WEPA

5 November 2026
CIRM, Marseille, France

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]
Tom Yungmann [2]

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





Gray codes

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, Mütze 2023)
- At even steps, flip the rightmost bit
- At odd steps, flip the bit directly left of the rightmost 1.

RBGC

0 0 0 0
0 0 0 1
0 0 1 1
0 0 1 0
0 1 1 0
0 1 1 1
0 1 0 1
0 1 0 0
1 1 0 0
1 1 0 1
1 1 1 1
1 1 1 0
1 0 1 0
1 0 1 1
1 0 0 1
1 0 0 0

Constant Delay, Constant Auxiliary Space

A Gray code implementation has **constant delay** if
- computing the next bit to flip requires 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 impl. of RBGC: maintain a linked list of 1's.

RBGC

0 0 0 0
0 0 0 1
0 0 1 1
0 0 1 0
0 1 1 0
0 1 1 1
0 1 0 1
0 1 0 0
1 1 0 0
1 1 0 1
1 1 1 1
1 1 1 0
1 0 1 0
1 0 1 1
1 0 0 1
1 0 0 0

Constant Delay, Constant Auxiliary Space

A Gray code implementation has **constant delay** if
- computing the next bit to flip requires constant* time

A Gray code implementation uses **constant auxiliary space** if
- computing the next bit to flip 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 impl. of RBGC: maintain a linked list of 1's.
- Constant-aux.-space impl. of RBGC: scan for the rightmost 1.

RBGC

0 0 0 0
0 0 0 1
0 0 1 1
0 0 1 0
0 1 1 0
0 1 1 1
0 1 0 1
0 1 0 0
1 1 0 0
1 1 0 1
1 1 1 1
1 1 1 0
1 0 1 0
1 0 1 1
1 0 0 1
1 0 0 0

Constant Delay, Constant Auxiliary Space

A Gray code implementation has **constant delay** if
- computing the next bit to flip requires constant* time

A Gray code implementation uses **constant auxiliary space** if
- computing the next bit to flip 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 impl. of RBGC: maintain a linked list of 1's.
- Constant-aux.-space impl. of RBGC: scan for the rightmost 1.
- **No implemention of RBGC is both!!**

RBGC

0 0 0 0
0 0 0 1
0 0 1 1
0 0 1 0
0 1 1 0
0 1 1 1
0 1 0 1
0 1 0 0
1 1 0 0
1 1 0 1
1 1 1 1
1 1 1 0
1 0 1 0
1 0 1 1
1 0 0 1
1 0 0 0

Our computational model: Tape machines

"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
- A head that reads/writes in a window of size n
- After a transition, the tape is produced in constant time

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

Our computational model: Tape machines

"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
- A head that reads/writes in a window of size n
- After a transition, the tape is produced in constant time

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

Why will it be...

- Constant aux. space? Machine is independent of ℓ
- Constant-delay? Single transition between outputs

A Gray code is ***k*-skew-tolerant** if

- two bits flipped consecutively are at distance at most k

$\rightarrow$ RBGC is not skew-tolerant

RBGC

0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

A Gray code is ***k*-skew-tolerant** if
- two bits flipped consecutively are at distance at most k

$\rightarrow$ RBGC is not skew-tolerant

$\rightarrow$ Implementable in a tape machine $\implies$ skew tolerant

RBGC

0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

A Gray code is ***k*-skew-tolerant** if
- two bits flipped consecutively are at distance at most k

$\rightarrow$ RBGC is not skew-tolerant
$\rightarrow$ Implementable in a tape machine $\implies$ skew tolerant

Theorem (Sac Himelfarb, Schwartz 2025)
- There exists a 2-skew-tolerant Gray code

RBGC

0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

A Gray code is ***k*-skew-tolerant** if
- two bits flipped consecutively are at distance at most k

→ RBGC is not skew-tolerant
→ Implementable in a tape machine $\implies$ skew tolerant

Theorem (Sac Himelfarb, Schwartz 2025)
- There exists a 2-skew-tolerant Gray code

