Internet Initiative Japan

Recursive Lattice Search: Hierarchical Heavy Hitters Revisited

Kenjiro Cho
IIJ Research Laboratory

IMC'17 November 2, 2017

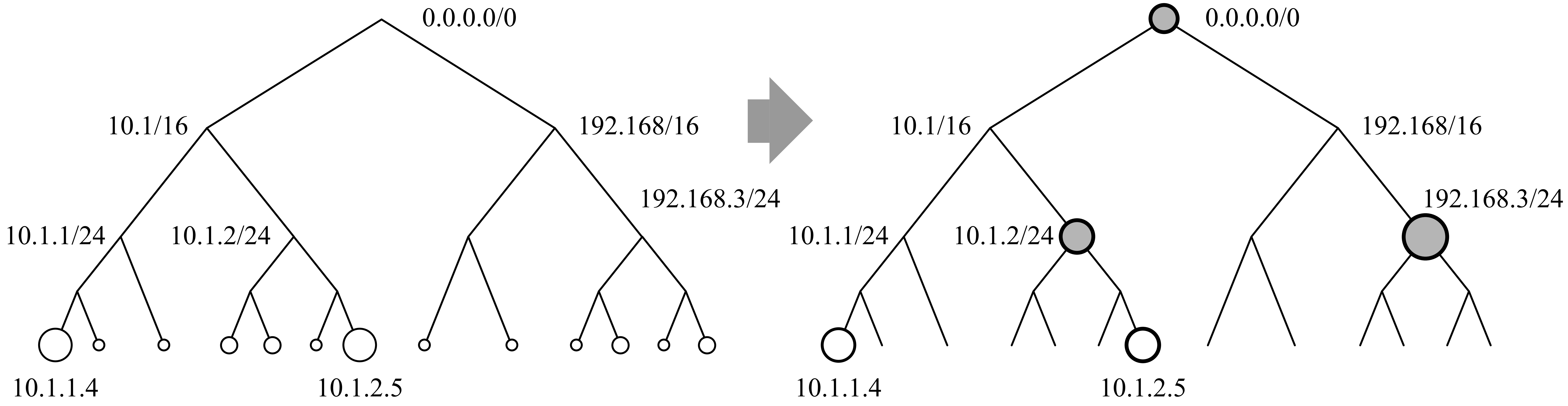
WIDE

Hierarchical Heavy Hitters (HHHs)

- identifying significant clusters across multiple planes
  - exploiting underlying hierarchical IP address structures
  - e.g., (src, dst) address pairs
    - *(1.2.3.4, *)* → one-to-many: e.g., scanning
    - *(*, 5.6.7.8)* → many-to-one: e.g., DDoS
    - *(1.2.3.0/24, 4.5.6.0/28)* → subnet-to-subnet
  - can be extended to higher dimensions (e.g., 5-tuple)
- powerful tool for traffic monitoring/anomaly detection

Unidimensional HHH

- an HHH: an aggregate with count $c \geq \varphi N$
  - φ: threshold N: total input (e.g., packets or bytes)
- HHHs can be uniquely identified by depth-first tree traversal
  - aggregating small nodes until it exceeds the threshold



Multi-dimensional HHH

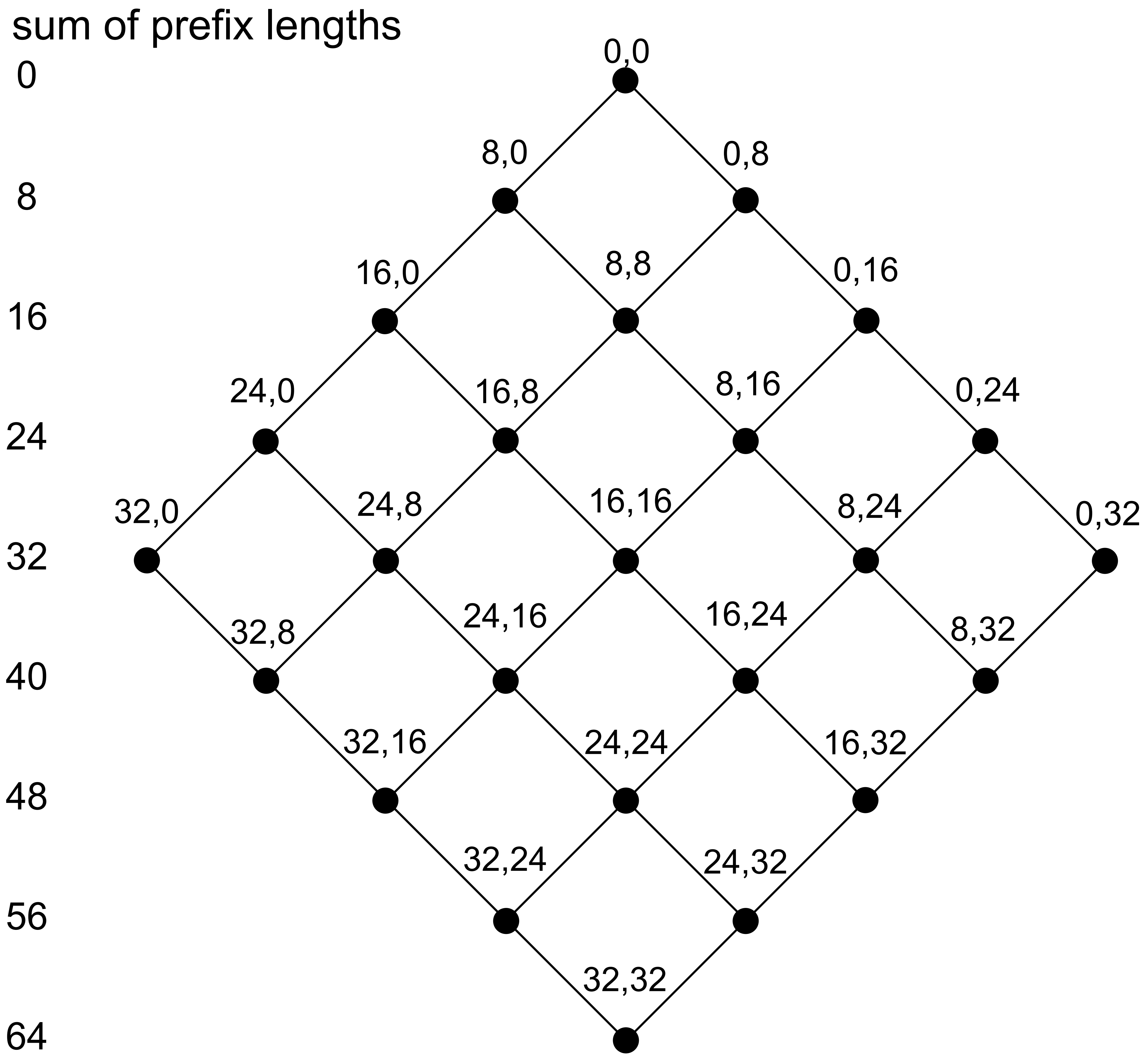
- each node has multiple parents
  - many combinations for aggregation
  - much harder than one-dimension
- search space for 2-dimensional IPv4 addrs
  - 5×5=25 for bytewise aggregation
  - 33×33=1089 for bitwise aggregation

src: 1.2.3.4 dst: 5.6.7.8

[1.2.3.4/32,5.6.0.0/16] [1.2.3.0/24,5.6.7.0/24] [1.2.0.0/16,1.2.3.4/32]

[1.2.3.4/32,5.6.7.0/24] [1.2.3.0/24,5.6.7.8/32]

[1.2.3.4/32,5.6.7.8/32]



Lattice for IPv4 prefix length pair
with 8-bit granularity

Challenges

- performance
  - bitwise aggregation is costly
- operational relevance
  - ordering: e.g., [32, *] and [16, 16]
  - broad and redundant aggregates: (e.g., 128/4 and 128/2)
- re-aggregation
  - useful for interactive analysis (for zoom-in/out)

Contributions

- new efficient HHH algorithm for bitwise aggregation
  - matches operational needs, supports re-aggregation
- open-source tool and open datasets
- more broadly, transforming the existing hard problem into a tractable one, by revisiting the commonly accepted definition

Various HHH definitions

- discounted HHH ⬅ we also employ this
  - exclude descendant HHHs' counts for concise outputs

$$c_i' = \sum_j c_j' \ \ where \ \{ j \in child(i) \mid c_j' < \varphi N \}$$

- rollup rules: how to aggregate counts to parent
  - overlap rule: allows double-counting to detect all possible HHHs
  - split rule: preserves counts ⬅ we use a simple first-found split rule
- aggregation ordering
  - sum of prefix lengths ⬅ we'll revisit this ordering

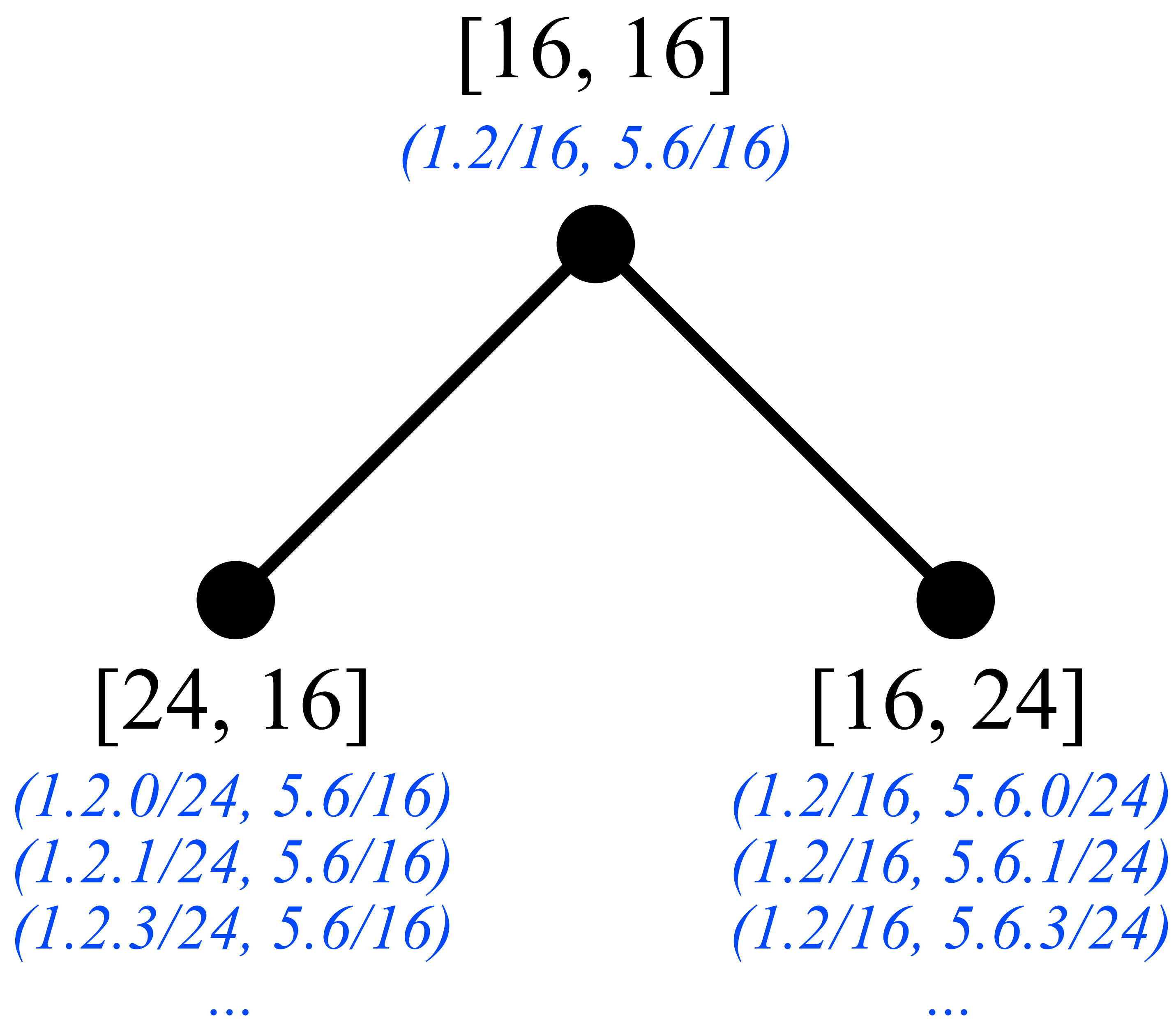
Previous algorithms

- elaborate structures
  - cross-producting, grid-of-trie, rectangle-search
- theoretical analyses
  - streaming approximation algorithms w/ error bounds
- all the existing methods are bottom-up

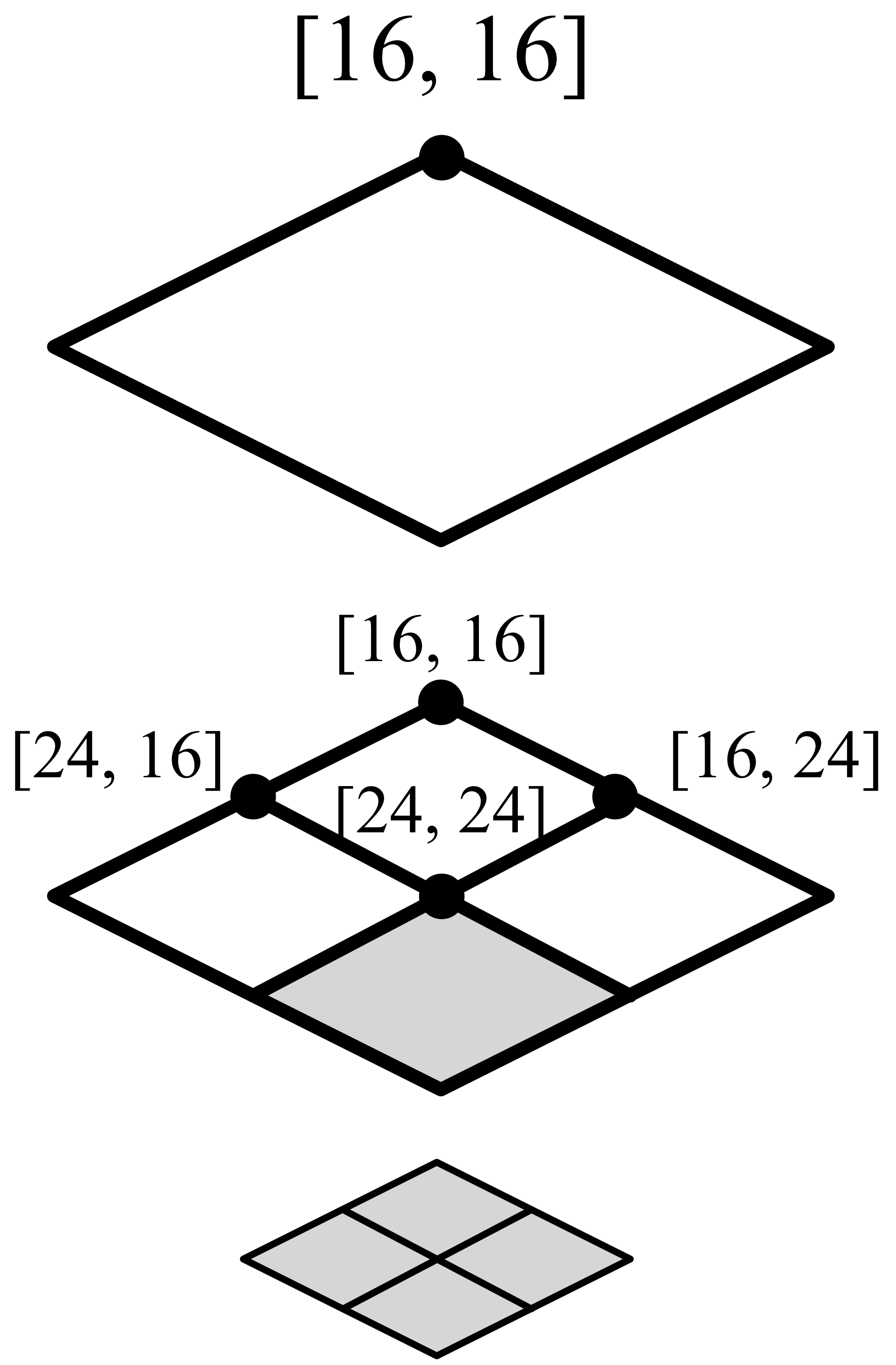
- our algorithm: top-down, deterministic
  - no elaborate structure, no approximation, no parameter

