m}_k$ distinct boxes accessed during this sequence must spill out from $\overline{C}_{k-1}$ to an ℓ_1 distance of at least $(3/4)2^k - 2^{k-1} = (1/2)2^{k-1} = 2^{k-2}$ beyond $\overline{C}_{k-1}$'s outer boundary. It follows that the contribution to the cost of $T_{\text{opt}}(S)$ of these boxes is at least $(\overline{m}_k/3)2^{k-2} = 2^k\overline{m}_k/12$. Because all of these box motions are contained within C_k, there is no double counting of this cost between containers.

The generalization to the case of a bounded rectangular domain Ω is straightforward but tedious. The key difference is that, due to the bounded nature of Ω, the sizes of consecutive containers may grow only linearly, not quadratically, with the ℓ_1 radius of the container. (This happens, for example, if the domain is a long, thin strip.) Further, the size of the last container may even be smaller than its predecessor as one approaches the outer edges of the domain. However, the key is that, since the radius value grows exponentially, consecutive container sizes differ by a constant factor for all but a constant number of containers, and this is all that the above analysis requires.

Let s_k denote the number of complete segments for level k. Summing all the segments and all the levels of the hierarchy, one obtains

$$T_{\text{opt}}(S) \;\geq\; \sum_{k\geq 2} s_k 2^{k-2}\overline{m}_k.$$

Adding in a term $f_2(S)$ to account for the final (incomplete) segments, noting that $\overline{m}_0$ and $\overline{m}_1$ are both constants, and combining with the earlier bound on $W_{\text{lru}}(S)$,

results in the following, for a suitable constant c_3.

$$\begin{aligned} W_{\text{lru}}(S) &\leq \sum_{k\geq 0} s_k 2^k \overline{m}_k + f_2(S) = s_0\overline{m}_0 + s_1 2\overline{m}_1 + \sum_{k\geq 2} s_k 2^k \overline{m}_k + f_2(S) \\ &\leq c_3(s_0 + s_1) + 4T_{\text{opt}}(S) + f_2(S). \end{aligned}$$

The term $c_3(s_0 + s_1)$ is just a constant times the total number of access requests and is not dominant. It follows that there is a constant c_2 such that $W_{\text{lru}}(S) \leq c_2 T_{\text{opt}}(S) + f_2(S)$, which establishes inequality (2). Note that $f_2(S)$ does not grow with the length of the access sequence.

Finally, by combining inequalities (1) and (2), one obtains

$$\begin{aligned} T_{\text{lru}}(S) &\leq c_1 W_{\text{lru}}(S) + f_1(S) \leq c_1(c_2 T_{\text{opt}}(S) + f_2(S)) + f_1(S) \\ &\leq c_1 c_2 T_{\text{opt}}(S) + (c_1 f_2(S) + f_1(S)) \leq cT_{\text{opt}}(S) + f(S), \end{aligned}$$

for some constant $c \geq c_1 c_2 \geq 32$ and quantity $f(S)$ that does not grow with the length of the access sequence. For all sufficiently long access sequences, this final term will be negligible. This completes the proof. □

5.2.3 Online Algorithm for Sliding Motion

In order to accommodate the added constraints involved in sliding boxes around the space, the manner in which boxes are arranged throughout the domain is constrained. An obvious solution would be to arrange the boxes in rows connected by empty corridors, as in typical warehouses. However, this is not efficient asymptotically,

because it implies that the number of unoccupied squares in any region of space is at least a constant fraction of the available space. A more space-efficient approach is adopted herein by packing distant boxes more densely. While these distant boxes will require more cost to access, this cost can be amortized against the cost incurred by their distance from the access point.

To make this formal, a *layout scheme* is defined to be a subset of the integer grid $\mathbb{Z}^2$, which can be thought of as a subset of the unit squares. For each integer s, define $n(s)$ to be the number of squares of the layout that lie within an $s \times s$ square that is centered about the origin. Define the *asymptotic density* to be the limiting ratio of the fraction of squares in the layout lying within such origin-centered squares, that is, $\lim_{s\to\infty} n(s)/s^2$. For example, the layout that places boxes at every point of the grid has an optimal asymptotic density of 1, and a layout that places boxes only on the white squares of an infinite chessboard has an asymptotic density of $1/2$.

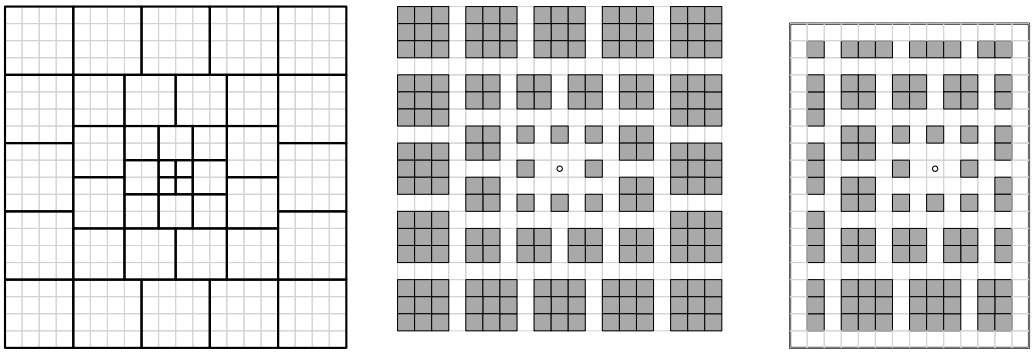
In this section, a layout that achieves the optimal asymptotic density of 1 is described, and it is shown how to convert the swapping-based Block-LRU$_A$ algorithm to the sliding context at the expense of an additional constant factor in cost.

5.2.3.1 The Nicomachus Layout

The presented layout scheme is inspired by a well-known visual proof of Nicomachus's Theorem [97], which is shown in Fig. 5.5(a).[1] The grid is partitioned

[1]Nicomachus's Theorem states that $\sum_{k=1}^{n} k^3 = \left(\sum_{k=1}^{n} k\right)^2$. If both sides of the equation are multiplied by 4, the layout of Fig. 5.5(a) provides a proof, where the left side arises by summing the number of blocks ring-by-ring (the kth ring has $4k$ blocks, each with k^2 squares) and the right side comes from the overall area (since the side length of the nth ring is $n(n+1) = 2\sum_{k=1}^{n} k$).

into expanding concentric *rings* of square regions, denoted $r_1, r_2, \ldots$. The innermost ring, r_1, consists of 4 unit squares. Ring r_2 consists of eight copies of a 2×2 square region surrounding r_1. In general, r_k consists of $4k$ copies of a $k \times k$ square region surrounding r_{k-1}.



(a) (b) (c)

Figure 5.5: (a) A geometric tiling based on Nicomachus's Theorem, (b) the associated layout scheme, and (c) restricted to a rectangular domain.

The layout for the warehouse problem presented here, called the *Nicomachus layout*, is constructed as follows. For each ring r_k of the aforementioned structure and for each $k \times k$ square region of this ring, the $(k-1) \times (k-1)$ unit squares are included in the upper left corner in the layout (shaded in Fig. 5.5(b).) Each of these is called a *block*. The upper-left cell of ring r_1 is designated as the access point. Finally, to accommodate a rectangular domain Ω, the layout is clipped to the boundary of the rectangle and the layout squares touching the domain's boundary are removed, thus creating corridors along the domain walls (see Fig. 5.5(c)). Observe that each block is surrounded by corridors that are one square wide. It is next shown that this layout achieves an optimal asymptotic density.

Lemma 5.2.1. *The Nicomachus layout achieves an asymptotic density of* 1.

Proof. It suffices to show that the *asymptotic wastage*, that is, the asymptotic density of the complement of the Nicomachus layout, is equal to zero. To see this, consider the first $\ell \geq 1$ rings of the layout. Each ring r_k, $1 \leq k \leq \ell$, consists of $4k$ blocks, each of size $(k-1) \times (k-1)$. The unused space per block is $k^2 - (k-1)^2 = 2k - 1$. Thus, the total wasted space for ring k is $4k(2k-1)$. Summing over all rings, the total wastage is $\sum_{k=1}^{\ell} 4k(2k-1) = 8\ell^3/3 + O(\ell^2)$. The first ℓ rings fill an origin-centered square of side length $\ell(\ell+1)$, which yields a total area of at least ℓ^4. Therefore, ignoring lower-order terms, the wastage for these rings is at most $(8\ell^3/3)/\ell^4 = 8/3\ell$. Clearly, this tends to zero in the limit. (Expressed as a function of n, the asymptotic density is the limit of $1 - 8/(3n^{1/4})$.) □

5.2.3.2 Accessing a Box

In order to access a box in the warehouse a robot must first travel to the block in which that box resides, retrieve it from the block, and then return it to the access point. The *depth* d of a box is defined to be the minimum number of boxes between it and the boundary of the block that contains it. So, a box on the perimeter of a block has depth $d = 0$, while one at the center of a block in ring r_i has depth $d = \lfloor \frac{i-2}{2} \rfloor$. (When the domain Ω is bounded, this is an upper bound, since peripheral blocks may be clipped.)

