5-2014

On the Complexity of Collecting Items With a Maximal Sliding Agent

Pedro J. Tejada
Utah State University

ON THE COMPLEXITY OF COLLECTING ITEMS WITH A MAXIMAL SLIDING AGENT

by

Pedro J. Tejada

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

DOCTOR OF PHILOSOPHY

in

Computer Science

Approved:

Dr. Minghui Jiang Major Professor	Dr. Haitao Wang Committee Member
Dr. Nicholas Flann Committee Member	Dr. Xiaojun Qi Committee Member
Dr. David Brown Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2014

ABSTRACT

On the Complexity of Collecting Items with a Maximal Sliding Agent

by

Pedro J. Tejada, Doctor of Philosophy

Utah State University, 2014

Major Professor: Dr. Minghui Jiang
Department: Computer Science

We study the computational complexity of collecting items inside a grid map with obstacles, using an agent that always slides to the maximal extend, until it is stopped by an obstacle. We show that the problem of deciding if all the items can be collected can be solved in polynomial time, and consider two natural optimization problems: one for determining the maximum number of items that can be collected, and another one for determining the minimum number of moves required to collect all the items. For the maximization problem we give a simple 2-approximation algorithm and for the parameterized version of the minimization problem we give an efficient fixed-parameter algorithm. We also show that both optimization problems are APX-hard, and show that the maximization problem is at least as hard to approximate as Max-2-SAT, while the minimization problem is NP-hard to approximate within $2 - \epsilon$, for any fixed $\epsilon > 0$. Furthermore, we show that the problem of deciding if all the items can be collected is NP-complete in higher dimensions, and that it is PSPACE-complete with blocks that can be pushed and slide with the agent.

(81 pages)

PUBLIC ABSTRACT

On the Complexity of Collecting Items with a Maximal Sliding Agent

Pedro J. Tejada

We study the computational complexity of collecting items inside a grid map with obstacles, using an agent that always slides to the maximal extend, until it is stopped by an obstacle. An agent could be, for example, a robot or a vehicle, while obstacles could be walls or other immovable objects, and items could be packages that need to be picked up.

This problem has very natural applications in robotics. The restricted type of motion of the agent naturally models movement on a frictionless surface, and movement of a robot with limited sensing capabilities and thus limited localization. For example, if a robot cannot determine the distance traveled once it starts moving, then it makes sense to keep moving until an obstacle is reached, even if the robot has a map of the environment.

With today's technology it is possible to create sophisticated robots but, since the complexity and the costs of such robots are high, it is sometimes better to use simple inexpensive robots that can still solve relatively complex tasks. In fact, simple robots are quite common and usually built using simple sensors that have limited capabilities, but that are easy to use and are considerably cheaper than more sophisticated ones.

The computational complexity of numerous problems with movable objects has been extensively studied before. However, only a few of them have maximal sliding agents, and they usually do not have the goal of collecting items. We show that the problem of deciding if all the items can be collected by a maximal sliding agent can be solved efficiently when the agent is the only moving object in the map. However, we show that optimization problems such as determining the minimum number of moves required to collect all the items, and also variants in more complex environments are computationally intractable. Hence, for those problems it is better to focus on using heuristics than on finding optimal solutions.

ACKNOWLEDGMENTS

I thank my advisor, Dr. Minghui Jiang, for teaching me about theoretical computer science, of which I knew little before meeting him, for his patience, and for his support during my time at Utah State University. Looking back at our time together, I am also grateful for some valuable life lessons I have learned from him that I believe will be useful in both my professional career and my personal life.

I thank the rest of my committee for their valuable feedback, and for the encouragement that helped me stay motivated to finish my dissertation.

I thank my family for their unconditional love and support. I thank Mari for believing in me and helping me smile during hard times. I thank Sun.

Pedro J. Tejada

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

We study the computational complexity of collecting *items* inside a *grid map* with *obstacles*, using an *agent* that always slides to the *maximal* extend, until it is stopped by an obstacle. An agent could be, for example, a robot or a vehicle, while obstacles could be walls or other immovable objects, and items could be packages that need to be picked up. Initially, the agent is placed at a starting location inside the map, and the goal is to determine how to move the agent to collect the items by visiting their locations. With the *maximal sliding* movement of the agent, an item is collected whenever the agent slides over it, even if the agent cannot stop at the item's location.

This problem has very natural applications in robotics. The restricted type of motion of the agent naturally models movement on a frictionless surface, and movement of a robot with limited sensing capabilities and thus limited localization. For example, if a robot cannot determine the distance traveled once it starts moving, then it makes sense to keep moving until an obstacle is reached, even if the robot has a map of the environment.

With today's technology it is possible to create sophisticated robots but, since the complexity and the costs of such robots are high, it is sometimes better to use simple inexpensive robots that can still solve relatively complex tasks [1, 2]. In fact, simple robots are quite common and usually built using inexpensive parts such as Arduino™ boards [3] together with simple sensors and actuators including touch sensors, photodiodes, and IR LEDs. Those sensors have limited sensing capabilities, but they are easy to use and they are considerably cheaper than more sophisticated ones.

1.1 The Game of Quell

Quell is a popular puzzle developed by Fallen Tree Games [4], which abstracts the

(a) (b)

Figure 1.1. Screenshots of two Quell maps. Obstacles are shown as square blocks, pearls as small golden circles, and the water droplet as a larger blue circle. (a) A simple map with only obstacles, pearls, and the droplet; the three pearls can be collected with seven moves: *left, up, left, down, right, up, left.* (b) A more complex map with gaps in the boundary and with special objects; after the droplet exits through a gap in the boundary, it enters from the gap on the opposite side; after the droplet enters one golden ring, it is teleported to the other ring and keeps sliding in the same direction; the one-pass gate, shown as a small dark green circle (below the left ring), turns into an obstacle after the droplet passes through it; the block with the ♂ symbol can be pushed by the droplet; the droplet is destroyed when it slides onto a spike; the switch (above the right ring) changes the directions of the spikes when the droplet passes through it; the four pearls can be collected with 26 moves.

problem that we study using a *droplet* of water as the agent, and *pearls* as the items. In its most basic form it is essentially the same as Andrea Gilbert's Tilt-Maze puzzle [5, 6], which precedes Quell by about a decade, but has not been rigorously studied from a computational complexity point of view. We started the study of our problem, together with Minghui Jiang and Haitao Wang, with the study of Quell in [7]. Thus our results in this dissertation are stated in terms of Quell, using the droplet as the agent and the pearls as the items.

In the original puzzle there are many types of objects other than obstacles and pearls, and moreover, the maps do not have to be bounded by obstacles; see Figure 1.1 for some example maps. However in [7] and in this dissertation, most of our results focus on the basic version of the puzzle where the maps contain only the obstacles, the pearls, and the droplet, and where the pearls and the droplet are contained in a connected region bounded by obstacles. If a map has no holes, then we call it a *simple* map. It is easy to verify that our results for Quell also apply for Tilt-Maze.

1.1.1 Computational Problems

Given a Quell map, the most fundamental problem is the problem of deciding if it is possible to collect all the pearls, but two optimization problems, as well as their parameterized decision versions, also arise naturally. If all the pearls cannot be collected, it is natural to ask for the maximum number of pearls that can be collected. Otherwise, if all the pearls can be collected, it is natural to ask for the minimum number of moves required to collect them. We next define these problems:

ANY-MOVES-ALL-PEARLS: decide if it is possible to collect all the pearls using any number of moves.

ANY-MOVES-MAX-PEARLS: determine the maximum number of pearls that can be collected using any number of moves.

MIN-MOVES-ALL-PEARLS: determine the minimum number of moves required to collect all the pearls.

ANY-MOVES-k-PEARLS: decide whether at least k pearls can be collected using any number of moves;

k-MOVES-ALL-PEARLS: decide whether all the pearls can be collected using at most k moves.

1.2 Summary of Results

In this dissertation, we first give algorithms for some of the computational problems that we study, including a polynomial-time exact algorithm for ANY-MOVES-ALL-PEARLS, a polynomial-time constant approximation for ANY-MOVES-MAX-PEARLS, and a fixed-parameter algorithm for k-MOVES-ALL-PEARLS. Then, we obtain approximation lower bounds for both optimization problems, and give evidence that a constant approximation for MIN-MOVES-ALL-PEARLS, if one exists, may be difficult to obtain. Finally, we analyze the complexity of ANY-MOVES-ALL-PEARLS, which we know can be solved in polynomial time in 2-dimensional maps where the droplet is the only moving object, in more complex environments, and show that it is NP-complete in higher dimensions, even if the droplet is

still the only moving object, and PSPACE-complete with blocks that can be pushed by and slide with the droplet, even if the number of dimensions of the map is still two.

Some of the results presented in this dissertation first appeared in [7] and are joint work with Minghui Jiang and Haitao Wang: the algorithms for any-Moves-all-Pearls and any-Moves-max-Pearls, and the lower bounds for both optimization problems. Other results in [7] include fixed-parameter algorithms for the parameterized versions of both optimization problems, and a proof that they are in NP and thus are NP-complete. The additional results presented in this dissertation include a faster algorithm for k-Moves-all-Pearls than the one in [7], the proof that any-Moves-all-Pearls is NP-complete in higher dimensions, and the proof that any-Moves-all-Pearls is PSPACE-complete with blocks that can be pushed by and slide with the droplet.

1.3 Related Work

Considerable work has been done in the field of recreational mathematics, with several books written on the study of games and puzzles, for example, the classic books by Berlekamp et al. [8] and Nowakowski [9], and the more recent book by Hearn and Demaine [10]. Also refer to the surveys by Kendal et al. [11] and by Demaine and Hearn [12], as well as the papers by Forišek [13] and Viglietta [14], for an extensive list of algorithmic and complexity results. In particular, there is a variety of games whose complexity has been analyzed, which are played by moving objects around a map (like the droplet in Quell) in order to get some of them to one or more specific locations. Those games include sliding-block and pushing-block puzzles, as well as games where the goal is to collect a set of items (like the pearls in Quell).

Sliding-block puzzles, like the famous 15-Puzzle and its generalization, the $(n^2 - 1)$-Puzzle [15], have several blocks that can be moved by an external agent; they include the Warehouseman's Problem [16], Sliding Blocks [17], Rush Hour [18], Lunar Lockout [19], Randolph's Robot Game [20], and Atomix [21]. Pushing-block puzzles, on the other hand, have an internal agent that moves inside the game world and is able to push blocks; they include the well known Sokoban [22, 23] where the goal is to move a set of blocks into special target locations, and other puzzles such as Push [24, 25] and PushPush [24, 26] where the

goal is to move the internal agent to a destination by pushing blocks out of the way. Both of these types of puzzles are usually complex because of the changing environment with multiple movable objects; they have generally been shown to be NP-hard [15, 24] or even PSPACE-hard [16, 17, 18, 19, 21, 23, 25, 26], which is a stronger result.

Games where the goal is to collect a set of items also have an internal agent that moves inside the game world trying to reach the locations of the items, so in a way they are similar to pushing block puzzles, but they are also different because they do not have to have pushable blocks or any other movable objects. Nevertheless, they often have special elements to make them more challenging; for example, classic video games such as Pac-Man, Lode Runner, and Boulder Dash have elements such as monsters and special abilities, and are known to be NP-hard [14]. In general, sliding and pushing-block puzzles that have multiple movable objects are related to motion planning problems, while collecting items in the absence of other movable objects is more closely related to path finding problems and the Traveling Salesman Problem (TSP).

Item collecting games using maximal sliding have not been extensively studied, but some simple results have been obtained that are independent of the way the agents move. For example, games where the player has to collect a set of items are NP-hard if there is a time limit [13] or if paths can be traversed at most once [14]. However, these results are not surprising and the proofs use simple reductions from Hamiltonian Cycle [27]. In general, the problems that we study are considerably easier to solve without maximal sliding. For example, if the agent can move from its current location to any adjacent cell in the grid map, then any-Moves-max-Pearls becomes merely a graph connectivity problem, which can be solved in linear time by doing a graph traversal, and min-Moves-all-Pearls turns into TSP in grid graphs, which is know to be NP-hard [28] but admits a 3/2-approximation by the well known Christofides algorithm [29]. In contrast, we show that with maximal sliding both optimization problems are APX-hard, and we also obtain a stronger lower bound for min-Moves-all-Pearls of $2 - \epsilon$, for any fixed $\epsilon > 0$.

Other work on problems with movable objects include, for example, the study by Wilfong of a motion planning problem where an internal agent is able to push and also pull blocks [30], and the study by Reif and Sharir of motion planning in the presence of moving objects of known easily computed trajectories [31]. Also, motion planning on graphs, which forms an abstraction of such problems, was considered by Papadimitriou et al. [32].

Because many problems with multiple movable objects are computationally intractable, work has been done using heuristics to find solutions for some of them. These heuristics can often lead to significant improvements for a state space search, but most of the time they do not offer guarantees on the time to find a solution. Moreover, in many cases even small instances of this type of problems with multiple movable objects are hard to solve, since the branching factor is usually very high. See for example [33, 34].

1.3.1 Other Problems with Maximal Sliding

As we have seen, the computational complexity of numerous games and puzzles with movable objects has been extensively studied before. However, only a few of them use maximal sliding. Puzzles with maximal sliding objects that have been studied before include Lunar Lockout [19, 35], Randolph's Robot Game [20], Atomix [21], PushPush and PushPushPush [26], and Pokémon [36]. The goal in Lunar Lockout, Randolph Robot, PushPush, PushPushPush, and Pokémon is to move a specific agent to a specific location. The goal in Atomix is to assemble a specific pattern with different moving objects.

Note that in Lunar Lockout, Atomix, and Randolph's Robot Game, there are multiple agents that can move. Also in PushPush and PushPushPush, there is only one agent that can move, but it can push some blocks, and in Pokémon, there is only one agent that can move, but it can trigger the movement of multiple enemy trainers by moving into their lines of sights. Thus in all of these games there are multiple objects that can move.

We next give more details about some of these games:

Lunar Lockout: There are multiple robots, and the goal is to move a specific robot to a specific location. Robots must slide until they hit another robot and sliding a

robot into the edge of the map is not allowed. It was shown to be NP-hard by Hock in 2002 [35], and a generalization with fixed obstacles was shown to be PSPACE-complete by Hartline and Libeskind-Hadas in 2003 [19]. Whether it is PSPACE-complete without fixed obstacles remains open.

Randolph's Robot Game: This puzzle is similar to Lunar Lockout with fixed obstacles. Again, the goal is to move a specific robot to a specific location, but robots are allowed to slide into the edge of the map. It was shown to be NP-hard by Engels and Kamphans in 2006 [20]. They also observed in a more complete technical report [37] that the 2004 proof for Atomix by Holzer and Schwoon [21] could be modified to show that the problem is actually PSPACE-complete.

Atomix: There are multiple agents (*atoms*), each labeled with a type, and the goal is to arrange them into a specific pattern (*molecule*); the location of the map where the pattern is arranged is irrelevant. Agents must slide until they hit another agent or an obstacle. It was first shown to be NP-hard by Hüffner et al. in 2001 [33], and then PSPACE-complete by Holzer and Schwoon in 2004 [21].

PushPush and PushPushPush: In PushPush there is one agent and blocks that can be pushed by the agent, and the goal is to move the agent to a specific location; the agent does not use maximal sliding but the blocks pushed by the agent do. PushPushPush is similar to PushPush, but a pushed block may push other blocks while sliding. Some versions are NP-hard while others are PSPACE-complete [24, 25, 26].

1.4 Relevance of Our Study

The problems that we study in this dissertation are interesting because agents with and without maximal sliding behave quite differently, and also because very few problems with maximal sliding have been studied before. Moreover, our results are important because:

- The same problems without maximal sliding are much simpler to solve.
- Non of the problems with maximal sliding objects mentioned before have the goal of collecting items.

- All of the problems with maximal sliding mentioned before have multiple objects that can move, and their complexity is mainly due to the interactions between those objects. In contrast, most of our results focus on the basic version of Quell where the droplet is the only moving object and the complexity is mainly due to the restrictions on the maximal sliding movement of the droplet.
- Most of the results mentioned before for games with movable objects focus on showing intractability of the basic decision versions of the problems. In contrast, we have a variety of results for different versions of Quell, including polynomial time solvability, approximation algorithms, fixed-parameter tractability, approximation lower bounds, and intractability in more complex environments.

1.5 Dissertation Outline

The rest of this dissertation is organized as follows. In Chapter 2, we give algorithms for some of the computational problems that we consider, including a polynomial-time exact algorithm for ANY-MOVES-ALL-PEARLS, a polynomial-time constant approximation for ANY-MOVES-MAX-PEARLS, and a fixed-parameter algorithm for k-MOVES-ALL-PEARLS. In Chapter 3, we give approximation lower bounds for both optimization problems, ANY-MOVES-MAX-PEARLS and MIN-MOVES-ALL-PEARLS, and give evidence that a constant approximation for MIN-MOVES-ALL-PEARLS, if one exists, may be difficult to obtain. In Chapter 4, we prove that ANY-MOVES-ALL-PEARLS, which we know can be solved in polynomial-time in 2-dimensional maps, turns out to be NP-complete even in very simple 3-dimensional maps. In Chapter 5, we prove that ANY-MOVES-ALL-PEARLS is PSPACE-complete in 2-dimensional maps with pushable blocks. In Chapter 6, we conclude, leave some open questions, and give directions for future work.

CHAPTER 2

POLYNOMIAL TIME ALGORITMS

In this chapter we give algorithms for some of the computational problems that we study. We give a polynomial-time exact algorithm for the basic decision problem ANY-MOVES-ALL-PEARLS, a polynomial-time constant approximation for ANY-MOVES-MAX-PEARLS, and a fixed-parameter algorithm for k-MOVES-ALL-PEARLS, which runs in polynomial time for any fixed value of the parameter k.

Observe that a map is described by specifying the location of every object in the grid, including the obstacles, the pearls, and droplet. However, it is possible to describe a map using other representations that are more useful for our algorithms. We describe those representations first. Refer to Figure 2.1 for an illustration.