HHH revisited

- key idea: redefine *child(i)* to allow space partitioning
  - *child(i)*: from bin-tree to quadtree



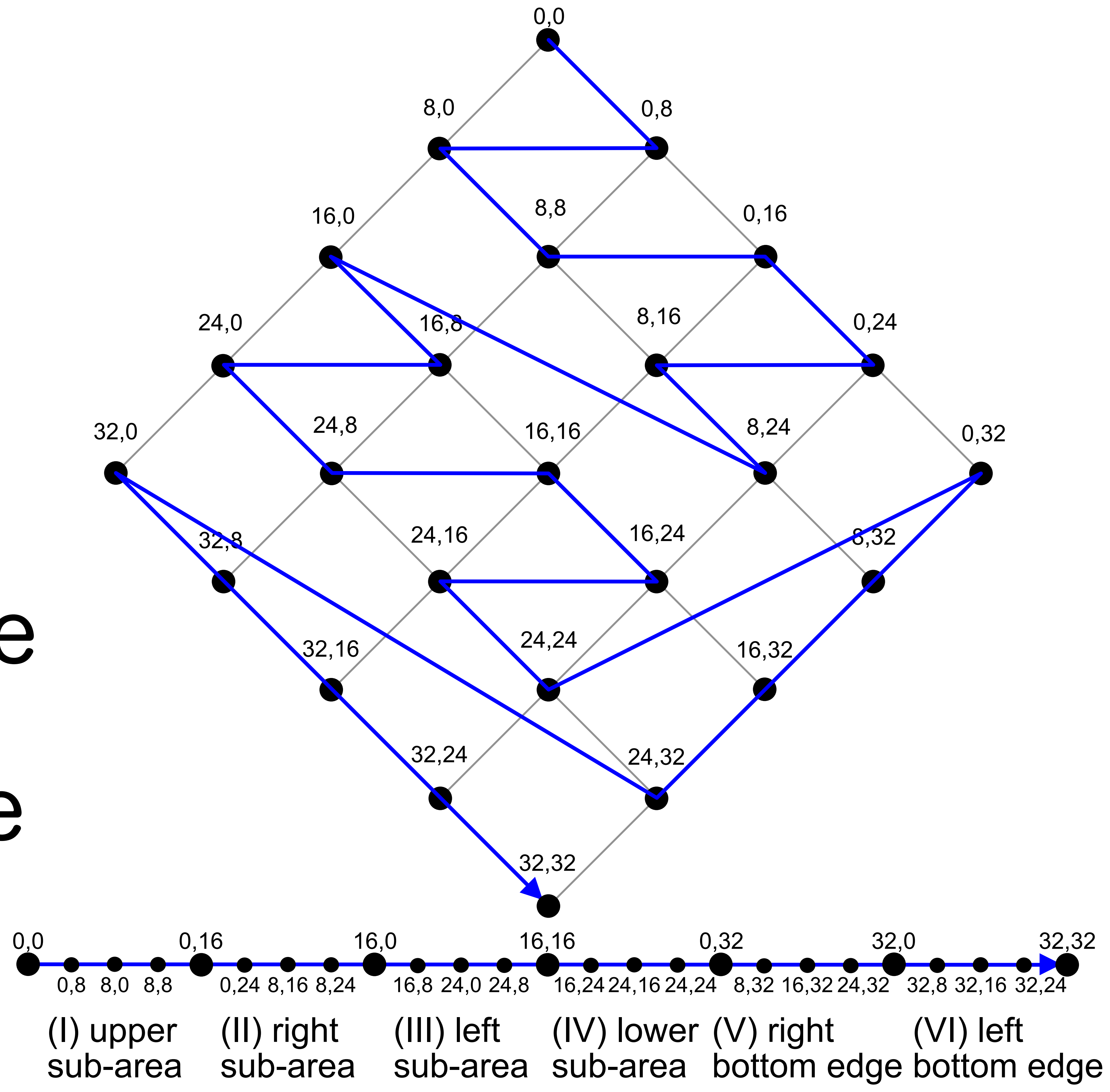
bottom-up aggregation



top-down space partitioning

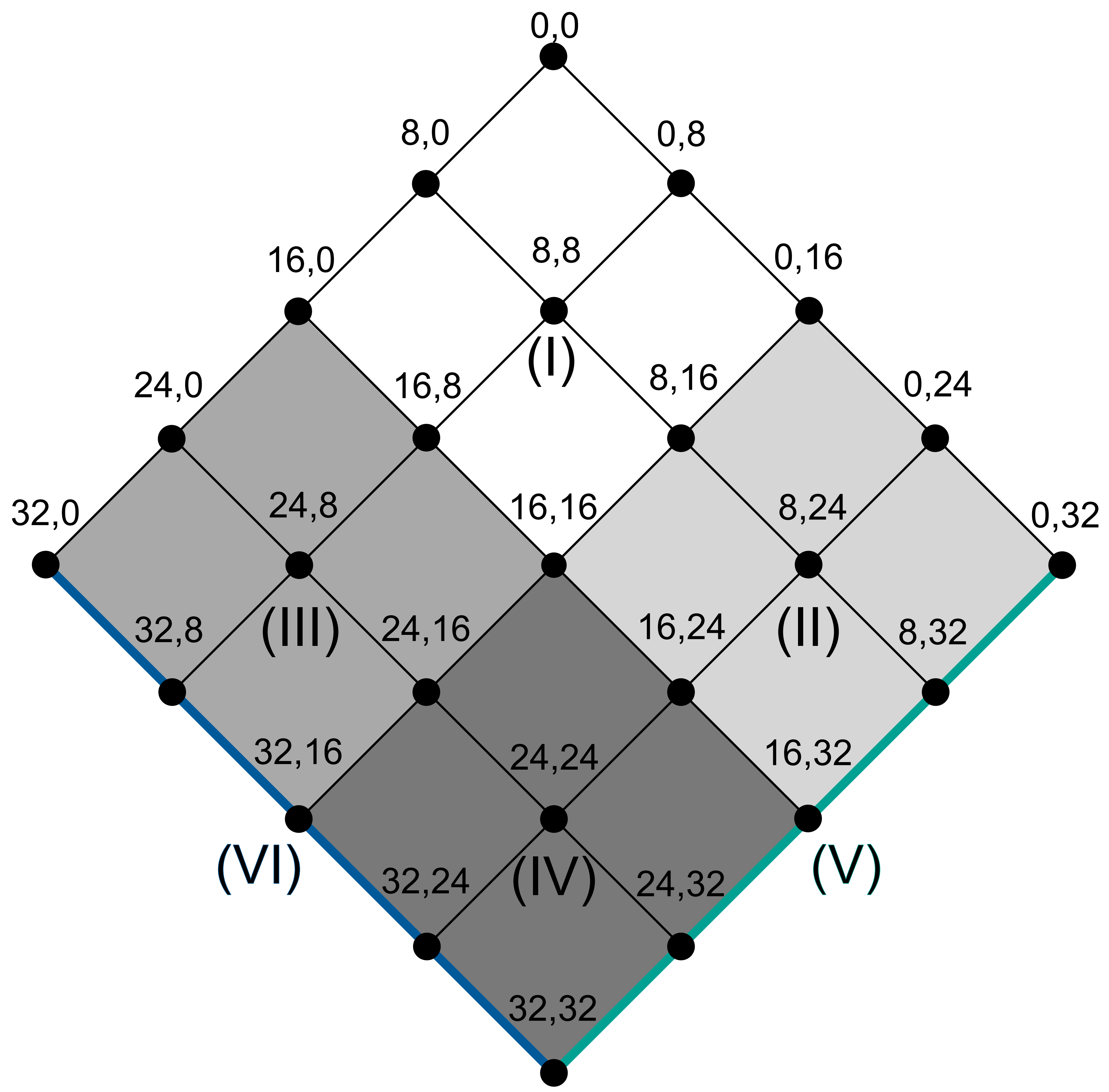
Z-order [Morton1966]

- a space filling curve
  - by bit-interleaving *(l_0, l_1)*
- prefers the largest value across dimensions
- looks different from standard Z-curve
  - [0..32] doesn't have full 5-bit space
  - makes /32 higher in the order



Recursive spatial partitioning

- visit regions from (VI) to (I) recursively
  - 2 bottom edges
    - (VI) left-bottom edge
    - (V) right-bottom edge
  - 4 quadrants
    - (IV) lower quadrant
    - (III) left quadrant
    - (II) right quadrant
    - (I) upper quadrant

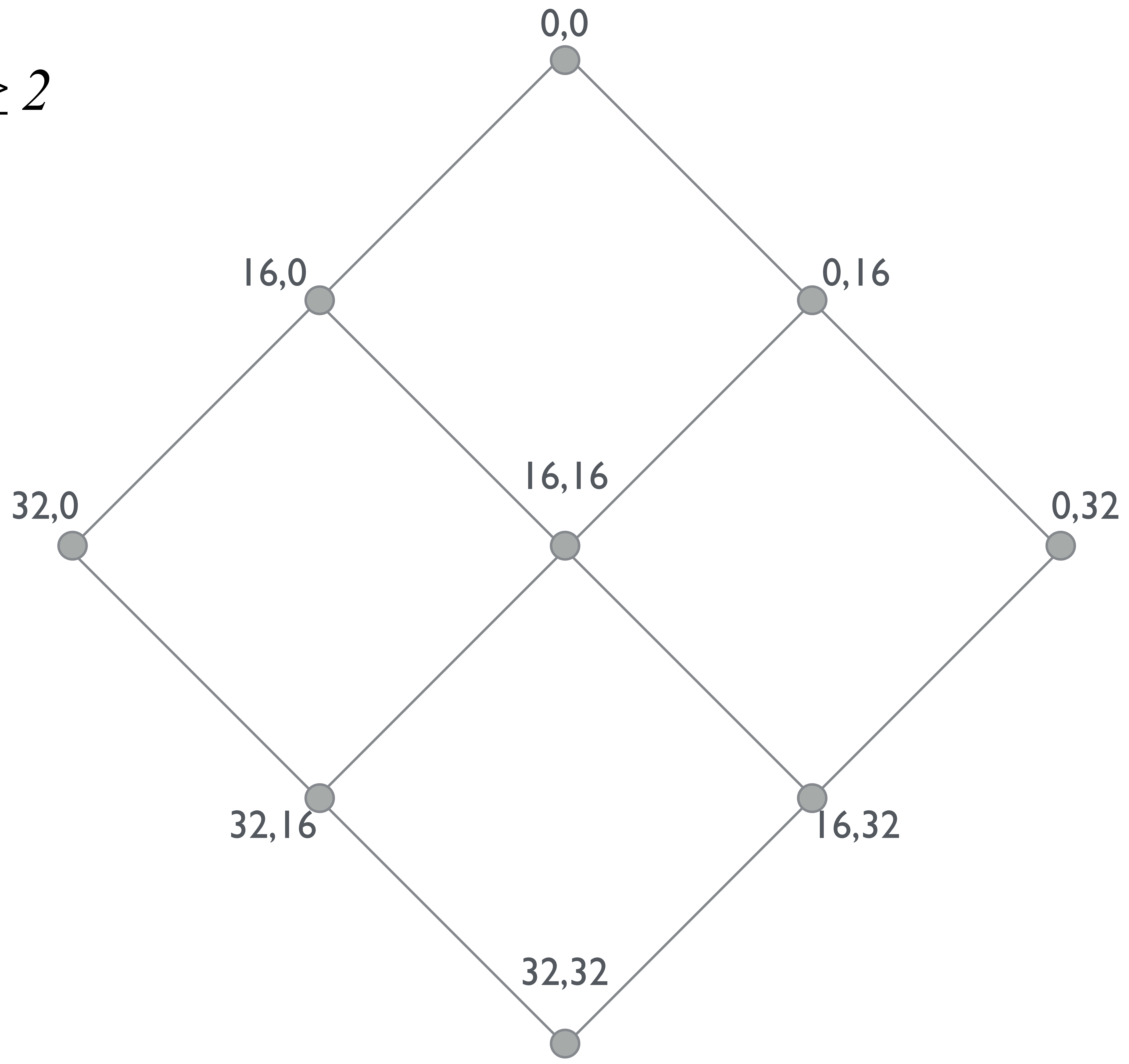


Recursive Lattice Search (RLS)

- idea: recursively subdivide aggregates by Z-order
- pros
  - recurse only for flows ≥ thresh
  - sub-division needs only parent's sub-flows
  - /32 becomes higher in the order
- cons
  - bias for the first dimension