In the Nicomachus layout, the cost of reaching a box in the arrangement and retrieving it from a block are both a function of the ring in which it resides. Let $M(r_i)$ denote the maximum cost of moving the robot from the access point to any

cell adjacent to a block of ring r_i, and let $C(r_i)$ be the maximum cost of retrieving a box from a block in ring r_i. First, consider the travel cost of reaching a cell on the perimeter of a block of boxes.

Lemma 5.2.2. *Travelling from the access point to any cell adjacent to a block in ring r_i requires at most $i^2 + i$ steps.*

Proof. To reach a box on the perimeter of a block in ring r_i from the access point, a robot must traverse each ring $k \leq i$ by circumnavigating one of its blocks. It is easy to see that a robot can move between any two cells adjacent to a $(k-1) \times (k-1)$ block of ring r_k in $2k$ steps, from which it is concluded that the total travel time is

$$M(r_i) \;\leq\; \sum_{k=1}^{i} 2k \;=\; i(i+1) \;\leq\; i^2 + i.$$

□

An equivalent distance is traveled to return the requested box to the access point.

Next, define a primitive Replace(d) that allows for the swapping of a box b_i placed in the aisle adjacent to a block B with a box $b_j \in B$ at depth d. For now this primitive will be used to establish an upper bound on the cost of accessing a box, while the need for actually swapping boxes will not become apparent until later. Conceptually, the Replace primitive must unbury the target box by moving the d boxes in the way. It does so by moving them each $d+1$ spaces away, retrieving the target box, and then replacing them for a total cost $O(d^2)$. A more careful analysis

yields the following.

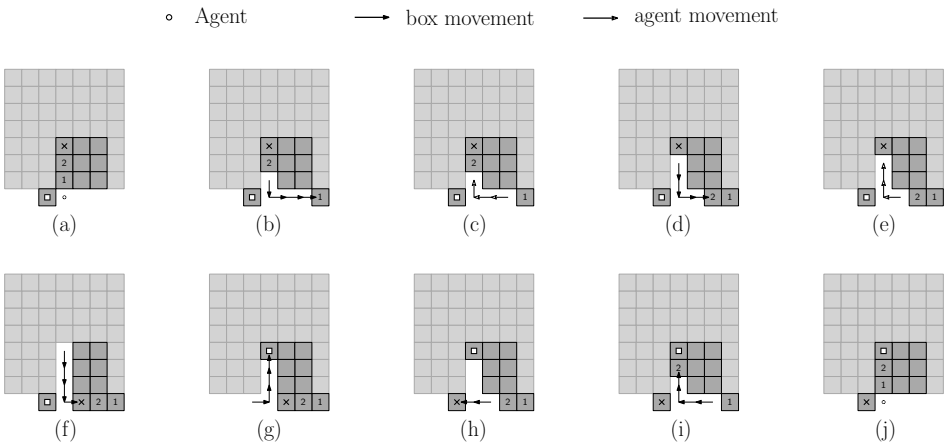


Figure 5.6: Swapping a pair of boxes, where the original box is at depth $d = 2$ within a 7×7 block in ring r_8.

Lemma 5.2.3. *The cost of* Replace(d) *is at most* $4d^2 + 8d + 6$*, where* d *is the depth of box* b_j*.*

Proof. First, number the boxes inward from box b_j's nearest boundary from 1 to d. Assume that the robot begins adjacent to box 1 and that box b_i is adjacent to the robot. Next, iteratively move each of the $d+1$ boxes (the d labeled boxes plus b_j) to a location that is $d+2$ units away along the side of the block (see Fig. 5.6). Accounting for the time to reach each box, pick it up, move it, put it down, and return to a position adjacent to the next box to be moved, each iteration has a total cost of $2d+3$, except the last, which does not require moving to the next box and so only costs $d+2$. In total, moving these boxes costs $d(2d+3)+(d+2) = 2d^2+4d+2$. Next, the process is reversed at the same cost, replacing box b_j with box b_i and restoring boxes 1 through d to their original positions. This process is briefly interrupted to move box b_j out of the way, adding a cost of 2 (Fig. 5.6(h)). Thus, in total, swapping a new box with an interior box comes at a cost of $2(2d^2 + 4d + 2) + 2 = 4d^2 + 8d + 6$. □

The depth of a box is bounded by the radius of the block in which it resides. Specifically, a box in ring r_i has a depth $d \leq \frac{i-2}{2}$ and so, along with Lemma 5.2.3, there is the following corollary:

Corollary 5.2.4. *Retrieving a box from a block in ring r_i has a cost of $C(r_i) \leq i^2+2$.*

Combining this corollary and Lemma 5.2.2, the total cost to move to a box in ring r_i, retrieve it, and return to the access point is at most

$$(i^2 + i) + (i^2 + 2) + (i^2 + i) \;=\; 3i^2 + 2i + 2 \tag{5.1}$$

Next, consider retrieval cost as a function of distance from the access point.

Lemma 5.2.5. *If a box is at ℓ_1 distance δ from the access point then it lies in a ring r_i, such that $i \leq \sqrt{3\delta}$.*

Proof. To reach the highest ring level possible at a distance δ, travel orthogonally in a straight line, traversing each ring's width in turn. As ring r_i has width i, the farthest ring that can be reached is the first ring r_i such that

$$\delta \;\leq\; \sum_{j=0}^{i} j \;=\; \frac{i^2 + i}{2} \tag{5.2}$$

Solving for i yields $i \geq \sqrt{2\delta + \frac{1}{4}} - \frac{1}{2}$.

It is easily seen that for all $\delta \geq 1$, $\sqrt{3\delta} \geq \sqrt{2\delta + \frac{1}{4}} - \frac{1}{2}$, thus $i = \sqrt{3\delta}$ suffices as an upper bound for the greatest ring index at a distance no more than δ. □

By combining Eq. (5.1) and Lemma 5.2.5, the following is obtained.

Lemma 5.2.6. *In the Nicomachus layout, retrieving a box at ℓ_1 distance δ from the access point is $O(\delta)$.*

Proof. Eq. (5.1) shows that retrieving a box in ring r_i has a maximum total cost of $3i^2 + 2i + 2$ and Lemma 5.2.5 shows that a box at distance δ will be in some ring r_i, where $i \leq \sqrt{3\delta}$. So, retrieving a box at distance δ incurs at most a cost of $3(\sqrt{3\delta})^2 + 2\sqrt{3\delta} + 2 = 9\delta + 2\sqrt{3\delta} + 2$, which is $O(\delta)$. □

From this it is concluded that trading the positions of two boxes can be done at a cost proportional to the sum of their ℓ_1 distances from the access point. A simple, naïve algorithm could use the access point as an intermediary, accessing both boxes at cost $O(\delta)$, and returning them to their opposing rather than original positions. Thus, the following:

Corollary 5.2.7. *If two boxes b_i and b_j are at ℓ_1 distances δ_i and δ_j from the access point, respectively, then the cost of swapping them is no more than $c(\delta_i + \delta_j)$, for some constant c.*

Given this corollary, it can now be shown that Block-LRU$_A$ is competitive in the sliding model. From the proof of Theorem 5.2.1 and the structure of Block-LRU$_A$, it suffices to bound the cost of evictions from each of the containers. For any $k \geq 0$, consider an eviction from container C_k to C_{k+1}. The contribution of this eviction to $W_{\text{lru}}(S)$ is 2^k. By Corollary 5.2.7, the cost of sliding one to the other is at most $c(2^{k-1} + 2^k) < 2c2^k$, implying that the sliding cost is within a constant factor of the

eviction cost (roughly 4). From the proof of Theorem 5.2.1 the eviction cost can be used as a proxy for its actual cost, and therefore the sliding cost is at most a constant factor more than the actual cost of Block-LRU$_A$ in the case of swapping motion. This implies that the cost of Block-LRU$_A$ in the sliding motion model is competitive with the optimum solution in the swapping motion model. The actual cost of the optimum algorithm in the sliding model cannot be lower than the actual cost of the optimum algorithm in the swapping model. With a roughly factor-4 cost ratio between the sliding and swapping models, the overall ratio is roughly 128. While this competitive ratio may be rather high, the analysis thus far has assumed worst case scenarios across multiple factors and the focus has been to prove the general competitiveness rather than finding the best competitive ratio. An empirical experiment would likely show that the average case scenario has a much more favorable competitive ratio. Regardless, as a consequence of the above discussion:

Theorem 5.2.2. *For any instance of the attic problem and any sufficiently long access sequence S, the cost of Block-LRU$_A(S)$ is within a constant factor of the cost of an optimal solution, assuming sliding motion.*

5.3 Online Solution to the Warehouse Problem

This section presents an online algorithm for the warehouse problem. As before, the algorithm for swapping motion is presented and then generalized to sliding motion. Recall that the warehouse problem differs from the attic problem in that there are multiple access points, all of which lie on the bottom side of the

rectangular domain Ω, which is assumed to lie on the x-axis. The presented algorithm, called Block-LRU$_W$, will be similar in spirit to online algorithms for hierarchical memory systems, but the combination of spatial locations and multiple access points adds considerable complexity. As with the attic problem, it will simplify matters to describe the algorithm first in an infinite context, where boxes may be placed anywhere above the x-axis, and then adjust the solution to the case of a rectangular domain. The approach will be to define containers based on a quadtree-like structure above the x-axis, and to evict boxes up the quadtree from child to parent. Each quadtree cell will be treated as if it were a cache in a memory hierarchy, with the least-recently used box evicted whenever more space is needed.

