

**Masterarbeit am Institut für Informatik der Freien Universität Berlin,
Arbeitsgruppe Theoretische Informatik**

The Complexity Class Unique End of Potential Line

Michaela Borzechowski

Matrikelnummer: 4677938

`michaela.borzechowski@fu-berlin.de`

Erstgutachter: Prof. Dr. Wolfgang Mulzer

Zweitgutachter: Prof. Dr. László Kozma

Berlin, September 17, 2021

Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Masterarbeit selbstständig und ohne unerlaubte Hilfe angefertigt habe. Alle verwendeten Quellen und Hilfsmittel sind im Literaturverzeichnis aufgeführt und wörtlich oder inhaltlich aus den benutzten Quellen entnommene Stellen sind als solche kenntlich gemacht. Ich erkläre weiterhin, dass diese Arbeit nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Michaela Borzechowski, Berlin, September 17, 2021

Abstract

The complexity class Unique End of Potential Line (**UniqueEOPL**) was introduced in 2018 by [Fea+18] and captures total search problems that are promised to have a unique solution. Therefore, the original promise version of each problem is transformed to a total search version. **UniqueEOPL** contains some interesting problems like **P-LCP**, **CUBE-USO** and **ARRIVAL**. It is an especially interesting class because it has a "natural" complete problem: **ONE PERMUTATION DISCRETE CONTRACTION (OPDC)**.

The goal of this work is to give a comprehensive introduction to the complexity theory of search problems with a focus on the class **UniqueEOPL**. A list of all currently known problems in **UniqueEOPL** is presented and for each problem, a definition and examples are given. Some selected reductions are shown in full detail and with corrections.

The main contribution is the new containment result of **GRID-USO** in **UniqueEOPL**, which is proven via a reduction from **GRID-USO** to **UNIQUE FORWARD EOPL**. From that result it also follows that **P-GLCP** is contained in **UniqueEOPL** too.

Contents

1. Introduction and Motivation	3
2. Types of Problems	5
2.1. Decision Problems	5
2.2. Search Problems	6
2.2.1. Reductions between Search Problems	6
2.3. Optimization Problems	7
2.4. Promise Problems	8
2.4.1. Turning Promise Problems into Search Problems	9
2.4.2. Promise Preserving Reductions	11
3. Complexity Classes	13
3.1. Semantic vs. Syntactic Classes	13
3.2. Complexity Classes for Decision Problems	14
3.2.1. P	14
3.2.2. NP	14
3.3. Complexity Classes for Search Problems	14
3.3.1. FP	14
3.3.2. FNP	14
3.3.3. TFNP	15
3.3.4. PLS	15
3.3.5. PPAD	15
3.3.6. CLS	16
3.3.7. EOPL	17
3.3.8. UniqueEOPL	19
3.3.9. Relationships between Classes	21
3.4. Complexity Classes for Promise Problems	22
3.4.1. Promise-P	22
3.4.2. Promise-NP	22
3.4.3. PromiseUEOPL	22
4. UniqueEOPL-complete Problems	23
4.1. ONE PERMUTATION DISCRETE CONTRACTION (OPDC)	23
4.1.1. OPDC is in UniqueEOPL	29
4.1.1.1. OPDC to UNIQUE FORWARD EOPL	30
4.1.1.2. UNIQUE FORWARD EOPL to UNIQUE FORWARD EOPL+1	43
4.1.1.3. UNIQUE FORWARD EOPL+1 to UNIQUE END OF Po- TENTIAL LINE	44
4.1.2. OPDC is UniqueEOPL-hard	47
4.1.2.1. UNIQUE END OF POTENTIAL LINE to UNIQUE EOPL+1	47
4.1.2.2. UNIQUE EOPL+1 to OPDC	48

5. Problems in UniqueEOPL	58
5.1. S-ARRIVAL	61
5.1.1. S-ARRIVAL to UNIQUE FORWARD EOPL+1	64
5.2. UNIQUE SINK ORIENTATION OF CUBES (CUBE-USO)	68
5.2.1. CUBE-USO to OPDC	72
5.3. P-MATRIX LINEAR COMPLEMENTARITY PROBLEM (P-LCP)	74
5.3.1. P-LCP to CUBE-USO	78
5.4. P-GENERAL-LCP (P-GLCP)	81
5.5. SIMPLE STOCHASTIC GAME (SSG)	83
5.6. DISCOUNTED GAME	86
5.7. MEAN PAYOFF GAME (MPG)	87
5.8. PARITY GAME	89
5.9. α -HAM SANDWICH	91
5.10. PAIRWISE LINEAR CONTRACTION (PL-CONTRACTION)	94
6. Trying to prove Grid-USO to be in UniqueEOPL	96
6.1. GRID-USO	96
6.1.1. GRID-USO- Failed definition	99
6.1.2. GRID-USO- Correct definition	102
6.2. GRID-USO to OPDC - Failed Construction	104
6.3. GRID-USO to UNIQUE FORWARD EOPL	108
7. Conclusion and Future Directions	132
Appendix	134
A. Background Information on LCP's	134
A.1. Solving an LCP	134
A.2. Principal Pivot Transformation	134
A.3. Perturbation	136
B. GRID-USO to OPDC - Working Solution Types	138
Bibliography	144

GLOSSARY OF NOTATION

Q	A decision problem
R	A search problem
Π	An optimization problem
Ψ	A promise problem
\mathcal{I}^B	The set of instances of a problem B
I	An instance $I \in \mathcal{I}^B$
\mathcal{S}^B	The set of solutions to a problem B
s	A solution $s \in \mathcal{S}^B$
A	An algorithm
S	Successor function
P	Predecessor function
c	Cost (also called potential) function
d	The dimension
o	The start node
f	A function
$const$	A constant
δ	A metric
TM	A Turing Machine
ϵ	A very small number
Σ	An alphabet
L	A language
i	An index
j	Another index
k	Another index
l	Another index

Graphs

G	A graph
V	Set of vertices
v	A single vertex $v \in V$
u	Another vertex $u \in V$
E	Set of edges
E^+	The set of outgoing edges
E^-	The set of incoming edges
e	A single edge $e \in E$

Matrices

M	A square matrix
N	A vertical block matrix
q	A vector
z	Another vector
w	Another vector
I	The identity matrix
e	The unit vector
α	A set of positive integers (usually indices)
$M_{\alpha,\beta}$	The submatrix of M is the matrix whose entries lie in the rows of M indexed by α and the columns indexed by β .

Boolean formulas

φ	A Boolean formula
C	A clause in a Boolean formula

Grids

Γ	A grid
p	A point in the grid
q	Another point
ω	The grid width
C	A cube - a special case of a grid with grid width 2 in every dimension.
K	A partition
κ	A single block of a partition $\kappa \in K$
\mathbf{f}	A face
\mathbf{s}	A slice
γ	A subgrid

Sets

N	A set
M	Another set

Problem specific variables

\mathcal{D}	A family of direction functions in OPDC
D	A single direction function $D \in \mathcal{D}$ in OPDC
ϕ	An orientation function from CUBE-USO and GRID-USO
σ	An outmap function from CUBE-USO and GRID-USO
ρ	A strategy of a probabilistic game
λ	The discount used in DISCOUNTED GAME

1. Introduction and Motivation

Complexity theory is the idea of clustering problems that are equally hard or easy to solve.

Such results are generally proven by showing that a given problem A can be reduced to a problem B whose complexity is known, thereby showing that A can be solved at least as fast as B. This also works the other way around. If we know that Problem A is hard to solve and we can reduce it to problem B, then it is very likely that B is hard to solve as well. Two problems are polynomial time equivalent if both of them can be reduced to one another in polynomial time.

For a given problem A, there are infinitely many possible algorithms, so how can it be proven that there does not exist an algorithm better than a given one? In this case, it can't. *But* it can be proven by reductions that if A is as hard to solve as many well-known problems to which no efficient solution has been found yet, it is very unlikely that A can be solved more efficiently. Ok, so finding a solution for A is hard. But is it still hard if randomness is used, or is it faster if we don't want the exact result but only an approximation, or is it easier to find the solution if we know that it is unique?

In computer science, there are many problems that are naturally formulated as search problems where the task is to *find* a solution. Promise problems, where a certain property is promised to hold, are also an intuitive approach to formulate certain problems. Nonetheless, classical complexity theory works with decision problems and it may happen that the decision and the search problem are not computationally equivalent [Bea+98]. Particularly this is the case if it is guaranteed that there exists a solution. Research about the complexity of search problems that always have a solution has been done since 1991 when [MP91] defined the class TFNP, which contains *total* search problems. But TFNP is a semantic class and therefore it is unlikely to have any complete problems. Since then, many subclasses of TFNP have been defined such as PPAD, PLS, CLS, EOPL and UniqueEOPL, some of which do indeed have complete problems.

UniqueEOPL, which is short for "unique end of potential line", was introduced in 2018 by [Fea+18]. It captures search problems that can be solved by local search, i.e. it is possible to jump from one candidate solution to the next one in polynomial time. Furthermore, problems in UniqueEOPL are promised to have a unique solution. A problem in UniqueEOPL can be interpreted as an exponentially large, directed, acyclic graph where each node is a candidate solution and has a cost (or potential). The in- and out-degree of each node is at most one which means that the nodes form an exponentially long line. The unique solution, the node with the highest cost, is at the end of that unique line. It is a subclass of CLS, which is equal to $PPAD \cap PLS$. UniqueEOPL has, besides UNIQUE END OF POTENTIAL LINE itself and some variants of the problem, currently only one "natural" complete problem: OPDC.

Since the class has been introduced only recently, the knowledge about it is still very limited. There aren't many problems proven to be in UniqueEOPL yet, though there

surely are more problems that fulfill the necessary conditions. Furthermore, it is likely that there are more UniqueEOPL-complete problems as well.

For the relationship between EOPL, UniqueEOPL and the other complexity classes there are still some open questions. In [Fea+18] it was already asked whether $\text{CLS} = \text{EOPL}$ or not, but since [Fea+20a] proved that $\text{CLS} = \text{PPAD} \cap \text{PLS}$, this question becomes even more interesting.

The intention of defining a class for problems with unique solutions is that it might be easier to solve these than the version with multiple solutions. Since UniqueEOPL is a real subset of CLS, TFNP and PLS, there certainly *is* a difference between the problems.

The aim of this work is to give an easy to understand introduction to the complexity theory of search problems with focus on the class UniqueEOPL. A list of all (to the best of my knowledge) currently known problems in UniqueEOPL is presented, as well as some selected proofs with the most interesting proof strategies in detail and with examples. This shall make it easier for future researchers to prove more problems to be in UniqueEOPL or to be UniqueEOPL-complete and to further investigate the characteristics of EOPL and UniqueEOPL.

2. Types of Problems

There are different kinds of problems that are analyzed and sorted into several complexity classes. In the following, we will take a closer look at decision problems, search problems, optimization problems and promise problems.

Let $\Sigma := \{0, 1\}$ be an alphabet. $\mathcal{I}^B \subseteq \Sigma^*$ denotes a set of all problem instances encoded as bit strings of a problem B . Let $\mathcal{S}^B(I) \subseteq \Sigma^*$ be the set of solutions to an instance $I \in \mathcal{I}^B$.

2.1. Decision Problems

Definition 2.1.1 (Decision Problem [SY82]). A decision problem $Q = (\mathcal{I}_{\text{YES}}^Q, \mathcal{I}_{\text{NO}}^Q)$ where $\mathcal{I}_{\text{YES}}^Q, \mathcal{I}_{\text{NO}}^Q \subseteq \mathcal{I}^Q$ are defined as

- $I \in \mathcal{I}_{\text{YES}}^Q : \mathcal{S}^Q(I) \neq \emptyset$ and
- $I \in \mathcal{I}_{\text{NO}}^Q : \mathcal{S}^Q(I) = \emptyset$.

It follows that $\mathcal{I}_{\text{YES}}^Q \cap \mathcal{I}_{\text{NO}}^Q = \emptyset$ and $\mathcal{I}_{\text{YES}}^Q \cup \mathcal{I}_{\text{NO}}^Q = \mathcal{I}^Q$.

The goal of the decision problem is to tell whether a given instance $I \in \mathcal{I}^Q$ is in $\mathcal{I}_{\text{YES}}^Q$ or $\mathcal{I}_{\text{NO}}^Q$, i.e. whether I has a solution or not. At this point it is not of interest *what* this solution is, only its existence is important.

An algorithm $A_Q : \mathcal{I}^Q \rightarrow \{0, 1\}$ solves a decision problem if

$$\forall I \in \mathcal{I}^Q : A_Q(I) \text{ terminates} \wedge (A_Q(I) = 1 \Leftrightarrow I \in \mathcal{I}_{\text{YES}}^Q).$$

The algorithm returns 1 if the given instance I has a solution. Analogously if $A_Q(I) = 0$, this means that $I \in \mathcal{I}_{\text{NO}}^Q$ and the instance I has no solution.

Example 2.1.2. Let Q be CIRCUIT-SAT (C-SAT). In an instance $I \in \mathcal{I}^{Q-\text{C-SAT}}$ we are given a boolean combinational circuit φ composed of arbitrary n boolean variables x_1, \dots, x_n connected with \wedge , \vee , and \neg gates. The decision to make is whether I is satisfiable or not [Cor+09, p.1072].

The set of all instances $\mathcal{I}^{Q-\text{C-SAT}}$ is the set of all boolean formulas φ . $\mathcal{I}_{\text{YES}}^{Q-\text{C-SAT}}$ is the set of all satisfiable boolean circuits and $\mathcal{I}_{\text{NO}}^{Q-\text{C-SAT}}$ contains all not satisfiable ones. For example:

$$\begin{aligned} (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) &=: \varphi_1 \in \mathcal{I}_{\text{YES}}^{Q-\text{C-SAT}} \\ (x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) &=: \varphi_2 \in \mathcal{I}_{\text{YES}}^{Q-\text{C-SAT}} \\ x_1 \wedge \overline{x_1} \wedge x_2 &=: \varphi_3 \in \mathcal{I}_{\text{NO}}^{Q-\text{C-SAT}}. \end{aligned}$$

2.2. Search Problems

Definition 2.2.1 (Search Problem). A search problem (or sometimes called function problem) R is a relation

$$R \subseteq \mathcal{I}^R \times \mathcal{S}^R := \{(I, s) \mid I \in \mathcal{I}^R, s \in \mathcal{S}^R(I)\}.$$

A pair $(I, s) \in R$ represents an input instance I and a solution s of instance I .

$\mathcal{I}_{\text{YES}}^R$ and $\mathcal{I}_{\text{NO}}^R$ are defined as in the decision problem in Definition 2.1.1 on the preceding page.

An algorithm $A_R : \mathcal{I}^R \rightarrow \mathcal{S}^R \cup \{\perp\}$ solves R if

- $\forall I \in \mathcal{I}_{\text{YES}}^R \exists s \in \mathcal{S}^R(I) : (A_R(I) \text{ terminates and } A_R(I) = s)$ and
- $\forall I \in \mathcal{I}_{\text{NO}}^R : (A_R(I) \text{ terminates and } A_R(I) = \perp).$

If the given instance has at least one solution, the algorithm returns it. If the instance does not have a solution, the algorithm correctly states that by returning \perp .

Example 2.2.2. Recall CIRCUIT-SAT from Example 2.1.2 on the previous page. As a search problem, the question is "What is an assignment of the variables such that the boolean formula φ is satisfied?"

$$\begin{aligned} \mathcal{S}^{R\text{-C-SAT}}(\varphi_1 := (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})) &= \{(x_1 = \text{true}, x_2 = \text{false}), \\ &\quad (x_1 = \text{false}, x_2 = \text{true})\} \\ \mathcal{S}^{R\text{-C-SAT}}(\varphi_2 := (x_1 \vee \overline{x_2}) \wedge (x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})) &= \{(x_1 = \text{true}, x_2 = \text{false})\} \\ \mathcal{S}^{R\text{-C-SAT}}(\varphi_3 := (x_1 \wedge \overline{x_1} \wedge x_2)) &= \emptyset. \end{aligned}$$

2.2.1. Reductions between Search Problems

Definition 2.2.3 (Polynomial time reduction for search problems). A search problem R is reducible in polynomial time to a search problem R' if there exist two polynomial time functions $f : \mathcal{I}^R \rightarrow \mathcal{I}^{R'}$ and $g : \mathcal{I}^R \times \mathcal{S}^{R'}(f(I)) \rightarrow \mathcal{S}^R(I)$ such that

- if I is an instance of R , $f(I)$ is an instance of R' ,
- if s' is a solution to $f(I)$, then $g(I, s')$ is a solution to I of R .

The function f transforms instances from the first problem to instances of the second problem. The function g transforms solutions that are found in the second problem back to solutions of the first problem. The idea is that instead of solving R , we reduce it to R' , solve that one instead and transform its solution back to R .

2.3. Optimization Problems

Definition 2.3.1 (Optimization Problem). An optimization problem Π is a search problem with specific properties. Let $c : \mathcal{S}^\Pi(I) \rightarrow \mathbb{R}$ be the function that assigns each solution a certain cost (or potential). For a particular problem instance $I \in \mathcal{I}^\Pi$ the goal is to find a globally optimal solution $s^* \in \mathcal{S}^\Pi(I)$ such that

- (i) $\forall s \in \mathcal{S}^\Pi(I) : c(s^*) \leq c(s)$ if Π is a **minimization problem** or
- (ii) $\forall s \in \mathcal{S}^\Pi(I) : c(s^*) \geq c(s)$ if Π is a **maximization problem**.

Let $\mathcal{I}_{\text{YES}}^\Pi, \mathcal{I}_{\text{NO}}^\Pi \subseteq \mathcal{I}^\Pi$ such that for a minimization problem (and respectively with \geq for a maximization problem) it holds that

- $I \in \mathcal{I}_{\text{YES}}^\Pi : \exists s^* \in \mathcal{S}^\Pi(I) \forall s \in \mathcal{S}^\Pi(I) c(s^*) \leq c(s)$
- $I \in \mathcal{I}_{\text{NO}}^\Pi : \nexists s^* \in \mathcal{S}^\Pi(I) \forall s \in \mathcal{S}^\Pi(I) c(s^*) \leq c(s)$.

It follows that $\mathcal{I}_{\text{YES}}^\Pi \cap \mathcal{I}_{\text{NO}}^\Pi = \emptyset$ and $\mathcal{I}_{\text{YES}}^\Pi \cup \mathcal{I}_{\text{NO}}^\Pi = \mathcal{I}^\Pi$.

Note that the set $\mathcal{S}^\Pi(I)$ in an optimization problem might be differently defined here than $\mathcal{S}^Q(I)$ in a decision or search problem. For example, \mathcal{S}^Π can contain all possible assignments of variables in a C-SAT formula to *true* or *false*, not just the ones that satisfy the formula. Searching for an optimal solution s^* doesn't necessarily mean that s^* solves the decision version of the problem. One might have a not satisfiable formula and still search for the best possible assignment as an approximation.

Therefore, $\mathcal{I}_{\text{YES}}^\Pi$ and $\mathcal{I}_{\text{NO}}^\Pi$ are differently defined as well. $\mathcal{I}_{\text{YES}}^\Pi$ contains all instances that have at least one optimum. $\mathcal{I}_{\text{NO}}^\Pi$ contains instances that do not have an optimum, for example an asymptotic function.

An algorithm $A_\Pi : \mathcal{I}^\Pi \rightarrow \mathcal{S}^\Pi$ solves Π if

- $\forall I \in \mathcal{I}_{\text{YES}}^\Pi : (A_\Pi(I) \text{ terminates} \wedge A_\Pi(I) = s^*)$ and
- $\forall I \in \mathcal{I}_{\text{NO}}^\Pi : (A_\Pi(I) \text{ terminates} \wedge A_\Pi(I) = \perp)$.

Example 2.3.2. Recall CIRCUIT-SAT from Example 2.2.2 on the preceding page. Let the cost function return the number of satisfied clauses of φ in its conjunctive normal form. As a maximization problem, CIRCUIT-SAT searches for the solution s^* with the highest cost, i.e. the highest number of satisfied clauses, independent of the overall satisfiability of the formula.

This problem always has an optimal solution. It might happen that all solutions are optimal, but it cannot happen that there does not exist an optimal solution.

$$\begin{aligned} \mathcal{S}^{\Pi\text{-C-SAT}}(I) &= \{\text{All possible assignments of } x_1 \text{ and } x_2 \text{ to } \textit{true} \text{ or } \textit{false}\} \\ \mathcal{I}_{\text{YES}}^{\Pi\text{-C-SAT}} &= \mathcal{I}^{\text{C-SAT}} \\ \mathcal{I}_{\text{NO}}^{\Pi\text{-C-SAT}} &= \emptyset \\ \varphi_3 &:= x_1 \wedge \overline{x_1} \wedge x_2 \end{aligned}$$

Solutions $\mathcal{S}^{\Pi-C-SAT}$		Cost
x_1	x_2	$c(x_1, x_2)$
<i>true</i>	<i>true</i>	2
<i>true</i>	<i>false</i>	1
<i>false</i>	<i>true</i>	2
<i>false</i>	<i>false</i>	1

φ_3 is not satisfiable at all, but it has still more than one optimal solution with maximal cost. An algorithm $A_{\Pi-C-SAT}(\varphi_3)$ returns either $(x_1 = true, x_2 = true)$ or $(x_1 = false, x_2 = true)$.

The polynomial time reduction defined in Chapter 2.2.1 on page 6 also applies to optimization problems. Additionally, it must be ensured that the optimal solution for one problem is mapped to the optimal solution of the other problem.

2.4. Promise Problems

A promise problem is a partial decision problem. Just like in a decision problem, the task is to distinguish between strings that represent YES-instances and strings that represent NO-instances, but additionally it is *promised* that all input instances fulfill a certain property. This might be a property that is easy to check, for example: "Given an acyclic graph, decide whether or not ...", or a property that is hard to check, for example: "Given a graph with a hamilton cycle, decide whether or not ...". If it is easy to recognize disallowed strings (strings that do not fulfill the promise), the promise is called trivial and the problem is a regular decision problem [Gol06, p. 255].

Definition 2.4.1 (Promise Problem). Let $\mathcal{I}_{YES}^\Psi, \mathcal{I}_{NO}^\Psi \subseteq \mathcal{I}^\Psi$ where $\mathcal{I}_{YES}^\Psi \cap \mathcal{I}_{NO}^\Psi = \emptyset$. The set $\mathcal{I}_{YES}^\Psi \cup \mathcal{I}_{NO}^\Psi =: \mathcal{I}_{PROMISE}^\Psi$ is called promise [Gol06, Definition 1].

A promise problem Ψ is either represented by the tuple $\Psi := (\mathcal{I}_{YES}^\Psi, \mathcal{I}_{NO}^\Psi)$ [Gol06, Definition 1] or alternatively by the tuple $\Psi := (\mathcal{I}_{YES}^\Psi, \mathcal{I}_{PROMISE}^\Psi)$ [SY82]. The third set can be calculated respectively.

An algorithm $A_\Psi : \mathcal{I}^\Psi \rightarrow \{0, 1\}$ solves Ψ if

$$\forall I \in \mathcal{I}^\Psi : (I \in \mathcal{I}_{PROMISE}^\Psi \Rightarrow (A_\Psi(I) \text{ terminates} \wedge (A_\Psi(I) = 1 \Leftrightarrow I \in \mathcal{I}_{YES}^\Psi)))$$

Note that any algorithm A_Ψ that solves such a promise problem still only needs to distinguish YES from NO instances. If given a NON-PROMISE instance, A_Ψ will not give a correct answer and might not even halt [Gol06, p. 255], [HMS89, p. 95].

For a promise problem where $\mathcal{I}_{YES}^\Psi \cup \mathcal{I}_{NO}^\Psi = \mathcal{I}^\Psi$, the promise is called trivial. It then is equivalent to a decision problem [Gol06, p. 256].

Example 2.4.2. An example of a promise is "there is a unique or no solution". The sets $\mathcal{I}_{YES}^\Psi, \mathcal{I}_{NO}^\Psi$ and $\mathcal{I}_{PROMISE}^\Psi$ can be characterized as follows:

$$\begin{aligned} \mathcal{I}_{YES}^\Psi &:= \{ \text{All instances that have a unique solution} \} \\ \mathcal{I}_{NO}^\Psi &:= \{ \text{All instances that have no solution} \} \\ \mathcal{I}_{PROMISE}^\Psi &:= \mathcal{I}_{YES}^\Psi \cup \mathcal{I}_{NO}^\Psi = \{ \text{All instances that have a unique or no solution} \}. \end{aligned}$$

The instances in the white area in Figure 2.1 are instances that do not fulfill the promise.

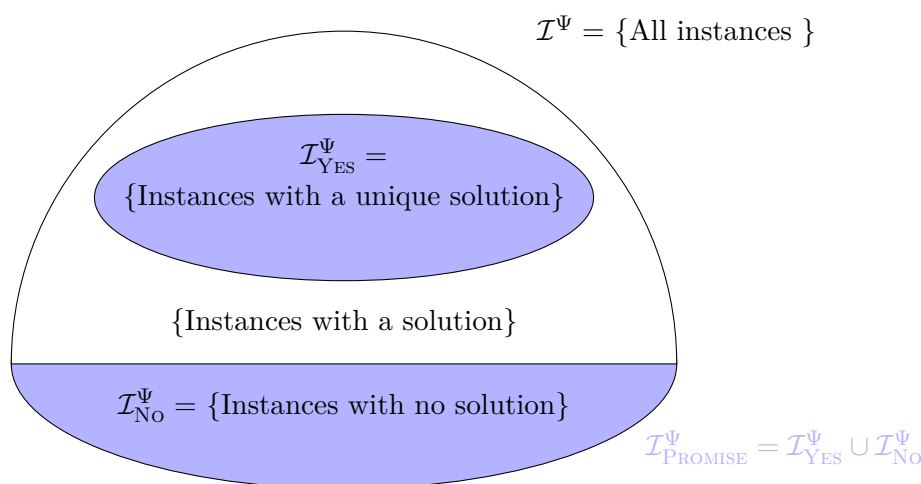


Figure 2.1.: Venn diagram of the sets $\mathcal{I}_{\text{YES}}^\Psi$, $\mathcal{I}_{\text{NO}}^\Psi$ and $\mathcal{I}_{\text{PROMISE}}^\Psi$ (colored in blue).

Example 2.4.3. Recall CIRCUIT-SAT from Example 2.1.2 on page 5. When adding the promise that "there is a unique or no solution", which is hard to verify, the problem is called UNIQUE-SAT [Cal+08], [VV86].

$$\begin{aligned} (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2) &=: \varphi_1 \notin \mathcal{I}_{\text{YES}}^{\text{UNIQUE-SAT}} \cup \mathcal{I}_{\text{NO}}^{\text{UNIQUE-SAT}} \\ (x_1 \vee \bar{x}_2) \wedge (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2) &=: \varphi_2 \in \mathcal{I}_{\text{YES}}^{\text{UNIQUE-SAT}} \\ x_1 \wedge \bar{x}_1 &=: \varphi_3 \in \mathcal{I}_{\text{NO}}^{\text{UNIQUE-SAT}}. \end{aligned}$$

φ_1 has multiple solutions. The promise does not hold. It is neither in $\mathcal{I}_{\text{YES}}^{\text{UNIQUE-SAT}}$ nor in $\mathcal{I}_{\text{NO}}^{\text{UNIQUE-SAT}}$.

φ_2 has exactly one solution, which is $x_1 = \text{true}$ and $x_2 = \text{false}$. Therefore, φ_2 is in $\mathcal{I}_{\text{YES}}^{\text{UNIQUE-SAT}}$.

φ_3 has no solutions and is therefore in $\mathcal{I}_{\text{NO}}^{\text{UNIQUE-SAT}}$.

What would happen if φ_1 is given to $A_{\text{UNIQUE-SAT}}$? Nothing defined happens. $A_{\text{UNIQUE-SAT}}$ might go berserk and answer "42" or just hang itself. It does not check if the promise holds.

2.4.1. Turning Promise Problems into Search Problems

Promise problems are of historic importance. The problems whose complexity is studied here are naturally formulated as promise problems, since there is often no efficient way to verify the promise. For example, it is as hard to check whether a matrix is a P-matrix as it is to solve the linear complementarity problem based on this matrix.

For problems studied in this work the promise is "there exists a unique solution" or some property that implies the uniqueness of the solution.

For an introduction to some basic promise problem complexity classes see for example [Gol06, Definition 2]. The problems studied in this work can all be sorted into the promise complexity class PromiseUEOPL (see Section 3.4.3 on page 22).

In this work we are also interested in the search problem complexity of these problems. Therefore, all of these promise problems are transformed to a total search version and sorted into search problem complexity classes. This works because for each problem studied here there exists a short certificate proving that the promise does not hold [Fea+18, p.2].

Informally, a problem of the structure "Given promise A , find a solution for B " is translated to: "Find either a solution for B or a violation against A ."

Formally, all instances of a promise problem Ψ that do not fulfill the promise are sorted from $\mathcal{I}_{\text{PROMISE}}^\Psi$ into $\mathcal{I}_{\text{YES}}^R$ of the respective search problem R . Therefore in the search problem they now have a valid solution: a violation-solution.

Transforming promise problem Ψ to search problem R :

- $\mathcal{I}_{\text{YES}}^R := \mathcal{I}_{\text{YES}}^\Psi \cup (\mathcal{I}^\Psi \setminus \mathcal{I}_{\text{PROMISE}}^\Psi)$,
- $\mathcal{I}_{\text{NO}}^R := \mathcal{I}_{\text{NO}}^\Psi$.

There are four possible outcomes for these transformed problems:

- 1) The promise holds and we find a solution.
- 2) The promise holds and we find no solution.
- 3) The promise doesn't hold and we find a violation.
- 4) The promise doesn't hold but we find a solution anyway.

The fourth case is the most confusing one. If we continued to search, at some point a violation would be found. But since we are interested in a solution, we are happy as soon as one is found. Note that if we are in case 4 and the promise is "there is a unique solution", the found solution isn't necessarily unique. If the promise doesn't hold, there might be several solutions and maybe we just found one of them before we could find a violation.

The point is that we want to find a result under any circumstances. If we find a solution, that's nice. If we can't find one, we want to know *why*. This reason we are told by the violation solutions. All transformed problems are *total*, i.e. they are always guaranteed to have a result. Therefore, they are contained in TFNP (see Section 3.3.3 on page 15).

This kind of transformation cannot be done for all promise problems. For example, if the promise does not have a certificate that proves them wrong in polynomial time, then we cannot formulate a proper violation. The violations need to be constructed carefully and a proof is needed that they accurately represent the promise.

[Fea+20b] conjectures that allowing different sets of violations might change the complexity of the problem. For example, for P-LCP (see Section 5.3 on page 74) there are two violations stated even though only one would be enough to make the problem total. There doesn't seem to be a polynomial time algorithm to convert one violation to the other. It is left open whether this is a general true statement or not.

The natural promise version of any problem with violations is: *Under the promise that there exists no violation, find a valid solution* [Fea+20b, p. 2].

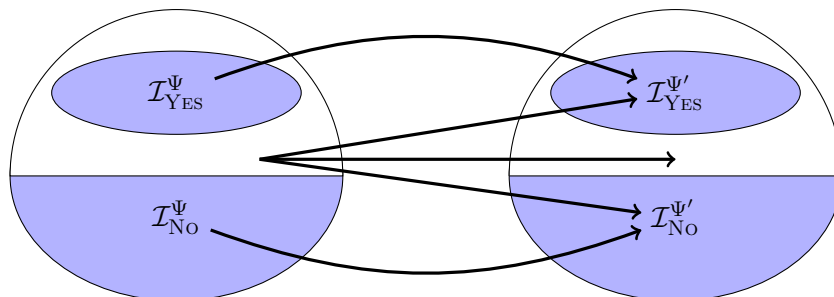
2.4.2. Promise Preserving Reductions

In order to analyze the complexity of promise problems, some kind of reduction is needed that preserves the promise (similar to preserving the number of solutions in parsimonious reductions). An instance that satisfies the promise in Ψ needs to satisfy the promise in the reduced promise problem Ψ' as well.

Definition 2.4.4 (Promise preserving reduction for promise problems [Gol06]). A promise problem $\Psi = (\mathcal{I}_{\text{YES}}^{\Psi}, \mathcal{I}_{\text{NO}}^{\Psi})$ is *promise Karp reducible* to the promise problem $\Psi' = (\mathcal{I}_{\text{YES}}^{\Psi'}, \mathcal{I}_{\text{NO}}^{\Psi'})$ if there exists a polynomial time computable function $f : \mathcal{I}^{\Psi} \rightarrow \mathcal{I}^{\Psi'}$ such that

- $\forall I \in \mathcal{I}_{\text{YES}}^{\Psi} : f(I) \in \mathcal{I}_{\text{YES}}^{\Psi'}$
- $\forall I \in \mathcal{I}_{\text{NO}}^{\Psi} : f(I) \in \mathcal{I}_{\text{NO}}^{\Psi'}$.

For all instances that do not fulfill the promise in Ψ it is not defined what happens to them in the reduced problem Ψ' . There might be a violating instance I in Ψ which reduced variant $f(I)$ is *not* a violation in Ψ' , since $f(\mathcal{I}_{\text{YES}}^{\Psi}) \cup f(\mathcal{I}_{\text{NO}}^{\Psi}) \subseteq \mathcal{I}_{\text{YES}}^{\Psi'} \cup \mathcal{I}_{\text{NO}}^{\Psi'}$.



It holds that if an instance I of Ψ fulfills the promise, then the resulting instance $f(I)$ of Ψ' fulfills the promise as well. For the transformed search problem it follows that if an instance I of the original problem Ψ has no violations, then $f(I)$ has no violations [Fea+18, p. 14].

If the original instance has a valid solution, the transformed instance must have a valid solution. If the transformed instance has a violation, the original instance must have had a violation. If the original instance has a violation, it doesn't matter what happens in the transformed instance. It might end in a violation or in a valid solution.

Promise preserving reductions are transitive [SY82, Lemma 1 (i)].

Definition 2.4.5 (Promise preserving polynomial time reduction for search problems [Fea+20b]). A search problem R is reducible in polynomial time to a search problem R' under promise preserving reduction if there exist two polynomial time functions $f : \mathcal{I}^R \rightarrow \mathcal{I}^{R'}$ and $g : \mathcal{I}^R \times \mathcal{S}^{R'}(f(I)) \rightarrow \mathcal{S}^R(I)$ such that

- if I is an instance of R , $f(I)$ is an instance of R' ,
- if s' is a solution to $f(I)$, then $g(I, s')$ is a solution to I of R ,
- if s' is a violation of $f(I)$, then $g(I, s')$ is a violation of I of R .

In the following, if a problem Ψ is reducible to Ψ' under promise preserving reduction, it is written as $\Psi \preceq_{\text{PROMISE}} \Psi'$.

If a promise problem is made total in the way that is described in Section 2.4.1 on page 9 and reduced via promise preserving reduction to another total search problem of that kind, their promise versions are reducible to one another as well.

A reduction between two common search problems without violations is always promise preserving. The specification only becomes relevant whenever one of the two involved problems has violations. A promise preserving reduction is stronger than the common search problem reduction, but not at all necessary to place search versions of promise problems (a "transformed promise problem") in search problem complexity classes. For example, it is possible to reduce a search problem to a transformed promise problem that will only have violations. Or the other way around, it is possible to reduce a transformed promise problem to a search problem that has only valid solutions.

3. Complexity Classes

It is important to distinguish complexity classes for decision, search (also called functional) and promise problems. Optimization problems are sorted into search problem complexity classes, since they are a special kind of search problems. When talking about complexity classes, usually complexity of decision problems is meant. P and NP for example are decision problem classes. FP and FNP are equivalent versions for search problems. They are written with an "F" because search problems sometimes are called functional problems. Promise-P and Promise-NP are for promise problems.

Solving a decision problem is not harder than solving the corresponding search or optimization problem. Imagine it was easy to solve a certain search problem. To solve the corresponding decision problem one could run an algorithm for the search problem and return "yes" if it returns a result, or "no" if it doesn't [Cor+09, p. 1054].

The other direction is the more interesting one. Is the search problem harder than its corresponding decision problem? There are cases where both search and decision problems are easy, for example all problems that are in P. Then again there are cases where both search and decision problems are hard, for example SAT and its search version [Pap95, p. 228].

But sometimes the decision problem is easy (or even trivial) whereas the search problem is still hard and unknown or believed to be in FP. The best example for that are *total* problems, i.e. problems that always have a solution. The answer to the decision problem is by definition always "yes". Nonetheless for many of them it is still hard to actually *find* that solution [Pap95, p. 230].

Even total problems seem to have different complexities, based on properties like uniqueness of the solution. There are several sub-classes for total problems that are examined in the following.

3.1. Semantic vs. Syntactic Classes

One can distinguish between *semantic* and *syntactic* complexity classes.

A complexity class is called a *semantic* class if the Turing Machine (TM) defining this class has a property that is undecidable. For example BPP which stands for Bounded-Error Probabilistic Polynomial-Time is a semantic class. It accepts each string with probability at least $2/3$ and at most $1/3$. Testing whether a given TM has this property is undecidable [AB09, p. 137]. The same holds for problems in TFNP, whose defining Turing Machine accepts problems that always have a solution. Semantic classes tend to have no complete problems.

A *syntactic* complexity class is a class for which we can check whether a given TM defines the language of the class [Pap95, p. 255]. For example, given a string, it is easy

to determine if it is a valid encoding of a non deterministic Turing Machine which defines NP [AB09, p. 137]. Any syntactic class has a standard complete language, namely

$$\{(M, x) \mid M \in \mathcal{M} \text{ and } M(x) = \text{"yes"}\}$$

where \mathcal{M} is the class of all machines of the variant that define the class [Pap95, p. 255].

3.2. Complexity Classes for Decision Problems

The following two classes are well known to anyone who has heard of complexity theory. They are included here for the purpose of a more complete overview. The question whether $P = NP$ or $P \neq NP$ is one of the best known open problems today.

3.2.1. P

A decision problem $Q = (\mathcal{I}_{\text{YES}}^Q, \mathcal{I}_{\text{NO}}^Q)$ is in P if and only if there exists a deterministic Turing machine that decides in polynomial time for any given instance I whether or not $I \in \mathcal{I}_{\text{YES}}^Q$.

3.2.2. NP

A decision problem $Q = (\mathcal{I}_{\text{YES}}^Q, \mathcal{I}_{\text{NO}}^Q)$ is in NP if and only if there exists a deterministic Turing machine TM and a constant $const \in \mathbb{R}^+$ so that $\mathcal{I}_{\text{YES}}^Q = \{I \in \mathcal{I}^Q \mid \text{there exists a certificate } s \in \mathcal{S}^Q(I) \text{ with } |s| = O(|I|^{const}) \text{ such that } TM(I, s) = 1\}$.

3.3. Complexity Classes for Search Problems

3.3.1. FP

A search problem R is in FP if it is in FNP and if there exists a deterministic polynomial time algorithm that solves it, i.e. finds a solution s to a given instance I such that $(I, s) \in R$ or it states correctly that such a solution s does not exist [Yan88, p.28], [MP91].

FP is for search problems what P is for decision problems. The question whether $FNP = FP$ is equivalent to the question if $NP = P$ [Yan88].

3.3.2. FNP

A search problem R is in FNP

- if there is a polynomial time algorithm A that, given a pair (I, s) , determines in deterministic polynomial time whether or not (I, s) is in R , and
- if $(I, s) \in R$, then $|s|$ is polynomially bounded in $|I|$.

FNP is the equivalent of NP is for search problems [MP91; Yan88, p. 28].

3.3.3. TFNP

The class TFNP was first defined by [MP91] to capture *total* search problems. A problem is called *total* if there always exist a solution, i.e. $\forall I \in \mathcal{I}^R \exists s \in \mathcal{S}^R(I) : (I, s) \in R$.

TFNP is a subclass of FNP, containing only the total problems of FNP. TFNP is a semantic class which means that it is unlikely to have any complete problems.

3.3.4. PLS

The complexity class PLS (Polynomial Local Search) was first defined by [JPY88] to capture problems that can be solved by local search. The goal is to find a local minimum of function c , in a context where each candidate solution s has a local neighborhood in which the best neighbor can be calculated in polynomial time.

Definition 3.3.1 (The search problem SINK OF DAG). Given a Boolean circuit $S : \{0, 1\}^d \rightarrow \{0, 1\}^d$ such that $S(0^d) \neq 0^d$ and a circuit $c : \{0, 1\}^d \rightarrow \{0, 1, \dots, 2^m - 1\}$, find a vertex $s \in \{0, 1\}^d$ such that $S(s) \neq s$ and either $S(S(s)) = s$ or $c(S(s)) \leq c(s)$.

PLS is the class of all problems that can be reduced in polynomial time to SINK OF DAG [Fea+20b].

A problem in PLS can be interpreted as a possibly exponentially large directed acyclic graph where each node has an out-degree of at most one. Every sink is a local optimum of c , since it has by definition no better neighbor. The problems in PLS are total (and PLS is therefore in TFNP) because *every finite directed acyclic graph has a sink* [Pap94, p. 499].

3.3.5. PPAD

Definition 3.3.2 (The search problem END OF LINE). [Pap94, p. 506], [Fea+18, Definition 7] Given Boolean circuits $S, P : \{0, 1\}^d \rightarrow \{0, 1\}^d$ such that $P(0^d) = 0^d \neq S(0^d)$, find one of the following:

- (U1) A point $x \in \{0, 1\}^d$ such that $P(S(x)) \neq x$.
- (U2) A point $x \in \{0, 1\}^d$ such that $S(P(x)) \neq x$ and $x \neq 0^d$.

S stands for successor and P for predecessor function. The problem can be interpreted as possibly exponentially large directed graph $G = (\{0, 1\}^d, E)$ with $(v, u) \in E \Leftrightarrow (v \neq u \wedge S(v) = u \wedge P(u) = v)$. Each vertex has an in- and out-degree of at most one. Each connected component of the graph forms a line of vertices.

The parity argument states that *any finite graph has an even number of odd-degree nodes* [Pap94, p. 500]. 0^d is the standard start point and therefore a leaf, and since each node has in- and out-degree at most one, there must by the application of the parity argument exist at least one other leaf. This leaf is searched for as a solution to the problem. A solution of type (U1) describes the end of such a line. A solution of type (U2) describe the start of another line. For every node in the middle of the line it holds that $P(S(x)) = x$.

The class PPAD (which stands for polynomial parity argument in a directed graph) contains every problem that can be reduced in polynomial time to END OF LINE [Pap94, p. 506], [Fea+18].

PPAD contains several complete problems such as computing a Nash-equilibrium in two player games [DGP06] or the Brouwer fixpoint theorem [Pap94, p. 526].

There is no polynomial time algorithm known for PPAD-complete problems [Fea+20b].

3.3.6. CLS

The class CLS (Continuous Local Search) was first introduced in [DP11] and captures local search problems whose neighborhood function S and the cost (or potential) function c both have a continuous domain.

Definition 3.3.3 (The search problem KD-CONTINUOUS LOCAL OPT [Fea+20a]). Let $S : [0, 1]^k \rightarrow [0, 1]^k$ and $c : [0, 1]^k \rightarrow [0, 1]$ be two well behaved arithmetic circuits and let $\epsilon, \lambda > 0$ be two real constants. Find a local minimum (or maximum) $s \in [0, 1]^k$ such that

(C1) $c(S(s)) \geq c(s) - \epsilon$ for a minimization problem

($c(S(s)) \leq c(s) + \epsilon$ for the respective maximization problem)

(C2) or two points $s, s' \in [0, 1]^k$ that violate the λ -Lipschitz continuity of either S or c :

(C2a) $\|S(s) - S(s')\| > \lambda \|s - s'\|$

(C2b) $\|c(s) - c(s')\| > \lambda \|s - s'\|$

CLS is the class of all problems that can be reduced in polynomial time to KD-CONTINUOUS LOCAL OPT. [DP11] first defined the problem for $k = 3$, [Fea+20a] defined the more general variant and proved their equivalence.

The class contains all problems that seek an approximate local optimum (or in other words an approximate stationary or Karush-Kuhn-Tucker (KKT) point) [DP11].

There are several known CLS-complete problems, for example META METRIC CONTRACTION [Fea+17, Theorem 9.] and BANACH [DTZ17, Theorem 2].

CLS is in $(\text{PPAD} \cap \text{PLS})$ as shown by a reduction from 3D-CONTINUOUS LOCAL OPT to the PPAD-complete problem BROUWERFIXPOINT and the fact that 3D-CONTINUOUS LOCAL OPT is a special case of REALLOCALOPT, which is PLS-complete [DP11, Theorem 2.3]. [Fea+20a] proved recently that $\text{CLS} = \text{PPAD} \cap \text{PLS}$ by showing that the unnatural but $(\text{PPAD} \cap \text{PLS})$ -complete problem EITHER-(END OF LINE, ITER) can be reduced to KKT. KKT on the other hand can be reduced to another generalized version of GENERAL CONTINUOUS LOCAL OPT which is equivalent to 3D-CONTINUOUS LOCAL OPT. KKT, BANACH and META METRIC CONTRACTION therefore all are complete for CLS.

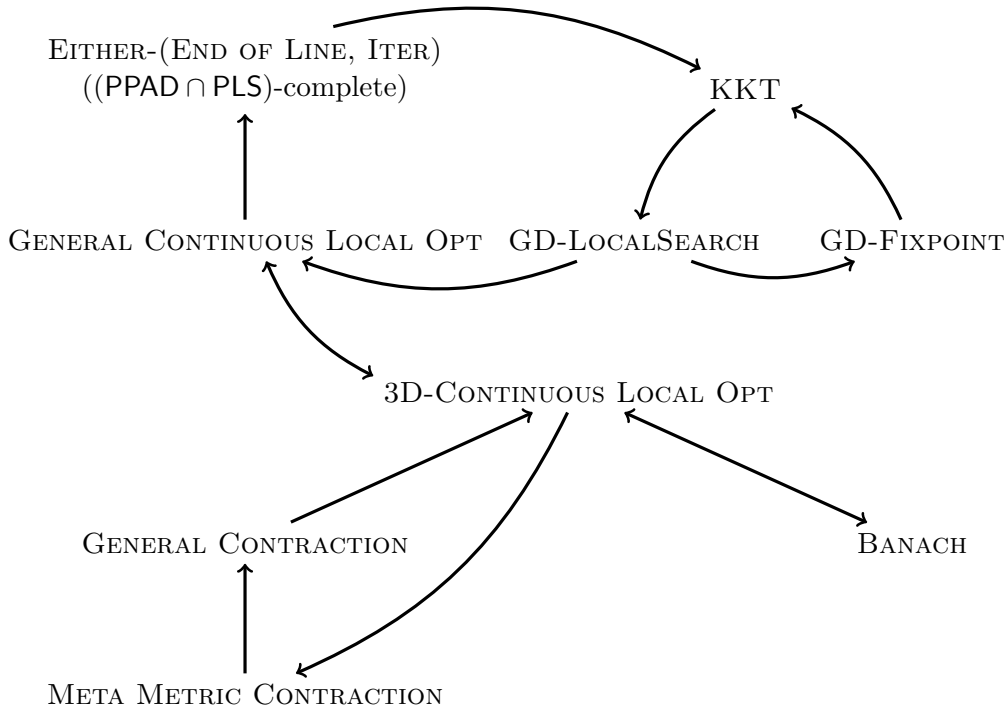


Figure 3.1.: Overview of reductions

3.3.7. EOPL

The class EOPL (End of potential line) describes problems in which it is possible to calculate the next solution as well as the previous solution from a given candidate solution in polynomial time. These calculations form a sequence of possible solutions, a *line* in which the solutions are ordered. Each solution also has a certain cost (or potential) which must be strictly increasing along the line.

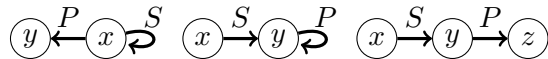
Definition 3.3.4 (The search problem END OF POTENTIAL LINE [Fea+20b]). Given the following Boolean circuits:

- $S, P : \{0, 1\}^d \rightarrow \{0, 1\}^d$ such that $P(0^d) = 0^d \neq S(0^d)$,
- $c : \{0, 1\}^d \rightarrow \{0, 1, \dots, 2^m - 1\}$ such that $c(0^d) = 0$.

The task is to find one of the following:

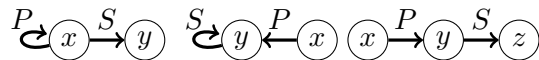
(U1) A point $x \in \{0, 1\}^d$ such that $P(S(x)) \neq x$.

x is the end of the line and therefore a valid solution. In this case, the structure of S and P might look like this:



(U2) A point $x \in \{0, 1\}^d$ such that $S(P(x)) \neq x \neq 0^d$.

x is the beginning of a different line which does not start at 0^d . It is a valid solution. In this case, the structure of S and P might look like this:



(U3) A point $x \in \{0, 1\}^d$ such that $P(S(x)) = x$, $S(x) \neq x$ and $c(S(x)) - c(x) \leq 0$.
 x is in the middle of a line, but the following vertex has lower cost. Therefore it is the end of the increasing line and a valid solution.

The problem can be interpreted as a possibly exponentially large directed graph $G = (V, E)$ with

- $v \in V \Leftrightarrow v \in \{0, 1\}^d \wedge (S(v) \neq v \vee P(v) \neq v)$ and
- $(v, u) \in E \Leftrightarrow (v \neq u \wedge S(v) = u \wedge P(u) = v)$.

Each vertex has an in- and out-degree of at most one. Each connected component of the graph forms a line of vertices. (U1) describes the end of such a line. For every node in the middle of the line it holds that $P(S(x)) = x$. The graph is acyclic. In comparison to UNIQUE END OF POTENTIAL LINE, there can be several lines with several endings in an END OF POTENTIAL LINE instance.

The problem combines the SINK OF DAG and END OF LINE problem. A solution is a sink, either because of increasing potential (like in SINK OF DAG) or because the successor and predecessor circuits don't form a valid edge for two nodes (like in END OF LINE).

The class EOPL contains all problems that can be reduced in polynomial time to END OF POTENTIAL LINE [Fea+18]. There are no known problems that are *in* EOPL but *not* in UniqueEOPL [Fea+20b, p. 3], the usefulness of the class EOPL is therefore still an open question.

Definition 3.3.5 (The search problem END OF METERED LINE [HY17]). Given the following Boolean circuits:

- $S, P : \{0, 1\}^d \rightarrow \{0, 1\}^d$ such that $P(0^d) = 0^d \neq S(0^d)$,
- $c : \{0, 1\}^d \rightarrow \{0, 1, \dots, 2^m\}$ such that $c(0^d) = 1$.

The task is to find one of the following:

- (U1) A point $x \in \{0, 1\}^d$ such that $P(S(x)) \neq x$.
 x is the end of the line and therefore a valid solution.
- (U2) A point $x \in \{0, 1\}^d$ such that $S(P(x)) \neq x \neq 0^d$.
 x is the beginning of a different line which does not start at 0^d . It is a valid solution.
- (U3) A point $x \in \{0, 1\}^d$ such that $x \neq 0^d$ and $c(x) = 1$.
 x is the beginning of a different line which does not start at 0^d . It is a valid solution.
- (U4) A point $x \in \{0, 1\}^d$ such that $c(x) > 0$ and $c(S(x)) - c(x) \neq 1$.
- (U5) A point $x \in \{0, 1\}^d$ such that $c(x) > 1$ and $c(x) - c(P(x)) \neq 1$.

[HY17] proved that END OF METERED LINE is in CLS.

The main difference from END OF POTENTIAL LINE and END OF METERED LINE is that in END OF METERED LINE the potential is increasing exactly by one. [Fea+18, Theorem 10] proved that END OF POTENTIAL LINE is polynomial time equivalent to END OF METERED LINE with the help of Lemma 4.1.26 on page 43.

It follows that END OF POTENTIAL LINE is in CLS and EOPL is a subclass of it. It is unknown whether CLS = EOPL.

3.3.8. UniqueEOPL

Like EOPL, the class UniqueEOPL (Unique End of Potential Line) captures search problems with the property that their solution space forms an exponentially large line with increasing cost. From one candidate solution we can calculate another candidate solution in polynomial time. The end of that line is the solution of the search problem. If the above properties hold as promised, this solution is unique. This is a *promise* problem that can be transformed into a total problem as described in Section 2.4.1 on page 9. For the cases where the cost are not increasing or the solution space does not form a single line, violations are introduced. This problem is therefore total.

Definition 3.3.6 (The search problem UNIQUE END OF POTENTIAL LINE [Fea+18]).
 $d, m \in \mathbb{N}^+, m \geq d$.

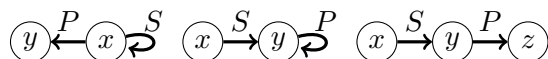
Given the following Boolean circuits:

- $S, P : \{0, 1\}^d \rightarrow \{0, 1\}^d$ such that $P(0^d) = 0^d \neq S(0^d)$ and
- $c : \{0, 1\}^d \rightarrow \{0, 1, \dots, 2^m - 1\}$ such that $c(0^d) = 0$.

The task is to find one of the following:

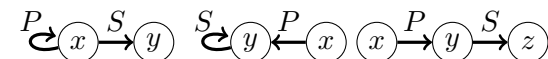
(U1) A point $x \in \{0, 1\}^d$ such that $P(S(x)) \neq x$.

x is the end of the line and therefore a valid solution. In this case, the structure of S and P might look like this:



(V1) A point $x \in \{0, 1\}^d$ such that $S(P(x)) \neq x \neq 0^d$.

x is the beginning of a different line which does not start at 0^d . In this case, the structure of S and P might look like this:



(V2) A point $x \in \{0, 1\}^d$ such that $P(S(x)) = x$, $x \neq S(x)$ and $c(S(x)) - c(x) \leq 0$.

x is in the middle of a line, but the following vertex has lower cost. It is a violation of the increasing potential.

(V3) Two points $x, y \in \{0, 1\}^d$ such that $x \neq y$, $x \neq S(x)$, $y \neq S(y)$ and either

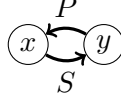
- $c(x) = c(y)$ or
- $c(x) < c(y) < c(S(x))$.

x and y are two different nodes that either have the same potential or that violate the increasing potential. Both cases imply that the instance has more than one line.