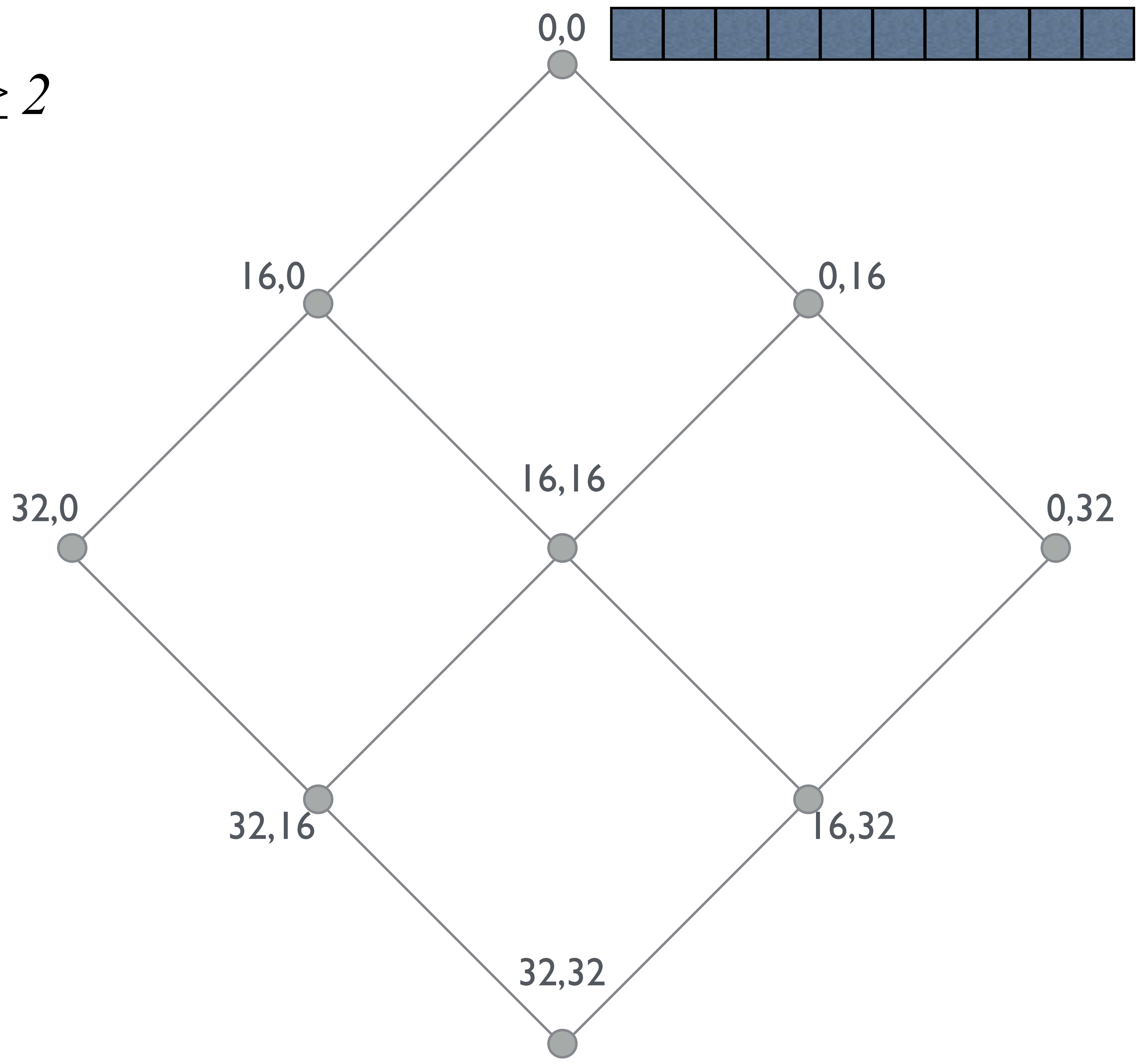
RLS Illustrated

10 inputs, HHH ≥ 2



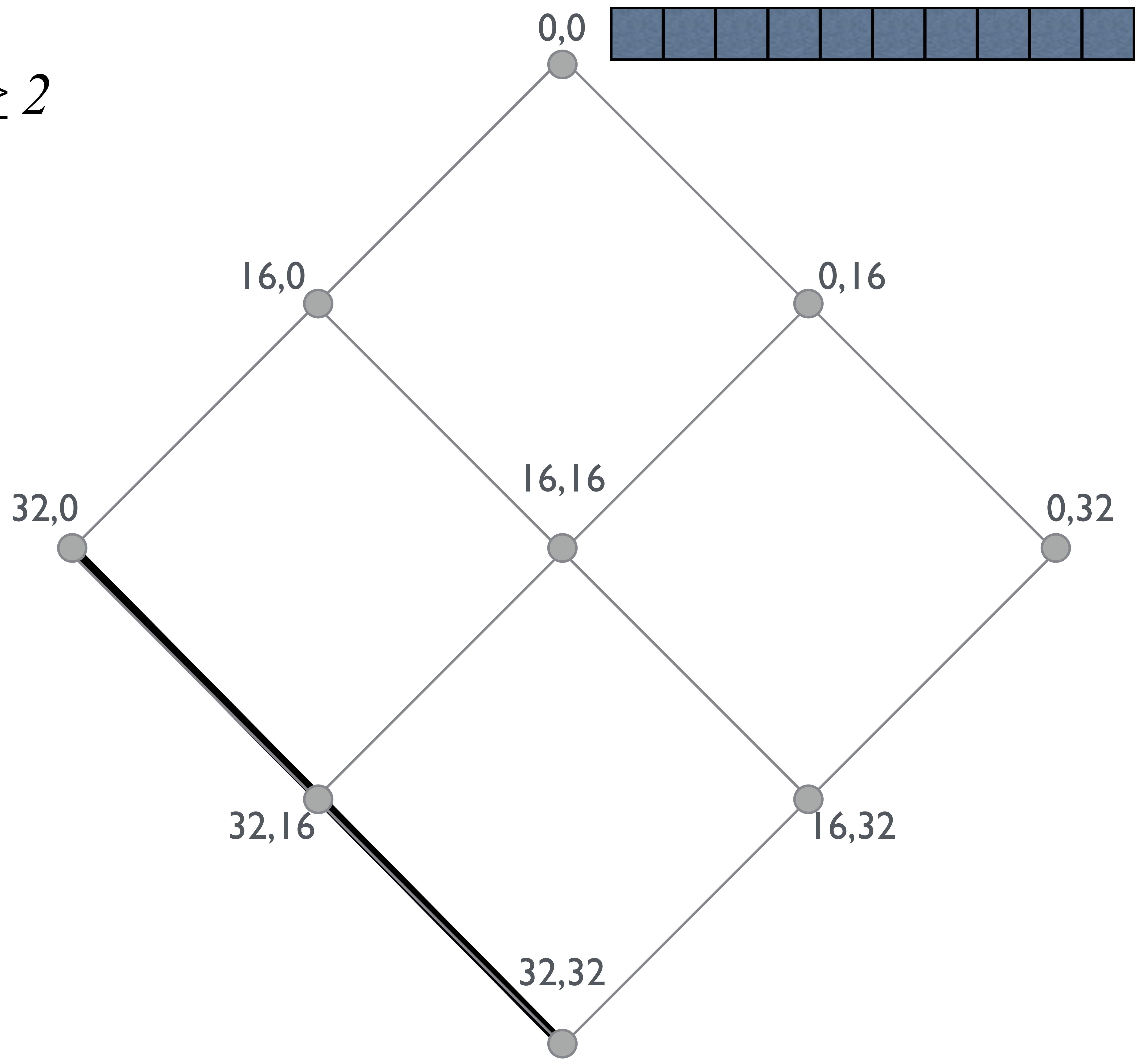
RLS Illustrated

10 inputs, HHH ≥ 2



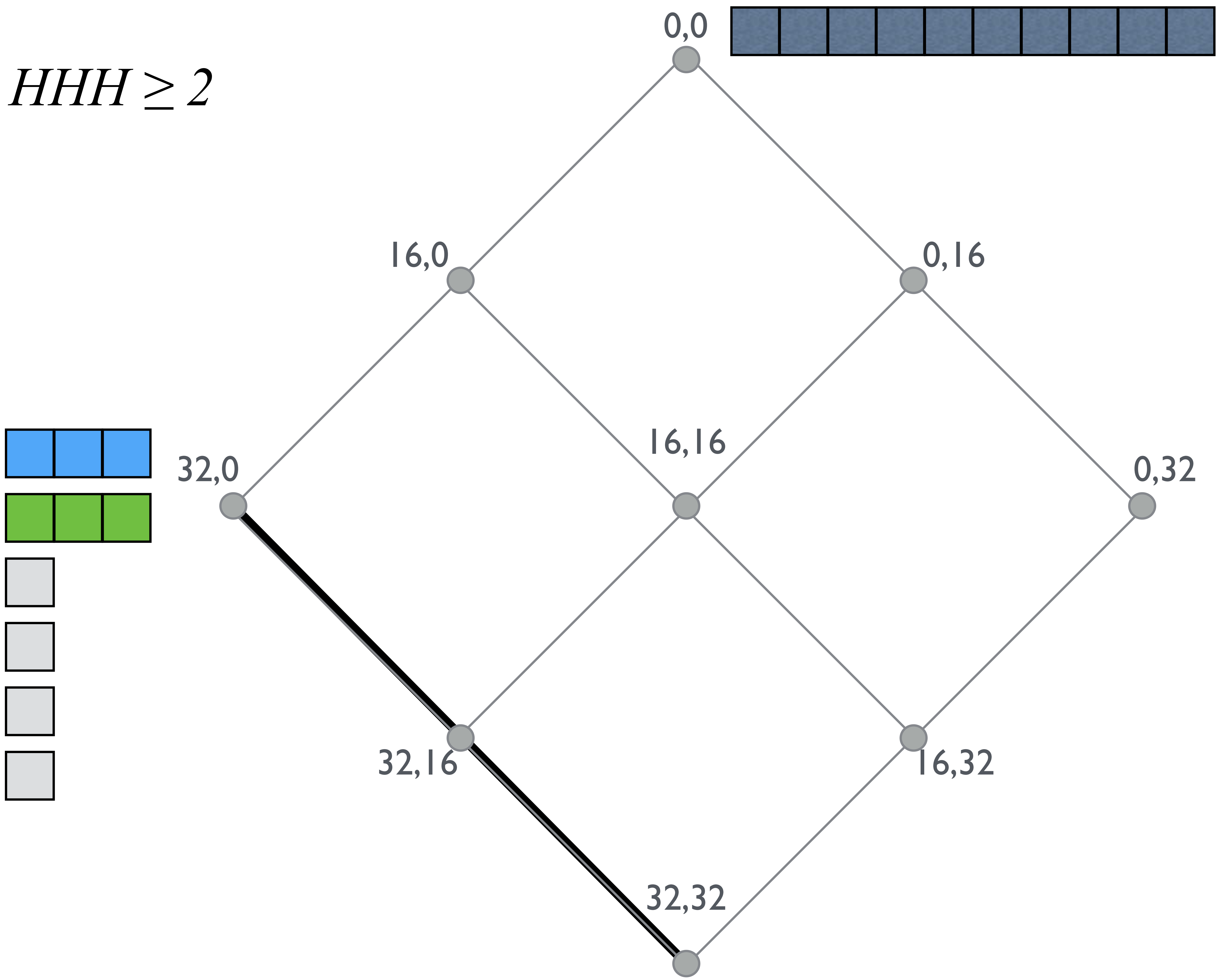
RLS Illustrated

10 inputs, HHH ≥ 2



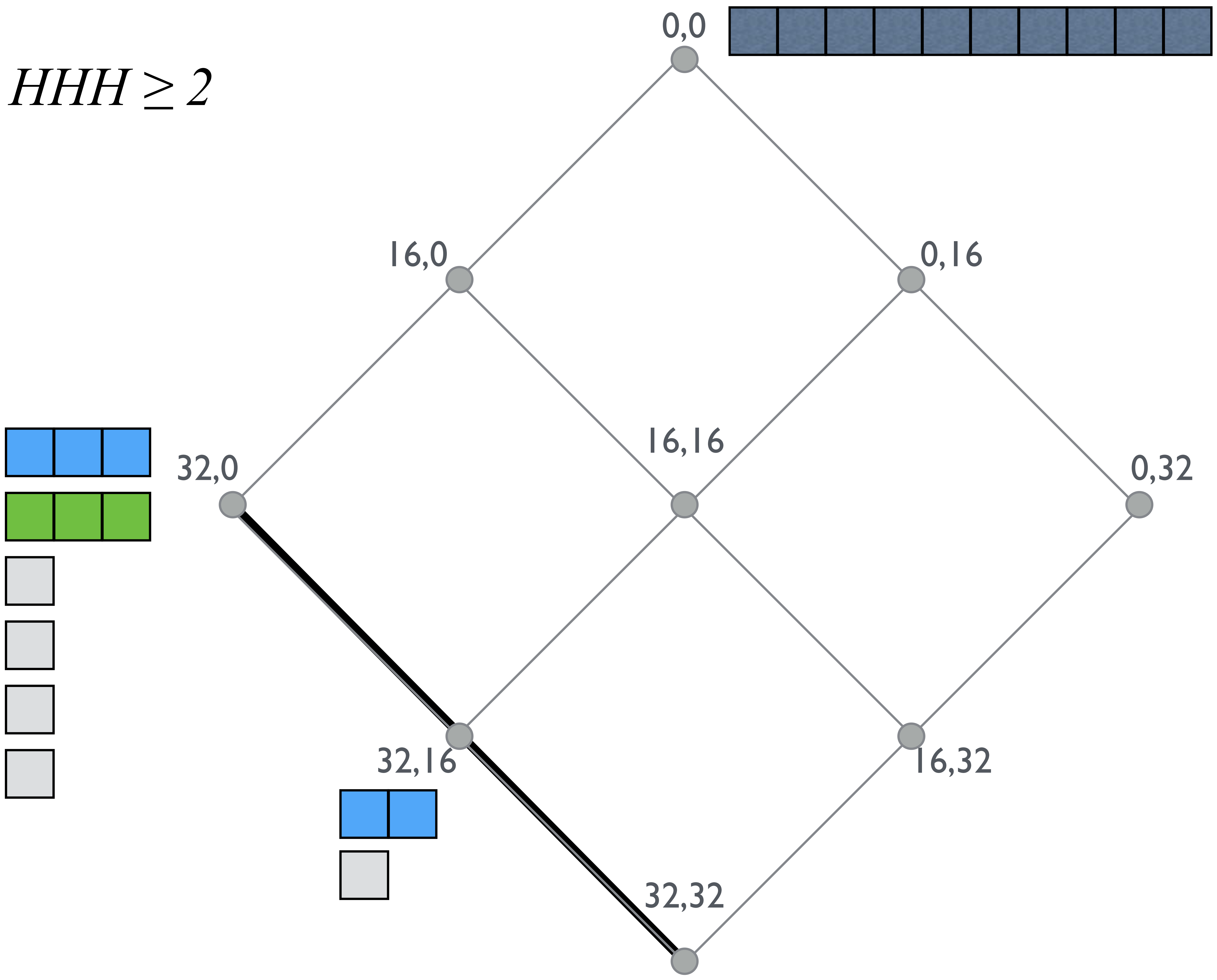
RLS Illustrated

10 inputs, HHH ≥ 2



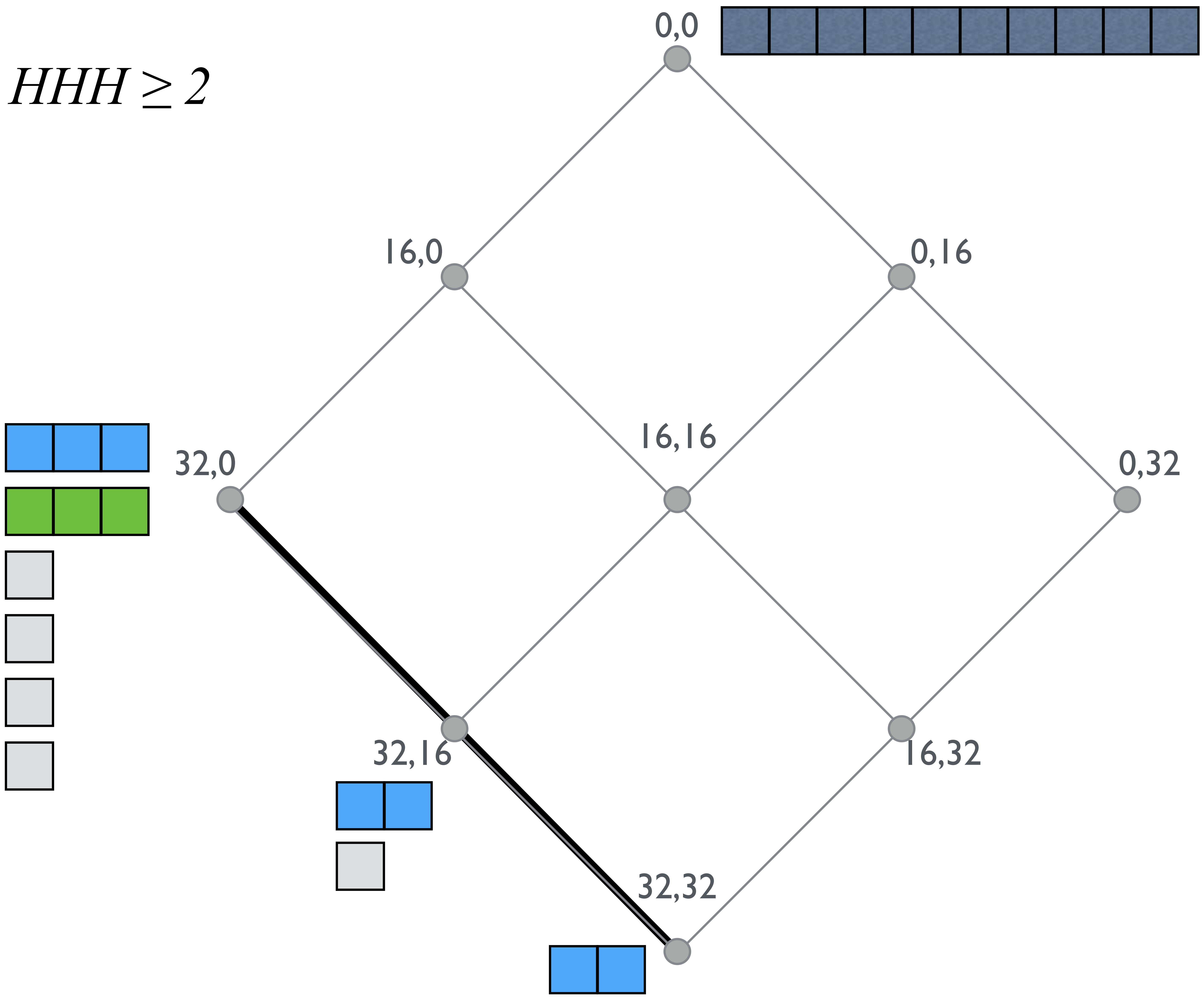
RLS Illustrated

10 inputs, HHH ≥ 2



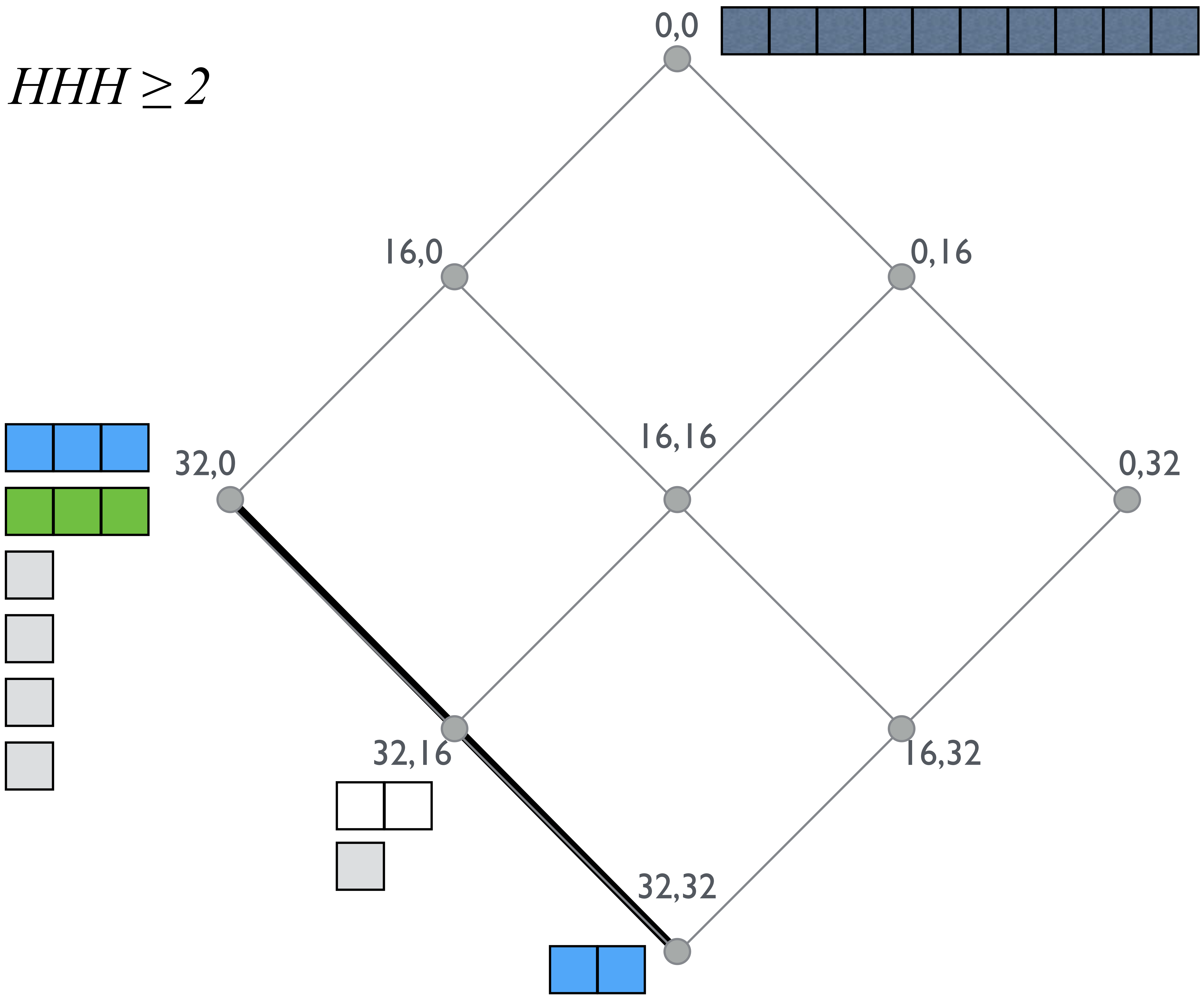
RLS Illustrated

10 inputs, HHH ≥ 2



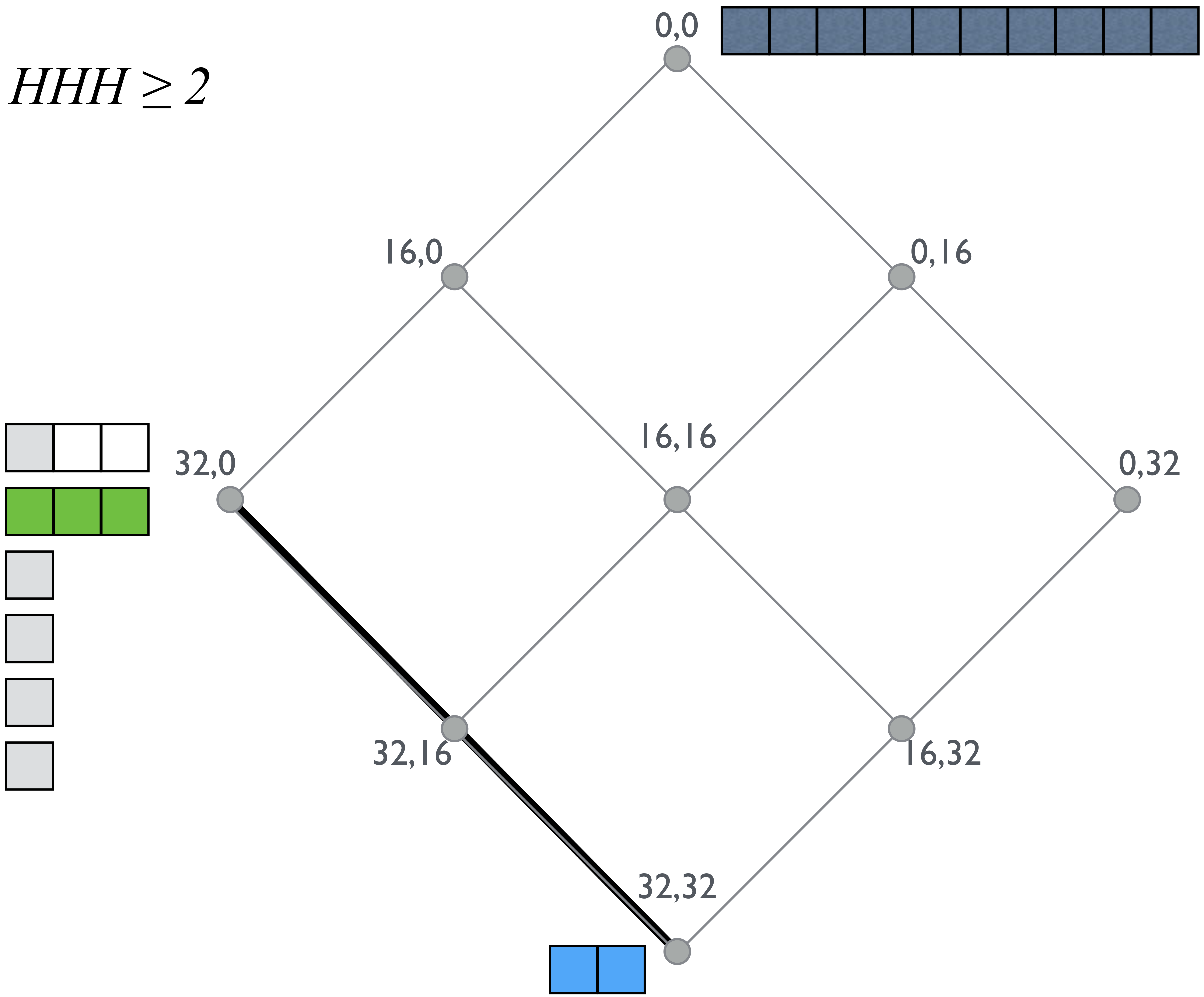
RLS Illustrated

10 inputs, HHH ≥ 2



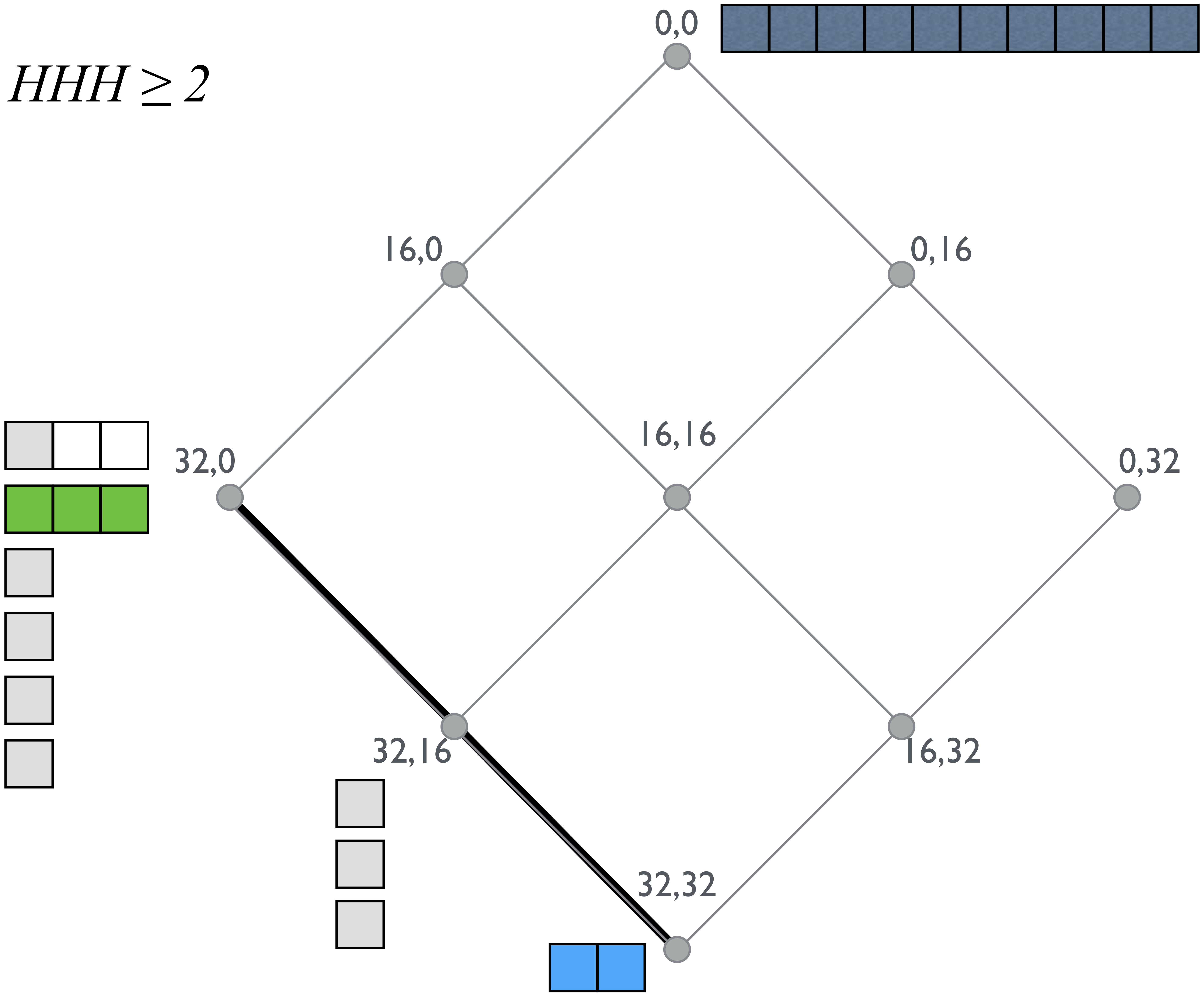
RLS Illustrated

10 inputs, HHH ≥ 2



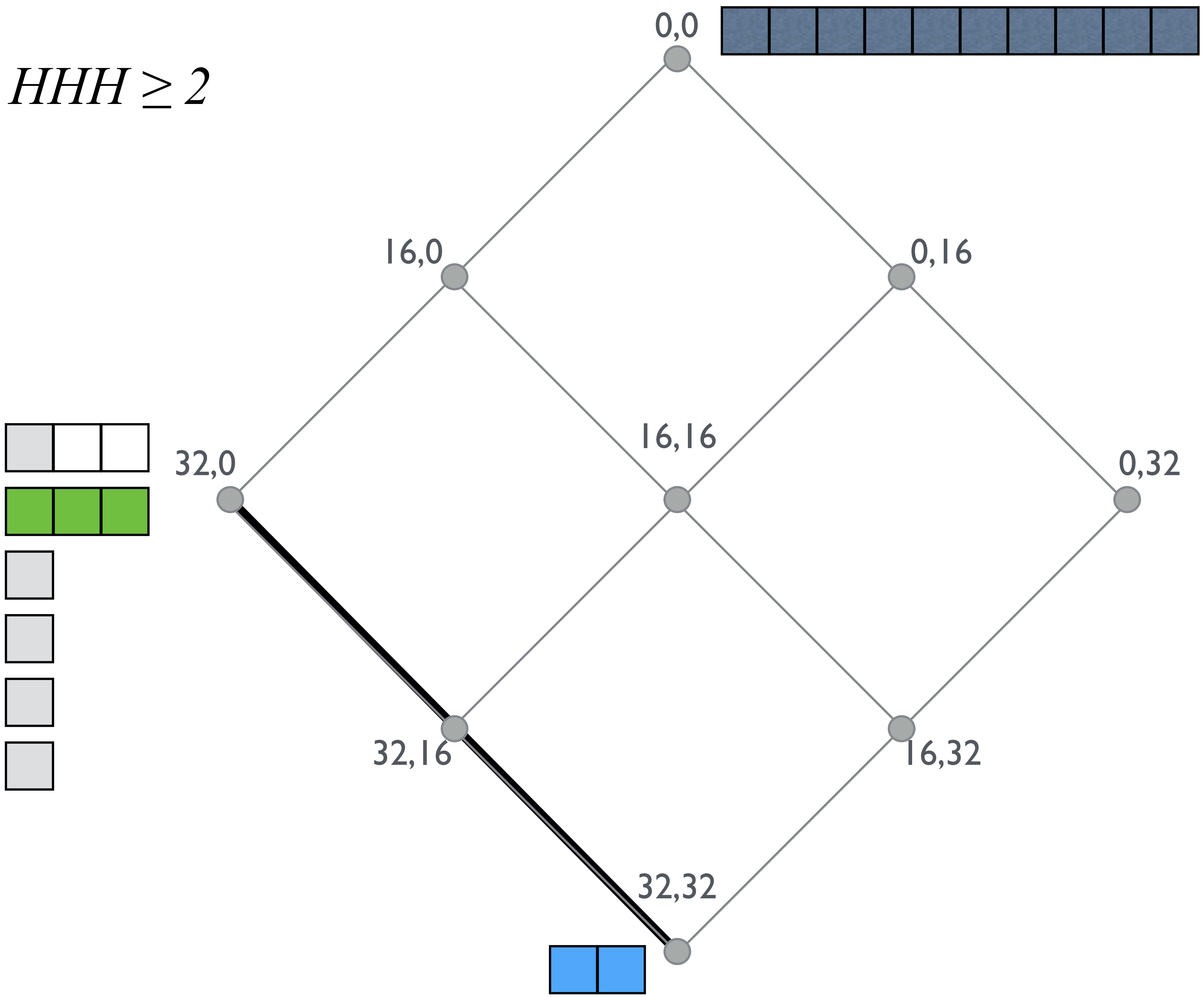
RLS Illustrated

10 inputs, HHH ≥ 2



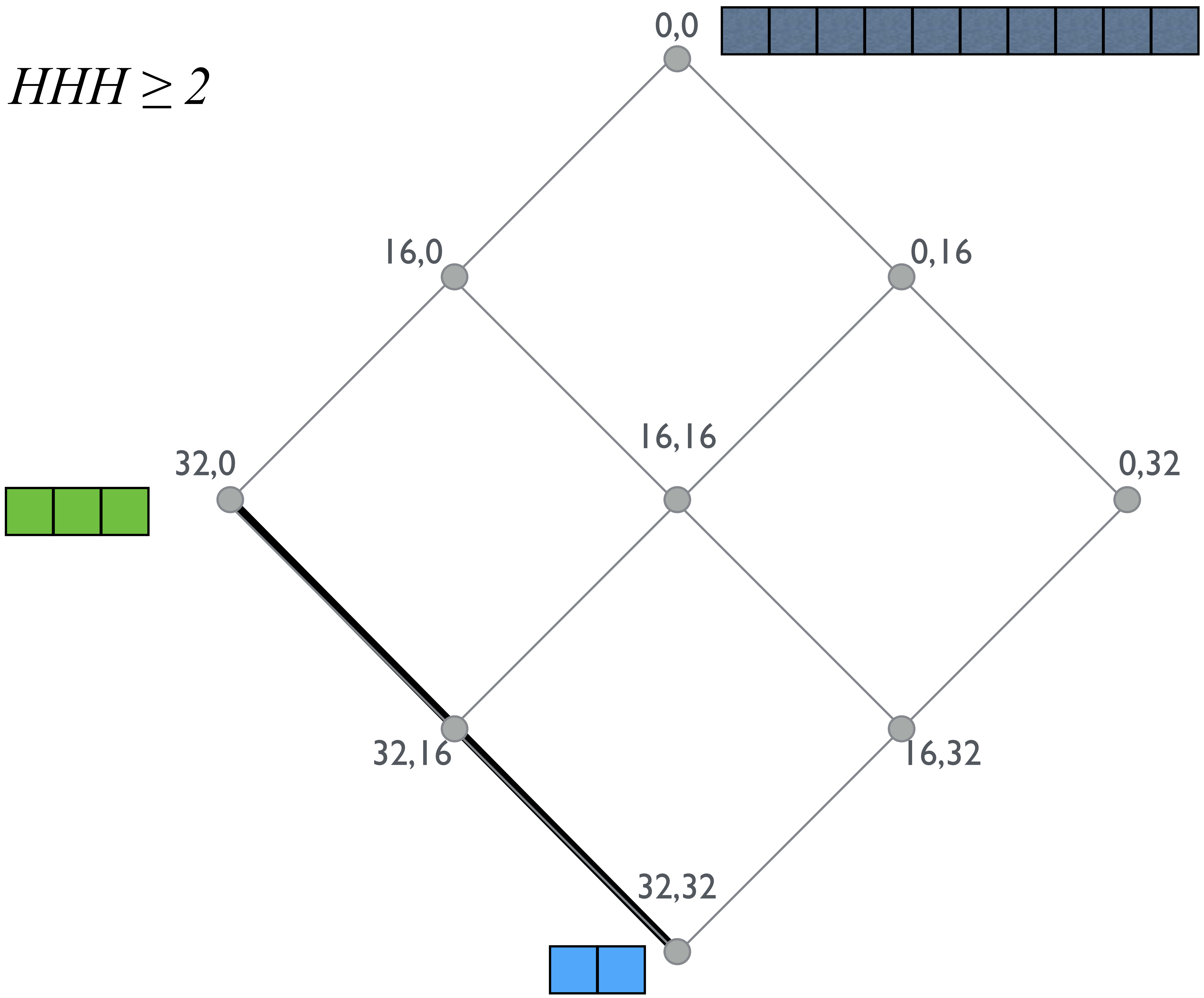
RLS Illustrated

10 inputs, HHH ≥ 2



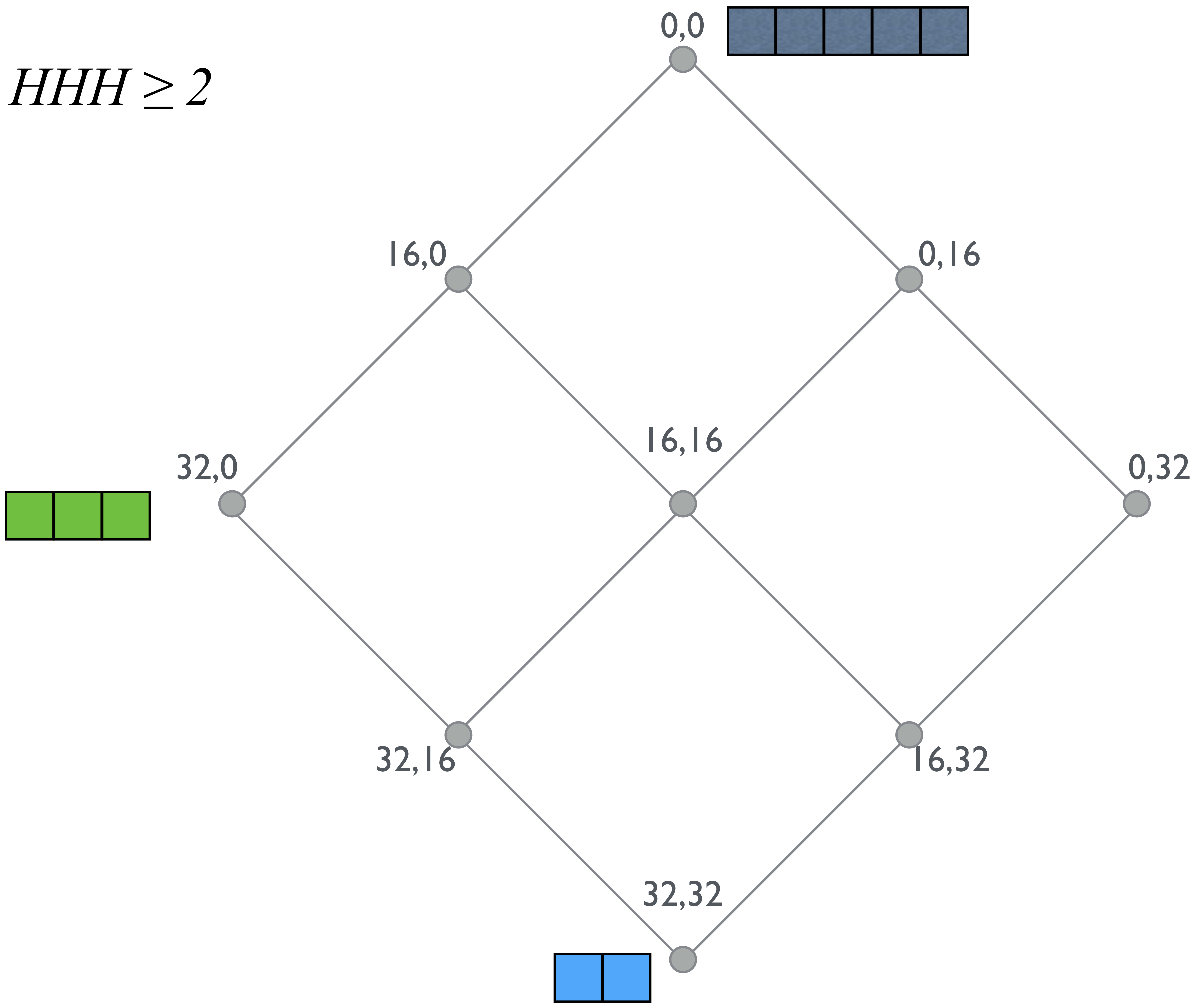
RLS Illustrated

10 inputs, HHH ≥ 2



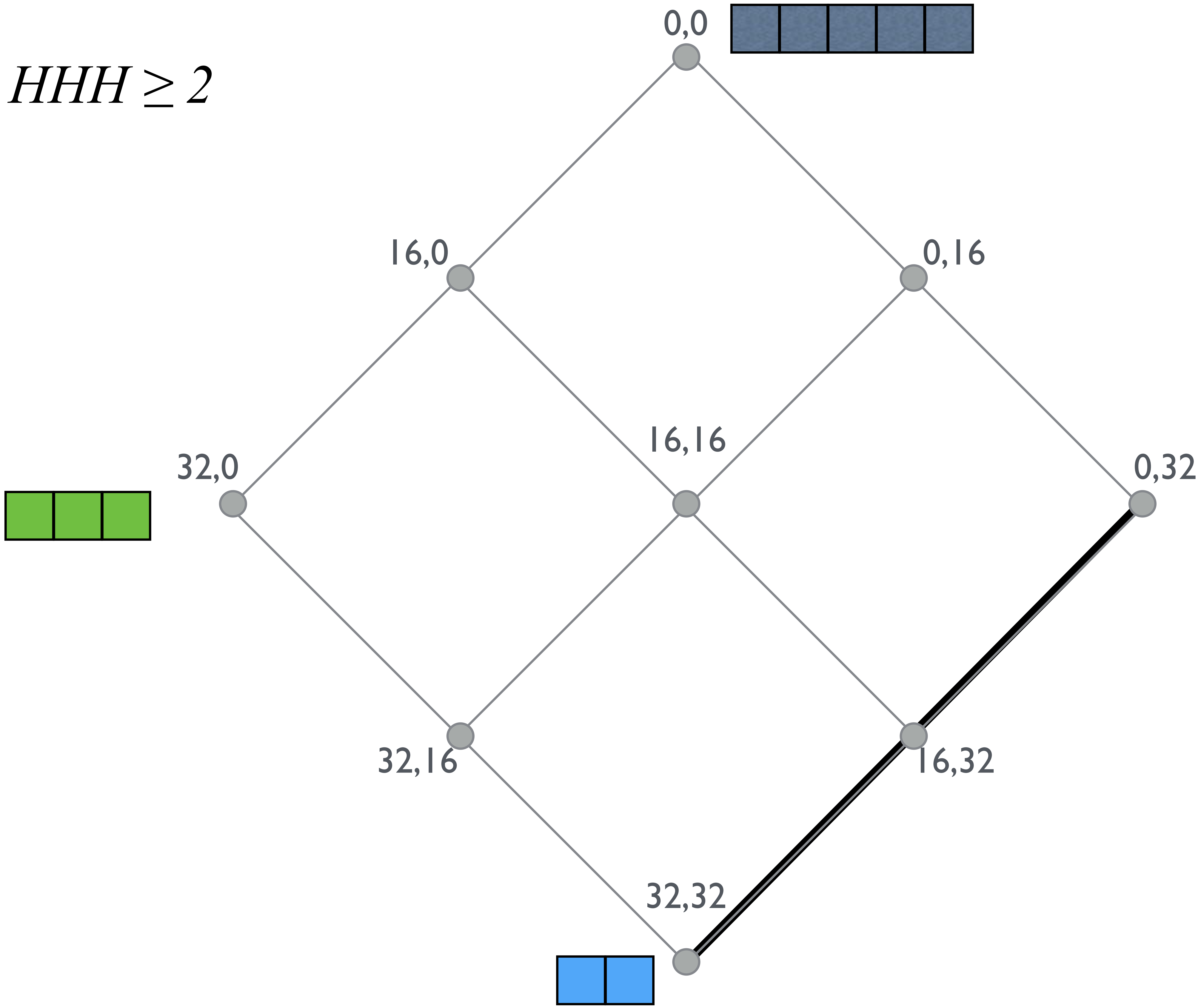
RLS Illustrated

10 inputs, HHH ≥ 2



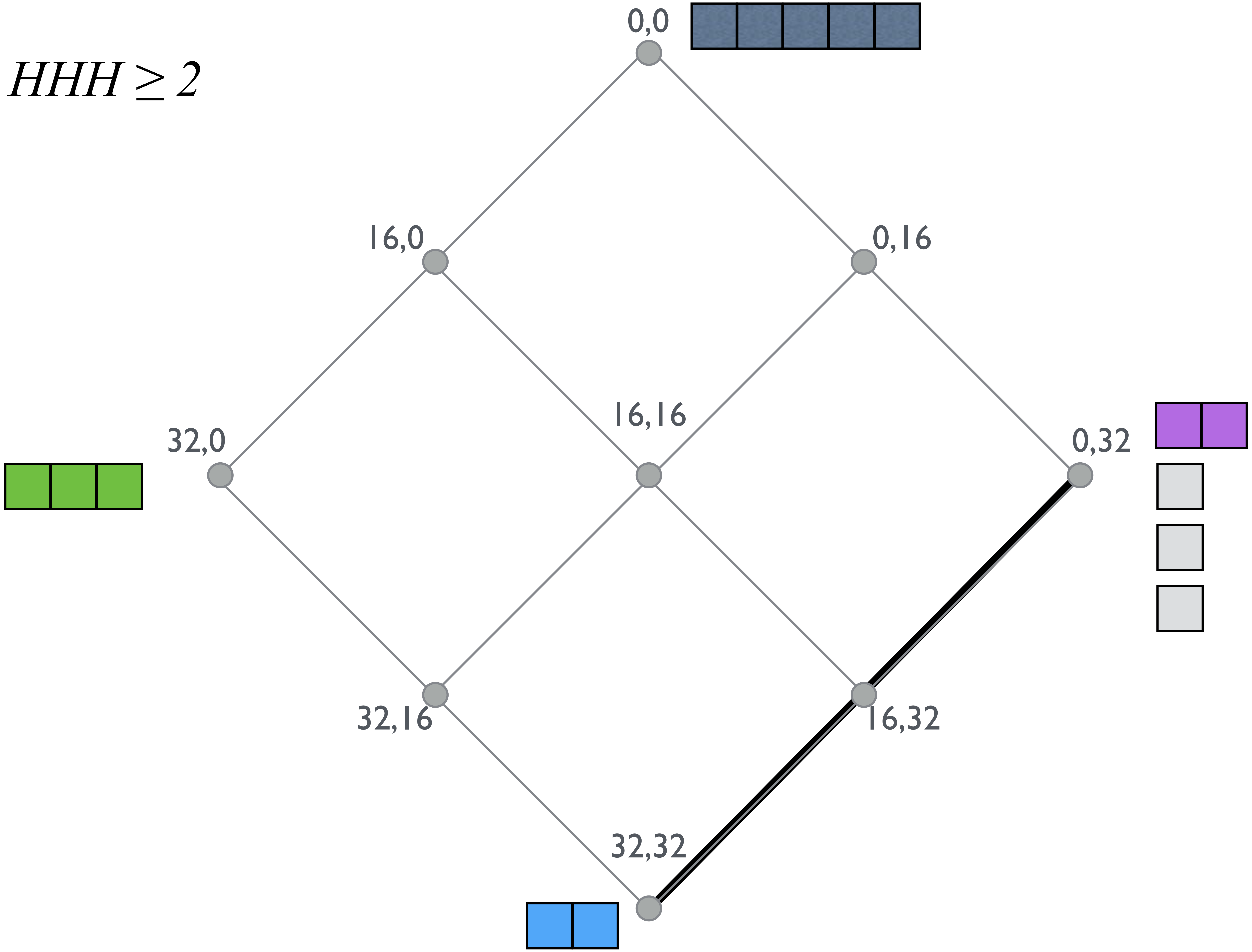
RLS Illustrated

10 inputs, HHH ≥ 2