5.3.1 Quadtree Model

As mentioned above, the online solution to the warehouse problem presented here employs a quadtree subdivision over the positive-y halfspace. The leaves of the quadtree, or *level* 0, consist of the unit squares whose lower left corners are the grid points on the x-axis, that is, $(x, 0)$ for $x \in \mathbb{Z}$. Level 1 consists of the 2×2 squares lying immediately above whose lower left corners are located at $(2x, 1)$ for $x \in \mathbb{Z}$. In general, for $k \geq 0$, level-k consists of the $2^k \times 2^k$ squares whose lower left corners lie on $(2^k x, 2^k - 1)$, for $x \in \mathbb{Z}$. Each level-k node u has a parent $p(u)$ of twice the side length lying immediately above on level $k+1$ (see Fig. 5.7(a)), and two children each of half the side length lying immediately below on level $k-1$. The set of unit squares associated with each node of the quadtree is called its *cell*. This structure covers the

infinite grid lying above the x-axis. Given a rectangular domain Ω whose lower side lies along the x-axis, clip the above structure to this rectangle (see Fig. 5.7(b)).

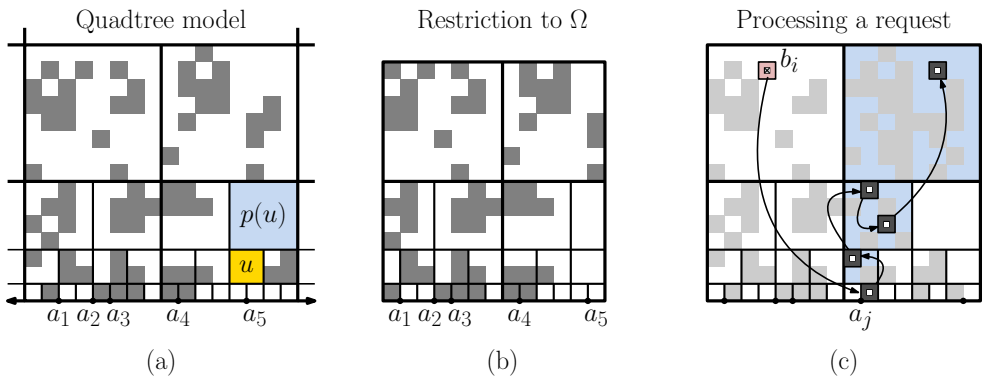


Figure 5.7: Quadtree layout.

To simplify the analysis of this solution, a variant of the warehouse problem with an alternate cost function based on this quadtree structure is first defined, which is called the *quadtree model*. Of course, an optimal solution does not need to follow this model, and later, the cost of the standard solution will be related to this variant. The processing of requests in this model differs from the standard model (described in Section 5.1.1) in that, after moving the box to the desired access point, the reorganization chain is allowed to move a box within its current quadtree cell, or it may move the box to the quadtree cell of an ancestor, but no other movements are allowed (see Fig. 5.7(c)).

More formally, consider a request for a box b to access point a. Let $Q_0(a)$ denote the quadtree cell containing a, and let $Q_1(a), Q_2(a), \ldots$ denote the successive quadtree ancestor cells of $Q_0(a)$. If a is unoccupied, the box is simply moved there. Otherwise, in order to make space for b_i, a chain of swaps is performed along some locations $p_0, p_1, \ldots, p_k$ such that $p_0 = a$, p_k is unoccupied (possibly the former

location of b), and if $p_i \in Q_j(a)$, then p_{i+1} is the same cell or an ancestor, that is, $p_{i+1} \in Q_{j'}(a)$ for $j' \geq j$. As described in Section 5.1.1, swaps are performed (in reverse order) along the resulting chain. Each swap that moves a box out of its current quadtree cell is called an *eviction*.

Costs are defined as follows in this model. A box may be moved within its quadtree cell free of charge, but when it is moved to an ancestor cell, it is charged 2^k, where k is the level of the quadtree cell into which the box is moved. (The analogy with hierarchical memory systems should be evident, where one can think of each quadtree cell as a cache, and eviction to an ancestor is analogous to moving a page to a larger cache in slower memory.)

5.3.2 Online Algorithm for Swapping Motion

This section presents a model for the warehouse problem, which is called Block-LRU$_W$. Consider a request (b, a) to bring box b to access point a. If this access point is unoccupied, the box is simply moved there. Otherwise, in order to make space for b, a sequence of evictions is performed from $Q_0(a)$, $Q_1(a)$, and so on until one encounters the first quadtree ancestor $Q_k(a)$ that has at least one unoccupied location (possibly b's location at the time of the request). More formally, let p_b denote b's location, let $p_0 = a$ denote the access point, and let $p_1, \ldots, p_{k-1}$ denote the locations of the least-recently used boxes of quadtree cells $Q_0(a)$ through $Q_{k-1}(a)$, respectively. Finally, let $p_k \in Q_k(a)$ denote the final unoccupied location (or former location of b). As described in Section 5.1.1, swaps are performed (in reverse order)

along the chain $\langle p_b, p_0, \ldots, p_k \rangle$. The main result of this section is showing that this algorithm is competitive.

Theorem 5.3.1. *For any instance of the warehouse problem and any sufficiently long access sequence S, the cost of Block-LRU$_W(S)$ is within a constant factor of the cost of an optimal solution, assuming swapping motion.*

Observe that Block-LRU$_W$ satisfies the requirements in the quadtree model. For the sake of the above theorem, its cost is computed in the standard manner, as the sum of the ℓ_1 distances of all swaps performed. Later, it will be shown that this is proportional to its cost in the quadtree model.

The remainder of this section is devoted to proving this theorem. First, consider how the behavior of a general solution to the warehouse problem can be simulated in the quadtree model. Rather than focusing on individual access requests, this will be done on a box-by-box basis. Consider input sequence S and any box b. Let S' denote a contiguous segment of S, which starts and ends at two consecutive access requests involving b. Denote these access points by a_1 and a_2, respectively. (For the segment prior to b's first access, set a_1 to the closest access point to b's initial location, and for the segment following b's last access, a_2 can be set arbitrarily to any access point.)

When the standard solution completes the processing of the first access request, b will reside at a_1. As a result of subsequent access requests in S', b may be moved to new locations in the domain as a result of swap operations. Let $\langle p_0, \ldots, p_k \rangle$ denote the sequence of locations through which b moves during S', so that $p_0 = a_1$, and p_k is

the location of b just prior to the upcoming access request at a_2. Since this is in the standard model, the points of this sequence are arbitrary. To perform the simulation, a function π is defined that maps the location of b at any time to the cell of some quadtree ancestor of a_1 in a manner such that, under this function, b will move in accordance with the quadtree model. This mapping is presented in the next section.

5.3.2.1 Container Structure for the Warehouse Problem

Before giving the details of the aforementioned mapping, it is helpful to start with an intuitive explanation. For each access point a let $Q_k(a)$ denote the quadtree cell associated with a's ancestor at level k. Define a collection of nested regions of exponentially increasing sizes, called *containers*, surrounding a and denoted $C_0(a) \subset C_1(a) \subset \cdots$ (see Fig. 5.8(a)). (Note that, unlike the containers of Section 5.2.1, which were pairwise disjoint, here each container includes all the squares of its predecessors.)

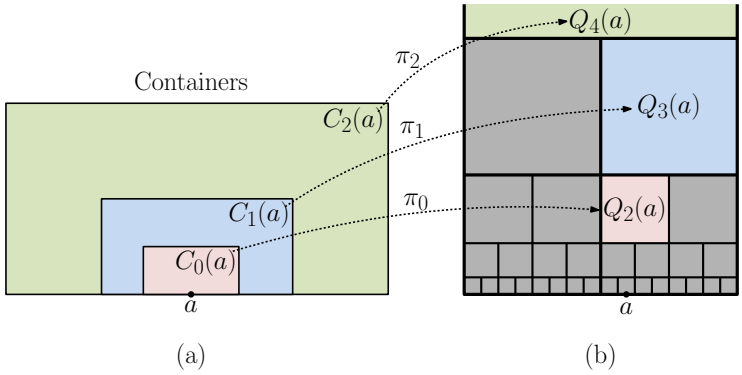


Figure 5.8: Intuitive structure of containers for the warehouse quadtree model.