→ Built upon a 3-skew-tolerant Gray code called A
→ Inductive definitions

RBGC

0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

A Gray code is ***k*-skew-tolerant** if
- two bits flipped consecutively are at distance at most k

→ RBGC is not skew-tolerant
→ Implementable in a tape machine $\implies$ skew tolerant

Theorem (Sac Himelfarb, Schwartz 2025)
- There exists a 2-skew-tolerant Gray code

→ Built upon a 3-skew-tolerant Gray code called A
→ Inductive definitions

- Our code is 3-skew-tolerant and related to A.
- A is actually implementable in a tape machine

RBGC

0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

Our code, 3-skew-tolerant and implementable on a tape machine

0	0	0	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	1	0	0
0	1	1	1	1	0
0	1	1	1	1	1
0	1	1	1	0	1
0	1	1	0	0	1
0	1	1	0	1	1
0	1	1	0	1	0
0	1	0	0	1	0
0	1	0	0	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	0	1	1	1
0	1	0	1	1	0

0	1	0	1	0	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	0	1	1	1
0	0	0	1	0	1
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	0	1	0
0	0	1	0	1	0
0	0	1	0	1	1
0	0	1	0	0	1
0	0	1	1	0	1
0	0	1	1	1	1
0	0	1	1	1	0
0	0	1	1	0	0
0	0	1	0	0	0

1	0	1	0	0	0
1	0	1	1	0	0
1	0	1	1	1	0
1	0	1	1	1	1
1	0	1	1	0	1
1	0	1	0	0	1
1	0	1	0	1	1
1	0	1	0	1	0
1	0	0	0	1	0
1	0	0	0	1	1
1	0	0	0	0	1
1	0	0	1	0	1
1	0	0	1	1	1
1	0	0	1	1	0
1	0	0	1	0	0
1	1	0	1	0	0

1	1	0	1	1	0
1	1	0	1	1	1
1	1	0	1	0	1
1	1	0	0	0	1
1	1	0	0	1	1
1	1	0	0	1	0
1	1	1	0	1	0
1	1	1	0	1	1
1	1	1	0	0	1
1	1	1	1	0	1
1	1	1	1	1	1
1	1	1	1	1	0
1	1	1	1	0	0
1	1	1	0	0	0
1	1	0	0	0	0
1	0	0	0	0	0

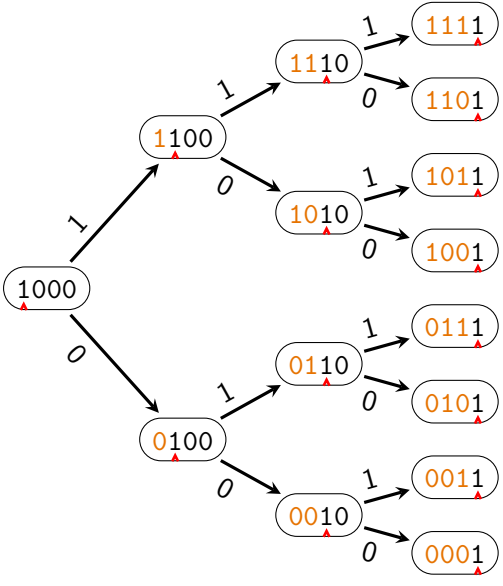
Basic idea ($\ell = 4$ in the example)

Building the tree

The string 0000 is treated separately

In the binary tree of depth $\ell - 1$:

- Write each string as $u\hat{1}0^*$ (head $\hat{}$ on the rightmost 1)
- Put $u\hat{1}0^*$ in the node reached by reading u from the root

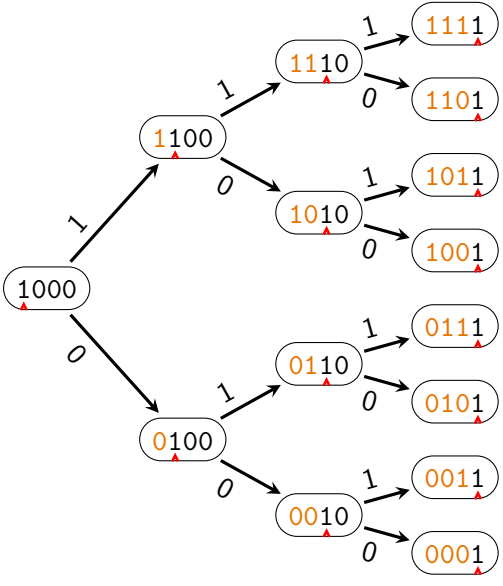


Basic idea ($\ell = 4$ in the example)