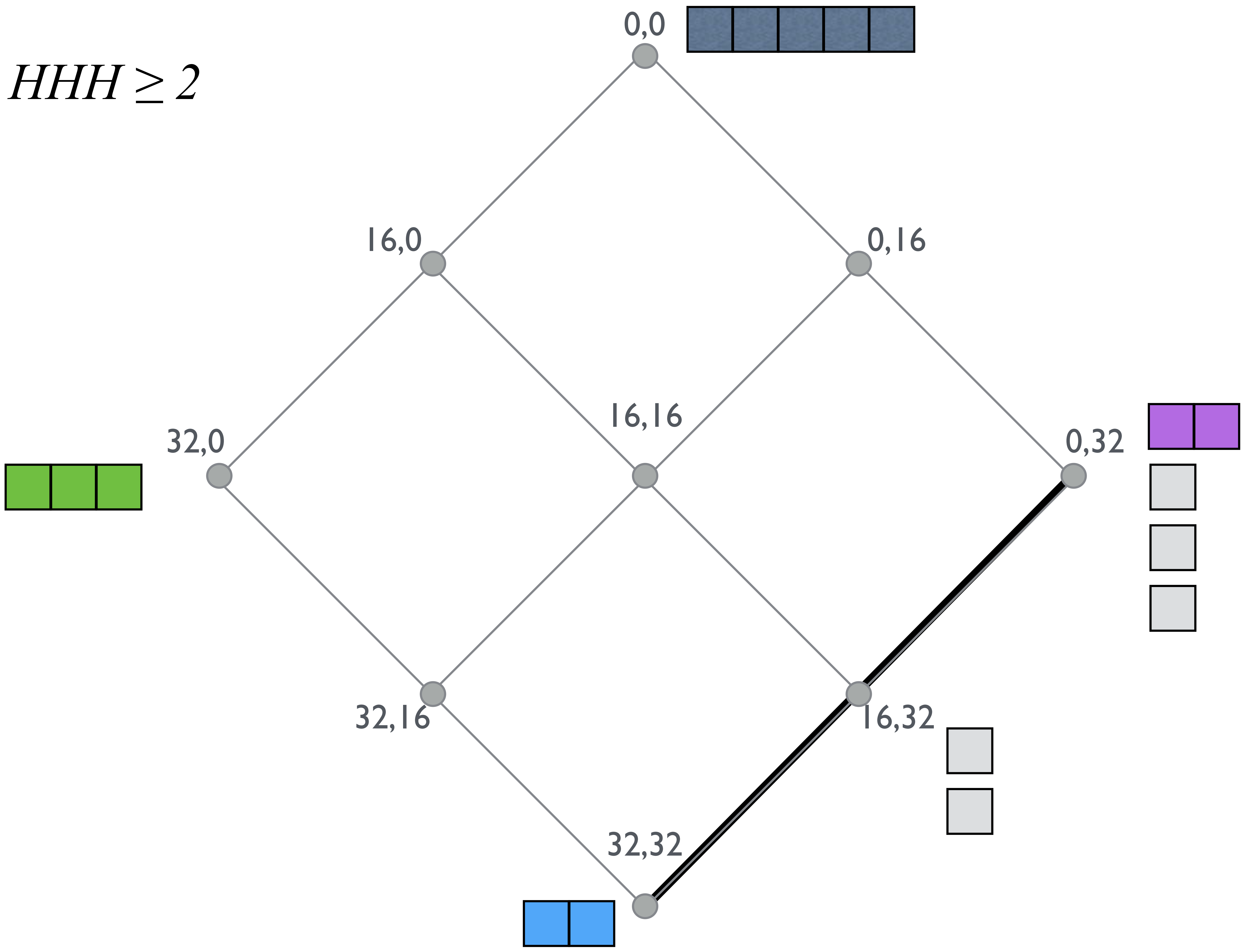
RLS Illustrated

10 inputs, HHH ≥ 2



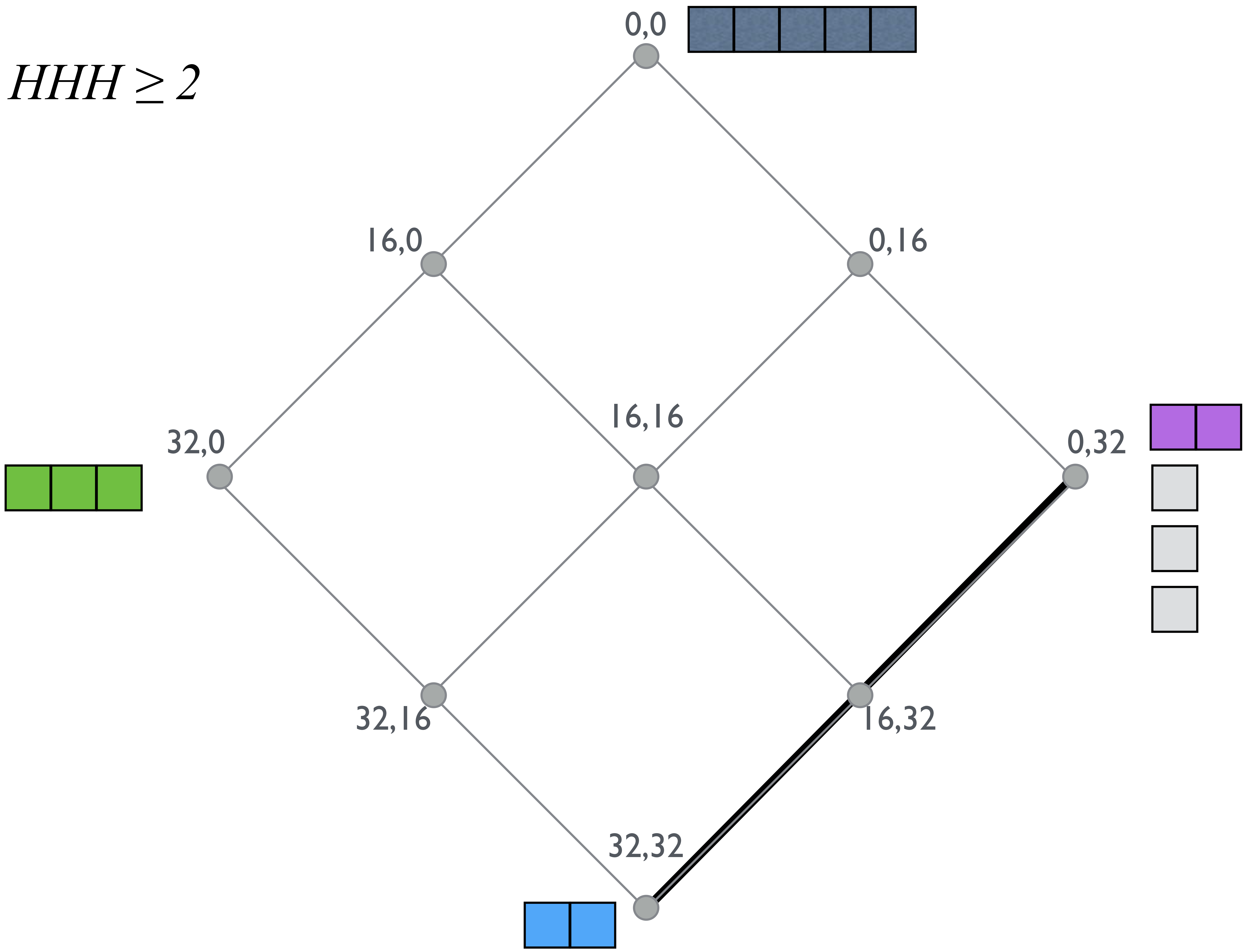
RLS Illustrated

10 inputs, HHH ≥ 2



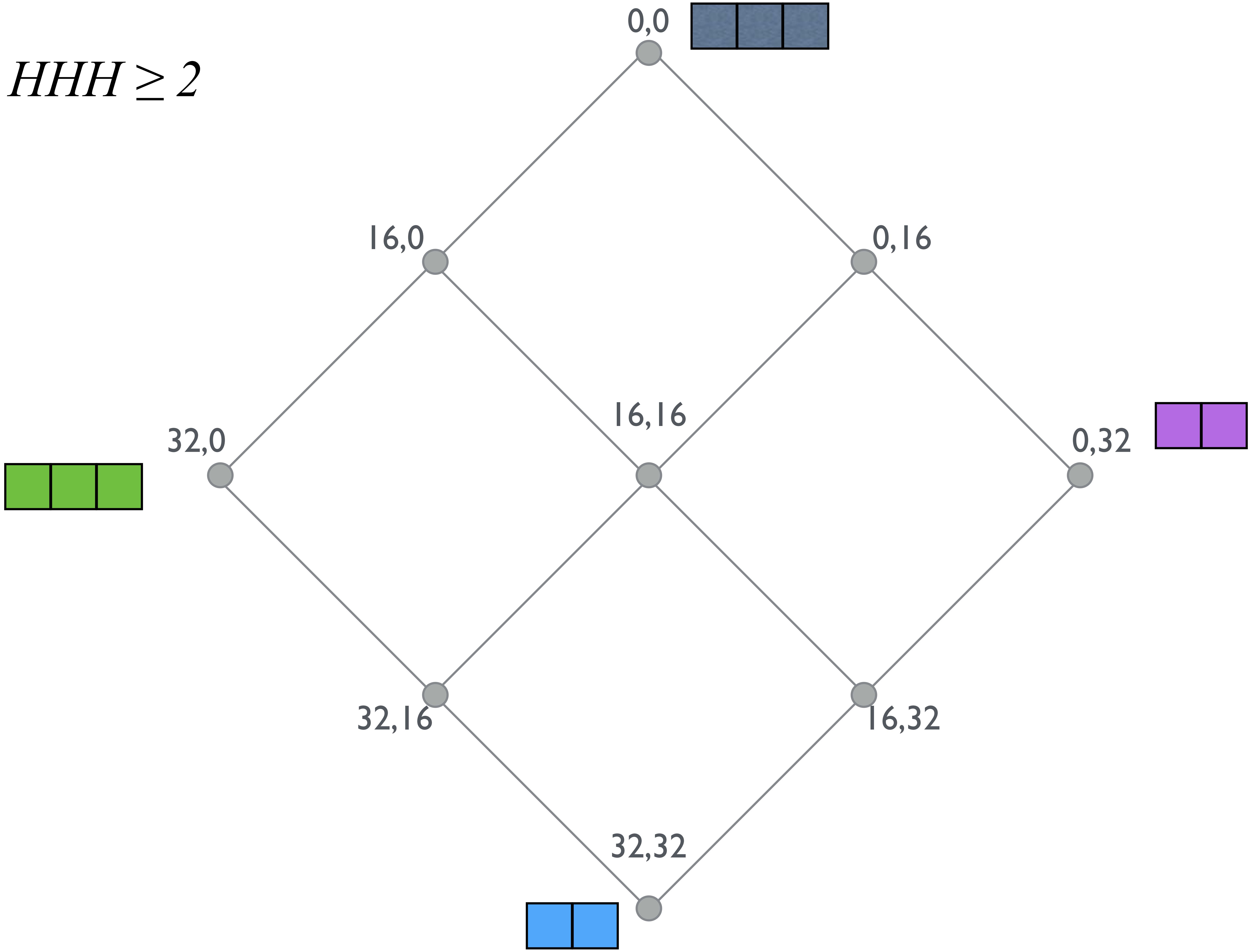
RLS Illustrated

10 inputs, HHH ≥ 2



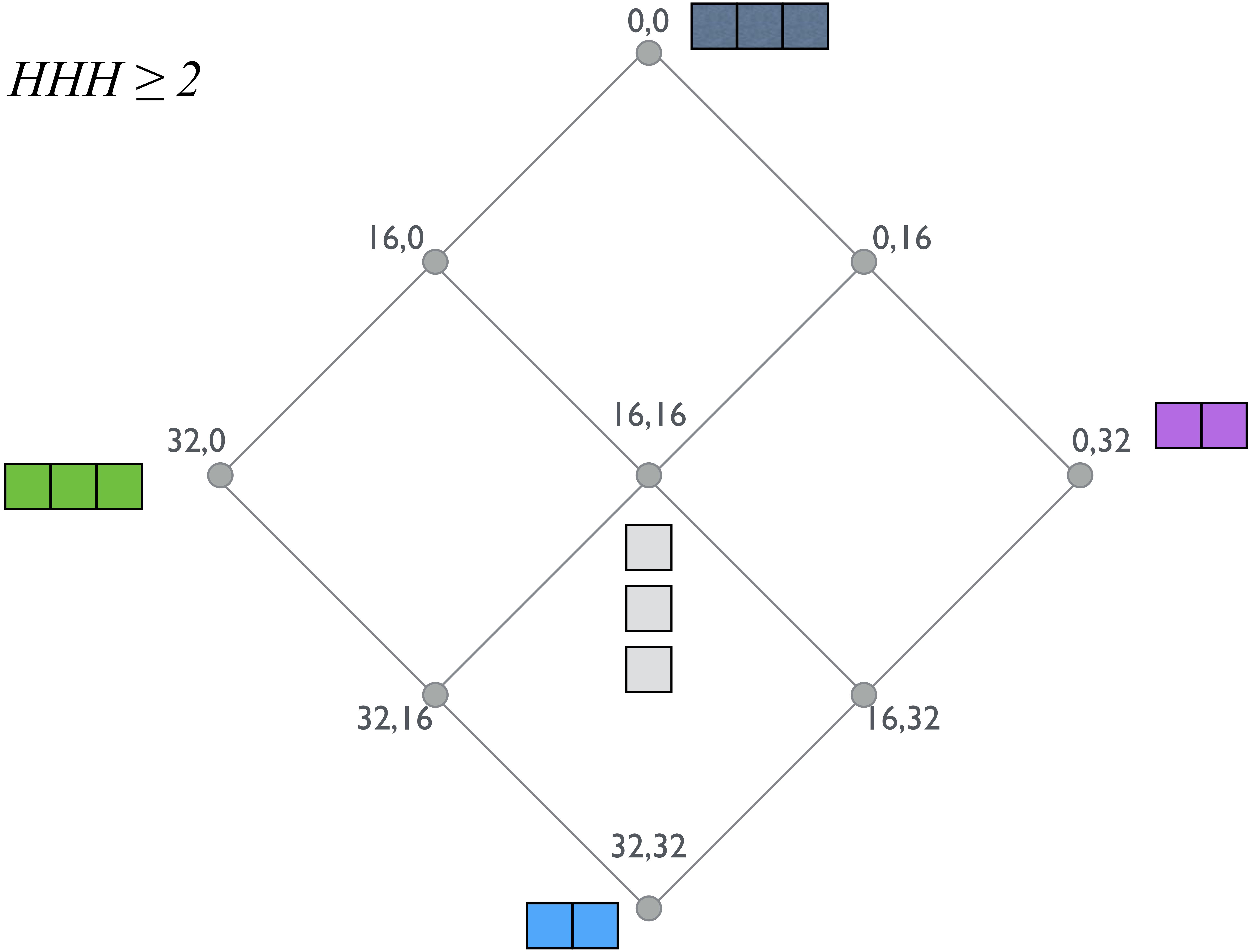
RLS Illustrated

10 inputs, HHH ≥ 2



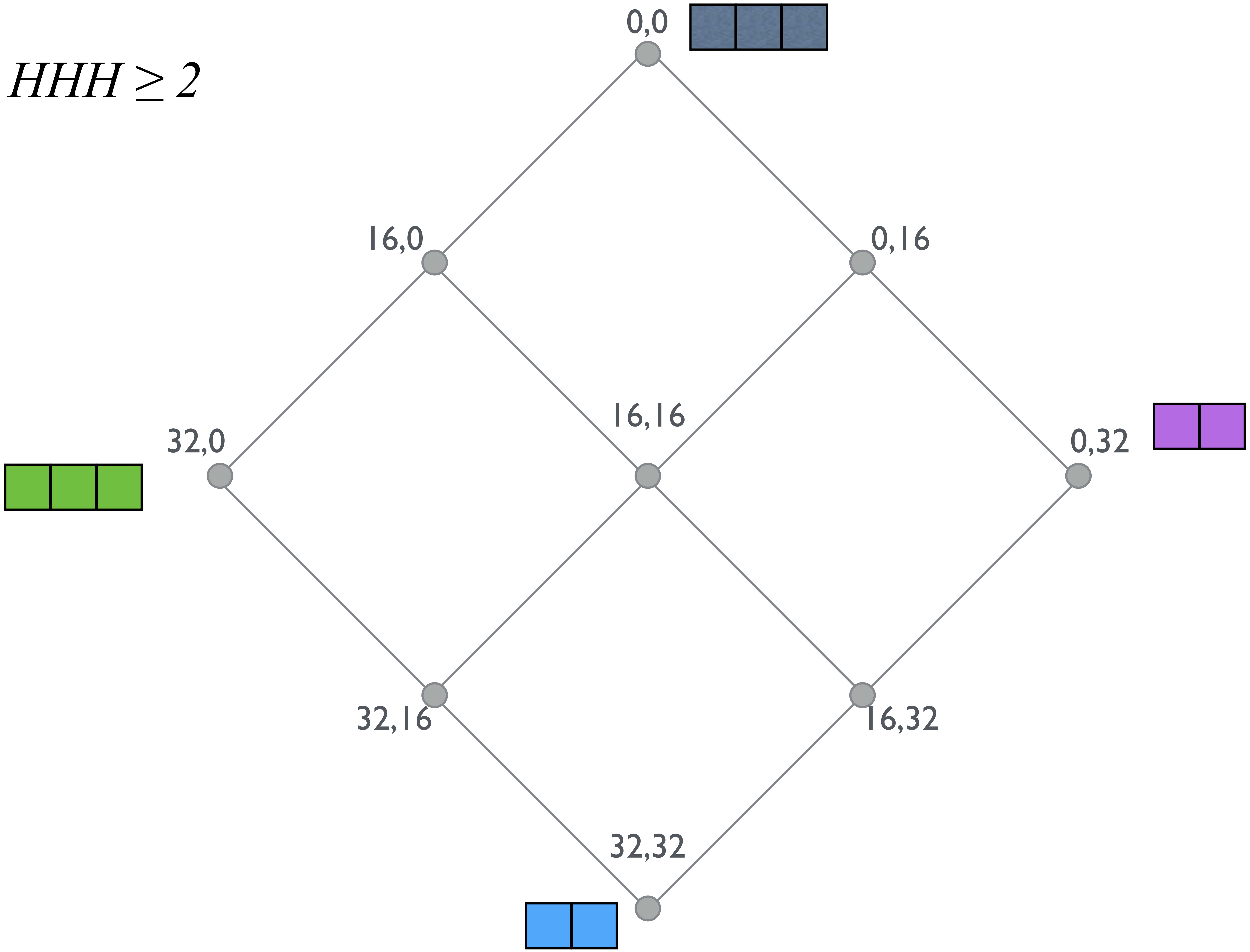
RLS Illustrated

10 inputs, HHH ≥ 2



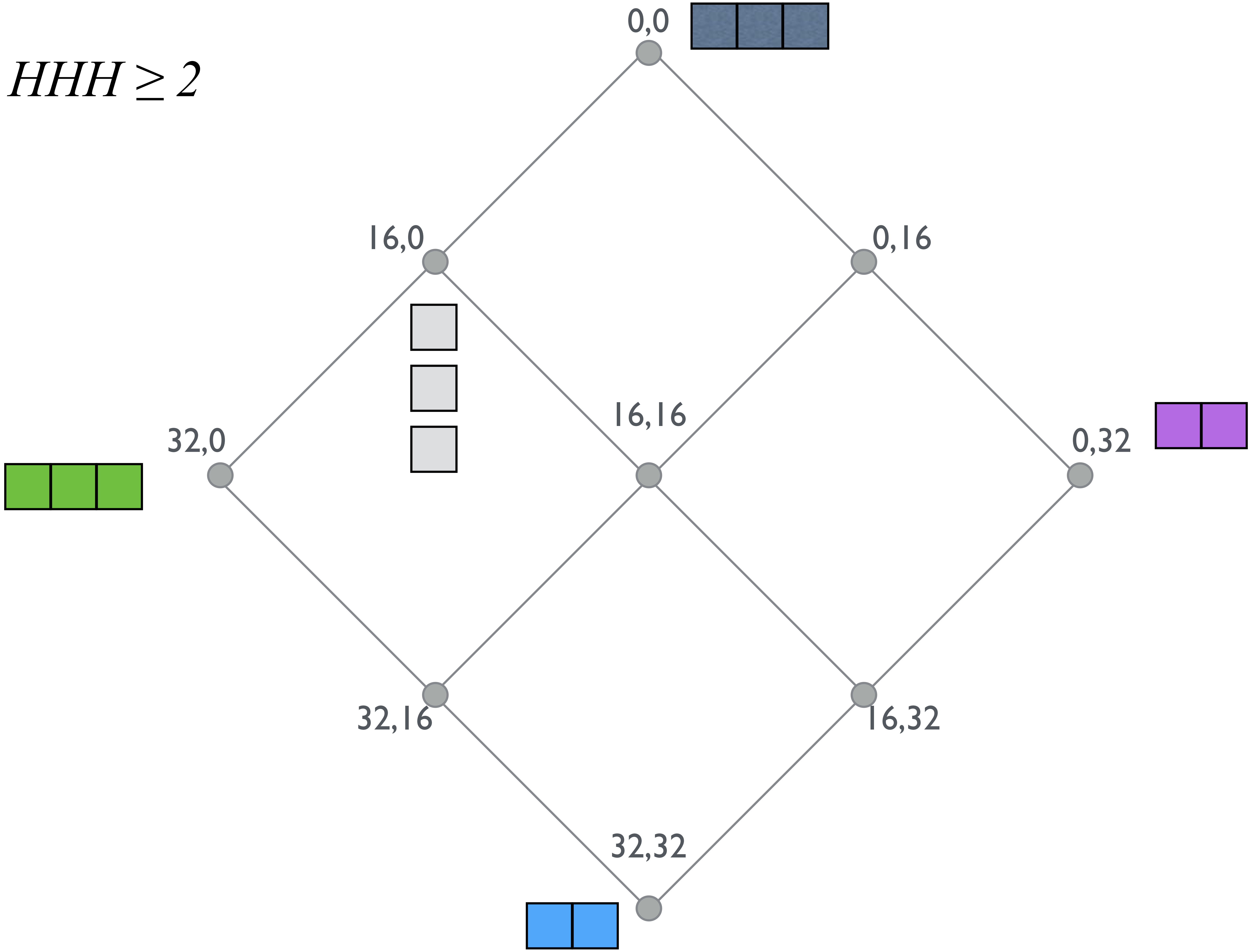
RLS Illustrated

10 inputs, HHH ≥ 2



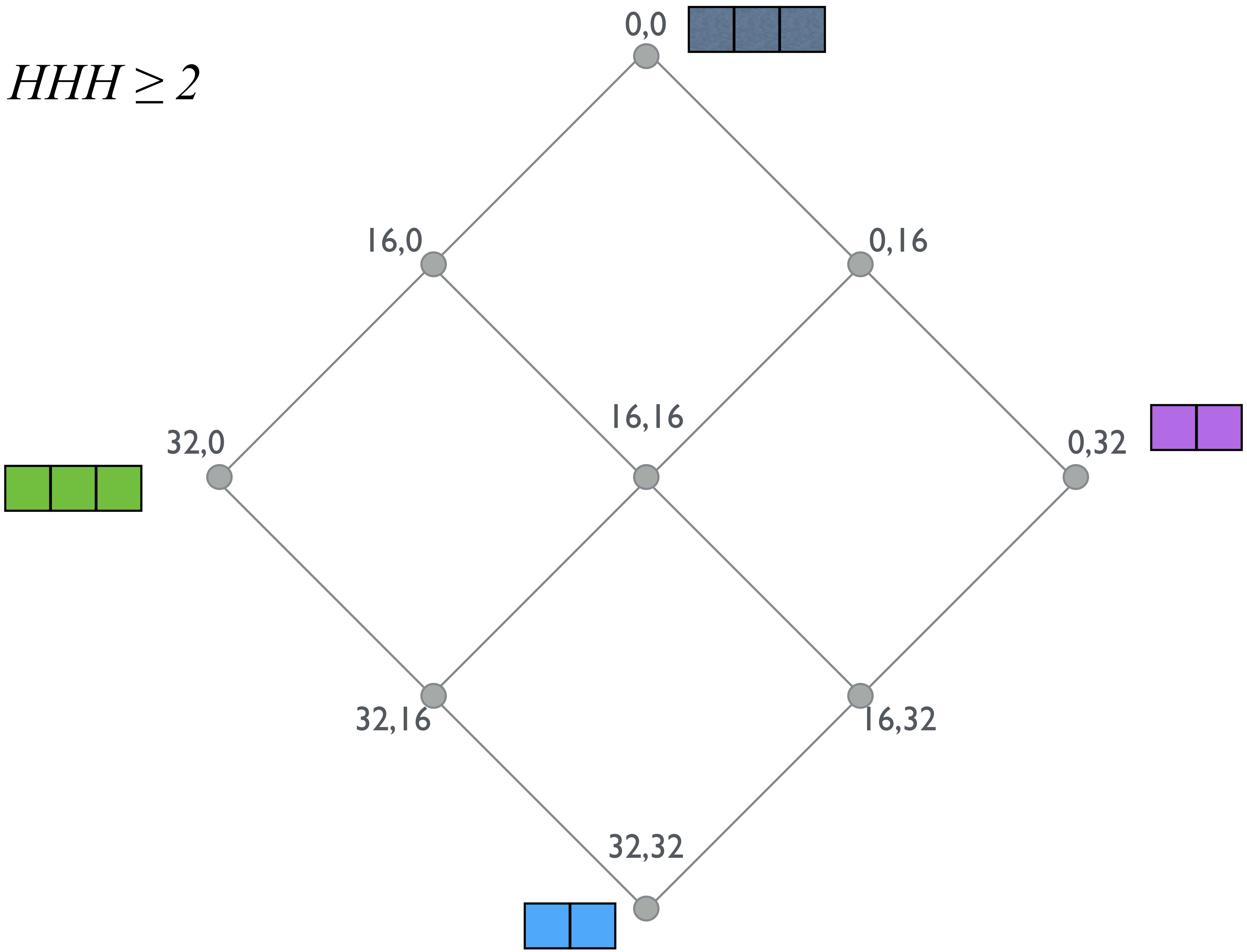
RLS Illustrated

10 inputs, HHH ≥ 2



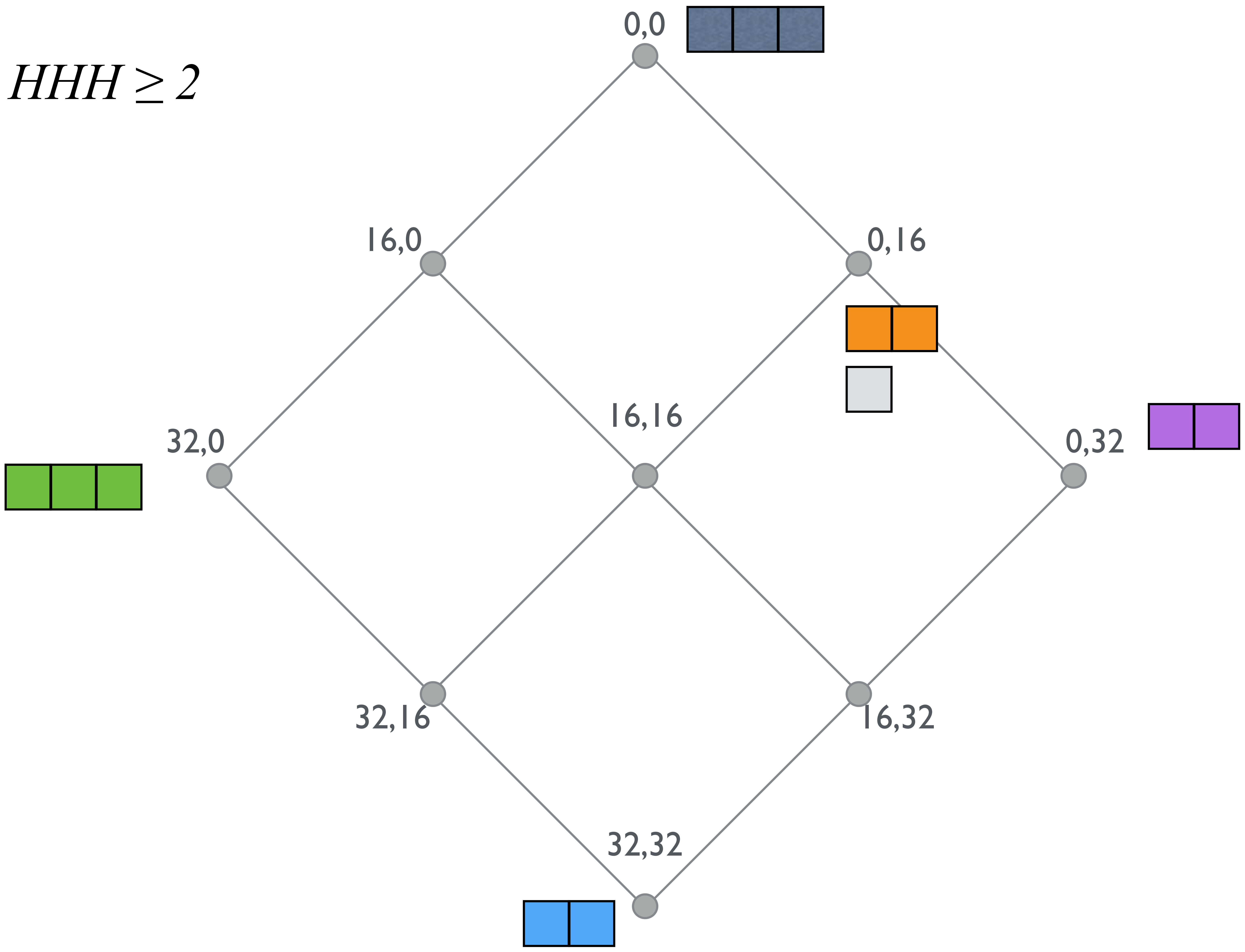
RLS Illustrated

10 inputs, HHH ≥ 2



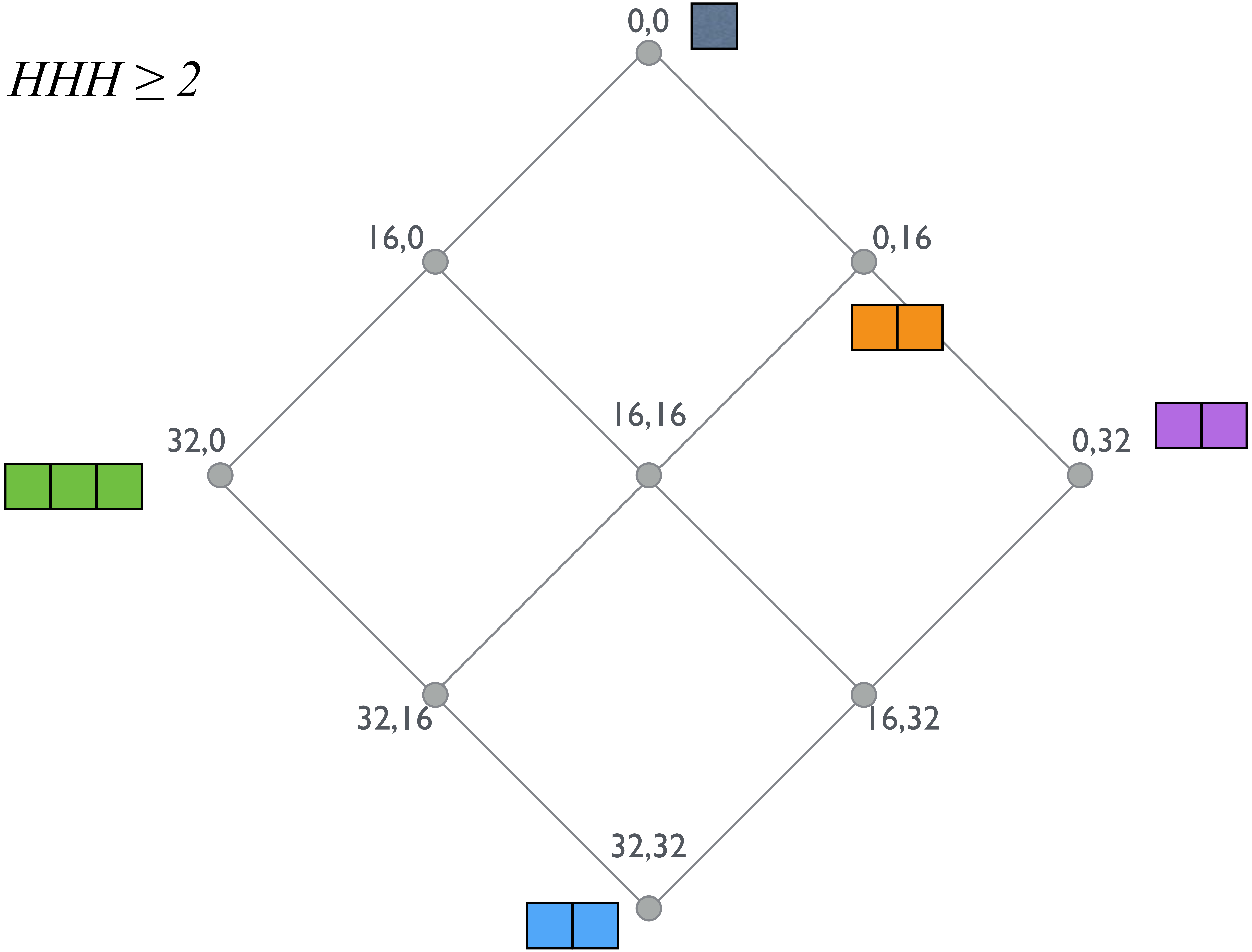
RLS Illustrated

10 inputs, HHH ≥ 2



RLS Illustrated

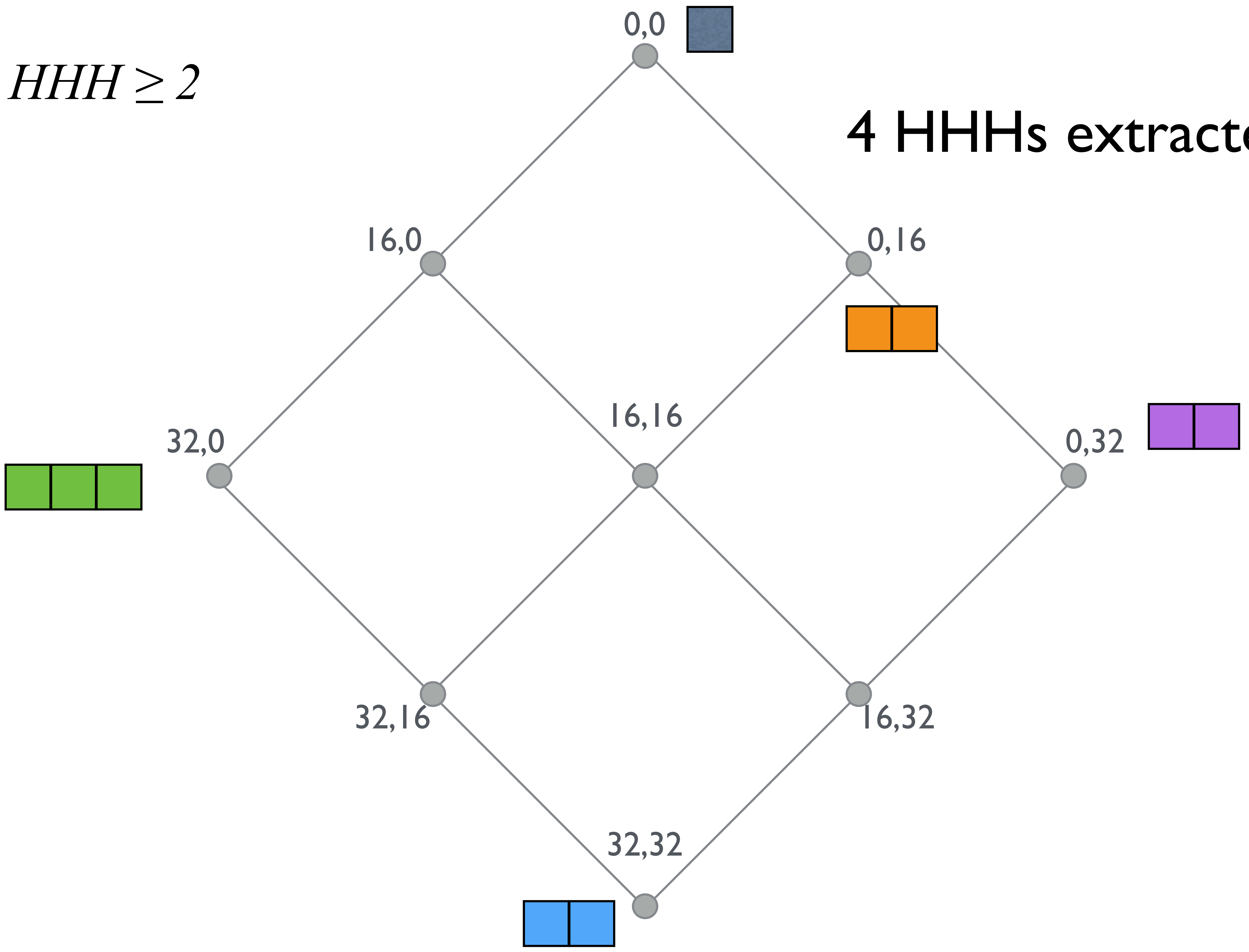
10 inputs, HHH ≥ 2



RLS Illustrated

10 inputs, HHH ≥ 2

4 HHHs extracted



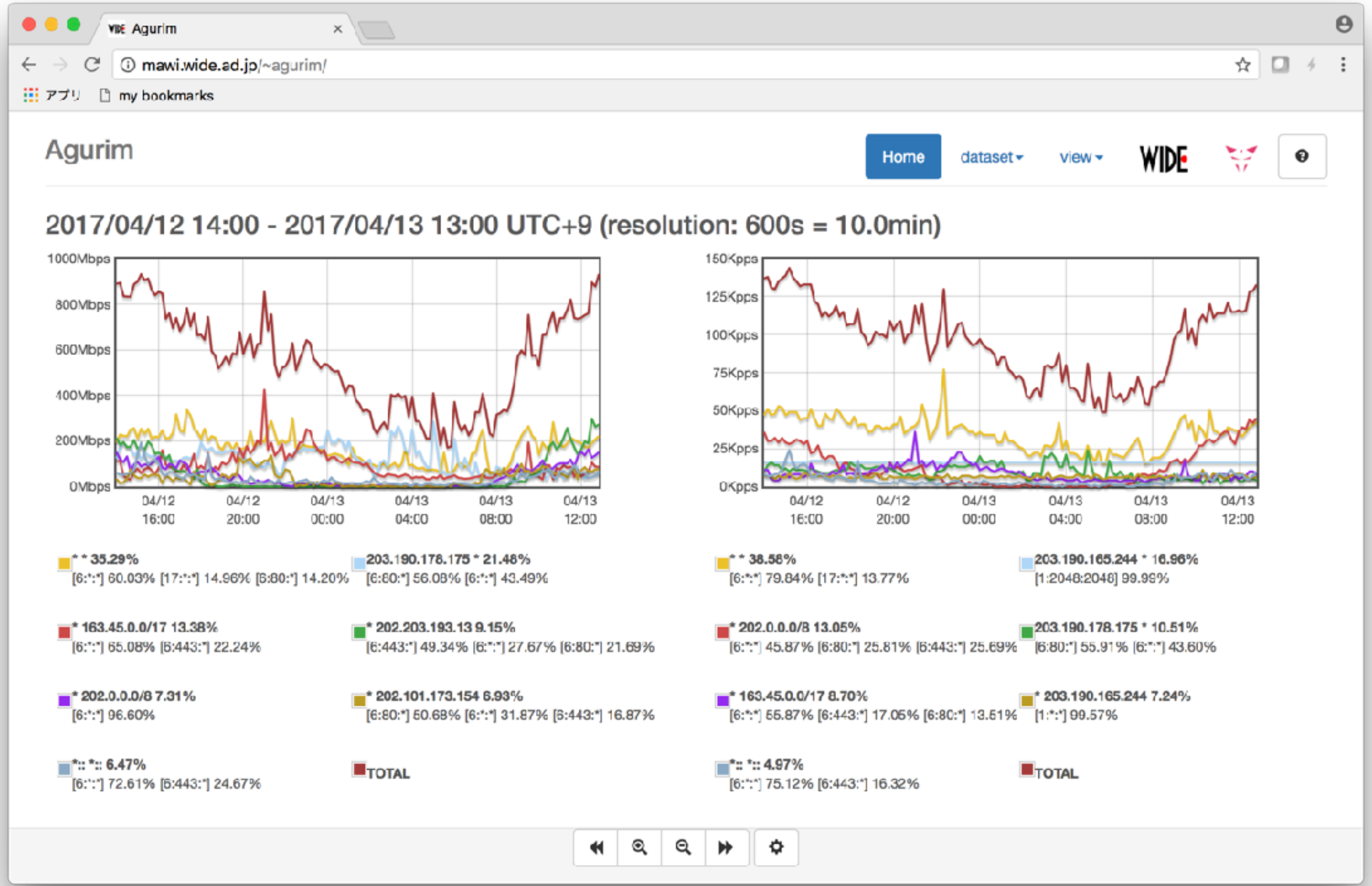
Evaluation (in the paper)

- ordering bias: (src, dst) vs (dst, src) ➡ negligible
- comparison with Space-Saving: to illustrate differences
