ICALP 2026
July 7
London, UK

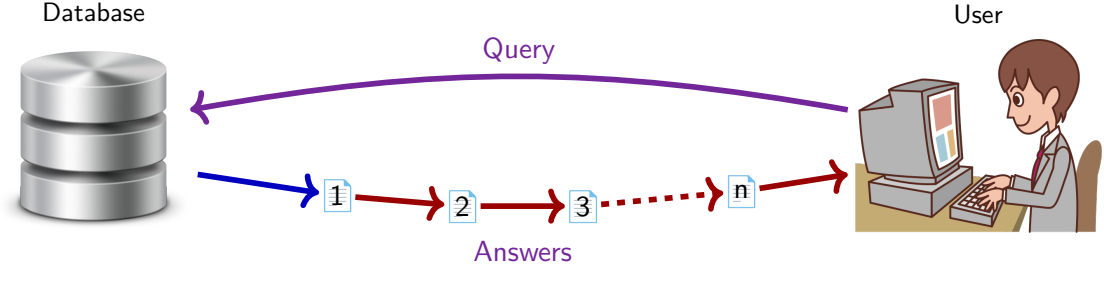
Gray Codes With Constant Delay and Constant Auxiliary Space

Antoine Amarilli [1]
Claire David [2]
Nadime Francis [2]
Victor Marsault [2]
Mikaël Monet [1]
Yann Strozecki [2,3]

1. Univ. Lille, Inria, CNRS, Centrale Lille, France
2. Univ. Gustave Eiffel, CNRS, France
3. Univ. de Versailles Saint-Quentin, France

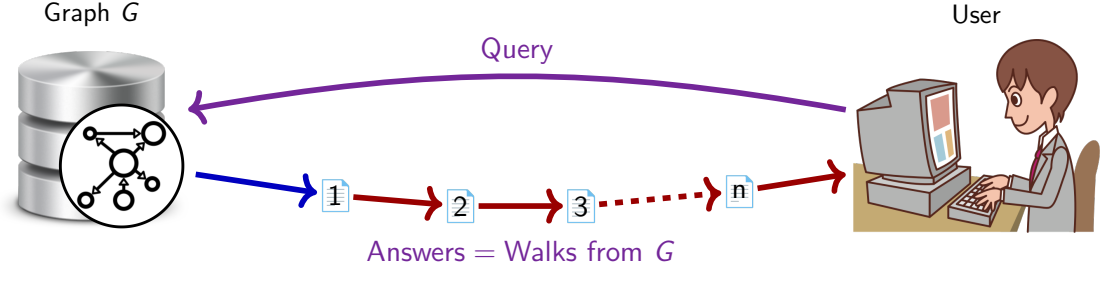
Motivation

From graph databases...



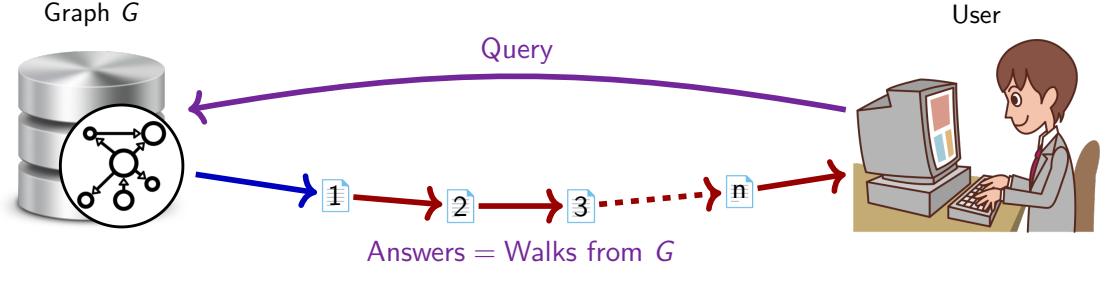
Motivation

From graph databases...



Motivation

From graph databases...