Let M be a map. Instead of specifying the location of every obstacle in the grid, we

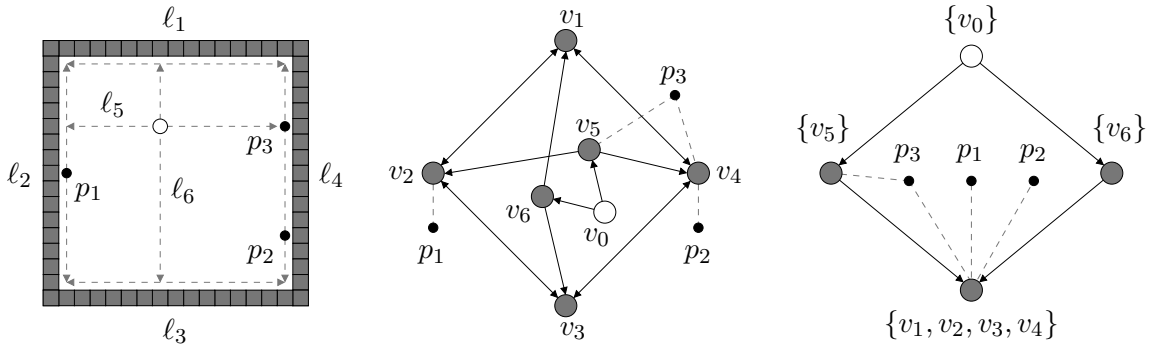


Figure 2.1. Directed graph representation of a Quell map. (a) Map M with $n = 4$ edges and $p = 3$ pearls; pearls are shown as black circles and the starting location is shown as a white circle; the droplet can traverse any of the dashed line segments. (b) Directed graph G; vertex v_0, shown as a white circle, represents the droplet's starting location, and each vertex v_i, shown as a gray circle, represents the line segment ℓ_i in M, for $i \geq 1$. (c) Strongly-connected component graph G' of G. The pearls are associated with the vertices of G and G' as indicated by dashed lines.

can represent the empty space in the map more compactly by describing the boundary of the obstacles as a set of closed rectilinear polygons. Let n be the number of edges of the set of closed rectilinear polygons in the compact representation of M, and let p be the number of pearls. Then the droplet can move along a *segment* incident to the droplet's starting location, or along a segment next to one of the edges in the set of polygons. Thus the droplet can move along at most $n + 2$ horizontal or vertical *maximal segments*.

Let G be the directed graph defined as follows: G has a special vertex v_0 for the starting location of the droplet, and it has a vertex for each maximal segment; and also, G has a directed edge from v_0 to the vertices corresponding to the maximal segments incident to the starting location of the droplet, and a directed edge from vertex u to vertex v if one of the endpoints of the maximal segment for u is on the maximal segment for v. Then G has at most $n + 3$ vertices and each vertex of G has out-degree at most two, and thus the size of G is $O(n)$. Moreover, let G' be the strongly-connected component graph of G, obtained by contracting each strongly-connected component of G into a single vertex. Then the special vertex v_0 must be in a component by itself, with no incoming edges. Each pearl is associated with a vertex v of G and with the component of G' containing v if and only if it is covered by the maximal segment for v.

Given the grid representation of a map, its compact representation can easily be obtained in time polynomial in the size of the grid, by traversing the grid. From the compact representation, we can construct the directed graph G in $O(n \log n)$ time by the standard sweeping technique in computational geometry [38], and after sorting the $O(n)$ maximal segments represented by the vertices in G, we can associate the p pearls in the map with the vertices of G in $O(p \log n)$ time. Moreover, from G, we can construct G' and associate all the pearls with the components in G' in $O(n)$ time by depth-first search.

We prove the following theorems:

Theorem 1. ANY-MOVES-ALL-PEARLS *admits a polynomial-time exact algorithm.*

Theorem 2. ANY-MOVES-MAX-PEARLS *admits a polynomial-time* 2*-approximation algorithm.*

Theorem 3. *k-Moves-all-Pearls admits a $2^k \cdot poly(n+p)$ time exact algorithm.*

Previously in [7], we proved that k-Moves-all-Pearls admits a $c^k \cdot \text{poly}(n+p)$ time exact algorithm where $c = 1 + \sqrt{3} = 2.73205\ldots$. The new algorithm that we propose in here for k-Moves-all-Pearls improves the value of the constant to $c = 2$.

2.1 Exact Algorithm for any-Moves-all-Pearls

In this section we prove Theorem 1. We reduce any-Moves-all-Pearls to 2-SAT. Given a set V of boolean variables and a set C of clauses, where each clause is the disjunction of at most two literals, 2-SAT is the problem deciding if there is an assignment of V that satisfies all the clauses in C.

Given a map M, we first compute the directed graph G and the strongly-connected component graph G'. Then, if a pearl is not associated with any of the components in G', it clearly cannot be collected. Otherwise, we construct a 2-SAT formula including a variable u for each component u in G', and including the following clauses:

- a *start clause* with a single literal u, where u is the component in G' that contains the special vertex v_0 in G;
- a *choice clause* $u \vee v$ for each pearl associated with components u and v in G', or with a single literal u if the pearl is associated with only one component u in G';
- a *conflict clause* $\bar{u} \vee \bar{v}$ for each pair of components u and v in G' such that there is no directed path in G' from either one to the other.

This completes the reduction. To collect all the pearls, the droplet must visit a subset of the components in G'. Observe how the choice clauses can be used to ensure that we choose set of components in G' such that each pearl is associated with at least one component. Moreover, observe how the start clause together with the conflict clauses can be used to ensure that all the chosen components can be visited by a single path in G' starting from the component containing the special vertex v_0.

The correctness of the algorithm is determined by the following lemma:

Lemma 1. *All the pearls can be collected if and only if the* 2-SAT *formula is satisfiable.*

Proof. We first prove the direct implication. Suppose there is a sequence of moves that collects all the pearls. This sequence corresponds to a walk in G starting from the special vertex v_0, which in turn corresponds to a path in G' starting from the component including v_0. For each component in G', we set the corresponding variable to true if and only if the component is visited by the path in G'. Then the start clause is clearly satisfied since the path starts from the component containing v_0. Since all the pearls are collected, each pearl must be associated with at least one of the components in the path. Thus, for each pearl, the corresponding choice clause has at least one of its positive literals set to true, and hence is satisfied. Since the path cannot visit any pair of components such that there is no directed path in G' from either one to the other, at least one of the components in each one of such pairs must be missed by the path. Thus, for each such pair of components, the corresponding conflict clause has at least one of its negative literals set to false, and hence is satisfied. Thus the 2-SAT formula is satisfied.

We next prove the reverse implication. Suppose the 2-SAT formula has a satisfying assignment. Consider the set of components in G' corresponding to the set of variables that are set to true by the satisfying assignment. Since all the conflict clauses are satisfied and G' is acyclic, there is a path in G' that visits all the components in this set in topological order. Moreover, since the start clause is satisfied, the path must start from the component containing v_0. For this path in G', obtain a walk in G that visits all the vertices in each one of the components in the path. Then, for the walk in G, obtain a sequence of moves for the droplet that completely traverses each one of the maximal segments for the vertices in the walk, using at most two moves for each segment. Since all the choice clauses are satisfied, each pearl is associated with at least one component in G', and thus at least one of the maximal segments containing the pearl must be completely traversed. Thus all the pearls are collected. □

We now analyze the running time of our algorithm. First, recall that obtaining the compact representation of a map can be done in time polynomial in the size of its grid

representation, and that from the compact representation, the directed graph G and the strongly-connected component graph G' can be constructed in $O((n+p)\log n)$ time. Then, the 2-SAT formula can be constructed in $O(n^2+p)$ time: the $O(p)$ choice clauses can easily be constructed in $O(p)$ time, and the $O(n^2)$ conflict clauses can be constructed in $O(n^2)$ time by doing a graph traversal from each one of the $O(n)$ components in G'. Finally, since the 2-SAT formula has $O(n)$ variables and $O(n^2+p)$ clauses, it can be solved in $O(n^2+p)$ time by the well-known linear-time algorithm for 2-SAT; see for example [39].

This concludes the proof of Theorem 1.

2.2 Constant Approximation for any-Moves-max-Pearls

In this section, we prove Theorem 2. We use a greedy algorithm.

Given a map, we first construct the strongly-connected component graph G', and we assign a weight $w(v)$ to each component v in G' equal to the number of pearls associated with the component; that is, the number of pearls that can be collected by traversing all the maximal segments corresponding to vertices in the component. Then, since G' is acyclic, a simple algorithm based on topological ordering can be used to find the path with largest weight: for example, let $T[v]$ be the weight of the path with largest weight ending at v, then $T[v] = \max_u T[u] + w(v)$, for all u with a directed edge from u to v. For this path in G' there is a sequence of moves that visits all the maximal segments associated with the components in the path, and thus collects all the pearls associated with them. Recall that a pearl can be collected by traversing either a horizontal segment or a vertical segment. Thus each pearl can be associated with at most 2 components, and hence counted at most twice, so the algorithm gives a 2-approximation.

This completes the proof of Theorem 2.

2.3 Fixed-Parameter Algorithm for k-Moves-all-Pearls

In this section we prove Theorem 3. We use a bounded search three algorithm, combined with a polynomial step for each node of the search three.

Observe that the direction of any move cannot be the same as the direction of the previous move, if any, so any move after the first one is either a back move (B), or a left turn (L), or a right turn (R). Thus, if we ignore the first move, any sequence of k moves can be represented as a string with length k over the alphabet $\{B, L, R\}$. Moreover, observe that any three consecutive moves never have to be on the same line, so the droplet never needs to be moved back twice consecutively. Thus we only have to consider sequences represented by strings without two consecutive Bs. The first observation clearly leads to a simple brute force algorithm with $O(3^k)$ branches, and we showed in [7] that by taking the second observation into account the total number of branches for k moves is $O(c^k)$, where $c = 1 + \sqrt{3} = 2.73205\ldots$.

We note that the algorithm in [7], based on the second observation described above, reduces the number of times it is necessary to choose among three possible moves but it may still have to do it quite often, which is why the number of branches it enumerates can still be much higher than $O(2^k)$. Thus, to further reduce the number of branches, we first enumerate a more restricted set of sequences, and then make modifications to those sequences to obtain other sequences that may collect all the pearls. Our algorithm may still have to choose among three possible moves sometimes, but the total number of branches it enumerates is $O(2^k)$ and the extra modifications for each enumerated sequence can be done in polynomial time.

The restricted set of sequences that we enumerate is obtained in part by ignoring some moves that are considered by the algorithm in [7] but are never part of an optimal solution. To ignore such moves we use a generalization of the second observation mentioned above. Observe that after any move the droplet stops at a *dead end* where only one move is possible, or at a 2-*way intersection* where two moves are possible, or at a 3-*way intersection* where three moves are possible. Moreover, we say that a *path* is a sequence of maximal segments with their endpoints connected at 2-way intersections, and it is easy to see that it is impossible to skip segments when traversing a path in any direction. Thus, once the droplet is moved back inside a path, it never has to be moved back again until after it exits

(a) (b)

Figure 2.2. Modifying a sequence enumerated by the first step of the algorithm to traverse a maximal segment that is only partially traversed by the original sequence after a left or right turn from a 3-way intersection. The segment is traversed by a turn in the opposite direction of the original turn, followed by a back move (from the place marked with a $\times$). Moves shown as solid lines are included in the original sequence, while moves shown as dashed lines are only included in the updated sequence. (a) A left turn L is replaced by two moves RB. (b) A right turn R is replaced by two moves LB. Note that these modifications are independent of the type of location reached after the turn, and that they only require one additional move in each case.

the path from one of its endpoints at a 3-way intersection.

Our algorithm has two steps. The first step enumerates all the sequences with length at most k excluding moves forbidden by our previous observation, which we know are never part of an optimal solution, and excluding back moves right after a left or right turn from a 3-way intersection. Then, for each sequence enumerated by the first step, we may obtain other sequences by making small local changes to traverse some extra maximal segments that are only partially traversed by the original sequence. If it is possible to collect all the remaining pearls by traversing some of those segments, the second step finds the minimum number of moves required to do so.

We first show how to obtain other sequences from one of the sequences enumerated by the first step of the algorithm. Observe that a back move always leaves the droplet on the same maximal segment traversed, perhaps only partially, on the previous move. Thus any sequence of moves including a back move right after a left or right turn from a 3-way intersection can easily be obtained from a similar sequence which turns in the opposite direction when it reaches that 3-way intersection. To obtain the new sequence including such a back move we simply have to replace a left turn L with two moves RB, or replace a right turn R with two moves LB. Note that these modifications are independent of the

type of location reached after the turn, and that they only require one additional move in each case. Refer to Figure 2.2 for an illustration.

We next explain the second step of the algorithm. Consider a sequence of i moves, $1 \leq i \leq k$, enumerated by the first step. We call S be the set of maximal segments that are only partially traversed by that sequence, after a left or right turn from a 3-way intersection, and including any of the remaining pearls. If there is a pearl that is not covered by one of the segments in S, then it is clearly impossible to collect all the pearls by making the changes described above to completely traverse some of the segments in S. Otherwise, all the remaining pearls can be collected and we have to find out if it is possible to collect them using at most $j = k - i$ additional moves.

Suppose that all the remaining pearls are covered by the segments in S. Since all the segments with pearls covered by only one segment in S must be traversed, we first select all of those segments and remove all the pearls covered by them. Then we can select another subset of S to cover the remaining pearls, if any. Let j_1 be the number of segments selected so far. We next find out if the remaining pearls can be covered by a subset of S with at most $j_2 = j - j_1$ segments.

Observe that each remaining pearl must be at the intersection of two maximal segments in S, one horizontal and one vertical, so we create a bipartite graph $G(S) = (V, H, E)$ with a vertex in V for each vertical segment in S, a vertex in H for each horizontal segment in S, and an edge in E between two vertices if and only if there is a pearl at the intersection of the corresponding segments. Then we find a minimum vertex cover in $G(S)$, and if the cover has size at most j_2 we return the sequence obtained by modifying the original sequence, enumerated by the first step, to traverse the j_1 segment selected before and the segments corresponding to the vertices in the cover.

This completes the description of our algorithm. We have the following lemma:

Lemma 2. *If all the pearls can be collected using at most k moves, the algorithm finds a sequence of at most k moves that collect all the pearls.*

Proof. Suppose all the pearls can be collected using at most k moves.

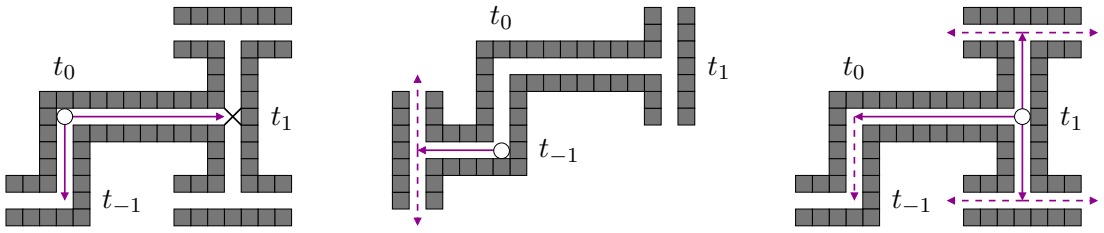
First note that the only optimal sequences excluded by the enumeration in the first step are the ones including back moves right after a left or right turn from a 3-way intersection. Hence, any optimal sequence can be obtained from one of the sequences enumerated in the first step by making the changes described above to traverse some extra maximal segments.

Consider a sequence enumerated by first step, from which an optimal sequence that collects all the pearls can be obtained. It is clear that the first j_1 segments selected by the second step, with pearls covered by only one segment in S, must always be traversed. Moreover, by finding a minimum vertex cover in $G(S)$ we determine the minimum number of extra segments required to cover the remaining pearls. Thus, the second step determines the minimum number of extra segments that must be traversed to collect all the pearls missed by the original sequence.

Recall that traversing each one of the extra segments selected by the second step requires one additional move. Thus the minimum number of extra segments that must be traversed to collect all pearls missed by the original sequence is equal to the minimum number of extra moves required to collect them. Since the second step finds a sequence that traverses at most $j = j_1 + j_2$ extra segments, that sequence has at most $k = i + j$ moves. □

We now analyze the running time of our algorithm. Finding an unweighted minimum vertex cover in a bipartite graph can be done in polynomial time based on the equivalence between vertex cover and maximum matching described in Köning's theorem [40, ch. 11, p. 178]. Thus the second step is polynomial and the exponential part of the running time is upper bounded by the number of sequences enumerated by the first step. Note that the first step of our algorithm greatly reduces the number of sequences that have to be enumerated when compared to the algorithm in [7]. However, it may still have to choose among three possible moves sometimes, so it is not obvious that the number of sequences it enumerates is $O(2^k)$. To show that it is, we have some case analysis.

Clearly, right after a move from a dead end, a back move is not necessary. Moreover, recall that right after a back move from a 3-way intersection, another back move is not necessary, and that right after a turn from a 3-way intersection the back move is ignored



(a) (b) (c)

Figure 2.3. Bounding the maximum number of branches after any three moves from a 2-way intersection that leads to a 3-way intersection (marked with a cross $\times$). The starting location in each figure is shown as a white circle. Moves shown as solid lines can be done from the starting location, while moves shown as dashed lines can be done after a first move. (a) After reaching a 2-way intersection, it is possible to move back or to turn right and reach the 3-way intersection. (b) If we move back, there are at most two branches for the next two moves. (c) If we turn right, there are at most five branches for the next two moves.

(by our definition of the first step). Thus the only cases where we may have to consider all three choices at a 3-way intersection is when that 3-way intersection is reached from a 2-way intersection. We next show that even if we have to consider all three choices at a 3-way intersection, the total number of branches after any three moves starting at the 2-way intersection from which that 3-way intersection can be reached is actually less than the number of branches if we had two choices every time.

Let t_1 be a 3-way intersection where we may have three choices, let t_0 be the 2-way intersection from which t_1 can be reached, and let t_{-1} be the location from which t_0 is reached. By the same reasons stated above, if t_0 is reached from a dead end or from a 3-way intersection, then a back move from t_0 does not have to be considered. Hence, to determine the maximum number of branches after any three moves starting at t_0, we focus on the case where t_{-1} is also a 2-way intersection and we actually have two possible choices after reaching t_0: to move back to t_{-1}, or to turn and reach t_1. Refer to Figure 2.3 for an illustration of the two possibilities.

Suppose we first move back from t_0 to t_{-1}. Then we do not have to move back again until after we exit the path including t_0 and t_{-1}, so after reaching t_{-1} we only have one choice, and for the next move we have at most two choices. Thus in total, we have at most

two branches for the next three moves if we move from t_0 to t_{-1} first; see Figure 2.3(b).