For each container $C_k(a)$, define a 1–1 function π_k that maps each point in $C_k(A)$ to a point within the cell of some quadtree ancestor of a. (For example,

in Fig. 5.8(a), π_k maps boxes from $C_k(a)$ to $Q_{k+2}(a)$.) In order to simulate the movement of a box that has been accessed most recently by a, its movement will be tracked through these containers. On first entering a container $C_k(a)$ at some point p, the box is mapped to the associated point $\pi_k(p)$ in the quadtree cell. When the box moves to a new point p' within the same container, the box is moved to $\pi_k(p')$. Observe that because the containers are nested, even if the box moves into a location in a smaller container, it will still be considered as lying within C_k and so will remain in the same quadtree cell in the simulation. Recall that in the quadtree model, movements within the same quadtree cell are free of charge, and hence there is no need to account for movements within a given container. Whenever the box is first moved into a new, larger container $C_{k'}$, it will be charged the eviction cost of $2^{k''}$, where $Q_{k''}(a)$ is the associated quadtree cell.

The containers and the associated functions will now be defined more formally. One complication that arises is that the functions π_k associated with two nearby access points may map locations to the same quadtree cell. When this happens, it must be guaranteed that two distinct locations in their containers are not mapped to the same location in this quadtree cell. To handle this, the container structure is carefully designed so that access points that map to the same quadtree cell will share the same container and the same mapping function.

To make this precise, consider any access point a and any quadtree ancestor of a at level k. The function π_k for a will map points from a's container $C_k(a)$ to $Q_{k+2}(a)$. This implies that the four grandchildren of $Q_{k+2}(a)$ at level k will do the same. So, all of them will be given a common container and a common function. (In

Fig. 5.9(a), the container $C_2(a)$ is shared by four 4×4 quadtree cells drawn in heavy black lines.) The associated container is defined as follows. First, imagine a square grid of side length 2^k covering the plane that is aligned with the quadtree cells. The container consists of the 16 grid cells that are ℓ_1 neighbors of the four grandchildren. (In Fig. 5.9(a), this container $C_2(a)$ is shaded in dark gray and includes the squares of $C_0(a)$ and $C_1(a)$. Note that the lowest tier of these grid squares falls one unit below the x-axis, but these nonexistent squares are simply ignored in the mapping.) The number of squares is at most $16 \cdot 2^k = 2^{k+2}$, and so there is sufficient space to map the squares of the container into $Q^{k+2}(a)$ (see Fig. 5.9(b)). π_k is defined for this container to be any such function. (It is not required that this function preserve distances, because according to the quadtree model, movements within a quadtree cell are free.)

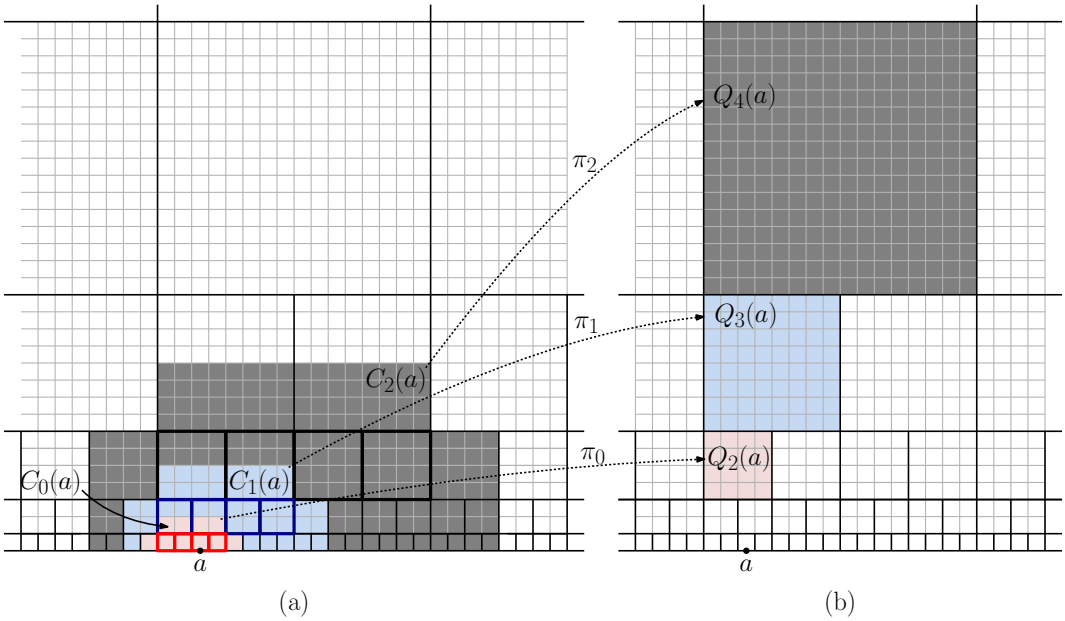


Figure 5.9: Actual structure of containers for the warehouse quadtree model.

5.3.2.2 Proving Competitiveness

This section presents a proof of Theorem 5.3.1.

Given an access sequence S, define $T_{\text{opt}}(S)$, $T_{\text{lru}}(S)$ to be the (standard) costs for Opt and Block-LRU$_W$, respectively. Define $W_{\text{lru}}(S)$ to be the cost of Block-LRU$_W$ in the quadtree cost model, and define $W_{\text{opt}}(S)$ to be the cost of the quadtree-simulated version of Opt in the quadtree cost model.

The analysis follows a similar structure to the one given in Theorem 5.2.1, and so this section will focus on just the major differences. The analysis is based on three inequalities, where c_1, c_2, and c_3 are constants and $f_2(S)$ and $f_3(S)$ are quantities that do not grow with the length of the access sequence:

$$(1) T_{\text{lru}}(S) \leq c_1 W_{\text{lru}}(S) \quad (2) W_{\text{lru}}(S) \leq c_2 W_{\text{opt}}(S) + f_2(S) \quad (3) W_{\text{opt}}(S) \leq c_3 T_{\text{opt}}(S) + f_3(S)$$

- $T_{\text{lru}}(S) \leq c_1 W_{\text{lru}}(S)$: Block-LRU$_W$ is running in the quadtree model, but it uses the standard (ℓ_1) costs, not the eviction costs. Also, it evicts from child to parent, never skipping ancestors. When moving a box from quadtree cell Q_{k-1} to Q_k the actual cost is at most the worst-case ℓ_1 distance between these cells, which is at most $2 \cdot 2^k = 2^{k+1}$, and the quadtree model assesses a charge of 2^k. Thus, setting $c_1 = 2$ yields the desired bound.

- $W_{\text{lru}}(S) \leq c_2 W_{\text{opt}}(S) + f_2(S)$: Let $m_k = 2^{2k}$ denote the number of boxes in a quadtree cell Q_k at level k. Let $\overline{m}_k$ be the sum of m_k for a quadtree cell and all its descendants (which is roughly $2m_k$). Focus on a single quadtree

cell at level k, call it Q_k. Consider the two child cells at level $k-1$, Q'_{k-1} and Q''_{k-1}. Let A' and A'' denote the subsets of access points descended from these two quadtree nodes, respectively. Now, break up the access sequence into contiguous segments, such that Q_k witnesses $\overline{m}_k$ evictions in the running of Block-LRU$_W$. Consider a single segment S'. Observe that, with respect to access points $A' \cup A''$, Block-LRU$_W$ is effectively running an LRU algorithm on the union of Q_k and the cells of all its children. (To see why, observe that the least-recently used boxes of each descendent are evicted to their parents and eventually up to to Q_k, and the least-recently used box within Q_k is evicted.)

Over segment S', at least $\overline{m}_k$ distinct box accesses have been processed by the access points $A' \cup A''$ combined. Now, consider how $W_{\text{opt}}(S)$ handles the same requests, but from the perspective of Q'_{k-1} and Q''_{k-1}. These two together (and their descendant cells) have a total capacity of $\overline{m}_{k-1} + \overline{m}_{k-1} \approx \overline{m}_k/2$. Thus, the remaining roughly $\overline{m}_k/2$ boxes must be evicted from these children by Opt. They may be evicted up one level to Q_k or up multiple levels. For the sake of simplicity, consider the case where they are evicted up just one level to Q_k. (The other case involves splitting the charge among the nodes along the path according to a geometric series.) Each evicted box is assessed a charge of 2^k, for a total of roughly $2^k\overline{m}_k/2 = 2^{k-1}\overline{m}_k$. Therefore, the total charge assessed to $W_{\text{opt}}(S)$ during this segment is at least $2^{k-1}\overline{m}_k$, while the total charge assessed to Q_k in $W_{\text{lru}}(S)$ is $2^{k+1}\overline{m}_k$. Summing over all the levels (and letting $f_2(S)$ account for the charges in the partial segment at the end of S)

yields $W_{\text{lru}}(S) \leq c_2 W_{\text{opt}}(S) + f_2(S)$, where c_2 is roughly 4.