  - outputs ➡ much more compact
    - differences due to different definitions
  - speed ➡ 100 times faster for bitwise aggregation
    - but requires more memory (as a non-streaming algo)

Implementations: RLS in agurim

- agurim: open-source tool
- 2-level HHH
  - main-attribute (src-dst adds), sub-attritbute (ports)
- protocol specific heuristics
  - change depth of recursions by protocol knowledge to meet operational needs
- online processing by exploiting multi-core CPU

agurim Web UI



http://mawi.wide.ad.jp/~agurim/

Summary

- Recursive Lattice Search algorithm for HHH
  - revisit the definition of HHH, apply Z-ordering
  - propose an efficient HHH algorithm
- open-source tool and open datasets from 2013

http://mawi.wide.ad.jp/~agurim/about.html

evaluation in detail

- simulation: code from SpaceSaving [Mitzenmacher2012]
  - quick hack to port agurim's RLS
  - input: a mawi packet trace from 2016-10-20
- order sensitivity: (src,dst) vs. (dst,src)
  - very similar outputs: not sensitive to the order
- comparing with SS (streaming algorithm, overlap rollup)
  - different definitions: just to illustrate major differences
  - outputs: comparable, except nodes in upper lattice
  - performance: 100x faster for bit-wise aggregation!

order sensitivity

(src,dst) vs. (dst,src)

region	no	src	dst	$c'/N(\%)$
		aggregated by (src,dst)		
VI	(1)	112.31.100.1/32	163.229.97.230/32	16.5
V	(2)	64.0.0.0/2	202.203.3.13/32	5.2
	(3)	128.0.0.0/1	202.203.3.13/32	5.8
	(4)	*	202.26.162.46/32	6.0
III	(5)	163.229.96.0/23	*	5.0
	(6)	203.179.128.0/20	*	6.8
II	(7)	*	202.203.3.0/24	5.9
	(8)	*	203.179.140.0/23	5.7
	(9)	*	163.229.128.0/17	5.1
I	(10)	0.0.0.0/1	202.192.0.0/12	5.3
	(11)	202.192.0.0/12	*	6.7
	(12)	*	202.0.0.0/7	7.6
	(13)	**128.0.0.0/4**	*	**5.0**
	(14)	**128.0.0.0/2**	*	**6.0**
	(15)	*	**128.0.0.0/2**	**5.4**
-		*	*	**2.0**
				100.0
		aggregated by (dst,src)		
	(1)-(12)	identical to (src,dst)		
I	(13)	**128.0.0.0/2**	**0.0.0.0/2**	**5.7**
	(14)	*	**128.0.0.0/3**	**5.3**
	(15)	**128.0.0.0/1**	*	**6.4**
-		*	*	**1.0**

- (1)-(12): identical
- (13)-(15): minor difference