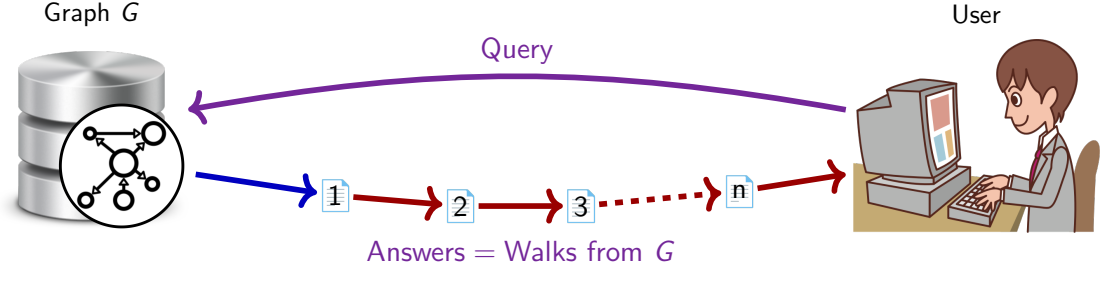


Situation:

- Some tasks have optimal solutions
- Can we reduce the output time w.r.t. the walk length ℓ ?

Motivation

From graph databases...

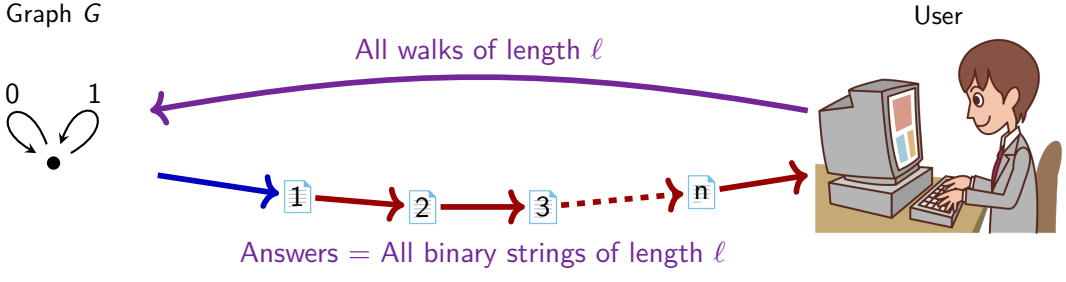


Situation:

- Some tasks have optimal solutions
- Can we reduce the output time w.r.t. the walk length ℓ ?

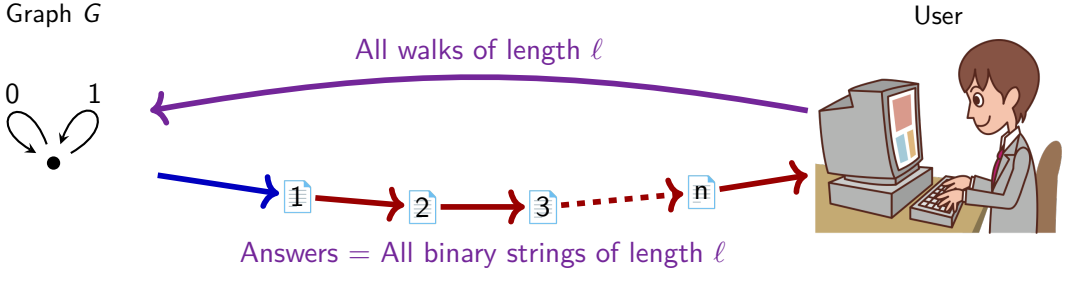
Our idea:

- Use Gray-code techniques
- I.e., output only the diff between walk with resources independent from ℓ


Graph G
User
All walks of length ℓ
0
1
1
2
3
n
Answers = All binary strings of length ℓ

Motivation

...to Gray codes



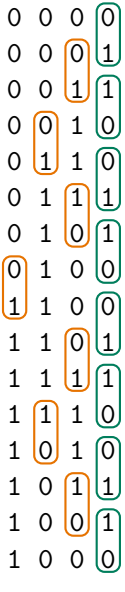
Our goal:

- Generate all binary strings of length ℓ...
- ...using resources independent from ℓ
  - Delay: time between consecutive output
  - Space (aside from the current output)

A **Gray code** defines
- for every length ℓ,
- a sequence of the 2^ℓ binary strings of length ℓ
- s.t. two successive strings differ by one bit.

Ex: Reflected Binary Gray Code (Gray 1947, Mutze 2023)
- At even steps, flip the rightmost bit
- At odd steps, flip the bit directly left of the rightmost 1.

RBGC



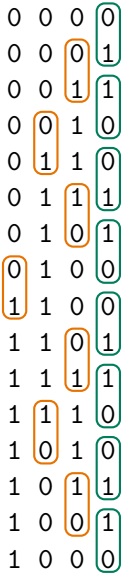
Constant Delay, Constant Auxiliary Space

A Gray code implementation has **constant delay** if
- the next bit to flip is computed in constant* time

* independent of ℓ

Ex: Reflected Binary Gray Code (Gray 1947, Mutze 2023)
- At even steps: flip the rightmost bit
- At odd steps: flip the bit directly left of the rightmost 1.
- Constant-delay implementation: maintain a linked list of 1's.