- $W_{\text{opt}}(S) \leq c_3 T_{\text{opt}}(S) + f_3(S)$: The analysis focuses on the activity involving a single box b between two consecutive accesses to a and a', say. (The additional $f_3(S)$ term handles the cost prior to the initial request for b and after the final request.) Observe that $W_{\text{opt}}(S)$ does not charge for movements within a quadtree cell, and (since this is the quadtree model) it never demotes a box to a lower level of the quadtree. It charges an eviction cost of 2^k whenever the box enters a quadtree cell at level k. This event corresponds to an event in standard Opt when this box enters $C_k(a) \setminus C_{k-1}(a)$ for the first time. Let k^* denote the highest container index into which Opt moves this box (formally, the highest k such that the box enters $C_k(a) \setminus C_{k-1}(a)$). Since this box might be evicted into all the containers from level 1 up to k^*, this box contributes at most $\sum_{k=1}^{k^*} 2^k \leq 2^{k^*+1}$ to $W_{\text{opt}}(S)$. On the other hand, Opt has to move this box from the access point to some point in $C_{k^*}(a) \setminus C_{k^*-1}(a)$. It is easy to see that this involves a distance of at least $2^{k^*} + 1$. It follows that this box contributes more than 2^{k^*} to $T_{\text{opt}}(S)$ and at most 2^{k^*+1} to $W_{\text{opt}}(S)$. Therefore, setting $c_3 = 2$ yields the desired result.

Together, the three inequalities imply that

$$\begin{aligned} T_{\text{lru}}(S) \;&\leq\; c_1 W_{\text{lru}}(S) \;\leq\; c_1(c_2 W_{\text{opt}}(S) + f_2(S)) \\ &\leq\; c_1(c_2(c_3 T_{\text{opt}}(S) + f_3(S)) + f_2(S)) \;\leq\; c T_{\text{opt}}(S) + f(S), \end{aligned}$$

where $c = c_1c_2c_3 = 16$ and $f(S) = c_1c_2f_3(S) + c_1f_2(S)$. This completes the proof of Theorem 5.3.1.

5.3.3 Online Algorithm for Sliding Motion

This section shows that the competitiveness of Block-LRU$_W$ in the case of swapping motion can be used to prove that the sliding version of the same algorithm is competitive. As in the attic problem, the approach will be to describe a layout of boxes that is amenable to efficient sliding motion.

This analysis makes use of a Nicomachus-like box layout. Rather than rings centered about the access point, rings are flattened into layers stacked above the x-axis. As before, the arrangement begins with a layer of 1×1 cell regions. Above this is a row of 2×2 regions, then 3×3, and so on, with each $i \times i$ region containing a block of $(i-1) \times (i-1)$ boxes (see Fig. 5.10). This is called the *flattened Nicomachus layout*.

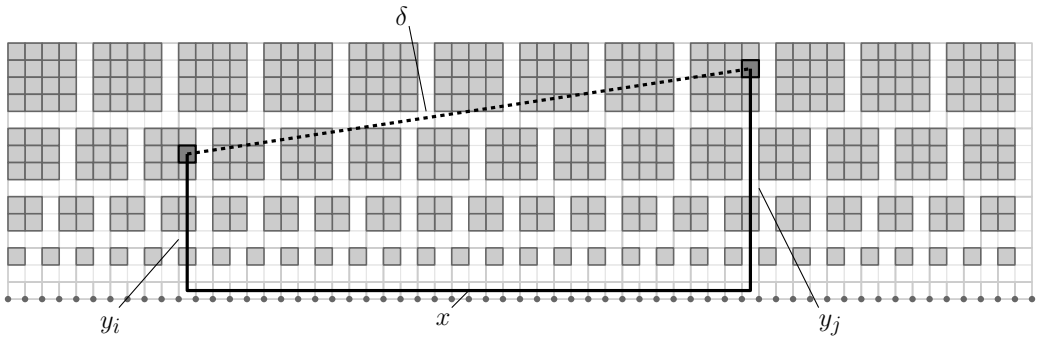


Figure 5.10: A flattened version of the Nicomachus layout for the warehouse problem, with a conceptual example of swapping two boxes. Pathfinding is ignored in this illustration, but accounted for in the supporting lemma.

Once again, one can make use of a simple naïve algorithm that can efficiently trade the positions of two boxes in the sliding model. More formally, the following is

proven:

Lemma 5.3.1. *If two boxes b_i and b_j are at ℓ_1 distances δ from each other and at vertical distances y_i and y_j from the x-axis, respectively, then the cost of swapping them in the flattened Nicomachus layout is no more than $c(\delta + y_i + y_j)$, for some constant c.*

Proof. A naïve algorithm can swap the two boxes b_i and b_j by: (1) bringing them to the x-axis, (2) swapping their positions along the x-axis, and (3) returning them to their new vertical positions. Notice that the cost of retrieving/replacing a box and bringing it to the x-axis is equivalent to the retrieval cost of a box positioned directly above the access point in the Attic Problem with Sliding Motion. As per Lemma 5.2.6, this access cost in both contexts is $O(y)$, where y is the distance to the x-axis or singular access point, respectively. Given this, both steps (1) and (3) of the algorithm occur at a constant factor of $(y_i + y_j)$. Clearly the horizontal distance traveled along the x-axis, x is less than or equal to δ, therefore, the total cost of swapping the two boxes must be no greater than $c(\delta + y_i + y_j)$, for some constant c. □

This lemma can be used to relate the cost of trading two elements in the swapping and sliding models. The following summarizes the main result.

Theorem 5.3.2. *For any instance of the warehouse problem and any sufficiently long access sequence S, the cost of Block-LRU$_W(S)$ is within a constant factor of the cost of an optimal solution, assuming sliding motion.*

Proof. From Theorem 5.3.1 and the structure of Block-LRU$_W$, it suffices to bound the cost of evictions from one quadtree node to its parent. Assuming that the node is at quadtree level $k-1$, and its parent is at level k, this swap incurs a cost of 2^k in the quadtree model. Letting y_1 and y_2 denote the vertical distances of these locations from the x-axis, then $y_1 \leq 2^k$ and $y_2 \leq 2^{k+1}$. Also, they are separated from each other by an ℓ_1 distance of $\delta \leq 2^{k+2}$. By Lemma 5.3.1, the cost of sliding one to the other is at most $c(\delta + y_i + y_j) \leq c(2^{k+2} + 2^k + 2^{k+1}) = 7c2^k$, implying that sliding cost is within a constant factor of the quadtree cost. From the proof of Theorem 5.3.1 and the structure of Block-LRU$_W$, the quadtree cost of Block-LRU$_W$ can be used as a proxy for its actual cost, and therefore the sliding cost is at most a constant factor more than the actual cost of Block-LRU$_W$ assuming swapping motion. This implies that the cost of Block-LRU$_W$ in the sliding motion model is competitive with the optimum solution in the swapping motion model. The actual cost of the optimum algorithm in the sliding model cannot be lower than the actual cost of the optimum algorithm in the swapping model. With a roughly factor-7 cost ratio between the sliding and swapping models, the overall ratio is roughly 112. As before, this is based on many worst-case assumptions and can likely be improved upon. □

Chapter 6: Conclusion

The goal of this dissertation has been to identify optimization problems that are simple enough to analyze formally, yet realistic enough to contribute to the eventual design of systems rooted in shared, physical spaces.

While other works tackle the low-level engineering challenges involved in designing such systems or the high-level decision problems one might face, the focus herein has been somewhere in between, drawing heavily from the tools and techniques of computational geometry and algorithmic analysis.

By applying these techniques to shared-space autonomous systems, their complexity has been explored, assessing the difficulty of the tasks to be solved, gaining insights into methods for finding efficient solutions, and, ultimately, demonstrating such solutions.

In particular, three problems have been presented in their respective chapters: automated vehicles and unregulated traffic crossings, a smart network for smooth city-wide traffic flow, and an online organizational scheme for automated warehouses.

A summary of the results follows, along with some discussion on future avenues of exploration.

6.1 On the Complexity of an Unregulated Traffic Crossing

Chapter 3 introduced the Traffic Crossing Problem (TCP). After formally defining it in Section 3.2, three results are given: this problem is proven NP-complete

(Section 3.3), a constrained version is solved in $O(n \log n)$ time (Section 3.5), and, in a discrete setting, an asymptotically optimal solution is provided that limits maximum delay of any vehicle (Section 3.6).

The definition of the TCP in this work is, of course, an abstraction of a true, real-world traffic crossing. While the formulations used may be less realistic than a more complex one, they avoid some of the more cumbersome elements of one that attempts to capture every conceivable detail. This admits a much clearer view of the sources of computational complexity while still capturing the most salient elements of a traffic crossing.