Suppose we first turn from t_0 and reach t_1. Then we have three possible choices: to move back to t_0, to turn left, or to turn right. If we move back to t_0, we do not have to move back again so we have only one choice. And if we turn left or right from t_0, we have at most two choices in each case, since we ignore back moves after a turn from a 3-way intersection. Thus in total, we have at most five branches for the next three moves if we move from t_0 to t_1 first; see Figure 2.3(c).

Then, considering all possible cases for the next three moves after reaching t_0 we have a total of at most seven branches, two if we move back to t_{-1} first and five if we turn to reach t_1 first, which is less than $2^3 = 8$ if we had two choices every time. Hence, the total number of branches enumerated by the algorithm is at most $O(2^k)$ and the running time of the algorithm is $2^k \cdot \text{poly}(n + p)$.

This concludes the proof of Theorem 3.

2.4 Discussion

We have given a polynomial-time exact algorithm for any-Moves-all-Pearls. This is interesting because it is not immediately obvious that any-Moves-all-Pearls is in P, and also because our algorithm shows an interesting relationship between any-Moves-all-Pearls and the 2-SAT problem.

We have also given a 2-approximation algorithm for any-Moves-max-Pearls. This algorithm is quite simple, so despite the fact that we do not have a better approximation algorithm, it is possible that a different algorithm could give a better approximation? We note that a simple example can be constructed to show that the 2-approximation given by the algorithm is tight.

Previously in [7], we proved that any-Moves-k-Pearls admits a $c^k \cdot \text{poly}(n + p)$ time exact algorithm for some high constant c, and that k-Moves-all-Pearls admits a $c^k \cdot \text{poly}(n + p)$ time exact algorithm where $c = 1 + \sqrt{3} = 2.73205\ldots$. According to Jiang [41], any-Moves-k-Pearls can be solved in $2^k \cdot \text{poly}(n + p)$ expected time by using the Koutis-Williams multilinear detection technique [42, 43], and here we have shown that

k-Moves-all-Pearls can be solved in $2^k \cdot \text{poly}(n+p)$ time. Both of these results together are a nice combination, and they leave the following interesting open question: does any-Moves-k-Pearls or k-Moves-all-Pearls admit a $c^k \cdot \text{poly}(n+p)$ time algorithm where c is strictly less than 2?

CHAPTER 3

APPROXIMATION LOWER BOUNDS

In this chapter, we give approximation lower bounds for both the maximization and the minimization problems. We first prove that ANY-MOVES-MAX-PEARLS is at least as hard to approximate as MAX-2-SAT, and then prove that MIN-MOVES-ALL-PEARLS is NP-hard to approximate within $2 - \epsilon$, for any fixed $\epsilon > 0$. Our proofs are based on polynomial-time reductions from the well known NP-hard problems MAX-2-SAT and 3-SAT, and we only create maps with obstacles in the boundary, so our bounds hold even in simple maps.

We have the following theorems:

Theorem 4. ANY-MOVES-MAX-PEARLS *is at least as hard to approximate as* MAX-2-SAT*, even in simple maps. In particular,* ANY-MOVES-MAX-PEARLS *is NP-hard to approximate within a factor of* $22/21 = 1.04761\ldots$*, and moreover it is NP-hard to approximate within a factor of* $1.05938\ldots$ *if the unique games conjecture is true.*

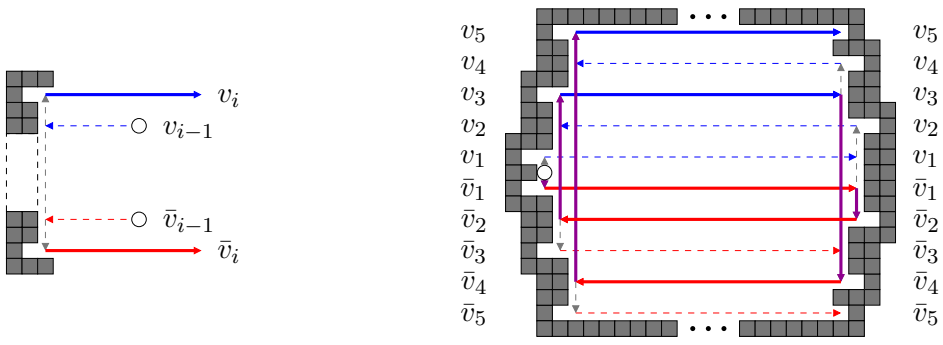
Theorem 5. MIN-MOVES-ALL-PEARLS *is NP-hard to approximate within* $2 - \epsilon$*, for any fixed* $\epsilon > 0$*, even in simple maps.*

In Chapter 2 we gave a polynomial-time 2-approximation algorithm for ANY-MOVES-MAX-PEARLS. However, we have not been able to obtain a constant approximation for MIN-MOVES-ALL-PEARLS. In fact, the best approximation that we know of for MIN-MOVES-ALL-PEARLS is an $O(n^2)$-approximation, which can be obtained by solving ANY-MOVES-ALL-PEARLS since collecting all the pearls always requires at most $O(n^2)$ moves; recall that the directed graph representation of a map has $O(n)$ vertices and edges. After proving our lower bounds for both problems, we give evidence suggesting that a constant approximation for MIN-MOVES-ALL-PEARLS, if one exists, may be difficult to obtain.

3.1 Approximation Lower Bound for ANY-MOVES-MAX-PEARLS

In this section we prove Theorem 4. We prove that ANY-MOVES-MAX-PEARLS is at least as hard to approximate as MAX-2-SAT by a gap-preserving reduction. Recall that, given a set V of boolean variables and a set C of clauses, where each clause is the disjunction of at most two literals, 2-SAT is the problem deciding if there is an assignment of V that satisfies all the clauses in C. Given a 2-SAT instance, MAX-2-SAT it is the problem of determining the maximum number of clauses in C that can be satisfied by any assignment of V. While 2-SAT can be solved in linear time by a well known algorithm, see for example [39], MAX-2-SAT is APX-hard [44, 45]. We refer the reader to [46], for example, for more details on MAX-2-SAT and complexity of approximation.

Let (V, C) be a MAX-2-SAT instance, where $V = \{v_1, \ldots, v_n\}$ is the set of variables and $C = \{c_1, \ldots, c_m\}$ is the set of clauses. Note that, since MAX-2-SAT without duplicate clauses is exactly as hard to approximate as MAX-2-SAT [47], we can assume without loss of generality that all the clauses in C are distinct. We create a map with two paths for the possible assignments of each variable, true or false, where each path intersects every other path, and we place m pearls, one pearl for each clause, at some of the intersections of those paths.



(a) (b)

Figure 3.1. Path choosing gadgets. (a) 2-choose-1 gadget for a single variable. After reaching the gadget from any of the paths for variable v_{i-1}, the droplet can traverse only one of the two paths for variable v_i. (b) 2-choose-1 gadgets for all the variables combined. The gadgets are arranged in layers, allowing the player to choose one path for each variable in sequence. The path shown here (using solid lines) corresponds to the assignment $v_1 =$ false, $v_2 =$ false, $v_3 =$ true, $v_4 =$ false, and $v_5 =$ true.

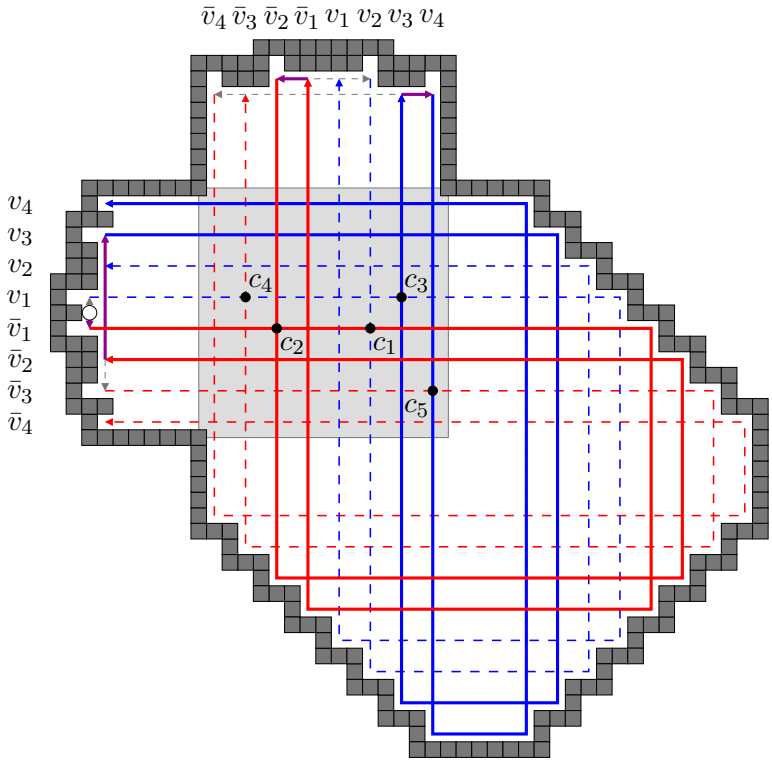


Figure 3.2. Construction for a MAX-2-SAT instance of $n = 4$ variables and $m = 5$ clauses $c_1 = \bar{v}_1 \vee v_2$, $c_2 = \bar{v}_1 \vee \bar{v}_2$, $c_3 = v_1 \vee v_3$, $c_4 = v_1 \vee \bar{v}_3$, and $c_5 = \bar{v}_3 \vee v_4$. Pearls are shown as black circles and the starting location, on the left border, is shown as a white circle. The basic layered structure of 2-choose-1 gadgets is twisted to ensure that each path intersects every other path in the shaded region. The path shown here (using solid lines) corresponds to the assignment v_1 = false, v_2 = false, v_3 = true, and v_4 = true, which satisfies all the clauses except c_4.

Refer to Figure 3.1 for an illustration of the basic gadget and the basic structure used in our construction. We use a 2-choose-1 gadget for each variable, and arrange the gadgets for all the variables into a layered structure. For each gadget, the upper (resp. lower) path in this structure is associated with the positive (resp. negative) literals of the corresponding variable. This structure allows the player to choose one path for each variable in sequence, and without allowing the player to move the droplet back to a previously visited gadget.

Refer to Figure. 3.2 for a complete example. Note how the basic layered structure illustrated in Figure 3.1(b) is twisted to ensure that each path intersects every other path in the shaded region. We place the pearl for each clause at the intersection of the paths for

its literals, inside the shaded region, using the horizontal segment from the path for the first literal, and the vertical segment from the path for the second literal.

This completes the construction. It is straightforward to verify the following lemma:

Lemma 3. *There is an assignment of V that satisfies k clauses of C if and only if k pearls in the map can be collected using any number of moves.*

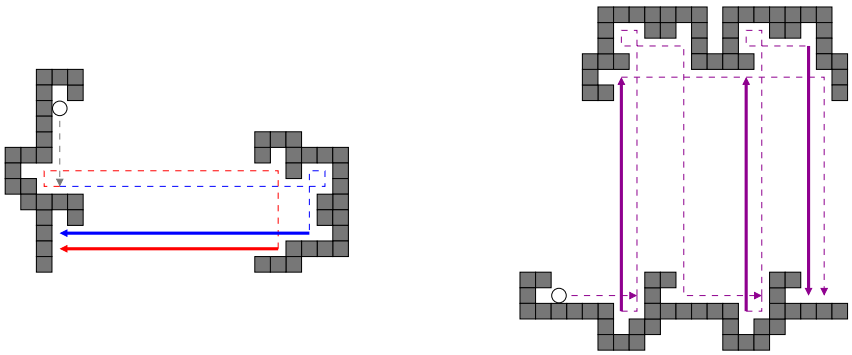
The reduction runs in polynomial time since the number of obstacles and pearls required is polynomial in the number of variables. Thus we have a gap-preserving reduction from Max-2-SAT to any-Moves-max-Pearls, and the lower bounds for any-Moves-max-Pearls follow from the known lower bounds for Max-2-SAT [44, 45].

This completes the proof of Theorem 4.

3.2 Approximation Lower Bound for min-Moves-all-Pearls

In this section we prove Theorem 5. We prove that min-Moves-all-Pearls is NP-hard to approximate within $2 - \epsilon$, for any fixed $\epsilon > 0$, by a reduction from 3-SAT. Given a set V of boolean variables and a set C of clauses, where each clause is the disjunction of at most three literals, 3-SAT is the problem deciding if there is an assignment of V that satisfies all the clauses in C. We refer the reader to [27], for example, for more details on 3-SAT and other NP-complete problems.

Let (V, C) be a 3-SAT instance, where $V = \{v_1, \ldots, v_n\}$ is the set of variables and $C = \{c_1, \ldots, c_m\}$ is the set of clauses. We create a map with two horizontal segments for the possible assignments of each variable, true or false, and with three vertical segments for the literals of each clause. The $2n$ segments for the variables intersect the $3n$ segments for the clauses, and we place $3m$ pearls, one pearl for each literal, at some of the intersections of those segments. All of the pearls in the map can always be collected, but we use gadgets to restrict the paths that can be followed, which makes a difference in the number of moves required to collect them. Then the minimum number of moves required to collect all the pearls determines whether the 3-SAT instance is satisfiable or not.



(a) (b)

Figure 3.3. Path choosing gadgets. (a) 2-choose-1 gadget. The droplet can traverse only one of the two solid horizontal segments. (b) 3-choose-1 gadget. The droplet can traverse only one of the three solid vertical segments.

Refer to Figure 3.3 for an illustration of the basic gadgets used in our construction. We use 2-choose-1 gadgets for the variables and 3-choose-1 gadgets for the clauses. The 2-choose-1 gadgets ensure that a path going through all the gadgets for the variables has to cover exactly one of the two horizontal segments for each variable, and the 3-choose-1 gadgets ensure that a path going through all the gadgets for the clauses once has to cover exactly one of the three vertical segments for each clause.

Refer to Figure 3.4 for a complete example. Any path through the map can only go through the 2-choose-1 gadgets for the variables once, and must go through all of them in sequence before going through any of the 3-choose-1 gadgets for the clauses. Then, it may go through all the 3-choose-1 gadgets for the clauses in sequence, and after that, it may follow a special back path to return and go through them again as many times as needed to collect the remaining pearls. Observe how the back path is constructed by adding the triangular region on the top right of the map.

For each literal of variable v_i that appears in clause c_j, we place a pearl at the intersection of one of the horizontal segments in the 2-choose-1 gadget for v_i and one of the vertical segments in the 3-choose-1 gadget for c_j. Denote by $l_{j,k}$ the k-th literal of c_j, $1 \leq k \leq 3$. We place the pearl for $l_{j,k}$ at the intersection of the k-th segment in the 3-choose-1 gadget for c_j, and the upper (resp. lower) segment in the 2-choose-1 gadget for v_i if the literal is a

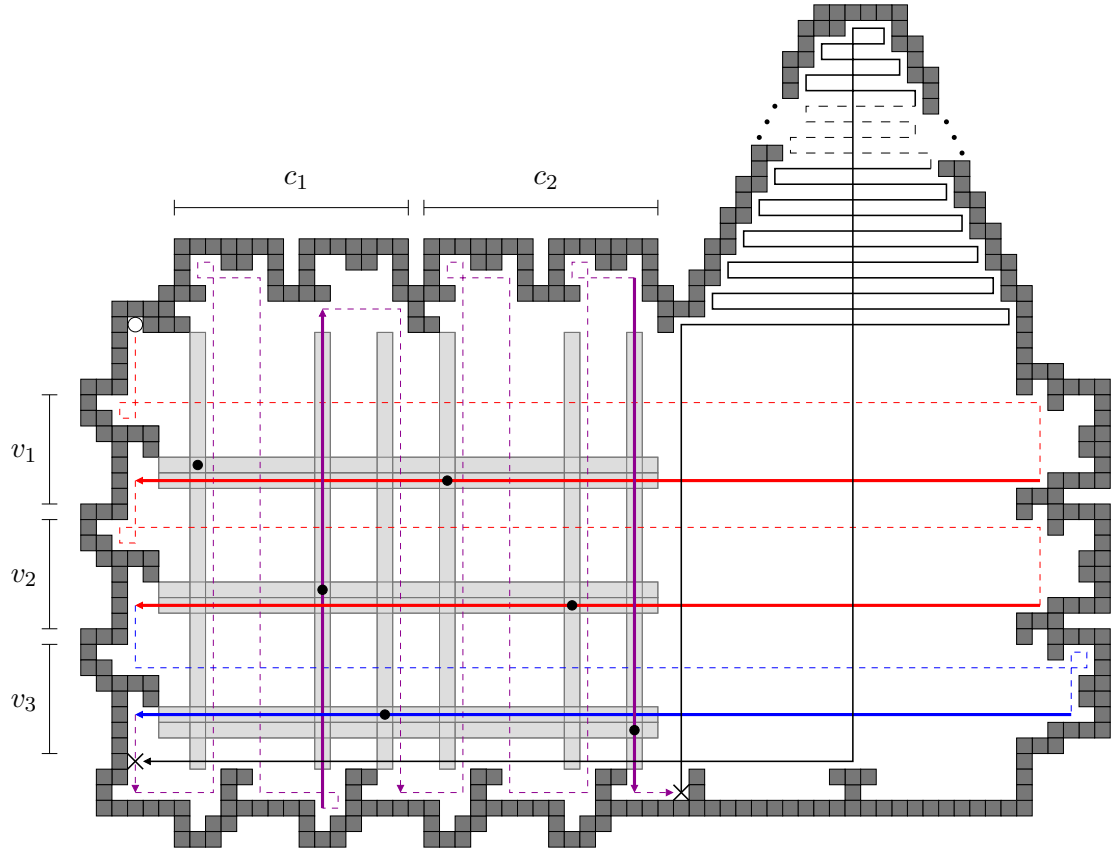


Figure 3.4. Construction for a 3-SAT instance of $n = 3$ variables and $m = 2$ clauses $c_1 = v_1 \vee v_2 \vee v_3$, and $c_2 = \bar{v}_1 \vee \bar{v}_2 \vee \bar{v}_3$. Pearls are shown as black circles and the starting location, at the top left corner, is shown as a white circle. After going through all the 2-choose-1 gadgets for the variables and then all the 3-choose-1 gadgets for the clauses, the droplet can follow the back path (with each end marked with a $\times$) to visit the 3-choose-1 gadgets again and collect the remaining pearls. The partial path shown here (using solid lines) corresponds to the satisfying assignment $v_1 =$ false, $v_2 =$ false, and $v_3 =$ true.

positive (resp. negative) literal of v_i.