RBGC



Constant Delay, Constant Auxiliary Space

A Gray code implementation has **constant delay** if
- the next bit to flip is computed in constant* time

A Gray code implementation uses **constant auxiliary space** if
- the next bit to compute requires constant* memory, aside from the current binary string.

* independent of ℓ

Ex: Reflected Binary Gray Code (Gray 1947, Mutze 2023)

- At even steps: flip the rightmost bit
- At odd steps: flip the bit directly left of the rightmost 1.

- Constant-delay implementation: maintain a linked list of 1's.
- Constant-aux-space implementation: scan for the rightmost 1.

RBGC

Constant Delay, Constant Auxiliary Space

A Gray code implementation has **constant delay** if
- the next bit to flip is computed in constant* time

A Gray code implementation uses **constant auxiliary space** if
- the next bit to compute requires constant* memory, aside from the current binary string.

* independent of ℓ

Ex: Reflected Binary Gray Code (Gray 1947, Mutze 2023)
- At even steps: flip the rightmost bit
- At odd steps: flip the bit directly left of the rightmost 1.

- Constant-delay implementation: maintain a linked list of 1's.
- Constant-aux-space implementation: scan for the rightmost 1.
- **No implementation of RBGC can be both.**

RBGC

"Turing machine with a fixed-sized tape"

A **tape machine** consists of

- A tape of **fixed** size ℓ with endmarkers $\triangleright$ and $\triangleleft$
- Finitely many states
- Output states, that produce the tape in constant time
- A head that reads/writes in a window of size n

Transitions look like: $\cdots 010 \cdots \rightsquigarrow \cdots 011 \cdots$

"Turing machine with a fixed-sized tape"

A **tape machine** consists of

- A tape of **fixed** size ℓ with endmarkers $\triangleright$ and $\triangleleft$
- Finitely many states
- Output states, that produce the tape in constant time
- A head that reads/writes in a window of size n

Transitions look like: $\cdots 010 \cdots \rightsquigarrow \cdots 011 \cdots$

Goal:

- Produce each binary string exactly once
- Description independent of the length ℓ
- The machine must be constant-delay

A **deque machine** is similar except:

- It works on **deque** of **fixed** size ℓ
- Transitions push/pop k characters on either ends

Transitions look like: $10\cdots00 \rightsquigarrow 0\cdots001$

A **deque machine** is similar except:

- It works on **deque** of **fixed** size ℓ
- Transitions push/pop k characters on either ends

Transitions look like: $10\cdots00 \rightsquigarrow 0\cdots001$

⚠ Circular rotations are constant-time ⚠

A **deque machine** is similar except:

- It works on **deque** of **fixed** size ℓ
- Transitions push/pop k characters on either ends

Transitions look like: $10\cdots00 \rightsquigarrow 0\cdots001$

⚠ Circular rotations are constant-time ⚠

Why deque machines ?

- It generalizes better to walks (our motivation)
  $\rightarrow$ Edge substitution in the middle is not very useful

General idea:
- Distribute strings of length ℓ in a complete binary tree
- Perform a DFS of that tree

Issue:
- How to keep delay constant?

→
- Classical *"Even-odd trick"*

Issue for tape machines:
- 3 bit flips between outputs
- $3 \neq 1$

→
- Encode parity on the tape
- Smart choice of first child

Issue for deque machines:
- How to stop?
- Obscure parity swap

→
Either:
- Perform two DFS's
- Use *"lookahead"*

Technique overview

General idea:
- Distribute strings of length ℓ in a complete binary tree
- Perform a DFS of that tree

Issue:
- How to keep delay constant?