However, to be more generally applicable, the model will need to evolve to encompass many types of traffic conditions/situations. So, for example, the movement model for vehicles in the system is fairly restrictive, as it only allows for linear, monotonic motion within a global speed interval. Allowing for nonlinear motion, non-monotonicity, or per vehicle/lane speed limits may have a dramatic effect on the behavior of traffic in the system or the efficacy of current solution methods.

Additionally, in the model, streets are laid out on a unit grid. Not all real-world street networks are laid out in such a convenient manner and even those that are are likely to contain slight angular and distance differences that may be enough to disturb the provided algorithms. The model would need to handle curving or turning roads, roads that lie in arbitrary locations and orientations on the plane, over/under passes, and so forth, to be more generally applicable and robust.

Other assumptions have been made with regard to the general traffic patterns,

as well. For example, in the unit-delay solution the assumption is made that no caravan of vehicles is long enough to overflow into an adjacent intersection. This so-called short-caravan assumption was made to avoid a lengthy digression into how to handle cyclic dependencies in city blocks, but is a relatively common occurrence in urban traffic settings.

So too is imperfect knowledge. Having *a priori* knowledge of all vehicles provides a sizable advantage that is atypical of real-world systems. Instead, a solution to an online version of the problem that could, for example, take preventative measures rather than suffering the consequences of reactive strategies would be a very powerful tool.

Finding solutions to the TCP free of these assumptions and model restrictions would go a long way toward bridging the gap between the theoretical results presented here and their real-world application.

6.2 Modular Circulation and Applications to Traffic Management

Chapter 4 built upon the well-known circulation problem, by introducing a variant referred to as the λ-*CMD problem*. Given a directed graph G and vertex demands $d \in \mathbb{Z} \pmod{\lambda}$, the λ-CMD problem is that of computing a minimum-cost flow that satisfies the modular demands of every vertex. A modular demand d is satisfied by an inbound flow of d or an outbound flow of $\lambda - d$.

This chapter showed that 2-CMD can be solved exactly in polynomial time as, thanks to the nature of modular demands, edge direction is irrelevant and finding

a minimum-cost satisfying flow is equivalent to finding a minimum-cost perfect matching between vertices (see Section 4.4 for details).

Additionally, by way of a reduction from positive 1-in-3-SAT, it was shown that λ-CMD is NP-hard for $\lambda \geq 3$. This reduction is first illustrated with an example of a reduction to 3-CMD (see Section 4.5) and later generalized for larger values of λ through a simple extension of the gadgets used in the $\lambda = 3$ example (see Section 4.5.4).

The final result is a polynomial time $4(\lambda - 1)$-approximation to λ-CMD. This approximation employs a Kruskal-like algorithm to build balanced neighborhoods that can satisfy their demands without costly external flows. To do so, however, requires that the edge directions are ignored, which induces a λ factor penalty to the approximation (see Section 4.6).

Future work could be done to do away with this limitation and decrease the approximation factor, perhaps removing the λ factor entirely. Additionally, the reduction used in the hardness proof of λ-CMD is for generalized directed graphs, meaning that these graphs are not necessarily planar. While studies of actual road networks have shown that crossings are not uncommon [98], in many urban road networks the fraction of crossings to total intersections is very small [99]. It would therefore be of interest to know whether the λ-CMD problem is hard even for planar graphs. Although there are planar variants of satisfiability, which make use of cross-over gadgets to eliminate non-planar elements (see, e.g., [100]), such a gadget was not discovered in the λ-CMD context in the course of the work described in this dissertation. An open problem, therefore, is whether planar λ-CMD is NP-hard as

well.

Finally, as before, this work could be extended by generalizing it to more robust models of traffic flow, including but not limited to multi-lane roads, bidirectional streets, mixed blocks with differing numbers of sides (particularly a mix of even- and odd-numbered sides, as this disrupts the periodic patterns), and vehicles that exhibit more complex behaviors such as turning at intersections, changing lanes, etc.

6.3 Online Algorithms for Warehouse Management

Chapter 5 presented a model for an automated warehouse management system containing a set of standardized portable storage units or boxes, a robot that moves these boxes around the warehouse in one of two ways (swapping or sliding), and a set of access points where requested boxes must be delivered. It then presented online algorithms for two natural instances of the warehouse problem, one involving a single access point within a rectangular domain and the other involving a sequence of access points along the bottom side of a rectangular domain. The algorithms presented in this chapter were proven to be competitive with respect to an optimal (offline) algorithm with full knowledge of the access sequence. However, the competitive ratios are relatively high, and are likely far from tight, but tightening these bounds will involve either significantly more complex algorithms or better lower bounds.

Some interesting open problems are left for future work. Recall that the model presented assumes that access requests are processed sequentially. This simplifying assumption removes the extremely difficult issue of motion coordination, which

arises when multiple robots are present [31, 33, 77]. Clearly, any realistic solution should consider an environment with multiple robots where requests are processed concurrently. Because the layout of boxes in the domain is controlled, it may be possible to insert additional *slack space* into the layout to facilitate efficient motion coordination. Another interesting question in this vein is how to handle the insertion/deletion of boxes from the collection. Perhaps memory management schemes such as [101], which efficiently handle the reallocation of 2D memory, could be further leveraged.

Also, how does competitiveness change, if at all, when the model becomes less uniform? In the currently presented model, all actions taken by the robot are of unit cost, regardless of factors like whether or not the robot is laden or what sort of path a robot takes to retrieve a box. Çelik and Süral [102], for example, show that the number of turns a robot makes in a parallel-aisle warehouse can have a significant impact on retrieval efficiency. Fekete and Hoffmann [82] look at the online problem of packing differently sized squares into a dynamically-sized square container. Applying this work to a warehouse which does not use standardized containers would be a natural continuation of the work presented here. Further generalizing the presented model to account for differing action costs and box dimensions would increase its real-world applicability and may lead to some interesting insights.

Bibliography

[1] G. B. Dantzig and J. H. Ramser, "The truck dispatching problem," *Management Science*, vol. 6, no. 1, pp. 80–91, Oct. 1959.

[2] G. Clarke and J. W. Wright, "Scheduling of vehicles from a central depot to a number of delivery points," *Oper. Res.*, vol. 12, no. 4, pp. 568–581, Aug. 1964.

[3] M. M. Solomon, "Algorithms for the vehicle routing and scheduling problems with time window constraints," *Oper. Res.*, vol. 35, no. 2, pp. 254–265, Apr. 1987.

[4] J. Yu and S. M. LaValle, "Multi-agent path planning and network flow," in *Algorithmic Foundations of Robotics X: Proc. 10th Workshop Algorithmic Foundations Robotics*, E. Frazzoli, T. Lozano-Perez, N. Roy, and D. Rus, Eds., 2013, pp. 157–173.

[5] R. Fenton, G. Melocik, and K. Olson, "On the steering of automated vehicles: Theory and experiment," *IEEE Trans. Autom. Control*, vol. 21, no. 3, pp. 306–315, Jun. 1976.

[6] R. Rajamani, *Vehicle Dynamics and Control*, 2nd ed., ser. Mechanical Engineering Series. New York: Springer, 2012.

[7] L. J. Guibas, "Kinetic data structures: A state of the art report," in *Robotics: The Algorithmic Perspective: 3rd Workshop Algorithmic Foundations Robotics*, P. Agarwal, L. Kavraki, and M. Mason, Eds., Aug. 1998, pp. 191–209.

[8] J. Basch, L. Guibas, and J. Hershberger, "Data structures for mobile data," *J. Algorithms*, vol. 31, no. 1, pp. 1–28, Apr. 1999.

[9] D. Kirkpatrick, J. Snoeyink, and B. Speckmann, "Kinetic collision detection for simple polygons," in *Proc. 16th Annu. Symp. Computational Geometry*, 2000, pp. 322–330.

[10] P. K. Agarwal, J. Basch, L. J. Guibas, J. Hershberger, and L. Zhang, "Deformable free-space tilings for kinetic collision detection," *Int. J. Robot. Res.*, vol. 21, no. 3, pp. 179–197, Mar. 2002.

[11] J. Basch, J. Erickson, L. J. Guibas, J. Hershberger, and L. Zhang, "Kinetic collision detection between two simple polygons," *Computational Geometry*, vol. 27, no. 3, pp. 211–235, Mar. 2004.

[12] J. Gao, L. Guibas, J. Hershberger, L. Zhang, and A. Zhu, "Discrete mobile centers," in *Proc. 17th Annu. Symp. Computational Geometry*, 2001, pp. 188–196.

[13] J. Hershberger, "Smooth kinetic maintenance of clusters," in *Proc. 19th Annu. Symp. Computational Geometry*, 2003, pp. 48–57.

[14] Y. Li, J. Han, and J. Yang, "Clustering moving objects," in *Proc. 10th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, Aug. 2004, pp. 617–622.

[15] J. Basch, L. Guibas, and L. Zhang, "Proximity problems on moving points," in *Proc. 13th Annu. Symp. Computational Geometry*, 1997, pp. 344–351.