The problem intuitively defines a directed graph $G = (V, E)$, with

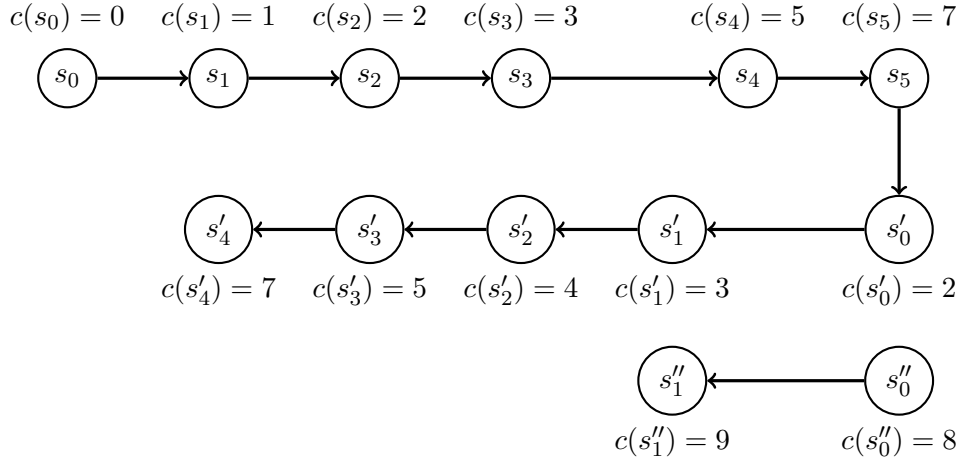
- $v \in V \Leftrightarrow v \in \{0, 1\}^d \wedge (S(v) \neq v \vee P(v) \neq v)$ and
- $(v, u) \in E \Leftrightarrow (v \neq u \wedge S(v) = u \wedge P(u) = v)$.

Each vertex has an in- and out-degree of at most one. Each connected component of the graph forms a line of vertices. (U1) describes the end of such a line. For every node in the middle of the line it holds that $P(S(x)) = x$, i.e. the structure of S and P look like this:



The given problem is a maximization problem, as well as all other problems examined in this thesis. An equivalent minimization problem can be constructed by defining $c'(s) := 2^m - c(s)$ and inverting the above conditions respectively.

Example 3.3.7. A UNIQUE END OF POTENTIAL LINE instance with violations [Fea+18, Figure 2].



There are two lines in this picture, therefore it contains several violations. The first line starts at s_0 and the second line starts at s''_0 .

s''_1 is a solution of type (U1), as well as s'_4 .

s_5 is a violation of type (V2). The successor of s_5 is s'_0 , which has a lower potential than s_5 . Even though $S(s_5) = s'_0$ and $P(s'_0) = s_5$, it does not hold that $c(s_5) < c(s'_0)$.

s''_0 is a violation of type (V1).

s_2 and s'_0 are a violation of type (V3a) since $c(s'_0) = 2 = c(s_2)$.

s_3 and s'_2 are a violation of type (V3b). $S(s_3) = s_4$ and $c(s_3) = 3 < c(s'_2) = 4 < c(s_4) = 5$.

The complexity class UniqueEOPL contains all problems that can be reduced in polynomial time to UNIQUE END OF POTENTIAL LINE [Fea+20b].

Note that the reduction does not need to be promise preserving. It could happen that a problem can be reduced to UNIQUE END OF POTENTIAL LINE with a simple search problem reduction, which is sufficient to show that the problem is in UniqueEOPL. All problems in this thesis (except P-GLCP) can be reduced under promise preserving reductions though, which implies that their promise versions are in PromiseUEOPL.

In contrast to the class EOPL it follows from the uniqueness that only one line is allowed here. The goal is to find the end of that unique line or a violation of the line-structure or the increasing cost. Circles are not allowed and the start of the line is always 0^d . UniqueEOPL is by definition a subclass of EOPL, they are considered unlikely to be equal [Fea+20b].

3.3.9. Relationships between Classes

[MP91, p. 319]:	$PLS \subseteq TFNP$
[Pap94]	$PPAD \subseteq TFNP$
[DP11, Theorem 4.1]:	$CLS \subseteq PLS \cap PPAD$
[Fea+20a]:	$CLS = PLS \cap PPAD$
[Fea+18, Corollary 11.]:	$EOPL \subseteq CLS$
[Fea+18, p.14]:	$UniqueEOPL \subseteq EOPL$

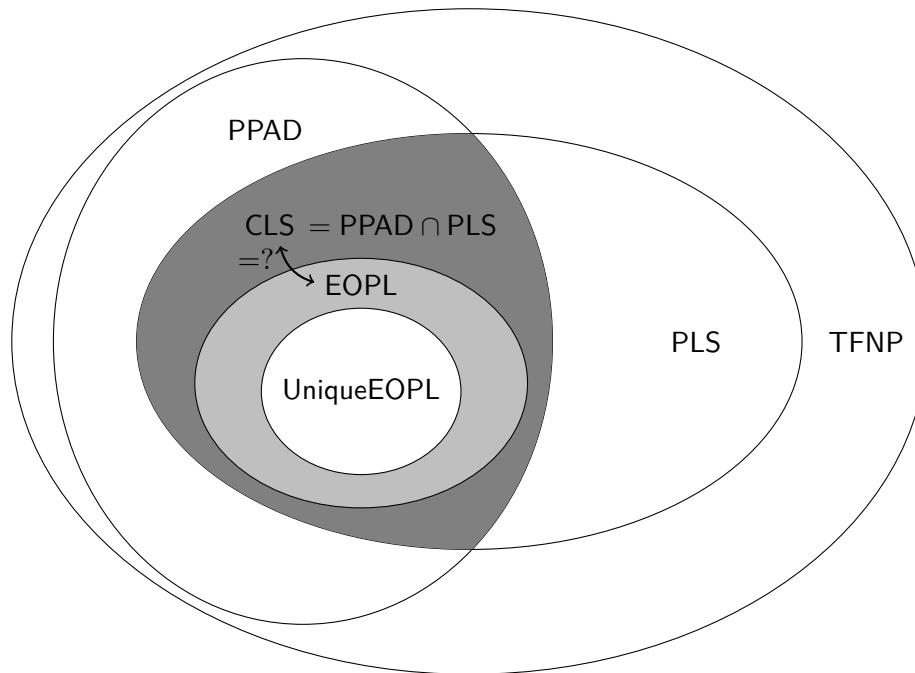


Figure 3.2.: Relationships between classes

3.4. Complexity Classes for Promise Problems

3.4.1. Promise-P

Promise-P is the class of promise problems that are solvable in deterministic polynomial time. It is the promise problem equivalent of the decision problem class P.

Definition 3.4.1 (Promise-P [Gol06]). A promise problem $\Psi = (\mathcal{I}_{\text{YES}}^{\Psi}, \mathcal{I}_{\text{NO}}^{\Psi})$ is in Promise-P if there exists a polynomial time algorithm A such that:

- $\forall I \in \mathcal{I}_{\text{YES}}^{\Psi} : A(I) = 1,$
- $\forall I \in \mathcal{I}_{\text{NO}}^{\Psi} : A(I) = 0.$

3.4.2. Promise-NP

Promise-NP is the class of promise problems that have polynomial long proofs of membership that are verifiable in polynomial deterministic time. It is the promise problem equivalent of the decision problem class NP.

Definition 3.4.2 (Promise-NP [Gol06]). A promise problem $\Psi = (\mathcal{I}_{\text{YES}}^{\Psi}, \mathcal{I}_{\text{NO}}^{\Psi})$ is in Promise-NP if there exists a polynomial bounded relation $R: \{0, 1\}^* \rightarrow \{0, 1\}^*$ that is recognized by a polynomial time algorithm such that:

- $\forall I \in \mathcal{I}_{\text{YES}}^{\Psi} : \text{there exists a solution } s \text{ such that } (I, s) \in R.$
- $\forall I \in \mathcal{I}_{\text{NO}}^{\Psi} : \text{there exists no solution } s \text{ such that } (I, s) \in R.$

We say that R is polynomial time recognizable if there exists a polynomial p such that for every $(I, s) \in R$ it holds that $|s| \leq p(|I|)$ and R is recognized by an algorithm A if $A(I, s) = 1 \Leftrightarrow (I, s) \in R$.

3.4.3. PromiseUEOPL

Definition 3.4.3 (PROMISE UNIQUE END OF POTENTIAL LINE). Recall the Definition 3.3.6 on page 19 of UNIQUE END OF POTENTIAL LINE. The problem PROMISE UNIQUE END OF POTENTIAL LINE is defined as: Under the promise that there is no violation of type (V1), (V2) and (V3), find a solution of type (U1).

The class PromiseUEOPL contains all promise problems that can be reduced in polynomial time to PROMISE UNIQUE END OF POTENTIAL LINE [Fea+18, p. 3].

Problems in UniqueEOPL are *not* guaranteed to have unique solutions since there might be violations. Problems in PromiseUEOPL *have* a unique solution, since they are promised not to have violations.

PromiseUEOPL is the promise problem equivalent of the decision problem class UniqueEOPL. The promise version of the problems that are proven to be in UniqueEOPL via a *promise preserving* reduction are in PromiseUEOPL.

4. UniqueEOPL-complete Problems

UNIQUE END OF POTENTIAL LINE is trivially complete for UniqueEOPL. There is currently only one "natural" complete problem known for UniqueEOPL, which is ONE PERMUTATION DISCRETE CONTRACTION (OPDC). Several variations of UniqueEOPL that are used for proving OPDC to be UniqueEOPL-complete are also UniqueEOPL-complete. To prove a problem to be UniqueEOPL-complete, it needs to be shown that it is in UniqueEOPL and that another complete problem can be reduced to the new one.

Complete Problem	Proven by	Reference
UNIQUE END OF POTENTIAL LINE	definition	
UNIQUE FORWARD EOPL	\succeq_{PROMISE} ONE PERMUTATION DISCRETE CONTRACTION (OPDC)	[Fea+20b, Lemma 11]
UNIQUE EOPL+1	\succeq_{PROMISE} UNIQUE END OF POTENTIAL LINE	[Fea+20b, Lemma 17]
OPDC	\succeq_{PROMISE} UNIQUE EOPL+1	[Fea+20b, Lemma 19]
UNIQUE FORWARD EOPL+1	\succeq_{PROMISE} UNIQUE FORWARD EOPL	[Fea+20b, Lemma 13]

4.1. One Permutation Discrete Contraction (OPDC)

Definition 4.1.1 (CONTRACTION [Fea+18]). $C := [0, 1]^d \subset \mathbb{R}^d$ is a hyper cube. A contraction map is a function $f : [0, 1]^d \rightarrow [0, 1]^d$. It is *contracting* under a metric δ if $\forall x, y \in [0, 1]^d : \delta(f(x), f(y)) \leq \text{const} \cdot \delta(x, y)$ for some constant $0 < \text{const} < 1$.

The Banach fixed point theorem states that for a function f that is contracting under a metric δ , there exists a unique fixpoint of f , i.e. $f(x) = x$ [Bar11, p. 146].

The contraction map is defined on a continuous space and [DTZ17] proved that finding the fixpoint whose existence is guaranteed by the Banach fixed point theorem, is CLS-complete. Therefore the problem is unlikely to be in UniqueEOPL since this would imply that $\text{CLS} = \text{UniqueEOPL}$. In the following, a discrete (and rather specific) version of the contraction problem is examined.

The cube C is discretized with an overlying grid Γ with grid widths $(\omega_1, \dots, \omega_d)$ where $\omega_i \in \mathbb{N}$:

$$\Gamma := [0, 1, \dots, \omega_1] \times [0, 1, \dots, \omega_2] \times \dots \times [0, 1, \dots, \omega_d].$$

A point in the grid $p \in \Gamma$ is matched to the cube by dividing each dimension p_i by the corresponding grid width ω_i :

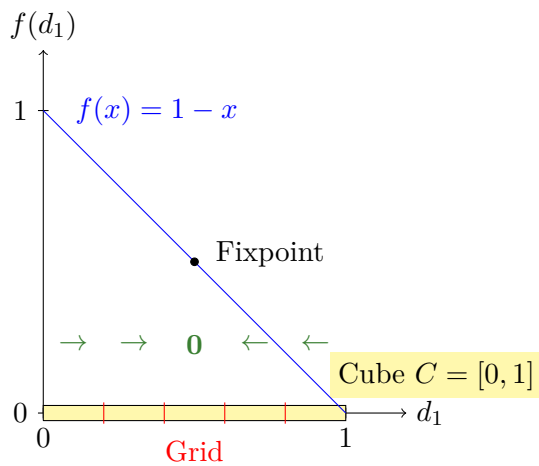
$$q := (p_1/\omega_1, p_2/\omega_2, \dots, p_d/\omega_d) \in [0, 1]^d.$$

Definition 4.1.2 (Direction functions). Let $\mathcal{D} := \{D_i \mid i = 1, \dots, d\}$ be a set of direction functions given as circuit with $D_i : \Gamma \rightarrow \{\text{Up}, \text{Down}, \text{Zero}\}$ so that for every point $p \in \Gamma$:

- $D_i(p) = \text{Up} \Leftrightarrow f(q)_i > q_i$,
- $D_i(p) = \text{Down} \Leftrightarrow f(q)_i < q_i$,
- $D_i(p) = \text{Zero} \Leftrightarrow f(q)_i = q_i$.

The goal of the modified problem is to find a point $p \in \Gamma$ that is a fixpoint for each direction function: $\forall i = 1, \dots, d : D_i(p) = \text{Zero}$.

Example 4.1.3. $d = 1$

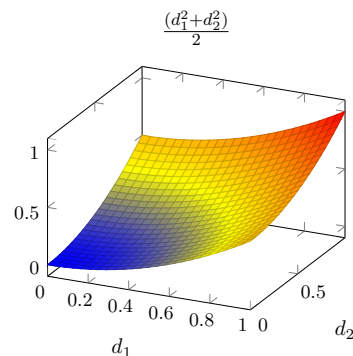
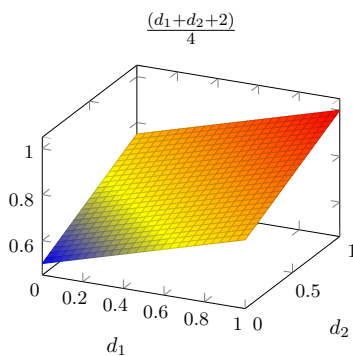


- Dimension $d := 1$
- Cube $C = [0, 1]$
- Grid width $\omega_1 := 4$
- Grid $\Gamma := \{0, 1, 2, 3, 4\}$
- Direction function (represented by the arrows):
 - $f(0) = 1 > 0 \Rightarrow D_1(0) = \text{Up}$
 - $f(\frac{1}{4}) = \frac{3}{4} > \frac{1}{4} \Rightarrow D_1(1) = \text{Up}$
 - $f(\frac{2}{4}) = \frac{2}{4} = \frac{2}{4} \Rightarrow D_1(2) = \text{Zero}$
 - $f(\frac{3}{4}) = \frac{1}{4} < \frac{3}{4} \Rightarrow D_1(3) = \text{Down}$
 - $f(1) = 0 < 1 \Rightarrow D_1(4) = \text{Down}$

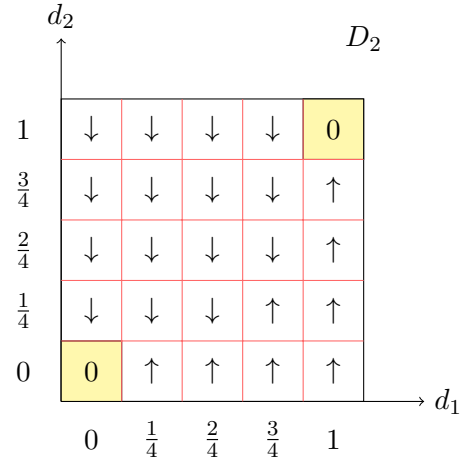
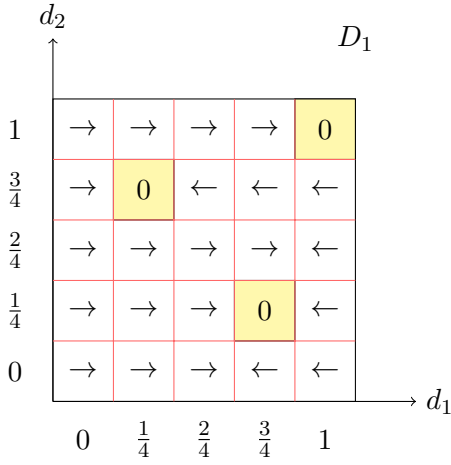
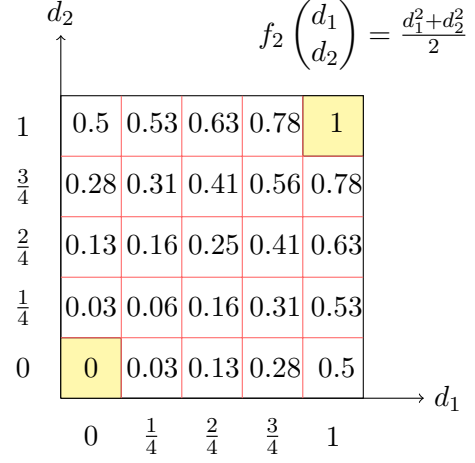
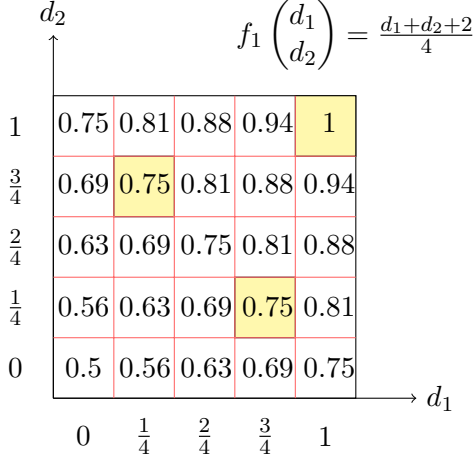
Example 4.1.4. $d = 2$

$$f : [0, 1]^2 \rightarrow [0, 1]^2$$

$$f \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} = \begin{pmatrix} \frac{(d_1 + d_2 + 2)}{4} \\ \frac{(d_1^2 + d_2^2)}{2} \end{pmatrix}$$



The goal is to find a point (p_1, p_2) that is fixpoint in both of the above plots. The solution is obviously $(1, 1)$.



Note that up until now there might not be a unique fixpoint in the grid. It can happen that the cube is discretized in an unfortunate way so that the fixpoint is not captured. To ensure that there is a fixpoint, extra conditions need to be formulated. In order to do so, we need the following definitions first.

Definition 4.1.5 (Slice \mathbf{s} [Fea+18]). Let $\text{Slices}_d := [0, 1, \dots, \omega_1] \cup \{*\} \times [0, 1, \dots, \omega_2] \cup \{*\} \times \dots \times [0, 1, \dots, \omega_d] \cup \{*\}$.

A slice is a vector $\mathbf{s} = (s_1, s_2, \dots, s_d) \in \text{Slices}_d$ with $s_i \in [0, 1, \dots, \omega_i] \cup \{*\}$. If $s_i \in [0, 1, \dots, \omega_i]$, this means that dimension i is fixed, if $s_i = *$, dimension i is free to vary.

For $\mathbf{s} \in \text{Slices}_d$ let $\Gamma_{\mathbf{s}} := \{p \in \Gamma \mid \forall i = 1, \dots, d : s_i \neq * \Rightarrow p_i = s_i\}$ be the set of grid points in the slice \mathbf{s} .

Definition 4.1.6 (Sub-Slices [Fea+18]). A slice $\mathbf{s}' \in \text{Slices}_d$ is a sub slice of $\mathbf{s} \in \text{Slices}_d$ if

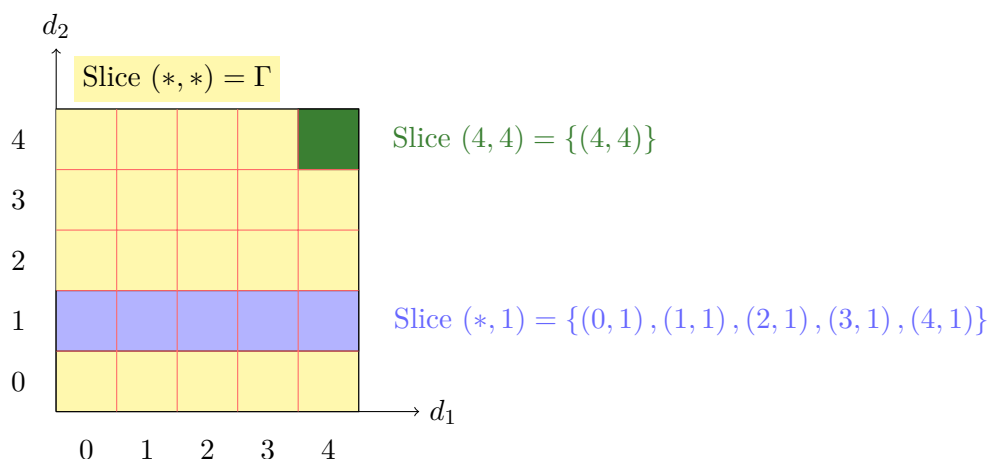
$$\forall j = 1, \dots, d : s_j \neq * \Rightarrow s'_j = s_j.$$

Definition 4.1.7 (i -Slice [Fea+18]). An i -Slice is a slice \mathbf{s} for which holds:

$$\forall j = 1, \dots, d : (j \leq i \Rightarrow s_j = *) \wedge (j > i \Rightarrow s_j \neq *).$$

Example 4.1.8. Recall Example 4.1.4 on the previous page. There are 3 i – Slice types:

- one 2 – Slice, which is $(*, *)$,
- for each $y \in [0, 1, \dots, \omega_2]$ there is one 1 – Slice $(*, y)$,
- for each $x \in [0, 1, \dots, \omega_1]$, $y \in [0, 1, \dots, \omega_2]$ there is one 0 – Slice (x, y) , which is exactly one point in the grid.



Definition 4.1.9 (One-permutation discrete contraction map [Fea+18]). Let Γ be a grid and let $\mathcal{D} := \{D_i \mid i = 1, \dots, d\}$ be a family of direction functions. \mathcal{D} and Γ form a *one-permutation discrete contraction map* if for every i – Slice \mathbf{s} the following two conditions hold:

- There is a unique fixpoint of \mathbf{s} in all dimensions smaller than i (i.e. a point $p^* \in \Gamma_{\mathbf{s}}$ is fixpoint of slice \mathbf{s} if $D_i(p^*) = \text{Zero}$ for all dimensions i where $\mathbf{s}_i = *$).
- Let $\mathbf{s}' \in \text{Slices}_d$ be a sub-slice of \mathbf{s} where the coordinate i for which $\mathbf{s}_i = *$ has been fixed to some value and all other coordinates are unchanged. \mathbf{s}' is an $(i - 1)$ – Slice. If p^* is the unique fixpoint of \mathbf{s} , and q^* is the unique fixpoint of \mathbf{s}' , then
 - $q_i^* < p_i^* \Rightarrow D_i(q^*) = \text{Up}$ and
 - $q_i^* > p_i^* \Rightarrow D_i(q^*) = \text{Down}$.

Since $(*, \dots, *)$ is an i – Slice, the first condition promises that there is a unique fixpoint for the overall problem. The second condition promises that if we have found a unique fixpoint for a sub-slice \mathbf{s}' , then the direction function will tell us where to go to find the unique fixpoint of \mathbf{s} .

Note that these two conditions are rather specific and not really relevant in practice.

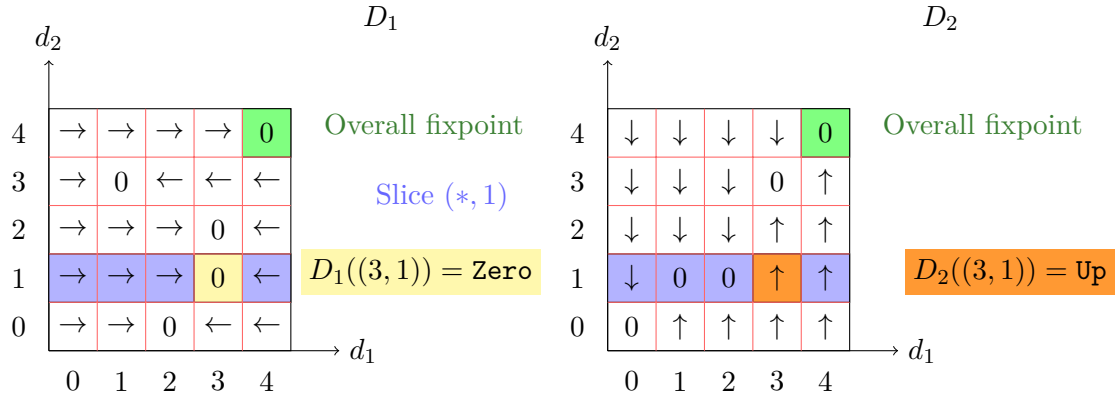
It is hard to check whether a given instance fulfills the conditions or not. These conditions are the promise. In the search version of the problem they are formulated as violations.

Example 4.1.10. (First condition)

Recall Example 4.1.4 on page 24. It does not satisfy the first condition. Not every i – Slice has a unique fixpoint, the slice $(*, 2)$ for example has none.

The following image shows a function which *does* satisfy the first condition. The 1–Slice $(*, 1)$ has a unique fixpoint in dimension d_1 (picture on the left) at $(3, 1)$. For the second

dimension d_2 (picture on the right) there are two fixpoints of D_2 in that slice, but $s_2 \neq *$ and therefore there doesn't need to be a unique fixpoint for D_2 .



Example 4.1.11. (Second condition)

Let $\mathbf{s} := (*, *)$ and let \mathbf{s}' be sub-slice of \mathbf{s} which is fixed in the coordinate $i = 2$: $\mathbf{s}' := (*, 1)$.

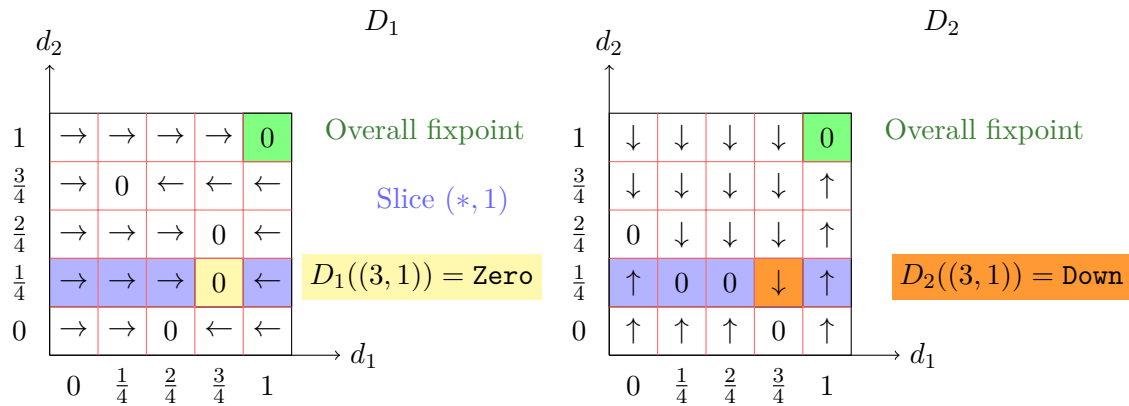
$p^* = (4, 4)$ is a unique fixpoint of \mathbf{s} and $q^* = (3, 1)$ is a unique fixpoint of \mathbf{s}' .

Given the second condition, the following should hold:

$$q_2^* = 1 < 4 = p_2^* \Rightarrow D_2(q^*) = \text{Up}.$$

In Example 4.1.10 on the previous page, this is the case. The orange colored box points upwards in the direction of the overall fixpoint of the problem.

Consider the following image. Here, $D_2(q^*)$ doesn't point towards the overall fixpoint. This simply means that the given example is no valid one-permutation discrete contraction map.



Definition 4.1.12 (The search problem ONE PERMUTATION DISCRETE CONTRACTION (OPDC)). Let Γ be a discrete grid points over $[0, 1]^d$ and let \mathcal{D} be a family of direction functions as defined in Definition 4.1.2 on page 24.

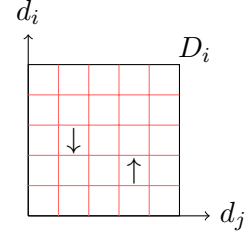
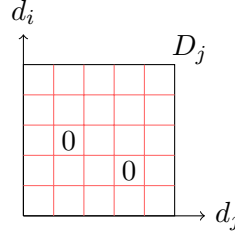
Find one of the following:

(O1) A point $p \in \Gamma$ such that $\forall i = 1, \dots, d : D_i(p) = \text{Zero}$.

(OV1) An i – Slice \mathbf{s} with two fixpoints $p, q \in \Gamma_{\mathbf{s}}$ with $p \neq q$ such that $\forall j \leq i : D_j(p) = D_j(q) = \text{Zero}$.

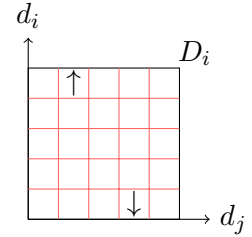
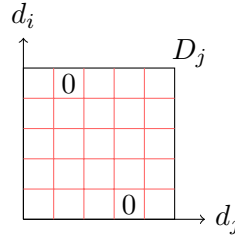
(OV2) An i – Slice \mathbf{s} and two points $p, q \in \Gamma_{\mathbf{s}}$ such that

- $\forall j < i : D_j(p) = D_j(q) = \text{Zero}$,
- p_i and q_i are adjacent to each other in dimension i :
 $p_i = q_i + 1$,
- $D_i(p) = \text{Down}$ and $D_i(q) = \text{Up}$.



(OV3) An i – Slice \mathbf{s} and a point $p \in \Gamma_{\mathbf{s}}$ such that

- $\forall j < i : D_j(p) = \text{Zero}$ and either
- $p_i = 0$ and $D_i(p) = \text{Down}$ or
- $p_i = \omega_i$ and $D_i(p) = \text{Up}$.



Solution (O1) describes a fixpoint.

Violation (OV1) is clearly a violation of condition (ii) of the one permutation discrete contraction map from Definition 4.1.9 on page 26.

Violation (OV2) is a violation of condition (ii) of the one permutation discrete contraction map. By condition (ii) both arrows should point towards the fixpoint of the next higher dimension. But when they are pointing in disagreeing directions, they cannot both point to the fixpoint.

Violation (OV3) is also a violation of condition (ii) of the one permutation discrete contraction map. By condition (ii) the arrow should point towards the fixpoint of the next higher dimension, but this fixpoint cannot be somewhere out of the grid, therefore the arrow must be wrong.

The conditions of the one-permutation discrete contraction map as defined in Definition 4.1.9 on page 26 are here formulated as violations. They are the reason why this problem is called *one permutation* discrete contraction. The order of the dimensions we chose is relevant to the validity of the instance. If we chose a different order, it is a different problem.

Example 4.1.13. The following example is a perfectly valid OPDC instance:



If the two dimensions are flipped, the points $(0,1)$ and $(2,0)$ violate of type $(OV2)$.



The problem has no natural cost or potential function. Also the points of the grid don't have an intuitive neighborhood or successor as needed for a UNIQUE END OF POTENTIAL LINE instance. Nonetheless, there exists a reduction from OPDC to UNIQUE END OF POTENTIAL LINE which proves that OPDC is in UniqueEOPL.

The two conditions defined for a one-permutation discrete contraction map in definition 4.1.9 on page 26 allow us to define a line-following algorithm along which we can walk through an OPDC instance. We start with 0^d - the grid cell in the bottom left corner. From there, the direction function shows us the way to the fixpoint of dimension i . When reached, we can check with the direction function of dimension $i + 1$ which direction to go in the next dimension. So we reset dimension i to 0, move along the told direction and start increasing the coordinates in dimension i again. When the mutual fixpoint of dimension i and $i + 1$ is found, the direction function of $i + 2$ shows us the way to go, and so on, until the overall fixpoint is found.

4.1.1. OPDC is in UniqueEOPL

Theorem 4.1.14. ONE PERMUTATION DISCRETE CONTRACTION (OPDC) is in UniqueEOPL.

It is not proven directly that $OPDC \preceq_{PROMISE} UNIQUE\ END\ OF\ POTENTIAL\ LINE$, but instead a sequence of reductions is performed:

$$\begin{aligned}
 OPDC &\preceq_{PROMISE} UNIQUE\ FORWARD\ EOPL \\
 &\preceq_{PROMISE} UNIQUE\ FORWARD\ EOPL+1 \\
 &\preceq_{PROMISE} UNIQUE\ END\ OF\ POTENTIAL\ LINE.
 \end{aligned}$$

4.1.1.1. OPDC to Unique Forward EOPL

Definition 4.1.15 (UNIQUE FORWARD END OF POTENTIAL LINE [Fea+20b]). Given a Boolean circuit $S : \{0, 1\}^d \rightarrow \{0, 1\}^d$ such that $S(0^d) \neq 0^d$ and a Boolean circuit $c : \{0, 1\}^d \rightarrow \{0, 1, \dots, 2^m - 1\}$ such that $c(0^d) = 0$, find one of the following:

- (UF1) A point $x \in \{0, 1\}^d$ such that $S(x) \neq x$ and either $S(S(x)) = S(x)$ or $c(S(x)) \leq c(x)$.
 x is a sink and a valid solution.
- (UFV1) Two points $x, y \in \{0, 1\}^d$ such that $x \neq y, S(x) \neq x, S(y) \neq y$ and either
 - a) $c(x) = c(y)$ or
 - b) $c(x) < c(y) < c(S(x))$.

x and y are two different nodes that either have the same potential, or that violate the increasing potential. Both cases imply that the instance does not have the form of an increasing line.

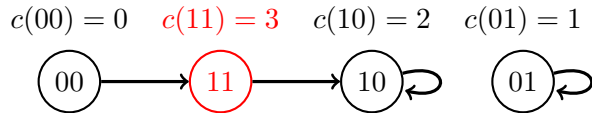
- (UFV2) Two points $x, y \in \{0, 1\}^d$ such that x is a solution of type (UF1), $x \neq y, S(y) \neq y$ and $c(x) < c(y)$.

This encodes a break in the line. x is the end of one line, but there exists a different line with a node y that has higher cost than x .

Again, this problem can be interpreted as a directed acyclic graph $G = (V, E)$ with $V := \{v \mid v \in \{0, 1\}^d \wedge S(v) \neq v\}$ and $(v, u) \in E \Leftrightarrow (v \neq u \wedge S(v) = u \wedge c(v) < c(u))$. A bit string v only encodes a vertex if $S(v) \neq v$. Each vertex has out-degree at most one. (UF1) describes a sink of the graph. If no violation of type (UFV1) is found, it implies that the graph actually has the form of a line. So if a sink is found and if there exists no violation, the sink is the end of a line.

(UFV2) implies that there is another line with a node that has bigger cost than the end of the first line. This type of solution might seem a bit pointless at first, since when any algorithm finds a solution of type (UF1), it is happy and stops searching for violations. This type of solution is needed to make the reductions promise preserving. UNIQUE FORWARD EOPL+1 has a violation of that type as well. It becomes important in the reduction from UNIQUE FORWARD EOPL+1 to UNIQUE END OF POTENTIAL LINE, when there is a violation found in the UNIQUE END OF POTENTIAL LINE instance, it must be matched back to a violation in the UNIQUE FORWARD EOPL+1 instance. This can be seen in detail in [Fea+20b, Proof of Lemma 27, p. 28].

Example 4.1.16. Consider the following UNIQUE FORWARD EOPL instance for $d = 2$. The arrows represent function S , not the edges of the graph. The vertices (10) and (01) aren't actual vertices. $V = \{00, 11\}$, $E = \{(00, 11)\}$. (11) is a solution of type (UF1).



Theorem 4.1.17. *There exists a promise preserving polynomial time reduction from OPDC to UNIQUE FORWARD EOPL.*

Proof. Given an OPDC instance $I = (\Gamma, \mathcal{D})$, construct a UNIQUE FORWARD EOPL instance $I' = (S, c)$.

The idea of the reduction is to find a line following algorithm through the OPDC grid as described in Example 4.1.13 on page 29, which ends in the unique fixpoint. The algorithm works recursively. The states of this local search line following algorithm are represented as vertices of UNIQUE FORWARD EOPL. The nodes of the UNIQUE FORWARD EOPL instance are tuples with certain properties in which the unique fixpoint of all i dimensions is saved, before we start searching for the fixpoint of $i + 1$ dimensions. This ensures that every point *after* the unique fixpoint is not a valid vertex. Furthermore it also allows us to define an order on the vertices, which is needed to define the cost function. Every tuple can be evaluated by a set of polynomial time verifiable conditions.

For the correctness of the reduction it must be proven that valid solutions of OPDC are only mapped to valid solutions of UNIQUE FORWARD EOPL, and that every violation in the constructed UNIQUE FORWARD EOPL instance was a violation in the original OPDC instance in the first place.

A point $p \in \Gamma$ is on the i – **surface** if $\forall j = 1, \dots, i: D_j(p) = \text{Zero}$.

The vertices The dimension of the grid Γ is d . Let $V := (\Gamma \cup \{\perp\})^{d+1}$ be the set of vertices for the UNIQUE FORWARD EOPL instance. Since $\Gamma = [0, 1, \dots, \omega_1] \times [0, 1, \dots, \omega_2] \times \dots \times [0, 1, \dots, \omega_d]$ does not consist of 0-1-bit strings, this is actually not a set that can be used for the construction of S . Let k_i be the number of bits needed to represent $p_i \in [0, 1, \dots, \omega_i]$ and let $k := \max_{i=1, \dots, d} k_i$. Let $\Gamma' := \{0, 1\}^{k \times d}$ and $V' := (\Gamma' \cup \{\perp\})^{d+1}$.

To make it easier to understand, the rest of the reduction will work with V . But it is important to note that all the intervals *can* be represented as 0-1-bit Strings.

A vertex v on the line of I' is a vector $v := (p_0, p_1, \dots, p_d)$ with each $p_i \in \Gamma \cup \{\perp\}$. If a point $p_i = \perp$, it indicates that we have not encountered a point on the i – surface yet. If point $p_i \neq \perp$, p_i is the value of the most recent visited point on the i – surface.

Only some of the vectors are valid vertices. To be valid, a vertex $v = (p_0, p_1, \dots, p_d)$ must fulfill all of the following conditions:

- for $i = 1, \dots, d$:
 - (a) $p_i \neq \perp \Rightarrow \forall j = 1, \dots, i: D_j(p_i) = \text{Zero}$.
This means that if p_i is not bottom, p_i has to be on the i – surface of the OPDC instance.
- for $i = 0, \dots, d - 1$:
 - (b) $p_i \neq \perp \Rightarrow D_{i+1}(p_i) \neq \text{Down}$.
 - (c) $p_i \neq \perp \wedge p_j = \perp \wedge i < j \Rightarrow (p_i)_{j+1} = 0$.
 - (d) $p_i \neq \perp \wedge p_j \neq \perp \wedge i < j \Rightarrow (p_i)_{j+1} = (p_j)_{j+1} + 1$.

The function $\text{isVertex} : V \rightarrow \{\text{true}, \text{false}\}$ returns whether or not v fulfills all these conditions. The function can be calculated in polynomial time.

Example 4.1.18. $d = 6$

$$\begin{array}{c}
 \begin{array}{cccccc}
 & & & = 0 & & = (p_5)_6 + 1 \\
 & & & \boxed{(p_0)_4} & (p_0)_5 & \boxed{(p_0)_6} \\
 & & & \boxed{(p_1)_4} & (p_1)_5 & \boxed{(p_1)_6} \\
 & & & \boxed{(p_2)_4} & (p_2)_5 & \boxed{(p_2)_6} \\
 p_3 = \perp & \boxed{(p_3)_1} & \boxed{(p_3)_2} & \boxed{(p_3)_3} & (p_3)_4 & (p_3)_5 & (p_3)_6 \\
 & \boxed{(p_4)_1} & \boxed{(p_4)_2} & \boxed{(p_4)_3} & \boxed{(p_4)_4} & \boxed{(p_4)_5} & \boxed{(p_4)_6} \\
 p_5 \neq \perp & \boxed{(p_5)_1} & \boxed{(p_5)_2} & \boxed{(p_5)_3} & \boxed{(p_5)_4} & \boxed{(p_5)_5} & \boxed{(p_5)_6} \\
 & (p_6)_1 & (p_6)_2 & (p_6)_3 & (p_6)_4 & (p_6)_5 & (p_6)_6
 \end{array}
 \end{array}$$

The above matrix represents a vertex $v \in V$. The column colored in blue is an example of condition (c): $p_3 = \perp$, therefore all points above which are not \perp are 0 at index 4. The column in red is an example of condition (d): Since $p_5 \neq \perp$, all points above which are not \perp are $(p_5)_6 + 1$ at index 6.

The start vertex o is defined as $o := (0^d, \perp, \dots, \perp)$.

Construction of the successor function S Let $S: V \rightarrow V$.

Given a vertex $v = (p_0, p_1, \dots, p_d)$, construct $S(v)$:

(1) If $\text{isVertex}(v) = \text{false}$ then $S(v) := v$ to indicate that v is not a valid vertex.

(2) If $\text{isVertex}(v) = \text{true}$, let i be the smallest index such that $p_i \neq \perp$.

(2.1) If $i = d$ then $v = (\perp, \dots, \perp, p_d)$. Furthermore, p_d is on the d – **surface** which means that $\forall j \leq d: D_j(p) = \text{Zero}$. Therefore, p_d is a solution to the discrete contraction map. We set $S(v) := v$ to indicate that the point before v is the end of the line.

(2.2) If $D_{i+1}(p_i) = \text{Zero}$ and $i < d$ then $S(v) := u$ where

$$u_j = \begin{cases} \perp & \text{if } j < i + 1 \\ p_i & \text{if } j = i + 1 \\ p_j & \text{if } j > i + 1. \end{cases}$$

This overwrites the point at position $i + 1$ with p_i and sets the position i to \perp . All other components are unchanged.

$$S(\perp, \dots, \perp, p_i, p_{i+1}, p_{i+2}, \dots, p_d) = (\perp, \dots, \perp, \perp, p_i, p_{i+2}, \dots, p_d).$$

This case describes the situation when a point is reached which is **Zero** in all dimensions $j \leq i + 1$. In the next step we want to check the next higher dimension.

(2.3) If $D_{i+1}(p_i) \neq \text{Zero}$ and $0 < i < d$ then let q be the point

$$(q)_j = \begin{cases} 0 & \text{if } j < i + 1 \\ (p_i)_{i+1} + 1 & \text{if } j = i + 1 \\ (p_i)_j & \text{if } j > i + 1. \end{cases}$$

(a) If q is a point in Γ then $S(v) = (q, p_1, \dots, p_d)$.

In this case we follow the direction which tells us where the overall fixpoint is and reset the lower dimensions.

(b) Otherwise it holds that $(p_i)_{i+1} = \omega_{i+1}$. p_i is the last point on the grid in dimension $i + 1$. This implies a solution of type *(OV3)* since $\text{isVertex}(v) = \text{true}$ implies that $D_{i+1}(p_i) = \text{Up}$. We set $S(v) = v$.

(2.4) If $D_{i+1}(v_i) \neq \text{Zero}$ and $i = 0$ then let q be the point such that $(q)_j = (p_0)_j$ for all $j > 1$ and $(q)_1 = (p_0)_1 + 1$.

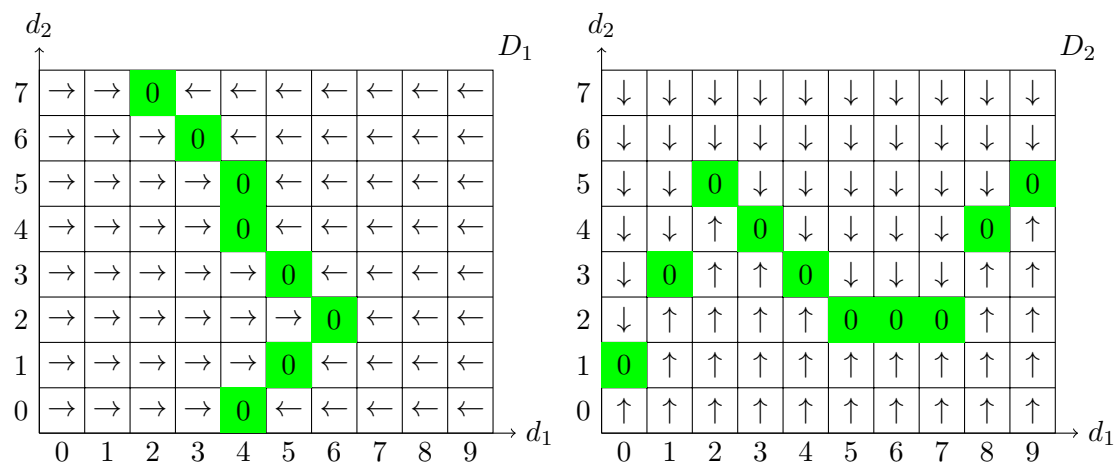
(a) If q is a point in Γ then $S(v) = (q, p_1, \dots, p_d)$.

We follow the direction of dimension 1 to find the fixpoint of this slice.

(b) Otherwise we have another solution of type *(OV3)*, since $(p_0)_1 = \omega_1$. We set $S(v) = v$.

Case (2.4) actually behaves the same as case (2.3) when it is called with $i = 0$, but when we prove the correctness of the reduction the two cases lead to different violations. Therefore they are kept separate.

Example 4.1.19. Consider the following OPDC instance for $d = 2$ from [Fea+18]. The table below shows the repeated call of S on the start node o . It contains the case that is entered to determine the successor and the result.



Vertex v	i	$D_{i+1}(p_i)$	Case	$S(v)$
$(00, \perp, \perp)$	0	$D_1(00) = \text{Up}$	$(4a)$	$(10, \perp, \perp)$
$(10, \perp, \perp)$	0	$D_1(10) = \text{Up}$	$(4a)$	$(20, \perp, \perp)$
$(20, \perp, \perp)$	0	$D_1(20) = \text{Up}$	$(4a)$	$(30, \perp, \perp)$
$(30, \perp, \perp)$	0	$D_1(30) = \text{Up}$	$(4a)$	$(40, \perp, \perp)$
$(40, \perp, \perp)$	0	$D_1(40) = \text{Zero}$	(2)	$(\perp, 40, \perp)$
$(\perp, 40, \perp)$	1	$D_2(40) = \text{Up}$	$(3a)$	$(01, 40, \perp)$
$(01, 40, \perp)$	0	$D_1(01) = \text{Up}$	$(4a)$	$(11, 40, \perp)$
$(11, 40, \perp)$	0	$D_1(11) = \text{Up}$	$(4a)$	$(21, 40, \perp)$
...
$(51, 40, \perp)$	0	$D_1(51) = \text{Zero}$	(2)	$(\perp, 51, \perp)$
$(\perp, 51, \perp)$	1	$D_2(51) = \text{Up}$	$(3a)$	$(02, 51, \perp)$
$(02, 51, \perp)$	0	$D_1(02) = \text{Up}$	$(4a)$	$(12, 51, \perp)$
...
$(62, 51, \perp)$	0	$D_1(62) = \text{Zero}$	(2)	$(\perp, 62, \perp)$
$(\perp, 62, \perp)$	1	$D_2(62) = \text{Zero}$	(2)	$(\perp, \perp, 62)$
$(\perp, \perp, 62)$	2	$i = d$	(1)	$(\perp, \perp, 62)$

The end of the line is $(\perp, 62, \perp)$.

Cost function c First, the help function $g: (\Gamma \cup \{\perp\}) \times \{0, 1, \dots, d-1\} \rightarrow \mathbb{N}$ is defined as follows:

$$g(p, i) := \begin{cases} (p)_{i+1} + 1 & \text{if } p \neq \perp \\ 0 & \text{otherwise} \end{cases}$$

$$h: V \rightarrow \mathbb{N}^d$$

$$h(v) := (g(p_0, 0), g(p_1, 1), \dots, g(p_{d-1}, d-1)) =: (l_0, l_1, \dots, l_{d-1})$$

Recall Example 4.1.18 on page 32. Function g , with the parameters that h calls it with, returns the same value that condition (c) and (d) of `isVertex` enforce for all smaller points at that index.

Let \triangleleft be a relation between two elements of \mathbb{N}^d such that it orders them lexicographically from the right.

$$\triangleleft := \{(x, y) \mid x, y \in \mathbb{N}^d, j := \max_{i=1, \dots, d} \{i \mid x_i \neq y_i\}, x_j < y_j\}$$

Let $\omega \in \mathbb{N}$ be a variable larger than any grid width used in any dimension such that $\forall i = 1, \dots, d: \omega_i + 1 < \omega$. With these two help functions, c is defined as follows:

$$c(v) := \left(\sum_{i=0}^{d-1} \omega^i l_i \right) + \begin{cases} 0 & \text{if } p_d = \perp \\ \omega^d & \text{if } p_d \neq \perp \end{cases}$$

If $(p_d = \perp = q_d)$ or $(p_d \neq \perp \wedge q_d \neq \perp)$, and $c(v) < c(u)$, it holds that $h(v) \triangleleft h(u)$ with $u = (q_0, \dots, q_d)$.

Example 4.1.20. Consider Example 4.1.19 on the preceding page. Let $\omega = 12$ because the largest grid width in this example is $\omega_1 = 10$. The following table shows the cost of each node that is visited during the line-following algorithm.

Vertex $v = (p_0, p_1, p_2)$	$g(p_0, 0)$	$g(p_1, 1)$	$c(v)$
(00, \perp , \perp)	1	0	1
(10, \perp , \perp)	2	0	2
(20, \perp , \perp)	3	0	3
(30, \perp , \perp)	4	0	4
(40, \perp , \perp)	5	0	5
(\perp , 40, \perp)	0	1	12
(01, 40, \perp)	1	1	13
(11, 40, \perp)	2	1	14
...
(51, 40, \perp)	6	1	18
(\perp , 51, \perp)	0	2	24
(02, 51, \perp)	1	2	25
...
(62, 51, \perp)	7	2	31
(\perp , 62, \perp)	0	3	36
(\perp , \perp , 62)	0	0	144

Originally, [Fea+20b] defined the cost function without the possible $+\omega^d$ at the end. This definition is not sufficient because there are always two nodes that define a violation of type (UFV1) even though the OPDC instance is correct. By the definition of a promise preserving reduction, this is not allowed.

Example 4.1.21. Let c' be the cost function as defined in [Fea+20b]:

$$c'(v) := \sum_{i=0}^{d-1} \omega^i l_i$$

Consider an OPDC instance for dimension $d = 3$ that has a unique fixpoint at $(1, 2, 3)$. Let

$$v = \begin{pmatrix} (0, 0, 0) \\ \perp \\ \perp \\ \perp \end{pmatrix}, u = \begin{pmatrix} (0, 0, 0) \\ \perp \\ \perp \\ (1, 2, 3) \end{pmatrix}.$$

Since the cost function ignores the last element, it holds that $c'(v) = c'(u)$. Furthermore $\text{isVertex}(v) = \text{isVertex}(u) = \text{true}$, since $(1, 2, 3)$ is the unique fixpoint. Also $S(v) \neq v$ and $S(u) \neq u$. Therefore, v and u are a violation of $(UFV1)$.

Such an example can be constructed for every OPDC instance with a unique fixpoint. In order to prevent that, the cost function is defined differently here.

Correctness In order to prove the correctness of this construction, it must be shown that each solution to the constructed UNIQUE FORWARD EOPL instance matches back to the correct OPDC instance.

The general idea of the proof done here is the same as in [Fea+20b]: We will take a look at each type of solution of UNIQUE FORWARD EOPL and argue why and which solution of OPDC is represented by it. The argumentation of the single steps might vary from the original paper in order to make them easier to understand.

A solution of type $(UF1)$ Let $v = (p_0, p_1, \dots, p_d)$ be a solution of type $(UF1)$ and $u = S(v)$. This means that $S(v) \neq v$ and either $S(u) = u$ or $c(u) \leq c(v)$.

Assume $S(v) \neq v$ and $S(u) = u$ with $u = (q_0, \dots, q_d)$. We now take a look at the rules that were used to determine $S(u)$. We can check that in polynomial time.

First we consider the case that $\text{isVertex}(u) = \text{true}$.

- 2.1. If $S(u) = u$ is determined by rule (2.1) , it follows that $q_d \neq \perp$. Since $\text{isVertex}(u) = \text{true}$, it holds that $D_i(q_d) = \text{Zero}$ for all i , which means that q_d is a solution of type $(O1)$.
- 2.2. $S(u) = u$ cannot be determined by rule (2.2) , $(2.3a)$ or $(2.4a)$ since by the definition of these rules the created successor is unequal to the input.
- 2.3. If $S(u) = u$ is determined by rule $(2.3b)$, we found a solution of type $(OV3)$, as already argued in the construction.
- 2.4. If $S(u)$ is determined by rule $(2.4b)$, we found a solution of type $(OV3)$, as already argued in the construction.

In case (1) it holds that $S(u) = u$ because $\text{isVertex}(u) = \text{false}$. In this case we have to take a closer look at how $S(v)$ was determined.

- 2.1. $S(v)$ cannot be determined by rule (2.1) , since then it would hold that $S(v) = v$, which is a contradiction to our assumption.

2.2. If $S(v)$ is determined by rule (2.2), let i be the smallest index of v such that $p_i \neq \perp$.

Vertex	Index						
	0	...	$i-1$	i	$i+1$	$i+2$...
v	\perp	...	\perp	x	y	z	...
$S(v)$	\perp	...	\perp	\perp	x	z	...

v and $S(v)$ differ only in the two points at position i and $i+1$. Since $\text{isVertex}(v) = \text{true}$, one of these two points of $S(v)$ must violate the conditions of isVertex . More specifically it is x at position $i+1$ that must violate a condition of isVertex because position i is set to \perp and therefore it is irrelevant for all conditions of isVertex .

- (a) $(S(v))_{i+1}$ cannot break condition (a) of isVertex since $D_{i+1}((S(v))_{i+1}) = D_{i+1}(x) = \text{Zero}$ because we are in case (2.2) of the construction.
- (b) If $(S(v))_{i+1}$ violates condition (b) of isVertex , then $D_{i+2}((S(v))_{i+1}) = D_{i+2}(x) = \text{Down}$.