- not sensitive to src-dst order

HHHs reported by RLS vs. SS

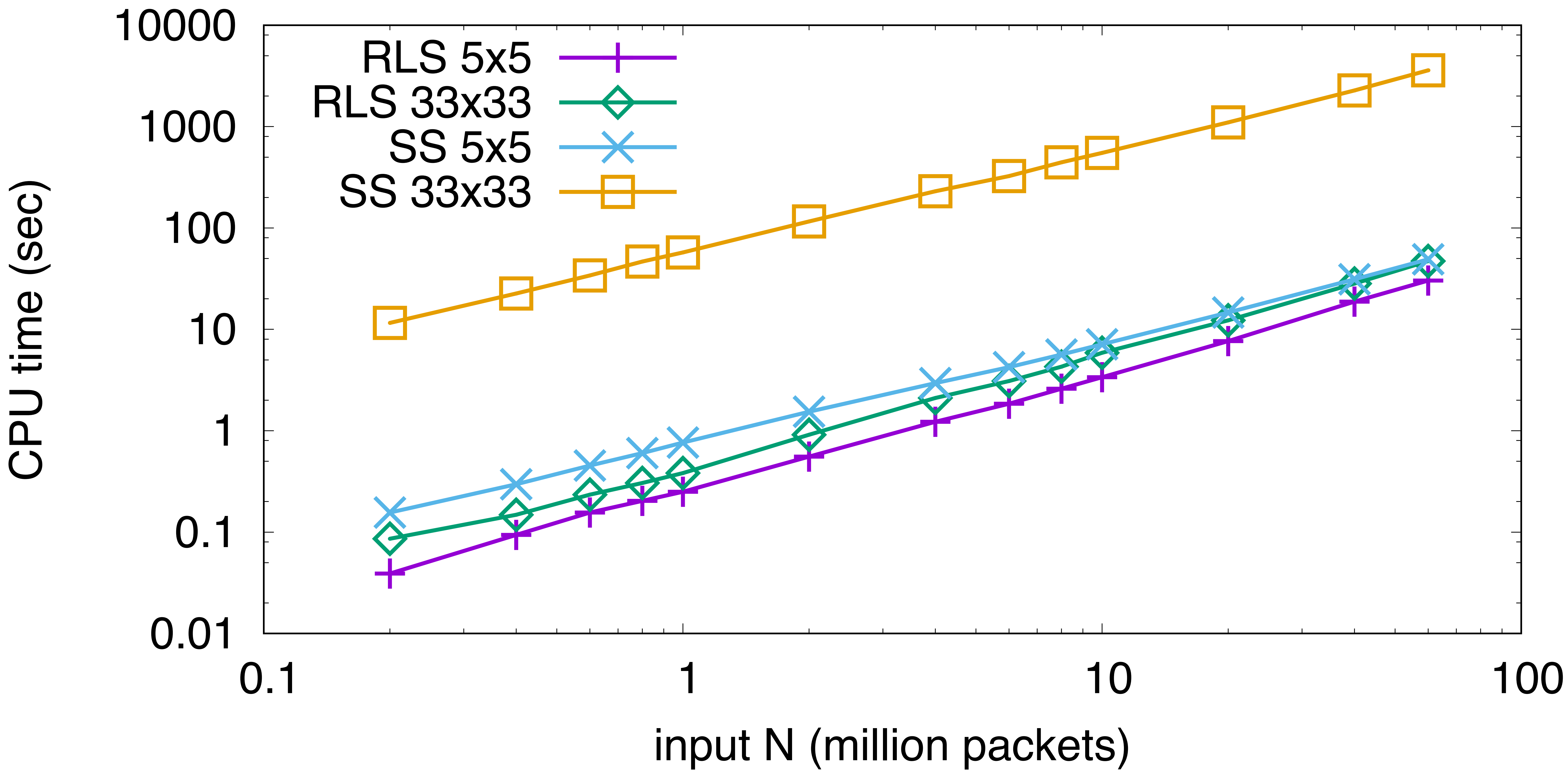
- # of HHHs: RLS:15, SS:52
- missing HHHs: not informative
  - double-counting / short prefix lengths
  - 40 missing HHHs, 35 in (I), 4 in (II), 1 in (III)
- RLS: concise and compact summary

region	no	aggregated by (src,dst) src	dst	c'/N(%)
VI	(1)	112.31.100.1/32	163.229.97.230/32	16.5
V	(2)	64.0.0.0/2	202.203.3.13/32	5.2
	(3)	128.0.0.0/1	202.203.3.13/32	5.8
	(4)	*	202.26.162.46/32	6.0
III	(5)	163.229.96.0/23	*	5.0
	(6)	203.179.128.0/20	*	6.8
II	(7)	*	202.203.3.0/24	5.9
	(8)	*	203.179.140.0/23	5.7
	(9)	*	163.229.128.0/17	5.1
I	(10)	0.0.0.0/1	202.192.0.0/12	5.3
	(11)	202.192.0.0/12	*	6.7
	(12)	*	202.0.0.0/7	7.6
	(13)	**128.0.0.0/4**	*****	**5.0**
	(14)	**128.0.0.0/2**	*	**6.0**
	(15)	*	**128.0.0.0/2**	**5.4**
-		*	*	**2.0**
				100.0

no	RLS(%)	SS(%)	missing SS HHHs with their c'/N(%)
(1)	16.5	16.5	-
(2)	5.2	5.2	-
(3)	5.8	5.8	-
(4)	6.0	6.0	-
(5)	5.0	5.0	-
(6)	6.8	6.8	-
(7)	**5.9**	**16.9**	-
(8)	5.7	5.7	-
(9)	5.1	5.1	-