[16] P. K. Agarwal, H. Kaplan, and M. Sharir, "Kinetic and dynamic data structures for closest pair and all nearest neighbors," *ACM Trans. Algorithms*, vol. 5, no. 1, pp. 4:1–4:37, Dec. 2008.

[17] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the positions of continuously moving objects," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 331–342, May 2000.

[18] C. Ó Dúnlaing, M. Sharir, and C. K. Yap, "Retraction: A new approach to motion-planning," in *Proc. 15th Annu. ACM Symp. Theory Computing*, Dec. 1983, pp. 207–220.

[19] E. M. Arkin, S. P. Fekete, and J. S. Mitchell, "The lawnmower problem," in *Proc. 5th Canadian Conf. Computational Geometry*, Aug. 1993, pp. 461–466.

[20] E. M. Arkin, S. P. Fekete, and J. S. B. Mitchell, "Approximation algorithms for lawn mowing and milling," *Comput. Geom.*, vol. 17, no. 1, pp. 25–50, Oct. 2000.

[21] S. Arya, S.-W. Cheng, and D. M. Mount, "Approximation algorithm for multiple-tool milling," *Int. J. Comput. Geom. Appl.*, vol. 11, no. 03, pp. 339–372, Jun. 2001.

[22] F. Berger and R. Klein, "A traveller's problem," in *Proc. 26th Annu. Symp. Computational Geometry*, Jun. 2010, pp. 176–182.

[23] E. M. Arkin, J. S. Mitchell, and V. Polishchuk, "Maximum thick paths in static and dynamic environments," in *Proc. 24th Annu. Symp. Computational Geometry*, Jun. 2008, pp. 20–27.

[24] A. Kawamura and Y. Kobayashi, "Fence patrolling by mobile agents with distinct speeds," *Distrib. Comput.*, vol. 28, no. 2, pp. 147–154, 2015.

[25] F. Pasqualetti, A. Franchi, and F. Bullo, "On cooperative patrolling: Optimal trajectories, complexity analysis, and approximation algorithms," *IEEE Trans. Robot.*, vol. 28, no. 3, pp. 592–606, Jun. 2012.

[26] Y. Chiang, J. Klosowski, C. Lee, and J. Mitchell, "Geometric algorithms for conflict detection/resolution in air traffic management," in *Proc. 36th IEEE Conf. Decision and Control*, vol. 2, Dec. 1997, pp. 1835–1840.

[27] L. J. Guibas, J. S. B. Mitchell, and T. Roos, "Voronoi diagrams of moving points in the plane," in *WG 1991: Graph-Theoretic Concepts Computer Science*, ser. Lecture Notes in Computer Science, G. Schmidt and R. Berghammer, Eds., vol. 570. Springer, 1992, pp. 113–125.

[28] S. H. Arul, A. J. Sathyamoorthy, S. Patel, M. Otte, H. Xu, M. C. Lin, and D. Manocha, "LSwarm: Efficient collision avoidance for large swarms with coverage constraints in complex urban scenes," *IEEE Robot. Autom. Lett.*, vol. 4, no. 4, pp. 3940–3947, Oct. 2019.

[29] E. D. Demaine, S. P. Fekete, P. Keldenich, H. Meijer, and C. Scheffer, "Coordinated motion planning: Reconfiguring a swarm of labeled robots with bounded stretch," *SIAM J. Comput.*, vol. 48, no. 6, pp. 1727–1762, Jan. 2019.

[30] J. Reif and M. Sharir, "Motion planning in the presence of moving obstacles," *J. ACM*, vol. 41, no. 4, pp. 764–790, Jul. 1994.

[31] R. Sharma and Y. Aloimonos, "Coordinated motion planning: The warehouseman's problem with constraints on free space," *IEEE Trans. Syst. Man Cybern.*, vol. 22, no. 1, pp. 130–141, Jan. 1992.

[32] G. Flake and E. Baum, "Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants"," *Theor. Comput. Sci.*, vol. 270, no. 1-2, pp. 895–911, Jan. 2002.

[33] R. A. Hearn and E. D. Demaine, "PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation," *Theoretical Computer Science*, vol. 343, no. 1, pp. 72–96, Oct. 2005.

[34] N. Gupta and D. S. Nau, "On the complexity of blocks-world planning," *Artificial Intelligence*, vol. 56, no. 2, pp. 223–254, Aug. 1992.

[35] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ, USA: Prentice Hall, 1993.

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[37] J. Kleinberg and É. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley, 2005.

[38] A. V. Goldberg and R. E. Tarjan, "Finding minimum-cost circulations by canceling negative cycles," *J. ACM*, vol. 36, no. 4, pp. 873–886, Oct. 1989.

[39] J. Edmonds and R. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM*, vol. 19, no. 2, pp. 248–264, Apr. 1972.

[40] J. B. Orlin, "A polynomial time primal network simplex algorithm for minimum cost flows," *Math. Program.*, vol. 78, pp. 109–129, 1997.

[41] L. R. Ford and D. R. Fulkerson, "Constructing maximal dynamic flows from static flows," *Oper. Res.*, vol. 6, no. 3, pp. 419–433, May 1958.

[42] D. R. Ford and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ, USA: Princeton University Press, 2010.

[43] L. Fleischer and É. Tardos, "Efficient continuous-time dynamic network flow algorithms," *Oper. Res. Lett.*, vol. 23, no. 3–5, pp. 71–80, Oct. 1998.

[44] L. Fleischer and M. Skutella, "Quickest flows over time," *SIAM J. Comput.*, vol. 36, no. 6, pp. 1600–1630, Jan. 2007.

[45] K. Dresner and P. Stone, "A multiagent approach to autonomous intersection management," *J. Artif. Intell. Res.*, vol. 31, no. 1, pp. 591–656, Mar. 2008.

[46] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI Mag.*, vol. 29, no. 1, pp. 9–19, Mar. 2008.

[47] P. Fiorini and Z. Shiller, "Motion planning in dynamic environments using velocity obstacles," *Int. J. Robot. Res.*, vol. 17, no. 7, pp. 760–772, Jul. 1998.

[48] J. van den Berg, M. Lin, and D. Manocha, "Reciprocal Velocity Obstacles for real-time multi-agent navigation," in *2008 IEEE Int. Conf. Robotics and Automation*, May 2008, pp. 1928–1935.

[49] J. van den Berg, S. J. Guy, M. Lin, and D. Manocha, "Reciprocal n-body collision avoidance," in *Robotics Research*, ser. Springer Tracts in Advanced Robotics, B. Siciliano, O. Khatib, F. Groen, C. Pradalier, R. Siegwart, and G. Hirzinger, Eds. Berlin, Germany: Springer, 2011, no. 70, pp. 3–19.

[50] S. Akella and S. Hutchinson, "Coordinating the motions of multiple robots with specified trajectories," in *Proc. 2002 IEEE Int. Conf. Robotics and Automation*, May 2002, pp. 624–631.

[51] S. Petti and T. Fraichard, "Safe motion planning in dynamic environments," in *2005 IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS 2005)*, Aug. 2005, pp. 2210–2215.

[52] S. Rodriguez, J. Lien, and N. Amato, "A framework for planning motion in environments with moving obstacles," in *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2007)*, Oct. 2007, pp. 3309–3314.

[53] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robot. Autom.*, vol. 12, no. 4, pp. 566–580, Aug. 1996.

[54] S. LaValle and S. Hutchinson, "Path selection and coordination for multiple robots via Nash equilibria," in *Proc. 1994 IEEE Int. Conf. Robotics and Automation*, vol. 3, May 1994, pp. 1847–1852.

[55] N. Nisan, T. Roughgarden, É. Tardos, and V. V. Vazirani, *Algorithmic Game Theory*. Cambridge, UK: Cambridge University Press, Sep. 2007.

[56] R. J. Aumann, "Acceptable points in general cooperative n-person games," in *Contributions to the Theory of Games*, ser. Annals of Mathematics Studies, A. W. Tucker and R. D. Luce, Eds. Princeton, NJ, USA: Princeton University Press, 1959, vol. 4, no. 40, pp. 287–324.

[57] R. Koch and M. Skutella, "Nash equilibria and the price of anarchy for flows over time," *Theory Comput. Syst.*, vol. 49, no. 1, pp. 71–97, 2011.

[58] S. Kopparty and C. V. Ravishankar, "A framework for pursuit evasion games in $\mathbb{R}^n$," *Inf. Process. Lett.*, vol. 96, no. 3, pp. 114–122, Nov. 2005.

[59] D. Carlino, S. D. Boyles, and P. Stone, "Auction-based autonomous intersection management," in *16th Int. IEEE Conf. Intelligent Transportation Systems*, Oct. 2013, pp. 529–534.