Let's take a look at y .

- (i) $y \neq \perp$

It holds that $D_{i+2}(y) = \text{Up}$: At some point there was a node $w = (\perp, \dots, \perp, y, \dots)$ with y at position $(i+1)$. The successor of w was not determined by rule (2.1), nor was it pushed any further by rule (2.2). Therefore, it holds that $D_{i+2}(y) \neq \text{Zero}$ and since $\text{isVertex}(v) = \text{true}$, it also holds that $D_{i+2}(y) \neq \text{Down}$. Therefore, Up is the only direction left.

Since $\text{isVertex}(v) = \text{true}$, it also holds by condition (d) that $(x)_{i+2} = (y)_{i+2} + 1$.

Moreover $\forall j \leq i+1: D_j(x) = D_j(y) = \text{Zero}$. In particular $D_{i+1}(x) = \text{Zero}$ holds because we are in case (2.2) of the construction.

This means that we found a violation of type (OV2).

- (ii) $y = \perp$

Because $\text{isVertex}(v) = \text{true}$ and condition (c), it holds that $(x)_{i+2} = 0$. Since $D_{i+2}(x) = \text{Down}$, we found a violation of type (OV3).

- (c) If conditions (c) or (d) do not hold for $S(v)$, it follows that it could not have been true for v either. Therefore $S(v)$ cannot break these conditions.

2.3. $S(v)$ is determined by rule (2.3a). Since $S(v) \neq v$, it could not have been case (2.3b).

$S(v) = (q, p_1, \dots, p_d)$ must violate one of the conditions of isVertex . Specifically q must cause a violation, since all the other points stayed the same.

- (a) Condition (a) cannot be violated, since $i = 0$.
- (b) If q violates condition (b) of isVertex then $D_1(q) = \text{Down}$. This means that we found a violation of type (OV3), since $q_1 = 0$ by definition.

- (c) $S(v)$ fulfills condition (c) by definition: Let i be the smallest index of v such that $p_i \neq \perp$.

For $j = 0, \dots, i - 1$ it holds that $q_{j+1} = 0$ by definition.

For $j = i$ it holds $p_j \neq \perp$ and the condition is skipped.

For $j = i + 1, \dots, d - 1$ it holds that $q_{j+1} = (p_i)_{j+1}$ for which the condition holds since $\text{isVertex}(v) = \text{true}$. Therefore, it holds for q_{j+1} as well.

Example 4.1.22. Consider $S(v)$ represented as matrix. Here it is easier to see why condition (c) holds for $j = i + 1, \dots, d - 1$.

$$S(v) = \begin{pmatrix} & & & = (p_i)_{j+1} = 0 & & \\ (q)_1 & \dots & (q)_j & \boxed{(q)_{j+1}} & \dots & (q)_d \\ & & & \perp & & \\ & & & \dots & & \\ & & & \perp = 0 & & \\ (p_i)_1 & \dots & (p_i)_j & \boxed{(p_i)_{j+1}} & \dots & (p_i)_d \\ & & & \dots & & \\ p_j = \perp & \boxed{(p_j)_1} & \dots & \boxed{(p_j)_j} & \boxed{(p_j)_{j+1}} & \dots & \boxed{(p_j)_d} \\ & & & \dots & & \\ (p_d)_1 & \dots & (p_d)_j & (p_d)_{j+1} & \dots & (p_d)_d \end{pmatrix} \begin{array}{l} \leftarrow \text{Holds because} \\ \text{isVertex}(v) = \text{true} \end{array}$$

- (d) $S(v)$ fulfills condition (d) by definition: Let i be the smallest index of v such that $p_i \neq \perp$.

For $j = 0, \dots, i - 1$ $p_j = \perp$, since i is by definition the smallest and therefore this condition is skipped.

For $j = i$ it holds that $q_{j+1} = (p_j)_{j+1} + 1$ by definition.

For $j = i + 1, \dots, d - 1$ it holds that $q_{j+1} = (p_i)_{j+1}$ for which the condition holds, since $\text{isVertex}(v) = \text{true}$. Therefore it holds for q_{j+1} as well.

Example 4.1.23. Consider $S(v)$ represented as matrix. Here it is easier to see why condition (d) holds for $j = i, \dots, d - 1$.

$$S(v) = \begin{pmatrix} & & & = (p_i)_{j+1} = (p_j)_{j+1} + 1 & & \\ (q)_1 & \dots & (q)_j & \boxed{(q)_{j+1}} & \dots & (q)_d \\ & & & \perp & & \\ & & & \dots & & \\ & & & \perp = (p_j)_{j+1} + 1 & & \\ (p_i)_1 & \dots & (p_i)_j & \boxed{(p_i)_{j+1}} & \dots & (p_i)_d \\ & & & \dots & & \\ p_j \neq \perp & \boxed{(p_j)_1} & \dots & \boxed{(p_j)_j} & \boxed{(p_j)_{j+1}} & \dots & \boxed{(p_j)_d} \\ & & & \dots & & \\ (p_d)_1 & \dots & (p_d)_j & (p_d)_{j+1} & \dots & (p_d)_d \end{pmatrix} \begin{array}{l} \leftarrow \text{Holds because} \\ \text{isVertex}(v) = \text{true} \end{array}$$

2.4. $S(v)$ is determined by rule (2.4a). Since $S(v) \neq v$ it could not have been case (2.4b).

$S(v) = ((p_0)_1 + 1, (p_0)_2, \dots, (p_0)_d, p_1, \dots, p_d)$ must violate one of the conditions of isVertex . Specifically $((p_0)_1 + 1)$ must cause a violation, since all the other points stayed the same.

- (a) Condition (a) cannot be violated since $i = 0$.
- (b) If $(S(v))_0$ violates condition (b) of `isVertex`, then $D_1((S(v))_0) = \text{Down}$.
 $D_1(p_0) = \text{Up}$ because it is not `Zero` since we are in case (2.4) and it is not `Down` because `isVertex(v) = true`.
This means that we found a violation of type (OV2).
- (c) $S(v)$ fulfills condition (c) and (d) by definition. Since only $(p_0)_1$ is changed, all other points must have fulfilled the condition already before. For $(p_0)_1$ itself the condition never is relevant.

In conclusion, every case leads to a solution, a violation or the case is impossible to reach.

Second, assume $S(v) \neq v$ and $c(u) \leq c(v)$. Note that $S(u) \neq u$. Based on the rules that are used to calculate $S(v)$ we can argue that this is impossible:

1. $S(v)$ cannot be determined by rule (1), since then it would hold that $S(v) = v$, which is a contradiction to our assumption.
- 2.1. $S(v)$ cannot be determined by rule (2.1), since then it would hold that $S(v) = v$, which is a contradiction to our assumption.
- 2.2. If $S(v)$ was determined by the second rule, it can be proven that $c(v) < c(u)$:

v and u differ only in the two points at position i and $(i + 1)$. The cost of v and u differs only in the terms of the sum of i and $(i + 1)$.

If $p_{i+1} \neq \perp$, it holds that $(p_i)_{i+2} = ((p_{i+1})_{i+2} + 1)$ because of condition (d) of `isVertex` and `isVertex(v) = true`.

$$\begin{aligned}
c(u) &\stackrel{!}{>} c(v) \\
\sum_{j=0}^{d-1} \omega^j l_j &> \sum_{j=0}^{d-1} \omega^j l'_j \\
\omega^i g(\perp, i) + \omega^{i+1} g(p_i, i+1) &> \omega^i g(p_i, i) + \omega^{i+1} g(p_{i+1}, i+1) \\
\omega^{i+1} ((p_i)_{i+2} + 1) &> \omega^i ((p_i)_{i+1} + 1) + \omega^{i+1} ((p_{i+1})_{i+2} + 1) \\
\omega((p_i)_{i+2} + 1) &> ((p_i)_{i+1} + 1) + \omega((p_{i+1})_{i+2} + 1) \\
\omega((p_i)_{i+2} + 1) &> ((p_i)_{i+1} + 1) + \omega((p_i)_{i+2}) \\
\omega((p_i)_{i+2}) + \omega &> ((p_i)_{i+1} + 1) + \omega((p_i)_{i+2}) \\
\omega &> ((p_i)_{i+1} + 1)
\end{aligned}$$

This holds because ω is by definition larger than any grid width plus one.

If $p_{i+1} = \perp$ then $(p_i)_{i+2} = 0$ because of condition (c) of `isVertex` and `isVertex(v) = true`.

$$\begin{aligned}
c(u) &\stackrel{!}{>} c(v) \\
\sum_{j=0}^{d-1} \omega^j l_j &> \sum_{j=0}^{d-1} \omega^j l'_j \\
\omega^i g(\perp, i) + \omega^{i+1} g(p_i, i+1) &> \omega^i g(p_i, i) + \omega^{i+1} g(p_{i+1}, i+1) \\
\omega^{i+1} ((p_i)_{i+2} + 1) &> \omega^i ((p_i)_{i+1} + 1) \\
\omega^{i+1} &> \omega^i ((p_i)_{i+1} + 1)
\end{aligned}$$

This holds because $((p_i)_{i+1} + 1) < \omega$.

- 2.3. If $S(v)$ was determined by the third rule, it can be proven again that $c(v) < c(u)$. v and u only differ in the first position. Furthermore we know that $p_0 = \perp$ and $(q)_1 = 0$ since $i > 0$.

$$\begin{aligned}
c(u) &\stackrel{!}{>} c(v) \\
\sum_{j=0}^{d-1} \omega^j l_j &> \sum_{j=0}^{d-1} \omega^j l'_j \\
g(q, 0) &> g(p_0, 0) \\
(q)_1 + 1 &> 0 \\
1 &> 0
\end{aligned}$$

- 2.4. If $S(v)$ was determined by the fourth rule, it can be proven again that $c(v) < c(u)$. v and u only differ in the first position. We know that $(q)_1 = ((p_0)_1 + 1)$.

$$\begin{aligned}
c(u) &\stackrel{!}{>} c(v) \\
\sum_{j=0}^{d-1} \omega^j l_j &> \sum_{j=0}^{d-1} \omega^j l'_j \\
g(q, 0) &> g(p_0, 0) \\
(q)_1 + 1 &> (p_0)_1 + 1 \\
((p_0)_1 + 1) + 1 &> (p_0)_1 + 1 \\
1 &> 0
\end{aligned}$$

In conclusion, it can never happen that $S(v) \neq v$ and $c(u) \leq c(v)$.

A violation of type (UFV1) Consider the case that for the transformed instance a violation solution of type (UFV1) is found. It needs to be proven that the original instance is violated too.

Given $v = (p_0, \dots, p_d)$ and $u = (q_0, \dots, q_d)$ with $v \neq u$, $v \neq S(v)$, $u \neq S(u)$ and $\text{isVertex}(v) = \text{isVertex}(u) = \text{true}$. For a violation of type (UFV1) it holds that either

- (a) $c(v) = c(u)$ or

(b) $c(v) < c(u) < c(S(v))$.

If we are in case (a) it follows that there is a violation of (OV1):

If $c(v) = c(u)$ then, if you imagine the cost as matrix again, the entries on the diagonal are equal, i.e. for all $m = 0, \dots, d-1$: $(p_m)_{m+1} = (q_m)_{m+1}$. This implies that the entries above the diagonal are equal as well because of condition (c) and (d) of `isVertex`.

Let j be the largest index in which v and u differ, i.e. $p_j \neq q_j$. It cannot happen that $p_j = \perp$ or $q_j = \perp$ since this would imply that $c(v) \neq c(u)$. For $m > j$ it holds that $(p_j)_m = (q_j)_m$. This means, p_j and q_j are in the same j -Slice. Because of condition (a) of `isVertex` it must hold for p_j and q_j that $\forall n \leq j$: $D_n(p_j) = \text{Zero}$ and $D_n(q_j) = \text{Zero}$. p_j and q_j are two fixpoints for the j -Slice. It follows that p_j and q_j form a violation of type (OV1).

If we are in case (b), we take a look at how $S(v)$ was determined in order to show that the original OPDC instance had a violation as well or that the case cannot possibly be reached by the construction.

- 2.1. $S(v)$ cannot be determined by rule (2.1), since then it would hold that $S(v) = v$, which is a contradiction to our assumption.
- 2.2. If $S(v)$ was determined by the second rule, let i be the smallest index of v such that $p_i \neq \perp$.

Since $p_d = S(v)_d$ and $c(v) < c(u) < c(S(v))$, it follows that $q_d = \perp$ if $p_d = \perp$ or $q_d \neq \perp$ if $p_d \neq \perp$. Therefore $h(v) \triangleleft h(u) \triangleleft h(S(v))$:

Vertex	Index						
	0	...	$i-1$	i	$i+1$...	$d-1$
$h(v)$	0	...	0	$(p_i)_{i+1} + 1$	$(p_{i+1})_{i+2} + 1$...	$(p_{d-1})_d + 1$
$h(u)$	l_0	...	l_{i-1}	l_i	$(q_{i+1})_{i+2} + 1$ $= (p_{i+1})_{i+2} + 1$...	$(q_{d-1})_d + 1$ $= (p_{d-1})_d + 1$
$h(S(v))$	0	...	0	0	$(p_i)_{i+2} + 1$...	$(p_{d-1})_d + 1$

Because of `isVertex(v) = true`, if $p_{i+1} \neq \perp$ it holds that $(p_i)_{i+2} = (p_{i+1})_{i+2} + 1$. $h(v)_i$ and $h(S(v))_i$ differ only by 1 and the values at index greater than $i+1$ are all equal, it must hold that $h(v)_{i+1} = h(u)_{i+1}$.

Let j be the largest index such that $h(v)_j \neq h(u)_j$. We now show that p_j and q_j form a violation of type (OV1) like in case (a).

Let \mathbf{s} be the j -Slice such that:

Slice	Index						
	0	...	$j-1$	j	$j+1$...	d
\mathbf{s}	*	...	*	*	$(p_j)_{j+1}$...	$(p_j)_d$

p_j lies within the slice \mathbf{s} . In order to prove that q_j lies in slice \mathbf{s} as well, we need to show that $\forall m > j$: $(p_j)_m = (q_j)_m$.

For all $m > j$ it holds that the elements on the diagonal are the same:

$$\begin{aligned} h(v)_m &= h(u)_m \\ g(p_m, m) &= g(q_m, m) \\ (p_m)_{m+1} + 1 &= (q_m)_{m+1} + 1 \\ (p_m)_{m+1} &= (q_m)_{m+1} \end{aligned}$$

Since $\text{isVertex}(v) = \text{isVertex}(u) = \text{true}$ it follows that all elements above the diagonal are equal as well. Condition (d) enforces that for $j < m$ if $p_m \neq \perp$ and $q_m \neq \perp$ that $(p_j)_{m+1} = (p_m)_{m+1} + 1 = (q_m)_{m+1} + 1 = (q_j)_{m+1}$ and condition (c) enforces that if $p_m = \perp$ and $q_m = \perp$, $(p_j)_{m+1} = 0 = (q_j)_{m+1}$.

Therefore q_j is in slice \mathbf{s} as well.

Furthermore, condition (a) of isVertex implies that $\forall n \leq j: D_n(p_j) = \mathbf{Zero}$ and $D_n(q_j) = \mathbf{Zero}$. So p_j and q_j are two distinct fixpoints in the same slice which is a violation of type (OV1).

2.3. It will be proven that $S(v)$ cannot be determined by the third rule.

If $S(v)$ was determined by the third rule, $h(v) = (0, l_1, \dots, l_d)$ and $h(S(v)) = (1, l_1, \dots, l_d)$. There doesn't exist a tuple $t = (l'_0, l'_1, \dots, l'_d)$ such that $h(v) \triangleleft t \triangleleft h(S(v))$.

2.4. Similar to case 2.3. it will be proven that $S(v)$ cannot be determined by the fourth rule.

If $S(v)$ was determined by the fourth rule, $h(v) = (l_0, l_1, \dots, l_d)$ and $h(S(v)) = (l_0 + 1, l_1, \dots, l_d)$. Again there cannot exist a tuple t such that $h(v) \triangleleft t \triangleleft h(S(v))$.

A violation of type (UFV2) Let v and x be the given vertices of the violation (UFV2) with $v \neq x$, $S(x) \neq x$ and $c(v) < c(x)$. Since v is also a solution of type (UF1), it holds that $S(v) = u$, $v \neq u$ and either $S(u) = u$ or $c(u) \leq c(v)$.

If $S(u)$ is determined by anything other than rule (2.1), there is a violation in the OPDC instance as shown in case (UF1).

So if $S(u)$ is indeed determined by rule (2.1), we must show that this, in combination with x , leads to another violation. Since x is a node on a line, there must exist an end of that line somewhere. Let y be that end.

It holds that $c(y) > c(x)$. Therefore it must hold that $v \neq y$ because $c(x) > c(v)$.

If $S(S(y)) = S(y)$ is determined by anything other than rule (2.1), we found a violation. Otherwise v and y are two distinct fixpoints and therefore two solutions of type (O1), which together form a violation of type (OV1).

Conclusion All in all, every solution that is found in the constructed UNIQUE FORWARD EOPL instance can be mapped back to the corresponding OPDC instance in polynomial time.

The reduction is promise preserving. Every solution of type $(O1)$ is only mapped to solutions of type $(UF1)$. Every violation of the created UNIQUE FORWARD EOPL instance is mapped back to a violation of the OPDC instance.

□

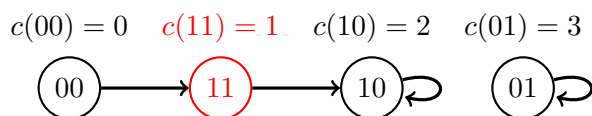
4.1.1.2. Unique Forward EOPL to Unique Forward EOPL+1

Definition 4.1.24 (UNIQUE FORWARD EOPL+1). Given a Boolean circuit $S : \{0, 1\}^d \rightarrow \{0, 1\}^d$ such that $S(0^d) \neq 0^d$ and a Boolean circuit $c : \{0, 1\}^d \rightarrow \{0, 1, \dots, 2^n - 1\}$ such that $c(0^d) = 0$, find one of the following:

- (*UFP1*) A point $x \in \{0, 1\}^d$ such that $S(x) \neq x$ and either $S(S(x)) = S(x)$ or $c(S(x)) \neq c(x) + 1$.
 x is a sink and a valid solution.
- (*UFVP1*) Two points $x, y \in \{0, 1\}^d$ such that $x \neq y, S(x) \neq x, S(y) \neq y$ and $c(x) = c(y)$.
 x and y are two different nodes that have the same potential, which implies that the instance has not the form of an increasing line.
- (*UFVP2*) Two points $x, y \in \{0, 1\}^d$ such that x is a solution of type (*UFP1*), $x \neq y, S(y) \neq y$ and $c(x) < c(y)$.
This encodes a break in the line. x is the end of one line, but there exists a different line with a node y that has larger cost than x .

This problem can be interpreted as a directed acyclic graph $G = (V, E)$ with $V := \{v \mid v \in \{0, 1\}^d \wedge S(v) \neq v\}$ and $(v, u) \in E \Leftrightarrow (v \neq u \wedge S(v) = u \wedge c(u) = c(v) + 1)$.

Example 4.1.25. Consider the following UNIQUE FORWARD EOPL+1 instance for $d = 2$. The arrows represent function S , *not* the edges of the graph. The vertices (10) and (01) aren't actual vertices. $V = \{00, 11\}$, $E = \{(00, 11)\}$. (11) is a solution of type (*UFP1*).



Lemma 4.1.26. *Given an END OF POTENTIAL LINE instance $I = (S, P, c)$, it is possible to promise reduce it in polynomial time to another END OF POTENTIAL LINE instance $I' = (S', P', c')$ such that $c'(S'(v)) = c'(v) + 1$ [Fea+18, Lemma 22].*

Proof. A node of instance I' is defined as tuple (v, i) with v being a node from instance I and $i \in \{1, \dots, 2^d\}$.

Let v and u be two nodes of instance I such that $S(v) = u$. If $c(u) > c(v) + 1$, they will be replaced by the following sequence of nodes in I' :

$$(v, 0) \rightarrow (v, 1) \rightarrow \dots \rightarrow (v, c(u) - c(v) - 1) \rightarrow (u, 0)$$

Formally, $S'(v, i)$ is defined as in [Fea+18]:

- (1) If $S(v) = v$ then for all i : $S'(v, i) = (v, i)$.
If v is not a vertex in I , then for all i , (v, i) is not a vertex in I' .

- (2) If $S(v) \neq v$ and $P(S(v)) \neq v$ then for all i : $S'(v, i) = (v, i)$.
 v is the end of the original line, therefore $(v, 0)$ is the end of I' and for all other i : (v, i) is not a valid vertex.
- (3) If $S(v) \neq v = P(S(v))$ and $c(S(v)) < c(v) + i + 1$ then $S'(v, i) = (v, i)$.
- (4) If $S(v) \neq v = P(S(v))$ and $c(S(v)) = c(v) + i + 1$ then $S'(v, i) = (S(v), 0)$.
- (5) If $S(v) \neq v = P(S(v))$ and $c(S(v)) > c(v) + i + 1$ then $S'(v, i) = (v, i + 1)$.

The predecessor $P'(v, i)$ can be constructed analogously:

- (1) If $P(v) = v$ then for all i : $P'(v, i) = (v, i)$.
- (2) If $S(v) \neq v = P(S(v))$ and $c(S(v)) < c(v) + i + 1$ then $P'(v, i) = (v, i)$.
- (3) If $S(v) \neq v = P(S(v))$ and $i \neq 0$ then $P'(v, i) = (v, i - 1)$.
- (4) If $S(v) \neq v = P(S(v))$ and $i = 0$ then $P'(v, 0) = (P(v), c(v) - c(P(v)) - 1)$.
- (5) If $P(S(v)) \neq v$ and $i = 0$ then $P'(v, 0) = (P(v), c(v) - c(P(v)) - 1)$.
- (6) If $P(S(v)) \neq v$ and $i \neq 0$ then $P'(v, i) = (v, i)$.

The cost function c' is defined as follows:

$$c'(v, i) = c(v) + i.$$

These functions can be computed in polynomial time.

The correctness of this construction holds since all types of solution of I' can be mapped back to the same type of solution in I .

□

Lemma 4.1.26 on the previous page holds as well when considering instances with a unique line and instances without a predecessor circuit. It follows that there exists a promise preserving polynomial time reduction from UNIQUE FORWARD EOPL to UNIQUE FORWARD EOPL+1.

4.1.1.3. Unique Forward EOPL+1 to Unique End of Potential Line

This reduction is mainly inspired by the reduction from SINK OF VERIFIABLE LINE to END OF LINE [BPR15] and to END OF METERED LINE [HY17]. SINK OF VERIFIABLE LINE is very similar to UNIQUE FORWARD EOPL+1. The instances created by the reduction consist of a single line, which is why the reduction can be easily adapted to UNIQUE END OF POTENTIAL LINE.

Theorem 4.1.27. *There exists a promise preserving polynomial time reduction from UNIQUE FORWARD EOPL+1 to UNIQUE END OF POTENTIAL LINE [Fea+20b, Lemma 15].*

Proof sketch. Given an instance $I = (S, c)$ of UNIQUE FORWARD EOPL+1, construct an instance $I' = (S', P', c')$ of UNIQUE END OF POTENTIAL LINE.

The hard part of the reduction is to create the predecessor circuit which is not given in UNIQUE FORWARD EOPL+1. The idea of the reduction is based on the pebbling game technique. The pebbling game is played by placing d pebbles on the vertices of a line based on the following rules:

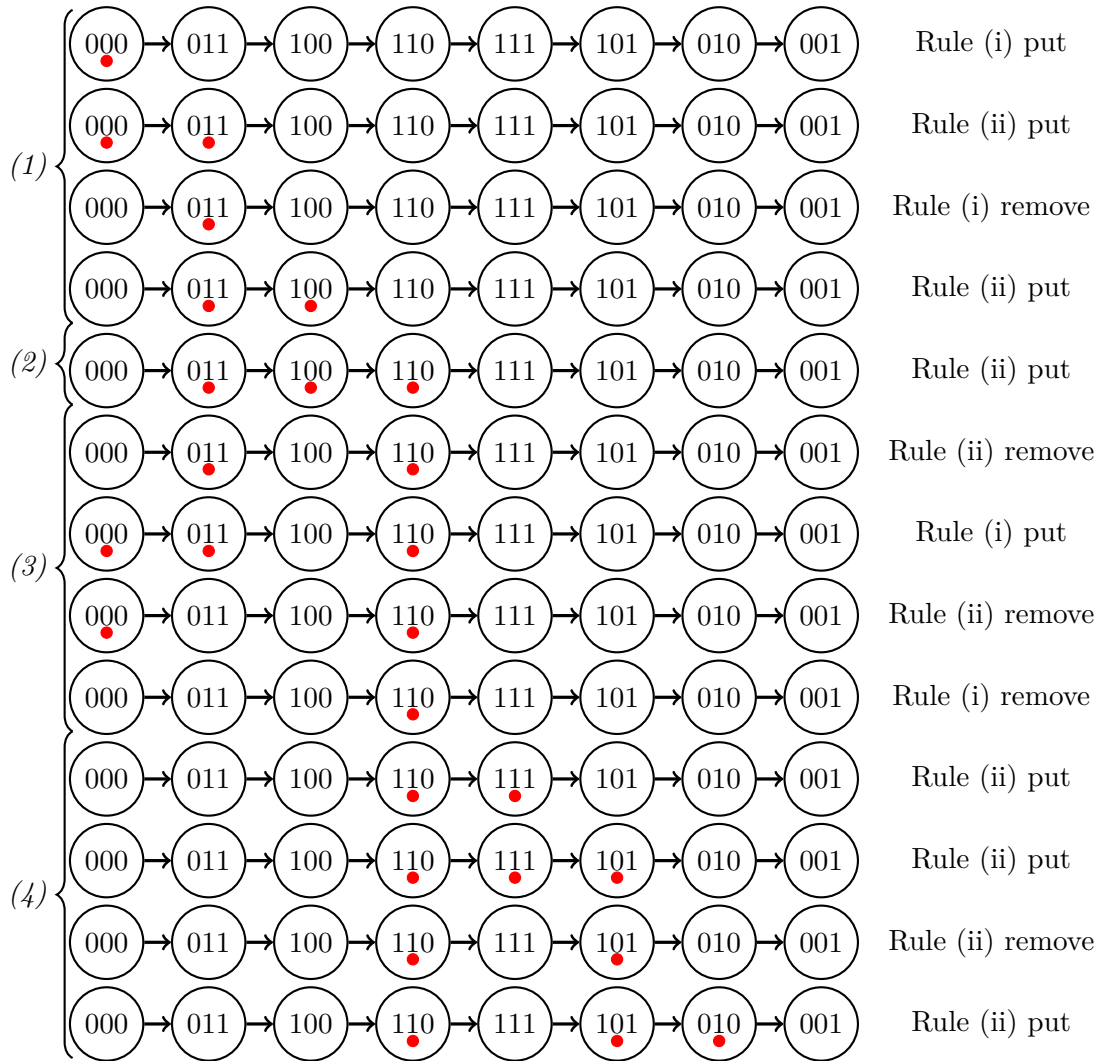
- (i) A pebble may be put or removed from the starting vertex o at any time.
- (ii) A pebble may be put or removed on a vertex $v \neq o$ if and only if there is a pebble on the vertex u with $S(u) = v$.

Given d pebbles, the goal is to place a pebble at position $2^d - 1$ of the line by following these rules.

There exists a recursive optimal strategy to do that:

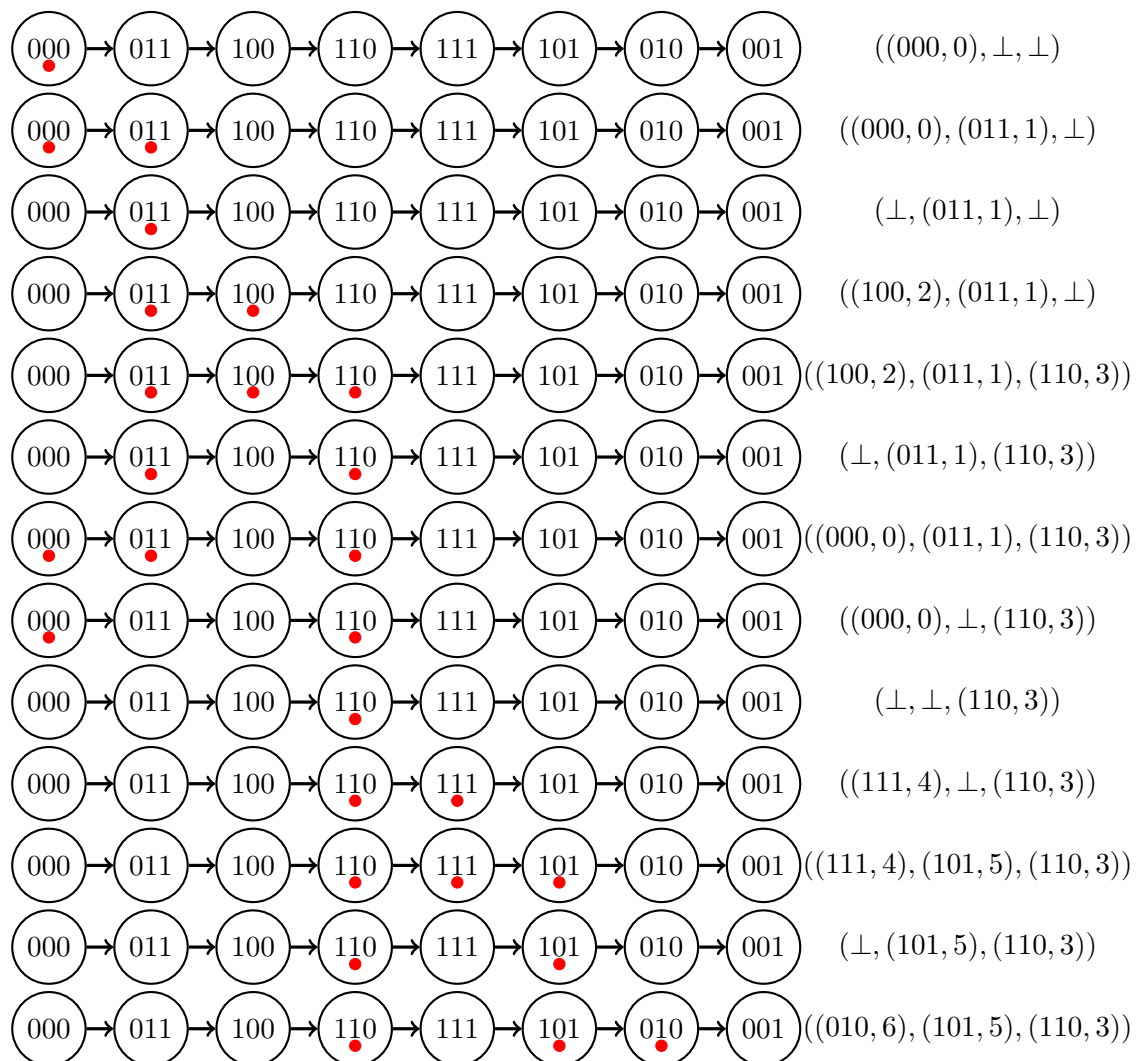
- (1) Use $b - 1$ many pebbles to place a pebble at position $2^{b-1} - 1$.
- (2) Place the b 'th pebble at position 2^{b-1} .
- (3) Retrieve the first $b - 1$ pebbles by reversing the sequence used to place them.
- (4) Use the same strategy from step (1) again to place the pebbles behind the pebble at position 2^{b-1} to reach $2^b - 1$.

Example 4.1.28. $d = 3$. We have 3 pebbles which we can use and want to place a pebble on node at $2^d - 1 = 7$ which is the node (010).



Let a vertex q of I' be a tuple of pairs that encode the state of the pebble game. $t = ((u_1, b_1), \dots, (u_d, b_d))$ with $u_i \in \{0, 1\}^d$ and $b_i = c(u_i)$. b_i tells us that the i 'th pebble is at position b_i . If a pebble is not in the game at all, the tuple is \perp .

Example 4.1.29. Recall Example 4.1.28 on the preceding page. This is how the vertices look like.



The functions S' and P' are defined recursively by deciding in which of the four steps of the optimal strategy we currently are [BPR15].

Given a state, it is possible to determine how far the execution of the optimal strategy is. This value is returned by the cost function [HY17].

[Fea+20b] shows that every violation of UNIQUE END OF POTENTIAL LINE can be mapped back to a violation of UNIQUE FORWARD EOPL+1 and that the reduction is promise preserving.

□

4.1.2. OPDC is UniqueEOPL-hard

Theorem 4.1.30. *OPDC is UniqueEOPL-hard.*

The proof is done in two steps:

$$\begin{aligned} \text{UNIQUE END OF POTENTIAL LINE} &\preceq_{\text{PROMISE}} \text{UNIQUE EOPL+1} \\ &\preceq_{\text{PROMISE}} \text{OPDC}. \end{aligned}$$

From Theorem 4.1.14 on page 29 and Theorem 4.1.30 it follows that OPDC is UniqueEOPL-complete.

4.1.2.1. Unique End of Potential Line to Unique EOPL+1

UNIQUE EOPL+1 describes a UNIQUE END OF POTENTIAL LINE instance in which the line has length 2^n and the costs are increasing strictly by 1. It follows that the end of the line v must have cost $c(v) = 2^n - 1$.

Definition 4.1.31 (UNIQUE EOPL+1). Given:

- $o \in \{0, 1\}^d$ as start node of the line.
- $S, P : \{0, 1\}^d \rightarrow \{0, 1\}^d$ two Boolean circuits such that $P(o) = o \neq S(o)$.
- $c : \{0, 1\}^d \rightarrow \{0, 1, \dots, 2^n - 1\}$ a Boolean circuit such that $c(o) = 0$.
- $n \in \mathbb{N}$ is the logarithm of the length of the line.

The task is to find one of the following:

- (U1.1) A point $x \in \{0, 1\}^d$ such that $P(S(x)) \neq x$ and $c(x) = 2^n - 1$.
 x is the end of the line and therefore a valid solution.
- (V1.1) A point $x \in \{0, 1\}^d$ such that $P(S(x)) = x$, $x \neq S(x)$ and $c(S(x)) - c(x) \neq 1$.
 x is in the middle of a line, but the costs of the following vertex are not increasing exactly by one.
- (V1.2) A point $x \in \{0, 1\}^d$ such that $S(P(x)) \neq x \neq o$.
 x is the beginning of a different line which does not start at o .
- (V1.3) Two points $x, y \in \{0, 1\}^d$ such that $x \neq y$, $x \neq S(x)$, $y \neq S(y)$ and either
 - a) $c(x) = c(y)$ or
 - b) $c(x) < c(y) < c(S(x))$.

x and y are two different nodes that either have the same potential, or that violate the increasing potential. Both cases imply that the instance has more than one line.

Theorem 4.1.32. *There exists a promise preserving polynomial time reduction from UNIQUE END OF POTENTIAL LINE to UNIQUE EOPL+1.*

Proof sketch. The idea is again that every UNIQUE END OF POTENTIAL LINE instance can be filled up with dummy nodes, as proven in Lemma 4.1.26 on page 43.

Additionally, some more dummy vertices need to be introduced at the end of the line to ensure the correct cost of the last node.

The solutions of the UNIQUE EOPL+1 instance map back one to one to the solutions of the UNIQUE END OF POTENTIAL LINE instance.

□

4.1.2.2. Unique EOPL+1 to OPDC

Theorem 4.1.33. *There exists a polynomial time reduction from UNIQUE EOPL+1 to ONE PERMUTATION DISCRETE CONTRACTION (OPDC).*

Proof. Given a UNIQUE EOPL+1 instance $I = (o, S, P, c, n)$ with $S, P : \{0, 1\}^d \rightarrow \{0, 1\}^d$, $c : \{0, 1\}^d \rightarrow \{0, 1, \dots, 2^n - 1\}$ and $n \in \mathbb{N}$, construct an OPDC instance $I' = (\Gamma, \mathcal{D})$ where Γ is a discrete grid of points over $[0, 1]^d$ and \mathcal{D} is a family of direction functions.

First, it is important to know that we are able to split a UNIQUE EOPL+1 instance $I = (o, S, P, c, n)$ at a certain node x into two halves I_I and I_{II} in polynomial time. This can simply be done by creating $I_I = (o_I, S_I, P_I, c_I, n_I)$ and $I_{II} = (o_{II}, S_{II}, P_{II}, c_{II}, n_{II})$ that check whether for any node v $c(v) \geq c(x)$ or not.

An instance can be split at any arbitrary node x . If $c(x) = 2^{n-1}$ then the instance is split exactly in the middle. In the following reduction the line is *always* split in the middle. *firstHalf* is given the cost of the node at which the line should be split whereas *secondHalf* is given the node itself. This is because *secondHalf* needs the concrete node as a new start point of the line. When we call *firstHalf* later, we might not know which node is the one in the middle of the line, but we know its cost. The functions *firstHalf* and *secondHalf* are defined as follows:

$$S_I(v) = \begin{cases} S(v) & \text{if } c(v) < t \\ v & \text{if } c(v) \geq t \end{cases} \quad P_I(v) = \begin{cases} P(v) & \text{if } c(v) < t \\ v & \text{if } c(v) \geq t \end{cases} \quad c_I(v) = c(v)$$

$$I_I : \{0, 1\}^n \times \mathcal{I}^{\text{UNIQUE EOPL+1}} \rightarrow \mathcal{I}^{\text{UNIQUE EOPL+1}}$$

$$I_I := \text{firstHalf}(t, I) = (o, S_I, P_I, c_I, \log t)$$

$$S_{II}(v) = \begin{cases} v & \text{if } c(v) < c(x) \\ S(v) & \text{if } c(v) \geq c(x) \end{cases} \quad P_{II}(v) = \begin{cases} v & \text{if } c(v) < c(x) \\ P(v) & \text{if } c(v) \geq c(x) \end{cases} \quad c_{II}(v) = c(v) - c(x)$$

$$I_{II} : \{0, 1\}^d \times \mathcal{I}^{\text{UNIQUE EOPL+1}} \rightarrow \mathcal{I}^{\text{UNIQUE EOPL+1}}$$

$$I_{II} := \text{secondHalf}(x, I) = (x, S_{II}, P_{II}, c_{II}, \log(2^n - c(x)))$$

Example 4.1.34. Consider the following line. In this example the line is split at the node (10). Therefore *firstHalf* and *secondHalf* actually represent one half of the original line.

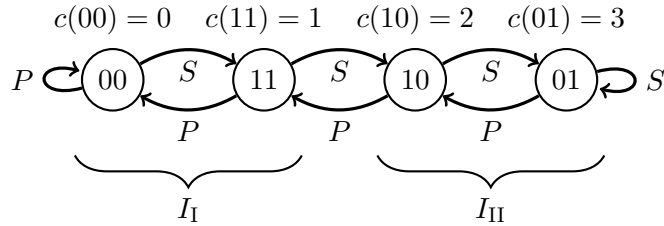


Table 4.2.: First and second half of the above instance split at $x = (10)$

Vertex v	$S(v)$	$P(v)$	$c(v)$	$S_I(v)$	$P_I(v)$	$c_I(v)$	$S_{II}(v)$	$P_{II}(v)$	$c_{II}(v)$
00	11	00	0	11	00	0	00	00	-2
11	10	00	1	10	11	1	11	11	-1
10	01	11	2	10	10	2	01	11	0
01	01	10	3	01	01	3	01	10	1
	$I = ((00), S, P, c, 2)$			$I_I = ((00), S_I, P_I, c_I, 1)$			$I_{II} = ((10), S_{II}, P_{II}, c_{II}, 1)$		

I_I acts like the original instance I as long as the functions are called with a node of the actual first half of the instance. If a node of the second half is put into one of the functions, they still are defined, but they just return what was put in as a dummy result. This can be seen in the gray boxes in Table 4.2. The same holds for the second half.

Figure 4.1.: I_I

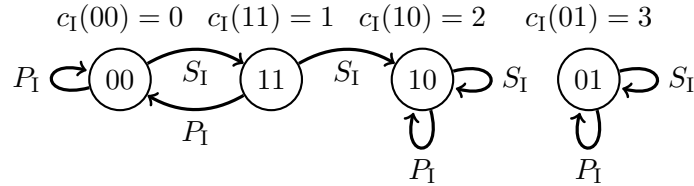
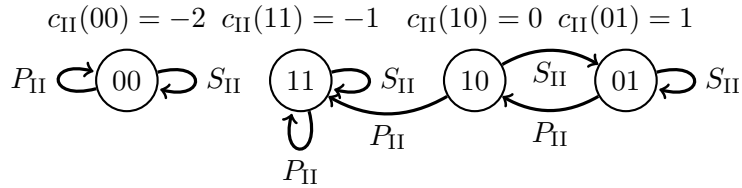


Figure 4.2.: I_{II}



With this we are able to split an instance into two halves in polynomial time, since we can redefine the functions without actually knowing which node is the one in the middle. This process can be done recursively until there are only instances with one node on the line left.

Note that this works even if the nodes don't form a correct line or if the cost of the nodes are shuffled and not increasing. *firstHalf* and *secondHalf* work with the cost of the node, *firstHalf* returns valid functions for all nodes with cost less than 2^{j-1} and *secondHalf* does the same for nodes with cost greater or equal than 2^{j-1} . For doing that, it doesn't matter where the nodes are on the line.

The grid Let $d' := d \cdot n$ and $\Gamma := \{0, 1\}^{d'}$. Γ is a hypercube. Each point $p \in \Gamma$ can be seen as a vector (v_1, v_2, \dots, v_n) where $v_i \in \{0, 1\}^d$ is a node in the UNIQUE EOPL+1 instance.

The *subline* function takes an UNIQUE EOPL+1 line of length j and splits it in half. If v_j of the given point p is the middle node of the line, the second half is returned.

Otherwise the first half is returned.

$$\begin{aligned}
 & \text{subline}: \{0, 1\}^{dn} \times \mathcal{I}^{\text{UNIQUE EOPL}+1} \rightarrow \mathcal{I}^{\text{UNIQUE EOPL}+1} \\
 \text{subline}(p = (v_1, \dots, v_n), I = (o, S, P, c, j)) & := \begin{cases} \text{firstHalf}(2^{j-1}, I), & \text{if } c(v_j) \neq 2^{j-1} \\ \text{secondHalf}(v_j, I), & \text{if } c(v_j) = 2^{j-1}. \end{cases}
 \end{aligned}$$

Each point $p = (v_1, v_2, \dots, v_n)$ of the hypercube Γ can be interpreted as node of the original instance I . The *decode* function takes a point of the hypercube C and constructs the corresponding node on the line of the UNIQUE EOPL+1 instance I . To do so, *subline* is called recursively n times and the line is split into halves until only one node is left. This node is returned as the node that was decoded by p .

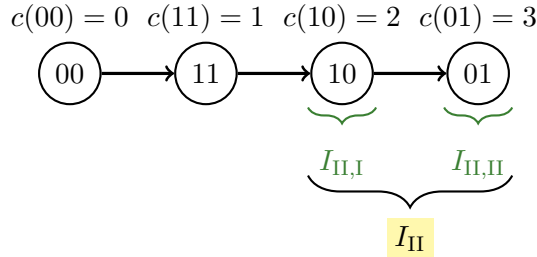
$$\begin{aligned}
 & \text{decode}: \{0, 1\}^{d \cdot n} \times \mathcal{I}^{\text{UNIQUE EOPL}+1} \rightarrow \{0, 1\}^d \\
 I' = (o', S', P', c', n') & := \underbrace{\text{subline}(p, \text{subline}(p, \dots \text{subline}(p, I) \dots))}_{n \text{ times}} \\
 \text{decode}(p, I) & := o'
 \end{aligned}$$

Note that the second half of the (sub)line in the j 'th step is only returned if v_j is the middle of the line. For all other nodes the first half is returned. Therefore, the first half is returned much more often while the second half is only returned once.

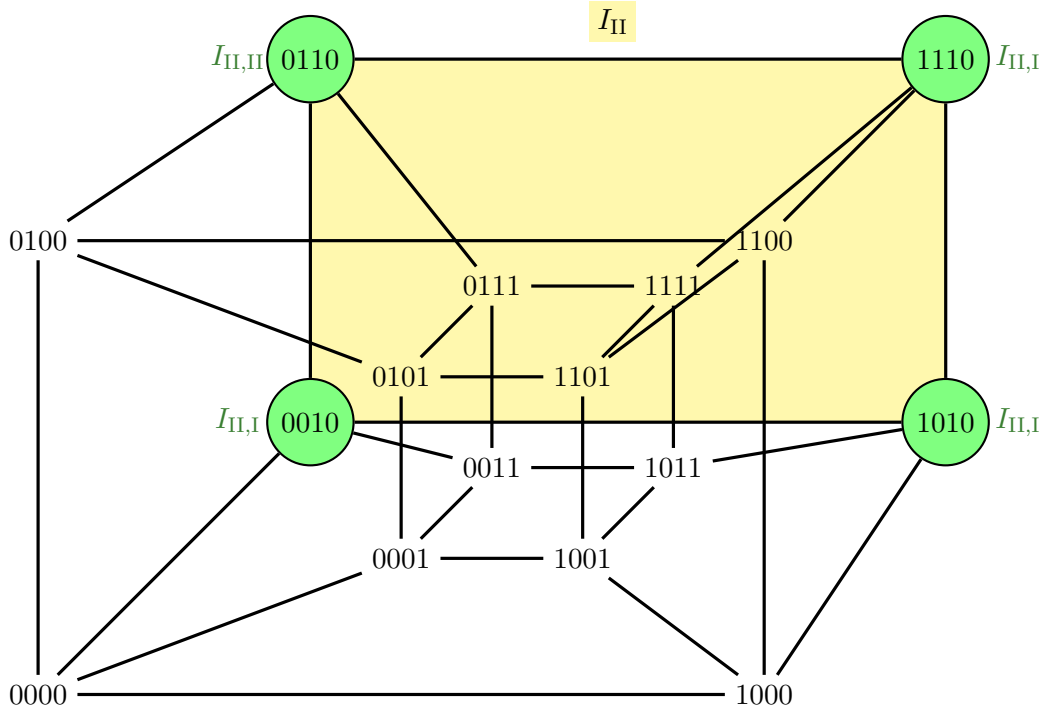
Only if a point consists (from the back to front) always of the middle node of the remaining line, it encodes the end of the original line.

Example 4.1.35. $d = 2$

Let $S, P : \{0, 1\}^2 \rightarrow \{0, 1\}^2$ be defined such that they form the following line:



$n = 2$, since (01) is the end of the line and therefore a solution with $c(01) = 2^2 - 1$. It follows that $\Gamma = \{0, 1\}^4$.



The node $(v_1, v_2) = (1110)$ can be interpreted as follows: $v_2 = (10)$ with $c(10) = 2 = 2^{n-1}$. Therefore (10) marks the beginning of the second half of the original line. $v_1 = (11)$ with $c_{II}(11) = -1 \neq 2^{1-1}$. This means that (11) is not the beginning of the second half of the second half of the line ((01) is). It isn't even the first half of the second half of the line (which is (10)) but still (10) is the node which is represented by (1110) : $decode(1110) = (10)$.

$I_{II} = \text{subline}(1110, I)$ is the instance: $c_{II}(10) = 0 \quad c_{II}(01) = 1$

$\text{subline}(1110, I_{II}) = (10) = I_{II,I}$.

$decode(1110, I) = (10)$.

The following point encodes the last node on the line. It is the only point that encodes that node.

$\text{subline}(0110, I_{II}) = (01) = I_{II,II}$.

$decode(0110, I) = (01)$.

The direction functions $D : \{0, 1\}^{dn} \rightarrow \{\text{Up}, \text{Down}, \text{Zero}\}^{dn}$.

There is only one node in Γ that corresponds to the end of the line in I . The direction functions of the other nodes need to point towards that node.

For $p = (v_1, \dots, v_n)$, $i = 1, \dots, n$ and $j = 1, \dots, d$, let be

$$I_n(p) := I$$

$$I_i(p) = (o_i, S_i, P_i, c_i, i) := \underbrace{\text{subline}(p, \text{subline}(p, \dots \text{subline}(p, I) \dots))}_{(n-i) \text{ times}}.$$

I_i is the instance that results in calling *subline* for the elements v_n to v_i .

Construction of $D_{d(i-1)+j}(p)$:

1. If $c_i(v_i) \neq 2^{i-1}$, it is the case that v_i is not the first node on the second half of $I_i(p)$ and therefore $decode((v_1, \dots, v_n), I)$ is a vertex in the first half of $I_i(p)$.
 - (a) $c_i(decode((v_1, \dots, v_n), I)) = 2^{i-1} - 1$ means that $decode(p, I)$ is the last vertex on the first half of the line $I_i(p)$. Then we want the direction function to point towards $S(decode((v_1, \dots, v_n), I))$:

$$D_{d(i-1)+j}(p) = \begin{cases} \text{Up} & \text{if } (v_i)_j = 0 \wedge S(decode((v_1, \dots, v_n), I))_j = 1 \\ \text{Down} & \text{if } (v_i)_j = 1 \wedge S(decode((v_1, \dots, v_n), I))_j = 0 \\ \text{Zero} & \text{otherwise.} \end{cases}$$

- (b) $c_i(decode((v_1, \dots, v_n), I)) \neq 2^{i-1} - 1$. This means that $decode(p, I)$ is in the first half of $I_i(p)$ and specifically it is *not* the last node in that half. Everything is oriented towards 0.

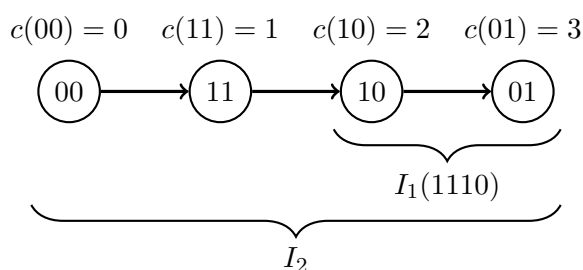
$$D_{d(i-1)+j}(p) = \begin{cases} \text{Down} & \text{if } (v_i)_j = 1 \\ \text{Zero} & \text{if } (v_i)_j = 0. \end{cases}$$

2. If $c_i(v_i) = 2^{i-1}$, then v_i is the first node on the second half of $I_i(p)$ and therefore $decode((v_1, \dots, v_n), I)$ is on the second half of the line $I_i(p)$. For all $j = 1, \dots, d$, set

$$D_{d(i-1)+j}(p) = \text{Zero}.$$

Example 4.1.36. $d = 2$

Recall the UNIQUE EOPL+1 instance from Example 4.1.35 on page 50. Let $S, P : \{0, 1\}^2 \rightarrow \{0, 1\}^2$ be defined such that they form the following line:



Let's take a closer look at (1110) and $i = 1$.

$c_1(11) = c(11) - 2^{2-1} = -1 \neq 2^{i-1} \Rightarrow$ case 1, $decode(1110, I) = (10)$ is a vertex in the first half of $I_1(1110)$. Note that $I_1(1110)$ is the second half of the line of I .

$c_1(decode(1110, I)) = c_1(10) = c(10) - 2^{2-1} = 0 = 2^{i-1} - 1 \Rightarrow$ case 1a. This means that $decode(1110) = (10)$ is the last vertex of the first half of the line $I_1(1110)$.

$(v_1)_1 = 1$ and $S(decode(1110, I))_1 = S(10)_1 = (01)_1 = 0 \Rightarrow D_1(1110) = \text{Down}$.

$(v_1)_2 = 1$ and $S(decode(1110, I))_2 = S(10)_2 = (01)_2 = 1 \Rightarrow D_2(1110) = \text{Zero}$.

For (1110) and $i = 2$ case 2 holds. The node (10) is the first node in the second half of the line I_2 , therefore $D_3(1110) = D_4(1110) = \text{Zero}$.

Here is a complete overview of the direction function for this example:

$D(0000) = [\text{Up}, \text{Up}, \text{Zero}, \text{Zero}]$	$decode(0000) = 00$
$D(0001) = [\text{Up}, \text{Up}, \text{Zero}, \text{Down}]$	$decode(0001) = 00$
$D(0010) = [\text{Zero}, \text{Up}, \text{Zero}, \text{Zero}]$	$decode(0010) = 10$
$D(0011) = [\text{Up}, \text{Up}, \text{Down}, \text{Down}]$	$decode(0011) = 00$
$D(0100) = [\text{Up}, \text{Zero}, \text{Zero}, \text{Zero}]$	$decode(0100) = 00$
$D(0101) = [\text{Up}, \text{Zero}, \text{Zero}, \text{Down}]$	$decode(0101) = 00$
$D(0110) = [\text{Zero}, \text{Zero}, \text{Zero}, \text{Zero}]$	$decode(0110) = 01$
$D(0111) = [\text{Up}, \text{Zero}, \text{Down}, \text{Down}]$	$decode(0111) = 00$
$D(1000) = [\text{Zero}, \text{Up}, \text{Zero}, \text{Zero}]$	$decode(1000) = 00$
$D(1001) = [\text{Zero}, \text{Up}, \text{Zero}, \text{Down}]$	$decode(1001) = 00$
$D(1010) = [\text{Down}, \text{Up}, \text{Zero}, \text{Zero}]$	$decode(1010) = 10$
$D(1011) = [\text{Zero}, \text{Up}, \text{Down}, \text{Down}]$	$decode(1011) = 00$
$D(1100) = [\text{Zero}, \text{Zero}, \text{Up}, \text{Zero}]$	$decode(1100) = 11$
$D(1101) = [\text{Zero}, \text{Zero}, \text{Up}, \text{Down}]$	$decode(1101) = 11$
$D(1110) = [\text{Down}, \text{Zero}, \text{Zero}, \text{Zero}]$	$decode(1110) = 10$
$D(1111) = [\text{Zero}, \text{Zero}, \text{Zero}, \text{Down}]$	$decode(1111) = 11$

For the construction of $D_{d(i-1)+j}(p)$, the line is always split exactly at the middle. If the line was valid at the beginning, there will be no problems during the construction. If the line wasn't valid, for example due to shuffled cost of the node, this construction will run into some problems. In the following it will be proven that these problems can always be traced back to a violation in the UNIQUE EOPL+1 instance.