→
- Classical *"Even-odd trick"*

Issue for tape machines:
- 3 bit flips between outputs
- $3 \neq 1$

→
- Encode parity on the tape
- Smart choice of first child

Issue for deque machines:
- How to stop?
- Obscure parity swap

→ Either:
- Perform two DFS's
- Use *"lookahead"*

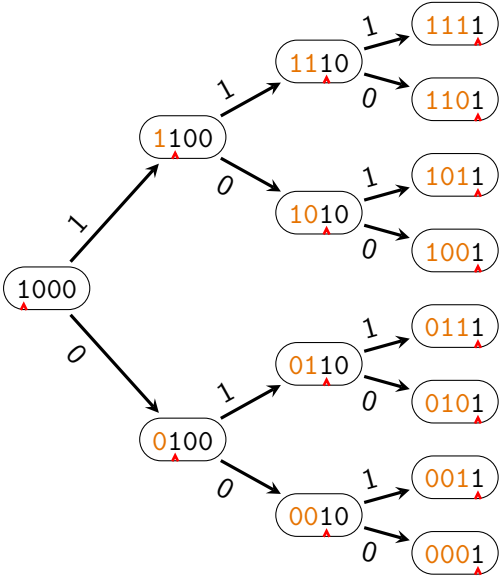
General idea ($\ell = 4$ in the example)

Building the tree

The string 0000 is treated separately

Other binary strings encode nodes
in the complete binary tree of depth $\ell - 1$

- Write each string as $u\underset{\wedge}{1}0^*$
- Place the head $\underset{\wedge}{\ }$ on the rightmost 1
- It encodes the node reached by reading u from the root

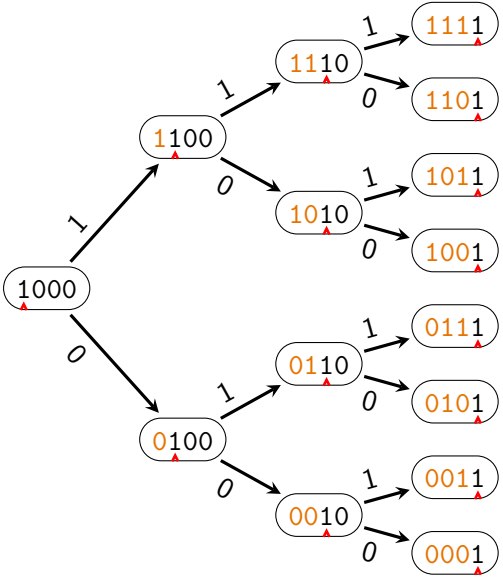


General idea ($\ell = 4$ in the example)

Navigating the tree with local transformations:

Reminder: $u\underset{\uparrow}{1}0^k$ reached by u from the root
$\triangleright$ and $\triangleleft$ are end markers

- Detect the root: $\triangleright\underset{\uparrow}{1}\cdots$
- Detect leaves: $\cdots\underset{\uparrow}{1}\triangleleft$



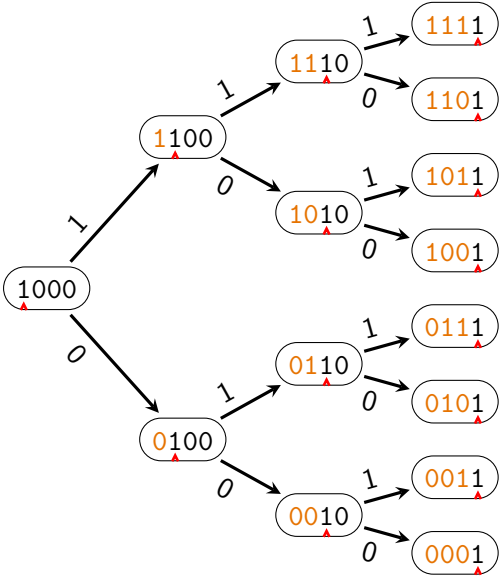
General idea ($\ell = 4$ in the example)

Navigating the tree with local transformations:

Reminder: $u\underset{\uparrow}{1}0^k$ reached by u from the root
$\triangleright$ and $\triangleleft$ are end markers

- Detect the root: $\triangleright\underset{\uparrow}{1}\cdots$
- Detect leaves: $\cdots\underset{\uparrow}{1}\triangleleft$
- Going from a node to its children:

$$\cdots\underset{\uparrow}{1}0\cdots \rightsquigarrow \begin{cases} \cdots 0\underset{\uparrow}{1}\cdots \\ \cdots \underset{\uparrow}{1}\underset{\uparrow}{1}\cdots \end{cases}$$