This completes the construction. We have the following lemma:

Lemma 4. *If the* 3-SAT *instance* (V, C) *is satisfiable, then there is a path through the map that covers all the pearls and follows the back path at most once; otherwise, every path of the droplet that covers all the pearls must follow the back path at least twice.*

Proof. Consider any path through the map that covers all the pearls. Such a path must go through all the 2-choose-1 gadgets for the variables, and may have to go through some of

the 3-choose-1 gadgets for the clauses to collect any remaining pearls.

The part of the path that goes through all the 2-choose-1 gadgets corresponds to an assignment of the variables, where v_i is set to true (resp. false) if the path covers the upper (resp. lower) segment in the 2-choose-1 gadget for v_i. This assignment, which satisfies a subset of clauses, must cover at least one of the pearls for the literals of each satisfied clause, and none of the pearls for the literals of each unsatisfied clause. Thus, after going through all the 2-choose-1 gadgets, there could be at most two pearls remaining for each satisfied clause, and there must be exactly three pearls remaining for each unsatisfied clause.

By going through all the 3-choose-1 gadgets once, the path can cover at most one of the pearls for the literals of each clause. Hence, all the pearls for each satisfied clause must be covered by going through the 3-choose-1 gadgets at most twice and the back path at most once, while all the pearls for each unsatisfied clause can only be covered by going through the 3-choose-1 gadgets three times and the back path twice. □

By setting the length of the back path sufficiently large, but still polynomial in $(n+m)/\epsilon$, we make sure that the reduction runs in polynomial time and obtain the $2-\epsilon$ lower bound for any fixed $\epsilon > 0$.

Let s_1 be the maximum number of moves required to reach the start of the back path from the starting location, by going through all the 2-choose-1 and all the 3-choose-1 gadgets without any back moves, and let s_2 be the number of moves required to traverse the back path. Clearly, if the 3-SAT instance is satisfiable, the minimum number of moves required to collect all the pearls is at most $2s_1 + s_2$ since the back path has to be followed at most once, and otherwise, the instance is unsatisfiable and the minimum number of moves required is at least $2s_2$ since the back path has to be followed twice.

Suppose there is a $(2-\epsilon)$-approximation algorithm for MIN-MOVES-ALL-PEARLS. If the 3-SAT instance is satisfiable, the approximation algorithm must find a path that covers all the pearls using most $(2s_1 + s_2)(2-\epsilon)$ moves. By letting $(2s_1 + s_2)(2-\epsilon) = 2s_2$ and solving for s_2 we get $s_2 = (4/\epsilon - 2)s_1$. Thus for any s_2 greater than $(4/\epsilon - 2)s_1$, if the algorithm returns a value less than $2s_2$, the formula is satisfiable, and otherwise it is not.

Moreover, since s_1 is polynomial in $n+m$, then $s_2 = (4/\epsilon - 2)s_1$ is polynomial in $(n+m)/\epsilon$ and the reduction runs in polynomial time.

This completes the proof of Theorem 5.

3.3 More on the Inapproximability of MIN-MOVES-ALL-PEARLS

So far we have obtained approximation lower bounds for both optimization problems, ANY-MOVES-MAX-PEARLS and MIN-MOVES-ALL-PEARLS, and we know that their corresponding decision problems are fixed-parameter tractable; see [7] and our algorithm for k-MOVES-ALL-PEARLS in Chapter 2. However, we have only obtained a constant approximation for the maximization problem. It is an interesting open question whether MIN-MOVES-ALL-PEARLS admits a constant approximation too. In this section, we show that MIN-MOVES-ALL-PEARLS is likely to be at least as hard to approximate as a difficult variant of the TRAVELING SALESMAN PROBLEM (TSP) for which there is no known constant approximation despite numerous attempts to improve the best known algorithms for it.

TSP with triangle inequality admits a 3/2-approximation by the well know Christofides algorithm [29] when the edge weights are symmetric. However, when the edge weights can be asymmetric, TSP turns out to be much more difficult to approximate. When the edge weights can be asymmetric, TSP is called asymmetric TSP or ATSP, and even though the best known lower bound for ATSP is only $117/116 = 1.00862\ldots$ [48], there is no known constant approximation for it. A first $O(\log n)$-approximation for ATSP, where n is the number of vertices of the graph, was given by Frieze et al. [49] in 1982. Then, for almost three decades, subsequent works only improved the constant factor of the approximation ratio [50, 51, 52], until Asadpour et al. [53] slightly improved the $O(\log n)$ ratio with their randomized $O(\log n/\log\log n)$-approximation algorithm in 2010. Whether ATSP can be approximated within a constant factor remains a major open question.

We now give a reduction from a restricted version of ATSP to MIN-MOVES-ALL-PEARLS, which shows that MIN-MOVES-ALL-PEARLS is at least as hard to approximate as that restricted version, and thus likely to be at least as hard to approximate as ATSP. Let G be an instance of ATSP in which the edge weights are positive integers polynomial

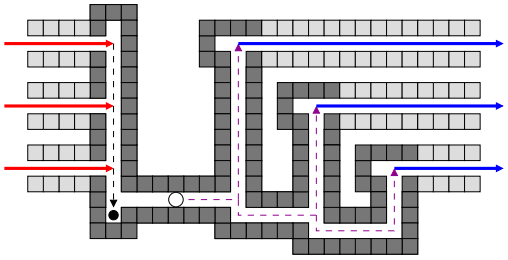


Figure 3.5. A vertex gadget in the reduction from ATSP to MIN-MOVES-ALL-PEARLS. Incoming tunnels corresponding to incoming edges are on the left, and outgoing tunnels corresponding to outgoing edges are on the right. The vertex gadget illustrated here is the gadget for the starting vertex, and the location of the droplet (shown as a white circle) is the starting location in the map. Note that after the pearl (shown as a black dot) inside the gadget is collected the droplet cannot go out, so the pearl inside this gadget must be the last one to be collected. For any other vertex, the corresponding vertex gadget is similar, with the only difference that the pearl is not inside a trap, so that after the pearl is collected the droplet can still go out through one of the outgoing tunnels.

in the number of vertices n. We reduce G to an instance of MIN-MOVES-ALL-PEARLS in polynomial time by creating a Quell map M with a vertex gadget for each vertex in G, and with a one-way tunnel connecting two vertex gadgets for each edge in G; note that the crossing of tunnels is not a problem since each move continues until an obstacle is reached.

Refer to Figure 3.5 for an illustration of how the vertex gadgets are constructed. The construction of this gadget is quite straightforward: it simply connects the tunnels for all incoming edges and all the outgoing edges, and uses a few traps to prevent the droplet from traversing any of the tunnels in the wrong direction. Fix an arbitrary vertex v of G as the starting vertex. We place the droplet in the vertex gadget for v, and place a pearl in each vertex gadget so that all gadgets must be visited to collect all the pearls; note that the vertex gadget for v is slightly different than the gadget for all other vertices to make sure that the pearl inside must be the last one to be collected. For each edge in G with weight w, we add enough turns to the corresponding tunnel in M so that it requires exactly $f(n) \cdot w$ moves to be traversed, where $f(n)$ is some function polynomial in n. Then G has a Hamiltonian cycle of weight k starting from v if and only if there is a sequence of $f(n) \cdot k$ moves for the droplet starting in the vertex gadget for v in M that collects all the pearls. Consequently,

any α-approximation for MIN-MOVES-ALL-PEARLS would give an α-approximation for the restricted version of ATSP. This concludes our proof.

3.4 Discussion

Probably, the most important open question about the optimization problems, ANY-MOVES-MAX-PEARLS and MIN-MOVES-ALL-PEARLS, is whether MIN-MOVES-ALL-PEARLS admits a constant approximation. However, we have shown that MIN-MOVES-ALL-PEARLS is likely to be at least as hard to approximate as ATSP. Whether ATSP can be approximated within a constant factor remains a major open question, so this suggests that MIN-MOVES-ALL-PEARLS may not admit a constant approximation.

There is some evidence suggesting that a constant approximation for ATSP may exist [54], but it is based on conjectures that have not been proven in a long time. Moreover, even if ATSP turns out to have a constant approximation, it would not imply that MIN-MOVES-ALL-PEARLS has one. In fact, we believe that MIN-MOVES-ALL-PEARLS may be significantly harder to approximate than ATSP. For example, the best known lower bound for ATSP is only $117/116 = 1.00862\ldots$ [48], while we obtained a stronger lower bound for MIN-MOVES-ALL-PEARLS of $2 - \epsilon$, for any fixed $\epsilon > 0$. Furthermore, MIN-MOVES-ALL-PEARLS is likely to be at least as hard to approximate as ATSP even if every pearl is associated with only one maximal segment; this is easy to prove with a small change to our vertex gadget to ensure that the pearl is always associated with a single segment.

Our lower bounds for both optimization problems hold even in simple maps. However, our reduction from ATSP from MIN-MOVES-ALL-PEARLS constructs maps that are not simple. Creating the gadgets for the vertices and the tunnels for the edges is quite straightforward, but it is not obvious how to do it when the maps are required to be simple. Perhaps in simple maps, MIN-MOVES-ALL-PEARLS is easier to approximate than in general maps, and may even admit a constant approximation.

CHAPTER 4

INTRACTABILITY IN HIGHER DIMENSIONS

In Chapter 2 we proved that ANY-MOVES-ALL-PEARLS in $\mathbb{R}^2$ can be solved in polynomial time. However, in Chapter 3 we proved that the optimization problems that we consider, ANY-MOVES-MAX-PEARLS and MIN-MOVES-ALL-PEARLS, are both APX-hard in $\mathbb{R}^2$. Thus ANY-MOVES-ALL-PEARLS is the only problem that we can solve in polynomial time in the plane, as neither one of the optimization problems can even be approximated arbitrarily well unless P = NP. For completeness, we now show that the basic decision problem ANY-MOVES-ALL-PEARLS is NP-complete in higher dimensions.

Recall that a Quell map in $\mathbb{R}^2$ is bounded by a rectilinear polygon, possibly with holes, and if the polygon has no holes we call the map simple. In other words, we say that a Quell map in $\mathbb{R}^2$ is simple if it is topologically equivalent to a disk. Similarly, we say that a Quell map in $\mathbb{R}^3$ is simple if it is topologically equivalent to a ball, and in general, in $\mathbb{R}^d$, we say that a Quell map is simple if it is topologically equivalent to a d-dimensional ball.

We prove the following theorem:

Theorem 6. ANY-MOVES-ALL-PEARLS *in* $\mathbb{R}^d$ *is NP-complete for any fixed* $d \geq 3$*, even in simple maps. Moreover, it is NP-hard in* $\mathbb{R}^3$ *even if the map has at most four layers in the third dimension, where only two layers allow movement while the other two are just boundary layers.*

We first prove that ANY-MOVES-ALL-PEARLS in $\mathbb{R}^d$ is in NP for any fixed $d \geq 2$, and then prove that ANY-MOVES-ALL-PEARLS in $\mathbb{R}^3$ is NP-hard by a reduction from 3-SAT.

4.1 Membership in NP

Observe that, due to the nature of the game, only the relative order of the coordinates of the map elements is important, and thus any map in $\mathbb{R}^2$ with n vertices, p pearls, and

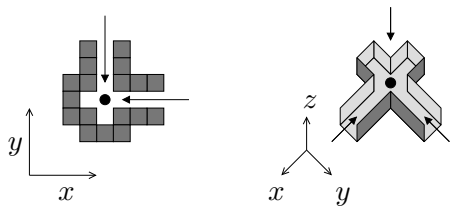
the droplet can be compressed into an $m \times m$ grid, where $m = n + p + 1$. Similarly, in $\mathbb{R}^d$, any map with n vertices, p pearls, and the droplet can be compressed into a d-dimensional cubical grid of size m with m^d cells, where $m = n + p + 1$. Therefore, since the number of cells is at most m^d and the number of maximal segments going through each cell is at most d, the number of maximal segments that can be reached by the droplet is at most $d \cdot m^d = O(m^d)$, for any fixed $d \geq 2$.

Note that in any map, if the number of maximal segments that can be reached by the droplet is x, the number of moves required to reach a segment from any location is $O(x)$, and thus the number of moves required to collect all the pearls, if they can be collected, is $O(x^2)$. Hence, since the number of maximal segments that can be reached by the droplet is $O(m^d)$, the number of moves required to collect all the pearls in any map where all the pearls can be collected is $O((m^d)^2) = O(m^{2d})$. Therefore, any optimal solution can be verified in polynomial time, so the problem is in NP.

4.2 NP-Hardness

We prove that ANY-MOVES-ALL-PEARLS in $\mathbb{R}^3$ is NP-hard in simple maps where the droplet is allowed to move in at most two layers in the third dimension by a reduction from 3-SAT. Let (V, C) be a 3-SAT instance, where $V = \{v_1, \ldots, v_n\}$ is the set of variables and $C = \{c_1, \ldots, c_m\}$ is the set of clauses. We create a Quell map with a variable gadget for each variable, including two paths for the possible assignments of the variable, true or false, a clause gadget for each clause, with a pearl inside, and a path for each literal connecting a variable path to a clause gadget.

The pearl inside each clause gadget can be collected whenever the gadget is reached, and the path for each literal branches out of the appropriate variable path and reaches the appropriate clause gadget to ensure that all the pearls can be collected if and only if the right variable paths are traversed. Denote by $l_{j,k}$ the k-th literal of c_j, $1 \leq k \leq 3$, denote by $p(l_{j,k})$ the path corresponding to $l_{j,k}$, and denote by $p(v_i)$ and $p(\bar{v}_i)$ the paths corresponding to the assignments $v_i = \text{true}$ and $v_i = \text{false}$, respectively. Then path $p(l_{j,k})$ branches out of path $p(v_i)$ (resp. $p(\bar{v}_i)$) and reaches the clause gadget corresponding to c_j if $l_{j,k}$ is a positive



(a) (b)

Figure 4.1. Basic idea for constructing the clause gadget. The pearl inside is shown as a black circle, and it can be collected by traversing any one of three paths. (a) Top view: The pearl can be collected by moving left along the x-axis, or down along the y-axis. (b) 3-dimensional view: The pearl can be collected by moving along any one of the axes. After collecting the pearl by moving along the x-axis or the y-axis, the droplet has to be moved back out of the gadget by following the same path used to reach the pearl. However, after collecting the pearl by moving along the z-axis, the droplet could be moved out of the gadget by following any one of the three paths that reach the pearl.

(resp. negative) literal of v_i.

In our construction, we call the lower layer where the droplet is allowed to move the *bottom layer*, and the upper layer where the droplet is allowed to move the *top layer*. However, we note that below this "bottom" layer and above this "top" layer there are two implicit boundary layers filled with obstacles to prevent the droplet from being moved further down or up, so the map actually has four layers even if only two allow movements.

4.2.1 Basic Ideas

Observe that in $\mathbb{R}^3$ a pearl can be collected by traversing any one of three segments, where each segment is parallel to a different axis, x, y, or z. Thus, when we construct the clause gadget we make sure that it has three paths, one for each literal, with each path reachable from a different entrance and including one of the three segments that may be traversed to collect the pearl inside. The basic idea for constructing the clause gadget is illustrated in Figure 4.1. We make sure that two of the paths reach a dead end, so after traversing any of them the droplet has to be moved back out of the gadget along the same path. However, with only two layers that allow movement we cannot make the other path reach a dead end, so after traversing it the droplet could be moved out of the gadget along

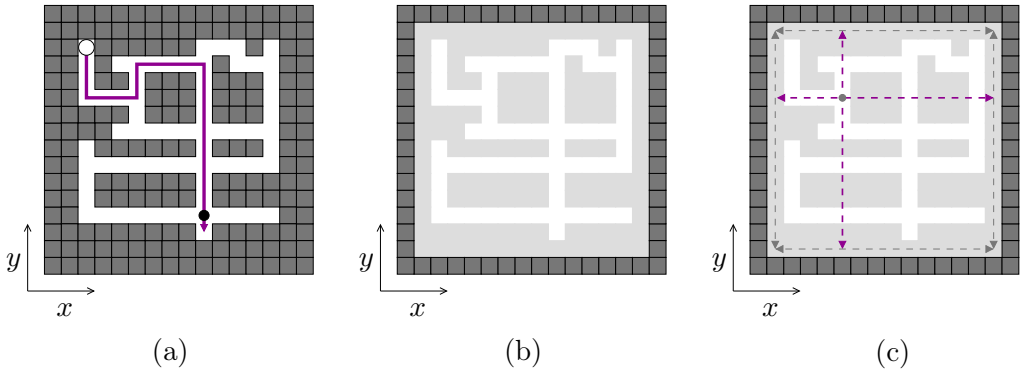


Figure 4.2. Basic idea for constructing a simple map in $\mathbb{R}^3$ including a 2-dimensional layer with holes and arbitrary paths. Pearls are shown as black circles and the starting location is shown as a white circle. (a) Bottom layer: The pearls and the starting location are in this layer, and all the pearls can be collected by following the illustrated path. (b) Top layer: It is an empty rectangle including the whole area occupied by the map in the bottom layer; obstacles in the bottom layer are shown in light gray. (c) After any move on the top layer, the droplet can no longer leave the border of the top layer and go back to the bottom layer; the place where the droplet is moved from the bottom to the top layer is marked with a dot.

any of the three paths; we will explain how to prevent this later.

Observe that a layer of a 3-dimensional map can be seen as a 2-dimensional map, and note that it is easy to create a 2-dimensional map with holes and arbitrary paths (so it is easy to complete the construction if maps are not required to be simple and the droplet is allowed to move in at least three layers inside the clause gadgets). Thus, when we create the variable gadgets and the paths for the literals we place all of them in the bottom layer of the 3-dimensional map and make sure that the droplet has to follow the paths on that layer. The basic idea for constructing a simple map in $\mathbb{R}^3$ including a 2-dimensional layer with holes and arbitrary paths is illustrated in Figure 4.2. Note that even though the bottom layer has holes, the 3-dimensional map is simple since the top layer is empty except for the border. Moreover, after any move on the top layer, the droplet can no longer leave the border of the top layer and go back to the bottom layer.