Correctness In order to prove the correctness of this construction, it must be shown that each solution to the constructed OPDC instance matches back to the correct solution of the UNIQUE EOPL+1 instance.

A solution of type (O1) Assume $p = (v_1, \dots, v_n)$ is a solution of type (O1). This means, $D(p) = (\text{Zero})^{dn}$. We will now prove that $decode(p, I)$ is a solution of type (U1.1) in the original UNIQUE EOPL+1 problem.

The argumentation in [Fea+18] is slightly wrong though. They state:

Since the dimensions corresponding to v_n are all zero, we know that $decode(p)$ is in the second half of the line [Fea+18, p.59, l.5f].

This is not true: It can happen that the direction function in the dimensions corresponding to v_n is **Zero** because of case (1b) and not because of case (2). Consider the line from Example 4.1.36 on the previous page. $D_3(0000) = D_4(0000) = \text{Zero}$ but $decode(0000) = (00)$ is not on the second half of the line, it is the start node.

In [Fea+20b] they correct this statement, but here an alternative proof will be given.

Lemma 4.1.37. *If for v_i case (1a) is entered, $\exists j \in \{1, \dots, d\} : D_{d(i-1)+j}(p) \neq \text{Zero}$.*

Proof. For v_i case (1a) is entered, therefore $c_i(v_i) \neq 2^{i-1}$ and $decode((v_1, \dots, v_n), I)$ is the last vertex on the first half of the line $I_i(p)$. It follows that $S(encode(p, I))$ is the first node on the second half of the line of $I_i(p)$. It must hold that $S(encode(p, I)) \neq v_i$ since v_i cannot be the first node on the second half of $I_i(p)$. If it was, case (2) would have been entered instead of case (1a). Hence there exists a $j \in \{1, \dots, d\}$ such that $S(encode(p, I))_j \neq (v_i)_j$ and therefore $D_{d(i-1)+j}(p) \neq \text{Zero}$.

□

Lemma 4.1.38. *If for any v_i case (1) is entered, there exists a v_k with $k \leq i$ for which case (1a) is entered.*

Proof. This will be proven by induction. Let v_i be the node with the highest index that enters case (1).

Base: $i = 1$

$I_1(p)$ is an instance with line length 2. Since $decode(p, I)$ is on the first half of the line $I_1(p)$, which only consists of just one node, it is automatically the last node on the first half of the line. This implies that case (1a) is entered for $k = i$.

Assumption: If for any v_i case (1) is entered, there exists a v_k with $k \leq i$ for which case (1a) is entered.

Induction: $i = i + 1$. v_{i+1} enters case (1).

(a) v_{i+1} enters case (1a). Then we are done.

(b) v_{i+1} enters case (1b).

- v_i enters case (1). By the induction assumption there exists a $k \leq i$ so that v_k enters case (1a). Therefore, there exists a $k \leq i + 1$ for which v_k enters case (1a).
- v_i enters case (2). We know that $decode(p, I)$ is in the first half of $I_{i+1}(p)$ and specifically is *not* the last node in that half. Therefore v_1, \dots, v_{i-1} cannot always enter case (2) since this would imply that $decode(p, I)$ is the last node on the line $I_{i+1}(p)$. So there is a $k \leq i$ for which v_k doesn't enter case (2) but case (1). It can't happen that for all v_1, \dots, v_{i-1} case (1b) is entered because by the induction assumption there there exists a k for which case (1a) is entered.

□

Lemma 4.1.39. *If $D(p) = (\text{Zero})^{dn}$, only case (2) is entered.*

Proof. Given $D(p) = (\text{Zero})^{dn}$.

Assume for some v_i case (1a) was entered. By Lemma 4.1.2.2 this implies that there exists a $j \in \{1, \dots, d\} : D_{d(i-1)+j}(p) \neq \text{Zero}$. This is a contradiction to the assumption that $D(p) = (\text{Zero})^{dn}$.

Assume case (1b) was entered. By Lemma 4.1.2.2 this implies that case (1a) is entered as well which implies the same contradiction as the previous case.

It follows that only case (2) is entered.

□

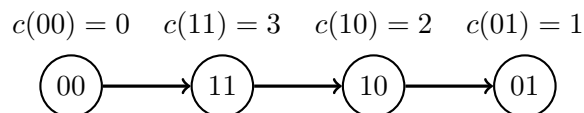
Since only case (2) is entered this implies that $decode(p, I)$ is in the second half of the second half of the second half ... of the line, which is the last node of the line. It also implies that $decode((v_1, \dots, v_n), I) = v_1$. Since case (2) implies that $c_1(v_1) = 1$ and by the definition of *secondHalf*:

$$\begin{aligned} c_n(u) &= c(u) - 2^{n-1} \\ c_{n-1}(u) &= c_n(u) - 2^{n-2} = c(u) - 2^{n-1} - 2^{n-2} \\ &\dots \\ c_1(u) &= c(u) \underbrace{-2^{n-1} - 2^{n-2} - \dots - 2}_{=-2^{n+2}} \end{aligned}$$

It follows that $c(decode(p, I)) = 2^n - 1$.

The construction allows us to find the node with the correct cost. This doesn't mean that the node is the last one on the line though. Consider for example a UNIQUE EOPL+1 instance in which the nodes form a line and all costs only exist once, but the costs are shuffled.

Example 4.1.40. An instance with shuffled costs:



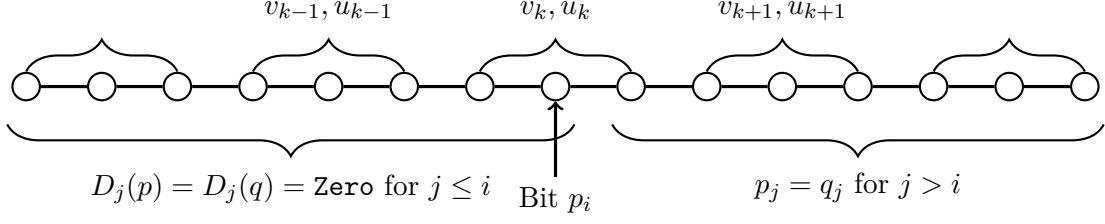
In this case a solution of type (01) would be $s = decode(1110) = (11)$ since it has the appropriate cost. Also it holds that $D(1110) = \mathbf{Zero}^4$ because the construction of the direction function works only with the cost and not the position of the node on the line. But (11) it is not the end of the line. To fix that issue, one must check for every found type (01) solution whether $P(S(s)) \neq s$ or not. If yes, then it truly is a (U1.1) solution. If not we found a violation of type (V1.1) or (V1.3) because it must hold that $c(S(s)) \leq c(s)$, since s has the highest possible cost.

A solution of type (OV1) When we are given a solution of type (OV1), there are two points $p = (v_1, \dots, v_n)$ and $q = (u_1, \dots, u_n)$ that are both fixpoints in the same i -Slice s :

- $p \neq q$ and
- $\forall j \leq i : D_j(p) = D_j(q) = \mathbf{Zero}$.

Consider v_k with $k = \lceil \frac{i}{d} \rceil$ to be the string that contains bit i . Because p and q are in the same i -Slice, it holds that $p_j = q_j$ for $j > i$ and

$$\begin{aligned} I_k(p) &:= \underbrace{subline(p, subline(\dots subline(p, I)))}_{(n-k) \text{ times}} \\ &= \underbrace{subline(q, subline(\dots subline(q, I)))}_{(n-k) \text{ times}} \\ &=: I_k(q). \end{aligned}$$



v_1 to v_{k-1} encode the last vertex on the subline $I_{k-1}(p)$. u_1 to u_{k-1} encode the last vertex on the subline $I_{k-1}(q)$.

The important difference of p and q is their difference in v_k and u_k .

(a) $v_k = u_k$.

This implies that $I_{k-1}(p) = I_{k-1}(q)$. Since $p \neq q$ there exists a $j < i$ such that $p_j \neq q_j$ and hence, there is a $l < k$ with $v_l \neq u_l$. It holds $c_l(v_l) = 2^{l-1} = c_l(u_l)$ because of Lemma 4.1.39 on page 54 we know that for v_l and u_l case (2) is entered. This implies that $c(v_l) = c(u_l)$ which is a violation-solution of type (V1.3).

(b) $v_k \neq u_k$ and $\forall l = d(k-1) + 1, \dots, dk: D_l(p) = \mathbf{Zero}$.

This means that the direction function for each bit in v_k is **Zero**. This and the fact that $D_j(p) = \mathbf{Zero}$ for all $j \leq i$ combined with Lemma 4.1.39 on page 54, means that $I_{k-1}(p)$ is the second half of $I_k(p)$ and $c_k(v_k) = 2^{k-1}$. $I_{k-1}(q)$ cannot be on the second half of $I_k(p)$ as well because there is only one bit string v_k for which the second half is entered and $v_k \neq u_k$.

$decode(q, I)$ represents the last vertex on the first half of the line of $I_k(q)$ because, as argued before, $decode(q, I)$ is in the first half of the line and furthermore all direction functions $D_j(q) = \mathbf{Zero}$ for all $j \leq i$. Therefore for u_k case (1a) is entered.

$S(decode(q, I))$ then should be the first node on the second half of line $I_k(q, I)$. If $c_k(S(decode(q, I))) \neq 2^{k-1}$ this means that $S(decode(q, I))$ is *not* the first node on the second half of the line. Therefore we found a (V1.1) violation.

If $c_k(S(decode(q, I))) = 2^{k-1} = c_k(v_k)$, we found a violation of type (V1.3) as long as $S(decode(q, I)) \neq v_k$, which will be proved in the following:

$\exists l: d(k-1) + 1 \leq l \leq i \wedge p_l \neq q_l \wedge D_l(p) = D_l(q) = \mathbf{Zero}$. Because for u_k case (1a) is entered and $D_l(q) = \mathbf{Zero}$, we know that $(u_k)_l = S(decode(q, I))_l$. Since $(v_k)_l \neq (u_k)_l$ it follows that $v_k \neq S(decode(q, I))$.

(c) $v_k \neq u_k$ and $\forall l = d(k-1) + 1, \dots, dk: D_l(q) = \mathbf{Zero}$.

This case is symmetric to case (b).

(d) $v_k \neq u_k$, $\exists l_1 \in \{d(k-1) + 1, \dots, dk\}: D_{l_1}(p) \neq \mathbf{Zero}$ and $\exists l_2 \in \{d(k-1) + 1, \dots, dk\}: D_{l_2}(q) \neq \mathbf{Zero}$.

It holds that $l_1 > i$ and $l_2 > i$. Furthermore $I_{k-1}(p) = I_{k-1}(q)$ because for both v_k and u_k case (1) is entered because there is a direction which is not **Zero**. More specifically for both nodes case (1a) is entered since $D_j(q) = \mathbf{Zero}$ for all $j \leq i$ and Lemma 4.1.39 on page 54. Thus, the direction function for point v_k points towards $S(decode(p, I))$ and for point u_k it points towards $S(decode(q, I))$. It also holds that $c_k(decode(p, I)) = c_k(decode(q, I))$ which is a violation of type (V1.3) if $decode(p, I) \neq decode(q, I)$ which will be proven in the following:

There exists an $l \in \{d(k-1) + 1, \dots, i\}$ such that $p_l \neq q_l$ and $D_l(p) = D_l(q) = \mathbf{Zero}$. Let l' be the position of p_l in v_k . It follows that $(v_k)_{l'} = S(decode(p, I))_{l'}$ and

$(u_k)_{i'} = S(\text{decode}(q, I))_{i'}$. Therefore $S(\text{decode}(p, I)) \neq S(\text{decode}(q, I))$ and hence $\text{decode}(p, I) \neq \text{decode}(q, I)$.

A solution of type (OV2) Given two solutions $p = (v_1, \dots, v_n)$ and $q = (u_1, \dots, u_n)$ in an i – Slice s with the following properties:

- $\forall j < i : D_j(p) = D_j(q) = \text{Zero}$,
- p_i and q_i are adjacent in the grid: $p_i = q_i + 1$,
- $D_i(p) = \text{Down}$ and $D_i(q) = \text{Up}$.

Consider v_k with $k = \lceil \frac{i}{d} \rceil$ to be the string that contains bit i .

Because p and q are in the same i – Slice, it holds that $p_j = q_j$ for $j > i$ and

$$\begin{aligned} I_k(p) &:= \underbrace{\text{subline}(p, \text{subline}(\dots \text{subline}(p, I)))}_{(n-k) \text{ times}} \\ &= \underbrace{\text{subline}(q, \text{subline}(\dots \text{subline}(q, I)))}_{(n-k) \text{ times}} \\ &=: I_k(q) \end{aligned}$$

Since $D_i(p) \neq \text{Zero}$ and $D_i(q) \neq \text{Zero}$, both $\text{decode}(p, I)$ and $\text{decode}(q, I)$ are points on the first half of the line of $I_k(p)$. Furthermore, $\text{decode}(p, I)$ and $\text{decode}(q, I)$ must be at the end of the first half of line $I_k(p)$ because $\forall j < i : D_j(p) = D_j(q) = \text{Zero}$ and Lemma 4.1.39 on page 54.

It follows that $c_k(\text{decode}(p, I)) = c_k(\text{decode}(q, I)) = 2^{k-1} - 1$ and therefore $c(\text{decode}(p, I)) = c(\text{decode}(q, I))$. If we prove now that $\text{decode}(p, I) \neq \text{decode}(q, I)$ we found a (V1.3) solution:

We know that for both p_i and q_i the construction uses case (1a). $D_i(p)$ points towards $S(\text{decode}(p, I))$ and $D_i(q)$ points towards $S(\text{decode}(q, I))$. But they both point into different directions because it is given that $D_i(p) = \text{Up}$ and $D_i(q) = \text{Down}$. Therefore $S(\text{decode}(p, I)) \neq S(\text{decode}(q, I))$ which implies that $\text{decode}(p, I) \neq \text{decode}(q, I)$.

A solution of type (OV3) A solution $p \in \{0, 1\}^{dn}$ of type (OV3) requires that there exists an $i \in \{1, 2, \dots, dn\}$ with either

- $p_i = 0$ and $D_i(p) = \text{Down}$ or
- $p_i = 1$ and $D_i(p) = \text{Up}$.

But the construction never sets $D_i(p)$ to **Down** when $p_i = 0$ and it never sets $D_i(p)$ to **Up** when $p_i = 1$. Therefore solutions of type (OV3) are by the definition of the construction impossible.

Conclusion Every solution that is found in the constructed OPDC instance can be mapped back to the corresponding UNIQUE EOPL+1 instance in polynomial time.

The reduction is promise preserving because all end-of-line solutions (U1.1) are only mapped to (O1) solutions and all violations of the OPDC instance can be mapped back to a violation of the original UNIQUE EOPL+1 instance. \square

5. Problems in UniqueEOPL

To prove that a problem is in UniqueEOPL, it needs to be shown that it can be reduced to a problem that already is in UniqueEOPL.

In this chapter an overview over all currently known problems that are in UniqueEOPL is presented. For each problem a definition and an example is given, as well as a reference to the reduction that proves it to be in UniqueEOPL. For S-ARRIVAL, CUBE-USO and P-LCP, detailed proves are given.

For none of the problems listed below there is a polynomial time algorithm known. Every problem can be trivially solved by exhaustive search, which is of course not very efficient. Some of the problems can be solved by subexponential algorithms. [Ald83] introduced an algorithm to solve LOCAL OPT in $\mathcal{O}(1,4143^d) = \mathcal{O}((\sqrt{2})^d)$ time. The algorithm picks $2^{d/2}$ random nodes, chooses the one with maximal cost and starts to follow the line from there on. This algorithm can solve any problem in PLS and therefore any problem in UniqueEOPL [Fea+18, Section 5.2].

Problem in UniqueEOPL	Proven by reduction to	Reference
α -HAM SANDWICH	\preceq_{PROMISE} UNIQUE END OF POTENTIAL LINE	[CCM20, Theorem 10]
BINARY SIMPLE STOCHASTIC GAME (BINARY-SSG)	\preceq_{PROMISE} P-LCP	[GR05]
BIPARTITE-MPG	\preceq_{PROMISE} MEAN PAYOFF GAME (MPG)	[ZP96, Section 1]
CUBE-USO	\preceq_{PROMISE} OPDC	Theorem 5.2.12 on page 72
CUBE-USO- \perp	\preceq_{PROMISE} OPDC \preceq_{PROMISE} CUBE-USO	[Fea+20b, Lemma 23] [Fea+20b, Lemma 22]
DISCOUNTED GAME	\preceq_{PROMISE} SSG	[ZP96, Section 6]
FINITE-MPG	\preceq_{PROMISE} MEAN PAYOFF GAME (MPG)	[EM79]
GRID-USO	\preceq_{PROMISE} UNIQUE FORWARD EOPL	Theorem 6.3 on page 108
MEAN PAYOFF GAME (MPG)	\preceq_{PROMISE} DISCOUNTED GAME	[ZP96, Section 5]
ONE PERMUTATION DISCRETE CONTRACTION (OPDC)	\preceq_{PROMISE} UNIQUE FORWARD EOPL	[Fea+20b, Lemma 11]
PAIRWISE LINEAR CONTRACTION (PL-CONTRACTION)	\preceq_{PROMISE} OPDC	[Fea+20b, Lemma 29]
PARITY GAME	\preceq_{PROMISE} MEAN PAYOFF GAME (MPG) \preceq_{PROMISE} SSG	[Pur95, Section 5.5] [CF11]
P-MATRIX LINEAR COMPLEMENTARITY PROBLEM (P-LCP)	\preceq_{PROMISE} CUBE-USO- \perp	[Fea+20b, Theorem 33]

P-GENERAL-LCP (P-GLCP)	\preceq GRID-USO	[Rüs07, Theorem 3.5]
S-ARRIVAL	\preceq_{PROMISE} END OF METERED LINE (And the fact that the problem has by definition a unique solution) \preceq_{PROMISE} UNIQUE FORWARD EOPL+1	[Gär+18, Theorem 20] [Fea+20b, p.3] Theorem 5.1.13 on page 64
SIMPLE STOCHASTIC GAME (SSG)	\preceq_{PROMISE} BINARY-SSG \preceq_{PROMISE} PL-CONTRACTION	[ZP96, p. 12] [EY10, Corollary 5.3]
UNIQUE END OF POTENTIAL LINE	\preceq_{PROMISE} UNIQUE EOPL+1	[Fea+20b, Lemma 17]
UNIQUE EOPL+1	\preceq_{PROMISE} OPDC	[Fea+20b, Lemma 19]
UNIQUE FORWARD EOPL	\preceq_{PROMISE} UNIQUE FORWARD EOPL+1	[Fea+20b, Lemma 13]
UNIQUE FORWARD EOPL+1	\preceq_{PROMISE} UNIQUE END OF POTENTIAL LINE	[Fea+20b, Lemma 15]

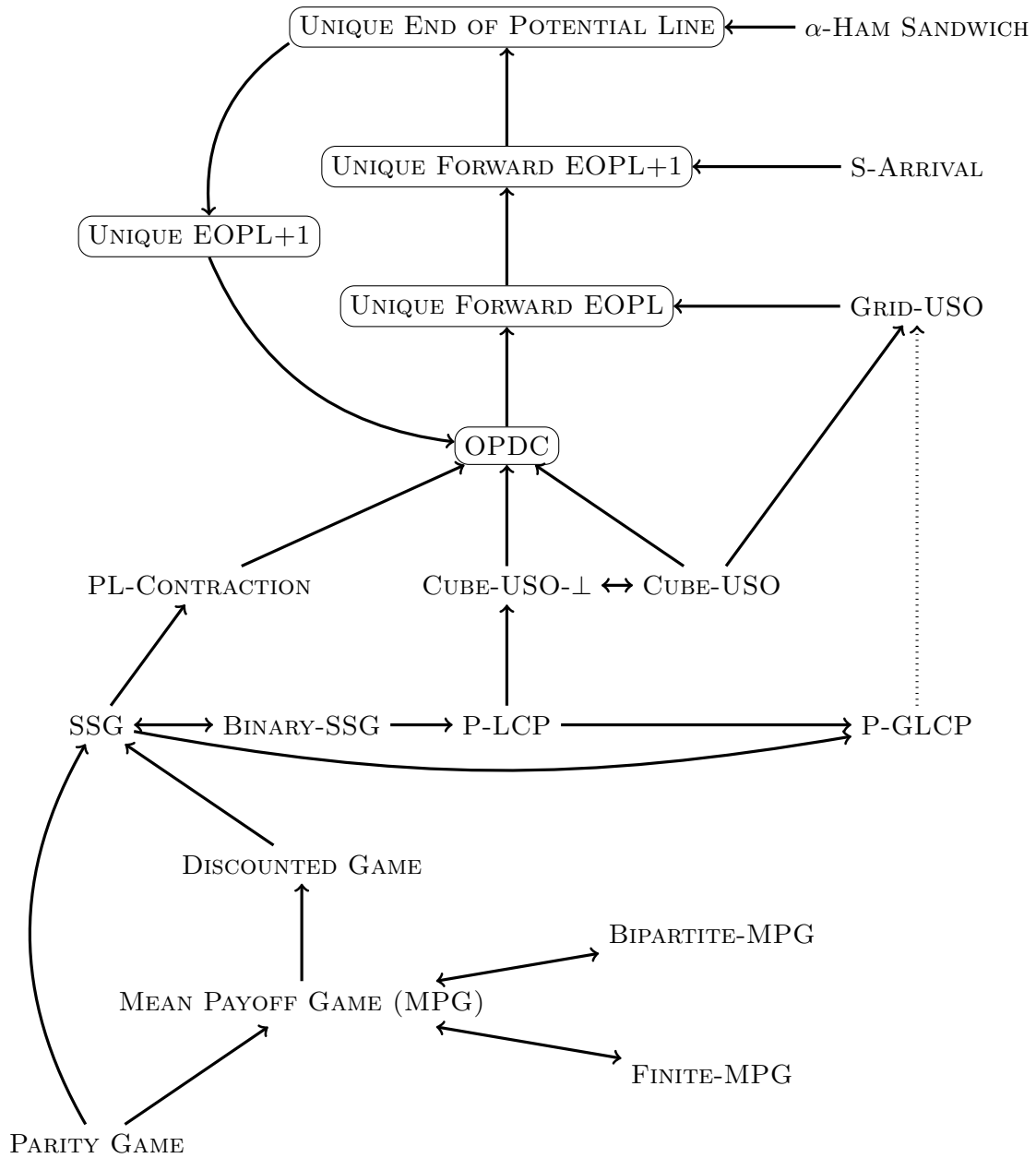


Figure 5.1.: Overview of reductions for problems in UniqueEOPL. The problems in the boxes are UniqueEOPL-complete. A solid arrow means there exists a promise preserving reduction from problem L to problem Q: $L \rightarrow Q$. A dashed arrow means there exists a reduction, but it is not promise preserving.

5.1. S-Arrival

The ARRIVAL Problem was first introduced in [Doh+16]. It works on a directed switch graph in which each vertex has two outgoing edges. While traversing the graph, each of the outgoing edges is used in turns. The question for a given start node is whether a certain end node can be reached or not.

The idea was originally inspired by a railway network, in which each switch changes its position immediately after the train passed it. The question is whether or not the train reaches its destination.

ARRIVAL can be viewed as a deterministic version of a random walk, similar to the *Rotor-Router model* (also called *Eulerian Walk* [Pri+96]). In a random walk on a graph one chooses randomly the edge along which to move. The question whether (and after what expected time) a node β is reached starting at a node α is called the *hitting time* or *access time* [Lov+93]. In the Rotor-Router model, for each node the outgoing edges are ordered in a sequence in which they are traversed deterministically during the walk [FS10]. ARRIVAL is therefore a Rotor-Router model in which each node has exactly two outgoing edges.

Definition 5.1.1 (Switch graph). A switch graph is a 4-tuple $G = (V, E, S_0, S_1)$ with $|V| = d$, $S_0, S_1 : V \rightarrow V$ and $E := \{(v, S_0(v)) \mid v \in V\} \cup \{(v, S_1(v)) \mid v \in V\}$. It might be that $S_0(v) = S_1(v)$. Cycles are allowed.

$(v, S_0(v))$ are called *even* edges and represented by a solid line in the examples, whereas $(v, S_1(v))$ are called *odd* edges and drawn as a dashed line.

Definition 5.1.2 (ARRIVAL (Decision problem)). Given a switch graph $G = (V, E, S_0, S_1)$, an origin $\alpha \in V$ and a destination $\beta \in V$, find out if the following procedure terminates, i.e. if there is a path from α to β .

Algorithm 1: RUN-ARRIVAL

Data: G, α, β
 $\forall v \in V: \mathbf{s_curr}[v] := S_0(v);$
 $\forall v \in V: \mathbf{s_next}[v] := S_1(v);$
 $\forall e \in E: \mathbf{count}[e] = 0;$
 $\mathbf{x} := \alpha;$
while $\mathbf{x} \neq \beta$ **do**
 $\mathbf{y} := \mathbf{s_curr}[\mathbf{x}];$
 $\mathbf{count}[(\mathbf{x}, \mathbf{y})]++;$
 $\text{swap}(\mathbf{s_curr}[\mathbf{x}], \mathbf{s_next}[\mathbf{x}]);$
 $\mathbf{x} := \mathbf{y};$

Example 5.1.3. In the following, we are given two switch graphs, G and G' . They differ only slightly: $S_0(v_1) = v_3$ whereas $S'_0(v_1) = v_2$, and $S_1(v_1) = v_2$ whereas $S'_1(v_1) = v_3$. For G , the algorithm RUN-ARRIVAL terminates, but for G' it doesn't. In G' , the algorithm is caught in an endless loop in v_4 .

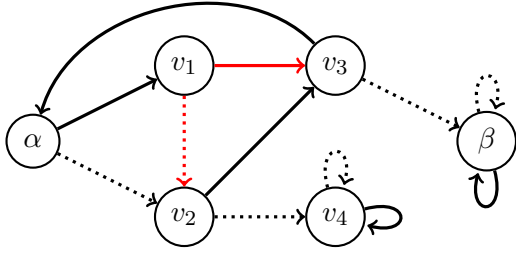


Figure 5.2.: Switch graph G

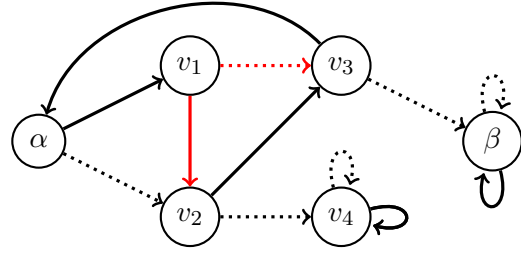


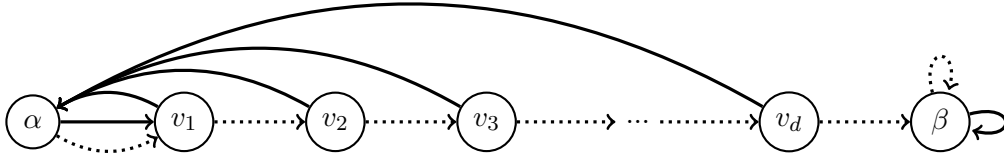
Figure 5.3.: Switch graph G'

Lemma 5.1.4. *ARRIVAL is decidable [Doh+16, Theorem 1].*

Proof. RUN-ARRIVAL is deterministic. Let a **state** of a graph G be the current position of the train and the state of all switches: $\text{state} := (x, \mathbf{s}_{\text{curr}}, \mathbf{s}_{\text{next}})$. There are at most $d2^d$ many different states. Either RUN-ARRIVAL terminates after at most that many steps or if it doesn't, there must be a state that occurred at least twice, which means RUN-ARRIVAL entered an infinite loop. Hence, after $d2^d$ iterations of RUN-ARRIVAL we know whether there is a path from α to β or not [Doh+16].

□

The following graph has exponentially many states:



Note that if an instance of the ARRIVAL problem has a solution (i.e. a path from α to β) that solution is unique.

ARRIVAL is in NP [Doh+16, Theorem 2] and also in co-NP [Doh+16, Theorem 3].

Definition 5.1.5 (Switching flow [Doh+16]). Let $G = (V, E, S_0, S_1)$ be a switch graph and $E^+(v)$ denote the outgoing edges of vertex v and $E^-(v)$ the incoming edges.

For start node α and an end node β , let $s_{\alpha,\beta} \in \mathbb{N}_0^{|E|}$ be a vector where each entry assigns a value to each edge. Let $s_{\alpha,\beta}(v, u)$ reference the entry of s that represents the value assigned to the edge (v, u) . The vector $s_{\alpha,\beta}$ is called *switching flow* if for all entries and all nodes the following two conditions hold:

(i)

$$\sum_{e \in E^+(v)} s_{\alpha,\beta}(e) - \sum_{e \in E^-(v)} s_{\alpha,\beta}(e) = \begin{cases} 1 & v = \alpha \\ -1 & v = \beta \\ 0 & \text{otherwise.} \end{cases}$$

This condition essentially states that each node that is entered also is left again, except for the origin node α (which is left once more than any other node) and the destination node β (which is entered once more than any other node).

(ii) $\forall v \in V : 0 \leq s_{\alpha,\beta}(v, S_1(v)) \leq s_{\alpha,\beta}(v, S_0(v)) \leq s_{\alpha,\beta}(v, S_1(v)) + 1$

This condition ensures that the number of times that the even and the odd edges of one node were traversed, differs at most by 1.

For two arbitrary nodes a and b that are not necessarily α and β , $s_{a,b}$ is called an *intermediate switching flow* [Gär+18, Definition 3, also called partial switching flow].

Note that there are infinitely many (intermediate) switching flows for two vertices a and b . Given any switching flow, adding the vector $n \cdot 1^{|E|}$ for an arbitrary $n \in \mathbb{N}^+$ to $s_{a,b}$ preserves the two conditions.

It is easy to verify whether or not a vector $s_{a,b} \in \mathbb{N}_0^{|E|}$ is a valid switching flow for a given graph: Simply test these two conditions. This can be done in polynomial time [Doh+16]. The end vertex of a switching flow can be calculated in polynomial time by checking which edge is the one for which condition (i) is -1 [Gär+18, Observation 6].

Definition 5.1.6 (Run profile [Doh+16]). A run profile $r \in \mathbb{N}_0^{|E|}$ is the **count** array returned by RUN-ARRIVAL upon termination. It assigns to each edge of a switch graph the number of times it has been traversed during the RUN-ARRIVAL procedure.

An *intermediate run profile* \tilde{r} corresponds to some intermediate value of the **count** array after a completed run of the while loop of the RUN-ARRIVAL algorithm [Gär+18, Definition 4, also called partial run profile]. The end vertex b of an intermediate run profile can be calculated in polynomial time.

Lemma 5.1.7. *Every (intermediate) run profile is an (intermediate) switching flow [Doh+16, Observation 1]. Not every (intermediate) switching flow is an (intermediate) run profile.*

Example 5.1.8. Consider the following example from [Doh+16].

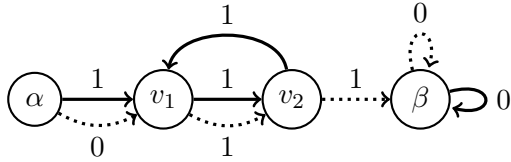


Figure 5.4.: Valid switching flow and run profile

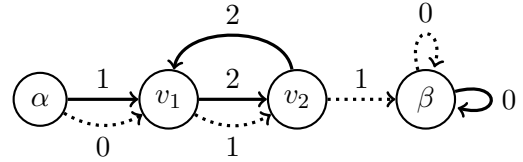


Figure 5.5.: Valid switching flow but not run profile

Lemma 5.1.9. *For a switch graph $G = (V, E, S_0, S_1)$ and an origin and a destination node $\alpha, \beta \in V$ with $\alpha \neq \beta$, if there exists a valid switching flow $s_{\alpha,\beta}$ then RUN-ARRIVAL terminates. For the run profile r returned upon termination it holds that $r \leq s_{\alpha,\beta}$ component wise [Doh+16, Lemma 1].*

Proof. Imagine we put on each edge as many pebbles as $s_{\alpha,\beta}$ tells us to. During the run of RUN-ARRIVAL we collect a pebble whenever traversing an edge. It is claimed that we never run out of pebbles, which would prove the above Lemma.

First observe that during a run when we are at a vertex x , condition (i) and (ii) of the intermediate switching flow $s_{\alpha,x}$ with origin α and destination x hold. Starting with the

even successor we take a pebble from each edge in turns, so the number of pebbles on them never differs by more than 1.

The claim is proven by contradiction. Assume $e = (v, u)$ is the first edge at which we run out of pebbles, i.e. e now has -1 pebbles left on it. By the observation above, the other outgoing edge of v must have 0 pebbles left. Since condition (i) held before v was entered and both outgoing edges had 0 pebbles, this means that the incoming edge of v must have had 0 pebbles as well. This is a contradiction because if the incoming edge has 0 pebbles, we could have never entered node v [Doh+16]. \square

Lemma 5.1.10. *It is possible to verify in polynomial time whether a given (intermediate) switching flow s is an (intermediate) run profile [Gär+18, Lemma 11].*

Definition 5.1.11 (S-ARRIVAL (Search problem) [Gär+18]). Given a switch graph $G = (V, E, S_0, S_1)$, an origin $\alpha \in V$ and a destination $\beta \in V$, find one of the following:

- (A1) A run profile r from α to β .
- (A2) An intermediate run profile \tilde{r} from α to any vertex b such that there exists an edge e that was traversed too often: $\tilde{r}(e) = 2^d + 1$. The current state of the switches must have existed already.

There is no polynomial time algorithm known for S-ARRIVAL. [Gär+18] proved that the Aldous algorithm from [Ald83] can be used to solve S-ARRIVAL. Its runtime is $\mathcal{O}((\sqrt{2})^d)$.

5.1.1. S-Arrival to Unique Forward EOPL+1

Theorem 5.1.12. *S-ARRIVAL can be reduced to END OF METERED LINE and is thus in CLS [Gär+18, Theorem 20].*

For a definition of END OF METERED LINE see 3.3.5 on page 18.

Since END OF METERED LINE and END OF POTENTIAL LINE are polynomial time equivalent, S-ARRIVAL is in EOPL [Fea+18, Theorem 10]. Furthermore, S-ARRIVAL has by definition a unique solution and is therefore in UniqueEOPL [Fea+18, Theorem 4]. Nonetheless, it is possible to reduce S-ARRIVAL to UNIQUE FORWARD EOPL+1 by a very similar but easier construction than done in [Gär+18]. Basically it is the same construction without the need to construct a predecessor circuit.

For a definition of UNIQUE FORWARD EOPL+1, see Definition 4.1.24 on page 43.

Theorem 5.1.13. *S-ARRIVAL can be reduced to UNIQUE FORWARD EOPL+1 in polynomial time and is thus in UniqueEOPL.*

Proof. Given an S-ARRIVAL instance $I = (G = (V, E, S_0, S_1), \alpha, \beta)$, create in polynomial time an instance of UNIQUE FORWARD EOPL+1 $I' = (S, c)$.

The nodes Let be $d' := |E| = 2|V| = 2d$. A node $s_{\alpha,b}$ of the UNIQUE FORWARD EOPL+1 will be an intermediate switching flow with $s_{\alpha,b} \in \{0, 1, \dots, 2^d + 1\}^{d'}$. The start node of I' is $0^{d'}$.

The successor function S By Lemma 5.1.10 on the previous page, there is a polynomial time function $\text{isRunProfile}: \{0, 1, \dots, 2^d + 1\}^{d'} \rightarrow \{\text{true}, \text{false}\}$.

$S(s_{\alpha,b})$ is defined as follows:

- (1) If $\text{isRunProfile}(s_{\alpha,b}) = \text{false}$ then $S(s_{\alpha,b}) = s_{\alpha,b}$.
If the given point is not a valid run profile, it will be a self loop.
- (2) If $\text{isRunProfile}(s_{\alpha,b}) = \text{true}$ and there $\exists e \in E: s_{\alpha,b}(e) = 2^d + 1$ then $S(s_{\alpha,b}) = s_{\alpha,b}$.
If this is the case, the RUN-ARRIVAL procedure took too many steps, by Lemma 5.1.4 on page 62 this means there is no way from α to β .
- (3) If $\text{isRunProfile}(s_{\alpha,b}) = \text{true}$ then let b be the end vertex of the current switching flow.
 - (1) If $b = \beta$ then $S(s_{\alpha,b}) = s_{\alpha,b}$.
We reached the destination. Therefore this is a valid end of the line in UNIQUE FORWARD EOPL+1.
 - (2) If $b \neq \beta$ then let $n \in \{0, 1\}$ be the variable that defines if the even or the odd successor is supposed to be traversed next: $n := s_{\alpha,b}(b, S_0(b)) - s_{\alpha,b}(b, S_1(b))$. Let u be the next node that RUN-ARRIVAL traverses: $u := S_n(b)$. Increase the vector $s_{\alpha,b}$ at the position that stands for the edge (b, u) by one and leave everything else unchanged. Now u is the new end vertex of the current switching flow $s_{\alpha,u}$.

The cost function c The cost function returns 0 if the given point is no valid run profile. Otherwise it returns the number of steps plus 1 that RUN-ARRIVAL made so far to reach the end node b of $s_{\alpha,b}$. The costs are calculated by summing up all entries of $s_{\alpha,b}$ plus 1, so that we can differ $0^{d'}$ from invalid run profiles.

$$c(s_{\alpha,b}) := \begin{cases} 0 & \text{if } \text{isRunProfile}(s_{\alpha,b}) = \text{false} \\ 1 + \sum_{i=1}^{d'} (s_{\alpha,b})_i & \text{if } \text{isRunProfile}(s_{\alpha,b}) = \text{true}. \end{cases}$$

The circuits S and c can be constructed in polynomial time.

Correctness In order to prove the correctness of this construction, it must be shown that each solution to the constructed UNIQUE FORWARD EOPL+1 instance matches back to the correct S-ARRIVAL solution.

A solution of type (UFP1) Let $s_{\alpha,b} \in \{0, 1, \dots, 2^d + 1\}^{d'}$ be a solution of type (UFP1) and $s_{\alpha,p} = S(s_{\alpha,b})$. This means that $S(s_{\alpha,b}) \neq s_{\alpha,b}$ and either $S(s_{\alpha,p}) = s_{\alpha,p}$ or $c(s_{\alpha,p}) \neq c(s_{\alpha,b}) + 1$.

First, assume that $S(s_{\alpha,b}) \neq s_{\alpha,b}$ and $S(s_{\alpha,p}) = s_{\alpha,p}$. It must hold that $\text{isRunProfile}(s_{\alpha,b}) = \text{true}$. Let's take a look at the rules that could have been used to set $S(s_{\alpha,p}) = s_{\alpha,p}$.

- (1) If $\text{isRunProfile}(s_{\alpha,p}) = \text{false}$ this means $s_{\alpha,p}$ is not a valid run profile. This case cannot happen, since $\text{isRunProfile}(s_{\alpha,b}) = \text{true}$ and the construction of $S(s_{\alpha,b})$ follows the same rules as RUN-ARRIVAL. Therefore $s_{\alpha,p}$ must be a valid run profile.

- (2) If $\exists e \in E: s_{\alpha,p}(e) = 2^d + 1$ then we found a valid solution of type (A2): There is no path from α to β .
- (3.1) If $\text{isRunProfile}(s_{\alpha,p}) = \text{true}$, it must hold that the end node of the switching flow p is equal to β . Therefore $s_{\alpha,p}$ is a valid solution of type (A1).

Second, assume that $S(s_{\alpha,b}) \neq s_{\alpha,b}$ and $c(s_{\alpha,p}) \neq c(s_{\alpha,b}) + 1$. This cannot happen. $S(s_{\alpha,b}) = s_{\alpha,p}$ was constructed by case (3.2), which means the potential increases by one.

A violation of type (UFVP1) If a violation of type (UFVP1) is found, there exist two points $s_{\alpha,b} \in \{0, 1, \dots, 2^d + 1\}^{d'}$ and $s_{\alpha,p} \in \{0, 1, \dots, 2^d + 1\}^{d'}$ with $s_{\alpha,b} \neq s_{\alpha,p}$, $s_{\alpha,b} \neq S(s_{\alpha,b})$, $s_{\alpha,p} \neq S(s_{\alpha,p})$ and $\text{isRunProfile}(s_{\alpha,b}) = \text{isRunProfile}(s_{\alpha,p}) = \text{true}$ and $c(s_{\alpha,b}) = c(s_{\alpha,p})$.

This cannot happen. Because $\text{isRunProfile}(s_{\alpha,b}) = \text{isRunProfile}(s_{\alpha,p}) = \text{true}$ it holds that $c(s_{\alpha,b}) = c(s_{\alpha,p}) > 0$. It will be proven by induction over the cost that if $c(s_{\alpha,b}) = c(s_{\alpha,p})$ it follows that $s_{\alpha,b} = s_{\alpha,p}$.

Base: $n = 1$, i.e. $c(s_{\alpha,b}) = c(s_{\alpha,p}) = 1$ This means that $s_{\alpha,b} = 0^{d'} = s_{\alpha,p}$.

Assumption: If $c(s_{\alpha,b}) = c(s_{\alpha,p}) = n$ then $s_{\alpha,b} = s_{\alpha,p}$.

Induction: $n = n+1$

Since $\text{isRunProfile}(s_{\alpha,b}) = \text{isRunProfile}(s_{\alpha,p}) = \text{true}$, this means that there is a point $s_{\alpha,x}$ such that $S(s_{\alpha,x}) = s_{\alpha,b}$ and a point $s_{\alpha,y}$ such that $S(s_{\alpha,y}) = s_{\alpha,p}$. It holds that $s_{\alpha,x} \neq s_{\alpha,b}$ and $s_{\alpha,y} \neq s_{\alpha,p}$ because otherwise it would be a contradiction to the assumption $S(s_{\alpha,b}) \neq s_{\alpha,b}$ and $s_{\alpha,p} \neq S(s_{\alpha,p})$.

$S(s_{\alpha,x})$ and $S(s_{\alpha,y})$ must have been constructed by rule (3.2.). It follows that $c(s_{\alpha,x}) - 1 = c(s_{\alpha,b}) = c(s_{\alpha,p}) = c(s_{\alpha,y}) - 1$ and by the assumption of the induction that means that $s_{\alpha,x} = s_{\alpha,y}$. This again implies that the same successor edge is chosen and therefore it follows that $s_{\alpha,b} = s_{\alpha,p}$.

This induction might seem trivial, but it is important to exclude the case that an arbitrary chosen switching flow does belong to another line.

A violation of type (UFVP2) Let $s_{\alpha,b}$ and $s_{\alpha,q}$ be the given vertices of the violation (UFVP2) with $s_{\alpha,b} \neq s_{\alpha,q}$, $S(s_{\alpha,q}) \neq s_{\alpha,q}$ and $c(s_{\alpha,b}) < c(s_{\alpha,q})$. Since $s_{\alpha,b}$ is also a solution of type (UFVP1), it holds that $S(s_{\alpha,b}) = s_{\alpha,p}$, $s_{\alpha,b} \neq s_{\alpha,p}$ and $S(s_{\alpha,p}) = s_{\alpha,p}$.

This cannot happen: Since $S(s_{\alpha,q}) \neq s_{\alpha,q}$, it must hold that $\text{isRunProfile}(s_{\alpha,q}) = \text{true}$. This again means that there exists a point $s_{\alpha,x}$ with $S(s_{\alpha,x}) = s_{\alpha,q}$ which has exactly one cost less than $s_{\alpha,q}$. This argument can be repeated recursively until there is a predecessor $s_{\alpha,y}$ of $s_{\alpha,q}$ with $c(s_{\alpha,y}) = c(s_{\alpha,b})$. By the induction proof from the previous case, this implies that $s_{\alpha,b}$ is a predecessor of $s_{\alpha,q}$. This is a contradiction to the fact that $s_{\alpha,b}$ is the end of the line.

Conclusion All in all, every solution that is found in the constructed UNIQUE FORWARD EOPL+1 instance can be mapped back to a solution of the corresponding S-ARRIVAL instance in polynomial time.

The reduction is promise preserving. Every solution of the S-ARRIVAL instance is only mapped to solutions of type *(UFP1)*. For every violation of the created UNIQUE FORWARD EOPL+1 instance it can be shown that this cannot possibly be constructed from an S-ARRIVAL instance.

It follows that the promise version of S-ARRIVAL is in PromiseUEOPL.

□

5.2. Unique Sink Orientation of Cubes (Cube-USO)

Let $C = \{0, 1\}^d$ be a d -dimensional hyper cube, where each bit-string represents one vertex of the cube.

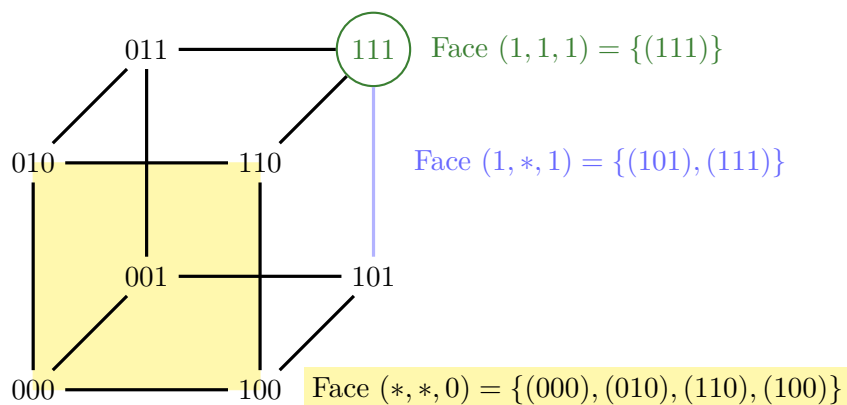
Definition 5.2.1 (Face of a hypercube). A face (or a subcube) C_α of a hypercube $C = \{0, 1\}^d$ is defined by a tuple $C_\alpha := (\mathbf{f}_1, \mathbf{f}_2, \dots, \mathbf{f}_d)$ with $\mathbf{f}_i \in \{0, 1, *\}$ for $i = 1, \dots, d$. α is the set containing all indices at which C_α is not $*$: $\alpha := \{i \mid \mathbf{f}_i \neq *\}$. If $\mathbf{f}_i \in \{0, 1\}$, this means that dimension i is fixed. If $\mathbf{f}_i = *$, dimension i is free to vary.

$v \in C$ is in face C_α if and only if $\forall i \in \alpha: v_i = \mathbf{f}_i$.

The face of a hypercube is a hypercube itself with a smaller dimension. The faces of dimension 0, 1, $(d-1)$ are called vertices, edges and facets [SW01].

Example 5.2.2. Let $d = 3$ and $C = \{0, 1\}^3$.

Figure 5.2.2 shows examples of a 0-face (face $(1, 1, 1)$ in green), a 1-face (face $(1, *, 1)$ in blue) and a 2-face (face $(*, *, 0)$ in yellow). The whole cube itself is a 3-face. Note that since C is not defined on an interval but a set, the points between the vertices are actually not part of the face. Only the vertices themselves are a face. So the 2-face is not the whole colored area but $(*, *, 0) = \{(000), (010), (110), (100)\}$.



Definition 5.2.3 (Orientation function ϕ). [Fea+18, p. 25] Let $\phi : C \rightarrow \{0, 1\}^d$ be a direction function that assigns each edge of C a direction. The i 'th component of $\phi(v)$ represents the direction of the edge in dimension i .

- $\phi(v)_i = 0 \Leftrightarrow \textcircled{v} \leftarrow$ the edge points towards v .
- $\phi(v)_i = 1 \Leftrightarrow \textcircled{v} \rightarrow$ the edge points away from v .

Two nodes v and u that are adjacent to each other in dimension i are said to agree on their orientation in dimension i if $\phi(v)_i \neq \phi(u)_i$. This definition does not necessarily imply that all nodes agree to their orientation. Later on, this property will be ensured by a violation that forbids such constellations.

The cube C and the orientation function ϕ can be viewed as a directed graph if all nodes agree on their directions. Note that it is not necessarily an acyclic graph. A vertex $v \in C$ is called a *sink* if v only has incoming edges, i.e. $\forall i = 1, \dots, d: \phi(v)_i = 0$.

An orientation ϕ is called *unique sink orientation of C* if all subgraphs induced by nonempty faces of C have a unique sink. Since the whole cube is a face as well, this implies that the cube has a unique, global sink [SW01, p. 1], [GS06, Definition 2.1].

The problem UNIQUE SINK ORIENTATION OF CUBES (CUBE-USO) is to find the unique sink of that graph formed by C and ϕ under the promise that the given orientation fulfills the properties of a unique sink orientation.

Since a d -hypercube has 2^d many vertices and $2^{d-1}d$ edges, it is hard to check whether a given direction function fulfills the promise and really is a unique sink orientation.

Example 5.2.4. Let $d = 3$ and $C = \{0, 1\}^3$. The orientation function ϕ is represented by the arrows.

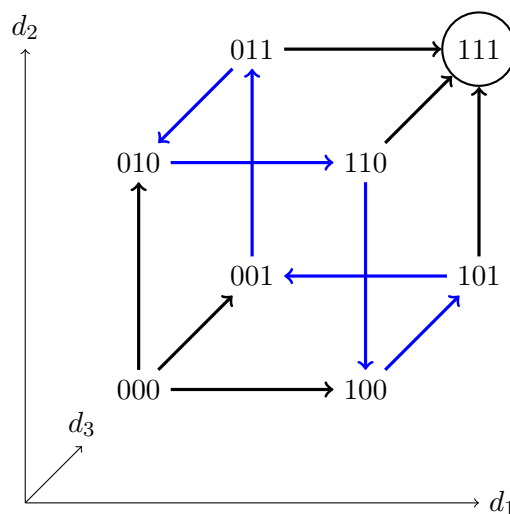


Figure 5.6.: Example of a $d = 3$ cube with a unique sink orientation

In this example, the edges form a unique sink orientation. The sink is (111) . $\phi(111) = (000)$, since the node has only incoming edges. Note that even when every face of the cube has a unique sink, it is not necessarily an acyclic graph. The cycle in the example is colored in blue.

This promise problem can be transformed into a total search problem by adding violations. The property that is used as violation originally comes from [SW01, Lemma 2.3] but I find it better explained in [Sch04]. To give a better understanding of why the condition that is used in the end actually works, we will now take a closer look at the relevant properties of ϕ .

Lemma 5.2.5. ([SW01, Lemma 2.1], [Sch04, Lemma 4.1]) *If ϕ is a unique sink orientation it follows that ϕ with all its edges in one dimension flipped is still a unique sink orientation.*

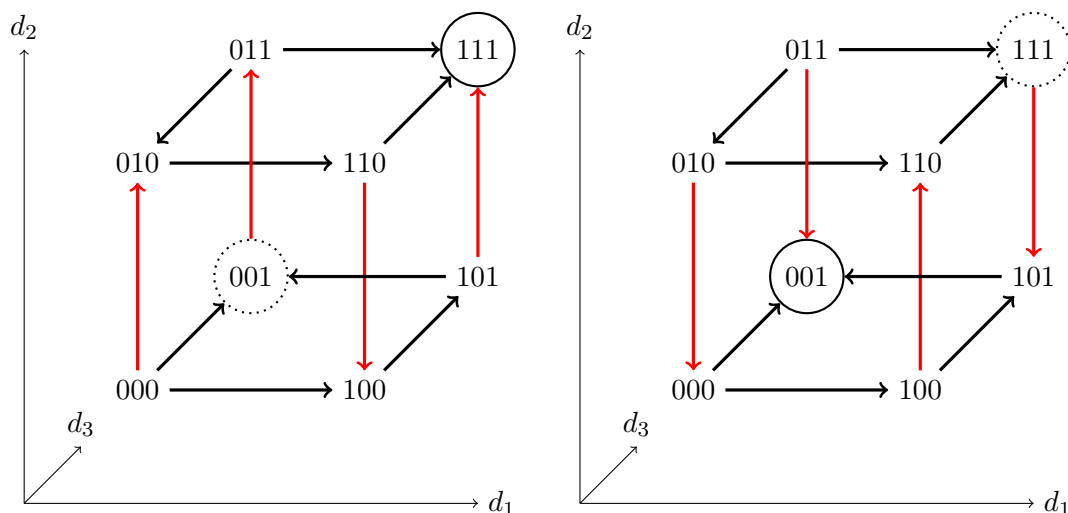
Proof. Let $C \in \{0, 1\}^d$. Let dimension i be arbitrary but fix. The edges of dimension i connect two subcubes of C with one another. These edges are colored in red in Example 5.2.6 on the following page. Since ϕ is a unique sink orientation, each of both subcubes has a unique sink, s_1 and s_2 . One of these two sinks is the global unique sink, without loss of generality let that be s_1 . It follows that $\phi(s_1)_i = 0$ because it is the

unique sink, and $\phi(s_2)_i = 1$ because it is *not* the unique sink. If all edges in dimension i are flipped then s_2 is the new global unique sink.

For the faces that do contain two or more red edges, the argument above can be applied recursively. Therefore each subcube still is a unique sink orientation after flipping the edges as well.

The faces that do not contain a red edge were valid unique sink orientations beforehand. Nothing has changed for them so they still are a valid unique sink orientation.

Example 5.2.6. $d = 3$ and $i = 2$.



□

Lemma 5.2.7. ([SW01, Lemma 2.2], [Sch04, Corollary 4.2]) ϕ is a unique sink orientation if and only if ϕ is a bijection in every subcube.

Proof. "⇒" Let ϕ be a unique sink orientation. Assume ϕ is not a bijection in every subcube. This means there exist two nodes v and u with $\phi(v) = \phi(u)$. By Lemma 5.2.5 on the previous page all dimensions i for which $\phi(v)_i = 1$ can be flipped and ϕ' is still a unique sink orientation. So after flipping all of these dimensions, it holds that $\phi'(v) = \phi'(u) = 0^d$, which means that v and u both are a sink. Therefore ϕ could not have been a unique sink orientation in the first place.

"⇐" If ϕ is a bijection in every subcube then for every subcube $C_\alpha = \{0, 1, *\}^d$ with $\alpha \subseteq \{1, \dots, d\}$ it holds that there is exactly one node v for which holds $\forall i \in \alpha: \phi(v)_i = 0$. Since this includes $\alpha = \{1, \dots, d\}$, ϕ is a unique sink orientation.