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: $\cdots\underset{\uparrow}{1}\triangleleft$



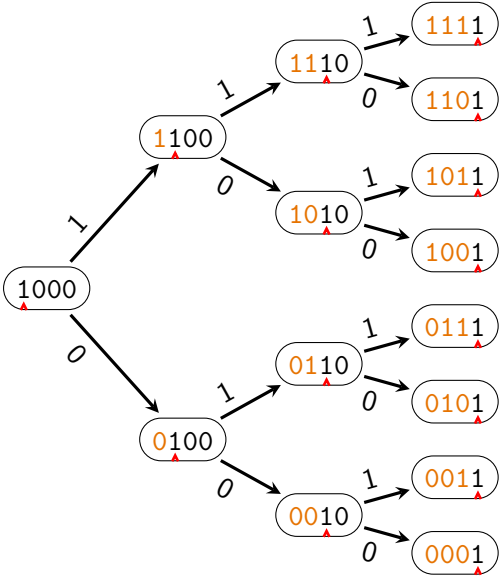
Basic idea ($\ell = 4$ in the example)

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:

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

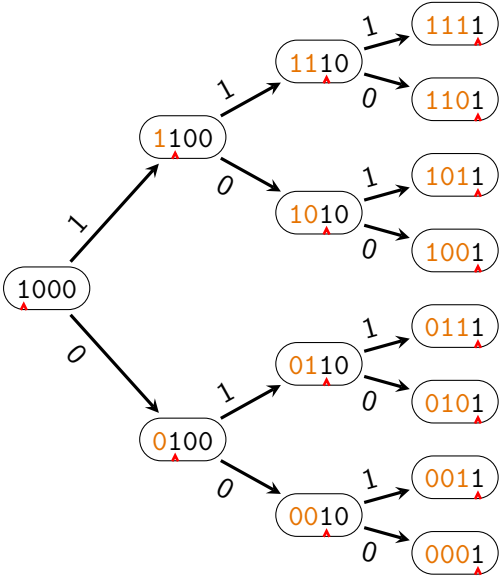


Basic idea ($\ell = 4$ in the example)

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:
$$\left.\begin{array}{l} \cdots 0\underset{\uparrow}{1}\cdots \\ \cdots \underset{\uparrow}{1}\underset{\uparrow}{1}\cdots \end{array}\right\} \rightsquigarrow \cdots\underset{\uparrow}{1}\underset{\uparrow}{0}\cdots$$



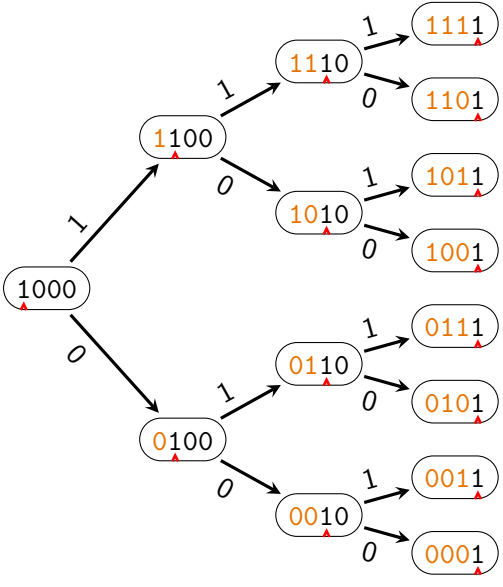
Basic idea ($\ell = 4$ in the example)

Navigating the tree with local transformations:

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

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

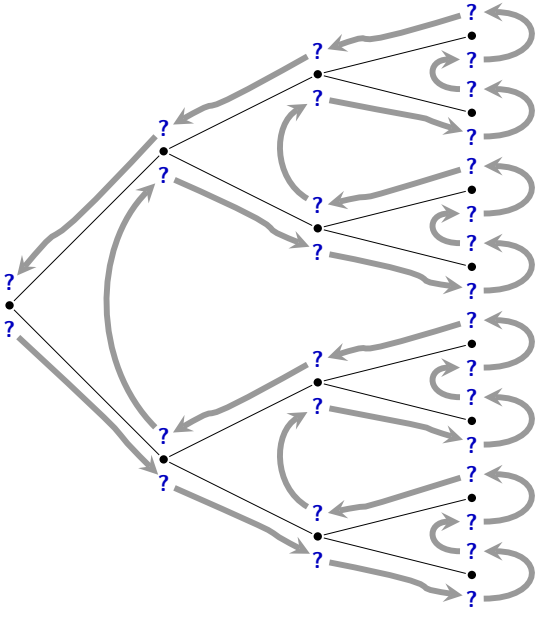
$\Rightarrow$ DFS implementable in a tape machine



Even-odd trick

Sekanina 1960; Karaganis 1968; Feder 1994

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



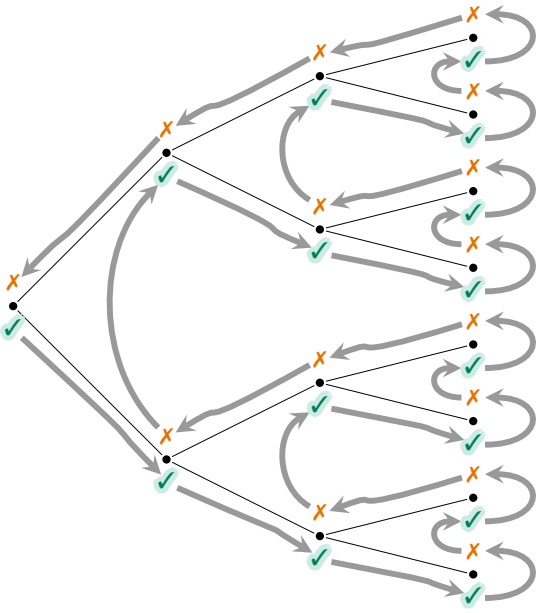
Even-odd trick

Sekanina 1960; Karaganis 1968; Feder 1994

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

Naive approach

- Always output when entering subtrees



Even-odd trick

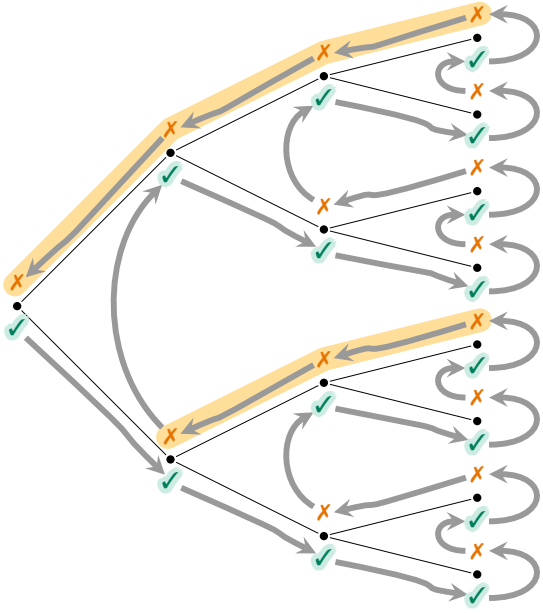
Sekanina 1960; Karaganis 1968; Feder 1994