4.2.2 The Gadgets

Variable gadgets are used to ensure that a walk through the map can cover only one

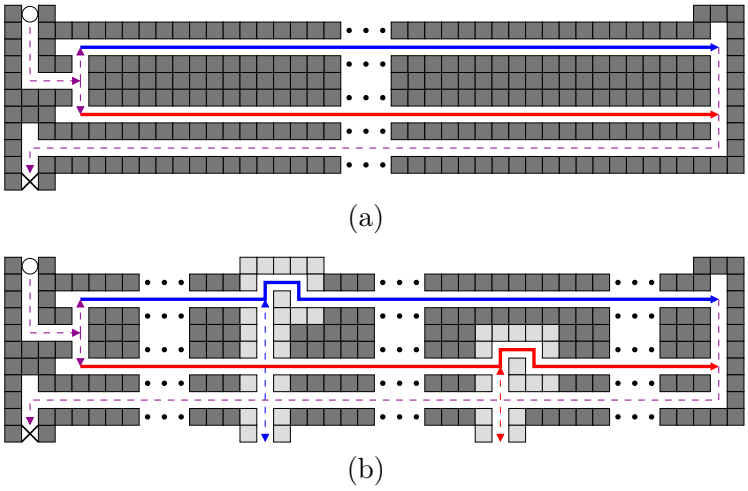


Figure 4.3. Variable gadget used for proving NP-hardness of ANY-MOVES-ALL-PEARLS in $\mathbb{R}^3$. The gadget entrance (with the droplet shown as a white circle) is on the top left corner, and the gadget exit (marked with a cross $\times$) is on the bottom left corner. (a) Basic gadget. The player can choose to follow either the blue solid line or the red solid line, but not both; the blue line on top corresponds to a true assignment, and the red line at the bottom corresponds to a false assignment. (b) Adding paths for the clause literals. When following one of the variable paths, the player can move the droplet down along any branching path for a clause literal and then move it back up to continue along the same variable path.

path from the two paths of each variable. Figure 4.3(a) illustrates the basic gadget with the paths corresponding to the possible assignments of a variable, and Figure 4.3(b) illustrates how branching paths can be added for the clause literals. Note that after choosing a path it is impossible to follow the other one, and after leaving the gadget it is impossible to go back. It is easy to see that the gadgets for several variables can be stacked, with the exit of one gadget connected to the entrance of the next gadget, allowing the player to choose paths for several variables in sequence.

When following one of the variable paths, the player can move the droplet down along any branching path for a clause literal that is connected to the variable path, and then move it back up to continue along the same variable path. Each clause gadget has three entrances and the path for each one of its clause literals is connected to a different entrance.

The clause gadget is illustrated in Figure 4.4. Note that by moving in the bottom layer it is only possible to enter the gadget using one of the entrances for the literals, and that

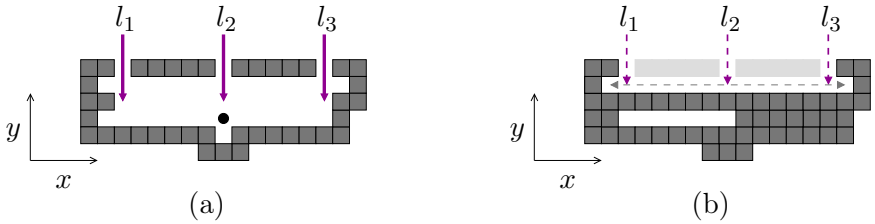


Figure 4.4. Clause gadget used for proving NP-hardness of ANY-MOVES-ALL-PEARLS in $\mathbb{R}^3$. (a) Bottom layer containing the pearl and the gadget entrances for the literals. (b) Top layer; obstacles in the bottom layer are shown in light gray. Note that it is impossible to reach the back of the gadget by moving on the top layer.

by moving in the top layer it is impossible to move past the entrances and into the back of the gadget. Moreover, observe that after reaching the gadget by moving in the top layer, a move to the side traps the droplet inside the gadget even if the droplet is moved to the bottom layer first.

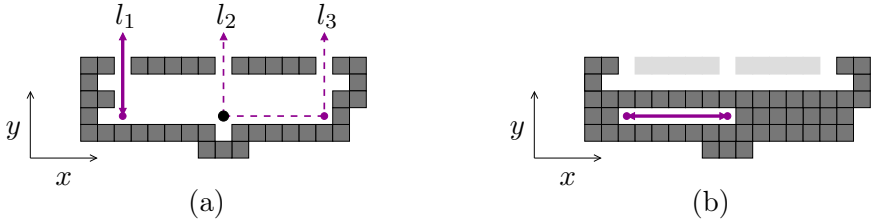


Figure 4.5. Collecting the pearl inside the clause gadget by using the entrance for the first literal. (a) Bottom layer. (b) Top layer. After collecting the pearl it is possible for the droplet to be moved out of the gadget by following any of the three paths for the literals.

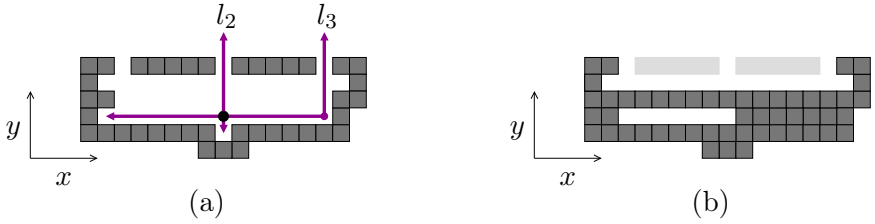


Figure 4.6. Collecting the pearl inside the clause gadget by using one of the entrances for the second or third literal. (a) Bottom layer. (b) Top layer. All of the moves are forced and they are on the bottom layer. After collecting the pearl the droplet has to be moved out of the gadget by following the same path used to get in.

Figure 4.5 shows how the pearl is collected by using the entrance for the first literal, and Figure 4.6 shows how it is collected by using one of the entrances for the second or third literal. Note that after using the entrance for the first literal, it is possible for the droplet to be moved out of the gadget by following any of the three paths for the literals. However, after using the entrance for any of the other two literals, all the moves along the path to reach the pearl inside are forced, and after reaching the end of the path the droplet has to be moved back out of the gadget by following the same path used to get in.

4.2.3 The Construction

We first note that we can assume without loss of generality that all the literals of the same clause are occurrences of different variables. Duplicates of a positive or negative literal in the same clause can always be removed, as well as clauses that have both a positive literal and a negative literal of the same variable; note that such clauses are true for any assignment of the variables. If no clauses remain after this simplification step, any assignment of the variables satisfies all the clauses and we can output a simple map with a single pearl that can always be collected. Otherwise, it is clear that there is an assignment that satisfies all the original clauses if and only if there is an assignment that satisfies all the remaining clauses, and we can just create a map for this simplified set of clauses.

Refer to Figure 4.7 for a complete example. We create a map with a main section containing the variable gadgets and the paths for the literals in the bottom layer. This main section of the map is rectangular, everything in the bottom layer outside of the gadgets is filled with obstacles, and the top layer is left empty. Variable gadgets are stacked, sorted in increasing order from top to bottom, and the clause gadgets are placed next to the border of the main section, with the paths for the literals of each clause aligned with the entrances of the corresponding clause gadget. We close the entrance of the first variable gadget, and remove the exit of the last variable gadget. The droplet starts in the bottom layer of the main section, by the entrance of the first variable gadget.

Observe that we shift right some of the corners of the paths for the literals to make sure that it is only possible to reach the clause gadgets using the columns for their entrances.

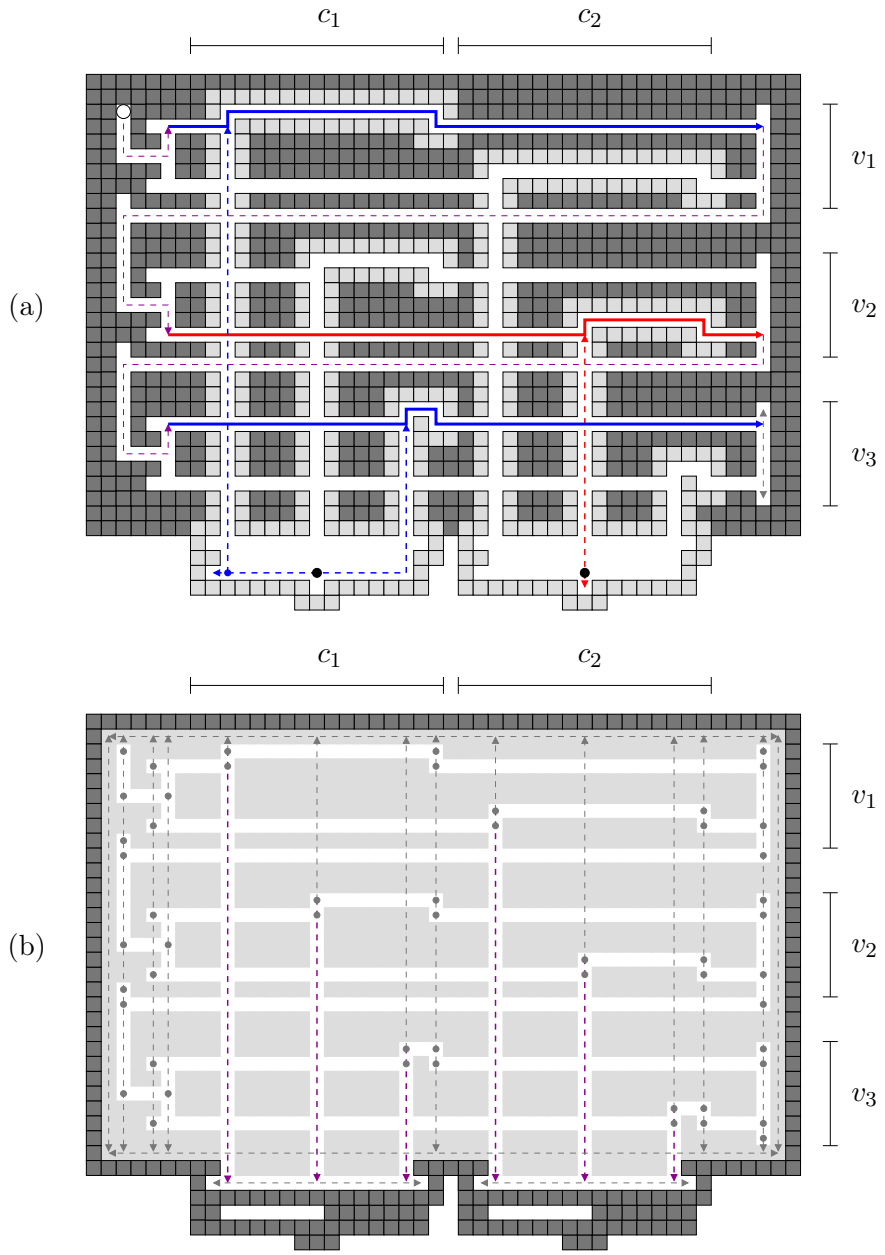


Figure 4.7. Construction for a 3-SAT instance of $n = 3$ variables and $m = 2$ clauses $c_1 = v_1 \vee v_2 \vee v_3$, and $c_2 = \bar{v}_1 \vee \bar{v}_2 \vee \bar{v}_3$, after the literals of each clause have been sorted in ascending order. (a) Bottom layer. Pearls are shown as black circles and the starting location, near the top left corner, is shown as a white circle. The paths for the literals and the clause gadgets are shown in light gray. The walk shown corresponds to the satisfying assignment $v_1 =$ true, $v_2 =$ false, and $v_3 =$ true. (b) Top layer. Obstacles in the bottom layer are shown in light gray, and places where the droplet can move to the top layer are marked with a dot.

Moreover, note that the paths for the literals of each clause are sorted from left to right in increasing order of variable index. This is important, as it prevents the droplet from being able to go back to a previously visited variable gadget; recall that this is possible by using the entry for the first literal of a clause gadget and then the exit for one of the other literals.

This completes the construction. The reduction runs in polynomial time since the number of obstacles required is polynomial in the number of variables and clauses. The main section of the map with the variable gadgets and the paths for literals is clearly simple, and so are the clause gadgets. It is thus easy to see that the complete map is simple since each clause gadget is connected to the rest of the map by a single hole.

We have the following lemma:

Lemma 5. *There is an assignment of V that satisfies all the clauses of C if and only if all the pearls in the Quell map can be collected using any number of moves.*

Proof. We first prove the direct implication. Suppose there is an assignment of V that satisfies all the clauses of C. There is a corresponding walk through the map that covers path $p(v_i)$ if v_i is set to true and covers path $p(\bar{v}_i)$ if v_i is set to false, and that follows all the paths for clause literals reached while traversing the chosen variable paths, and collects the pearl inside every clause gadget that is reached.

Consider any clause c_j satisfied by the assignment of variable v_i. If v_i is set to true then c_j has a positive literal of v_i, so path $p(l_{j,k})$ branches out of path $p(v_i)$ and reaches the clause gadget corresponding to c_j. Similarly, if v_i is set to false then c_j has a negative literal of v_i, so path $p(l_{j,k})$ branches out of path $p(\bar{v}_i)$ and reaches the clause gadget corresponding to c_j. In any case, if a clause is satisfied by the assignment, the corresponding clause gadget is visited by the walk. Since all the clauses are satisfied, the walk visits all the clause gadgets. Thus all the pearls can be collected.

We next prove the reverse implication. Suppose that all the pearls in the Quell map can be collected using any number of moves. Observe in Figure 4.7(b) that a clause gadget can only be reached by moving down along one of the columns including an entrance of a clause gadget, and that those columns can only be reached by moving in the bottom layer

of the main section of the map. Moreover, observe that reaching a clause gadget by moving along one of those columns in the top layer is useless, because after reaching the gadget the only useful move is to move back to the bottom layer. Hence we can assume that a walk that collects all the pearls only has to move on the bottom layer of the main section of the map.

The variable gadgets ensure that only one of the two paths of each variable is chosen. Then, since all the literals of the same clause are occurrences of different variables, a clause gadget is never connected to both paths of the same variable. Moreover, since the paths for the literals of each clause are sorted in increasing order of variable index, by using the entry for the first literal of a clause and then the exit for the second or the third literal, it is impossible to go back to a previous variable gadget. Hence, any walk through the map can only visit one of the two paths for each variable, and for every walk that visits all the variable gadgets there is a corresponding assignment that makes v_i = true if path $p(v_i)$ is traversed, and makes v_i = false if path $p(\bar{v}_i)$ is traversed.

Consider any clause gadget for a clause c_j visited by the walk using a path for a literal $l_{j,k}$ of variable v_i. If $p(l_{j,k})$ branches out of path $p(v_i)$ then c_j has a positive literal of v_i, and the walk must have followed path $p(v_i)$ to reach the clause gadget for c_j. Similarly, if $p(l_{j,k})$ branches out of path $p(\bar{v}_i)$ then c_j has a negative literal of v_i, and the walk must have followed path $p(\bar{v}_i)$ to reach the clause gadget for c_j. In any case, if a clause gadget is visited by the walk, the corresponding clause is satisfied by the assignment of the variable. Since the walk collects all the pearls, the walk visits all the clause gadgets. Thus there is an assignment of V that satisfies all the clauses of C. □

This concludes the proof of Theorem 6.

4.3 Discussion

We have shown that ANY-MOVES-ALL-PEARLS, which can be solved in polynomial time in $\mathbb{R}^2$, is NP-complete in higher dimensions. It is not hard to see that a simpler reduction from the same problem, 3-SAT, can be used to prove NP-hardness in $\mathbb{R}^3$ if the maps are

not required to be simple and the number of layers that allow movement is unbounded. Our proof is interesting because it shows that ANY-MOVES-ALL-PEARLS in $\mathbb{R}^3$ remains NP-hard even in simple maps with a minimum of two layers that allow movement.

CHAPTER 5

INTRACTABILITY WITH PUSHABLE BLOCKS

In Chapter 2 we proved that ANY-MOVES-ALL-PEARLS in $\mathbb{R}^2$ can be solved in polynomial time. However, this is true when the droplet is the only moving object in the map, and there are no special objects other than the obstacles and the pearls. It is easy to show that ANY-MOVES-ALL-PEARLS with pushable blocks is NP-hard by a simple reduction from HAMILTONIAN CYCLE in planar cubic graphs [27], using blocks to make sure that edges can only be traversed once. In this chapter, we strengthen that NP-hardness result by proving the following theorem:

Theorem 7. ANY-MOVES-ALL-PEARLS *with pushable blocks is PSPACE-complete.*

We fist prove that ANY-MOVES-ALL-PEARLS with pushable blocks is in PSPACE using Savitch's Theorem [55], and then prove that it is PSPACE-hard by a reduction from a restricted version of a graph problem called NONDETERMINISTIC CONSTRAINT LOGIC [10, 17]. After showing membership in PSPACE, we describe this problem and the restricted version that we use before giving our reduction.

5.1 The Complexity Class PSPACE

PSPACE is the class of decision problems that can be solved by deterministic Turing machines using polynomial space and unlimited amount of time. Every algorithm uses at least as much time as space, so P is clearly contained in PSPACE. Moreover, Savitch's Theorem states that PSPACE and its nondeterministic analog NPSPACE are identical [55], so NP is also contained in PSPACE. Essentially, nondeterminism does not help when measuring space since space can be reused and a nondeterministic Turing machine can be simulated by a deterministic Turing machine using exponentially more time, but about the same amount

of space. However, it is conjectured that there are problems in PSPACE that are not in NP, and that PSPACE-complete problems are strictly harder than NP-complete problems.

A problem X is PSPACE-complete if it is in PSPACE and every other problem in PSPACE is polynomial-time reducible to X. Note that even though we are considering space complexity, the reductions must be polynomial-time reductions, and not just polynomial-space reductions. It is well known that NP-complete problems cannot be solved in polynomial-time unless P = NP. A PSPACE-complete problem cannot be solved in polynomial-time, and a solution to such a problem cannot even be verified in polynomial-time, unless NP = PSPACE. In fact, there is no known way to specify solutions to some PSPACE-complete problems succinctly, so it is possible that optimal solutions can have length superpolynomial in the size of the input.

The canonical PSPACE-complete problem is the Quantified Boolean Formula problem (QBF) [56], which is a generalization of SAT with existential ($\exists$) and universal ($\forall$) quantifiers. QBF is the problem of deciding if a fully quantified boolean formula, that is, a formula where each variable is within the scope of a quantifier, is true. For example, the formula $\phi = \forall x\ \exists y\ [(x \vee \bar{y}) \wedge (\bar{x} \vee y)]$ is a fully quantified boolean formula which is true. Note that the order of the quantifiers is important, as ϕ would be false if the order of $\forall x$ and $\exists y$ were reversed. If all the quantifiers appear at the beginning of the formula and the scope of each quantifier is everything following it, the formula is said to be in *prenex normal form*.