□

Lemma 5.2.8. ([SW01, Lemma 2.3], [Sch04, Proposition 4.3]) ϕ is a unique sink orientation if and only if for all $v, u \in C$ with $v \neq u$ it holds that

$$(v \oplus u) \wedge (\phi(v) \oplus \phi(u)) \neq 0^d, \quad (5.1)$$

where \oplus is the bit wise xor operation and \wedge is the bit wise and operation.

$(v \oplus u)$ is a vector that has a 1 at each index of the smallest subcube that contains both v and u . For example, for $C \in \{0, 1\}^3$, the smallest subcube containing (000) and (111) is the whole cube. The smallest subcube containing (000) and (110) is the face $(*, *, 0)$. In this case $((000) \oplus (110)) = (110)$. The vector has a 1 at every index where the face has a $*$.

In addition with $(\phi(v) \oplus \phi(u))$, this ensures that the orientation function is a bijection.

Proof. "⇒" Assume condition (5.1) does not hold. Then there exists some subcube C_α with $\alpha \subseteq \{1, \dots, d\}$ which contains two different nodes v and u that have the same orientation in all dimensions of that subcube $\forall i \in \alpha: \phi_i(v) = \phi_i(u)$, since otherwise the result of the condition would not be 0 in each component.

This is a contradiction to Lemma 5.2.7 on the preceding page which means that ϕ is not a unique sink orientation.

"⇐" Assume ϕ is not a unique sink orientation and therefore, by Lemma 5.2.7 on the previous page, not a bijection in every subcube.

Let C_α be the subcube with the smallest dimension in which ϕ is not a bijection and let v and u be the nodes that violate the bijectivity, i.e. $\forall i \in \alpha: \phi_i(v) = \phi_i(u)$. Therefore, for all $i \in \alpha$ the right side of the condition is 0.

For all $i \notin \alpha$ the left side of the condition is 0 since v and u are in the same subcube.

In conclusion, if ϕ is not a unique sink orientation, then condition (5.1) does not hold. This condition also ensures that all two nodes agree on their orientation. \square

Definition 5.2.9 (\perp -UNIQUE SINK ORIENTATION OF CUBES (CUBE-USO- \perp)). [Fea+20b, Definition 21] Given an orientation function $\phi : \{0, 1\}^d \rightarrow \{0, 1\}^d \cup \{\perp\}$ find one of the following:

($U\perp 1$) A point $v \in \{0, 1\}^d$ such that $\forall i = 1, \dots, d: \phi(v)_i = 0$.
 v is a sink.

($U\perp V1$) Two points $v, u \in \{0, 1\}^d$ such that $v \neq u$ and $(v \oplus u) \wedge (\phi(v) \oplus \phi(u)) = 0^d$.
 ϕ does not fulfill the unique sink orientation property. Note that $\phi(v) \neq \perp \neq \phi(u)$, since then it is a violation of type ($U\perp V2$).

($U\perp V2$) A point $v \in \{0, 1\}^d$ such that $\phi(v) = \perp$.
 ϕ is no valid orientation function at all.

($U\perp 1$) describes a unique sink. ($U\perp V1$) certifies, as explained before, that ϕ is not a unique sink orientation.

($U\perp V2$) might seem ridiculous at first, and it is indeed not necessary to define a valid total version of CUBE-USO. But later on, the problem P-LCP will be reduced to CUBE-USO and the reduction is more convenient if violation ($U\perp V2$) is included here. The violation does not alter the problem in any significant way. One can formulate an equivalent version of CUBE-USO without ($U\perp V2$).

Definition 5.2.10 (UNIQUE SINK ORIENTATION OF CUBES (CUBE-USO)). Given an orientation function $\phi : \{0, 1\}^d \rightarrow \{0, 1\}^d$ find one of the following:

($U1$) A point $v \in \{0, 1\}^d$ such that $\forall i = 1, \dots, d: \phi(v)_i = 0$.
 v is a sink.

($UV1$) Two points $v, u \in \{0, 1\}^d$ such that $v \neq u$ and $(v \oplus u) \wedge (\phi(v) \oplus \phi(u)) = 0^d$.
 ϕ does not fulfill the unique sink orientation property.

Lemma 5.2.11. *There exists a promise preserving polynomial time reduction from CUBE-USO- \perp to CUBE-USO [Fea+20b, Lemma 22].*

Proof. Given an instance I of CUBE-USO- \perp , create an instance I' of CUBE-USO. Let $C' := C$. If $\phi(v) \neq \perp$, let $\phi'(v) = \phi(v)$ and $\phi'(v) = 0^d$ otherwise.

Given a solution v of type $(U1)$ for ϕ' , then either $\phi(v) = 0^d$ and therefore v is a solution of type $(U\perp 1)$, or $\phi(v) = \perp$ and v is a violation of type $(U\perp V2)$. Given a solution v, u of type $(UV1)$ for ϕ' , either $\phi(v) = \perp$ or $\phi(u) = \perp$ which results in a violation of type $(U\perp V2)$, or otherwise the nodes are a solution of type $(U\perp V1)$.

The reduction is promise preserving, since every valid solution of type $(U\perp 1)$ is also a valid solution of type $(U1)$ in the transformed instance. Every violation in the CUBE-USO instance are mapped back to a violation in the CUBE-USO- \perp instance.

□

The other direction is trivial: An instance of CUBE-USO- \perp created from an CUBE-USO instance can never have a violation of type $(U\perp V2)$ by definition. Therefore, CUBE-USO and CUBE-USO- \perp are polynomial time equivalent.

There is no polynomial time algorithm known for CUBE-USO. [SW01] proved that there is a deterministic algorithm with runtime $\mathcal{O}(1, 61^d)$ and a randomized algorithm with runtime $\mathcal{O}(1, 47^d)$.

5.2.1. Cube-USO to OPDC

The problem CUBE-USO has naturally no cost or potential function. Neither do the points of the hypercube have an intuitive neighborhood or successor as needed for a UNIQUE END OF POTENTIAL LINE instance. Nonetheless, there exists a reduction from CUBE-USO to OPDC which proves that CUBE-USO is in UniqueEOPL. In [Fea+20b] the reduction is done from CUBE-USO- \perp to OPDC, the reduction from CUBE-USO to OPDC is basically the same but without the special cases for $\phi(v) = \perp$. For a definition and examples of OPDC, see Section 4.1 on page 23.

Theorem 5.2.12. *There exists a promise preserving polynomial time reduction from CUBE-USO to OPDC.*

Proof. Given an instance $I = (C, \phi)$ of CUBE-USO, create an instance $I' = (\Gamma, \mathcal{D})$ of OPDC. Let $\Gamma := C$. The direction function D is defined as follows:

$$D_i(v) = \begin{cases} \text{Zero} & \text{if } \phi(v)_i = 0 \\ \text{Up} & \text{if } \phi(v)_i = 1 \text{ and } v_i = 0 \\ \text{Down} & \text{if } \phi(v)_i = 1 \text{ and } v_i = 1. \end{cases}$$

Correctness In order to prove the correctness of this construction, it must be shown that each solution of the constructed OPDC instance I' matches back to the correct solution of the CUBE-USO instance I .

A solution of type (O1) Let point $p \in \Gamma$ be a solution of type (O1). It holds that $\forall i = 1, \dots, d: D_i(p) = \text{Zero}$. This implies that $\phi_i(p) = 0$ for all i which means that p is a sink and a solution of type (U1).

A solution of type (OV1) For an i – Slice \mathbf{s} a solution of type (OV1) consists of two fixpoints $p, q \in \Gamma_{\mathbf{s}}$ with $p \neq q$ such that $\forall j \leq i: D_j(p) = D_j(q) = \text{Zero}$. This implies that p and q are both sinks on the same i -face, which is a violation of type (UV1).

A solution of type (OV2) If there is a violation of type (OV2), then there is an i – Slice \mathbf{s} and two points $p, q \in \Gamma_{\mathbf{s}}$ such that

- $\forall j < i: D_j(p) = D_j(q) = \text{Zero}$
- p_i and q_i are adjacent to each other in dimension $i: p_i = q_i + 1$
- $D_i(p) = \text{Down}$ and $D_i(q) = \text{Up}$.

For all $j < i$, $\phi_j(p) = \phi_j(q) = 0$. For all $j = i$, $\phi_j(p) = \phi_j(q) = 1$. Therefore p and q have the same orientation in the face described by \mathbf{s} . They form a violation of type (UV1).

A solution of type (OV3) A violation of type (OV3) is never produced by the construction. By definition of D , for a point $p_i = 0$ the direction is never Down and for a point $p_i = 1$ the direction is never Up.

Conclusion The reduction is promise preserving: Solutions of type (U1) of the CUBE-USO instance are only mapped to solutions of type (O1) of the OPDC instance. If the CUBE-USO instance has no violations, then the OPDC instance has no violations. \square

5.3. P-Matrix Linear Complementarity Problem (P-LCP)

Definition 5.3.1 (LINEAR COMPLEMENTARITY PROBLEM (LCP)). [DP11, p. 794] Given a matrix $M \in \mathbb{R}^{d \times d}$ and a vector $q \in \mathbb{R}^{d \times 1}$, find two vectors $z, w \in \mathbb{R}^{d \times 1}$ such that

- (1) $w = Mz + q$,
- (2) $z, w \geq 0$ component wise and
- (3a) $z^T w = 0$ or
- (3b) $\forall i \in \{1, \dots, d\}: z_i \cdot w_i = 0$.

Conditions (3a) and (3b) are equivalent if condition (2) is satisfied. For each i either z_i or w_i must be 0, since none of the entries are allowed to be negative.

Condition (1) of the LCP can be stated alternatively: $Iw - Mz = q$.

If $q \geq 0$ in each component, then there exists the trivial solution $z := 0^d$ and $w := q$.

Deciding if a given LCP has a solution is NP-complete [Chu89]. For an approach on how to solve an LCP, see Appendix A.1 on page 134.

Definition 5.3.2 (P-Matrix). A matrix $M \in \mathbb{R}^{n \times n}$ is called P-Matrix if all its principal minors are positive [CPS92, Definition 3.3.1].

For each $n \times n$ matrix M and for each $\alpha, \beta \subseteq \{1, \dots, n\}$, the submatrix $M_{\alpha, \beta}$ of M is the matrix whose entries lie in the rows of M indexed by α and the columns indexed by β . If $\alpha = \beta$, the matrix $M_{\alpha, \alpha}$ is called *principal submatrix* of M . The determinant of $M_{\alpha, \alpha}$ is called *principal minor* [CPS92, Definition 2.2.1]. For $1 \leq k \leq d$ and $\alpha = \{1, \dots, k\}$, $M_{\alpha, \alpha}$ is called *leading principal submatrix* and its determinant is called *leading principal minor* [CPS92, Definition 2.2.2]:

$$\|(m_{11})\|, \left\| \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \right\|, \left\| \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \right\|, \dots, \left\| \begin{pmatrix} m_{11} & \dots & m_{1d} \\ \dots & & \dots \\ m_{d1} & \dots & m_{dd} \end{pmatrix} \right\|$$

Example 5.3.3. $d = 3$

$$M := \begin{pmatrix} 1 & -1 & 0 \\ 1 & 3 & 1 \\ 0 & -10 & 2 \end{pmatrix}$$

The principal minors of M are:

$$\|(1)\| = 1, \|(3)\| = 3, \|(2)\| = 2$$

$$\left\| \begin{pmatrix} 1 & -1 \\ 1 & 3 \end{pmatrix} \right\| = 4, \left\| \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \right\| = 2, \left\| \begin{pmatrix} 3 & 1 \\ -10 & 2 \end{pmatrix} \right\| = 16$$

$$\left\| \begin{pmatrix} 1 & -1 & 0 \\ 1 & 3 & 1 \\ 0 & -10 & 2 \end{pmatrix} \right\| = 18$$

Since they are all positive, M is a P-Matrix.

A P-Matrix is not necessarily positive (semi) definite. The matrix in the above example is not positive definite.

For example, there exists a vector x such that $x^T M x < 0$:

$$(0, 1, 1) \cdot \begin{pmatrix} 1 & -1 & 0 \\ 1 & 3 & 1 \\ 0 & -10 & 2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = -4.$$

Every positive definite matrix is a P-Matrix [CPS92, p. 147]. A symmetric real matrix $M \in \mathbb{R}^{d \times d}$ is positive definite if and only if all its principal minors are greater than 0 [KS, p. 349, Sylvester's Criterion]. If $M \in \mathbb{R}^{d \times d}$ is positive definite, then the LCP (q, M) has a unique solution for all $q \in \mathbb{R}^d$ [CPS92, Theorem 3.1.6.]. If M is positive definite, LCP can be solved in polynomial time [MY88].

Checking if a matrix is a P-Matrix is co-NP-complete [Cox94]. A matrix M is a P-Matrix if and only if the LCP (q, M) has a unique solution for every $q \in \mathbb{R}^d$ [CPS92, Theorem 3.3.7]. If $q \geq 0$ component wise, the unique solution is the trivial solution.

Example 5.3.4. (Trivial solution)

Let be $d = 2$ and the P-LCP problem (q, M) be

$$M := \begin{pmatrix} 1 & -1 \\ 1 & 3 \end{pmatrix}, q := \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

M is a P-Matrix. The only and unique solution to the given P-LCP is the trivial solution:

$$\begin{aligned} w_1 &= z_1 - z_2 + 1 \\ w_2 &= z_1 + 3z_2 + 2 \end{aligned} \quad z = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, w = \begin{pmatrix} 1 \\ 2 \end{pmatrix} = q.$$

Example 5.3.5. (M is not a P-Matrix)

Let be $d = 2$ and the LCP problem (q, M) be

$$M := \begin{pmatrix} -1 & 1 \\ 1 & 3 \end{pmatrix}, q := \begin{pmatrix} 1 \\ 2 \end{pmatrix}.$$

M is *not* a P-Matrix. There are multiple solutions for the given LCP:

$$\begin{aligned} w_1 &= -z_1 + z_2 + 1 \\ w_2 &= z_1 + 3z_2 + 2 \end{aligned} \quad \begin{aligned} z &= \begin{pmatrix} 0 \\ 0 \end{pmatrix}, w = \begin{pmatrix} 1 \\ 2 \end{pmatrix} = q \text{ and} \\ z &= \begin{pmatrix} 1 \\ 0 \end{pmatrix}, w = \begin{pmatrix} 0 \\ 3 \end{pmatrix}. \end{aligned}$$

Example 5.3.6. (Non-trivial solution)

Let be $d = 2$ and the P-LCP problem (q, M) be

$$M := \begin{pmatrix} 1 & -1 \\ 1 & 3 \end{pmatrix}, q := \begin{pmatrix} 1 \\ -2 \end{pmatrix}.$$

M is a P-Matrix. Not all components of q are positive, therefore the trivial solution is not a valid solution. The following is the unique solution:

$$\begin{aligned} w_1 &= z_1 - z_2 + 1 \\ w_2 &= z_1 + 3z_2 - 2 \end{aligned} \quad z = \begin{pmatrix} 0 \\ 2/3 \end{pmatrix}, w = \begin{pmatrix} 1/3 \\ 0 \end{pmatrix}.$$

For the search problem definition of P-LCP that will be shown to be in UniqueEOPL, a different representation of a solution (z, w) is used: For a given candidate solution (z, w) we can construct the set $\alpha := \{i \mid i \in \{1, \dots, d\}, z_i > 0\}$ containing all indices of z that are not 0.

The other way around, for any given $\alpha \subseteq \{1, \dots, d\}$ it is possible to check whether this is a set that was constructed from a valid solution. How to check that is shown in the following.

An LCP can be viewed in terms of complementary cones [CPS92, Section 1.3]. A nonempty set X is a cone in \mathbb{R}^d if for any $x \in X$ and any $t \geq 0$ it holds that $xt \in X$. For a matrix $A \in \mathbb{R}^{d \times n}$, let $\mathcal{C}(A) := \{q \in \mathbb{R}^d : q = Ay \text{ for some } y \in \mathbb{R}_+^n\}$ be a convex cone obtained by nonnegative linear combinations of the columns of A .

Let $M \in \mathbb{R}^{d \times d}$ be the matrix of an LCP. Consider the matrix $(I, -M) \in \mathbb{R}^{d \times 2d}$. When solving the linear complementarity problem (q, M) , we are searching for $(w, z) \in \mathbb{R}^{2d}$ such that

- (1) $Iw - Mz = q$.
- (2) $z, w \geq 0$ component wise and
- (3) $\forall i \in \{1, \dots, d\} : z_i \cdot w_i = 0$.

We want to know if q is an element of $\mathcal{C}(I, -M)$ so that for every i not both $I_{\cdot,i}$ and $M_{\cdot,i}$ are used because of condition (3) of LCP. To formalize this additional condition, we define the following matrix: For any $\alpha \subseteq \{1, \dots, d\}$ let $(A_\alpha) \in \mathbb{R}^{d \times d}$ be the matrix whose columns are for all i in α equal to the negative i 'th column of matrix M and the i 'th unit column vector e_i otherwise:

$$(A_\alpha)_{\cdot,i} := \begin{cases} -M_{\cdot,i} & \text{if } i \in \alpha \\ e_i & \text{if } i \notin \alpha \end{cases}$$

(A_α) is called complementary basis of M , $\mathcal{C}(A_\alpha)$ is called complementary cone. (A_α) can be constructed in polynomial time. For a $d \times d$ matrix, there are 2^d many different sets α and therefore complementary cones.

Given a vector q , to decide whether the LCP (q, M) has a solution, it suffices to check whether q belongs to one of the complementary cones, i.e. if there exists an α such that $q \in \mathcal{C}(A_\alpha)$. If q is in $\mathcal{C}(A_\alpha)$, it holds that $y = w + z$, where all α components of y belong to z and all the others belong to w because of condition (3) of the LCP. All α components of w and all non α components of z are 0.

This is equivalent to testing if there exists a solution to the system $(A_\alpha)y = q$ with $y \geq 0$ for some $\alpha \subseteq \{1, \dots, d\}$. If (A_α) is nonsingular (which it always is if M is a P-Matrix), then this is equivalent to testing whether $(A_\alpha)^{-1}q \geq 0$ component wise.

In conclusion, we can check in polynomial time for any given α if q is in the cone $\mathcal{C}(A_\alpha)$, i.e. if (A_α) is nonsingular and $(A_\alpha)^{-1} \cdot q$ is non negative in each component. Then we

know that α corresponds to a valid solution. Since $(A_\alpha)y = q$ is a linear program, its solution and thereby the solution represented by α can be calculated in polynomial time.

Example 5.3.7. Recall the LCP from Example 5.3.6 on page 75. Let $\alpha = \{1\}$. Note that this is not the α of the unique solution. (A_α) is nonsingular, but the components of $(A_\alpha)^{-1} \cdot q$ are negative.

$$(A_\alpha) = (A_\alpha)^{-1} = \begin{pmatrix} -1 & 0 \\ -1 & 1 \end{pmatrix}$$

$$(A_\alpha)^{-1} \cdot q = \begin{pmatrix} -1 \\ -3 \end{pmatrix}$$

If $\alpha = \{2\}$, which is the correct α for the unique solution, the (A_α) is nonsingular and the components of $(A_\alpha)^{-1} \cdot q$ are all positive:

$$(A_\alpha) = \begin{pmatrix} 0 & 1 \\ 1 & -3 \end{pmatrix} \qquad (A_\alpha)^{-1} = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix}.$$

$$(A_\alpha)^{-1} \cdot q = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

We define the function $\text{out}: \mathcal{P}(\{1, \dots, d\}) \rightarrow \{0, 1\}^d \cup \{\perp\}$ as $\text{out}(\alpha) = \perp$ if $\det(A_\alpha) = 0$ and otherwise

$$(\text{out}(\alpha))_i := \begin{cases} 1 & \text{if } ((A_\alpha)^{-1} \cdot q)_i < 0 \\ 0 & \text{if } ((A_\alpha)^{-1} \cdot q)_i \geq 0. \end{cases}$$

α corresponds to a valid solution if and only if $\text{out}(\alpha) = 0^d$.

We define the function $\text{char}: \mathcal{P}(\{1, \dots, d\}) \rightarrow \{0, 1\}^d$ as characteristic vector of α :

$$(\text{char}(\alpha))_i := \begin{cases} 1 & \text{if } i \in \alpha \\ 0 & \text{if } i \notin \alpha. \end{cases}$$

P-LCP is a special case of LCP with the promise that M is a P-Matrix. In order to define a total search problem, a violation solution is needed for the case that M is not a P-Matrix.

Definition 5.3.8 (P-MATRIX LINEAR COMPLEMENTARITY PROBLEM (P-LCP) [Fea+20b]). Given a matrix $M \in \mathbb{R}^{d \times d}$ and a vector $q \in \mathbb{R}^{d \times 1}$, find one of the following:

- (P1) Two vectors $z, w \in \mathbb{R}^{d \times 1}$ that are a valid solution for the LCP:
 - (1) $w = Mz + q$,
 - (2) $z, w \geq 0$ component wise and
 - (3a) $z^T w = 0$ or
 - (3b) $\forall i \in \{1, \dots, d\}: z_i \cdot w_i = 0$.
- (PV1) A set $\alpha \subseteq \{1, \dots, d\}$ such that the corresponding principal minor of matrix M is not positive: $\det(M_{\alpha, \alpha}) \leq 0$.
- (PV2) Two distinct sets $\alpha, \beta \subseteq \{1, \dots, d\}$ with $(\text{char}(\alpha) \oplus \text{char}(\beta)) \wedge (\text{out}(\alpha) \oplus \text{out}(\beta)) = 0^d$, where \oplus is the bit wise *xor* and \wedge is the bit wise *and* function of two vectors.
(Note that in this case *out* does not return \perp , since this already implies a violation of type (PV1).)

One of the two violations would be enough to make the problem total. Both violations are needed to make the reduction easier that proves that P-LCP is in UniqueEOPL.

(PV1) is a certificate for M not being a P-Matrix because there exists a principal minor that is not greater than 0.

(PV2) corresponds to the properties of the complementary cone that are fulfilled by definition if M is a P-Matrix. For more detailed proofs see [SW78, Proof of Property 2] and [SW78, Property 5]. Therefore, if this condition is fulfilled, it is a sufficient proof of M not being a P-Matrix.

There is no polynomial time algorithm known for solving P-LCP. [Fea+18, Corollary 42] states that Aldous Algorithm that was introduced in [Ald83] can be applied to P-LCP too, thus providing an algorithm that runs in $\mathcal{O}((\sqrt{2})^d)$ time. Since P-LCP can be reduced to CUBE-USO, the algorithms of CUBE-USO can also be applied to solve this problem.

5.3.1. P-LCP to Cube-USO

[AV11] proved P-LCP to be in PPAD and [DP11] showed that P-LCP is in CLS. It is not proven complete for either of these classes.

Theorem 5.3.9. *There exists a promise preserving polynomial time reduction from P-LCP to CUBE-USO- \perp , which proves that P-LCP is in UniqueEOPL.*

Proof. Given a P-LCP instance $I = (q, M)$ with $q \in \mathbb{R}^d$ and $M \in \mathbb{R}^{d \times d}$, construct a CUBE-USO- \perp instance $I' = (C, \phi)$ with $C = \{0, 1\}^d$ and $\phi : \{0, 1\}^d \rightarrow \{0, 1\}^d$.

Construction For each subset $\alpha \subseteq \{1, \dots, d\}$, $v := \text{char}(\alpha)$ is a node of C . Every node of C can be distinctly mapped back to α . The orientation function ϕ behaves exactly like the *out* function: $\phi(v) := \text{out}(\text{char}^{-1}(v))$.

Handling degeneracy There are different types of degeneracy of an LCP:

- A degenerate matrix M : M is degenerate if there exists a principal minor of M which is 0. In case of a P-LCP instance, this case is already covered as violation (PV1).
- A degenerate vector q (sometimes also called a degenerate solution (w, z)): An LCP instance has a degenerate q if q is a linear combination of $d - 1$ or less columns of $(I, -M)$ [SW78], i.e. $(A_\alpha)^{-1}q$ has a zero entry for some $\alpha \subseteq \{1, \dots, d\}$ [Fea+20b]. This implies that if q is degenerate, in the valid solution (z, w) there exists an index i such that $z_i = w_i = 0$.

Nondegeneracy is important for the bard type algorithm to terminate and not to cycle [CPS92, Section 4.2], but it also needs to be eliminated to make the reduction work.

Example 5.3.10. (Degenerate q)

$$M := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, q := \begin{pmatrix} 0 \\ -1 \end{pmatrix}.$$

M is a P-Matrix, q is degenerate. It can be represented as a linear combination of the second column $M_{\cdot 2}$. Nonetheless, the given LCP has a unique solution:

$$w = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, z = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

For $\alpha_1 = \{2\}$ it holds that $(A_{\alpha_1}) = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ and $(A_{\alpha_1})^{-1}q = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. $\text{out}(\alpha_1) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and therefore α_1 is a sink in the constructed CUBE-USO- \perp instance.

For $\alpha_2 = \{1, 2\}$ it holds that $(A_{\alpha_2}) = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$ and $(A_{\alpha_2})^{-1}q = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. $\text{out}(\alpha_2) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ as well and therefore α_2 is a second sink in the constructed CUBE-USO- \perp instance.

There are multiple sets α for which the condition $((A_\alpha)^{-1}q) \geq 0$ holds. For $\alpha_1 = \{2\}$ and for $\alpha_2 = \{1, 2\}$ this yields in an equivalent output of out , which results in an invalid CUBE-USO- \perp instance. This is a problem because we created an CUBE-USO- \perp instance with a violation from a valid LCP instance, which contradicts the idea of a promise preserving reduction.

To avoid problems caused by degenerate LCP's, a strategy called *perturbation*, which is usually used in linear programming, is applied to (q, M) before doing the actual reduction in order to dissolve its degeneracy.

A linear program is degenerate if and only if for the given constraints $Ax = b, x \geq 0$ it holds that b can be expressed as a linear combination of $d - 1$ or less column vectors of A , i.e. that for a basis B there exists a component in $B^{-1}b$ that is 0 [Mur83, p. 323].

The idea of perturbation is to slightly modify b of the given linear program such that b' cannot be represented by a linear combination of $d - 1$ columns anymore. By shifting b slightly, the intersection of the constraints shifts apart.

We can use this technique here because we can represent the part of the LCP that causes the degeneracy as a linear program:

$$(A_\alpha)^{-1}q \geq 0 \Leftrightarrow (A_\alpha)y = q \text{ with } y \geq 0 \text{ component wise.} \quad (5.2)$$

Let ϵ be an arbitrarily small positive number and $q(\epsilon) := q + (\epsilon, \epsilon^2, \dots, \epsilon^d)^T$.

Lemma 5.3.11. [Mur83, Theorem 10.1]

Given any $q \in \mathbb{R}^d$, there exists a positive number $\tilde{\epsilon} > 0$ such that whenever $0 < \epsilon < \tilde{\epsilon}$, the following perturbed problem is nondegenerate:

$$\begin{aligned} (A_\alpha)y &= q(\epsilon) \\ y &\geq 0 \text{ component wise.} \end{aligned} \tag{5.3}$$

For a proof see [Mur83, p. 324] or Appendix A.3 on page 137.

Lemma 5.3.12. [Mur83, Theorem 10.5]

If B is a feasible basis for the perturbed LCP 5.3 with ϵ being arbitrarily small, then B is a feasible basis for the original un-perturbed LCP 5.2.

For a proof see either [Mur83, p. 326] or Appendix A.3 on page 137.

So if $(A_\alpha)^{-1}$ is a feasible solution for 5.3, then it is a solution for 5.2 as well. Therefore if α and (A_α) represent a valid solution to $(q(\epsilon), M)$, they also represent a valid solution to (q, M) .

Example 5.3.13. (Perturbation) Recall Example 5.3.10 on the previous page. The following LCP is its perturbed version:

$$M := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, q(\epsilon) := \begin{pmatrix} 0 + \epsilon \\ -1 + \epsilon^2 \end{pmatrix}.$$

M is a P-Matrix, the given LCP has a unique solution:

$$\begin{aligned} w_1 &= z_1 + \epsilon \\ w_2 &= z_2 - 1 + \epsilon^2 \end{aligned} \quad w = \begin{pmatrix} \epsilon \\ 0 \end{pmatrix}, z = \begin{pmatrix} 0 \\ 1 - \epsilon^2 \end{pmatrix}.$$

Note that the solution of the perturbed LCP is different, but not the corresponding α and (A_α) . There is only one set α that represents that solution: $\alpha = \{2\}$. $(A_\alpha)^{-1} = (A_\alpha)$ is a feasible basis for $(A_\alpha)y = q(\epsilon)$ and $(A_\alpha)y = q$.

The construction for the reduction explained above is not applied to the original P-LCP but on the perturbed one.

Correctness

- A solution of $(U \perp 1)$ of I' maps back to a solution $(P1)$ of I .
- A solution of $(U \perp V2)$ of I' maps back to a solution $(PV1)$ of I .
- A solution of $(U \perp V1)$ of I' maps back to a solution $(PV2)$ of I .

The reduction is promise preserving.

□

5.4. P-General-LCP (P-GLCP)

GENERAL-LCP (GLCP) was first introduced by [CD70] (in [CPS92, p. 32] it is also called *vertical LCP*). It is like a usual LCP with the exception that the matrix M is not necessarily a square matrix.

Definition 5.4.1 (Vertical Block Matrix). [Rüs07, Definition 2.1]

Let $(\omega_1, \dots, \omega_d)$ be d many block-sizes with $\omega_i \in \mathbb{N}^+$. Let n be the sum of all block sizes: $n := \sum_{i=1}^d \omega_i$.

The Matrix $N \in \mathbb{R}^{n \times d}$ is called *vertical block matrix* if it has the form

$$N = \begin{pmatrix} N^1 \\ \vdots \\ N^d \end{pmatrix}$$

with $N^i \in \mathbb{R}^{\omega_i \times d}$. N^i is called the i 'th block of the matrix.

Analogously a *vertical block vector* can be defined.

Definition 5.4.2 (GENERAL-LCP (GLCP)). [Rüs07, p. 22] Given a vertical block matrix $N \in \mathbb{R}^{n \times d}$ and a vertical block vector $q \in \mathbb{R}^n$, both with block sizes $(\omega_1, \dots, \omega_d)$, find two vectors $z \in \mathbb{R}^d$ and $w \in \mathbb{R}^n$ such that

- (1) $w - Nz = q$,
- (2) $z, w \geq 0$ component wise and
- (3) $z_i \cdot \prod_{j=1}^{\omega_i} w_j^i = 0$ for all $i = 1, \dots, d$.

In comparison to the LCP defined in Definition 5.3.1 on page 74, the third condition differs here. The complementarity of z and w holds block wise, i.e. if for some index i it holds that $z_i \neq 0$ then at least one entry of block w^i is 0: $\exists j \in \{1, \dots, \omega_i\}: w_j^i = 0$. The other way around it must hold that if there doesn't exist a zero entry in a block w^i , then $z_i = 0$.

An d -square submatrix M of N is called a *representative submatrix* if its j 'th row is taken from the j 'th block N_j [HS95, Definition 2.2]. There are $\prod_{j=1}^d \omega_j$ many representative submatrices for a vertical block matrix. A vertical block matrix is a P-Matrix if and only if all principal minors of all representative submatrices are positive [HS95, Definition 2.6].

The GLCP $= (N, q)$ has a unique solution for each $q \in \mathbb{R}^d$ if and only if N is a P-Matrix [HS95, Theorem 4.3].

Definition 5.4.3 (P-GENERAL-LCP (P-GLCP)). Given a vertical block matrix $N \in \mathbb{R}^{n \times d}$ and a vertical block vector $q \in \mathbb{R}^n$, both with block sizes $(\omega_1, \dots, \omega_d)$, find one of the following:

(GP1) Two vectors $z \in \mathbb{R}^d$ and $w \in \mathbb{R}^n$ that are a valid solution for the LCP:

(1) $w - Nz = q$,

(2) $z, w \geq 0$ component wise and

(3) $z_i \cdot \prod_{j=1}^{\omega_i} w_j^i = 0$ for all $i = 1, \dots, d$.

(GPV1) A representative submatrix M of N and a set $\alpha \subseteq \{1, \dots, d\}$ such that the corresponding principal minor of M is not positive: $\det(M_{\alpha, \alpha}) \leq 0$.

Theorem 5.4.4. P-GLCP can be reduced in polynomial time to GRID-USO [Rüs07, Theorem 3.5].

The definitions of GRID-USO and P-GLCP in [Rüs07] do not work with violations. They state that the grid orientation induced by a valid P-GLCP is a unique sink orientation. There is no statement about what happens if N of the P-GLCP was not a valid P-Matrix. All in all, their reduction is not promise preserving. It is not necessary to prove containment in UniqueEOPL via a promise preserving reduction. Thus, P-GLCP is in UniqueEOPL.

That the reduction is not promise preserving implies that P-GLCP is not necessarily in PromiseUEOPL. For example, it can happen that the reduction creates a valid GRID-USO instance from a broken P-GLCP instance. Nevertheless, I think that the reduction easily could be altered to be promise preserving, similar as it has been done from P-LCP to CUBE-USO.

5.5. Simple Stochastic Game (SSG)

A simple stochastic game is a 2-player game played on a directed graph with some edges having a transition probability. Therefore, this game is not deterministic. Stochastic games were first introduced by [Sha53] but more extensively studied by [Con90] and [Con92].

Definition 5.5.1 (Simple Stochastic Game [Con92; GR05]). Let $G = (V, E)$ be a directed graph with a partition of the vertices such that $V = \{\beta_{max}, \beta_{min}\} \cup V_{max} \cup V_{min} \cup V_n$. Each vertex other than β_{max} and β_{min} is either a *min*-vertex $v \in V_{min}$, a *max*-vertex $v \in V_{max}$ or a *neutral*-vertex $v \in V_n$.

There is one start vertex $\alpha \in V$ and an end vertex for each of the two players β_{max} and β_{min} . Let $E^+(v)$ be the set of outgoing edges of a vertex v .

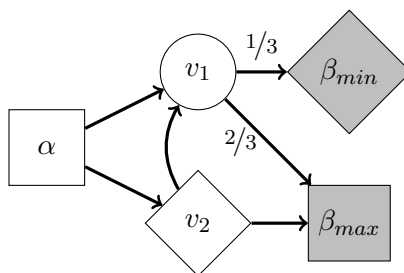
In the game, a token is placed on the start vertex. The players move the token from a vertex v to another by the following rules:

- If $v = \beta_{max}$, the game is over and Player *Max* wins.
- If $v = \beta_{min}$, the game is over and Player *Min* wins.
- If $v \in V_{max}$, player *Max* can choose to which neighbor of v the token is moved.
- If $v \in V_{min}$, player *Min* can choose to which neighbor of v the token is moved.
- If $v \in V_n$, an outgoing edge $e \in E^+(v)$ is chosen with probability $P_r[e] > 0$, where $\sum_{e \in E^+(v)} P_r[e] = 1$.

The game ends when the token reaches either β_{max} or β_{min} .

The players are called *Max* and *Min* because *Max* decides to move the token such that the probability of reaching β_{max} is maximized and *Min* tries to minimize the probability of reaching β_{max} [Con92].

Example 5.5.2. Let the following graph be an SSG. Nodes from $V_{max} = \{\alpha\}$ are drawn with a square around them, vertices from $V_{min} = \{v_2\}$ are drawn with a diamond and neutral nodes $V_n = \{v_1\}$ are round.



Player *Max* begins. When he chooses to go to node v_2 it is Player *Min*'s turn. In this case, Player *Min* would go to v_1 , since otherwise Player *Max* would win. From v_1 Player *Max* wins with probability $2/3$ and Player *Min* wins with probability $1/3$.

Definition 5.5.3 (BINARY-SSG [Con92]). An SSG where each node, except β_{max} and β_{min} has exactly two outgoing edges is called BINARY SIMPLE STOCHASTIC GAME (BINARY-SSG).

Theorem 5.5.4. Every non-binary SSG can be reduced in polynomial time to a BINARY-SSG [ZP96, p. 12].

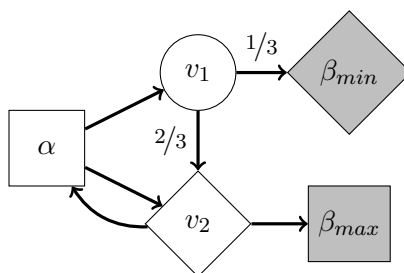
Proof sketch. As described in [ZP96, p. 12], the idea is to replace a node with $k > 2$ outgoing edges by a binary tree with k leaves. The probabilities can be adjusted such that the two outgoing edges of the leaf node have the same probability as their corresponding outgoing edges in the original node. This increases the number of nodes and edges of the resulting BINARY-SSG, but only by a constant. Vertices with only one outgoing edge can be removed from the game without affecting the result [GR05, p. 214].

□

A *strategy* is a rule that tells the player how to play. It is called *pure* (or *positional*) if the strategy does not depend on the history of moves that have been made so far and no probabilistic choices are made. A strategy of a player is optimal if it results in the maximal (or minimal) possible value for that player. In SSG's, the optimal strategy for both players is a pure strategy [Con92].

An SSG (or BINARY-SSG) is called *stopping* if, no matter what the players do, with probability 1 the token eventually reaches either β_{max} or β_{min} starting from *any* arbitrary node [GR05]. Recall Example 5.5.2 on the previous page. It is a stopping game.

Example 5.5.5. Let the following graph be an SSG. This game is non-stopping. It would stop if Player *Min* went to β_{max} which will never happen and analogously Player *Max* will never go to v_1 since then it might end in β_{min} . Therefore the game never ends.



Lemma 5.5.6. *Every non-stopping SSG (or BINARY-SSG) can be transformed in polynomial time to a stopping SSG (or BINARY-SSG respectively) [Con90].*

Proof sketch. This can be achieved by adding a transition from every non-sink node to β_{min} with a suitable small probability ϵ . With probability $(1 - \epsilon)$, the original path is followed. For each outgoing edge of a *max*-vertex and a *min*-vertex, a *neutral*-vertex is inserted with the behavior described above. Let SSG' be an SSG modified as described above. If ϵ is a small rational of size polynomial in the size of SSG, SSG' and SSG have the same optimal strategy [EY10, p. 2582].

□

For a stopping SSG a value $\varphi: V \rightarrow [0, 1]$ can be assigned to each vertex:

$$\varphi(v) = \begin{cases} 1 & \text{if } v = \beta_{max} \\ 0 & \text{if } v = \beta_{min} \\ \max_{(v,u) \in E^+(v)} \varphi(u) & \text{if } v \in V_{max} \\ \min_{(v,u) \in E^+(v)} \varphi(u) & \text{if } v \in V_{min} \\ \sum_{(v,u) \in E^+(v)} P_r[(v,u)] \cdot \varphi(u) & \text{if } v \in V_n. \end{cases}$$

The value corresponds to the probability of reaching β_{max} from the current node when both players play optimal [GR05, Lemma 1]. This optimal strategy can be obtained by following the sequence of nodes defined by φ . Calculating this function may take exponential time.

The decision versions of SSG and BINARY-SSG ask whether the probability that player *Max* wins is higher than some probability p (usually $p = 1/2$) if both players play optimal. This problem is in $\text{NP} \cap \text{co-NP}$ [Con92].

There is no polynomial time algorithm known to solve that problem. There are different strategies for solving an SSG (or BINARY-SSG), for example value iteration, strategy iteration (or policy iteration), and quadratic programming. For an overview of these strategies see for example [Kř+20] or [Con90]. [Lud95] presented a randomized backtracking algorithm that solves the problem in expected subexponential time $2^{\mathcal{O}(\sqrt{|V|})}$.

The search problem asks for the highest probability that β_{max} is reached, i.e. for the maximum possible value of $\varphi(\alpha)$.

This problem has, as opposed to the problems inspected so far, no violations. There *always* exists a unique solution unconditionally.

Theorem 5.5.7. *There exists a promise preserving polynomial time reduction from BINARY-SSG to P-LCP [GR05, p. 211].*

Theorem 5.5.8. *There exists a promise preserving polynomial time reduction from SSG to PL-CONTRACTION [EY10, Corollary 5.3].*

Theorem 5.5.9. *Because of Theorem 5.5.7 or 5.5.8, BINARY-SSG is in UniqueEOPL. By Theorem 5.5.4 on page 83, SSG is also in UniqueEOPL.*

5.6. Discounted Game

Discounted games are another version of the MEAN PAYOFF GAME (MPG) (see Section 5.7 on the next page). They were first introduced by [ZP96] as intermediate step of the reduction from MPG to SSG.

Definition 5.6.1 (DISCOUNTED GAME). Let $G = (V, E)$ be a finite directed graph where each vertex has at least one outgoing edge and let $o \in V$ be the start node. The vertices are partitioned into two sets V_{max} and V_{min} such that $V_{max} \cup V_{min} = V$ and $V_{max} \cap V_{min} = \emptyset$. Let $c: E \rightarrow \{-w, \dots, 0, \dots, w\}$ be a function that assigns an integer weight to each edge.

There are two players, Player *Max* and Player *Min*. A token is placed on the start node o . Player *Max* starts to choose an edge along which the token is moved. Then it is Player *Min*'s turn to choose the next edge, and so on, indefinitely. The chosen edges are numbered: (e_0, e_1, \dots) .

Let $0 < \lambda < 1$ be a real value, called discounting factor. For each edge in the sequence of chosen edges, the weight of the i 'th edges is multiplied by $(1 - \lambda)\lambda^i$. The outcome of the game is defined as

$$(1 - \lambda) \sum_{i=0}^{\infty} \lambda^i w(e_i).$$

Player *Max* tries to maximize this outcome, while Player *Min* tries to minimize it.

The unique value $\varphi(v)$ of the game starting at v can be calculated by the following equation [ZP96, Theorem 5.1]:

$$\varphi(v) = \begin{cases} \max_{(v,u) \in E} (1 - \lambda)w(v, u) + \lambda\varphi(u) & \text{if } v \in V_{max} \\ \min_{(v,u) \in E} (1 - \lambda)w(v, u) + \lambda\varphi(u) & \text{if } v \in V_{min}. \end{cases}$$

The decision version of the problem asks whether $\varphi(o)$ is smaller than some value k . The goal of the search problem is to calculate the value of $\varphi(o)$.

There is no polynomial time algorithm known for that problem [ZP96]. Since there is a reduction to SSG, the subexponential algorithms that can be used to solve an SSG can also be used to solve the DISCOUNTED GAME.

Theorem 5.6.2. *There exists a polynomial time reduction from DISCOUNTED GAME to SIMPLE STOCHASTIC GAME (SSG) [ZP96].*

Thus, the decision version of DISCOUNTED GAME's is in $\text{NP} \cap \text{co-NP}$ and the search version is in UniqueEOPL.

5.7. Mean Payoff Game (MPG)

A mean payoff game is a deterministic 2-player game played on a directed, weighted graph and was first introduced by [EM79]. For a good summary and proofs for the complexity of this problem see [ZP96].

Definition 5.7.1 (MEAN PAYOFF GAME (MPG) [ZP96]). Let $G = (V, E)$ be a finite directed graph where each vertex has at least one outgoing edge and let $o \in V$ be the start node. Let $c: E \rightarrow \{-w, \dots, 0, \dots, w\}$ be a function that assigns an integer weight to each edge.

As in DISCOUNTED GAME's, there are two players, Player *Max* and Player *Min*. A token is placed on the start node o . Player *Max* starts to choose an edge along which the token is moved. Then it's Player *Min*'s turn to choose the next edge, and so on, indefinitely. The chosen edges are numbered: (e_1, e_2, \dots) .

Player *Max* tries to maximize the sum of edge weights along which the token is moved, while Player *Min* tries to minimize it. More formally, Player *Max* tries to maximize $\liminf_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n c(e_i)$ and Player *Min* tries to minimize $\limsup_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n c(e_i)$.

[EM79, Theorem 1] shows that there exists a value φ with

$$\liminf_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n c(e_i) \geq \varphi \geq \limsup_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n c(e_i)$$

when both players use an optimal strategy that only depends on the node on which the token currently is and not on what happened previously in the game.

The decision version of the problem asks whether φ is smaller than some value k . The goal of the search problem is to calculate this value.

There are different versions of this game:

- **BIPARTITE-MPG** : The vertices are partitioned into two sets V_{max} and V_{min} such that $V_{max} \cup V_{min} = V$ and $V_{max} \cap V_{min} = \emptyset$. Whenever the token is on a vertex in V_{max} , Player *Max* chooses the next edge and when the token is on a vertex in V_{min} , Player *Min* is allowed to choose. Any MPG can be transformed to an BIPARTITE-MPG with a bipartite graph by duplicating all vertices and adjusting the edges properly [ZP96].
- **FINITE-MPG** : [EM79] proposes a version in which the game stops as soon as the chosen edges form a cycle. The value of the finite MPG is equivalent to infinite MPG [EM79, Theorem 2].

There is no polynomial time algorithm known to solve a MEAN PAYOFF GAME (MPG). It exists an algorithm to find the optimal strategies that runs in $\mathcal{O}(|V|^4 \cdot |E| \log(|E|/|V|) \cdot w)$ pseudo-polynomial time in w [ZP96, Theorem 3.1] and an algorithm that runs in $\mathcal{O}(|V|^3 \cdot |E| \cdot w)$ pseudo-polynomial time in w to find the optimal value of the game [ZP96, Theorem 2.3]. Furthermore, since there is a reduction to DISCOUNTED GAME and by extension to SSG, the subexponential algorithms that can be used to solve an SSG can also be used to solve the MEAN PAYOFF GAME (MPG).

Theorem 5.7.2. *There exists a polynomial time reduction from MEAN PAYOFF GAME (MPG) to DISCOUNTED GAME [ZP96, Section 5].*

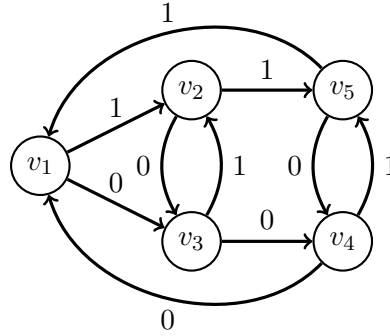
By that Theorem it follows that the decision version of MEAN PAYOFF GAME (MPG) is in $\text{NP} \cap \text{co-NP}$ and the search version is in UniqueEOPL. The same holds for BIPARTITE-MPG and FINITE-MPG .

5.8. Parity Game

A ω -Automata (also called Church's problem) consists of a transition structure \mathcal{T} and an objective function σ .

The transition structure \mathcal{T} consists of $\mathcal{T} = (V, o, \Sigma, S)$, with a set of vertices V , a start node $o \in V$, an alphabet Σ and a transition function $S: V \times \Sigma \rightarrow V$. If the game is deterministic, the transition function S is total [Kri+95, Section 2.1], i.e. for each node and for each element of the alphabet, there is a successor node.

Example 5.8.1. Let $\Sigma = \{0, 1\}$, $V = \{v_1, \dots, v_5\}$ and $o = v_1$. The transition function is indicated by the arrows.



A *run* r is the infinite sequence of nodes visited during an infinite game with $r(0) = o$. $inf(r) \subseteq V$ is the set of vertices that are visited infinitely often during a run r .

The objective function σ gets a run and returns either 0 if it rejects it or 1 if it accepts it. σ is a boolean formula where the nodes in V are interpreted as boolean variables. A variable v_i is assigned the value 1 if and only if $v_i \in inf(r)$ and 0 otherwise. There are different classical examples of objective functions [Pur95, Definition 5.3.4]:

- A *disjunctive formula* (or *Buchi formula*): For $A = \{A_1, \dots, A_k\} \subseteq V$, $\sigma(inf(r)) := \bigvee_{i=1}^k A_i$. The objective function accepts if and only if $A \cap inf(r) \neq \emptyset$.
- A *Rabin formula* of the form $\sigma(inf(r)) := \bigvee_{i=1}^k (A_i \wedge \neg B_i)$, where A_i, B_i for $i = 1, \dots, k$ are disjunctive formulas. The objective function accepts if and only if there exists a pair (A_i, B_i) with $A_i \subseteq inf(r)$ and $B_i \not\subseteq inf(r)$. It still might happen that B_i has been visited once by the run, but not infinitely many times.
- A *Streett formula* of the form $\sigma(inf(r)) := \bigwedge_{i=1}^k (A_i \vee \neg B_i)$, where A_i, B_i for $i = 1, \dots, k$ are disjunctive formulas. The objective function accepts a run if and only if for all $i = 1, \dots, k$: $A_i \subseteq inf(r)$ or $B_i \not\subseteq inf(r)$.
- A *Chain formula*: Let $A_k \subset B_k \subset A_{k-1} \subset \dots \subset A_1 \subset B_1 \subseteq V$ where each A_i and B_i are disjunctive formulas. The objective function is defined as $\sigma(inf(r)) := \bigvee_{i=1}^k (B_i \wedge \neg A_i)$. Intuitively, the ω -Automata accepts, if and only if there exists some pair (A_i, B_i) for which A_i is visited only finitely often and B_i infinitely often.

The chain formula can be stated equivalently as described in [Tho97, Definition 6.2]: For each $i = 1, \dots, k$, assign the index (or priority) $2i - 1$ to the elements of the set $A_i \setminus B_{i-1}$ and the index $2i$ to the set $B_i \setminus A_i$. Then look at all elements in $inf(r)$ and their corresponding set indices. If the smallest index in the set is even, then the run is accepted.

Example 5.8.2. Recall the transition structure from Example 5.8.1 on the preceding page. $V = \{v_1, \dots, v_5\}$ and

- $B_2 = \{v_2, v_3, v_4, v_5\}$
- $A_2 = \{v_2, v_3, v_4\}$
- $B_1 = \{v_2, v_3\}$
- $A_1 = \{v_2\}$.

It holds that $A_1 \subset B_1 \subset A_2 \subset B_2$.

The Chain formula then is

$$\sigma = ((v_2 \vee v_3 \vee v_4 \vee v_5) \wedge \neg(v_2 \vee v_3 \vee v_4)) \vee ((v_2 \vee v_3) \wedge \neg(v_2)).$$

For $i = 2$

- all elements of the set $A_2 \setminus B_1 = \{v_4\}$ are indexed with 3 and
- all elements of the set $B_2 \setminus A_2 = \{v_5\}$ are indexed with 4.

For $i = 1$

- all elements of the set $A_1 = \{v_2\}$ are indexed with 1 and
- all elements of the set $B_1 \setminus A_1 = \{v_3\}$ are indexed with 2.

If $\text{inf}(r) = \{v_1, v_3, v_4\}$, then the $v_1 = v_3 = v_4 = 1$ and $v_2 = v_5 = 0$. With this assignment, the formula *is* satisfied. Equivalently, the smallest index assigned to the elements in $\text{inf}(r)$ is 2, which is assigned to v_3 . This index *is* even, hence this run is accepted.

If $\text{inf}(r) = \{v_1, v_2, v_5\}$, then the $v_1 = v_2 = v_5 = 1$ and $v_3 = v_4 = 0$. With this assignment, the Chain formula is not satisfied. Equivalently, the smallest index assigned to the elements in $\text{inf}(r)$ is 1, which is assigned to v_2 . This index is not even, hence this run is not accepted.

A game with any of the above mentioned formulas as objective function can be translated to a Chain formula, i.e. a Chain game [Pur95, Theorem 5.3.2].

A PARITY GAME is a Chain game with two players, where Player *Max* tries to play such that that σ accepts the resulting run, Player *Min* tries to play such that the result is rejected.

Deciding a PARITY GAME means, given a start node, decide which player wins. The search version asks for the strategy with which the player can win. Solving a chain game is polynomial time equivalent to the model checking problem for μ -calculus [Pur95, Theorem 5.3.6, Theorem 5.3.7]. The decision problem of the Chain game is in $\text{NP} \cap \text{co-NP}$ [Pur95, p. 72].

There is no polynomial time algorithm known for that problem [Pur95]. Since there is a reduction to MEAN PAYOFF GAME (MPG) and by extension to DISCOUNTED GAME and SSG, the sub-exponential algorithms that can be used to solve an SSG can also be used to solve the PARITY GAME.

Theorem 5.8.3. *There exists a polynomial time reduction from PARITY GAME to MEAN PAYOFF GAME (MPG) [Pur95, Section 5.5].*

Theorem 5.8.4. *There exists a polynomial time reduction from PARITY GAME to SIMPLE STOCHASTIC GAME (SSG) [CF11].*

Thus, the search version of the parity game is in UniqueEOPL.

5.9. α -Ham Sandwich

The α -HAM SANDWICH problem is a discrete version based on the Ham-Sandwich Theorem [ST42] as introduced by [SZ10].

Let $\kappa_1, \dots, \kappa_d$ be a collection of d finite point sets in \mathbb{R}^d . Each set is interpreted as representation of one color. The set $N = \kappa_1 \cup \dots \cup \kappa_d$ is the union of all points.

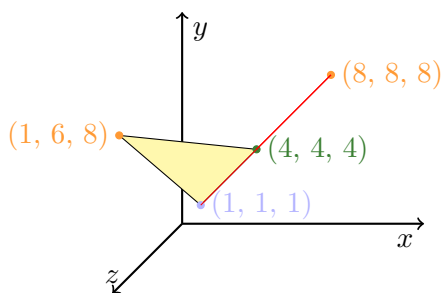
A set $\{p_1, \dots, p_m\} \subseteq N$ is called *colorful* if no two points p_i, p_j have the same color, i.e. if for all $i, j = 1, \dots, m: i \neq j \wedge p_i \in \kappa_l \Rightarrow p_j \notin \kappa_l$.

As defined in [SZ10], a set N has a *very weak general position* if for every choice C of points $p_1 \in \kappa_1, \dots, p_d \in \kappa_d$ the affine hull of the set $C = \{p_1, \dots, p_d\}$ is $(d-1)$ -flat and does not contain any other point of N .

[CCM20] extended the definition as follows: Let $C \subset N$ be a set with $|C| = d$ and C contains points with exactly $d-1$ different colors, i.e. one color is in the set twice. If for every choice of C the affine hull of C is $(d-1)$ -flat and does not contain any other point of N , C has a *weak general position*. Checking if a planar point set is in weak general position is NP-hard [CCM20, p. 5].