(10)	**5.3**	-	(96/3,202.203/16):5.4 (0/2,202.203/16):5.6 (112/4,202.192/12):5.2 (64/2,202.192/12):9.0
(11)	6.7	6.7	-
(12)	**7.6**	-	(0/1,203.179.128/20):6.0 (128/2,202.203/16):5.5 (192/4,202/8):5.1 (*,202.192/12):25.5 (16/4,202/7):5.4 (128/1,202.128/9):10.6 (64/2,202/7):15.5 (128/1,202/7):17.7
(13)	**5.0**	**5.2**	-
(14)	**6.0**	-	(163.229/16,0/1):6.0 (144/4,128/1):5.3 (128/2,96/3):5.0 (128/3,0/1):5.3 (160/3,128/1):7.0 (128/2,0/2):5.7 (128/2,0/1):11.4
(15)	**5.4**	**33.1**	(128/1,160/6):5.0 (192/4,128/2):5.2 (0/1,128/2):22.7 (*,128/3):7.1
-	**2.0**	-	(202/7,0/2):5.4 (192/8,128/1):5.6 (202/8,0/1):5.7 (202/7,128/1):6.0 (192/3,200/5):10.5 (128/1,112/6):5.1 (112/5,128/1):21.8 (200/5,*):17.0 (192/4,128/1):13.6 (128/1,16/4):6.2 (*,200/5):42.4 (64/3,128/1):6.0 (96/3,128/1):29.7 (128/1,64/2):10.4 (0/1,128/1):46.7 (128/1,*):53.3 (*,128/1):78.3

CPU time: RLS vs. SS

- RLS: lower cost for finer granularity
  - 100+ times faster for bit-wise aggregation!



memory usage: RLS vs. SS

- RLS: proportional to inputs (ok for modern PCs)
- SS: fixed memory usage