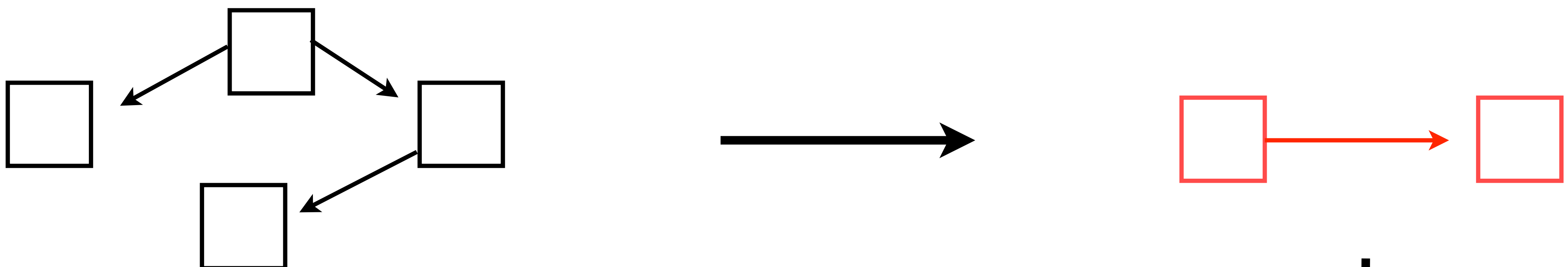
1.6 Bit Pattern Databases

Teresa M. Breyer and **Richard E. Korf**
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
{tbreyer,korf}@cs.ucla.edu

Presentation by Damian Murezzan

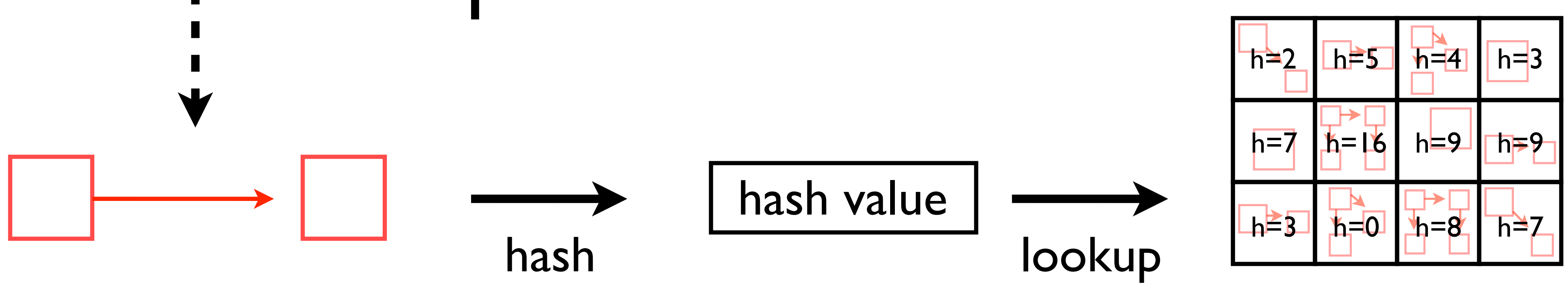
What is a pattern database?

pattern:

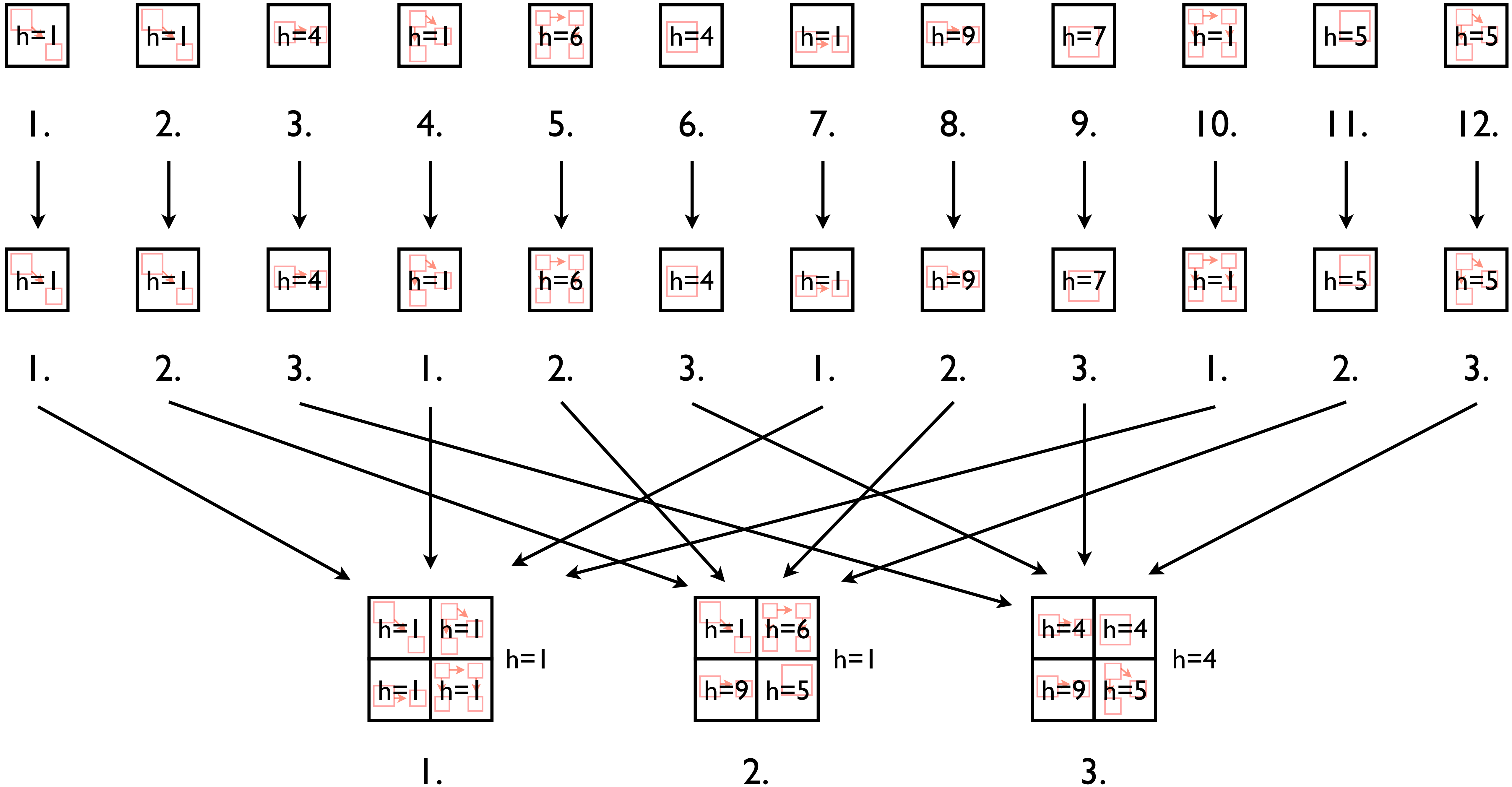


Projection of a state onto pattern space

pattern database:



Compressed Pattern Databases



How should we group the patterns?

How should we group the patterns?

Different methods, eg. cliques introduced by Felner et al. in 2007.

Cliques are a set of patterns reachable from each other by only one move, and stores the minimum value of this set.
→ Introduces a error of at most one move.

One can also define cliques k moves apart. Then the maximum error is increased to k.

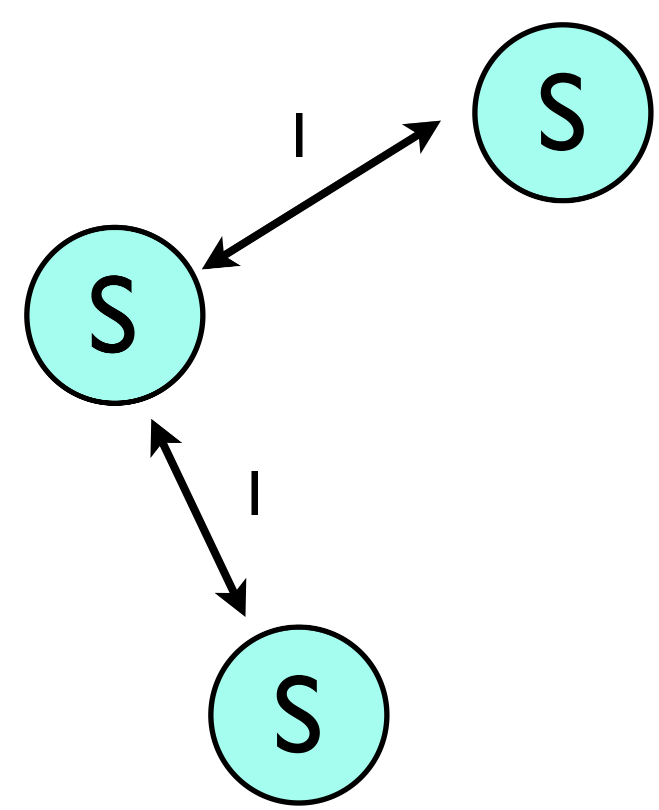
A example for a clique: smallest disk pattern in the 4-peg Tower of Hanoi problem form cliques of size 4. This results in a compression of factor 4.