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

Naive approach

- Always output when entering subtrees

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



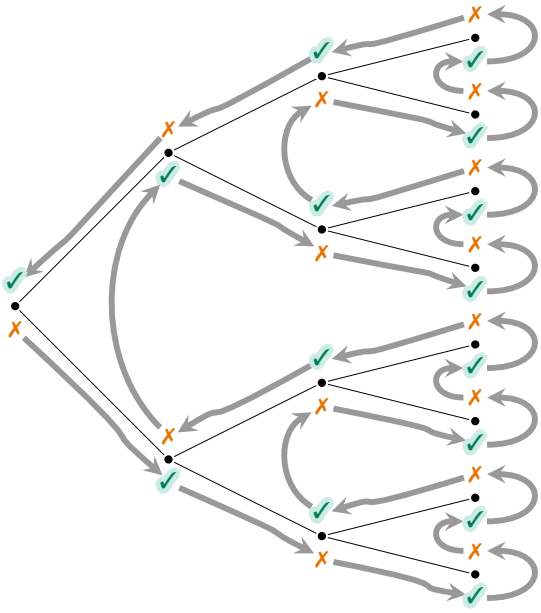
Even-odd trick

Sekanina 1960; Karaganis 1968; Feder 1994

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

Even-odd trick

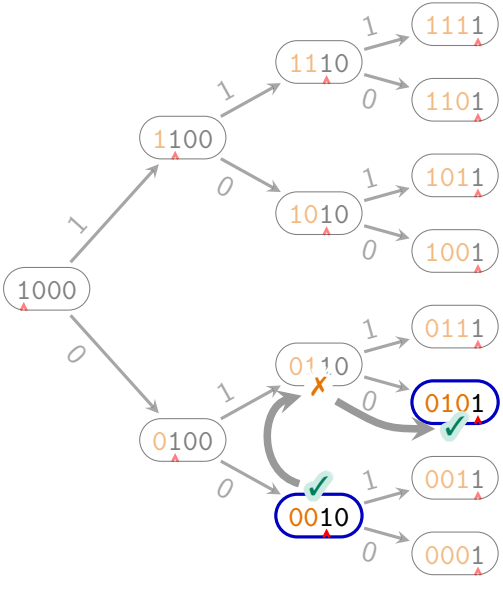
- **Odd** depth → when **entering** subtrees
- **Even** depth → when **leaving** subtrees



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



General idea:

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

Issue 1:

- Duplicates

Solution:

- Classical *"Even-odd trick"*

Issue 2:

- Up to 3 bit flips

Solution:

- Write depth parity on the tape
- Smart choice of first child

Generalization to b-ary Gray Codes

For any $b \geq 2$, a **b-ary Gray code** defines

- for every length ℓ,
- a sequence of the b^ℓ strings of length ℓ over $\{0, \ldots, b-1\}$
- s.t. two successive strings differ by one elementary operation.

Generalization to b-ary Gray Codes

For any $b \geq 2$, a **b-ary Gray code** defines

- for every length ℓ,
- a sequence of the b^ℓ strings of length ℓ over $\{0, \ldots, b-1\}$
- s.t. two successive strings differ by one elementary operation.

Theorem (Yungmann 2026, unpublished):

For every $b \geq 3$, there exists a b-ary Gray code

- implemented on a tape machine
- elementary operations are: $\pm 1 \mod b$

For any $b \geq 2$, a **b-ary Gray code** defines

- for every length ℓ,
- a sequence of the b^ℓ strings of length ℓ over $\{0, \ldots, b-1\}$
- s.t. two successive strings differ by one elementary operation.

Theorem (Yungmann 2026, unpublished):

For every $b \geq 3$, there exists a b-ary Gray code

- implemented on a tape machine
- elementary operations are: $\pm 1 \mod b$

Ideas

- Add "waiting" states to browse through $\{0, \ldots, b-1\}$
- Alternate visit order: odd child $0 \to b-1$, even child $b-1 \to 0$
- No parity bit $\to$ 2-skew-tolerant

- There are b-ary Gray codes implementable in **tape machines**

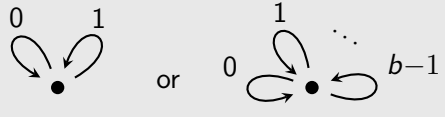
- There are b-ary Gray codes implementable in **tape machines**

Our original motivation:

- Graph databases
- Enumerating walks satisfying constraints

$\rightarrow$ Tape machines do not generalize

Here: output all walks of length ℓ in



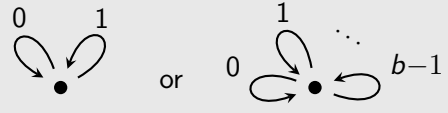
- There are b-ary Gray codes implementable in **tape machines**

Our original motivation:

- Graph databases
- Enumerating walks satisfying constraints

$\rightarrow$ Tape machines do not generalize

Here: output all walks of length ℓ in



In the paper we also:

- Use **deque machines** (no head, push/pop on either end)
- Show there are Gray-like codes implementable on a **deque** machine

Ongoing and future work:

- Generalize to all graphs
- Consider more complex queries

Thank you for your attention!