Any quantified boolean formula can be put in prenex normal form with alternating existential and universal quantifiers. Thus, QBF has a natural interpretation as a two-player game where the first (resp. second) player a selects the values of the variables bounded to $\exists$ (resp. $\forall$) quantifiers, and wins if after all the values have been selected, the remaining formula is true (resp. false).

A game is said to have (polynomially) bounded-length if the maximum number of moves before the game ends is bounded (by a polynomial of the input size), and it is said to have unbounded-length otherwise. Many bounded-length two-player games are PSPACE-complete, and many unbounded-length two-player games are even harder. Also, many

bounded-length puzzles (one-player games) are NP-complete, and many unbounded-length puzzles are PSPACE-complete. For example, Sokoban and Rush Hour are unbounded-length puzzles and PSPACE-complete [18, 23].

Note that Theorem 7 is not obvious, but intuitively it seems possible because ANY-MOVES-ALL-PEARLS with pushable blocks is an unbounded-length puzzle. We know that ANY-MOVES-ALL-PEARLS without special objects is in P and it is easy to show that optimal solutions always require $O(n^2)$ moves. However, optimal solutions for ANY-MOVES-ALL-PEARLS with pushable blocks could be longer, since blocks may have to be pushed back and forth multiple times, and the problem may not even be in NP.

5.2 Membership in PSPACE

Savitch's Theorem states that PSPACE = NPSPACE [55], so in order to prove that ANY-MOVES-ALL-PEARLS with pushable blocks is in PSPACE we just have to prove that it can be solved by a nondeterministic Turing machine that uses polynomial space.

Observe that, due to the nature of the game, only the relative order of the coordinates of the map elements is important. Therefore, any map with n vertices, p pearls, b pushable blocks, and the droplet can be compressed into an $m \times m$ grid where $m = n + p + b + 1$, and where each cell can be empty or it can contain an obstacle, a pearl, a pushable block, or the droplet. Hence, the grid has at most 5^{m^2} distinct configurations, and if all the pearls can be collected, they can be collected using at most 5^{m^2} moves.

Consider a nondeterministic Turing machine that chooses a move nondeterministically at each step, and also counts the number of moves so that it can stop after 5^{m^2} moves have been made without finding a solution. Since 5^{m^2} can be stored using $O(m^2)$ bits, which is polynomial in the size of the input, the space used by this Turing machine is polynomial and thus the problem is in PSPACE.

5.3 A Restricted Constraint Logic Problem

PSPACE-hardness results for multiple games have been obtained by direct reductions from QBF, but the geometric constraints of many games do not have a natural correspon-

dence with the properties of formula problems. In [10, 17], Hearn and Demaine described several variants of Constraint Logic, which can be thought of as games played on graphs, and thus are more amenable to reductions to real games than formula problems. One of those variants, which they used to prove or strengthen previous PSPACE-hardness results for several unbounded-length puzzles such as Sliding Blocks, Sokoban, and Rush Hour, is called Nondeterministic Constraint Logic (NCL). We now describe this variant and the restricted version that we use in our reduction.

5.3.1 The Original NCL Problem

A *constraint graph* is a directed graph obtained by assigning a direction to every edge of a simple undirected graph, where each edge has a weight and each vertex has a non-negative minimum inflow. The *inflow* at each vertex is the sum of the weights of the inward-directed edges, and a legal configuration of the graph has an inflow of at least the minimum inflow at each vertex. A legal move on a constraint graph is the *reversal* of an edge orientation that results in another legal configuration.

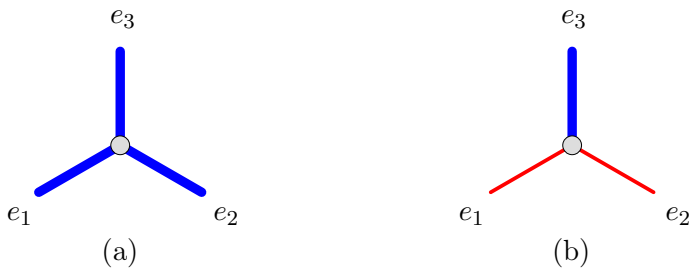


Figure 5.1. Basic NCL vertices. Thin red edges have a weight of 1, thick blue edges have a weight of 2, and the vertices have a minimum inflow constraint of 2. (a) OR vertex: edge e_3 may be directed outward if and only if either edge e_1 or edge e_2 is directed inward. (b) AND vertex: edge e_3 may be directed outward if and only if both edges e_1 and e_2 are directed inward.

NCL is the following decision problem: given a constraint graph G and a goal edge e in G, is there a sequence of legal moves on G that eventually reverses e? It is PSPACE-complete even on planar graphs that use only the types of vertices shown in Figure 5.1 [10, 17]. Note

how those vertices behave similar to logical OR and AND gates if we consider e_1 and e_2 to be the *input edges* and e_3 to be the *output edge*.

5.3.2 Our Restricted NCL Problem

Since NCL is PSPACE-complete on planar graphs that use only the types of vertices shown in Figure 5.1, it is often possible to prove that other problems are PSPACE-hard by reductions from NCL that use only two types of gadgets: one for the OR vertices and one for the AND vertices. However, for some problems it is difficult to construct simple gadgets even for those types of vertices; especially for the AND vertices, which are more complex than the OR vertices. Hence, we impose additional restrictions on the vertices, that allow us to create simpler gadgets.

Suppose that the incident edges of each vertex can be locked to the vertex or they can be unlocked, and when they are *locked* they cannot be reversed until they are *unlocked*. If the edges can be locked at any time, it is easy to show that NCL remains PSPACE-complete when using special types of vertices that require some of its incident edges to be locked before another incident edge e_i can be reversed, even if the locked edges cannot be unlocked until after e_i is reversed, since we can always wait to lock the required edges until we are ready to reverse e_i. Moreover, note that since we can always wait to lock the required edges, the order in which they have to be locked before we can reversed e_i is irrelevant.

We show that NCL remains PSPACE-complete on graphs that use only the basic type of OR vertex shown in Figure 5.1 and a special type of AND vertex, which we call an AND* vertex, that require both input edges to be locked before its output edge can be directed outward. The minimum inflow of an AND* vertex and the weights of its input edges and its output edge are the same as the minimum inflow of an AND vertex and the weights of its input edges and its output edge. Thus, just like in the case of an AND vertex, the output edge of an AND* vertex may be directed outward if and only if both of its input edges are directed inward. However, an AND* vertex has the following additional restrictions on when the states of its edges may be changed (refer to Figure 5.1(b)):

(a) e_1 may be locked if and only if it is directed inward;

(b) e_2 may be locked if and only if it is directed inward and e_1 is locked;

(c) e_1 and e_2 may be unlocked at the same time if and only if they are both locked and e_3 is directed inward;

(d) e_3 may be directed outward if and only if both e_1 and e_2 are directed inward and are locked.

Lemma 6. NCL *is PSPACE-complete even on planar graphs that use only* OR *and* AND* *vertices.*

Proof. It is easy to show that this restricted version of NCL is also in PSPACE since it can be solved by nondeterministically choosing the best move at each step and only maintaining the current state of the graph; note that the additional space required for the locks is only linear. Let (G, e) be an NCL instance, where G is a constraint graph that uses only the types of vertices shown in Figure 5.1, and e is the goal edge. We create an NCL instance (G', e), where G' is obtained by replacing every basic AND vertex in G with a special AND* vertex. Then there is a sequence of edge reversals on G that reverses e if and only if there is a sequence of edge reversals on G' that reverses e.

Suppose there is a sequence of edge reversals on G that reverses e. The same sequence of reversals can be done on G' if we wait to lock the input edges of any AND* vertex until its output edge is to be directed outward; then we can lock e_1, lock e_2, and reverse e_3, in that order. Note that we can always unlock e_1 and e_2 when e_3 is directed inward, but they are always locked whenever e_3 is directed outward. This, however, is not a problem because neither e_1 nor e_2 may be directed outward when e_3 is directed outward, as that would violate the vertex's inflow constraint.

Suppose there is a sequence of edge reversals on G' that reverses e. The same sequence of reversals can be done on G, since locking and edge on G' can only restrict the reversals that can be done at any given time. Thus any reversal that can be done on G' can also be done on G when corresponding edges in G' and G have the same orientations. □

5.4 PSPACE-Hardness

We prove that ANY-MOVES-ALL-PEARLS with pushable blocks is PSPACE-hard by a reduction from NCL on graphs that use only OR and AND* vertices. Let (G, e) be an NCL instance, where G is a constraint graph that uses only OR and AND* vertices, and e is the goal edge. We create a Quell map with pushable blocks and with a single pearl that can be collected if and only if there is a sequence of moves on G that eventually reverses e.

To create the Quell map we need to construct gadgets for the NCL vertices, and we need to explain how those gadgets are laid out and how they are connected so that they can be visited by the droplet. First, we describe some basic gadgets, then we explain how several of those basic gadgets are connected to construct the vertex gadgets, and finally we explain the overall construction.

5.4.1 Basic Gadgets

We use two basic gadgets: an edge gadget and a lock gadget. The *edge* gadget is used to encode the orientation of an edge, and the *lock* gadget is used to lock the input edges of an AND* vertex and make sure that its output edge may be directed outward if and only if both input edges are directed inward. We create an edge gadget for each edge in G and a lock gadget for each AND* vertex in G. No additional gadgets are required for the OR vertices.

Each basic gadget has one pushable block and several possible states, and its state at any given time is determined by the location of its pushable block. Moreover, each basic gadget has multiple entries and exits which can be connected to other parts of the map using tunnels. Those tunnels allow the player to move the droplet between gadgets, and we use *one-way traps* to make sure that it is only possible for the droplet to enter a gadget using an *entry* and leave a gadget using an *exit*.

We say that the droplet *visits* a basic gadget when it enters the gadget, and that the visit is complete when it leaves the gadget. Depending on the entry used, the droplet may leave without changing the gadget's state, or it may change the gadget's state by moving its pushable block.

When the droplet visits a basic gadget, the entry used determines if the state of the gadget is changed, and the state of the gadget before the droplet leaves determines the exit that is used. Later, when we explain how the basic gadgets are connected, we will see how most of the entries that can be used to visit a gadget depend on the states of other gadgets. This allows us to change the state of a gadget only when other gadgets are in certain states.

Edge Gadget. This gadget is used to encode the orientation of an edge. Figure 5.2 illustrates how an edge gadget is constructed. It has two possible states and two possible locations of the block, X and Z, corresponding to the possible orientations of the edge. Depending on the orientation of the edge, the block is initially place at X or Z. The edge is *reversed* by moving the block from X to Z or from Z to X.

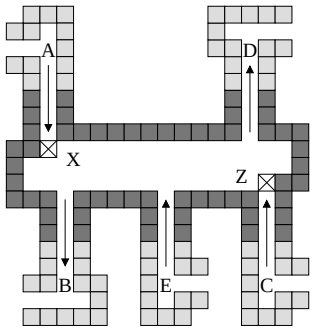


Figure 5.2. Edge gadget. The two white squares with a cross (×) represent the locations of the pushable block, X and Z, corresponding to the possible orientations of the edge. The arrows indicate possible entries (A, C, and E) and exits (B and D). Observe how one-way traps, shown in light gray, are used to make sure that it is only possible for the droplet to enter the gadget using an entry and leave the gadget using an exit.

We call entries A and C *vertex entries*, exits B and D *vertex exits*, and entry E the *edge entry*. Consider an edge $e_i = (u, v)$ and its corresponding edge gadget, as shown in Figure 5.2. Suppose e_i is directed toward u when its block is located at X, and it is directed toward v when its block is located at Z. When we construct the vertex gadgets for u and v, entry A and exit B are used for u's vertex gadget, while entry C and exit D are used for v's vertex gadget. Later, when we describe the vertex gadgets, we will see how each vertex entry can only be used when other edges incident to u or v are in certain states, and how this allows us to enforce the NCL vertex constraints. Moreover, when we describe the

overall construction, we will see how the edge entry, which is independent of u and v, can be used to visit the edge gadget at any time.

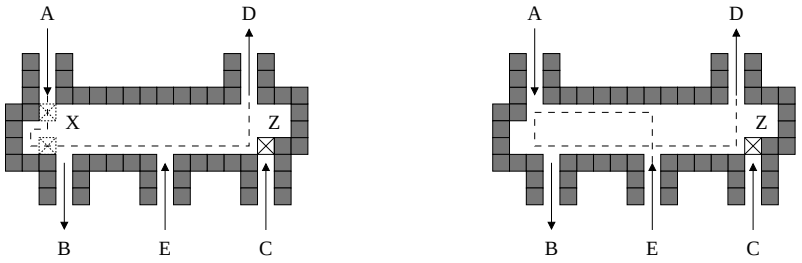


Figure 5.3. How the edge gadget works. (a) Reversing an edge by moving its pushable block from X to Z; the squares with dotted lines indicate the intermediate locations of the block before it is finally moved to Z. (b) Visiting an edge gadget without moving its pushable block, using the edge entry and vertex exit D.

Figure 5.3 illustrates how the edge gadget works. Vertex entries A and C can be used to move the pushable block and change the orientation of the edge, while vertex exits B and D can be used if the pushable block is located at X or Z, respectively. Edge entry E can be used to visit the edge without moving the pushable block.

We have the following proposition:

Proposition 1. *The edge gadget shown in Figure 5.2 satisfies the following properties:*

(a) If the block is located at X (resp. Z), it can only be moved by entering the gadget using entry A (resp. C), and it must be moved to Z (resp. X) to be able to leave using exit D (resp. B).

(b) If the block is located at X (resp. Z) and it is not moved when entering the gadget, it cannot be moved before leaving the gadget and it is only possible to leave using exit B (resp. D).

(c) Whenever the droplet is outside the gadget, the block can only be located at X or Z.

Proof. Figure 5.3(a) provides a proof of property (a): it is easy to see that if the block is located at X, it can only be moved down by entering the gadget using entry A, and that after it is moved down, it must be moved right to Z to be able to leave using exit D; the

other case is symmetric. Figure 5.3(b) provides a proof of property (b): it is easy to see that if the block is located at Z, it cannot be moved when the droplet is inside the gadget, and that once the droplet is inside, it is only possible to leave using exit D; the other case is symmetric. Property (c) follows from properties (a) and (b), since the block is initially placed at X or Z. □

These properties are important for constructing the vertex gadgets. For example, consider a vertex v with two incident edges e_i and e_j, and the edge gadget corresponding to e_i, as shown in Figure 5.2. Suppose that e_j may be directed outward from v when e_i is directed inward toward v, and that and e_i is directed inward toward v when its block is located at Z. Then, we can make sure that e_j can be directed outward by leaving e_i's edge gadget using vertex exit D, which is only possible if its block is located at Z.

Lock Gadget. This gadget is used to lock the input edges of an AND* vertex and make sure that its output edge may be directed outward if and only if both input edges are directed inward. Figure 5.4 illustrates how a lock gadget is constructed. It has three possible states and three possible locations of the block, X, Y, and Z. Depending on the orientation of the vertex's output edge, the block is initially place at X or Z. The location of the block determines which input edges are locked.

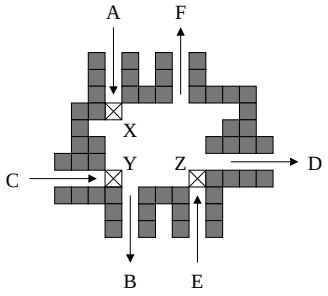


Figure 5.4. Lock gadget. The three white squares with a cross (×) represent the possible locations of the pushable block, X, Y, and Z, corresponding to the possible states of the gadget. The arrows indicate possible entries (A, C, and E) and exits (B, D, and F). One-way traps (not shown) are used, like they are used for the edge gadget, to make sure that it is only possible for the droplet to enter the gadget using an entry and leave the gadget using an exit.

We call entries A and C *input entries*, exits B and D *input exits*, entry E the *output entry*, and exit F the *output exit*. Consider an AND* vertex v and its corresponding lock gadget, as shown in Figure 5.4. When we construct the vertex gadget for v, the input entries and exits of the lock gadget are connected to the edge gadgets for v's input edges, using one input entry of the lock gadget for each input edge of v, while the output entry and exit of the lock gadget are connected to the edge gadget for v's output edge.

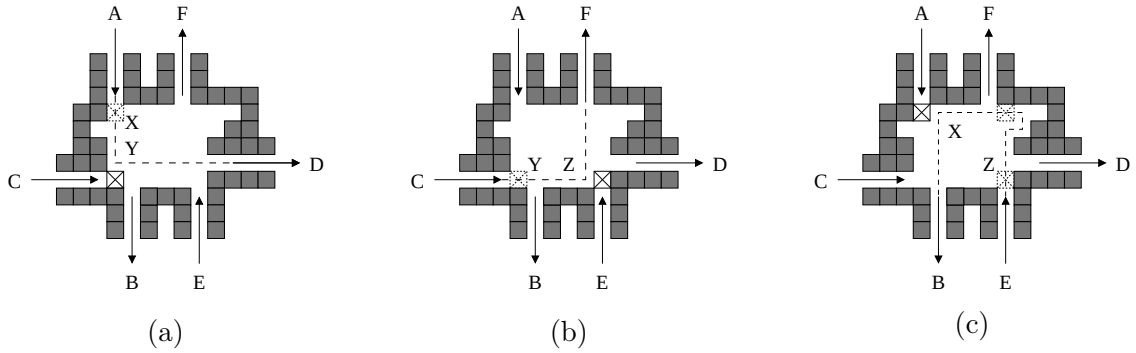


Figure 5.5. Changing the state of a lock gadget by moving its pushable block. The squares with dotted lines indicate the intermediate locations of the pushable block before it is moved to its final location in each figure. (a) Moving the block from X to Y by using input entry A. (b) Moving the block from Y to Z by using input entry C. (c) Moving the block from Z to X by using the output entry.

Figures 5.5 and 5.6 illustrate how the lock gadget works. Entries A, C, and E can be used to move the pushable block and change the state of the gadget, while exits B, D, and F can be used if the pushable block is located at X, Y, or Z, respectively. Note that the lock gadget is similar to the edge gadget, with corresponding X and Z locations. However, moving the block of a lock gadget from X to Z requires two visits instead of one: one to move the block from X to Y, and another one to move the block from Y to Z.