Constructing Two-Bit Pattern Databases

Use mod three breath First Search in the pattern space with the goal pattern as the root to construct a **lossless** compressed PDB which uses only two bits per entry.

Requirements:

-Unit Edge Costs
-Reversible Operators
-Consistent Heuristic



Mod Three Breath-First Search

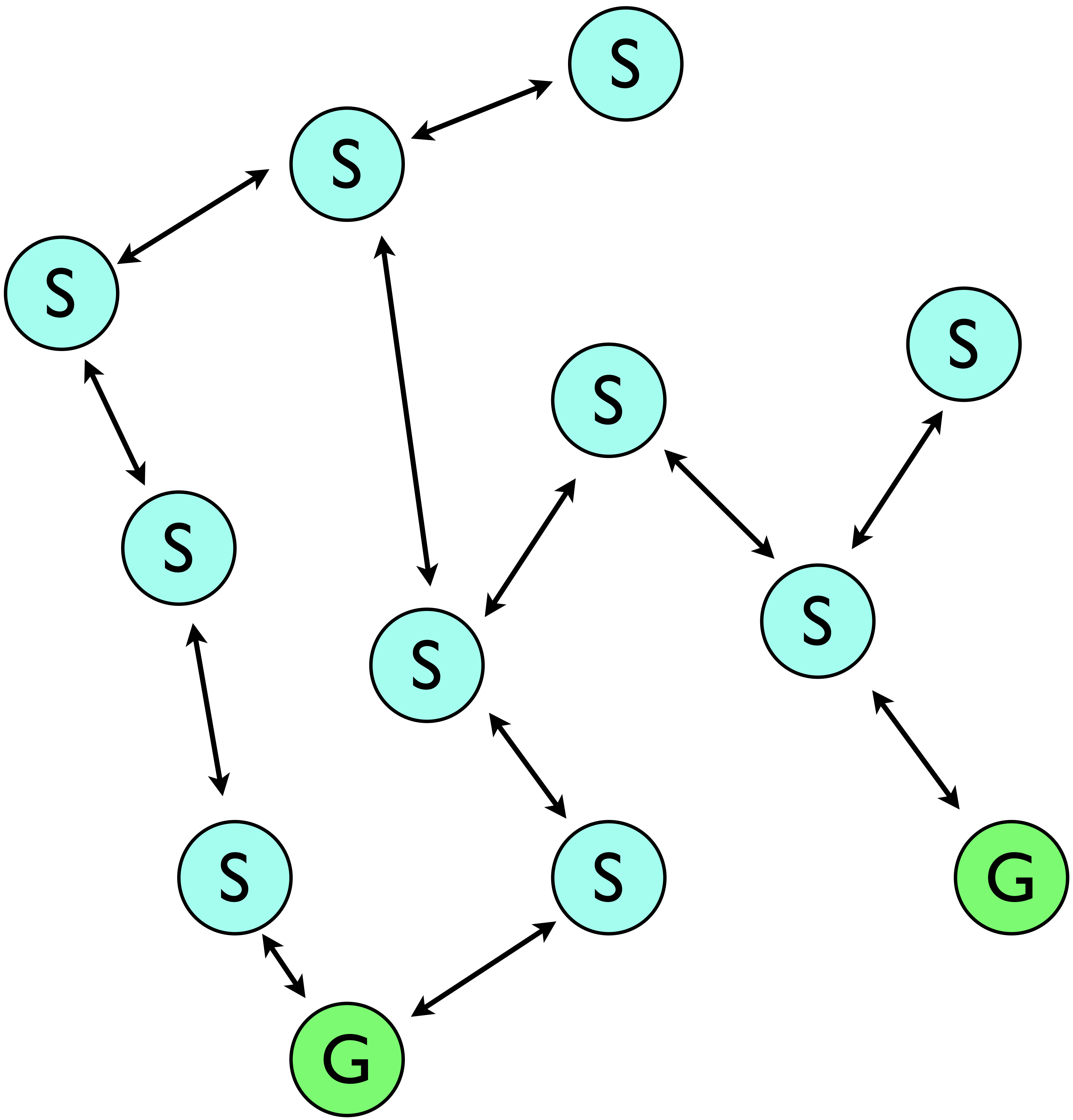
Introduced by Cooperman and Finkelstein in 1992.