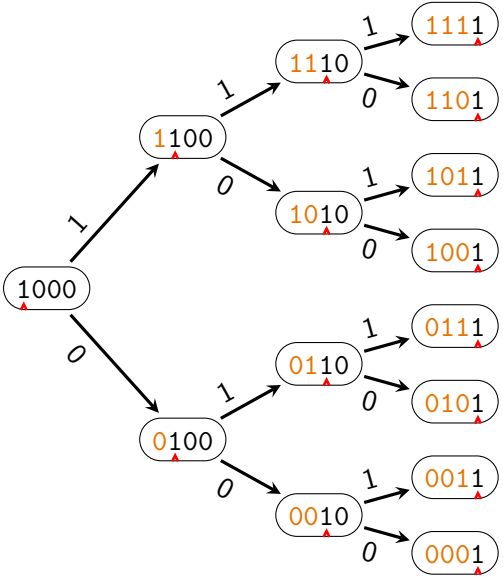


General idea ($\ell = 4$ in the example)

Navigating the tree with local transformations:

Reminder: $u\underset{\uparrow}{1}0^k$ reached by u from the root
$\triangleright$ and $\triangleleft$ are end markers

- Detect the root: $\triangleright\underset{\uparrow}{1}\cdots$
- Detect leaves: $\cdots\underset{\uparrow}{1}\triangleleft$
- Going from a node to its children:
$$\cdots\underset{\uparrow}{1}0\cdots \rightsquigarrow \begin{cases} \cdots 0\underset{\uparrow}{1}\cdots \\ \cdots \underset{\uparrow}{1}\underset{\uparrow}{1}\cdots \end{cases}$$
- Going from a node to its parent:
$$\left.\begin{array}{l} \cdots 0\underset{\uparrow}{1}\cdots \\ \cdots \underset{\uparrow}{1}\underset{\uparrow}{1}\cdots \end{array}\right\} \rightsquigarrow \cdots\underset{\uparrow}{1}\underset{\uparrow}{0}\cdots$$



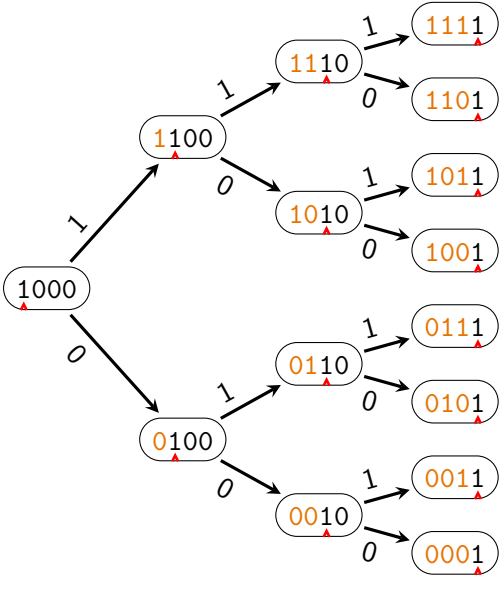
General idea ($\ell = 4$ in the example)

Navigating the tree with local transformations:

Reminder: $u\underset{\uparrow}{1}0^k$ reached by u from the root
$\triangleright$ and $\triangleleft$ are end markers

- Detect the root: $\triangleright\underset{\uparrow}{1}\cdots$
- Detect leaves: $\cdots\underset{\uparrow}{1}\triangleleft$
- Going from a node to its children:
$$\cdots\underset{\uparrow}{1}0\cdots \rightsquigarrow \begin{cases} \cdots 0\underset{\uparrow}{1}\cdots \\ \cdots \underset{\uparrow}{1}\underset{\uparrow}{1}\cdots \end{cases}$$
- Going from a node to its parent:
$$\left.\begin{array}{l} \cdots 0\underset{\uparrow}{1}\cdots \\ \cdots \underset{\uparrow}{1}\underset{\uparrow}{1}\cdots \end{array}\right\} \rightsquigarrow \cdots\underset{\uparrow}{1}0\cdots$$

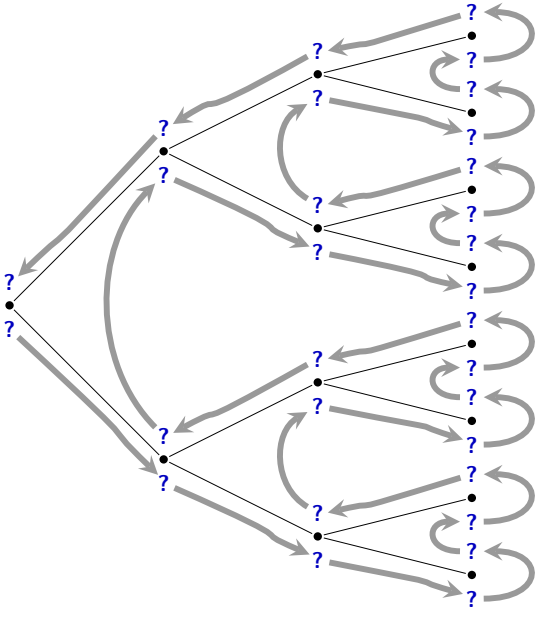
$\Rightarrow$ DFS implementable in a tape machine