An affine set is a set A that contains for each two points $x, y \in A$ also all points on the line between x and y . The affine hull of a set N is the smallest affine set containing N [JS19, Definition 7.1.8.]. For example, the affine hull of three points in 3 dimensions is the plane that contains all three points. If they are on a line, then the affine hull is the line segment that goes through all of them. If all points are equal, the affine hull is the set of the point itself. The d in the name d -flat corresponds to the dimension of the affine hull. A plane is 2-flat, a line is 1-flat and a point is 0-flat. Analogously this works for higher dimensions.

Example 5.9.1. Let $d = 3$, $\kappa_1 = \{(1, 1, 1)\}$, $\kappa_2 = \{(8, 8, 8), (1, 6, 8)\}$ and $\kappa_3 = \{(4, 4, 4)\}$. It follows that $N = \{(1, 1, 1), (8, 8, 8), (1, 6, 8), (4, 4, 4)\}$.



The set $\{(1, 1, 1), (8, 8, 8), (4, 4, 4)\}$ is colorful, as well as the set $\{(1, 1, 1), (1, 6, 8), (4, 4, 4)\}$.

The set N is not in weak general position. The affine hull of $\{(1, 1, 1), (1, 6, 8), (4, 4, 4)\}$ is a plane (colored in yellow) and therefore 2-flat, but the affine hull of $\{(1, 1, 1), (8, 8, 8), (4, 4, 4)\}$ is a line (colored in red) and only 1-flat.

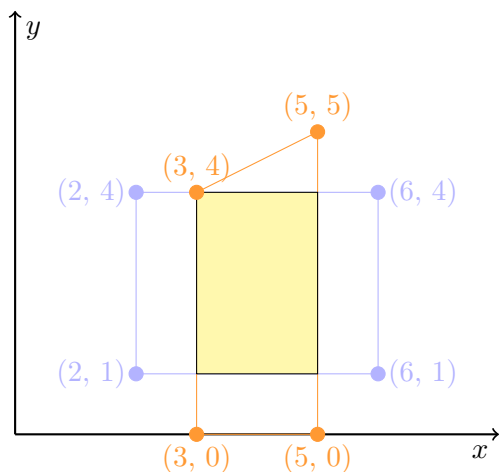
A point set N is *well separated* if for every choice of points from the convex hull of each color $q_1 \in \text{conv}(\kappa_{i_1}), \dots, q_k \in \text{conv}(\kappa_{i_k})$, where $i_1, \dots, i_k \in \{1, \dots, d\}$ are distinct indices and $1 \leq k \leq d+1$, the affine hull of $\{q_1, \dots, q_k\}$ is $(k-1)$ -flat.

Equivalently, N is well separated if for every two index sets $I, J \subset \{1, \dots, d\}$ there exists a hyperplane that strictly separates the sets $\text{conv}(\{\bigcup_{i \in I} \kappa_i\})$ and $\text{conv}(\{\bigcup_{j \in J} \kappa_j\})$.

Intuitively this means that no two convex hulls of the different colors (or union of several colors) are allowed to intersect. There must exist a hyperplane that strictly separates them.

The complexity of testing well-separation is unknown [CCM20, p. 5].

Example 5.9.2. Let $d = 2$, $\kappa_1 = \{(2, 1), (2, 4), (6, 1), (6, 4)\}$ and $\kappa_2 = \{(3, 0), (5, 0), (3, 4), (5, 5)\}$. It follows that $N = \{(2, 1), (2, 4), (6, 1), (6, 4), (3, 0), (5, 0), (3, 4), (5, 5)\}$.



The convex hull of κ_1 is the square colored in blue. The convex hull of κ_2 is the square colored in orange.

This example is not well separated. The convex hulls intersect each other (colored in yellow).

The point $(4, 3)$ is in both convex hulls. The affine hull of the choice $q_1 = q_2 = (4, 3)$ is 0-flat since it is only a single point.

For any set of positive integers $\{\alpha_1, \dots, \alpha_d\}$ with $\alpha_i \in \{1, \dots, |\kappa_i|\}$ for every $i = 1, \dots, d$, a $(\alpha_1, \dots, \alpha_d)$ -cut is an oriented hyperplane H that contains at least one point from each color, i.e. $H \cap \kappa_i \neq \emptyset$ and satisfied $|H^+ \cap \kappa_i| = \alpha_i$ for every $i = 1, \dots, d$. H^+ is the closed positive half space defined by H [SZ10, Definition 4].

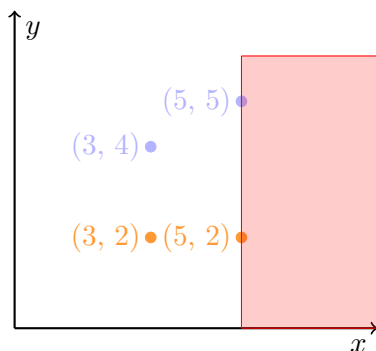
Intuitively, an α -cut is a hyperplane going through at least one point of each color and that cuts the points in half such that there are exactly α_i many points of color κ_i on one side of the plane.

Theorem 5.9.3. (*α -Ham-Sandwich Theorem [SZ10, Theorem 1]*)

Let $\kappa_1, \dots, \kappa_d$ be finite, well-separated point sets in \mathbb{R}^d . Let $\alpha = (\alpha_1, \dots, \alpha_d)$ be a vector with $\alpha_i \in \{1, \dots, |\kappa_i|\}$ for all $i = 1, \dots, d$.

1. If an α -cut exists, then it is unique.
2. If N has a weak general position, then a cut exists for each choice of α with $\alpha_i \in \{1, \dots, |\kappa_i|\}$.

Example 5.9.4. Let $d = 2$, $\kappa_1 = \{(3, 4), (5, 5)\}$ and $\kappa_2 = \{(3, 2), (5, 2)\}$. It follows that $N = \{(3, 4), (5, 5), (3, 2), (5, 2)\}$.



The given example is well-separated and the points are in weak general position. By Theorem 5.9.3 on the previous page, for each choice of α with $\alpha_i \in \{1, \dots, |\kappa_i|\}$, there exists a unique α -cut.

Let for example $\alpha = (1, 1)$. There are infinitely many hyperplanes that divide the points such that there is one point of each color on its side, but only one line that also *contains* a point of each color. It is the line that is defined by $(5, 5)$ and $(5, 2)$ (colored in red).

Definition 5.9.5 (The search problem α -HAM SANDWICH [CCM20]). Given d many finite sets of points $N = \kappa_1 \cup \dots \cup \kappa_d$ in \mathbb{R}^d and a vector $(\alpha_1, \dots, \alpha_d)$ of positive integers with $\alpha_i \leq |\kappa_i|$ for all $i = 1, \dots, d$, find one of the following:

(HSS) A $(\alpha_1, \dots, \alpha_d)$ -cut.

(HSV1) A subset of N of size $(d+1)$ and at least $(d-1)$ colors that lie on a hyperplane.

(HSV2) A disjoint pair of sets $A, B \subset \{1, \dots, d\}$ such that:

$$\text{conv}(\bigcup_{a \in A} \kappa_a) \cap \text{conv}(\bigcup_{b \in B} \kappa_b) \neq \emptyset.$$

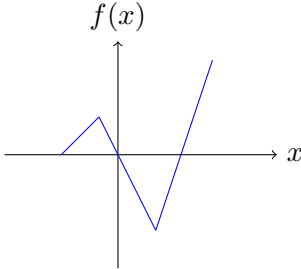
There is no polynomial time algorithm known that solves α -HAM SANDWICH [CCM20, p. 2]. Since it can be reduced to UNIQUE EOPL, it can be solved by Aldous algorithm in $\mathcal{O}(1, 4143^d)$ time.

Theorem 5.9.6. *There exists a promise preserving polynomial time reduction from α -HAM SANDWICH to UNIQUE END OF POTENTIAL LINE and thus, α -HAM SANDWICH is in UniqueEOPL [CCM20, Section 3, Lemma 17, Theorem 10].*

5.10. Pairwise Linear Contraction (PL-Contraction)

A pairwise linear function is a function that is composed of linear segments.

Example 5.10.1. Example of a pairwise linear function:

$$f(x) = \begin{cases} x + 3 & \text{if } x < -1 \\ -2x & \text{if } -1 \leq x < 2 \\ 3x - 10 & \text{if } 2 \leq x. \end{cases}$$


The function f is given as an arithmetic circuit (also called LinearFIXP circuit) consisting of the operations \min , \max , $+$, $-$ and $\times b$ (which is the multiplication by a constant b).

The function $f : [0, 1]^d \rightarrow [0, 1]^d$ is *contracting* under the ℓ_p norm with $p \in \mathbb{N} \cup \{\infty\}$ if $\forall x, y \in [0, 1]^d : |f(x) - f(y)|_p \leq \text{const} \cdot |x - y|_p$ for some constant $0 < \text{const} < 1$. The Banach fixed point theorem states that for a function f that is contracting under a metric δ , there exists a unique fixpoint of f , i.e. $f(x) = x$ [Bar11, p. 146].

The problem PL-CONTRACTION is the problem of finding a fixpoint of a pairwise linear contraction map [Fea+20b, Section 5.2]. The definition of the actual search problem however is strongly shaped by the reduction that is done later from PL-CONTRACTION to OPDC. There are a few preliminaries needed.

Definition 5.10.2 (Continuous i -Slice). Recall Definition 4.1.5 on page 25 of a slice and Definition 4.1.7 on page 25 of an i -Slice for a discrete grid. Let an i -Slice for a continuous function be the following:

A slice is a vector $\mathbf{s} = (\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_d) \in \text{Slices}_d$ with $\mathbf{s}_i \in [0, 1] \cup \{*\}$. If $\mathbf{s}_i \in [0, 1]$, this means that dimension i is fixed. If $\mathbf{s}_i = *$, dimension i is free to vary. An i -Slice is a slice \mathbf{s} for which holds:

$$\forall j = 1, \dots, d : (j \leq i \Rightarrow \mathbf{s}_j = *) \wedge (j > i \Rightarrow \mathbf{s}_j \neq *).$$

A point $x \in [0, 1]^d$ is a fixpoint of a slice \mathbf{s} if $(x - f(x))_i = 0$ for all i where $\mathbf{s}_i = *$.

Since OPDC works with a discrete grid, the continuous space on which the function f operates needs to be discretised. For a valid PL-CONTRACTION instance it must be possible to discretise the space such that all fixpoints of f in all i -Slice's are contained within the grid.

Lemma 5.10.3. [Fea+20b, Lemma 26] *There exist integers $\omega = (\omega_1, \dots, \omega_d)$ such that for every i -Slice \mathbf{s} it holds that if $x \in [0, 1]^d$ is a fixpoint of \mathbf{s} according to f , then there exists a point $p \in [0, 1, \dots, \omega_1] \times [0, 1, \dots, \omega_2] \times \dots \times [0, 1, \dots, \omega_d]$ such that for all i where $\mathbf{s}_i = *$*

- $f(p_1/\omega_1, \dots, p_d/\omega_d)_i = p_i/\omega_i$ and
- $p_i = \omega_i \cdot x_i$.

Definition 5.10.4 (PL-CONTRACTION [Fea+20b]). Given a LinearFIXP circuit computing $f : [0, 1]^d \rightarrow [0, 1]^d$, a constant $a \in (0, 1)$ and $p \in \mathbb{N} \cup \{\infty\}$, find one of the following:

- (PLCS) A point $x \in [0, 1]^d$ such that $f(x) = x$.
- (PLCV1) Two points $x, y \in [0, 1]^d$ such that $|f(x) - f(y)|_p > a \cdot |x - y|_p$. This violation proves that f is not contracting.
- (PLCV2) A point $x \in [0, 1]^d$ such that $f(x) \notin [0, 1]^d$. This violation is necessary to make the problem total, since f might be contracting, but its fixpoint is not in $[0, 1]^d$.
- (PLCV3) An i -Slice \mathbf{s} and two points $x, y \in [0, 1]^d$ in \mathbf{s} such that:
 - $(f(x) - x)_j = (f(y) - y)_j$ for all $j < i$,
 - $\omega_i \cdot x_i = \omega_i \cdot y_i + 1$ where ω_i is the integer given by Lemma 5.10.3 on the preceding page, and
 - $f(x)_i < x_i$ and $f(y)_i > y_i$.

This violation is needed for the reduction to OPDC, more precisely to map the violation (OV2) back to a violation in PL-CONTRACTION.

There is no polynomial time algorithm known to solve that problem. [Fea+18, Theorem 40] states that there exists an algorithm that is polynomial in the size of the LinearFIXP circuit f and exponential in the size of d .

Theorem 5.10.5. *There exists a promise preserving polynomial time reduction from PL-CONTRACTION to OPDC [Fea+20b, Lemma 29]. Thus, PL-CONTRACTION is in UniqueEOPL.*

6. Trying to prove Grid-USO to be in UniqueEOPL

[GWJR+08] introduced the problem of finding a unique sink orientation on grids. Since it is very similar to unique sink orientations of cubes and also has a unique solution, it seemed a reasonable suggestion that GRID-USO might be in UniqueEOPL as well. This result also proves the general version of the P-Matrix linear complementarity problem to be in UniqueEOPL, since [GWJR+08; Rüs07] showed that there is a polynomial time reduction from GLCP to GRID-USO.

It took several attempts to prove that GRID-USO actually is in UniqueEOPL. Since the failed reduction is part of what I did for this master thesis, I will include it in this chapter as well as the reason why it did not work. Hopefully this has some educational value for the reader and deepens the understanding of these types of reductions.

6.1. Grid-USO

Definition 6.1.1 (*d*-dimensional grid [Rüs07]). Let $N = \{1, \dots, n\}$ be a well-ordered set and $K = (\kappa_1, \dots, \kappa_d)$ be a partition of N with $|\kappa_i| \geq 2$. Every block κ_i is well-ordered. The *d*-dimensional grid $\Gamma = (N, K, V, E)$ is the undirected graph with

$$V := \{v \subseteq N \mid i = 1, \dots, d, |v \cap \kappa_i| = 1\}$$

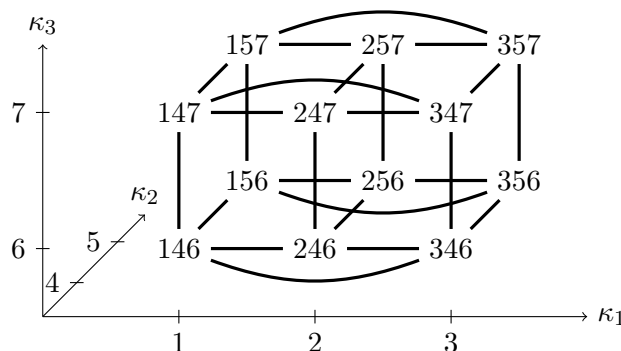
$$E := \{(v, u) \mid |v \oplus u| = 2\}.$$

The vertices correspond to the Cartesian product of K . A node $v \in V$ contains *d* many elements, exactly one of each block: $|v| = d$.

There is an edge between two nodes whenever they differ in exactly one value. We refer to each block as one dimension. All vertices in one dimension are a clique, i.e. their induced subgraph is complete. Every vertex has $(n - d)$ many edges.

Example 6.1.2. A *d*-dimensional grid for $d = 3$ [Rüs07, Figure 3.1].

Let $N = \{1, \dots, 7\}$ and $\kappa_1 = \{1, 2, 3\}$, $\kappa_2 = \{4, 5\}$ and $\kappa_3 = \{6, 7\}$.



All the sets we are talking about in this section are well-ordered sets. This means that we can refer to the i 'th element of a set, e.g. for a node $v \in V$, v_i is the element for which holds that $|v \cap \kappa_i| = 1$, i.e. v_i is an element in κ_i .

Let $v_{a,b} := \{v_k \mid a \leq k \leq b\}$ be the subset of elements of a point from index a to index b .

Lemma 6.1.3. $\forall v, u \in V, a, b \in \{1, \dots, d\}$ it holds that $a \neq b \Rightarrow v_a \neq u_b$. For each node it holds that if we look at two different indices of a node, their values cannot be the same.

Proof. Intuitively this holds because we constructed the nodes such that each entry i only contains an element of κ_i .

Let $v, u \in V$ and $a, b \in \{1, \dots, d\}$ with $a \neq b$. It holds that $v_a \in \kappa_a$ and $v_b \in \kappa_b$. Since $\kappa_a \cap \kappa_b = \emptyset$, this implies that $v_a \neq v_b$.

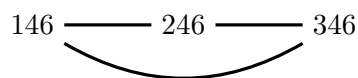
□

Definition 6.1.4 (Subgrid). Every subset $N' \subseteq N$ defines a subgrid $\Gamma' = (N', K', V', E')$ of Γ with $\kappa'_i := \kappa_i \cap N'$ for all $i = 1, \dots, d$.

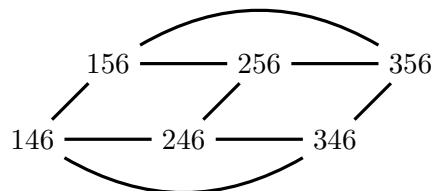
Note that in a subgrid it is not necessary that for each block i $|\kappa_i| \geq 2$. In a subgrid it holds that $|\kappa'_i| \geq 1$. There must be at least one element of each block contained in the subgrid since otherwise the subgrid would not be distinctly defined.

Example 6.1.5. Recall Example 6.1.2 on the previous page. Here are a few examples of subgrids.

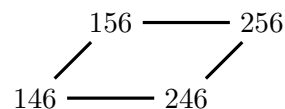
(i) Let $N' = \{1, 2, 3, 4, 6\}$. It follows that $\kappa'_1 = \kappa_1$, $\kappa'_2 = \{4\}$ and $\kappa'_3 = \{6\}$.



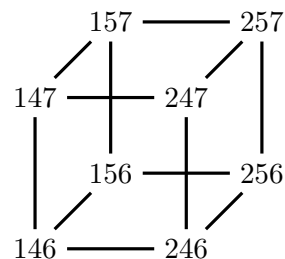
(ii) Let $N' = \{1, 2, 3, 4, 5, 6\}$. It follows that $\kappa'_1 = \kappa_1$, $\kappa'_2 = \kappa_2$ and $\kappa'_3 = \{6\}$.



(iii) Let $N' = \{1, 2, 4, 5, 6\}$. It follows that $\kappa'_1 = \{1, 2\}$, $\kappa'_2 = \kappa_2$ and $\kappa'_3 = \{6\}$.



(iv) Let $N' = \{1, 2, 4, 5, 6, 7\}$. It follows that $\kappa'_1 = \{1, 2\}$, $\kappa'_2 = \kappa_2$ and $\kappa'_3 = \kappa_3$.



A node has neighbors in $(n - d)$ many *directions*. They are not called dimensions on purpose because they aren't really dimensions. For every block κ_i the node has $|\kappa_i| - 1$ many directions in the dimension i . If the grid was a cube, the directions are the same as the dimensions.

Let $\mathcal{N}(v)$ be the set of all neighboring nodes of a given node v : $\mathcal{N}(v) := \{u \in V | (v, u) \in E\}$.

Let $\mathcal{D}(v) := \{i \in N | u \in \mathcal{N}(v) \wedge \{i\} = u \setminus v\} = N \setminus v$ be the set of the numbers which the neighbors of v contain, but not v . One element of this set is the direction in which v and one of its neighbors differ. The set is well-ordered.

For $v_{a,b}$, $\mathcal{D}(v_{a,b}) := \{i \in \bigcup_{j=a,\dots,b} \kappa_j | u \in \mathcal{N}(v) \wedge \{i\} = u \setminus v\} = \bigcup_{j=a,\dots,b} \kappa_j \setminus v$. Here only the directions for dimensions a to b are returned.

Let $\mathcal{N}(v, i) := u$ with $(v, u) \in E$ and $\{i\} = u \setminus v$. If $i \in v$, then $\mathcal{N}(v, i) := \emptyset$. The function returns the neighbor of v in direction i .

Definition 6.1.6 (Orientation function). Let $\phi: V \rightarrow \{0, 1\}^n$ be an orientation function that assigns each edge a direction such that

- $\phi_i(v) = 1 \Leftrightarrow v$ has an outgoing edge in direction $i \in \mathcal{D}(v)$, i.e. v points towards $\mathcal{N}(v, i)$.
- $\phi_i(v) = 0 \Leftrightarrow i \in v$ or v has an incoming edge in direction $i \in \mathcal{D}(v)$, i.e. $\mathcal{N}(v, i)$ points towards v .

Definition 6.1.7 (Outmap function). The outmap function returns a subset of $\mathcal{D}(v)$ containing all numbers of the neighboring nodes to which v points.

$$\sigma_\phi(v) := \{j \in \mathcal{D}(v) | \phi_j(v) = 1\}$$

$$\sigma_\phi(v_{a,b}) := \{j \in \mathcal{D}(v_{a,b}) | \phi_j(v) = 1\}.$$

In the following, for shortness the ϕ in the index will be omitted whenever it is clear which orientation function is meant and σ_ϕ will be just written as σ .

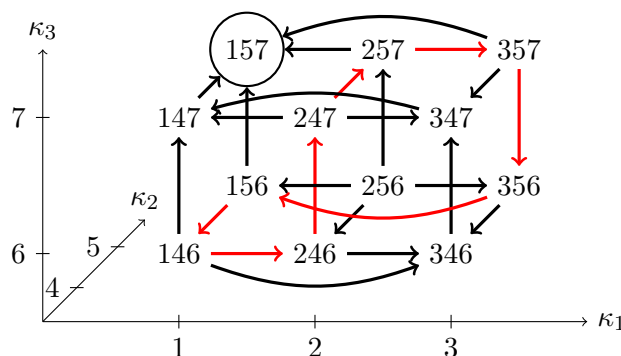
For a subgrid Γ' let σ' be the outmap function on that subgrid, i.e. for $v \in \Gamma'$: $\sigma'(v) = \sigma(v) \cap N'$.

Definition 6.1.8 (Unique sink orientation on grids). An orientation ϕ is called *unique sink orientation* if all nonempty subgrids have a unique sink [Rüs07, Definition 2.3].

As with cubes, a unique sink orientation of grids may contain cycles.

Example 6.1.9. A d -dimensional grid for $d = 3$ and a unique sink orientation with a cycle colored in red [Rüs07, Figure 3.1].

Let $N = \{1, \dots, 7\}$ and $\kappa_1 = \{1, 2, 3\}$, $\kappa_2 = \{4, 5\}$ and $\kappa_3 = \{6, 7\}$.



6.1.1. Grid-USO- Failed definition

Definition 6.1.10 (*h-vector* [GWJR+08]). Let $h(\sigma) = (h_0(\sigma), \dots, h_{n-d}(\sigma))$ be the vector that contains at position r the number of nodes that have r outgoing edges:

$$h_r(\sigma) := |\{v \in V \mid |\sigma(v)| = r\}|.$$

Lemma 6.1.11. *For any two unique sink orientations ϕ and ϕ' of a grid Γ , it holds that $h(\sigma_\phi) = h(\sigma_{\phi'})$ [GWJR+08, Theorem 2.6].*

It immediately follows that all unique sink orientations of a grid have the same h -vector. For a given grid this vector is called $h(\Gamma)$.

The h -vector is symmetric, i.e. $h(\Gamma)_r = h(\Gamma)_{n-d-r}$.

Lemma 6.1.12. *The h -vector of a given grid Γ can be calculated by the following recursive formula in polynomial time:*

$$\begin{aligned} h_r(\Gamma) &:= \mathbf{count}(r, d) \\ \mathbf{count}(r, t) &:= \sum_{a=0}^{|\kappa_t|-1} \mathbf{count}(r-a, t-1) \\ \mathbf{count}(r, 1) &:= \begin{cases} 1 & \text{if } 0 \leq r < |\kappa_1| \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Proof. We construct the orientation $\tilde{\phi}$ as:

$$\tilde{\phi}_i(v) := \begin{cases} 0 & \text{if } \mathcal{N}(v, i)_i > v_i \text{ or } i \in v \\ 1 & \text{if } \mathcal{N}(v, i)_i < v_i. \end{cases}$$

$\tilde{\phi}$ points in each direction i from the larger to the smaller node, i.e. an edge (v, u) in direction i points from v towards u if $v_i > u_i$. It is easy to see that the result is always a unique sink orientation. The bottom left node is then the unique sink. In the CUBE-USO case, this is equivalent to the orientation function being the identity.

Since by Lemma 6.1.11 all h -vectors for a given grid are identical, $h(\sigma_{\tilde{\phi}})$ is equal to $h(\Gamma)$. $h(\Gamma)$ can therefore be calculated by counting the nodes and their outgoing edges in $\tilde{\phi}$.

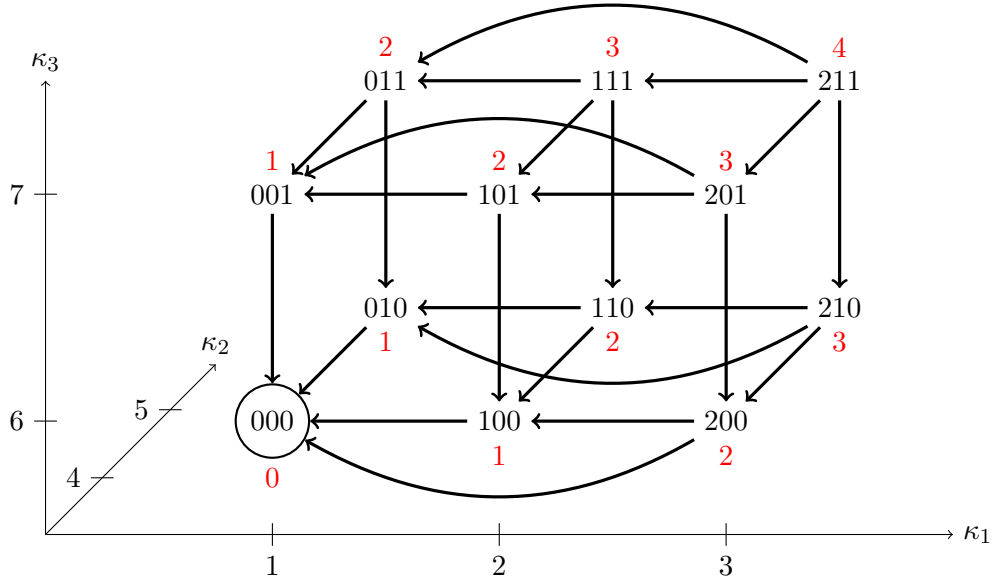
Definition 6.1.13 (Mapping the grid). Let $f: V \rightarrow \{0, \dots, |\kappa_1|-1\} \times \dots \times \{0, \dots, |\kappa_d|-1\}$ be a function that maps $v \in V$ of a grid Γ to a vector. $f_i(v)$ is defined as the number of elements in κ_i that are smaller than v_i . f is bijective. Let f^{-1} be the reverse function of f .

In $\tilde{\phi}$, the number of outgoing edges of each node v is the sum of all entries of $f(v)$:

$$|\sigma_{\tilde{\phi}}(v)| = \sum_{i=1}^d f_i(v).$$

This is due to the fact that $\tilde{\phi}$ always points to the smaller nodes, and for each node, in dimension i there are $f_i(v)$ many smaller nodes it points to.

Example 6.1.14. Recall the grid from the previous examples. The following shows this grid with the alternative node names mapped by f and the orientation function $\tilde{\phi}$. The number in red under each node is the number of this nodes outgoing edges. Observe that this numbers is always equal to the sum of the nodes entries.

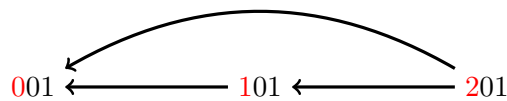


To calculate the r 'th entry of the h -vector, we have to calculate how many vectors in $\{0, \dots, |\kappa_1| - 1\} \times \dots \times \{0, \dots, |\kappa_d| - 1\}$ there are, whose sum of the entries is exactly r .

The recursive step in `count` is doing the following: The value of the last dimension t gets fixed to the value a . Then it is asked how many nodes exist which sum up to $(r - a)$ without using dimension t , i.e. using one dimension less. So for each a we are looking at a different subcube of the original cube and count the occurrences of a node with $(r - a)$ outgoing edges in this subcube. This is done for any possible value that a can take in dimension t , i.e. all values from 0 to $|\kappa_t| - 1$.

When there is only one dimension left, i.e. $d - 1$ dimensions are fixed, we have a single line left.

Example 6.1.15. For example if dimension 3 is fixed to 1 and dimension 2 is fixed to 0, then the following line is left in which the nodes differ only in the first entry:



The number of nodes with r outgoing edges is always 1 if r is within the domain of the respective dimension, and 0 otherwise.

When using dynamic programming and saving for $r = 0, \dots, (n - d)$ and $t = 1, \dots, d$ each result of `count(r, t)` in a table, then the h -vector can be calculated in polynomial time in d and n .

□

Corresponding to Lemma 5.2.8 on page 70 that holds for unique sink orientations of cubes, similar properties for unique sink orientations of grids can be proven.

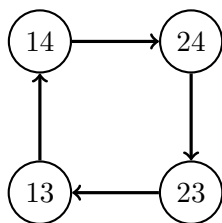
Lemma 6.1.16. [GWJR+08, Lemma 2.8] *Let σ be an outmap. It is the outmap of a unique sink orientation ϕ for a grid $\Gamma = (N, K, V, E)$ if and only if it satisfies the following conditions:*

- (i) $\forall v \in V: v \cap \sigma(v) = \emptyset$
- (ii) $\forall v, u \in V$ with $v \neq u$ it holds that $(v \oplus u) \cap (\sigma(v) \oplus \sigma(u)) \neq \emptyset$
- (iii) $h(\sigma) = h(\Gamma)$.

[GWJR+08, Lemma 2.8] remarks that unlike in the cube USO case, condition (iii) is not implied by the other two conditions as in CUBE-USO and therefore needs to be stated explicitly. Subgrids that have no unique sink but a cycle for example are not covered by the second condition, they are however covered by the third one.

Example 6.1.17. Let $N = \{1, 2, 3, 4\}$, $\kappa_1 = \{1, 2\}$ and $\kappa_2 = \{3, 4\}$.

Condition (ii) is true. For all combinations of nodes the resulting set is nonempty.



$$\begin{aligned} (\{13\} \oplus \{23\}) \cap (\{4\} \oplus \{1\}) &= \{1\} \\ (\{13\} \oplus \{14\}) \cap (\{4\} \oplus \{2\}) &= \{4\} \\ (\{13\} \oplus \{24\}) \cap (\{4\} \oplus \{3\}) &= \{3, 4\} \\ (\{23\} \oplus \{14\}) \cap (\{1\} \oplus \{2\}) &= \{1, 2\} \\ (\{23\} \oplus \{24\}) \cap (\{1\} \oplus \{3\}) &= \{3\} \\ (\{14\} \oplus \{24\}) \cap (\{2\} \oplus \{3\}) &= \{2\} \end{aligned}$$

Condition (iii) does not hold. The h -vector does not correspond to the correct h -vector of that grid: $h(\sigma) = (0, 4, 0) \neq (1, 2, 1) = h(\Gamma)$.

Definition 6.1.18 (GRID-USO (wrong definition)). Given a d -dimensional grid $\Gamma = (N, K, V, E)$ and an orientation function $\phi: V \rightarrow \{0, 1\}^n$, find one of the following:

- (GU1) A node $v \in \Gamma$ with $\sigma_\phi(v) = \emptyset$.
- (GUV1) A node $v \in \Gamma$ with $v \cap \sigma_\phi(v) \neq \emptyset$
- (GUV2) Two nodes $v, u \in \Gamma$ with $(v \oplus u) \cap (\sigma_\phi(v) \oplus \sigma_\phi(u)) = \emptyset$
- (GUV3) A subgrid Γ' and an index i for which $h_i(\sigma'_\phi) \neq h_i(\Gamma')$.

The problem with this definition The problem with this definition is (GUV3). We can calculate $h_i(\Gamma')$ in polynomial time, we can compare two vectors in polynomial time, but we can not verify in polynomial time, whether $h_i(\sigma'_\phi)$ is actually a valid h -vector for the given orientation. In order to do that, we would have to check every node in the subgrid and count their outgoing edges. This might take exponential time. So it is not possible to guess an h -vector and see if this is a violation, because we cannot verify, if the guessed h -vector actually is a valid h -vector for the given orientation function.

6.1.2. Grid-USO- Correct definition

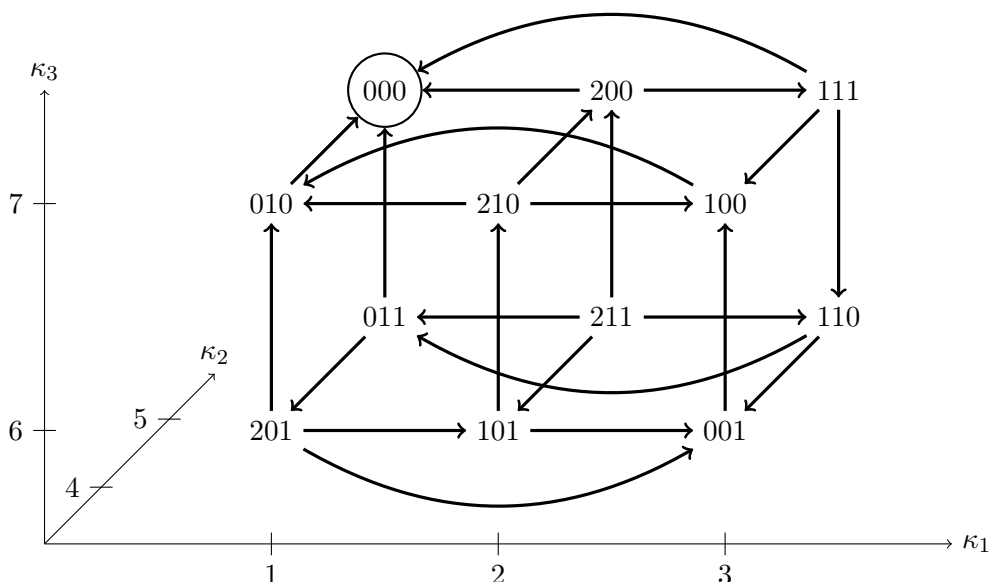
The refined index as defined by [GWJR+08] assigns each node a d -tuple containing at index i the number of outgoing edges in dimension i .

If the orientation always points towards the smaller node (as in Example 6.1.14 on page 100), then the refined index equals the nodes in Example 6.1.14 on page 100. For any other orientation, the node names from Example 6.1.14 on page 100 are still the same, but they might be shuffled through out the grid.

Definition 6.1.19 (Refined Index). The refined index r_σ of an outmap σ_ϕ with $r_\sigma: V \rightarrow \{0, \dots, |\kappa_1| - 1\} \times \dots \times \{0, \dots, |\kappa_d| - 1\}$ is defined as:

$$r_\sigma(v) := (|\sigma(v) \cap \kappa_1|, \dots, |\sigma(v) \cap \kappa_d|).$$

Example 6.1.20. Recall Example 6.1.9 on page 98. Let $N = \{1, \dots, 7\}$ and $\kappa_1 = \{1, 2, 3\}$, $\kappa_2 = \{4, 5\}$ and $\kappa_3 = \{6, 7\}$. The nodes in the figure below are labeled by the refined index.



Theorem 6.1.21. *If σ is a unique sink orientation, then r_σ is a bijection [GWJR+08, Theorem 2.14].*

This Theorem alone is not enough to define a total problem. For example it might happen, that there is an orientation, which is not a unique sink orientation, but still its refined index is a bijection. This is why we need the following Theorem:

Theorem 6.1.22. *σ is a unique sink orientation, if and only if for every subgrid Γ' it holds that r_σ is a bijection.*

Proof. If σ is a unique sink orientation, then all its subgrids are unique sink orientations as well. By Theorem 6.1.21 this means that for each subgrid Γ' , r_σ is a bijection as well.

If for some subgrid Γ' r_σ is a bijection, this means that there exists a unique node v with $r_\sigma(v) = (0, \dots, 0)$. This implies, that v is a unique sink. Therefore, every subgrid has a unique sink, which means that the orientation is a unique sink orientation.

□

Definition 6.1.23 (GRID-USO). Given a d -dimensional grid $\Gamma = (N, K, V, E)$ and an orientation function $\phi: V \rightarrow \{0, 1\}^n \cup \{\perp\}$, find one of the following:

(GU1) A node $v \in \Gamma$ with $\sigma_\phi(v) = \emptyset$.

(GUV1) A node $v \in \Gamma$ with $v \cap \sigma_\phi(v) \neq \emptyset$.

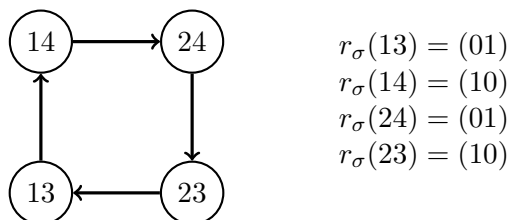
(GUV₄) A subgrid Γ' and two nodes $v, u \in \Gamma'$ with $v \neq u$ and $r_{\sigma'}(v) = r_{\sigma'}(u)$.

By Theorem 6.1.22 on the previous page, (GUV₄) is sufficient to make the problem total.

In comparison to the h-vector, we just need two nodes with the same refined index to prove a violation and not the h-vector for the orientation. The refined index can be checked for any given node in polynomial time.

Furthermore, condition (GUV2) can be removed as well, since it is covered by condition (GUV₄) too.

Example 6.1.24. Let $N = \{1, 2, 3, 4\}$, $\kappa_1 = \{1, 2\}$ and $\kappa_2 = \{3, 4\}$.



The above example is no unique sink orientation. There are several nodes that have the same refined index. Therefore the refined index is not a bijection, which proves that the orientation is no unique sink orientation.

6.2. Grid-USO to OPDC - Failed Construction

Given an instance $I = (\Gamma, \phi)$ of GRID-USO as defined in Definition 6.1.18 on page 101, create an instance $I' = (\Gamma', \mathcal{D})$ of OPDC.

The reduction is similar to the reduction from CUBE-USO to OPDC, with the exception that we need more information to decide whether a direction in OPDC should point upwards or downwards.

The idea is that the grid of the GRID-USO instance can be used as grid for the OPDC instance as well. A sink is encoded by setting the direction to Zero. The directions of the other grid-cells are then defined to point towards the sink.

The grid Since both problems work with a grid, the GRID-USO grid can be just reused for OPDC. Let $\Gamma' := \{0, \dots, |\kappa_1| - 1\} \times \dots \times \{0, \dots, |\kappa_d| - 1\}$. A grid cell can be converted from one grid to the other in polynomial time by f and f^{-1} as defined in Definition 6.1.13 on page 99.

The direction functions The function $\text{sinkIsRightOf}: \{(v, i) \mid v \in V, i \in \{1, \dots, d\}\} \rightarrow \{\text{true}, \text{false}\}$ returns true if the sink is right of given node v in dimension i and false otherwise.

Algorithm 2: sinkIsRightOf

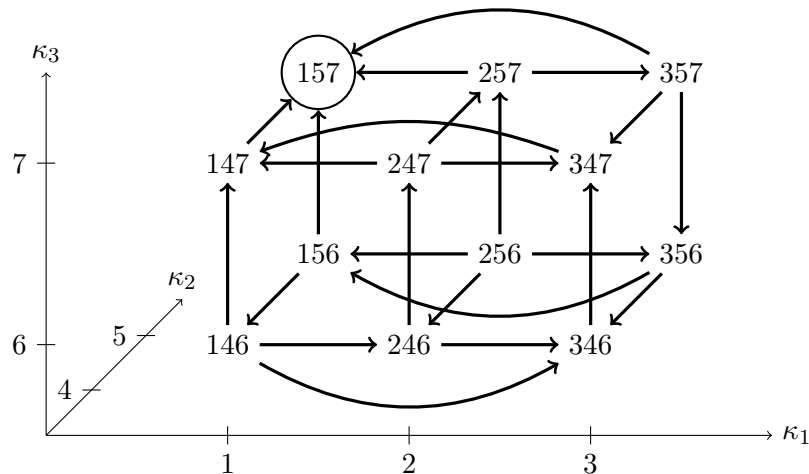
```

Data:  $v, i$ 
for  $u \in \{\mathcal{N}(v, j) \mid j \in \kappa_i\}$  do ; // For all neighbors of  $v$  in dimension  $i$ 
    if  $\sigma(u) \cap \kappa_i = \emptyset$  then ; //  $u$  is sink
        if  $v_i < u_i$  then
            return true;
        else
            return false;

```

Otherwise there exists no unique fixpoint in this subgrid. ϕ is not a unique sink orientation.

Example 6.2.1. Recall Example 6.1.9 on page 98.



Given the node $v = (246)$ and the direction $i = 1$, $\text{sinkIsRightOf}(v, i)$ looks at the following subgrid:



The algorithm iterates over each node and checks whether it is the sink of this subgrid. In this example the sink is (346) . It then is checked whether the given node is before the sink or not. In this case the sink is right of (146) (hence the name of the function) and $\text{sinkIsRightOf}(146, 1) = \text{true}$.

If called with $v = (247)$ and $i = 1$, $\text{sinkIsRightOf}(247, 1) = \text{false}$ since the sink in this subgrid is (147) , which is left of (247) .

sinkIsRightOf can be calculated in polynomial time in d and n . Therefore, the direction function D can be calculated in polynomial time for a given node $p \in \Gamma'$ as follows: For all $i = 1, \dots, d$

$$D_i(p) = \begin{cases} \text{Zero} & \text{if } \sigma_\phi(f^{-1}(p)) \cap \kappa_i = \emptyset \\ \text{Up} & \text{if } \sigma_\phi(f^{-1}(p)) \cap \kappa_i \neq \emptyset \text{ and } \text{sinkIsRightOf}(f^{-1}(p), i) \\ \text{Down} & \text{if } \sigma_\phi(f^{-1}(p)) \cap \kappa_i \neq \emptyset \text{ and not } \text{sinkIsRightOf}(f^{-1}(p), i). \end{cases}$$

Correctness In order to prove the correctness of this construction, it must be shown that each solution to the constructed OPDC instance matches back to the correct solution of the GRID-USO instance.

Unfortunately, the construction is not correct. Though the correctness can be proven for solutions of type $(O1)$, $(OV1)$ and $(OV3)$, the reduction has a major flaw for solutions of type $(OV2)$. For interested readers, the prove for the other solution types is given in Appendix B on page 138.

A solution of type $(OV2)$ If there is a violation of type $(OV2)$, there is an i -Slice \mathbf{s} and two points $p, q \in \Gamma_{\mathbf{s}}$ such that

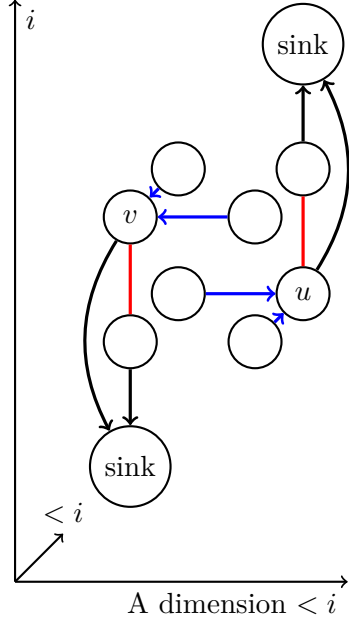
- $\forall j < i : D_j(p) = D_j(q) = \text{Zero}$
- $p_i = q_i + 1$
- $D_i(p) = \text{Down}$ and $D_i(q) = \text{Up}$

We will look at the subsets of the first $(1, \dots, i-1)$, (i) and the last $(i+1, \dots, d)$ elements separately. Let $v := f^{-1}(p)$ and $u := f^{-1}(q)$.

Since p and q are on the same i -Slice, their last $d-i$ elements are equal. This also means that the last $d-i$ elements of v and u are equal and therefore $v_{i+1,d} = u_{i+1,d}$. Hence $(v_{i+1,d} \oplus u_{i+1,d}) = \emptyset$. Therefore, if $(v \oplus u) \cap (\sigma_\phi(v) \oplus \sigma_\phi(u)) \neq \emptyset$, it must be due to the first $1, \dots, i$ elements of v, u and their orientation functions.

Now let's take a closer look at the first 1 to $i-1$ elements. Because of the first condition of the $(OV2)$ violation, it holds that $\forall j < i : \sigma_\phi(v) \cap \kappa_j = \sigma_\phi(u) \cap \kappa_j = \emptyset$ which implies that $\sigma_\phi(v_{1,i-1}) = \sigma_\phi(u_{1,i-1}) = \emptyset$.

So for the whole condition to be nonempty, dimension i must make the difference.



This diagram shows the current situation. We know that for all dimensions smaller than i , v and u are sinks. Here this is indicated by the blue edges. We also know that $\text{sinkIsRightOf}(v, i) = \text{true}$ (which is equivalent to $D_i(v) = \text{Down}$) and $\text{sinkIsRightOf}(u, i) = \text{false}$ (which is equivalent to $D_i(u) = \text{Up}$). This does not mean though that their direct neighbor in dimension i is the sink. So we know nothing about the edges that are colored in red.

In CUBE-USO this construction works, because there are only two nodes in each dimension and there the value of sinkIsRightOf tells us something about the orientation of the red edges. In the grid case it does not.

We know that $p_i = q_i + 1$, and therefore $v_i > u_i$ from which follows that $(\{v_i\} \oplus \{u_i\}) = \{v_i, u_i\}$ and therefore $\{v_i, u_i\} \subseteq (\{v\} \oplus \{u\}) \neq \emptyset$.

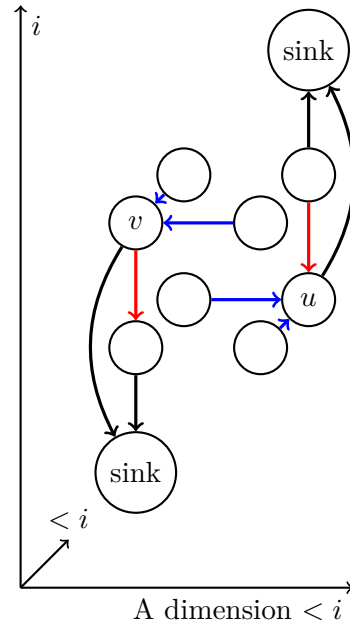
There are three cases that we need to look at, which are the possible directions of the red edges:

- 1) The edges point towards v and u : i.e. $u_i \in (\sigma_\phi(v) \cap \kappa_i)$ and $v_i \in (\sigma_\phi(u) \cap \kappa_i)$. This implies that there is a subcube with two sinks and therefore a violation of type $(GUV3)$ of that subcube and by extension of the whole instance.
- 2) The edges point away from v and u : i.e. $u_i \notin (\sigma_\phi(v) \cap \kappa_i)$ and $v_i \notin (\sigma_\phi(u) \cap \kappa_i)$. This means that in that subcube formed by v and u , the orientation function is not a bijection. It holds that $(v_{1,i} \oplus u_{1,i}) \cap (\sigma(v_{1,i}) \oplus \sigma(u_{1,i})) = \{v_i, u_i\} \cap \emptyset = \emptyset$. We found a solution of type $(GUV2)$.
- 3) One edge points away from v but the other one points towards u . The case works symmetrically for the one edge pointing towards v and the other one pointing away from u . Without loss of generality, $u_i \in (\sigma_\phi(v) \cap \kappa_i)$ and $v_i \notin (\sigma_\phi(u) \cap \kappa_i)$.

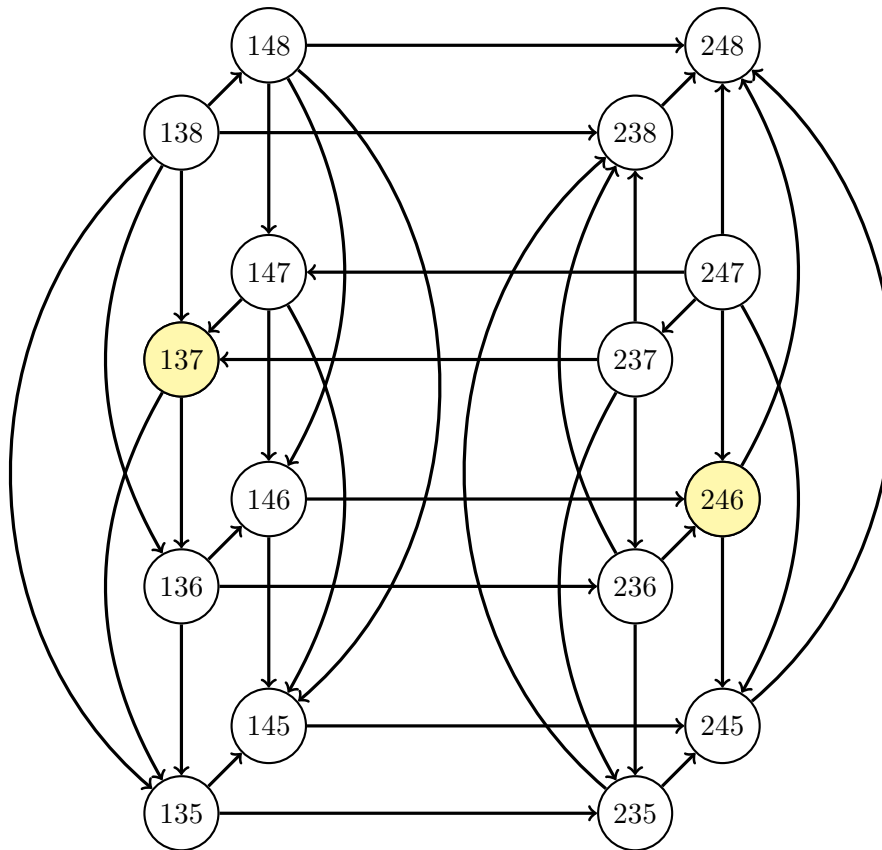
This is not a $(GUV2)$ violation, since $(v_{1,i} \oplus u_{1,i}) \cap (\sigma(v_{1,i}) \oplus \sigma(u_{1,i})) = \{v_i, u_i\} \cap \{u_i\} \neq \emptyset$. Furthermore, it is not necessarily a violation of type $(GUV3)$.

This case does not lead to a violation of GRID-USO. It might happen that this is a valid GRID-USO instance that is recognized as an OPDC violation.

Therefore, this reduction does not work.



Example 6.2.2. The following is an example of a valid GRID-USO instance. Its unique sink is the node (248). When using the construction to build an OPDC instance from this GRID-USO instance, the nodes (137) and (246) colored in yellow form a $(OV2)$ violation.



Conclusion This construction is not a valid reduction.

The general problem of this construction is that knowing where the sink of a one dimensional subgrid is has no influence on where the fixpoint of the higher dimensions is.

6.3. Grid-USO to Unique Forward EOPL

Theorem 6.3.1. *There exists a promise preserving polynomial time reduction from GRID-USO to UNIQUE FORWARD EOPL, and thus GRID-USO is in UniqueEOPL.*

Proof. Given an instance $I = (\Gamma, \phi)$ of GRID-USO as defined in Definition 6.1.23 on page 103 with $N = \{1, \dots, n\}$ being a well-ordered set and a partition of that set $K = (\kappa_1, \dots, \kappa_d)$, construct a UNIQUE FORWARD EOPL instance $I' = (S, c)$.

For a formal definition of UNIQUE FORWARD EOPL, see Definition 4.1.15 on page 30.

The idea for this reduction is based on the reduction from OPDC to UNIQUE FORWARD EOPL. [GWJR+08, Section 3.1] presented a line following algorithm to solve GRID-USO (called the *Product* algorithm) but here a slightly different algorithm is used. It is an algorithm that follows a line of nodes of the GRID-USO instance until it reaches the unique sink. In the OPDC case we look at i dimensions and find the unique sink before looking at $i + 1$ dimensions. Here, we look at i *directions* and find the sink of these before we search for the unique sink of $i + 1$ directions.

The line following algorithm works with the fact, that if we have the unique sink of some subgrid and add the next direction (and therefore another subgrid), the unique sink of the overall grid is either the sink of the first subgrid or the added subgrid. The states of this local search line following algorithm are represented as vertices of UNIQUE FORWARD EOPL.

First, let's reassign the elements in the blocks so that they are strictly increasing by 1, i.e. $\kappa_1 = \{1, 2, \dots, a\}, \kappa_2 = \{a+1, \dots, b\}$ and so on. We can simply rename the elements, the grid afterwards is the same. This can be done with any grid in polynomial time. For the rest of the proof we assume that the grid has this structure.

Definition 6.3.2 (subgrid γ). Let γ be a function that gets an integer i and a set $M \subseteq N$ and returns the following subgrid: $\gamma(i, M) := (N', K')$ where

$$N' := \bigcup_{j=1, \dots, d} \begin{cases} \{(\kappa_j)_1\} & \text{if } M \cap \kappa_j = \emptyset \wedge \kappa_j \cap \{1, \dots, i\} = \emptyset \\ \kappa_j \cap \{1, \dots, i\} & \text{if } M \cap \kappa_j = \emptyset \wedge \kappa_j \cap \{1, \dots, i\} \neq \emptyset \\ M \cap \kappa_j & \text{if } M \cap \kappa_j \neq \emptyset \end{cases}$$

and $K' := \{\kappa'_1, \dots, \kappa'_d\}$ with $\kappa'_j := \kappa_j \cap N'$ for all $j = 1, \dots, d$.

In the following we will use $\gamma(i, M)$ equivalently with N' .

Intuitively, if for $\gamma(i, M)$ M contains only one element j of block κ_k , then the directions $\{1, \dots, i\} \setminus \kappa_k$ are free to vary. For dimension k we fix it to the value j . All higher dimensions are fixed to the first value of their corresponding block.

If M contains more elements, then the directions $\{1, \dots, i\}$ that are not contained in the fixed blocks are still free to vary. But several dimensions are fixed to a certain value, namely to the values given by M . Again, all higher dimensions are fixed to the first value of their corresponding block.

Example 6.3.3. Here are a few examples of subgrids for the grid Γ with $N = \{1, \dots, 7\}$, $\kappa_1 = \{1, 2\}$, $\kappa_2 = \{3, 4\}$ and $\kappa_3 = \{5, 6, 7\}$.