Navigable Outline

1. Title page
2. Gray code
3. Constant delay, Constant auxiliary space
4. Tape machines
5. Skew-tolerant codes
6. Our code
7. Building the tree
8. Navigating the tree
9. Even-odd trick
10. Not a Gray code yet
11. Technique overview
12. Generalization to b-ary Gray Codes
13. Conclusion and perspectives
14. Thanks !

16. Bibliography
17. DAT/bit-probe model
18. Inductive definitions of Gray codes
19. Deque machines
20. Writing the parity on the tape
21. Induction for $b = 2$
22. Induction for odd $b > 2$

Bibliography I

Amarilli, David, Francis, Marsault, Monet, and Strozecki (2026). "Gray Codes With Constant Delay and Constant Auxiliary Space". In: *ICALP*. DOI: 10.4230/LIPIcs.ICALP.2026.160.

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.

Yungmann (2026). *B-ary Gray Codes Implemented on Tape Machines*. Undergraduate Internship Report. École Normale Supérieure de Lyon.

We're **not** using the DAT/bit-probe model

Codes implemented in the DAT model

- DAT = Decision Assignment Tree
- Based on circuits: one circuit per length ℓ.
  $\Rightarrow$ Non-uniform
- Free access to the full word
  $\Rightarrow$ Non-local
- Cannot be constant-delay
- Important result: Log lower-bound on circuit depth (See Brodal et al. 2014)

RBGC

$$G_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$G_{\ell+1} = \left[\begin{array}{c|c} 0 & \\ \vdots & G_\ell \\ 0 & \\ \hline 1 & \\ \vdots & \overleftarrow{G_\ell} \\ 1 & \end{array}\right]$$

Code A of Sac Himelfarb and Schwartz

$$A_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$A_{\ell+1} = \left[\begin{array}{c|c} 0 \cdots 0 & 0 \\ \hline & 1 \\ A_\ell & \vdots \\ & 1 \\ \hline \overleftarrow{A_\ell[1:]} & 0 \\ & \vdots \\ & 0 \end{array}\right]$$

Our code

$$T_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$T_{\ell+1} = \left[\begin{array}{c|c} 0 & 0 \cdots 0 \\ \hline 0 & \\ \vdots & \overleftarrow{T_\ell[1:]} \\ 0 & \\ \hline 1 & \\ \vdots & T_\ell[1:] \\ 1 & \\ \hline 1 & 0 \cdots 0 \end{array}\right]$$

RBGC

$$G_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$G_{\ell+1} = \left[\begin{array}{c|c} 0 & 0\cdots0 \\ \hline 0 & \\ \vdots & G_\ell[1:] \\ 0 & \\ \hline 1 & \\ \vdots & \overleftarrow{G_\ell[1:]} \\ 1 & \\ \hline 1 & 0\cdots0 \end{array}\right]$$

Code A of Sac Himelfarb and Schwartz

$$A_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$A_{\ell+1} = \left[\begin{array}{c|c} 0\cdots0 & 0 \\ \hline 0\cdots0 & 1 \\ \hline & 1 \\ A_\ell[1:] & \vdots \\ & 1 \\ \hline & 0 \\ \overleftarrow{A_\ell[1:]} & \vdots \\ & 0 \end{array}\right]$$

Our code

$$T_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$T_{\ell+1} = \left[\begin{array}{c|c} 0 & 0\cdots0 \\ \hline 0 & \\ \vdots & \overleftarrow{T_\ell[1:]} \\ 0 & \\ \hline 1 & \\ \vdots & T_\ell[1:] \\ 1 & \\ \hline 1 & 0\cdots0 \end{array}\right]$$

A **deque machine** is similar except:

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

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

⚠ Circular rotations are constant-time ⚠

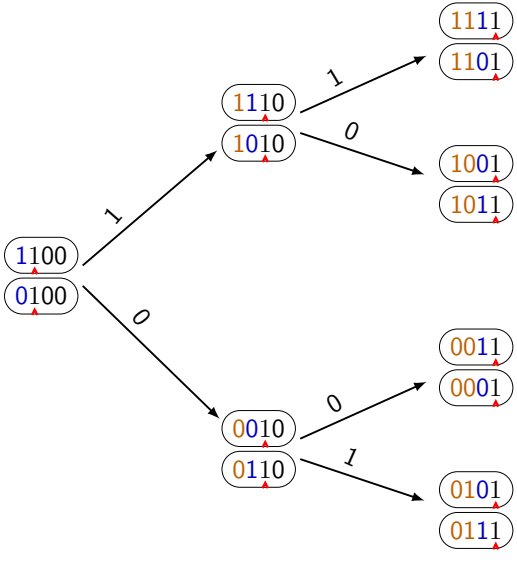
Why deque machines ?

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

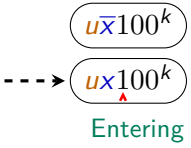
Writing the parity on the tape ($\ell = 4$ in the example)

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

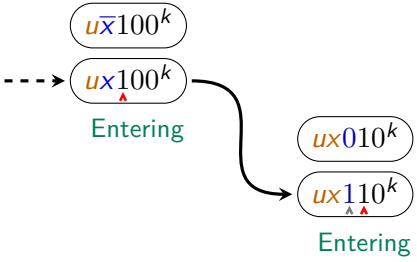


How the final tape machine works for $b = 2$



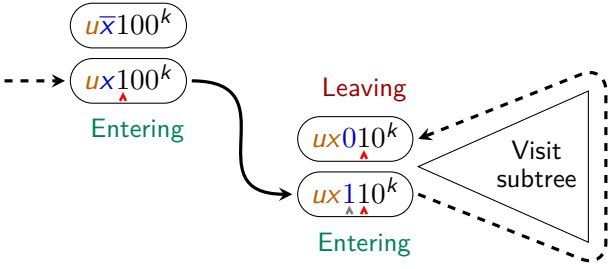
where u: path from the root
x: parity bit

How the final tape machine works for $b = 2$



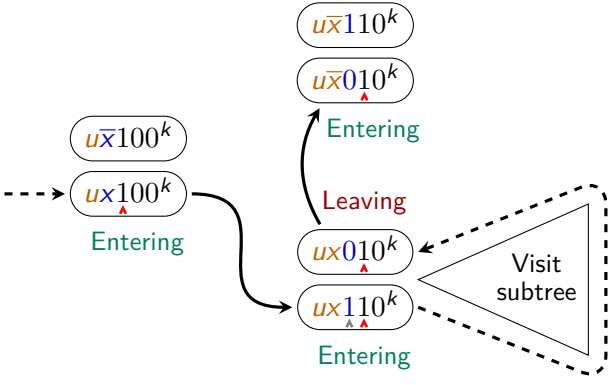
where u: path from the root
x: parity bit

How the final tape machine works for $b = 2$



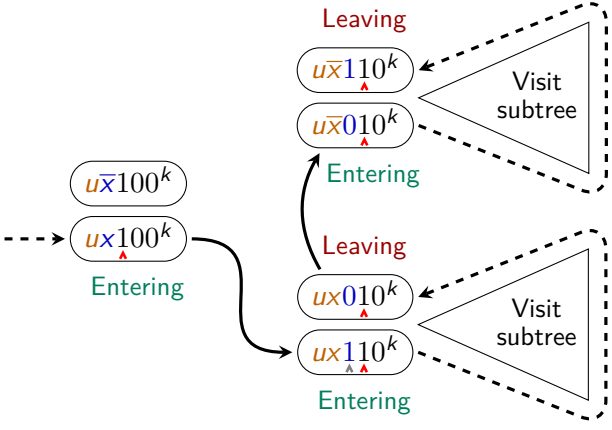
where u: path from the root
x: parity bit