[60] K. Dresner and P. Stone, "Multiagent traffic management: A reservation-based intersection control mechanism," in *Proc. 3rd Int. Joint Conf. Autonomous Agents and Multiagent Systems (AAMAS 2004)*, vol. 2, 2004, pp. 530–537.

[61] ——, "Multiagent traffic management: An improved intersection control mechanism," in *Proc. 4th Int. Joint Conference Autonomous Agents and Multiagent Systems (AAMAS 2005)*, Jul. 2005, pp. 471–477.

[62] M. VanMiddlesworth, K. Dresner, and P. Stone, "Replacing the stop sign: Unmanaged intersection control for autonomous vehicles," in *Proc. 7th Int. Joint Conf. Autonomous Agents and Multiagent Systems (AAMAS 2008)*, 2008, pp. 1413–1416.

[63] K. C. Wang and A. Botea, "MAPP: A scalable multi-agent path planning algorithm with tractability and completeness guarantees," *J. Artif. Intell. Res.*, vol. 42, pp. 55–90, Sep. 2011.

[64] M. Sasaki and T. Nagatani, "Transition and saturation of traffic flow controlled by traffic lights," *Physica A: Statistical Mechanics and its Applications*, vol. 325, no. 3, pp. 531–546, Jul. 2003.

[65] T. Nagatani, "The physics of traffic jams," *Rep. Prog. Phys.*, vol. 65, no. 9, pp. 1331–1386, Aug. 2002.

[66] C. Gershenson, "Self-organizing traffic lights," *Complex Syst.*, vol. 16, no. 1, pp. 29–53, 2005.

[67] S.-B. Cools, C. Gershenson, and B. D'Hooghe, "Self-organizing traffic lights: A realistic simulation," in *Advances in Applied Self-Organizing Systems*, 2nd ed., ser. Advanced Information and Knowledge Processing, M. Prokopenko, Ed. London, UK: Springer, 2013, pp. 45–55.

[68] A. John, A. Schadschneider, D. Chowdhury, and K. Nishinari, "Trafficlike collective movement of ants on trails: Absence of a jammed phase," *Phys. Rev. Lett.*, vol. 102, no. 10, p. 108001, Mar. 2009.

[69] S. Lammer and M. Treiber, "Self-healing networks: Gridlock prevention with capacity regulating traffic lights," in *IEEE 6th Int. Conf. Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, Sep. 2012, pp. 61–65.

[70] F. Amato, F. Basile, C. Carbone, and P. Chiacchio, "An approach to control automated warehouse systems," *Control Eng. Pract.*, vol. 13, no. 10, pp. 1223–1241, Oct. 2005.

[71] F.-L. Chang, Z.-X. Liu, Z. Xin, and D.-D. Liu, "Research on order picking optimization problem of automated warehouse," *Syst. Eng. - Theory Pract.*, vol. 27, no. 2, pp. 139–143, Feb. 2007.

[72] M. Sarrafzadeh and S. R. Maddila, "Discrete warehouse problem," *Theor. Comput. Sci.*, vol. 140, no. 2, pp. 231–247, Apr. 1995.

[73] K.-W. Pang and H.-L. Chan, "Data mining-based algorithm for storage location assignment in a randomised warehouse," *Int. J. Prod. Res.*, vol. 55, no. 14, pp. 4035–4052, Jul. 2017.

[74] A. Cahn, "The warehouse problem [abstract 505]," in *Bulletin of the American Mathematical Society*, vol. 54. American Mathematical Society, Nov. 1948, p. 1073.

[75] A. Charnes and W. W. Cooper, "Generalizations of the warehousing model," *Oper. Res. Q.*, vol. 6, no. 4, pp. 131–172, 1955.

[76] L. A. Wolsey and H. Yaman, "Convex hull results for the warehouse problem," *Discrete Optim.*, vol. 30, pp. 108–120, Nov. 2018.

[77] J. E. Hopcroft, J. T. Schwartz, and M. Sharir, "On the complexity of motion planning for multiple independent objects: PSPACE-hardness of the "warehouseman's problem"," *Int. J. Robot. Res.*, vol. 3, no. 4, pp. 76–88, 1984.

[78] J.-F. Cordeau and G. Laporte, "The Dial-a-Ride Problem: Models and algorithms," *Ann. Oper. Res.*, vol. 153, no. 1, pp. 29–46, Sep. 2007.

[79] E. Koutsoupias, "The k-server problem," *Comput. Sci. Rev.*, vol. 3, no. 2, pp. 105–118, May 2009.

[80] A. Lodi, S. Martello, and M. Monaci, "Two-dimensional packing problems: A survey," *Eur. J. Oper. Res.*, vol. 141, no. 2, pp. 241–252, Sep. 2002.

[81] K. Stephenson, *Introduction to Circle Packing: The Theory of Discrete Analytic Functions*. New York, NY, USA: Cambridge University Press, 2005.

[82] S. P. Fekete and H.-F. Hoffmann, "Online square-into-square packing," *Algorithmica*, vol. 77, no. 3, pp. 867–901, Mar. 2017.

[83] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir, "A model for hierarchical memory," in *Proc. 19th Annu. ACM Symp. Theory of Computing*, Jan. 1987, pp. 305–314.

[84] D. D. Sleator and R. E. Tarjan, "Amortized efficiency of list update and paging rules," *Commun. ACM*, vol. 28, no. 2, pp. 202–208, Feb. 1985.

[85] G. Cookson, "INRIX global traffic scorecard," INRIX Research, Kirkland, WA, USA, Tech. Rep., Feb. 2018.

[86] I. Karamouzas, N. Sohre, R. Narain, and S. J. Guy, "Implicit crowds: Optimization integrator for robust crowd simulation," *ACM Trans. Graph.*, vol. 36, no. 4, pp. 136:1–136:13, Jul. 2017.

[87] T.-C. Au and P. Stone, "Motion planning algorithms for autonomous intersection management," in *Workshop 24th AAAI Conf. Artificial Intelligence*, Jul. 2010, pp. 2–9.

[88] D. Fajardo, T. Au, S. Waller, P. Stone, and D. Yang, "Automated intersection control," *Transp. Res. Rec. J. Transp. Res. Board*, vol. 2259, no. 1, pp. 223–232, Dec. 2011.

[89] J. J. B. Vial, W. E. Devanny, D. Eppstein, and M. T. Goodrich, "Scheduling autonomous vehicle platoons through an unregulated intersection," in *16th Workshop Algorithmic Approaches Transportation Modelling, Optimization, and Systems (ATMOS 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Goerigk and R. Werneck, Eds., vol. 54. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 1–14.

[90] B. Aspvall, M. F. Plass, and R. E. Tarjan, "A linear-time algorithm for testing the truth of certain quantified boolean formulas," *Inf. Process. Lett.*, vol. 8, no. 3, pp. 121–123, Mar. 1979.

[91] R. Tachet, P. Santi, S. Sobolevsky, L. I. Reyes-Castro, E. Frazzoli, D. Helbing, and C. Ratti, "Revisiting street intersections using slot-based systems," *PLOS ONE*, vol. 11, no. 3, p. e0149607, Mar. 2016.

[92] S. I. Guler, M. Menendez, and L. Meier, "Using connected vehicle technology to improve the efficiency of intersections," *Transp. Res. Part C Emerg. Technol.*, vol. 46, pp. 121–131, Sep. 2014.

[93] J. Edmonds and E. L. Johnson, "Matching, Euler tours and the Chinese postman," *Math. Program.*, vol. 5, pp. 88–124, 1973.

[94] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1979.

[95] D. Jain and Y. Sharma, "Adoption of next generation robotics: A case study on Amazon," *Perspect. Case Res. J.*, vol. III, pp. 9–23, 2017.

[96] C. Lee, "Development of an Industrial Internet of Things (IIoT) based smart robotic warehouse management system," in *CONF-IRM 2018 Proc.*, ser. 43, 2018.

[97] R. B. Nelsen, *Proofs Without Words: Exercises in Visual Thinking.* Washington, DC, USA: The Mathematical Association of America, 1993.

[98] D. Eppstein and S. Gupta, "Crossing patterns in nonplanar road networks," in *Proc. 25th ACM SIGSPATIAL Int. Conf. Advances Geographic Information Systems*, Nov. 2017, pp. 1–9.

[99] G. Boeing, "Planarity and street network representation in urban form analysis," *Environ. Plan. B Urban Anal. City Sci.*, Nov. 2018.

[100] D. Lichtenstein, "Planar formulae and their uses," *SIAM J. Comput.*, vol. 11, no. 2, pp. 329–343, May 1982.

[101] S. P. Fekete, J.-M. Reinhardt, and C. Scheffer, "An efficient data structure for dynamic two-dimensional reconfiguration," *J. Syst. Archit.*, vol. 75, pp. 15–25, Apr. 2017.

[102] M. Çelik and H. Süral, "Order picking in a parallel-aisle warehouse with turn penalties," *Int. J. Prod. Res.*, vol. 54, no. 14, pp. 4340–4355, Jul. 2016.