We have the following proposition:

Proposition 2. *The lock gadget shown in Figure 5.4 satisfies the following properties:*

(a) If the block is located at X, Y, or Z, it can only be moved by entering the gadget using the right entry: it can be moved from X to Y by using entry A, from Y to Z by using entry C, and from Z to X by using entry E. Moreover, once it is moved from X to Y it

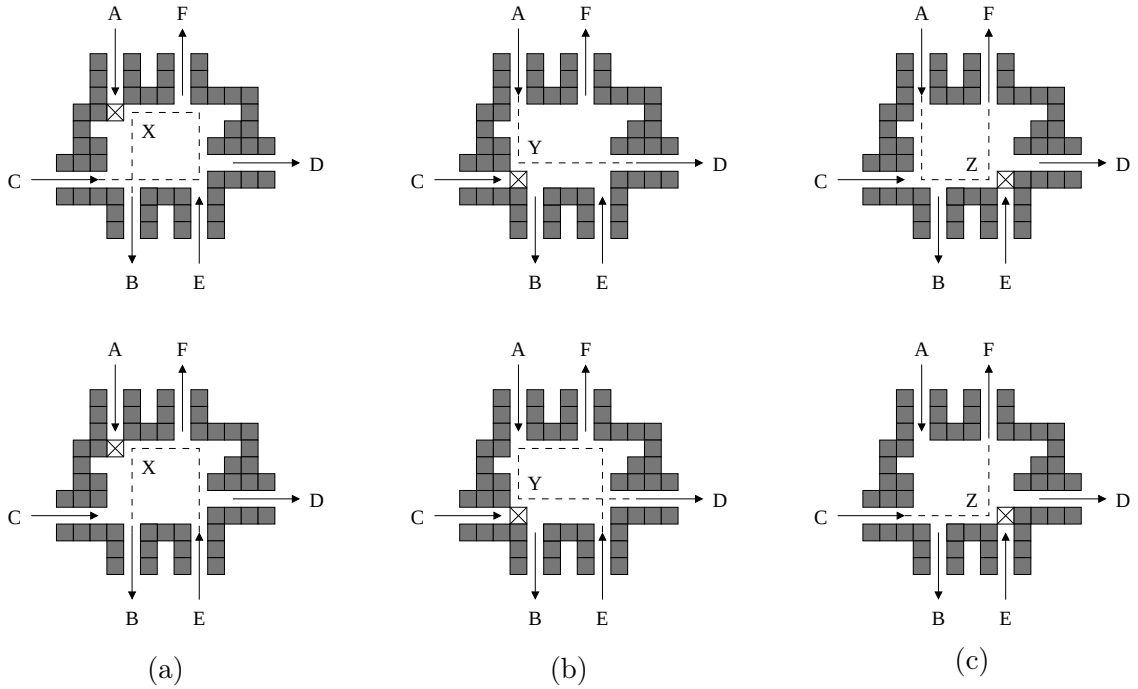


Figure 5.6. Visiting a lock gadget without moving its pushable block. (a) Input exit B is always used if the block is located at X. (b) Input exit D is always used if the block is located at Y. (c) The output exit is always used if the block is located at Z.

is only possible to leave using exit D, once it is moved from Y to Z it is only possible to leave using exit F, and once it is moved from Z to X it is only possible to leave using exit B.

(b) If the block is located at X, Y, or Z, and it is not moved when entering the gadget, it cannot be moved before leaving the gadget. Moreover, if the block is located at X it is only possible to leave using exit B, if the block is located at Y it is only possible to leave using exit D, and if the block is located at Z it is only possible to leave using exit F.

(c) Whenever the droplet is outside the gadget, the block can only be located at X, Y, or Z.

Proof. Figure 5.5 provides a proof of property (a), while Figure 5.6 provides a proof of property (b); note how in each case, once the droplet is inside the gadget, all the movements are forced. Property (c) follows from properties (a) and (b), since the block is initially placed at X or Z. □

These properties are important for constructing the AND* vertex gadgets. For example, consider an AND* vertex v and its corresponding lock gadget, as shown in Figure 5.4. Suppose that the output edge is directed inward toward v, and the lock gadget's block is located at X. To ensure that the output edge can only be directed outward from v if both input edges are directed inward toward v, we make sure that the input entries A and C can only be used to move the block to Z when the input edges are directed inward, using one input entry for each input edge, and we make sure that the output edge can only be directed outward by leaving the lock gadget using exit F, which is only possible if its block is located at Z.

5.4.2 Vertex Gadgets

We construct the NCL vertex gadgets by connecting the entries and exits of several basic gadgets with tunnels. For each vertex we use the edge gadgets for its three incident edges, and for each AND* vertex we use an additional lock gadget. Recall that the vertex entries and exits on each side of an edge gadget are used for a different vertex, so each edge gadget is used for two vertex gadgets.

We say that the droplet *visits* a vertex gadget when it enters any of its tunnels, and that the visit is complete when it enters a tunnel for another vertex gadget or when it leaves to return to the starting location. Later, when we describe the construction, we will see how the edge entry of every edge gadget can be reached from the starting location, and how the starting location can be reached from every exit of a basic gadget, allowing the droplet to visit any vertex gadget at any time.

The state of a vertex gadget is changed when the state of any of its basic gadgets is changed. In each vertex, we assume that the initial state satisfies the vertex constraints. Then any possible change of state, including edge reversals and change of lock states in the case of AND* vertices, maintains those constraints.

We note that any edge directed inward toward v can only be reversed by visiting v, so its orientation is always changed from inward to outward. Moreover, to reverse any edge directed inward toward v, we must visit v by first visiting one of its other incident edges.

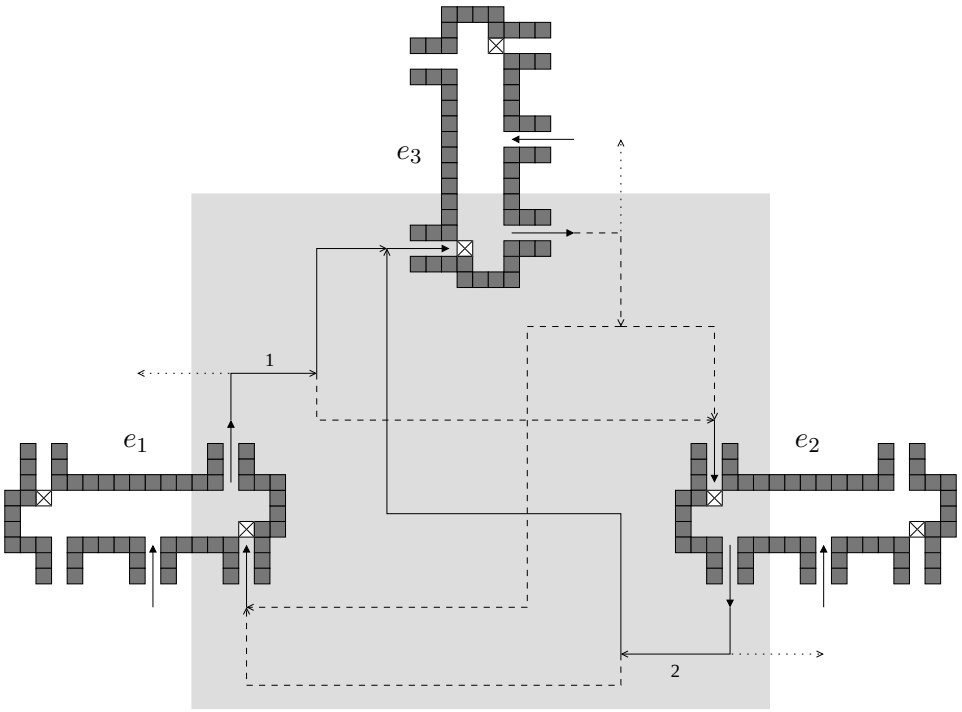


Figure 5.7. OR vertex gadget. Input edge e_1 is directed inward when its block is on the right, input edge e_2 is directed inward when its block is on the left, and output edge e_3 is directed inward when its block is at the bottom. The dashed and solid lines indicate how the entries and exits of the edge gadgets are connected by tunnels, and the arrows indicate the directions in which the tunnels can be traversed. The dotted lines going out of the gray rectangle indicate paths that go back to the starting location. Either one of the paths indicated by solid lines may be used to direct the output edge outward, while other paths indicated by dashed lines may be used to direct the input edges outward.

OR Vertex Gadget. This is the simplest of the two vertex gadgets, so we describe it first. It is symmetric and it does not require any special gadget other than the edge gadgets.

Figure 5.7 illustrates how the OR vertex gadget is constructed. The connections required by the gadget are shown inside the light gray rectangle using dashed and solid lines. Basically, the vertex exit of each edge gadget is connected to the vertex entry of the other two edge gadgets. The dotted lines going out of the gray rectangle indicate paths that go back to the starting location, which is outside of all the gadgets (this will be explained later), and no other paths are connected to the vertex entries and exits, or the tunnels inside the gray rectangle.

We have the following lemma:

Lemma 7. *The gadget shown in Figure 5.7 satisfies the same constraints as an NCL* OR *vertex.*

Proof. We need to show that e_3 may be directed outward if and only if either e_1 or e_2 is directed inward. By Proposition 1, it is easy to see that e_3 may be directed outward by moving its block up if and only if e_1 is directed inward with its block on the right or e_2 is directed inward with its block on the left. For example, e_3 may be directed outward if e_1 is directed inward, by following the path indicated by solid line 1, or it may be directed outward if e_2 is directed inward, by following the path indicated by solid line 2. In fact, since the vertex is symmetric, it is easy to see that any edge may be directed outward if and only if any other edge is directed inward. □

AND* Vertex Gadget. This is the most complex of the two vertex gadgets. Apart from the edge gadgets, it uses a lock gadget to lock the input edges and make sure that its output edge may be directed outward if and only if both input edges are directed inward.

Figure 5.8 illustrates how an AND* vertex gadget is constructed. The connections required by the gadget are shown inside the light gray rectangle using dashed and solid lines. The vertex exit of e_1 is connected to input entry A of the lock gadget, and the vertex exit of e_2 is connected to input entry C of the lock gadget. Input exit B of the lock gadget is connected to the vertex entries of both e_1 and e_2, and input exit D of the lock gadget is connected to the vertex entry of e_2 only. The output exit F of the lock gadget is connected to the vertex entry of e_3, and the vertex exit of e_3 is connected to the output entry E of the lock gadget. The dotted lines going out of the gray rectangle indicate paths that go back to the starting location, which is outside of all the gadgets (this will be explained later), and no other paths are connected to the vertex entries and exits, or the tunnels inside the gray rectangle.

Note that all the connections are between an edge gadget and the lock gadget (no edge gadget is directly connected to another edge gadget). Thus in order to reverse any of the

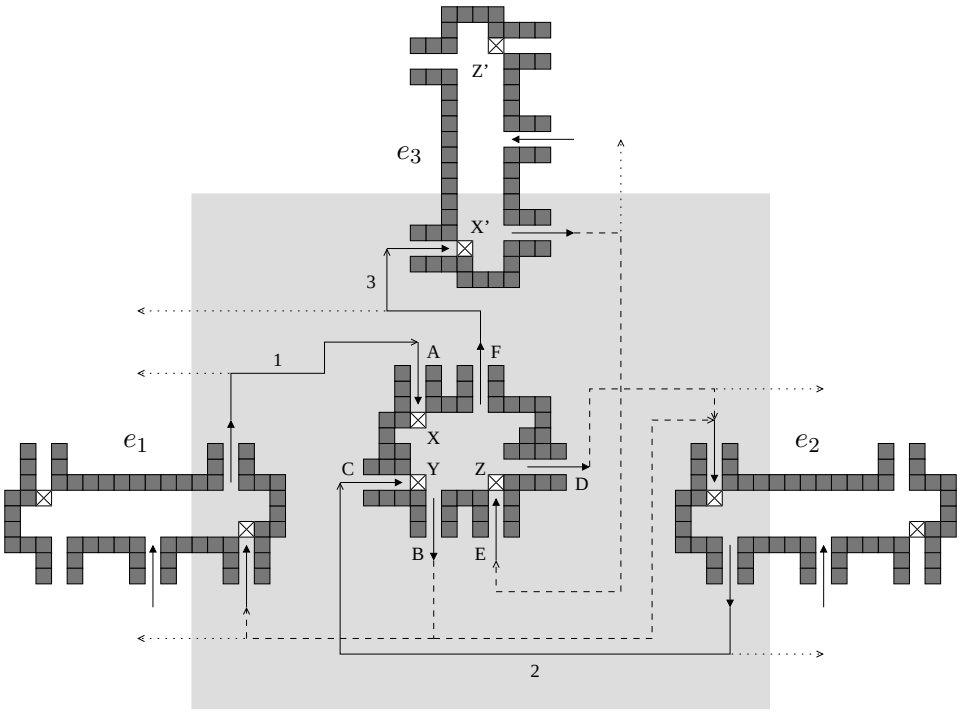


Figure 5.8. AND* vertex gadget. Input edge e_1 is directed inward when its block is on the right, input edge e_2 is directed inward when its block is on the left, and output edge e_3 is directed inward when its block is at the bottom. The dashed and solid lines indicate how the entries and exits of the edge gadgets and the lock gadget are connected by tunnels, and the arrows indicate the directions in which the tunnels can be traversed. The dotted lines going out of the gray rectangle indicate paths that go back to the starting location. The paths indicated by solid lines may be used to lock the input edges and direct the output edge outward, while other paths indicated by dashed lines may be used to unlock the input edges and direct them outward.

edges the droplet must first visit the lock gadget. Furthermore, note that each entry of the lock gadget can be reached from a different edge gadget, and only when the corresponding edge is directed inward. Hence, the edges that can be reversed at any given time depend on both the orientation of the other edges and the state of the lock gadget.

The state of the lock gadget determines which input edges are locked. By moving its block we can cycle between none, one, or two input edges locked:

- if the block is located at X, both e_1 are e_2 are unlocked;
- if the block is located at Y, e_1 is locked and e_2 is unlocked;
- if the block is located at Z, both e_1 and e_2 are locked.

By Propositions 1 and 2, it is easy to check that the locks work and each input edge may be directed outward if and only if it is unlocked. For example, when the lock gadget's block is located at X, both e_1 and e_2 are unlocked, and any one of them may be directed outward by visiting the lock gadget and leaving using input exit B. Also, when the block is located at Y, only e_1 is locked, and only e_2 may be directed outward by visiting the lock gadget and leaving using input exit D. And finally, when the block is located at Z, both e_1 and e_2 are locked, and none of them may be directed outward by visiting the lock gadget and leaving using the output exit F.

Initially, if e_3 is directed inward with its block located at X$'$, the lock gadget's block is placed at X, and if e_3 is directed outward with its block located at Z$'$, the lock gadget's block is placed at Z. Observe that in any case the initial lock states of the input edges are valid, since they only start locked when the lock gadget's block is placed at Z, and in that case they must be directed inward since e_3 is directed outward.

Lemma 8. *The gadget shown in Figure 5.8 satisfies the same constraints as an NCL* AND* *vertex.*

Proof. We first show that the input edges can only be locked at the right times (we already know that locked edges cannot be reversed). Note that e_1 is locked by moving the lock gadget's block from X to Y, e_2 is locked by moving the lock gadget's block from Y to Z, and both of them are unlocked by moving the lock gadget's block from Z to X. Therefore, by Propositions 1 and 2, it is easy to see that:

(a) e_1 may be locked if and only if it is directed inward, by visiting its gadget and entering the lock gadget using input entry A to move the lock gadget's block from X to Y (following the path indicated by solid line 1);

(b) e_2 may be locked if and only if it is directed inward and e_1 is locked, by visiting its gadget and entering the lock gadget using input entry C to move the lock gadget's block from Y to Z (following the path indicated by solid line 2);

(c) e_1 and e_2 may be unlocked at the same time if and only if they are both locked and e_3 is directed inward, by visiting e_3's gadget and entering the lock gadget using the output entry E to move the lock gadget's block from Z to X.

It remains to show that e_3 may be directed outward if and only if both e_1 and e_2 are directed inward and are locked. Again, by Propositions 1 and 2, it is easy to see that e_3 may be directed outward if and only if the lock gadget's block is located at Z, in which case both e_1 and e_2 are directed inward and are locked, by visiting the lock gadget and leaving using the output exit F (following the path indicated by solid line 3). Note that e_1 and e_2 are always locked when e_3 is directed outward, so they may be directed outward if and only if e_3 is directed inward, and thus the minimum inflow constraint is always satisfied. □

5.4.3 The Construction

When we described the vertex gadgets we showed how the vertex entries and exits of every edge gadget and all the entries and exits of every lock gadget are connected using tunnels. To complete the map we need to describe how the gadgets are laid out and how to add tunnels that allow the player to move the droplet around and visit any edge gadget.

First, we place all the basic gadgets in a row from left to right and place all the tunnels required by the vertex gadgets under the row of gadgets. We note that the placement of the basic gadgets in Figures 5.7 and 5.8 was only used to facilitate the explanation of how the vertex gadgets are constructed, but the basic gadgets used for a vertex gadget could be far away in the map. This, however, is not a problem. In reality, the placement of the basic gadgets is irrelevant as long as we have the right connections, and we can always connect the gadgets in any way we want since the crossing of tunnels is not a problem; note that because of this we do not really require G to be planar.

Then, we place the droplet's starting location outside (to the left) of all the basic gadgets, we add a path from the starting location to the edge entry of every edge gadget, and we add a path that leads back from each exit of a basic gadget to the starting location; the tunnels for these paths are also placed under the row of gadgets. This allows the player

to move the droplet around and visit any edge gadget at any time: the paths from the starting location to the edge gadgets can be used to visit any edge gadget, and after visiting any basic gadget, possibly changing its state by moving its block, we can always go back to the starting location and visit another edge gadget.

(a) (b)

Figure 5.9. Placing the pearl in the gadget for the goal edge. (a) Initial configuration: the pearl is shown as a black circle. (b) Collecting the pearl once the edge is reversed: the squares with dotted lines indicate the intermediate locations of the pushable block before it is moved to its final location.

Finally, we add a pearl in the gadget for the goal edge e, as illustrated in Figure 5.9. The edge gadget's block is initially placed on one side of the gadget, and the pearl is placed at the vertex exit on the opposite side so that it can be collected once the edge is reversed.