How the final tape machine works for $b = 2$



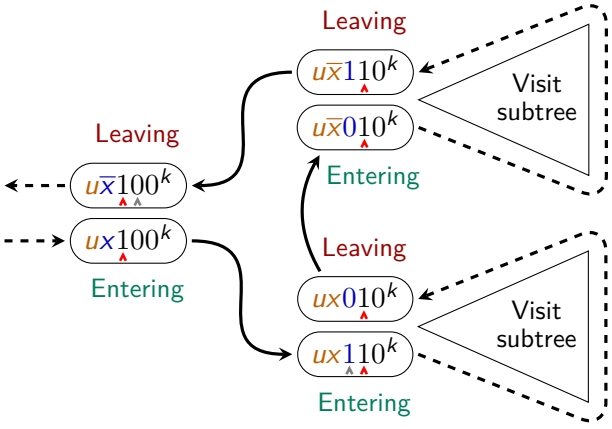
where u: path from the root
x: parity bit

How the final tape machine works for $b = 2$



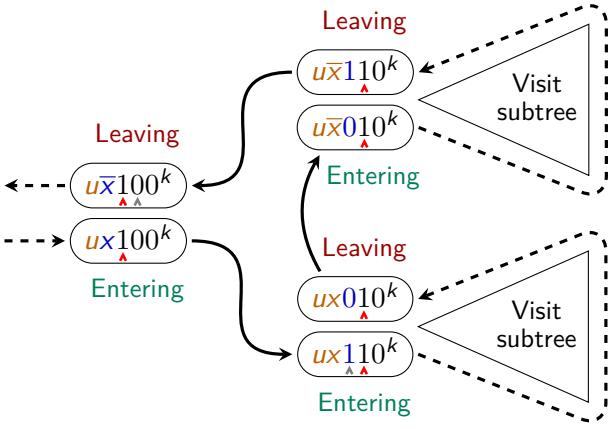
where u: path from the root
x: parity bit

How the final tape machine works for $b = 2$



where u: path from the root
x: parity bit

How the final tape machine works for $b = 2$

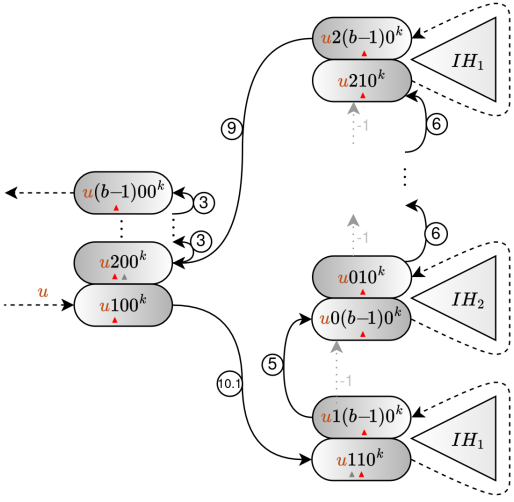


where u: path from the root
x: parity bit

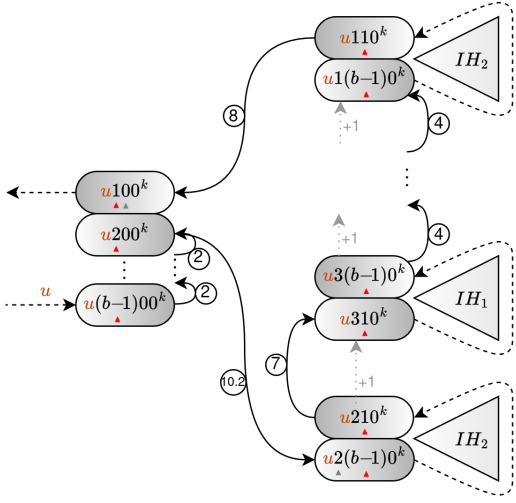
The tape machine is simple: 2 states (Entering and Leaving) and a window of length 4

Generalization to b-ary Gray Codes (2)

Induction for odd b's (Yungmann 2026)



Induction Hypothesis 1 (IH1)



Induction Hypothesis 2 (IH2)