Even-odd trick

Sekanina 1960; Karaganis 1968; Feder 1994

During a DFS, we want to
- output **once** at each node
- while each node is visited **twice**



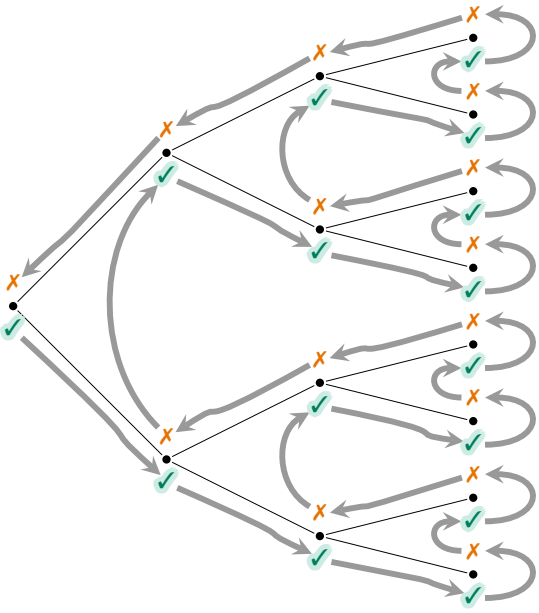
Even-odd trick

Sekanina 1960; Karaganis 1968; Feder 1994

During a DFS, we want to
- output **once** at each node
- while each node is visited **twice**

Naive approach

- Always output when entering subtrees



Even-odd trick

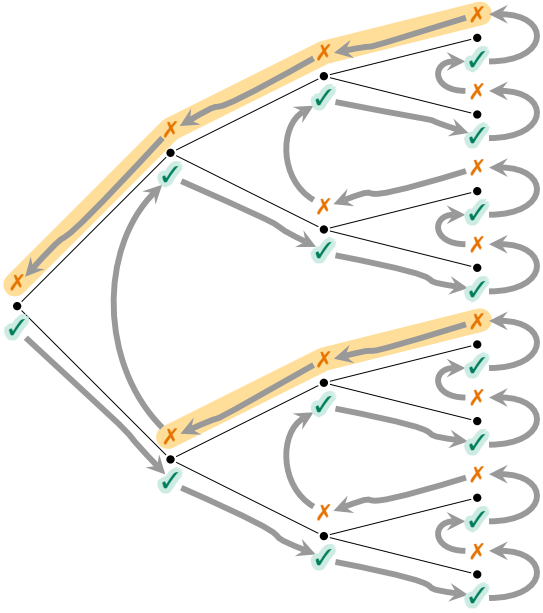
Sekanina 1960; Karaganis 1968; Feder 1994

During a DFS, we want to
- output **once** at each node
- while each node is visited **twice**

Naive approach

- Always output when entering subtrees

$\Rightarrow$ Long segments without output
$\Rightarrow$ In our case, linear delay



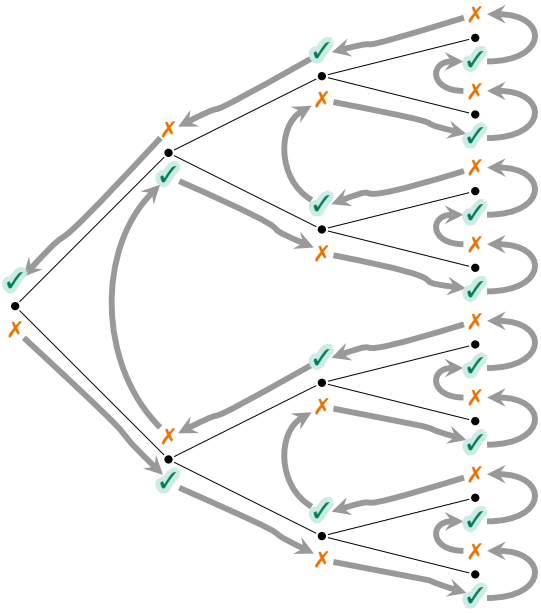
Even-odd trick

Sekanina 1960; Karaganis 1968; Feder 1994

During a DFS, we want to
- output **once** at each node
- while each node is visited **twice**