This completes the construction. The reduction runs in polynomial time since the number of obstacles required is polynomial in the number of vertices and edges. We have the following lemma:

Lemma 9. *There is a sequence of moves on G that eventually reverses the goal edge e if and only if the pearl in the Quell map can be collected.*

Proof. The correctness of the vertex gadgets and the fact that we can always visit any edge gadget ensure that the map emulates the constraint graph G, and it is easy to see that the pearl can be collected as soon as the goal edge e is reversed. Note that each edge gadget ensures that the local configuration remains legal for the corresponding vertex any time its state is changed by reversing one of its incident edges or changing its locks. We now explain why the global configuration also remains legal after every move.

Consider an edge $e_i = (u, v)$ directed inward toward v. By looking at Figures 5.7 and 5.8 we can see that e_i can only be reversed by visiting v, so its orientation can only be

changed from inward to outward. The vertex gadget for v guarantees that after every move the constraints are satisfied for v, and it is easy to see that they are also satisfied for u. When we reverse e_i and direct it inward toward u, we are increasing u's inflow, so the inflow constraint is satisfied. Moreover, if u is an AND* vertex and e_i is one of its input edges, it could not have been locked to u before being reversed since it was directed outward from u before the move, so the lock states for u are also valid. □

This completes the proof of Theorem 7.

5.5 Discussion

We have shown that ANY-MOVES-ALL-PEARLS with pushable blocks is a very difficult problem, by proving that it is PSPACE-complete. To prove this result we used a reduction from a restricted version of NCL that uses only OR vertices and AND* vertices. In our case it was easier to construct a vertex gadget for the AND* vertex than to construct a vertex gadget for the AND vertex. Perhaps the additional restrictions on the AND* vertex could be useful for showing that other problems are PSPACE-hard.

Moreover, we have shown how to construct the OR and AND* vertex gadgets out of two basic gadgets: an edge gadget and a lock gadget (actually, the edge gadget could easily be constructed from the lock gadget). It is also possible that other problems could be shown to be PSPACE-hard by designing these or similar basic gadgets, which could be potentially simpler than the NCL vertex gadgets, instead of designing the complete vertex gadgets.

However, we note that in our case we did not have to worry about tunnels crossing, but for other problems it may also be necessary to design a crossover gadget. A crossover gadget is often not necessary since NCL is PSPACE-complete on planar graphs. Nevertheless, it is sometimes necessary if there are crossings inside the vertex gadgets; for example, in the proof that Push-2-F is PSPACE-complete [25].

The fact that NCL does not usually require a crossover gadget is one of its main advantages because crossover gadgets are usually the hardest ones to design; see for example

the crossover gadgets in the PSPACE-completeness proofs for Sokoban [23] and for Push-2-F [25]. However, for puzzles with maximal sliding objects, the crossover gadget is usually trivial and the advantage of NCL does not seem to be so obvious.

NCL has been used to prove that several puzzles with movable objects are PSPACE-complete, but most of them do not have maximal sliding objects; see for example the proofs that Sliding Blocks, Sokoban, and Rush Hour are PSPACE-complete [10, 17]. In contrast, other reduction techniques have been used more often to show PSPACE-completeness of puzzles with maximal sliding objects. For example, simulation of a polynomial-space Turing machine has been used for Lunar Lockout with fixed blocks [19], and reductions from the non-empty intersection problem for finite automata have been used for Atomix [21], as well as PushPush-k and PushPushPush-k [26]. This may suggest that when a crossover gadget is required by NCL, or when a cross over gadget is trivial, then reductions from other problems may be simpler.

Despite this, our reduction is from NCL and the basic edge and lock gadgets used in our construction are quite simple. Moreover, it does not seem to be the case that the gadgets required by other techniques could be considerably simpler for our problem. For example, the key gadget required to simulate a polynomial-space Turing machine is a "lockable door" gadget [19] (also called "pass-reset" [23], or "gate" [22]). It has two independent paths and it is not obvious how to create it and avoid leakage between the two paths, at least with a single block; perhaps with multiple blocks it is possible to create such a gadget, but that would be more complex. Also, the key gadget required for a reduction from the non-empty intersection problem for finite automata is the "catalyst chamber" gadget [21]. This gadget requires more than one block, by definition, so it would also be more complex.

CHAPTER 6

CONCLUSION

In this dissertation, we have given a fairly complete characterization of the complexity of collecting items inside a grid map with obstacles, when using a maximal sliding agent. We studied the most fundamental problem of deciding if it is possible to collect all the items, and the two optimization problems of determining the maximum number of items that can be collected and determining the minimum number of moves required to collect all the items.

We showed that the problem of deciding if it is possible to collect all the items can be solved in polynomial time, and gave a simple 2-approximation algorithm for the maximization problem and an efficient fixed-parameter algorithm for the parameterized version of the minimization problem. We also gave approximation lower bounds for both optimization problems, showing that the maximization problem is at least as hard to approximate as Max-2-SAT, and showing that the minimization problem is NP-hard to approximate within $2 - \epsilon$, for any fixed $\epsilon > 0$, even in simple maps. Furthermore, we showed that the problem of deciding if it is possible to collect all the items is NP-complete in higher dimensions, even in simple 3-dimensional maps where the agent is allowed to move in only two layers in the third dimension, and that it is PSPACE-complete with blocks that can be pushed and slide with the agent, even in 2-dimensional maps.

Also, in our PSPACE-completeness proof (Chapter 5) we used a reduction from a restricted version of NCL that we propose, and that we believe may be useful for showing that other problems with movable objects are PSPACE-complete. Although it is out of the scope of this dissertation, in the future we may want to use this technique to try to show PSPACE-completeness of some of those problems.

Some interesting questions remain open. In particular, the possibility of finding better approximations for the optimization problems, and faster algorithms for the corresponding

parameterized decision problems. The evidence suggests that finding a constant approximation for the minimization problem is hard, but perhaps there are simpler maps, such as convex maps, for which there are better approximations or even exact polynomial-time algorithms.

In the future, we would like to work on other problems with movable objects including Lunar Lockout [19, 35], which uses maximal sliding, and Push-1[10, 25], which does not use maximal sliding but has pushable blocks. For many of those puzzles, the question of whether they are in NP or they are PSPACE-complete is still open. In particular, this question has remained open for more than a decade for both Lunar Lockout and Push-1, and we would like to consider those two problems next.

REFERENCES

[1] J. Brunner, M. Mihalák, S. Suri, E. Vicari, and P. Widmayer, "Simple robots in polygonal environments: a hierarchy," in *Proceedings of the 4th International Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS'08)*, 2008, pp. 111–124.

[2] S. Suri, E. Vicari, and P. Widmayer, "Simple robots with minimal sensing: from local visibility to global geometry," *The International Journal of Robotics Research*, vol. 27, pp. 1055–1067, 2008.

[3] Arduino. [Online]. Available: http://www.arduino.cc

[4] Fallen Tree Games. [Online]. Available: http://www.fallentreegames.com

[5] R. Abbott. Tilt Mazes. [Online]. Available: http://www.logicmazes.com/tilt.html

[6] A. Gilbert, B. Mitchell, and M. Cat. *click*mazes' tilt collection. [Online]. Available: http://www.clickmazes.com/indext.htm

[7] M. Jiang, P. J. Tejada, and H. Wang, "Quell," in *Proceedings of the 7th International Conference on Fun with Algorithms (FUN'14)*, 2014, pp. 240–251.

[8] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for Your Mathematical Plays*. Academic Press, 1982.

[9] R. J. Nowakowski, *More Games of No Chance*, ser. MSRI Publications, vol. 42. Cambridge, UK: Cambridge University Press, 2002.

[10] R. A. Hearn and E. D. Demaine, *Games, Puzzles, and Computation*. A K Peters/CRC Press, 2009.

[11] G. Kendall, A. Parkes, and K. Spoerer, "A survey of NP-complete puzzles," *International Computer Games Association Journal*, vol. 31, pp. 13–34, 2008.

[12] E. D. Demaine and R. A. Hearn, "Games of no chance 3," *MSRI Publications*, vol. 56, pp. 2–56, 2009.

[13] M. Forisěk, "Computational complexity of two-dimensional platform games," in *Proceedings of the 5th International Conference on Fun with Algorithms (FUN'10)*, 2010, pp. 214–227.

[14] G. Viglietta, "Gaming is a hard job, but someone has to do it!" in *Proceedings of the 6th International Conference on Fun with Algorithms (FUN'12)*, 2012, pp. 357–367.

[15] D. Ratner and M. Warmuth, "The $(n^2 - 1)$-puzzle and related relocation problems," *Journal of Symbolic Computation*, vol. 10, pp. 111–137, 1990.

[16] J. Hopcroft, J. Schwartz, and M. Sharir, "On the complexity of motion planning for multiple independent objects; PSPACE-hardness of the "warehouseman's problem"," *International Journal of Robotics Research*, vol. 3, no. 4, pp. 76–88, 1984.

[17] 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. 1–2, pp. 72–96, 2005.

[18] G. W. Flake and E. B. Baum, "Rush Hour is PSPACE-complete, or "why you should generously tip parking lot attendants"," *Theoretical Computer Science*, vol. 270, pp. 895–911, 2002.

[19] J. R. Hartline and R. Libeskind-Hadas, "The computational complexity of motion planning," *SIAM Review*, vol. 45, pp. 543–557, 2003.

[20] B. Engels and T. Kamphans, "Randolphs robot game is NP-hard!" *Electronic Notes in Discrete Mathematics*, vol. 25, pp. 49–53, 2006.

[21] M. Holzer and S. Schwoon, "Assembling molecules in ATOMIX is hard," *Theoretical Computer Science*, vol. 313, pp. 447–462, 2004.

[22] D. Dor and U. Zwick, "SOKOBAN and other motion planning problems," *Computational Geometry*, vol. 13, no. 4, pp. 215–228, 1999.

[23] J. C. Culberson, "Sokoban is PSPACE-complete," in *Proceedings of the International Conference on Fun with Algorithms (FUN'98)*, 1998, pp. 65–76.

[24] M. Hoffmann, "Push-$*$ is NP-hard," in *Proceedings of the 12th Canadian Conference on Computational Geometry (CCCG'00)*, 2000, pp. 205–210.

[25] E. D. Demaine, R. A. Hearn, and M. Hoffman, "Push-2-F is PSPACE-complete," in *Proceedings of the 14th Canadian Conference on Computational Geometry (CCCG'02)*, 2002, pp. 31–35.

[26] E. D. Demaine, M. Hoffmann, and M. Holzer, "PushPush-k is PSPACE-complete," in *Proceedings of the 3rd International Conference on Fun with Algorithms (FUN'04)*. Island of Elba, Italy: Edizioni Plus, Universitá di Pisa, May 26–28, 2004, pp. 159–170.

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

[28] A. Itai, C. H. Papadimitriou, and J. L. Szwarcfiter, "Hamilton paths in grid graphs," *SIAM Journal on Computing*, vol. 11, no. 4, pp. 676–686, 1982.

[29] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Tech. Rep., 1976.

[30] G. Wilfong, "Motion planning in the presence of movable obstacles," *Annals of Mathematics and Artificial Intelligence*, vol. 3, no. 1, pp. 131–150, 1991.

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

[32] C. H. Papadimitriou, P. Raghavan, M. Sudan, and H. Tamaki, "Motion planning on a graph," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS'94)*, 1994, pp. 511–520.

[33] F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier, "Finding optimal solutions to Atomix," in *Proceedings of the Joint German/Austrian Conference on AI: Advances in Artificial Intelligence*. Springer, 2001, pp. 229–243.

[34] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing general single-agent search methods using domain knowledge," *Artificial Intelligence*, vol. 129, pp. 219–251, 2001.

[35] M. Hock, "Exploring the complexity of the UFO puzzle," B.S. Thesis, Dept. Comp. Sci., Univ. California, Berkeley, 2002.

[36] G. Aloupis, E. D. Demaine, A. Guo, and G. Viglietta, "Classic nintendo games are (computationally) hard," in *Proceedings of the 7th International Conference on Fun with Algorithms (FUN'14)*, 2004, pp. 40–51.

[37] B. Engels and T. Kamphans, "On the complexity of Randolph's robot game," Rheinische Friedrich-Wilhelms-Universität Bonn. Institut für Informatik I., Tech. Rep. 005, 2005.

[38] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Santa Clara, CA, USA: Spring-Verlag, 2008.

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

[40] P. J. Cameron, *Combinatorics: Topics, Techniques, Algorithms*. Cambridge, UK: Cambridge University Press, 1994.

[41] M. Jiang, personal communication.

[42] I. Koutis, "Faster algebraic algorithms for path and packing problems," in *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP'08)*, 2008, pp. 575–586.

[43] R. Williams, "Finding paths of length k in $O(2^k)$ time," *Information Processing Letters*, vol. 109, no. 6, pp. 315–318, 2009.

[44] J. Håstad, "Some optimal inapproximability results," *Journal of the ACM*, vol. 48, no. 4, pp. 798–859, 2001.

[45] S. Khot, G. Kindler, E. Mossel, and R. O'Donnell, "Optimal inapproximability results for MAX-CUT and other 2-Variable CSPs?" in *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*, 2004, pp. 146–154.

[46] V. V. Vazirani, *Approximation Algorithms*. New York, NY, USA: Springer, 2002.

[47] P. Crescenzi, R. Silvestri, and L. Trevisan, "On weighted vs unweighted versions of combinatorial optimization problems," *Information and Computation*, vol. 167, no. 1, pp. 10–26, 2001.

[48] C. H. Papadimitriou and S. Vempala, "On the approximability of the traveling salesman problem," *Combinatorica*, vol. 26, pp. 101–120, 2006.

[49] A. M. Frieze, G. Galbiati, and F. Maffioli, "On the worst-case performance of some algorithms for the asymmetric traveling salesman problem," *Networks*, vol. 12, no. 1, pp. 23–39, 1982.

[50] M. Bläser, "A new approximation algorithm for the asymmetric TSP with triangle inequality," in *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03)*, 2003, pp. 638–645.

[51] U. Feige and M. Singh, "Improved approximation ratios for traveling salesperson tours and paths in directed graphs," in *Proceedings of the 10th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'07)*, 2007, pp. 104–118.

[52] N. S. H. Kaplan, M. Lewenstein, and M. Sviridenko, "Approximation algorithms for asymmetric TSP by decomposing directed regular multigraphs," *Journal of the ACM*, vol. 52, no. 4, pp. 602–626, 2005.

[53] A. Asadpour, M. X. Goemans, A. Madry, S. O. Gharan, and A. Saberi, "An $O(\log n / \log \log n)$-approximation algorithm for the asymmetric traveling salesman problem," in *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'10)*, 2010, pp. 379–389.

[54] R. D. Carr and S. Vempala, "Towards a 4/3 approximation for the asymmetric traveling salesman problem," in *Proceedings of the 11st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, 2000, pp. 116–125.

[55] W. J. Savitch, "Relationships between nondeterministic and deterministic tape complexities," *Journal of Computer and System Sciences*, vol. 4, pp. 177–192, 1970.

[56] L. J. Stockmeyer, "The polynomial-time hierarchy," *Theoretical Computer Science*, vol. 3, pp. 1–22, 1976.

CURRICULUM VITAE

Pedro J. Tejada

EDUCATION

Ph.D., Computer Science. Utah State University, Logan, Utah, USA. 2014.

M.S., Computer Science. Utah State University, Logan, Utah, USA. 2009.

B.E., Information Technology ("Ingeniería de Sistemas y Computación"). Pontificia Universidad Católica Madre y Maestra, Santo Domingo, Dominican Republic. 2003.

RESEARCH INTERESTS

Algorithms, computational complexity, games and puzzles.

JOURNAL PUBLICATIONS

Minghui Jiang, Vincent Pilaud, and Pedro J. Tejada. On a dispersion problem in grid labeling. *SIAM Journal on Discrete Mathematics*, 26:39–51, 2012.

Minghui Jiang, Xiaojun Qi, and Pedro J. Tejada. A computational-geometry approach to digital image contour extraction. *Transactions on Computational Science*, XIII, volume 6750 of *Lecture Notes in Computer Science*, pages 13–43, 2011.

Minghui Jiang, Pedro J. Tejada, Ramoni O. Lasisi, Shanhong Cheng, and D. Scott Fechser. K-partite RNA secondary structures. *Journal of Computational Biology*, 17:915–925, 2010.

CONFERENCE PUBLICATIONS

Minghui Jiang, Pedro J. Tejada, and Haitao Wang. Quell. In *Proceedings of the 7th International Conference on Fun with Algorithms* (FUN'14), volume 8496 of *Lecture Notes in Computer Science*, pages 240–251, Springer, July 1–3, 2014.

Guillaume Blin, Laurent Bulteau, Minghui Jiang, Pedro J. Tejada, and Stéphane Vialette. Hardness of longest common subsequence for sequences with bounded run-lengths. In *Proceedings of the 23rd Annual Symposium on Combinatorial Pattern Matching (CPM'12)*, volume 7354 of *Lecture Notes in Computer Science*, pages 138–148, Springer, July 3–5, 2012.

Minghui Jiang, Vincent Pilaud, and Pedro J. Tejada. On a dispersion problem in grid labeling. In *Proceedings of the 22nd Canadian Conference on Computational Geometry (CCCG'10)*, pages 75–78, August 9–11, 2010.

Minghui Jiang, Pedro J. Tejada, Ramoni O. Lasisi, Shanhong Cheng, and D. Scott Fechser. K-partite RNA secondary structures. In *Proceedings of the 9th Workshop on Algorithms in Bioinformatics (WABI'09)*, volume 5724 of *Lecture Notes in Bioinformatics*, pages 157–168, Springer, September 12–13, 2009.

Pedro J. Tejada, Xiaojun Qi, and Minghui Jiang. Computational geometry of contour extraction. In *Proceedings of the 21st Canadian Conference on Computational Geometry (CCCG'09)*, pages 25–28, August 17–19, 2009.

INDUSTRY EXPERIENCE

Software developer. Orange Dominicana, Santo Domingo, Dominican Republic. 2003 to 2007.

TEACHING EXPERIENCE

CS 1405 - CS 1 Lab (C++). Utah State University. 2011.

CS 2410 - Introduction to Graphical User Interface Development in Java. Utah State University. 2012 to 2014.

CS 5060 - Intensive Programming. Utah State University. 2011 to 2013.

HONORS AND AWARDS

Graduate Teaching Assistant of the year for the College of Engineering. Utah State University. 2013.

Outstanding Graduate Teaching Assistant Award for the Department of Computer Science. Utah State University. 2010.