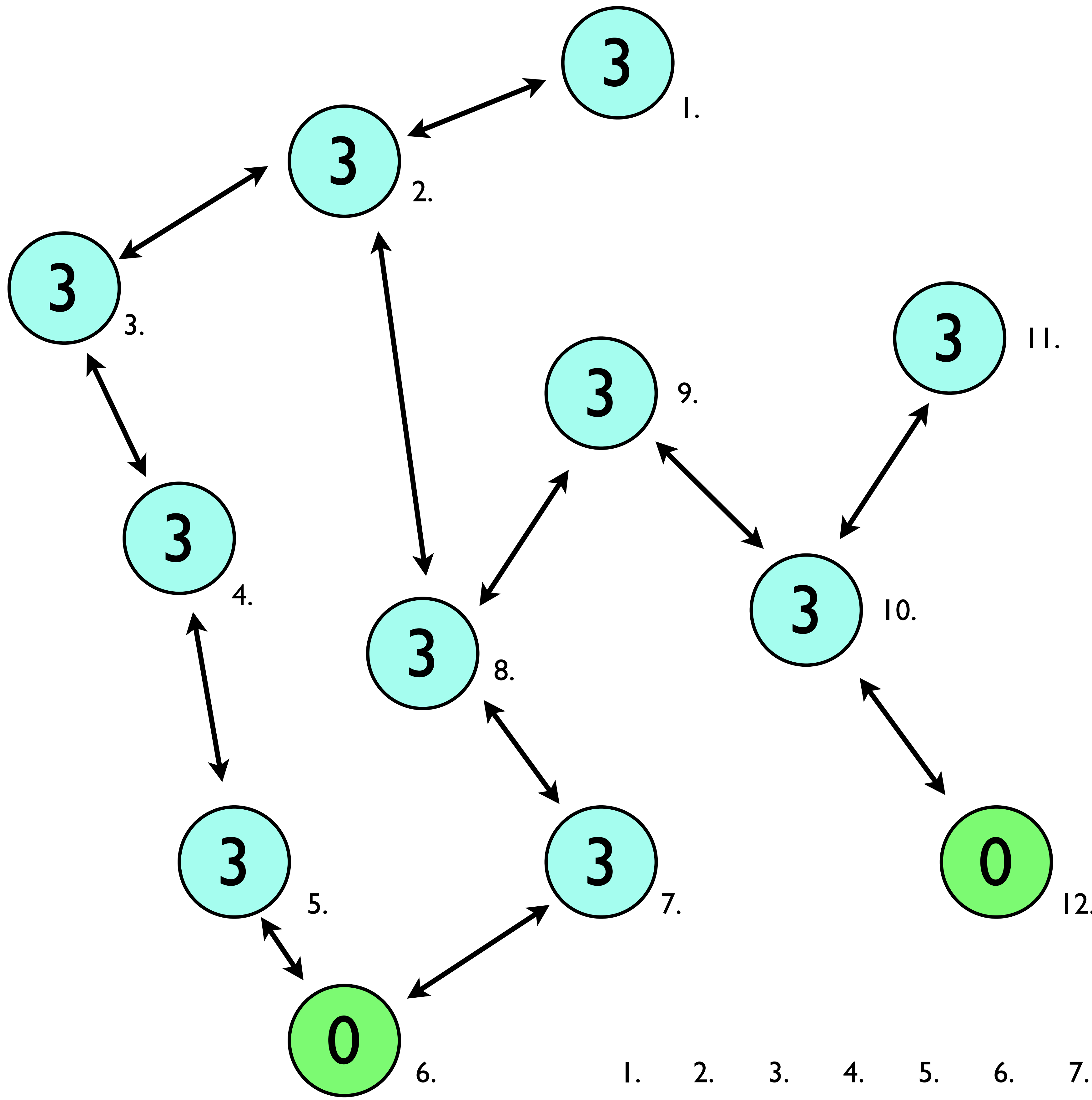
Method for constructing a hash table which can give us informations about the location of a node inside a graph and its distance to the root.