Even-odd trick

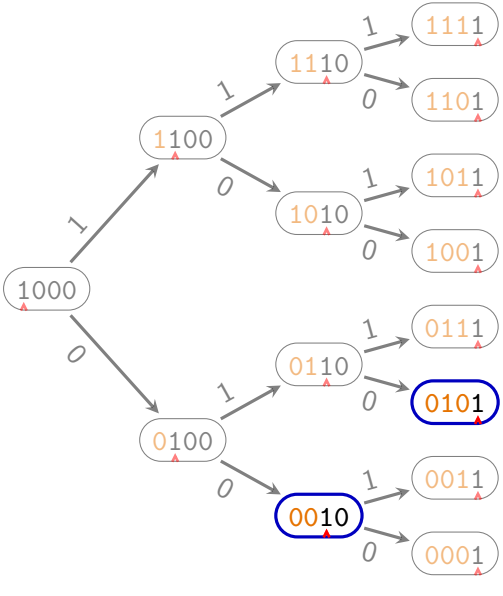
- **Odd** depth $\rightarrow$ when **entering** subtrees
- **Even** depth $\rightarrow$ when **leaving** subtrees



Not a Gray code (yet)

- Some navigation steps flip 2 bits
- The even-odd trick causes 2 steps between outputs

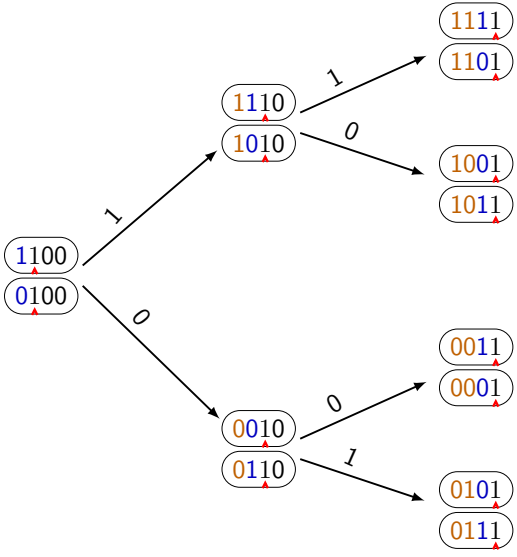
$\Rightarrow$ 3 flipped bits between some outputs



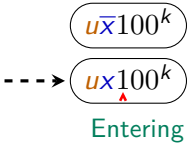
Writing the parity on the tape ($\ell = 4$ in the example)

We build the complete binary tree of depth $\ell - 2$

- Two strings in each node: $u\ 1\ 1\ 0^k$ / $u\ 0\ 1\ 0^k$
- Path from the root
- Parity bit

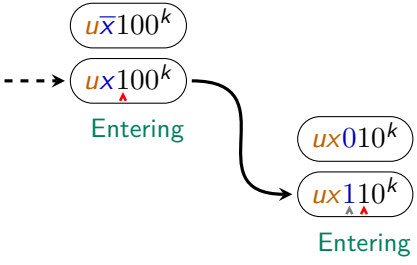


How the final tape machine works



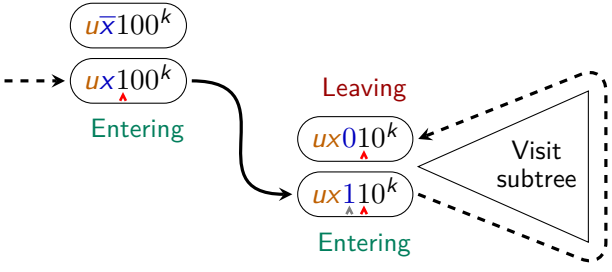
where u: path from the root
x: parity bit

How the final tape machine works



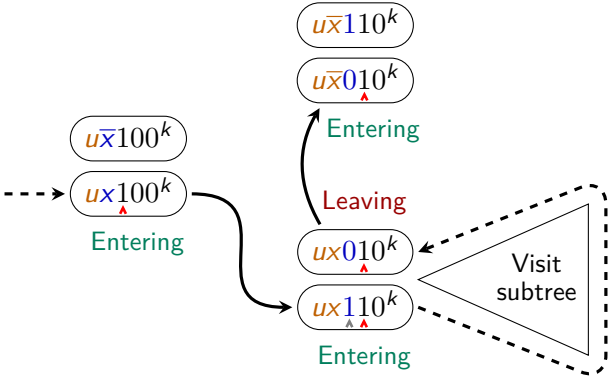
where u: path from the root
x: parity bit