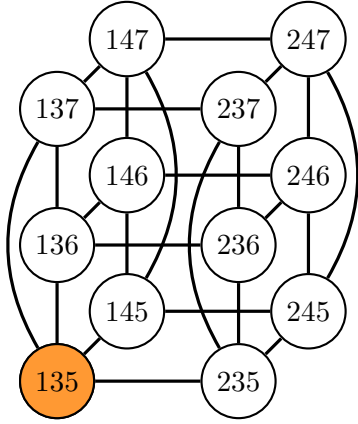


Figure 6.1.: $\gamma(1, \emptyset)$

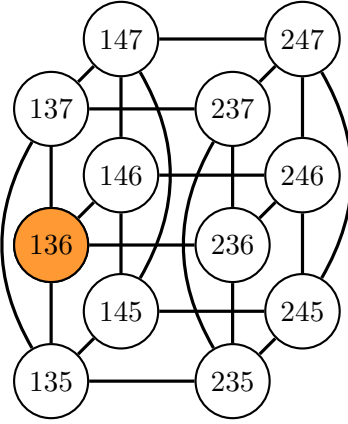


Figure 6.2.: $\gamma(1, \{6\})$

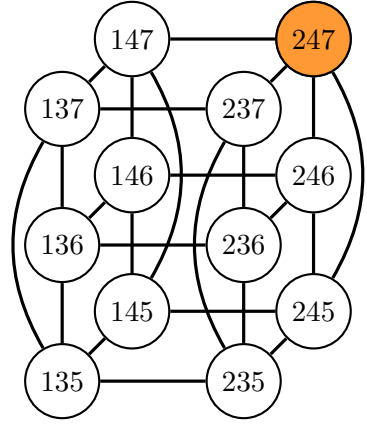


Figure 6.3.: $\gamma(i, \{2, 4, 7\})$

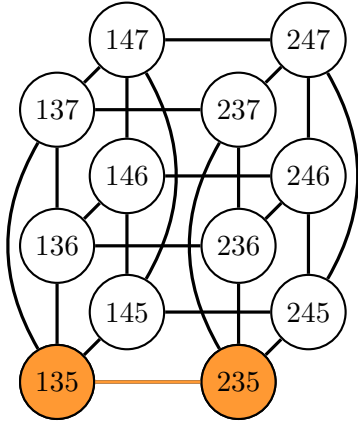


Figure 6.4.: $\gamma(2, \emptyset)$

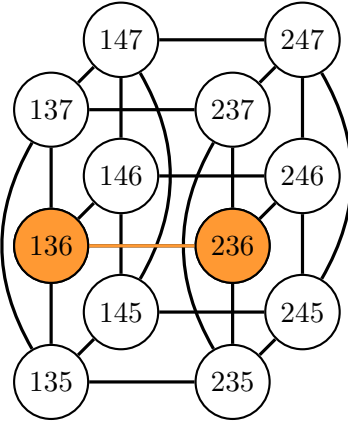


Figure 6.5.: $\gamma(2, \{6\})$

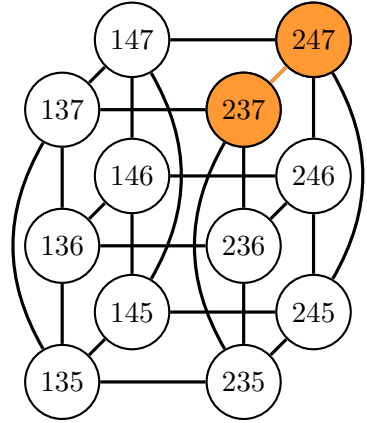


Figure 6.6.: $\gamma(4, \{2, 3, 4, 7\})$

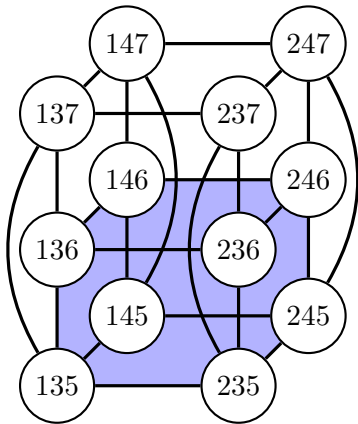


Figure 6.7.: $\gamma(6, \emptyset)$

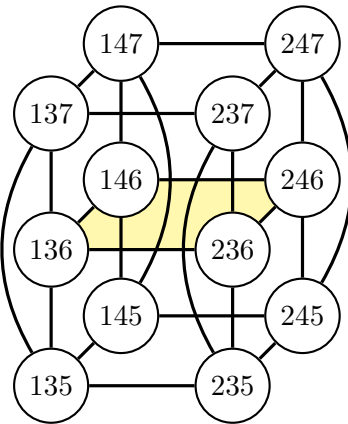


Figure 6.8.: $\gamma(6, \{6\})$

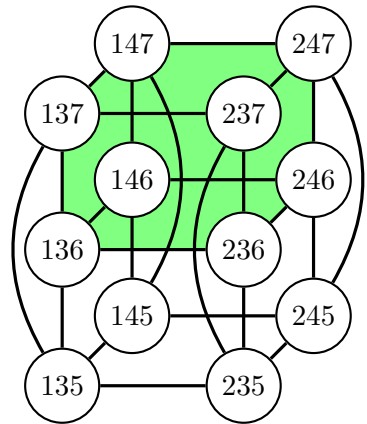


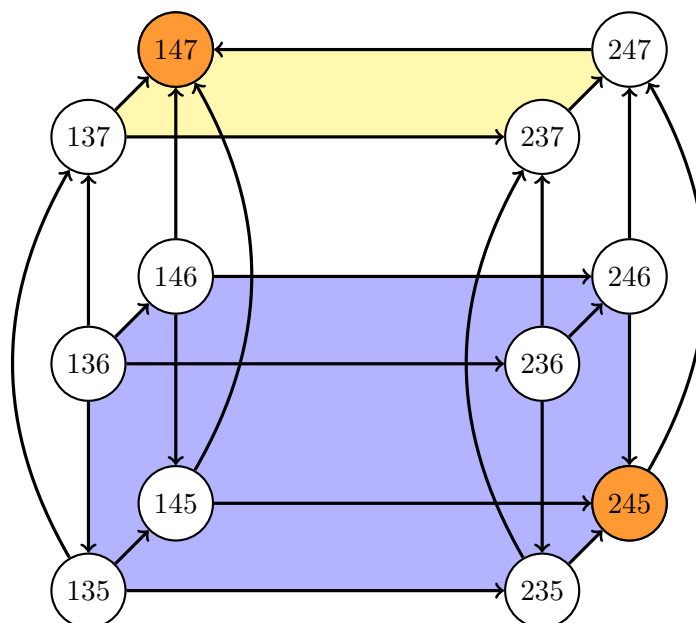
Figure 6.9.: $\gamma(7, \{6, 7\})$

Whenever M is empty, the subgrid starts at the bottom left corner and expands from there depending in i . If M is not empty, the subgrid only consists of nodes which contain the fixed elements.

Later on, we will only inspect subgrids whose set M contains at most one element of each block κ_i for each $i = 1, \dots, d$. This excludes Figure 6.6 on the preceding page and Figure 6.9 on the previous page.

Example 6.3.4. The following is an example of a valid GRID-USO instance. The blue colored area is the subgrid $\gamma(6, \emptyset)$. The yellow colored area is the subgrid $\gamma(7, \{7\})$

Since it is a valid unique sink orientation, both subgrids have a unique sink, (245) and (147), colored in orange. The unique sink of the subgrid combined by both subgrids is either (245) or (147). In this case, it is (147). None of the other nodes can be the unique sink.



Lemma 6.3.5. If σ is the outmap of a unique sink orientation, the unique sink of a subgrid $\gamma(i, M)$ with $|M \cap \kappa_j| \leq 1$ for all $j = 1, \dots, d$ is:

- (i) If $M \cap \kappa_k \neq \emptyset$ with $i \in \kappa_k$, the unique sink of subgrid $\gamma(i, M)$ is equal to the unique sink of the subgrid $\gamma(i - 1, M)$.
- (ii) If $M \cap \kappa_k = \emptyset$ the unique sink of subgrid $\gamma(i, M)$ is either equal to the unique sink of the subgrid $\gamma(i - 1, M)$ or $\gamma(i, M \cup \{i\})$.

The first condition ensures that we look at one direction after another, like the line-following algorithm intends. If $M \cap \kappa_k \neq \emptyset$, this means that we are in some kind of yellow subgrid, one where one direction of block k is fixed. Even though we could technically fix a subgrid like $\gamma(5, \{6, 7\})$ (which is the top cube in Example 6.3.4), this is omitted by this condition.

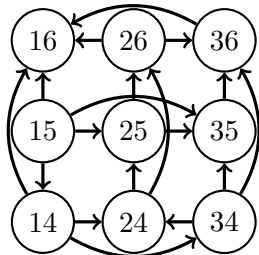
The second condition describes that if σ is a unique sink orientation, either the unique sink of the blue subgrid or the unique sink of the yellow subgrid is the unique sink of the overall subgrid.

When a subgrid only contains one node, this node is the sink.

Proof. Intuitively, the idea of the proof is that either one of the two sinks of the subgrids is the the sink of the overall grid or we can prove a (GUV_4) violation. It will be proven via induction.

Base: $i = 2$

M can contain an arbitrary number of elements but at most one of each block, specifying the subgrid we are in. Since $i = 2$, we know that $i \in \kappa_1$, because each partition must contain at most 2 elements.



If $M \cap \kappa_1 \neq \emptyset$ (case (i)), one dimension of the first block is fixed and the subgrid contains only one node. In the image on the left for example, for $i' = 1, 2, 3$, the subgrid $M(i', \{3\})$ contains only the node (34). It is clearly the unique sink.

For $M \cap \kappa_1 = \emptyset$ (case (ii)), there are the following cases.

The overall grid $\gamma(2, M)$ contains two nodes, one that contains direction "1" at the first position and one that contains a "2" at the first position, for example (1...) and (2...).

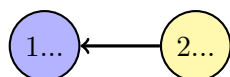
The blue subgrid $\gamma(1, M)$ consists of one node that contains direction "1".

The yellow subgrid $\gamma(2, M \cup \{2\})$ consists of one node that contains direction "2".

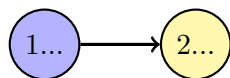
Since both the yellow and the blue subgrid contain only one node, they both are the unique sink of their respective subgrid.

If $M = \emptyset$, the nodes actually are the bottom left node and its neighbor in direction "2". If $M \neq \emptyset$, then some higher dimensions are fixed by the elements in M . Later on, this means we are recursively searching for a unique sink in a subgrid which has some dimensions fixed by M .

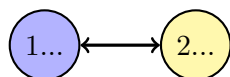
There are four cases that can occur:



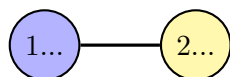
If the edge points towards (1...), (1...) is the unique sink of the overall grid.



If the edge points towards (2...), (2...) is the unique sink of the overall grid.



If both nodes have an outgoing edge, both of them have refined index of the subgrid $r_{\sigma'}(1...) = r_{\sigma'}(2...) = (1)$. Therefore we found a violation of type (GUV_4) .



If both nodes have no outgoing edge and therefore claim to be the sink, both of them have refined index of the subgrid $r_{\sigma'}(1...) = r_{\sigma'}(2...) = (0)$. Therefore we found a violation of type (GUV_4) .

Assumption: Let q is the unique sink of $\gamma(i-1, M)$ (the blue subgrid). Let p is the unique sink of $\gamma(i, M \cup \{i\})$ (the yellow subgrid), provided that $M \cap \kappa_k = \emptyset$ with $i \in \kappa_k$, since otherwise this subgrid violates the assumption that M must only contain at most one element of each block. Then no other node in the respective subgrids can be a sink, since this would be a refined index violation.

Induction: Let $(N, K) = \gamma(i, M)$ (the overall grid), $(N', K') = \gamma(i-1, M)$ (the blue subgrid) and $(N'', K'') = \gamma(i, M \cup \{i\})$ (the yellow subgrid).

First assume $M \cap \kappa_k \neq \emptyset$ for $i \in \kappa_k$.

This means that both $\gamma(i, M)$ and $\gamma(i-1, M)$ are fixed in some directions, namely the directions $M \cap \kappa_k$. We are recursively searching for a unique sink in a yellow subgrid. This is like imagining that the example above is itself the yellow subgrid of an even bigger grid.

In this case, the first two subgrids are equal and $N = N'$: For all $j < k$, $\kappa_j \cap \{1, \dots, i\} = \kappa_j \cap \{1, \dots, i-1\}$ because the only element in which they differ is i and we already know that i is in no κ_j , since it already is in κ_k . For all $j > k$, N and N' either both contain $(\kappa_j)_1$ or both contain $\kappa_j \cap M$. For κ_k , N and N' both contain exactly $\kappa_k \cap M$.

Therefore, their unique sink is also the same.

We don't have to consider $\gamma(i, M \cup \{i\})$, because M is only allowed to contain at most one element of κ_k , which already does.

Second, assume it holds that $M \cap \kappa_k = \emptyset$ for $i \in \kappa_k$.

This means that for both $\gamma(i, M)$ and $\gamma(i-1, M)$ there is no direction fixed in dimension k .

Let p be the unique sink of the blue subgrid $\gamma(i-1, M)$. We know that $\sigma(p) \cap N' = \emptyset$. Analogously, let q be the unique sink of the yellow subgrid $\gamma(i, M \cup \{i\})$ with $\sigma(q) \cap N'' = \emptyset$.

N' and N'' differ only in the elements of κ_k , where $i \in \kappa_k$: $(N' \oplus N'') \subseteq \kappa_k$.

The union of $\gamma(i-1, M)$ and $\gamma(i, M \cup \{i\})$ equals the grid $\gamma(i, M)$: Since $M \cap \kappa_k = \emptyset$, N contains all elements $\{x \mid x \in \kappa_k, x \leq i\} \cup \{(\kappa_k)_1\}$. N' contains all elements $\{x \mid x \in \kappa_k, x \leq i-1\} \cup \{(\kappa_k)_1\}$. N'' contains exactly one element of κ_k , namely i .

Therefore, $N' \cup N'' = N$. This means that $\gamma(i-1, M)$ and $\gamma(i, M \cup \{i\})$ together actually result in the subgrid $\gamma(i, M)$.

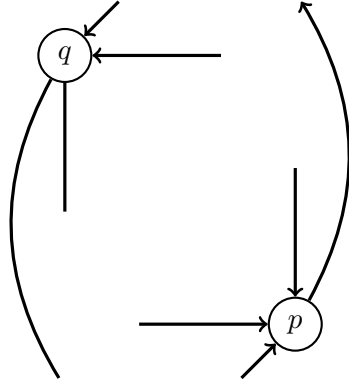
By induction assumption, there is only one node with 0 outgoing edges in $\gamma(i-1, M)$ as well as in $\gamma(i, M \cup \{i\})$. None of the other points in either subgrid can have 0 outgoing edges for all N' (and N'' respectively) directions. Therefore, none of the other points can be the unique sink of $\gamma(i, M)$.

If $\sigma(p) \cap N = \emptyset$ and $\sigma(q) \cap N \neq \emptyset$, p is the overall sink.

If $\sigma(p) \cap N \neq \emptyset$ and $\sigma(q) \cap N = \emptyset$, q is the overall sink.

If $\sigma(p) \cap N = \emptyset$ and $\sigma(q) \cap N = \emptyset$, this is a proof for two sinks in the grid, which is a refined index violation: $r_\sigma(p) = (0, \dots, 0) = r_\sigma(q)$.

If $\sigma(p) \cap N \neq \emptyset$ and $\sigma(q) \cap N \neq \emptyset$, none of them is the sink of $\gamma(i, M)$.

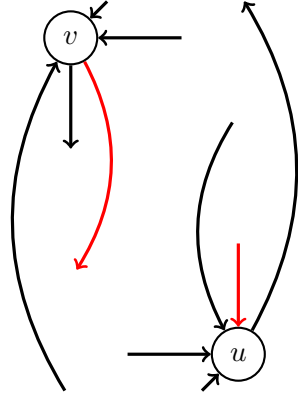


The refined index of p must be $r_\sigma(p) = (0, \dots, 0, 1)$. It is the sink of $i - 1$ directions, therefore it has 0 outgoing edges for all other directions, but since it is not the overall sink, it must have one outgoing edge in direction i .

The refined index of q is 0 for all dimensions except the last one: $r_\sigma(q) = (0, \dots, 0, x)$

If $x = 0$, q would be the unique sink.

If $x = 1$, there is clearly a refined index violation of type (GUV_4) since $r_\sigma(p) = (0, \dots, 0, 1) = r_\sigma(q)$.



If $x > 1$, q has some outgoing edges. In this case we can construct a subgrid which has a refined index violation of type (GUV_4) , by removing some of the middle directions. Whenever we remove a direction $(\kappa_k) < l < i$ with $l \in \sigma(q)$, we reduce x of the refined index of q by 1, while the refined index of p stays the same.

We can remove middle directions (for example the ones colored in red) until we reach a subgrid in which the refined index of q is the same as the refined index of p , i.e. $r_{\sigma'}(q) = (0, \dots, 0, 1)$. Therefore we found a violation of type (GUV_4) .

In conclusion, either p or q is the unique sink, or there is a violation of type (GUV_4) .

□

This Lemma can be applied recursively on any grid whose unique sink shall be found. For an instance of GRID-USO the grid whose sink we want to calculate is $\gamma(n, \emptyset)$. The unique sink of $\gamma(n, \emptyset)$ is either the unique sink of the subgrid $\gamma(n, \{n\})$ or $\gamma(n - 1, \emptyset)$. The unique sink of these subgrids can be calculated by the same rule until a subgrid $\gamma(1, M)$ is reached, for which the unique sink is trivially the one node it contains.

It can be done in a bottom up manner as well: We can start with the subgrid $\gamma(1, \emptyset)$ and use the Lemma to calculate the unique sink of $\gamma(2, \emptyset)$ and so on until we reach $\gamma(n, \emptyset)$.

This results in a line-following algorithm: For a given $\gamma(i - 1, M)$ with $i > 1$ and its unique sink p :

- (i) check if p is also the unique sink of $\gamma(i, M)$ and continue with p and $\gamma(i, M)$,

(ii) otherwise start calculating the unique sink of $\gamma(1, M \cup \{i\})$ and continue to increase the directions there since that will eventually lead to the unique sink of $\gamma(i, M \cup \{i\})$. This is then, by Lemma 6.3.5 on page 110, also the unique sink of $\gamma(i, M)$. M cannot contain any other element of κ_k with $i \in \kappa_k$, since then by the first case of Lemma 6.3.5 on page 110 p would already have been the sink of $\gamma(i, M)$.

Example 6.3.6. Recall Example 6.3.4 on page 110.

To calculate the unique sink of the grid $\gamma(7, \emptyset)$, we start with the subgrid $\gamma(1, \emptyset) = \{1, 3, 5\}$, which contains only the node (135). This node is trivially the unique sink of that subgrid.

1. Now step (i) checks, whether (135) is also the unique sink of $\gamma(2, \emptyset) = \{1, 2, 3, 5\}$. It is not, because $\sigma(135) \cap \{1, 2, 3, 5\} = \{2\}$.
- 1.ii. Therefore, step (ii) looks at $\gamma(1, \{2\}) = \{2, 3, 5\}$ which again contains only one node: (235).
 - 1.ii.1. Now, step (i) is executed again: $\gamma(2, \{2\}) = \{2, 3, 5\}$ and still (235) is the unique sink of that subgrid.
2. It is also the unique sink of : $\gamma(2, \emptyset) = \{1, 2, 3, 5\}$.
3. Step (i) is executed again: $\gamma(3, \emptyset) = \{1, 2, 3, 5\}$ and still (235) is the unique sink of that subgrid.
4. Step (i) is executed again: $\gamma(4, \emptyset) = \{1, 2, 3, 4, 5\}$ but now $\sigma(235) \cap \{1, 2, 3, 4, 5\} = \{4\}$.
- 4.ii. Therefore, step (ii) looks at $\gamma(1, \{4\}) = \{1, 4, 5\}$ which again contains only one node: (145).
 - And so on ...

Note that $\gamma(i, M)$ and $\gamma(i-1, M)$ might describe the same set. This happens for example when $(i-1) \in \kappa_k$ and i is the first element in κ_{k+1} . Lemma 6.3.5 on page 110 still applies though. It adds an unnecessary step to the algorithm (as can be seen in step 2 and 3 of Example 6.3.6), but it simplifies the reduction since we don't have to make distinctions of $(i-1)$ and i being in the same or in different blocks. We can treat both cases equally.

This line following algorithm is the basis of our reduction. The idea of this algorithm is used to implement the successor function.

The vertices For a given grid $\Gamma = (N, K, V', E')$, let the vertices V of the UNIQUE FORWARD EOPL instance be $V := (V' \cup \{\perp\})^{n+1}$.

A node can be viewed as vector of either points in V' or \perp : $v = (p_1, \dots, p_n, p_{n+1})$, with each $p_i \in V' \cup \{\perp\}$. Whenever a point $p_i \neq \perp$, this shall encode that p_i is a unique sink of $\gamma(i-1, M)$, where M is determined by the indices of v bigger than i that are not \perp : $M(v, i) := \{j \mid j > i \wedge p_j \neq \perp\}$.

The idea is that each node represents a step of the line following algorithm which contains all unique sinks of subgrids that were already visited.

Only some of the vectors are valid vertices. Formally, a vertex $v = (p_1, \dots, p_n, p_{n+1})$ is valid if and only if it fulfills all of the following conditions:

- for $i = 1, \dots, n+1$:
 - (a) If $i = 1$: $p_i \neq \perp \Rightarrow \sigma(p_i) \cap \{1\} = \emptyset$,
for $i > 1$: $p_i \neq \perp \Rightarrow \sigma(p_i) \cap \{1, \dots, i-1\} = \emptyset$
- for $i = 1, \dots, n$:
 - (b) Let p_l be the first point in v that is not \perp .
 $p_i \neq \perp \wedge i > l \Rightarrow \sigma(p_i) \cap \{1, \dots, i\} \neq \emptyset$.
 - (c) $p_i \neq \perp \wedge p_j = \perp \wedge i < j \Rightarrow j \notin (p_i)$ or j is the first of the block that contains j , i.e. $(p_i)_k = (\kappa_k)_1$ with $j \in \kappa_k$.
 - (d) $p_i \neq \perp \wedge p_j \neq \perp \wedge i < j \Rightarrow j \in (p_i)$, i.e. $(p_i)_k = j$ with $j \in \kappa_k$.
 - (e) $p_i \neq \perp \wedge p_j \neq \perp \wedge i < j \Rightarrow p_i$ is from right to left lexicographically bigger than p_j .

Condition (a) ensures that each point at a position i is a sink for all directions smaller than i .

Recall Example 6.3.4 on page 110. The node $(\perp, \perp, 135, \perp, \dots, \perp)$ is not a valid node, because (135) is not the unique sink of the subgrid $\gamma(2, \emptyset) = \{1, 2, 3, 5\}$. $\sigma(135) \cap \{1, 2\} = \{2\}$. $(\perp, \perp, 235, \perp, \dots, \perp)$ on the other hand is a valid vertex, since (235) is the sink of that subgrid.

Condition (b) ensures that we push a point through as long as it is the unique sink for the next higher dimension as well, i.e. it ensures that rule (i) of the line following algorithm is always applied if possible. We only start searching for the other sink in a new subgrid, if p_i is not the unique sink of $\gamma(i, M(v, i))$, even though it is the unique sink of $\gamma(i-1, M(v, i))$.

Condition (c) ensures two things. First, it ensures that for each i the blocks whose value is not fixed by $M(v, i)$, $(p_i)_k$ is set to the first value of the respective block k . Second, it ensures that if a value for a block is fixed, none of the other values of the block which are not fixed by $M(v, i)$ appear in p_i .

Recall Example 6.3.4 on page 110. The node $(235, \perp, \dots, \perp)$ is not a valid node. All other points of the node are \perp , therefore p_1 should only contain the first elements of each block, but 2 is not the first element of any block. The node $(135, \perp, \dots, \perp)$ on the other hand is a valid vertex.

Condition (d) ensures that for each i if the value of a certain block is fixed by $M(v, i)$, this value is actually contained in the point p_i .

Recall Example 6.3.4 on page 110. The node $(135, 235, \perp, \dots, \perp)$ is not a valid node. For $i = 1$, $M((135, 235, \perp, \dots, \perp), 1) = \{2\}$ because $p_2 = (235) \neq \perp$, but (135) does not contain the 2. The node $(235, 135, \perp, \dots, \perp)$ on the other hand is a valid vertex.

Condition (e) ensures the property stated by Lemma 6.3.5 on page 110. This will be explained in more detail later on when it is described what happens if this condition is violated.

The function $\text{isVertex} : V \Rightarrow \{true, false\}$ returns whether v fulfills all these conditions or not. The function can be calculated in polynomial time.

The start vertex o is defined as $o := (((\kappa_1)_1, \dots, (\kappa_d)_1), \perp, \dots, \perp)$.

Note that V must be encoded by 0-1-bit strings in order to actually be a valid set of vertices for the UNIQUE FORWARD EOPL instance. For convenience and the sake of better understanding we will continue to use V as defined above in the reduction.

Construction of the successor function S Let $S: V \rightarrow V$. Given a vertex $v = (p_1, \dots, p_n, p_{n+1})$, let $S(v)$ be constructed as follows:

- (1) If $\text{isVertex}(v) = \text{false}$ then $S(v) := v$ to indicate that v is not a valid vertex.
- (2) If $\text{isVertex}(v) = \text{true}$, let i be the smallest index such that $p_i \neq \perp$. This means we are in a subgrid $\gamma(i-1, M(v, i))$ with $M(v, i) := \{j \mid j > i \wedge p_j \neq \perp\}$ and p_i is its unique sink.
 - (2.1) If $i = n+1$ then $v = (\perp, \dots, \perp, p_{n+1})$. Furthermore, $\sigma(p_{n+1}) \cap \{1, \dots, n\} = \emptyset$, which means that p_{n+1} is a unique sink in all n directions. We set $S(v) := v$ to indicate that the point before v is the end of the line.
 - (2.2) If $\sigma(p_i) \cap \{1, \dots, i\} = \emptyset$ and $i < n+1$ then $S(v) := u$ where

$$u_j = \begin{cases} \perp & \text{if } j < i+1 \\ p_i & \text{if } j = i+1 \\ p_j & \text{if } j > i+1 \end{cases}$$

This operation overwrites the point at position $i+1$ with p_i and sets position i to \perp . All other components are unchanged.

$$S(\perp, \dots, \perp, p_i, p_{i+1}, p_{i+2}, \dots, p_n) = (\perp, \dots, \perp, \perp, p_i, p_{i+2}, \dots, p_n).$$

This means that p_i is also the unique sink of $\gamma(i, M(v, i))$, equivalently to case (i) of the line-following algorithm.

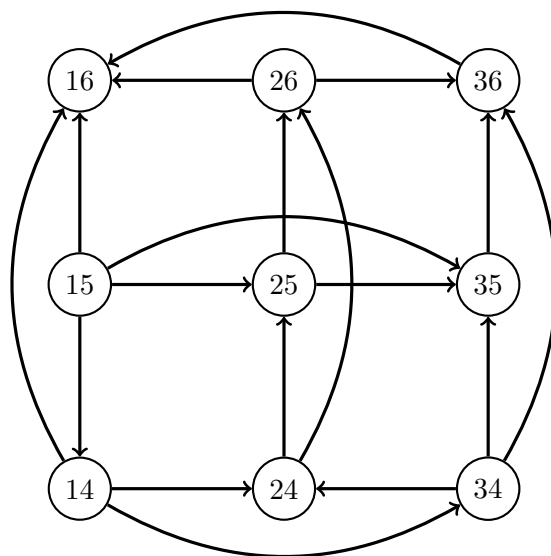
- (2.3) If $\sigma(p_i) \cap \{1, \dots, i\} \neq \emptyset$ and $i < n+1$, it holds that $\sigma(p_i) \cap \{1, \dots, i\} = \{i\}$: Since isVertex is true, we know that $\sigma(p_i) \cap \{1, \dots, i-1\} = \emptyset$. So if the intersection is nonempty, then it must contain i .

Let $S(v) := (q, p_2, \dots, p_{n+1})$ with

$$(q)_j := \begin{cases} (\kappa_j)_1 & \text{if } i \in \kappa_k \text{ and } j < k \\ i & \text{if } i \in \kappa_j \\ (p_i)_j & \text{if } i \in \kappa_k \text{ and } j > k. \end{cases}$$

This case is equivalent to step (ii) of the line following algorithm. We continue to search in $\gamma(1, M(v, i) \cup \{i\}) = \gamma(1, M(v, 1) \cup \{i\})$ in order to find the unique sink of $\gamma(i, M(v, i) \cup \{i\})$. i is automatically added to $M(v, i)$ by p_i being at position i . Condition (c) and (d) of isVertex ensure that i is contained in all points below index i .

Example 6.3.7. In this example $n = 6$ and therefore $N = \{1, \dots, 6\}$. There are $d = 2$ blocks $\kappa_1 = \{1, 2, 3\}$ and $\kappa_2 = \{4, 5, 6\}$. The outmap is given by the arrows in the figure below. It is a unique sink orientation. The unique sink is (16).



The vertices for this example have 7 entries. The start node is $(14, \perp, \perp, \perp, \perp, \perp, \perp)$.

Vertex v	i	$\sigma(p_i) \cap \{1, \dots, i\}$	Case	$S(v)$
$((14), \perp, \perp, \perp, \perp, \perp, \perp)$	1	$\sigma((14)) \cap \{1, \dots, 1\} = ()$	2.2	$(\perp, (14), \perp, \perp, \perp, \perp, \perp)$
$(\perp, (14), \perp, \perp, \perp, \perp, \perp)$	2	$\sigma((14)) \cap \{1, \dots, 2\} = (2)$	2.3	$((24), (14), \perp, \perp, \perp, \perp, \perp)$
$((24), (14), \perp, \perp, \perp, \perp, \perp)$	1	$\sigma((24)) \cap \{1, \dots, 1\} = ()$	2.2	$(\perp, (24), \perp, \perp, \perp, \perp, \perp)$
$(\perp, (24), \perp, \perp, \perp, \perp, \perp)$	2	$\sigma((24)) \cap \{1, \dots, 2\} = ()$	2.2	$(\perp, \perp, (24), \perp, \perp, \perp, \perp)$
$(\perp, \perp, (24), \perp, \perp, \perp, \perp)$	3	$\sigma((24)) \cap \{1, \dots, 3\} = ()$	2.2	$(\perp, \perp, \perp, (24), \perp, \perp, \perp)$
$(\perp, \perp, \perp, (24), \perp, \perp, \perp)$	4	$\sigma((24)) \cap \{1, \dots, 4\} = ()$	2.2	$(\perp, \perp, \perp, \perp, (24), \perp, \perp)$
$(\perp, \perp, \perp, \perp, (24), \perp, \perp)$	5	$\sigma((24)) \cap \{1, \dots, 5\} = (5)$	2.3	$((15), \perp, \perp, \perp, (24), \perp, \perp)$
$((15), \perp, \perp, \perp, (24), \perp, \perp)$	1	$\sigma((15)) \cap \{1, \dots, 1\} = ()$	2.2	$(\perp, (15), \perp, \perp, (24), \perp, \perp)$
$(\perp, (15), \perp, \perp, (24), \perp, \perp)$	2	$\sigma((15)) \cap \{1, \dots, 2\} = (2)$	2.3	$((25), (15), \perp, \perp, (24), \perp, \perp)$
$((25), (15), \perp, \perp, (24), \perp, \perp)$	1	$\sigma((25)) \cap \{1, \dots, 1\} = ()$	2.2	$(\perp, (25), \perp, \perp, (24), \perp, \perp)$
$(\perp, (25), \perp, \perp, (24), \perp, \perp)$	2	$\sigma((25)) \cap \{1, \dots, 2\} = ()$	2.2	$(\perp, \perp, (25), \perp, (24), \perp, \perp)$
$(\perp, \perp, (25), \perp, (24), \perp, \perp)$	3	$\sigma((25)) \cap \{1, \dots, 3\} = (3)$	2.3	$((35), \perp, (25), \perp, (24), \perp, \perp)$
$((35), \perp, (25), \perp, (24), \perp, \perp)$	1	$\sigma((35)) \cap \{1, \dots, 1\} = ()$	2.2	$(\perp, (35), (25), \perp, (24), \perp, \perp)$
$(\perp, (35), (25), \perp, (24), \perp, \perp)$	2	$\sigma((35)) \cap \{1, \dots, 2\} = ()$	2.2	$(\perp, \perp, (35), \perp, (24), \perp, \perp)$
$(\perp, \perp, (35), \perp, (24), \perp, \perp)$	3	$\sigma((35)) \cap \{1, \dots, 3\} = ()$	2.2	$(\perp, \perp, \perp, (35), (24), \perp, \perp)$
$(\perp, \perp, \perp, (35), (24), \perp, \perp)$	4	$\sigma((35)) \cap \{1, \dots, 4\} = ()$	2.2	$(\perp, \perp, \perp, \perp, (35), \perp, \perp)$
$(\perp, \perp, \perp, \perp, (35), \perp, \perp)$	5	$\sigma((35)) \cap \{1, \dots, 5\} = ()$	2.2	$(\perp, \perp, \perp, \perp, \perp, (35), \perp)$
$(\perp, \perp, \perp, \perp, \perp, (35), \perp)$	6	$\sigma((35)) \cap \{1, \dots, 6\} = (6)$	2.3	$((16), \perp, \perp, \perp, \perp, (35), \perp)$
$((16), \perp, \perp, \perp, \perp, (35), \perp)$	1	$\sigma((16)) \cap \{1, \dots, 1\} = ()$	2.2	$(\perp, (16), \perp, \perp, \perp, (35), \perp)$
$(\perp, (16), \perp, \perp, \perp, (35), \perp)$	2	$\sigma((16)) \cap \{1, \dots, 2\} = ()$	2.2	$(\perp, \perp, (16), \perp, \perp, (35), \perp)$
$(\perp, \perp, (16), \perp, \perp, (35), \perp)$	3	$\sigma((16)) \cap \{1, \dots, 3\} = ()$	2.2	$(\perp, \perp, \perp, (16), \perp, (35), \perp)$
$(\perp, \perp, \perp, (16), \perp, (35), \perp)$	4	$\sigma((16)) \cap \{1, \dots, 4\} = ()$	2.2	$(\perp, \perp, \perp, \perp, (16), (35), \perp)$
$(\perp, \perp, \perp, \perp, (16), (35), \perp)$	5	$\sigma((16)) \cap \{1, \dots, 5\} = ()$	2.2	$(\perp, \perp, \perp, \perp, \perp, (16), \perp)$
$(\perp, \perp, \perp, \perp, \perp, (16), \perp)$	6	$\sigma((16)) \cap \{1, \dots, 6\} = ()$	2.2	$(\perp, \perp, \perp, \perp, \perp, \perp, (16))$
$(\perp, \perp, \perp, \perp, \perp, \perp, (16))$	7	$\sigma((16)) \cap \{1, \dots, 7\} = ()$	2.1	$(\perp, \perp, \perp, \perp, \perp, \perp, \perp, (16))$

Cost function c The requirements for the cost function are that the cost increases the further on the line we are. Furthermore it needs to be represented that two sinks of the same subgrid in an invalid GRID-USO instance result in the same cost. The nodes

of the line following algorithm result in a sequence of nodes that are lexicographically increasing from right to left. This property is used to define the cost function.

Let $\omega := n + 2$.

First, the help function $g: (V' \cup \{\perp\}) \times \{1, \dots, n\} \times \{1, \dots, d\} \rightarrow \{0, \dots, \omega - 1\}$ will be called with a point p from a vertex and the index i at which p is at the vertex, i.e. $p = p_i$. j stands for the element of p we are looking at, i.e. $(p_i)_j$. It is defined as follows:

$$g(p, i, j) := \begin{cases} 0 & \text{if } p = \perp \\ \omega - 1 & \text{if } \sigma(p) \cap \{1, \dots, i\} = \emptyset \\ p_j & \text{otherwise.} \end{cases}$$

In the first case, if $p_i = \perp$, 0 is returned which will result in p_i not being relevant in the cost later. If $\sigma(p) \cap \{1, \dots, i\} = \emptyset$, this means that p_i will be pushed through by case 2.2 in the next step. This means p_i is more valuable than any other point which is not a sink of i directions. Therefore it returns $\omega - 1$, which is the highest possible value that g can return. In particular it is bigger than any value returned by the third case, in which the value of p_i at position j is returned. The third case enforces that if given two points which are both not the sink of direction i , the lexicographically bigger point is valued more. If there are two points that both *are* the sink of all directions $\{1, \dots, i\}$, then g returns the same value for both of them, which is $(\omega - 1)$.

The function L calls g for each point of v as well as for each index of each node.

$$L: V \rightarrow \mathbb{N}^{nd}$$

$$\begin{aligned} L(v) &:= (g(p_1, 1, 1), \dots, g(p_1, 1, d), g(p_2, 2, 1), \dots, g(p_2, 2, d), \dots, g(p_n, n, 1), \dots, g(p_n, n, d)) \\ &=: (l_1, \dots, l_{nd}). \end{aligned}$$

Example 6.3.8. Recall Example 6.3.7 on the preceding page. If we are given for example the node $v = (\perp, (35), (25), \perp, (24), \perp, \perp)$ with $\sigma(35) \cap \{1, 2\} = \emptyset$, $L(v)$ is:

v	\perp	3	5	2	5	\perp	2	4	\perp	\perp
i	1	2		3		4	5		6	7
j	1	2	1	2	1	2	1	2	1	2
$g(p_i, i, j)$	0	0	$\omega - 1$	$\omega - 1$	2	5	0	0	2	4

With these two helper functions, c is defined as follows:

$$\begin{aligned} c(v) &:= \left(\sum_{i=1}^{nd} \omega^i l_i \right) + \begin{cases} 0 & \text{if } p_{n+1} = \perp \\ \omega^{nd+1} & \text{if } p_{n+1} \neq \perp \end{cases} \\ &= \left(\sum_{i=1}^n \left(\omega^{i-1} \cdot \sum_{j=1}^d \omega^j g(p_i, i, j) \right) \right) + \begin{cases} 0 & \text{if } p_{n+1} = \perp \\ \omega^{nd+1} & \text{if } p_{n+1} \neq \perp. \end{cases} \end{aligned}$$

Example 6.3.9. Recall Example 6.3.7 on the previous page. $n = 6$, $d = 2$ and $\omega = 8$. Here are the costs of the nodes:

Node v	$L(v)$	$c(v)$
$((14), \perp, \perp, \perp, \perp, \perp, \perp)$	$(77, 00, 00, 00, 00, 00, 00)$	504
$(\perp, (14), \perp, \perp, \perp, \perp, \perp)$	$(00, 14, 00, 00, 00, 00, 00)$	16896
$((24), (14), \perp, \perp, \perp, \perp, \perp)$	$(77, 14, 00, 00, 00, 00, 00)$	17400
$(\perp, (24), \perp, \perp, \perp, \perp, \perp)$	$(00, 77, 00, 00, 00, 00, 00)$	32256
$(\perp, \perp, (24), \perp, \perp, \perp, \perp)$	$(00, 00, 77, 00, 00, 00, 00)$	2064384
$(\perp, \perp, \perp, (24), \perp, \perp, \perp)$	$(00, 00, 00, 77, 00, 00, 00)$	132120576
$(\perp, \perp, \perp, \perp, (24), \perp, \perp)$	$(00, 00, 00, 00, 24, 00, 00)$	4563402752
$((15), \perp, \perp, \perp, (24), \perp, \perp)$	$(77, 00, 00, 00, 24, 00, 00)$	4563403256
$(\perp, (15), \perp, \perp, (24), \perp, \perp)$	$(00, 15, 00, 00, 24, 00, 00)$	4563423744
$((25), (15), \perp, \perp, (24), \perp, \perp)$	$(77, 15, 00, 00, 24, 00, 00)$	4563424248
$(\perp, (25), \perp, \perp, (24), \perp, \perp)$	$(00, 77, 00, 00, 24, 00, 00)$	4563435008
$(\perp, \perp, (25), \perp, (24), \perp, \perp)$	$(00, 00, 25, 00, 24, 00, 00)$	4564779008
$((35), \perp, (25), \perp, (24), \perp, \perp)$	$(77, 00, 25, 00, 24, 00, 00)$	4564779512
$(\perp, (35), (25), \perp, (24), \perp, \perp)$	$(00, 77, 25, 00, 24, 00, 00)$	4564811264
$(\perp, \perp, (35), \perp, (24), \perp, \perp)$	$(00, 00, 77, 00, 24, 00, 00)$	4565467136
$(\perp, \perp, \perp, (35), (24), \perp, \perp)$	$(00, 00, 00, 77, 24, 00, 00)$	4695523328
$(\perp, \perp, \perp, \perp, (35), \perp, \perp)$	$(00, 00, 00, 00, 77, 00, 00)$	8455716864
$(\perp, \perp, \perp, \perp, \perp, (35), \perp)$	$(00, 00, 00, 00, 00, 35, 00)$	369367187456
$((16), \perp, \perp, \perp, \perp, (35), \perp)$	$(77, 00, 00, 00, 00, 35, 00)$	369367187960
$(\perp, (16), \perp, \perp, \perp, (35), \perp)$	$(00, 77, 00, 00, 00, 35, 00)$	369367219712
$(\perp, \perp, (16), \perp, \perp, (35), \perp)$	$(00, 00, 77, 00, 00, 35, 00)$	369369251840
$(\perp, \perp, \perp, (16), \perp, (35), \perp)$	$(00, 00, 00, 77, 00, 35, 00)$	369499308032
$(\perp, \perp, \perp, \perp, (16), (35), \perp)$	$(00, 00, 00, 00, 77, 35, 00)$	377822904320
$(\perp, \perp, \perp, \perp, \perp, (16), \perp)$	$(00, 00, 00, 00, 00, 77, 00)$	541165879296
$(\perp, \perp, \perp, \perp, \perp, \perp, (16))$	$(00, 00, 00, 00, 00, 00, 77)$	34634616274944

Correctness To prove the correctness of this construction, it must be shown that each solution of the constructed UNIQUE FORWARD EOPL instance matches back to the correct solution of the GRID-USO instance.

In order to prove that the reduction is also promise preserving, it will be shown that every solution of type $(GU1)$ is only mapped to solutions of type $(UF1)$ and every violation of the created UNIQUE FORWARD EOPL instance is mapped back to a violation of the GRID-USO instance.

A solution of type $(UF1)$ a Let $v = (p_1, \dots, p_n, p_{n+1})$ be a solution of type $(UF1)$ and $u = S(v)$. This means that $S(v) \neq v$ and either

(a) $S(u) = u$ or

(b) $c(u) \leq c(v)$.

Assume it holds that $S(u) = u$.

First we consider the case that $\text{isVertex}(u) = \text{true}$.

- 2.1. In this case we have a point p_{n+1} for which it holds that $\sigma(p_{n+1}) \cap \{1, \dots, n\} = \sigma(p_{n+1}) = \emptyset$. Therefore it is a sink and a solution of type $(GU1)$.
- 2.2. In this case it cannot happen that for a given u , S returns the same u .
- 2.3. In this case it might happen that $S(u) = u$ if $i = 1$ and $\sigma(p_1) \cap \{1\} = \{1\}$. It must hold that $p_1 = q$ because otherwise $S(u)$ wouldn't be u again. This means that $1 \in p_1$, which results in a self loop in the grid and a violation of type $(GUV1)$.

Second we consider the case that $\text{isVertex}(u) = \text{false}$. In this case we have to take a closer look at how $S(v)$ was determined.

1. Since $S(v) = u$ and $v \neq u$, this rule was not used to determine $S(v)$.
- 2.1. Since $S(v) = u$ and $v \neq u$, this rule was not used to determine $S(v)$.
- 2.2. If $S(v)$ is determined by rule 2.2, let i be the smallest index of v such that $p_i \neq \perp$.

Vertex	Index						
	1	...	$i - 1$	i	$i + 1$	$i + 2$...
v	\perp	...	\perp	x	y	z	...
$S(v)$	\perp	...	\perp	\perp	x	z	...

v and $S(v)$ differ only in the two points at position i and $i + 1$. Since $\text{isVertex}(v) = \text{true}$, one of these two points of $S(v)$ must violate the conditions of isVertex . More specifically, it is x at position $i + 1$ that makes the difference because position i is set to \perp and therefore none of the conditions of isVertex are interested in it.

- (a) It holds that $\sigma(x) \cap \{1, \dots, i - 1\} = \emptyset$. Since we only enter case 2.2 if $\sigma(x) \cap \{1, \dots, i\} = \emptyset$, this condition is also satisfied for $S(v)$.
- (b) Since this condition is only relevant for all p_k that are not the first non-bottom point in v , and since we only remove one non-bottom point and no new non-bottom point is added, this condition holds trivially for $S(v)$ as well.
 - If conditions (c), (d) or (e) do not hold for $S(v)$ it follows that it could not have been true for v either. Therefore, $S(v)$ cannot break these conditions.

It follows that if $\text{isVertex}(v)$ is true, after applying rule 2.2 it holds that $\text{isVertex}(S(v))$ is true. Therefore this rule could not have been used to determine $S(v)$.

- 2.3. If $u = (q, \perp, \dots, \perp, p_i, \dots)$, $\text{isVertex}(v) = \text{true}$ and $\text{isVertex}(u) = \text{false}$, then q must violate one of the conditions of isVertex , since nothing else changed from v to u . Let i be the first index of v that is not \perp .

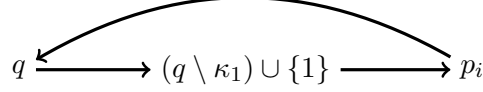
We want to show in the following for each condition of isVertex that it either always holds, or if it doesn't, that there is a violation in the GRID-USO instance.

- (a) Since $q \neq \perp$, if the first condition is violated, it holds that $\sigma(q) \cap \{1\} \neq \emptyset$, i.e. $\sigma(q) \cap \{1\} = \{1\}$.

If $1 \in q$ we found a self loop and therefore a violation of type (GUV1).

If $1 \notin q$, there must be another element $l \in \kappa_1$ with $l \in q$ instead. Furthermore, it must hold that $l = i$ and also $i \in \kappa_1$. There is a cycle in the grid: From q to $(q \setminus \kappa_1) \cup \{1\}$ to p_i back to q .

- From q to $(q \setminus \kappa_1) \cup \{1\}$: This clearly holds since $1 \in \sigma(q)$.
- From $(q \setminus \kappa_1) \cup \{1\}$ to p_i : For p_i it holds that $\sigma(p_i) \cap \{1, \dots, i - 1\} = \emptyset$, therefore p_i has an incoming edge from direction 1. Since $q_j = (p_i)_j$ for all $j > 1$, $(q \setminus \kappa_1) \cup \{1\}$ points towards p_i .
- From p_i to q : It holds that $\sigma(p_i) \cap \{1, \dots, i\} = \{i\}$ and $q_j = (p_i)_j$ for all $j > 1$. Therefore, p_i points towards q .



This results in an refined index violation (GUV_4) of the subgrid Γ' : $r_{\sigma'}(q) = (1) = r_{\sigma'}(p_i)$. So if this condition is broken, there is a violation in the GRID-USO instance.

(b) q is now the first point in $S(v)$ that is not \perp , whereas it was p_i before. So if this condition is broken now, it must be due to p_i . But we entered case 2.3 under the condition that $\sigma(p_i) \cap \{1, \dots, i\} \neq \emptyset$, therefore this condition cannot be violated and also holds for $S(v)$.

(c) Let k be the index of the block that contains i : $i \in \kappa_k$.

v has the form $(\perp, \dots, \perp, \underbrace{\perp, \dots, p_i, \dots, \perp}_{\kappa_k}, \dots)$.

$S(v)$ has the form $(q, \perp, \dots, \perp, \perp, \dots, p_i, \dots, \perp, \dots)$ with $i \in q$. We now look at each element of q individually.

For $j < k$ it holds that q_j is the first element of block κ_j because we are in the construction case 2.3. Therefore condition (c) holds for $S(v)$ for all $j < k$.

For $j > k$, q_j is set to $(p_i)_j$. Since $\text{isVertex}(v)$ is true, this condition holds for $S(v)$ for all $j > k$.

So if this condition is broken it must be due to $j = k$. We know that $q_k = i$.

Either (c) holds, or (c) and (d) are both broken.

Assume condition (d) is not broken (since otherwise this leads to a violation as described in the next case (d)). Then all indices of $S(v)$ associated with κ_k must be \perp except for p_i .

Since q already contains i , it cannot contain any other element of κ_k .

Therefore, if this condition holds for v , it also holds for $S(v)$, or there is *also* a violation of condition (d), which will be dealt with in the next case.

(d) If p_i is the only non-bottom element of the points representing κ_k in v , then condition (d) trivially also holds for $S(v)$ by the construction of 2.3.

Let l be the smallest index of v after i which is not \perp . There must exist one because otherwise this condition would not be broken.

v must have the form $(\perp, \dots, \perp, \underbrace{\perp, \dots, p_i, \dots, p_l, \dots}_{\kappa_k}, \dots)$.

$S(v)$ has the form $(q, \perp, \dots, \perp, p_i, \dots, p_l, \dots)$ with $i \in q$.

p_i is the smallest index of v which is not \perp . Let k be the index of the block that contains i : $i \in \kappa_k$. For $1 < j < k$, all elements in $S(v)_j$ are \perp . Therefore this case is irrelevant and this condition holds for all $1 < j < k$. For $j > k$, q_j is set to $(p_i)_j$. Since $\text{isVertex}(v)$ is true, this condition holds for all $j > k$.

So if this condition is broken it must be due to $j = k$ and the elements associated with κ_k .

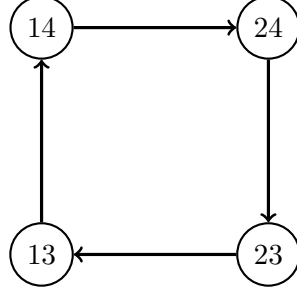
For $j = k$ and $i \in \kappa_j$ we set q_k to i . Since p_i is unchanged, it holds that $i \in q$.

There is a problem though if there exists another node p_l with $l > i$ and $l \in \kappa_k$. In this case, this condition asks for l to be in q as well, which is not possible since it contains only one element from each block and it already contains i .

We want to show that then Lemma 6.3.5 on page 110 is violated.

The most simple example of this case is a cycle:

Example 6.3.10. Consider the following example with a cycle. The relevant case is the last entry of the table. $i = 3$ and $k = 2$.



Vertex v	i	$\sigma(p_i) \cap \{1, \dots, i\}$	Case	$S(v)$
$((13), \perp, \perp, \perp, \perp)$	1	$\sigma((13)) \cap \{1, \dots, 1\} = ()$	2.2	$(\perp, (13), \perp, \perp, \perp)$
$(\perp, (13), \perp, \perp, \perp)$	2	$\sigma((13)) \cap \{1, \dots, 2\} = ()$	2.2	$(\perp, \perp, (13), \perp, \perp)$
$(\perp, \perp, (13), \perp, \perp)$	3	$\sigma((13)) \cap \{1, \dots, 3\} = ()$	2.2	$(\perp, \perp, \perp, (13), \perp)$
$(\perp, \perp, \perp, (13), \perp)$	4	$\sigma((13)) \cap \{1, \dots, 4\} = (4)$	2.3	$((14), \perp, \perp, (13), \perp)$
$((14), \perp, \perp, (13), \perp)$	1	$\sigma((14)) \cap \{1, \dots, 1\} = ()$	2.2	$(\perp, (14), \perp, (13), \perp)$
$(\perp, (14), \perp, (13), \perp)$	2	$\sigma((14)) \cap \{1, \dots, 2\} = (2)$	2.3	$((24), (14), \perp, (13), \perp)$
$((24), (14), \perp, (13), \perp)$	1	$\sigma((24)) \cap \{1, \dots, 1\} = ()$	2.2	$(\perp, (24), \perp, (13), \perp)$
$(\perp, (24), \perp, (13), \perp)$	2	$\sigma((24)) \cap \{1, \dots, 2\} = ()$	2.2	$(\perp, \perp, (24), (13), \perp)$
$(\perp, \perp, (24), (13), \perp)$	3	$\sigma((24)) \cap \{1, \dots, 3\} = (3)$	2.3	$((13), \perp, (24), (13), \perp)$
$((13), \perp, (24), (13), \perp)$	1	$\sigma((13)) \cap \{1, \dots, 1\} = ()$	1	$((13), \perp, (24), (13), \perp)$

This describes the case, where we are given the unique sink of $\gamma(l-1, M(v, l))$ (e.g. the blue area in Example 6.3.4 on page 110): p_l . It is the sink of $l-1$ directions but not of l directions, since then it would have been pushed further through. But it hasn't, there are elements unequal to \perp with a smaller index than l (i.e. p_i). Therefore by condition (b) of **isVertex**, it holds that $\sigma(p_l) \cap \{1, \dots, l\} \neq \emptyset$. It follows that p_l is not the sink of $\gamma(l, M(v, l))$.

By Lemma 6.3.5 on page 110 it must then hold that the unique sink of $\gamma(l, M(v, l))$ must be equal to the sink of $\gamma(l, M(v, l) \cup \{l\})$ (e.g. the yellow area in Example 6.3.4 on page 110).

p_l was not pushed further through by rule 2.2, therefore for all elements before p_l (including p_i), $l \in M(v, 1)$. This means that p_i is actually the sink of $\gamma(l, M(v, l) \cup \{l\})$:

By (a) of **isVertex** it holds that $\sigma(p_i) \cap \{1, \dots, i-1\} = \emptyset$. Since $l \in p_i$ it holds that $\sigma(p_i) \cap \{l\} = \emptyset$ because otherwise there would be a self loop.

Remember that $i, l \in \kappa_k$.

$$\begin{aligned}
\sigma(p_i) \cap \{1, \dots, i-1\} = \emptyset &\Rightarrow \sigma(p_i) \cap (\{1, \dots, i-1\} \setminus \kappa_k) = \emptyset \\
&\Rightarrow \sigma(p_i) \cap (\{1, \dots, i-1\} \setminus \kappa_k) \cup \{l\} = \emptyset \\
&\Rightarrow \sigma(p_i) \cap (\{1, \dots, l\} \setminus \kappa_k) \cup \{l\} = \emptyset \\
&\Rightarrow \sigma(p_i) \cap \gamma(l, M(v, l) \cup \{l\}) = \emptyset \\
&\Rightarrow p_i \text{ is the sink of } \gamma(l, M(v, l) \cup \{l\}).
\end{aligned}$$

But p_i is *not* the sink of $\gamma(l, M(v, l))$, since we are in case 2.3:

$$\sigma(p_i) \cap \{1, \dots, i\} \neq \emptyset \Rightarrow \sigma(p_i) \cap \gamma(l, M(v, l)) \neq \emptyset.$$

Therefore, neither the unique sink of $\gamma(l, M(v, l) \cup \{l\})$ nor the unique sink of $\gamma(l-1, M(v, l))$ is the sink of $\gamma(l, M(v, l))$. σ is not a unique sink orientation. This results in a violation of type (GUV_4) as described in the proof of the Lemma 6.3.5 on page 110.

(e) If (e) is broken, there must be a point that is equal or lexicographically bigger than q . Let p_j be that point.

If $q = p_j$, we know that p_j is not pushed further through and there is no violation of condition (b), therefore it must hold that $\sigma(p_j) \cap \{1, \dots, j\} = \{j\}$. Since there is no violation of (d), it must hold that $j \in q$. But we assumed that $q = p_j$, therefore $j \in p_j$. Thus we found a self loop violation of type $(GUV1)$.

If $q \neq p_j$, let l be the largest index at which $(p_j)_l > q_l$. For all $k > l$ it holds that $(p_j)_k = q_k$. There are four cases we have to take a closer look at.

First: $(q, \dots, \dots, \underbrace{p_j, \dots, \dots}_{\kappa_l})$.

In this case, $(q)_l \neq (p_j)_l$, this means either q or p_j violates (c) or (d) of **isVertex**.

Second: $(q, \dots, \dots, \underbrace{p_j, \dots, \dots}_{\kappa_l})$.

In this case we know by condition (d) of **isVertex** that $(q)_l = j$. It follows that for all k with $k > j$ and $k \in \kappa_l$, it must hold that $p_k = \perp$. Otherwise there would be a violation of condition (d), since then q would have to contain j as well as k , which is not possible. Condition (c) therefore enforces, that $(p_j)_l \neq k$. So $((p_j)_l) \leq j$, which is a contradiction to the assumption that q is lexicographically bigger than p_j .

Third: $(q, \dots, \dots, \underbrace{p_j, \dots, \dots}_{\kappa_k})$ and fourth: $(\underbrace{q, \dots, \dots}_{\kappa_l}, \dots, \underbrace{p_j, \dots, \dots}_{\kappa_k})$.

Let k be the block which contains j , i.e. $j \in \kappa_k$. If there is no violation of condition (d) of **isVertex**, it must hold that $q_k = j$. Furthermore, it must hold that $(p_j)_k = j$, because l is by assumption the largest index at which q and p_j differ. Since there is no violation of (b) it must hold that $\sigma(p_j) = \{j\}$. Thus, we found a self loop violation of type $(GUV1)$.

All in all, whenever $S(u) = u$ and **isVertex**(u) = *false*, there is a violation in the GRID-USO instance.

A solution of type (UF1) b Let $v = (p_1, \dots, p_n, p_{n+1})$ be a solution of type (UF1) and $u = S(v)$. This means that $S(v) \neq v$ and either

(a) $S(u) = u$ or

(b) $c(u) \leq c(v)$.

Assume it holds that $c(u) \leq c(v)$. Note that $S(u) \neq u$. Based on the rules that are used to calculate $S(v)$ we can argue that this is impossible:

1. $S(v)$ cannot be determined by rule (1), since then it would hold that $S(v) = v$, which is a contradiction to our assumption.
- 2.1. $S(v)$ cannot be determined by rule (2.1), since then it would hold that $S(v) = v$, which is a contradiction to our assumption.
- 2.2. If $S(v)$ was determined by the second rule, it can be proven that $c(u) > c(v)$.

v and u differ only in the two points at position i and $i + 1$. The cost of v and u differs only in the terms of the sum of i and $(i + 1)$.

$$\begin{aligned}
& c(u) \stackrel{!}{>} c(v) \\
& \Leftrightarrow \sum_{j=1}^{nd} \omega^j l_j > \sum_{j=1}^{nd} \omega^j l'_j \\
& \Leftrightarrow \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j l_j + \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j l_j > \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j l'_j + \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j l'_j \\
& \Leftrightarrow \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j g(\perp, i, j - (i-1) \cdot d) + \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j g(p_i, i+1, j - i \cdot d) \\
& > \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j g(p_i, i, j - (i-1) \cdot d) + \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j g(p_{i+1}, i+1, j - i \cdot d) \\
& \Leftrightarrow \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j g(p_i, i+1, j - i \cdot d) > \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j (\omega - 1) + \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j g(p_{i+1}, i+1, j - i \cdot d).
\end{aligned}$$

There are 4 cases that need to be inspected now. First, $g(p_i, i+1, j - i \cdot d)$ might either be $(\omega - 1)$ or $(p_i)_{j-i \cdot d}$. Second, $g(p_{i+1}, i+1, j - i \cdot d)$ might either be $(p_{i+1})_{j-i \cdot d}$ or 0 if $p_{i+1} = \perp$. We need to check all four combinations of them to make sure the inequality holds every time.

First, let $g(p_i, i+1, j - i \cdot d) = (\omega - 1)$ and $g(p_{i+1}, i+1, j - i \cdot d) = 0$:

$$\begin{aligned}
& \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j (\omega - 1) \stackrel{!}{>} \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j (\omega - 1) + \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j \cdot 0 \\
& \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j (\omega - 1) > \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j (\omega - 1) \\
& \omega^{id} \sum_{j=1}^d \omega^j (\omega - 1) > \omega^{(i-1)d} \sum_{j=1}^d \omega^j (\omega - 1) \\
& \omega^{id} > \omega^{(i-1)d}.
\end{aligned}$$

This is obviously true.

It holds that:

$$\sum_{j=1}^d \omega^j (\omega - 1) = \omega^{d+1} - \omega.$$

Second, let $g(p_i, i + 1, j - i \cdot d) = (\omega - 1)$ and $g(p_{i+1}, i + 1, j - i \cdot d) = (p_{i+1})_{j-i \cdot d}$:

$$\begin{aligned} & \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j (\omega - 1) \stackrel{!}{>} \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j (\omega - 1) + \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j (p_{i+1})_{j-i \cdot d} \\ & \omega^{id} \sum_{j=1}^d \omega^j (\omega - 1) > \omega^{(i-1) \cdot d} \sum_{j=1}^d \omega^j (\omega - 1) + \omega^{id} \sum_{j=1}^d \omega^j (p_{i+1})_j \\ & \omega^{id} \sum_{j=1}^d \omega^j (\omega - 1) - \omega^{id} \sum_{j=1}^d \omega^j (p_{i+1})_j > \omega^{(i-1) \cdot d} \sum_{j=1}^d \omega^j (\omega - 1) \\ & \omega^{id} \left(\sum_{j=1}^d \omega^j (\omega - 1) - \sum_{j=1}^d \omega^j (p_{i+1})_j \right) > \omega^{(i-1) \cdot d} (\omega^{d+1} - \omega) \\ & \omega^{id} \left(\sum_{j=1}^d \omega^j (\omega - 1) - \sum_{j=1}^d \omega^j (p_{i+1})_j \right) > \omega^{id+1} \omega^{-(d+1)} (\omega^{d+1} - \omega) \\ & \omega^{id+1} \left(\sum_{j=1}^d \omega^{j-1} (\omega - 1) - \sum_{j=1}^d \omega^{j-1} (p_{i+1})_j \right) > \omega^{id+1} (1 - \omega^{-d}) \\ & \underbrace{\left(\sum_{j=1}^d \omega^{j-1} (\omega - 1) - \sum_{j=1}^d \omega^{j-1} (p_{i+1})_j \right)}_{>1} > \underbrace{(1 - \omega^{-d})}_{<1}. \end{aligned}$$

Third, let $g(p_i, i + 1, j - i \cdot d) = (p_i)_{j-i \cdot d}$ and $g(p_{i+1}, i + 1, j - i \cdot d) = (p_{i+1})_{j-i \cdot d}$:

$$\begin{aligned} & \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j (p_i)_{j-i \cdot d} \stackrel{!}{>} \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j (\omega - 1) + \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j (p_{i+1})_{j-i \cdot d} \\ & \omega^{id} \sum_{j=1}^d \omega^j (p_i)_j > \omega^{(i-1) \cdot d} \sum_{j=1}^d \omega^j (\omega - 1) + \omega^{id} \sum_{j=1}^d \omega^j (p_{i+1})_j \\ & \omega^{id} \sum_{j=1}^d \omega^j (p_i)_j - \omega^{id} \sum_{j=1}^d \omega^j (p_{i+1})_j > \omega^{(i-1) \cdot d} (\omega^{d+1} - \omega) \\ & \omega^{id} \left(\sum_{j=1}^d \omega^j (p_i)_j - \sum_{j=1}^d \omega^j (p_{i+1})_j \right) > \omega^{id+1} \omega^{-(d+1)} (\omega^{d+1} - \omega) \\ & \omega^{id+1} \left(\sum_{j=1}^d \omega^{j-1} (p_i)_j - \sum_{j=1}^d \omega^{j-1} (p_{i+1})_j \right) > \omega^{id+1} (1 - \omega^{-d}) \\ & \underbrace{\left(\sum_{j=1}^d \omega^{j-1} (p_i)_j - \sum_{j=1}^d \omega^{j-1} (p_{i+1})_j \right)}_{>1} > \underbrace{(1 - \omega^{-d})}_{<1}. \end{aligned}$$

The left side of the inequality is larger than 1 because of condition (e) of `isVertex` we know that p_{i+1} is lexicographically smaller from right to left than p_i . Therefore this inequality holds.

Fourth, let $g(p_i, i + 1, j - i \cdot d) = (p_i)_{j-id}$ and $g(p_{i+1}, i + 1, j - i \cdot d) = 0$:

$$\begin{aligned}
\sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j (p_i)_{j-id} &\stackrel{!}{>} \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j (\omega - 1) + \sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j \cdot 0 \\
\sum_{j=i \cdot d+1}^{(i+1) \cdot d} \omega^j (p_i)_{j-id} &> \sum_{j=(i-1) \cdot d+1}^{i \cdot d} \omega^j (\omega - 1) \\
\omega^{id} \sum_{j=1}^d \omega^j (p_i)_j &> \omega^{(i-1)d} \sum_{j=1}^d \omega^j (\omega - 1) \\
\omega^{id+1} \sum_{j=1}^d \omega^{j-1} (p_i)_j &> \omega^{(i-1)d} (\omega^{d+1} - \omega) \\
\omega^{id+1} \sum_{j=1}^d \omega^{j-1} (p_i)_j &> \omega^{id+1} \omega^{-(d+1)} (\omega^{d+1} - \omega) \\
\omega^{id+1} \sum_{j=1}^d \omega^{j-1} (p_i)_j &> \omega^{id+1} (1 - \omega^{-d}) \\
\underbrace{\sum_{j=1}^d \omega^{j-1} (p_i)_j}_{>1} &> \underbrace{(1 - \omega^{-d})}_{<1}.
\end{aligned}$$

In conclusion, every inequality holds.

- 2.3. If $S(v)$ was determined by the third rule, it can be proven again that $c(v) < c(u)$. v and u only differ in the first position. Furthermore we know that $p_1 = \perp$. If it wasn't and i was equal to 1, then this is proof of a self loop and we found a violation of type (GUV1).

$$\begin{aligned}
c(u) &\stackrel{!}{>} c(v) \\
\sum_{j=1}^{nd} \omega^j l_j &> \sum_{j=1}^{nd} \omega^j l'_j \\
\sum_{j=1}^d \omega^j l_j &> \sum_{j=1}^d \omega^j l'_j \\
\sum_{j=1}^d \omega^j g(q, 1, j) &> \sum_{j=1}^d \omega^j g(p_1, 1, j) \\
\sum_{j=1}^d \omega^j g(q, 1, j) &> \sum_{j=1}^d \omega^j g(\perp, 1, j) \\
\sum_{j=1}^d \omega^j g(q, 1, j) &> 0.
\end{aligned}$$

Since $q \neq \perp$, this inequality is clearly true.

In conclusion, it can never happen that $S(v) \neq v$ and $c(u) \leq c(v)$.

A violation of type (UFV1) a Given $v = (p_1, \dots, p_n)$ and $u = (q_1, \dots, q_n)$ with $v \neq u$, $v \neq S(v)$, $u \neq S(u)$ and $\text{isVertex}(v) = \text{isVertex}(u) = \text{true}$, for a violation of type (UFV1) it holds that either

- (a) $c(v) = c(u)$ or
- (b) $c(v) < c(u) < c(S(v))$.

If we are in case (a) it follows that there are two unique sinks in a subcube and it follows a violation of type (GUV4):

If the cost of v and u are equal but $v \neq u$, there must be one or more points in the nodes for which it holds that $p_i \neq q_i$ but $g(p_i, i, j) = g(q_i, i, j)$ for all $j = 1, \dots, d$. This can only be true if $g(p_i, i, j) = g(q_i, i, j) = \omega - 1$. Therefore $\sigma(p_i) \cap \{1, \dots, i\} = \emptyset$ and $\sigma(q_i) \cap \{1, \dots, i\} = \emptyset$. Let $p_i \in v$, $q_i \in u$ be the ones with the largest index i for which that holds. Therefore for all points $l > i$ it holds that they are equal: $p_l = q_l$.

Let k be the index of the block that contains i , i.e. $i \in \kappa_k$. By condition (c) and (d) of isVertex it holds that for all $l \geq k$, it holds that $(p_i)_l = (q_i)_l$.

It follows that v and u are both in the subgrid $\gamma(i, M(v, i))$. Since they are both sinks, this is a (GUV4) violation. Both have the same refined index of $(0, \dots, 0)$ in the subgrid $\gamma(i, M(v, i))$.

A violation of type (UFV1) b If we are in case (b), we take a look at how $S(v)$ is determined in order to show that the original GRID-USO instance had a violation as well or that the case cannot possibly be reached by the construction.

1. $S(v)$ cannot be determined by rule 1, since then it would hold that $S(v) = v$, which is a contradiction to our assumption.
- 2.1. $S(v)$ cannot be determined by rule 2.1, since then it would hold that $S(v) = v$, which is a contradiction to our assumption.
- 2.2. If $S(v)$ was determined by the second rule, let i be the smallest index of v such that $p_i \neq \perp$.

Since $c(v) < c(u) < c(S(v))$ and v and $S(v)$ only differ in positions i and $i + 1$, it must hold:

Vertex	Index						
	1	...	$i - 1$	i	$i + 1$	$i + 2$...
v	\perp	...	\perp	x	y	z	...
u	?	...	?	a	b	z	...
$S(v)$	\perp	...	\perp	\perp	x	z	...

For all indices $k > i + 1$, v , u and $S(v)$ have the same entries because if they hadn't it would imply that for some k $g(p_k, k, j) = g(q_k, k, j) = \omega - 1$ which results in a violation of isVertex (b), since both nodes have an element unequal to \perp with an index smaller than k .

There are several cases in which it might hold that $c(v) < c(u) < c(S(v))$. In the following it will be shown that each case leads to a violation in the GRID-USO instance.

In order to fulfill $c(v) < c(u) < c(S(v))$ one of the following three cases must hold:

(i) $\forall j = 1, \dots, d: g(b, i + 1, j) = g(y, i + 1, j), g(a, i, j) = g(x, i, j)$ and $\exists l < i: u_l \neq \perp$.

Note that this does not mean that $y = b$ and $a = x$. It can happen that they are two different nodes for which g returns the same value $\omega - 1$.

(ii) $\forall j = 1, \dots, d: g(b, i + 1, j) = g(y, i + 1, j)$ and $\exists j \in \{1, \dots, d\}: g(a, i, j) > g(x, i, j)$ and $g(a, i, k) = g(x, i, k)$ for $j < k < d$.

(iii) $\exists j \in \{1, \dots, d\}: g(b, i + 1, j) > g(y, i + 1, j)$ and $g(b, i + 1, k) = g(y, i + 1, k)$ for $j < k < d$.

Case (i): Since we are in case 2.2, it holds that $\forall j = 1, \dots, d: g(x, i, j) = \omega - 1$. It follows that also $\forall j = 1, \dots, d: g(a, i, j) = \omega - 1$, but not all elements below i are \perp . Thus there is a violation against (b) of **isVertex**.

Case (ii): Since $\forall j = 1, \dots, d: g(x, i, j) = \omega - 1$, there is no possibility that there exists a $j \in \{1, \dots, d\}$ for which $g(a, i, j) > g(x, i, j)$. Therefore this case cannot happen.

Case (iii): It must hold that $b \neq \perp$, otherwise its cost could not be bigger than the cost of y .

In this case we have to take a closer look at some predecessor w of u that must exist, which is equal to u for all indices bigger than $i + 1$ and otherwise looks like the following:

Vertex	Index						
	1	...	$i - 1$	i	$i + 1$	$i + 2$...
w	\perp	...	\perp	b	r	z	...
$S(w)$	\perp	...	\perp	\perp	b	z	...
...	...						
u	?	...	?	a	b	z	...

We compare w to v : We know that for all $j = 1, \dots, d: g(b, i, j) = \omega - 1$ and $g(x, i, j) = \omega - 1$.

Vertex	Index						
	1	...	$i - 1$	i	$i + 1$	$i + 2$...
v	\perp	...	\perp	x	y	z	...
w	\perp	...	\perp	b	r	z	...

If $r = y = \perp$ or $r \neq \perp$ and $y \neq \perp$, M is equal for both x and b : $M(v, i) = M(w, i)$. Furthermore, since both x and b are pushed through in the next step, they are both sinks of the same subcube $\gamma(i, M)$. By Lemma 6.3.5 on page 110 This is a refined index violation of type (GUV_4) for that subcube.

If $r \neq \perp$ and $y = \perp$, M is different for x and b : $M(v, i) \neq M(w, i)$. They differ in exactly one element: $M(v, i) \cup \{i + 1\} = M(w, i)$. r is the sink of the subcube $\gamma(i, M(w, i + 1))$ and b is the sink of subcube $\gamma(i + 1, M(w, i + 1) \cup \{i + 1\}) = \gamma(i + 1, M(w, i))$. By Lemma 6.3.5 on page 110 one of them must be the sink for $\gamma(i + 1, M(w, i + 1))$. Since r was not pushed through, r is *not* the unique sink of $\gamma(i + 1, M(w, i + 1))$. b is the sink of $\gamma(i + 1, M(w, i + 1))$, which implies that $\forall j = 1, \dots, d: g(b, i + 1, j) = \omega - 1$. This means that $S(v)$ cannot possibly have bigger cost than u , which is a contradiction against our assumption. This case cannot happen.

If $r = \perp$ and $y \neq \perp$, M is different for x and b : $M(v, i) \neq M(w, i)$. We know that y is not the unique sink of $\gamma(i + 1, M(v, i + 1))$, since x is that sink. x is also the sink of $\gamma(i + 1, M(v, i + 1) \cup \{i + 1\})$. Therefore y must be the sink of $\gamma(i, M(v, i + 1))$. It follows that there exists a predecessor s of v with the following form:

Vertex	Index						
	1	...	$i - 1$	i	$i + 1$	$i + 2$...
s	\perp	...	\perp	y	\perp	z	...
v	\perp	...	\perp	x	y	z	...

We compare w to s :

Vertex	Index						
	1	...	$i - 1$	i	$i + 1$	$i + 2$...
s	\perp	...	\perp	y	\perp	z	...
w	\perp	...	\perp	b	\perp	z	...

M is equal for y and b : $M(s, i) = M(w, i)$. It holds that for all $j = 1, \dots, d$: $g(y, i, j) = \omega - 1$ and $g(b, i, j) = \omega - 1$. b and y are both sinks of the same subcube $\gamma(i, M(w, i))$. By Lemma 6.3.5 on page 110 This is a refined index violation of type (GUV_4) for that subcube.

In conclusion, every possible case either let to a violation or it was proven that it is impossible to reach.

2.3. v and $S(v)$ only differ in the first element. Therefore u must only differ from them in the first element as well.

Vertex	Index						
	1	2	...	$i - 1$	i	$i + 1$...
v	\perp	\perp	x	y	...
u	z	\perp	...	\perp	x	y	...
$S(v)$	q	\perp	...	\perp	x	y	...

It holds that $i \neq 1$, since otherwise we have a self loop. Therefore, $g(p_1, 1, j) = 0$ for all $j = 1, \dots, d$. It follows that $u_1 \neq \perp$ because otherwise $c(v) = c(u)$.

Since all other elements of u and $S(v)$ are equal, they enforce the same value of each element of the first point by rule (c) and (d) of **isVertex**. Therefore z must be equal to q and $u = S(v)$.

As in case $(UFV1 a)$ it follows a violation of type (GUV_4) .

A violation of type (UFV2) If there is a violation of type $(UFV2)$, there are two points v and x such that v is a solution of type $(UF1)$, $v \neq x$, $S(x) \neq x$ and $c(v) < c(x)$. Since v is also a solution of type $(UF1)$, it holds that $S(v) = u$, $v \neq u$ and either $S(u) = u$ or $c(u) \leq c(v)$.

If $S(u)$ is determined by anything other than rule (2.1), there is a violation in the GRID-USO instance as shown in case $(UF1)$.

So if $S(u)$ is indeed determined by rule (2.1), we must show that this, in combination with x , leads to another violation.

Since x is a node on a line, there must exist an end of that line somewhere. Let y be that end. It holds that $c(y) > c(x)$. Therefore it must hold that $v \neq y$ because $c(x) > c(v)$.

If y is not a violation, it was also determined by rule (2.1). Then u and y have the following form: $u = (\perp, \dots, \perp, p)$ and $y = (\perp, \dots, \perp, q)$.

This means that both p and q are a sink of the whole grid, which is a clear refined index violation of type (GUV4), since these are two nodes with 0 outgoing edges and $r_\sigma(p) = (0, \dots, 0) = r_\sigma(q)$.

Conclusion All in all, every solution that is found in the constructed UNIQUE FORWARD EOPL instance can be mapped back to the corresponding GRID-USO instance in polynomial time.

Every solution of type (GU1) is only mapped to solutions of type (UF1). Every violation of the created UNIQUE FORWARD EOPL instance is mapped back to a violation of the GRID-USO instance. Therefore this reduction is promise preserving.

□

7. Conclusion and Future Directions

In this thesis, in Chapter 3 on page 13 an overview over several complexity classes for search problems was given. In Chapter 4 on page 23 a detailed prove of OPDC being UniqueEOPL-hard was presented. An overview over all currently known problems in UniqueEOPL was given in Chapter 5 on page 58. We were able to prove that GRID-USO is in UniqueEOPL as well, as shown in Chapter 6 on page 96.

The complexity of many problems that were proven to be in UniqueEOPL is still unknown though, i.e. it is not known whether they are complete for UniqueEOPL, any other class or maybe even solvable in polynomial time. One objective for future research clearly is to find more problems contained in UniqueEOPL and more UniqueEOPL-complete problems.

As a summary, here is a checklist of properties that any new problem for UniqueEOPL must fulfill:

- The problem needs to have a unique solution under certain circumstances (i.e. the promise).
- There must be a polynomial time verifiable certificate that proves that the promise does not hold.
- It is not necessary that the promise can be checked in polynomial time.
- The problem must be solvable by local search, i.e. there must exist an algorithm that allows us to get monotonically nearer to the solution. Each step must be calculated in polynomial time, even if it takes more than polynomial many steps to reach the end.
- The problem must have a unique start node and a line-following algorithm that solves the problem.
- Any problem that is complete for another complexity class (e.g. PLS, PPAD, CLS) is unlikely to be in UniqueEOPL, since this would imply that UniqueEOPL is equal to that other class [Fea+17, p.2].
- Problems that are in CLS and have unique solutions but that are not CLS-complete are good candidates to be in UniqueEOPL.

[Fea+20b] considers CUBE-USO and PL-CONTRACTION likely to be UniqueEOPL-complete. If CUBE-USO was UniqueEOPL-complete, so would GRID-USO. There is also a relation between PARITY GAME and model checking propositional μ -calculus [Pur95, p. 60], maybe some problems from that area can be proven to be in UniqueEOPL as well. There are also other versions of P-LCP which might be proven to be in UniqueEOPL. For example, [AV11] suggests that every $P_0 \cap Q_0$ -Matrix in a perturbed LCP is a P-Matrix.

[Fea+20b] believes that there are more interesting subclasses of UniqueEOPL, since there are problems with a unique solution under a certain promise and problems that have a unique solution unconditionally (like SSG).

The question whether $EOPL = CLS$ is still unresolved. Since [Fea+20a] proved that $CLS = PPAD \cap PLS$, this question is even more interesting. [Fea+20a] conjectures that $EOPL \neq CLS$ and suggests to provide an oracle separation similar to the ones done in

[Bea+98]. To prove that $\text{EOPL} = \text{CLS}$, one could try to reduce a CLS -complete problem (like $\text{META METRIC CONTRACTION}$, KKT or GD-LOCALSEARCH) to a problem in EOPL . There is currently no problem known that is in EOPL but not in UniqueEOPL . Finding one is of great interest.

All in all, there are many possibilities to continue the research on this topic with the help of the basics presented in this work.

Appendix

A. Background Information on LCP's

A.1. Solving an LCP

One can solve an LCP by a *Bard-type algorithm* [SW78]:

Algorithm 3: SOLVE-LCP

Data: N, M, q
if $\exists k: q_k < 0$ **then**
 choose a pivot row k ;
 $(N', M', q') :=$ exchange z_k and w_k by a principal pivot transformation;
 return SOLVE – LCP(N', M', q');
return $(0, q)$;

Calling SOLVE – LCP($I, -M, q$), this algorithm solves the LCP (q, M) . The termination of the algorithm depends on the rule by which k is chosen and if the LCP is degenerate or not.

For example when k is chosen such that q_k is the smallest entry of q , it is possible that the algorithm cycles in an endless loop. There are different strategies to prevent that, for example lexicographic perturbation. If M is a P-Matrix, a different rule for choosing k was proposed by Murty: $k := \min\{i: q_i < 0\}$ [CPS92, p. 244].

The algorithm may take exponential time before it terminates [Fat79, Theorem 2.3].

There are other algorithms to solve an LCP, for example Lemkes algorithm [CPS92, Chapter 4.4].

A.2. Principal Pivot Transformation

For a good introduction, see [CPS92, p. 71ff.].

The basic idea of principal pivot transformation is to rewrite a given LCP to an equivalent LCP so that it might be simpler to solve. Ideally, after some transformation the result to the LCP is the trivial solution.

Given an LCP with $M \in \mathbb{R}^{d \times d}$ and $q \in \mathbb{R}^{d \times 1}$, let $i := \{1, \dots, k\}$ and $j := \{k + 1, \dots, d\}$ such that $M_{i,i}$ is the i 'th principal submatrix of M . Assume $M_{i,i}$ is nonsingular, i.e. it can be inverted.

$M_{i,j}, M_{j,i}$ and $M_{j,j}$ are the other rectangular parts of M :

$$M = \begin{pmatrix} \boxed{\begin{matrix} m_{1,1} & \dots & m_{1,k} \\ \dots & \dots & \dots \\ m_{k,1} & \dots & m_{k,k} \end{matrix}} & \boxed{\begin{matrix} m_{1,k+1} & \dots & m_{1,d} \\ \dots & \dots & \dots \\ m_{k,k+1} & \dots & m_{k,d} \end{matrix}} \\ \boxed{\begin{matrix} m_{k+1,1} & \dots & m_{k+1,k} \\ \dots & \dots & \dots \\ m_{d,1} & \dots & m_{d,k} \end{matrix}} & \boxed{\begin{matrix} m_{k+1,k+1} & \dots & m_{k+1,d} \\ \dots & \dots & \dots \\ m_{d,k+1} & \dots & m_{d,d} \end{matrix}} \end{pmatrix}$$

$M_{i,i}$
 $M_{i,j}$

$M_{j,i}$
 $M_{j,j}$

Furthermore one can write z as $z = \begin{pmatrix} z_i \\ z_j \end{pmatrix}$ and analogously w and q .

Note that it is always possible to rearrange M such that for any set $\alpha \subseteq \{1, \dots, d\}$, $M_{\alpha,\alpha}$ is the leading principal submatrix [CPS92, p. 59f]. So if we want to swap only two variables z_k and w_k , we can rearrange the LCP such that the k 'th line is the first one and the principal pivot transformation is done for the 1'st leading principal submatrix.

The LCP $w = Mz + q$ can be written as:

$$w_i = M_{i,i} \cdot z_i + M_{i,j} \cdot z_j + q_i \quad (7.1)$$

$$w_j = M_{j,i} \cdot z_i + M_{j,j} \cdot z_j + q_j \quad (7.2)$$

z_i and w_i can now be exchanged which results in the following system:

$$z_i = M'_{i,i} \cdot w_i + M'_{i,j} \cdot z_j + q'_i \quad (7.3)$$

$$w_j = M'_{j,i} \cdot w_i + M'_{j,j} \cdot z_j + q'_j \quad (7.4)$$

where

$$q'_i := -M_{i,i}^{-1} \cdot q_i \quad (7.5)$$

$$q'_j := q_j - M_{j,i} \cdot M_{i,i}^{-1} \cdot q_i \quad (7.6)$$

and

$$M'_{i,i} := M_{i,i}^{-1} \quad (7.7) \quad M'_{i,j} := -M_{i,i}^{-1} \cdot M_{i,j} \quad (7.9)$$

$$M'_{j,i} := M_{j,i} \cdot M_{i,i}^{-1} \quad (7.8) \quad M'_{j,j} := M_{j,j} - M_{j,i} \cdot M_{i,i}^{-1} \cdot M_{i,j}. \quad (7.10)$$

The resulting matrix and vector

$$M' := \begin{pmatrix} M'_{i,i} & M'_{i,j} \\ M'_{j,i} & M'_{j,j} \end{pmatrix} \quad (7.11) \quad q' := \begin{pmatrix} q'_i \\ q'_j \end{pmatrix} \quad (7.12)$$

are called principal pivot transform of M and q with respect to the index i . The LCP with the rearranged matrix is equivalent to the original one.

Example A.1. Recall Example 5.3.6 on page 75. Let $d = 2$ and the P-LCP problem (q, M) be

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1 \\ -2 \end{pmatrix}.$$

We want to exchange b and y . As a first step, we need to rearrange the LCP, since we need to be the line index k for which $q_k < 0$ to be 1.

$$\begin{pmatrix} b \\ a \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} y \\ x \end{pmatrix} + \begin{pmatrix} -2 \\ 1 \end{pmatrix}.$$

In this example $k = 1$. The LCP split apart is:

$$\begin{aligned} b &= 3 \cdot y + 1 \cdot x - 2 \\ a &= -1 \cdot y + 1 \cdot x + 1 \end{aligned}$$

$$\begin{aligned} q'_1 &= -1/3 \cdot -2 = 2/3 \\ q'_2 &= 1 - (-1 \cdot 1/3 \cdot (-2)) = 1/3 \end{aligned}$$

$$\begin{aligned} M'_{1,1} &= 1/3 \\ M'_{1,2} &= -1/3 \cdot 1 = -1/3 \\ M'_{2,1} &= -1 \cdot 1/3 = -1/3 \\ M'_{2,2} &= 1 - (-1 \cdot 1/3 \cdot 1) = 4/3 \end{aligned}$$

$$M' = \begin{pmatrix} 1/3 & -1/3 \\ -1/3 & 10/3 \end{pmatrix}$$

The transformed LCP is:

$$\begin{pmatrix} y \\ a \end{pmatrix} = \begin{pmatrix} 1/3 & -1/3 \\ -1/3 & 4/3 \end{pmatrix} \begin{pmatrix} b \\ x \end{pmatrix} + \begin{pmatrix} 2/3 \\ 1/3 \end{pmatrix}.$$

The solution for this LCP is the trivial solution. Since the principal pivot transformation creates an equivalent LCP, it is also the solution to the original LCP.

$$x = 0, y = 2/3, a = 1/3, b = 0.$$

A.3. Perturbation

The most common application of solving degeneracy is to prevent the simplex algorithm from cycling. Many methods are directly applied to the algorithm instead of the original linear program. Since we are not using the simplex algorithm in any way, we use a strategy called perturbation instead, which augments a given linear program so that it is nondegenerate.

For a good introduction to perturbation, see [Mur83, Chapter 10]. As this book is very hard to find online, the relevant Lemmata and their proofs will be given in the following.

Recall Lemma 5.3.11 on page 80 [Mur83, Theorem 10.1]:

Given any $q \in \mathbb{R}^d$, there exists a positive number $\tilde{\epsilon} > 0$ such that whenever $0 < \epsilon < \tilde{\epsilon}$, the following perturbed problem is nondegenerate:

$$\begin{aligned} (A_\alpha)y &= q(\epsilon) \\ y &\geq 0 \text{ component wise.} \end{aligned} \tag{7.13}$$

Proof. (As done in [Mur83])

Let B_1, \dots, B_K be all the bases to 7.13, i.e. each B is nonsingular and consists of $m = \text{rank}((A_\alpha))$ many linear independent columns of (A_α) .

Let B_k be one of these bases. We name the element of B_k at row i and column j by $\beta_{i,j}^k$. Since B_k^{-1} is nonsingular, it holds that $(\beta_{i1}^k, \dots, \beta_{im}^k) \neq 0$ for each $i = 1, \dots, m$.

Let $p := B_k^{-1}q$. Then

$$B_k^{-1}q(\epsilon) = \begin{pmatrix} p_1 + \beta_{1,1}^k \epsilon + \dots + \beta_{1,m}^k \epsilon^m \\ \dots \\ p_m + \beta_{m,1}^k \epsilon + \dots + \beta_{m,m}^k \epsilon^m \end{pmatrix}.$$

For $i = 1, \dots, m$, each polynomial $p_i + \beta_{i,1}^k \theta + \dots + \beta_{i,m}^k \theta^m$ has at most m roots, i.e. m assignments of θ such that the polynomial is 0. If ϵ is chosen to be a number that is not a root, the polynomial is non-zero as required.

Let R be the set of roots of all polynomials for $k = 1, \dots, K$ and $i = 1, \dots, m$. R is a finite set consisting of at most Km^2 real numbers. Hence it is possible to pick an $\tilde{\epsilon}$ such that R contains no element belonging to the open interval $0 < \epsilon < \tilde{\epsilon}$.

In any basic solution of 7.13 all basic variables are nonzero whenever $0 < \epsilon < \tilde{\epsilon}$ and hence 7.13 is nondegenerate. □

Recall Lemma 5.3.12 on page 80 [Mur83, Theorem 10.5]:

If B is a feasible basis for the perturbed LCP 7.13 with ϵ being arbitrarily small, then B is a feasible basis for the original un-perturbed LCP 5.2.

Proof. Again, we name the element of B at row i and column j by $\beta_{i,j}$. Let $p := B^{-1}q$. Then

$$B^{-1}q(\epsilon) = \begin{pmatrix} p_1 + \beta_{1,1} \epsilon + \dots + \beta_{1,m} \epsilon^m \\ \dots \\ p_m + \beta_{m,1} \epsilon + \dots + \beta_{m,m} \epsilon^m \end{pmatrix}.$$

If for some i it holds that $p_i < 0$, then $p_i + \beta_{i,1} \epsilon + \dots + \beta_{i,m} \epsilon^m < 0$ when ϵ is arbitrarily small. This is a contradiction to B being a feasible basis to the perturbed LCP 7.13. Hence $p_i \geq 0$ for all $i = 1, \dots, m$, which implies that B is a feasible solution for the original LCP 5.2. □

B. Grid-USO to OPDC - Working Solution Types

A solution of type (O1) Let point $p \in \Gamma'$ be a solution of (O1). It holds that $\forall i = 1, \dots, d: D_i(p) = \text{Zero}$. This implies that $\sigma_\phi(f^{-1}(p)) \cap \kappa_i = \emptyset$ for all i which means that $\sigma_\phi(f^{-1}(p)) = \emptyset$ and therefore $f^{-1}(p)$ is a sink and a solution of type (GU1).

A solution of type (OV1) For an i -Slice \mathbf{s} , a solution of type (OV1) consists of two fixpoints $p, q \in \Gamma'_\mathbf{s}$ with $p \neq q$ such that $\forall j \leq i: D_j(p) = D_j(q) = \text{Zero}$.

Let $v := f^{-1}(p)$ and let $u := f^{-1}(q)$.

This implies that $\forall j \leq i: \sigma_\phi(v) \cap \kappa_j = \sigma_\phi(u) \cap \kappa_j = \emptyset$ which means that v and u are both sinks on the same subgrid j , which is a violation of type (GUV2):

Let $v_{a,b} := v \cap \kappa_a \cap \dots \cap \kappa_b$. To formally prove this, we will look at the subsets of the first 1 to i and the last $i + 1$ to d elements separately. This can be done because of Lemma 6.1.3 on page 97.

First, we take a closer look at the last $i + 1$ to d elements. Since p and q are on the same i -Slice, their last $d - i$ elements are equal. This also means that the last $d - i$ elements of v and u are equal and therefore $v_{i+1,d} = u_{i+1,d}$. Hence $(v_{i+1,d} \oplus u_{i+1,d}) = \emptyset$. Therefore, if $(v \oplus u) \cap (\sigma_\phi(v) \oplus \sigma_\phi(u)) \neq \emptyset$, it must be due to the first $1, \dots, i$ elements of v, u and their orientation functions.

Now let's take a closer look at the first 1 to i elements. It holds that $v_{1,i} \neq u_{1,i}$ and hence $(v_{1,i} \oplus u_{1,i}) \neq \emptyset$. Since we have a (OV1) violation, it also holds that $\forall j \leq i: \sigma_\phi(v) \cap \kappa_j = \sigma_\phi(u) \cap \kappa_j = \emptyset$ which implies that $\sigma_\phi(v_{1,i}) = \sigma_\phi(u_{1,i}) = \emptyset$.

It follows that $(v \oplus u) \cap (\sigma_\phi(v) \oplus \sigma_\phi(u)) = \emptyset$. Therefore, we found a (GUV2) violation.

A solution of type (OV3) If there is a violation of type (OV3), there is an i -Slice \mathbf{s} and two points $p, q \in \Gamma'_\mathbf{s}$ such that

- $\forall j < i: D_j(p) = \text{Zero}$ and either
- $p_i = 0$ and $D_i(p) = \text{Down}$ or
- $p_i = \omega_i$ and $D_i(p) = \text{Up}$.

If this happens, it implies that the subgrid that is represented by the i -Slice does not have a sink. This immediately implies a violation of type (GUV3) for that subgrid and by extension a violation for the whole instance.

Acknowledgments

The author would like to thank Wolfgang Mulzer for his helpful comments and suggestions, and especially Susanne Borzechowski and Till-Julius Krüger for their support throughout the writing of this thesis.

Bibliography

- [ST42] A. H. Stone and J. W. Tukey. “Generalized “sandwich” theorems”. In: *Duke Math. J.* 9.2 (June 1942), pp. 356–359. DOI: 10.1215/S0012-7094-42-00925-6. URL: <https://doi.org/10.1215/S0012-7094-42-00925-6>.
- [Sha53] Lloyd S Shapley. “Stochastic games”. In: *Proceedings of the national academy of sciences* 39.10 (1953), pp. 1095–1100.
- [CD70] Richard W. Cottle and George B. Dantzig. “A generalization of the linear complementarity problem”. In: *Journal of Combinatorial Theory* 8.1 (1970), pp. 79–90. ISSN: 0021-9800. DOI: [https://doi.org/10.1016/S0021-9800\(70\)80010-2](https://doi.org/10.1016/S0021-9800(70)80010-2). URL: <http://www.sciencedirect.com/science/article/pii/S0021980070800102>.
- [SW78] Alan Stickney and Layne Watson. “Digraph Models of Bard-Type Algorithms for the Linear Complementarity Problem”. In: *Mathematics of Operations Research* 3.4 (1978), pp. 322–333. ISSN: 0364765X, 15265471. URL: <http://www.jstor.org/stable/3689630>.
- [EM79] A Ehrenfeucht and J Mycielski. “Positional strategies for mean payoff games”. eng. In: *International journal of game theory* 8.2 (1979), pp. 109–113. ISSN: 0020-7276.
- [Fat79] Yahya Fathi. “Computational complexity of LCPs associated with positive definite symmetric matrices”. eng. In: *Mathematical programming* 17.1 (1979), pp. 335–344. ISSN: 0025-5610.
- [SY82] Alan L. Selman and Yacov Yacobi. “The complexity of promise problems”. In: *Automata, Languages and Programming*. Ed. by Mogens Nielsen and Erik Meineche Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 502–509. ISBN: 978-3-540-39308-5.
- [Ald83] David Aldous. “Minimization Algorithms and Random Walk on the d -Cube”. In: *Ann. Probab.* 11.2 (May 1983), pp. 403–413. DOI: 10.1214/aop/1176993605. URL: <https://doi.org/10.1214/aop/1176993605>.
- [Mur83] Katta G Murty. *Linear programming*. eng. New York [u.a.]: Wiley, 1983, XIX, 482 S. : graph. Darst. ISBN: 0-471-09725-X. URL: https://primo.fu-berlin.de/FUB:FUB_ALMA_DS21892916160002883.
- [VV86] L.G. Valiant and V.V. Vazirani. “NP is as easy as detecting unique solutions”. In: *Theoretical Computer Science* 47 (1986), pp. 85–93. ISSN: 0304-3975. DOI: [https://doi.org/10.1016/0304-3975\(86\)90135-0](https://doi.org/10.1016/0304-3975(86)90135-0). URL: <http://www.sciencedirect.com/science/article/pii/0304397586901350>.
- [JPY88] David S Johnson, Christos H Papadimitriou, and Mihalis Yannakakis. “How easy is local search?” In: *Journal of computer and system sciences* 37.1 (1988), pp. 79–100.

- [MY88] Katta G Murty and Feng-Tien Yu. *Linear complementarity, linear and nonlinear programming*. Vol. 3. Citeseer, 1988.
- [Yan88] Mihalis Yannakakis. “Computational complexity”. In: *Local search in combinatorial optimization* (1988), pp. 19–55.
- [Chu89] S. J Chung. “NP-Completeness of the linear complementarity problem”. eng. In: *Journal of optimization theory and applications* 60.3 (1989), pp. 393–399. ISSN: 0022-3239.
- [HMS89] J. Hartmanis, American Mathematical Society. Meeting, and American Mathematical Society. *Computational Complexity Theory*. AMS short course lecture notes. American Mathematical Society, 1989. ISBN: 9780821801314. URL: <https://books.google.de/books?id=tHfHCQAAQBAJ>.
- [Con90] Anne Condon. “On Algorithms for Simple Stochastic Games.” In: *Advances in computational complexity theory* 13 (1990), pp. 51–72.
- [MP91] Nimrod Megiddo and Christos H Papadimitriou. “On total functions, existence theorems and computational complexity”. In: *Theoretical Computer Science* 81.2 (1991), pp. 317–324.
- [Con92] Anne Condon. “The complexity of stochastic games”. In: *Information and Computation* 96.2 (1992), pp. 203 –224. ISSN: 0890-5401. DOI: [https://doi.org/10.1016/0890-5401\(92\)90048-K](https://doi.org/10.1016/0890-5401(92)90048-K). URL: <http://www.sciencedirect.com/science/article/pii/089054019290048K>.
- [CPS92] R.W. Cottle, J.S. Pang, and R.E. Stone. *The Linear Complementarity Problem*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 1992. ISBN: 9780898719000. URL: https://books.google.de/books?id=bGM80_pSzNIC.
- [Lov+93] László Lovász et al. “Random walks on graphs: A survey”. In: *Combinatorics, Paul erdos is eighty* 2.1 (1993), pp. 1–46.
- [Cox94] Gregory E Coxson. “The P-matrix problem is co-NP-complete”. eng. In: *Mathematical programming* 64.1-3 (1994), pp. 173–178. ISSN: 1436-4646.
- [Pap94] Christos H. Papadimitriou. “On the complexity of the parity argument and other inefficient proofs of existence”. In: *Journal of Computer and System Sciences* 48.3 (1994), pp. 498 –532. DOI: [https://doi.org/10.1016/S0022-0000\(05\)80063-7](https://doi.org/10.1016/S0022-0000(05)80063-7).
- [HS95] GJ Habetler and BP Szanc. “Existence and uniqueness of solutions for the generalized linear complementarity problem”. In: *Journal of optimization theory and applications* 84.1 (1995), pp. 103–116.
- [Kri+95] Sriram C. Krishnan et al. “The Rabin Index and Chain automata, with applications to automata and games”. In: *In Computer Aided Verification, Proc. 7th Int. Conference, LNCS 939*. 1995, pp. 253–266.
- [Lud95] W. Ludwig. “A Subexponential Randomized Algorithm for the Simple Stochastic Game Problem”. In: *Information and Computation* 117.1 (1995), pp. 151–155. ISSN: 0890-5401. DOI: <https://doi.org/10.1006/inco.1995.1035>. URL: <https://www.sciencedirect.com/science/article/pii/S0890540185710358>.
- [Pap95] Christos H Papadimitriou. *Computational complexity*. eng. Reprint. with corr. Reading, Mass. [u.a.]: Addison-Wesley, 1995, XV, 523 S. : graph. Darst. ISBN: 0-201-53082-1.

- [Pur95] Anuj Puri. “Theory of Hybrid Systems and Discrete Event Systems”. PhD thesis. EECS Department, University of California, Berkeley, 1995. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1995/2950.html>.
- [Pri+96] Vyatcheslav B Priezzhev et al. “Eulerian walkers as a model of self-organized criticality”. In: *Physical Review Letters* 77.25 (1996), p. 5079.
- [ZP96] Uri Zwick and Mike Paterson. “The complexity of mean payoff games on graphs”. In: *Theoretical Computer Science* 158.1 (1996), pp. 343–359. ISSN: 0304-3975. DOI: [https://doi.org/10.1016/0304-3975\(95\)00188-3](https://doi.org/10.1016/0304-3975(95)00188-3). URL: <http://www.sciencedirect.com/science/article/pii/S0304397595001883>.
- [Tho97] Wolfgang Thomas. “Languages, Automata, and Logic”. In: *Handbook of Formal Languages: Volume 3 Beyond Words*. Ed. by Grzegorz Rozenberg and Arto Salomaa. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 389–455. ISBN: 978-3-642-59126-6. DOI: 10.1007/978-3-642-59126-6_7. URL: https://doi.org/10.1007/978-3-642-59126-6_7.
- [Bea+98] Paul Beame et al. “The Relative Complexity of NP Search Problems”. In: *Journal of Computer and System Sciences* 57.1 (1998), pp. 3–19. ISSN: 0022-0000. DOI: <https://doi.org/10.1006/jcss.1998.1575>. URL: <http://www.sciencedirect.com/science/article/pii/S0022000098915756>.
- [SW01] Tibor Szabo and Emo Welzl. “Unique Sink Orientations of Cubes”. In: (Sept. 2001).
- [Sch04] Ingo A Schurr. “Unique sink orientations of cubes”. PhD thesis. ETH Zurich, 2004.
- [GR05] Bernd Gärtner and Leo Rüst. “Simple Stochastic Games and P-Matrix Generalized Linear Complementarity Problems”. In: *Fundamentals of Computation Theory*. Ed. by Maciej Liśkiewicz and Rüdiger Reischuk. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 209–220. ISBN: 978-3-540-31873-6.
- [DGP06] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. “The Complexity of Computing a Nash Equilibrium”. In: *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*. STOC '06. Seattle, WA, USA: Association for Computing Machinery, 2006, pp. 71–78. ISBN: 1595931341. DOI: 10.1145/1132516.1132527. URL: <https://doi.org/10.1145/1132516.1132527>.
- [Gol06] Oded Goldreich. “On Promise Problems: A Survey”. In: *Theoretical Computer Science: Essays in Memory of Shimon Even*. Ed. by Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 254–290. ISBN: 978-3-540-32881-0. DOI: 10.1007/11685654_12. URL: https://doi.org/10.1007/11685654_12.
- [GS06] Bernd Gärtner and Ingo Schurr. “Linear programming and unique sink orientations”. eng. In: SODA '06. Society for Industrial and Applied Mathematics, 2006, pp. 749–757. ISBN: 9780898716054.
- [Rüs07] Leonard Y Rüst. “The P-matrix linear complementarity problem: generalizations and specializations”. PhD thesis. ETH Zurich, 2007.

- [Cal+08] Chris Calabro et al. “The complexity of Unique k-SAT: An Isolation Lemma for k-CNFs”. In: *Journal of Computer and System Sciences* 74.3 (2008). Computational Complexity 2003, pp. 386–393. ISSN: 0022-0000. DOI: <https://doi.org/10.1016/j.jcss.2007.06.015>. URL: <http://www.sciencedirect.com/science/article/pii/S0022000007000827>.
- [GWJR+08] Bernd Gärtner, D Walter Jr, Leo Rüst, et al. “Unique sink orientations of grids”. In: *Algorithmica* 51.2 (2008), pp. 200–235.
- [AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. ISBN: 0521424267.
- [Cor+09] Thomas H. Corman et al. *Introduction to algorithms*. Third edition. MIT press, 2009.
- [EY10] Kousha Etesami and Mihalis Yannakakis. “On the Complexity of Nash Equilibria and Other Fixed Points”. eng. In: *SIAM journal on computing* 39.6 (2010), pp. 2531–2597. ISSN: 1095-7111.
- [FS10] Tobias Friedrich and Thomas Sauerwald. “The Cover Time of Deterministic Random Walks”. In: *Computing and Combinatorics*. Ed. by My T. Thai and Sartaj Sahni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 130–139. ISBN: 978-3-642-14031-0.
- [SZ10] William Steiger and Jihui Zhao. “Generalized Ham-Sandwich Cuts”. In: *Discrete & Computational Geometry* 44 (Oct. 2010), pp. 535–545. DOI: 10.1007/s00454-009-9225-8.
- [AV11] Ilan Adler and Sushil Verma. “The linear complementarity problem, Lemke algorithm, perturbation, and the complexity class PPAD”. In: (2011).
- [Bar11] Martin Barner. *Analysis. Band 2, Analysis II*. ger. 3. durchgesehene Auflage 1996. Reprint 2011. Berlin ;Boston: De Gruyter, 2011. ISBN: 9783110808896.
- [CF11] Krishnendu Chatterjee and Nathanaël Fijalkow. “A reduction from parity games to simple stochastic games”. In: *Electronic Proceedings in Theoretical Computer Science* 54 (2011), pp. 74–86. ISSN: 2075-2180. DOI: 10.4204/eptcs.54.6. URL: <http://dx.doi.org/10.4204/EPTCS.54.6>.
- [DP11] Constantinos Daskalakis and Christos Papadimitriou. “Continuous Local Search”. In: *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’11. San Francisco, California: Society for Industrial and Applied Mathematics, 2011, pp. 790–804.
- [BPR15] Nir Bitansky, Omer Paneth, and Alon Rosen. “On the Cryptographic Hardness of Finding a Nash Equilibrium”. In: Oct. 2015, pp. 1480–1498. DOI: 10.1109/FOCS.2015.94.
- [Doh+16] Jérôme Dohrau et al. “A zero-player graph game in $NP \cap coNP$ ”. In: *CoRR* abs/1605.03546 (2016). URL: <http://arxiv.org/abs/1605.03546>.
- [DTZ17] Constantinos Daskalakis, Christos Tzamos, and Manolis Zampetakis. “A Converse to Banach’s Fixed Point Theorem and its CLS Completeness”. In: *CoRR* abs/1702.07339 (2017). arXiv: 1702.07339. URL: <http://arxiv.org/abs/1702.07339>.
- [Fea+17] John Fearnley et al. “CLS: New Problems and Completeness”. In: *CoRR* abs/1702.06017 (2017). URL: <http://arxiv.org/abs/1702.06017>.

- [HY17] Pavel Hubáček and Eylon Yogev. “Hardness of Continuous Local Search: Query Complexity and Cryptographic Lower Bounds”. In: *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms*. 2017, pp. 1352–1371. DOI: 10.1137/1.9781611974782.88. eprint: <https://epubs.siam.org/doi/pdf/10.1137/1.9781611974782.88>. URL: <https://epubs.siam.org/doi/abs/10.1137/1.9781611974782.88>.
- [Fea+18] John Fearnley et al. “Unique End of Potential Line”. In: *CoRR* abs/1811.03841 (2018). URL: <http://arxiv.org/abs/1811.03841>.
- [Gär+18] Bernd Gärtner et al. “ARRIVAL: Next Stop in CLS”. In: *CoRR* abs/1802.07702 (2018). URL: <http://arxiv.org/abs/1802.07702>.
- [JS19] Florian Jarre and Josef Stoer. *Optimierung: Einführung in mathematische Theorie und Methoden*. ger. 2. Aufl. 2019. Masterclass. Berlin, Heidelberg: Springer Berlin Heidelberg, 2019. ISBN: 9783662588543.
- [CCM20] Man-Kwun Chiu, Aruni Choudhary, and Wolfgang Mulzer. “Computational Complexity of the α -Ham-Sandwich Problem”. In: *arXiv preprint arXiv:2003.09266* (2020).
- [Fea+20a] John Fearnley et al. “The Complexity of Gradient Descent: $\text{CLS} = \text{PPAD} \cap \text{PLS}$ ”. In: (2020). eprint: 2011.01929.
- [Fea+20b] John Fearnley et al. “Unique end of potential line”. In: *Journal of Computer and System Sciences* 114 (2020), pp. 1–35. ISSN: 0022-0000. DOI: <https://doi.org/10.1016/j.jcss.2020.05.007>. URL: <http://www.sciencedirect.com/science/article/pii/S0022000020300520>.
- [Kř+20] Jan Křetínský et al. “Comparison of Algorithms for Simple Stochastic Games”. In: *Electronic Proceedings in Theoretical Computer Science* 326 (2020), pp. 131–148. ISSN: 2075-2180. DOI: 10.4204/eptcs.326.9. URL: <http://dx.doi.org/10.4204/EPTCS.326.9>.
- [KS] Christian Karpfinger and Hellmuth Stachel. *Lineare Algebra*. ger. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 9783662613399.