How the final tape machine works



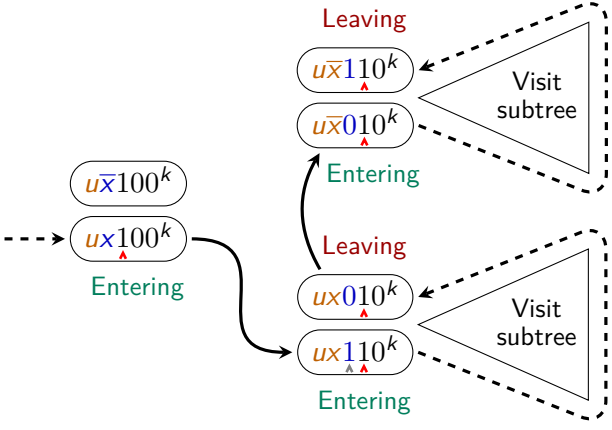
where u: path from the root
x: parity bit

How the final tape machine works



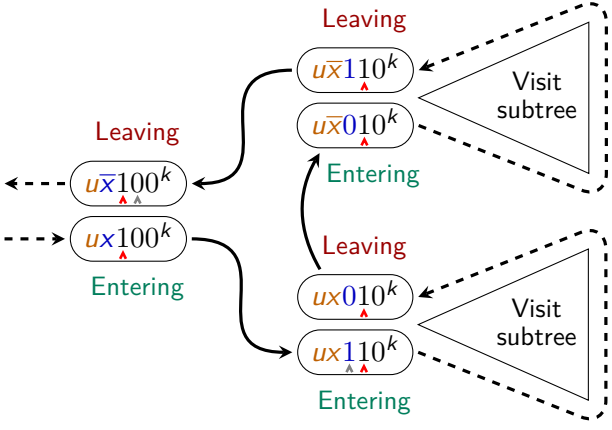
where u: path from the root
x: parity bit

How the final tape machine works



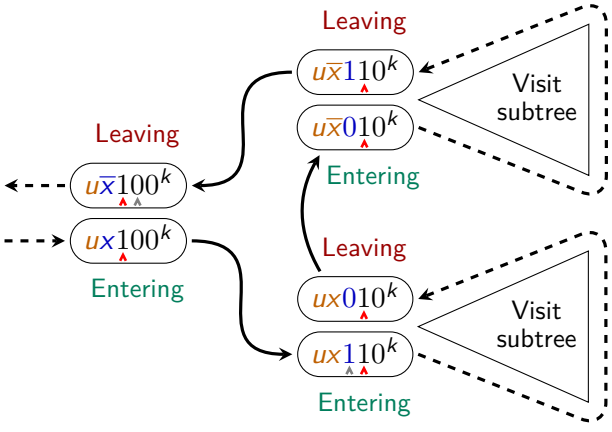
where u: path from the root
x: parity bit

How the final tape machine works



where u: path from the root
x: parity bit

How the final tape machine works



where u: path from the root
x: parity bit

The tape machine is simple: 2 states (Entering and Leaving) and a window of length 4

Contributions

1. Gray code implemented on a tape machine
2. Gray-like code implemented on a deque machine

Ongoing work

- Generalize 1 to non-binary alphabets
- Generalize 2 to strongly connected graphs

Future work

- Generalize 2 to all graphs
- Generalize 2 to more complex queries

Brodal, Greve, Pandey, and Satti (2014). "Integer representations towards efficient counting in the bit probe model". In: *Journal of Discrete Algorithms* 26. DOI: 10.1016/j.jda.2013.11.001.

Feder (1994). "Network flow and 2-satisfiability". In: *Algorithmica* 11.3. DOI: 10.1007/BF01240738.

Karaganis (1968). "On the cube of a graph". In: *Canadian Mathematical Bulletin* 11.2. DOI: 10.4153/CMB-1968-037-0.

Mütze (2023). "Combinatorial Gray Codes — an Updated Survey". In: *The Electronic Journal of Combinatorics* Dynamic Surveys. DOI: 10.37236/11023.

Sac Himelfarb and Schwartz (2025). "Improved Constructions of Skew-Tolerant Gray Codes". In: *IEEE Transactions on Information Theory* 71.10. DOI: 10.1109/TIT.2025.3592490.

Sekanina (1960). "On an ordering of the set of vertices of a connected graph". In: *Publ. Fac. Sci. Univ. Brno* 412.