Uses a perfect hash function to assign hash table entries to states.

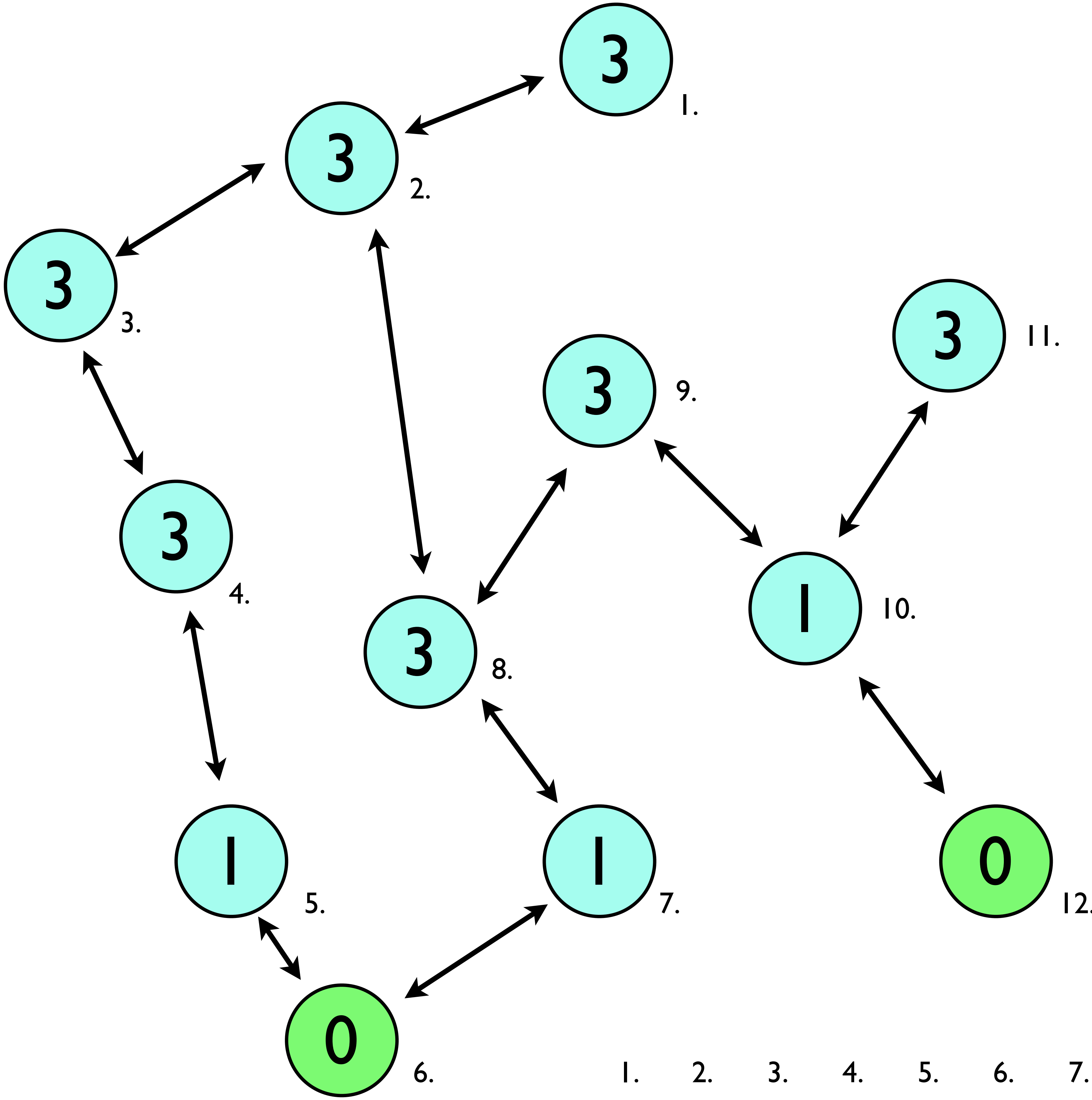
Gives each entry a value between 0 and 2, uses the value 3 as indicator whether a state was expanded already or not.

This means each entry in the hash tabe has one of 4 different values and can be stored in two bits.

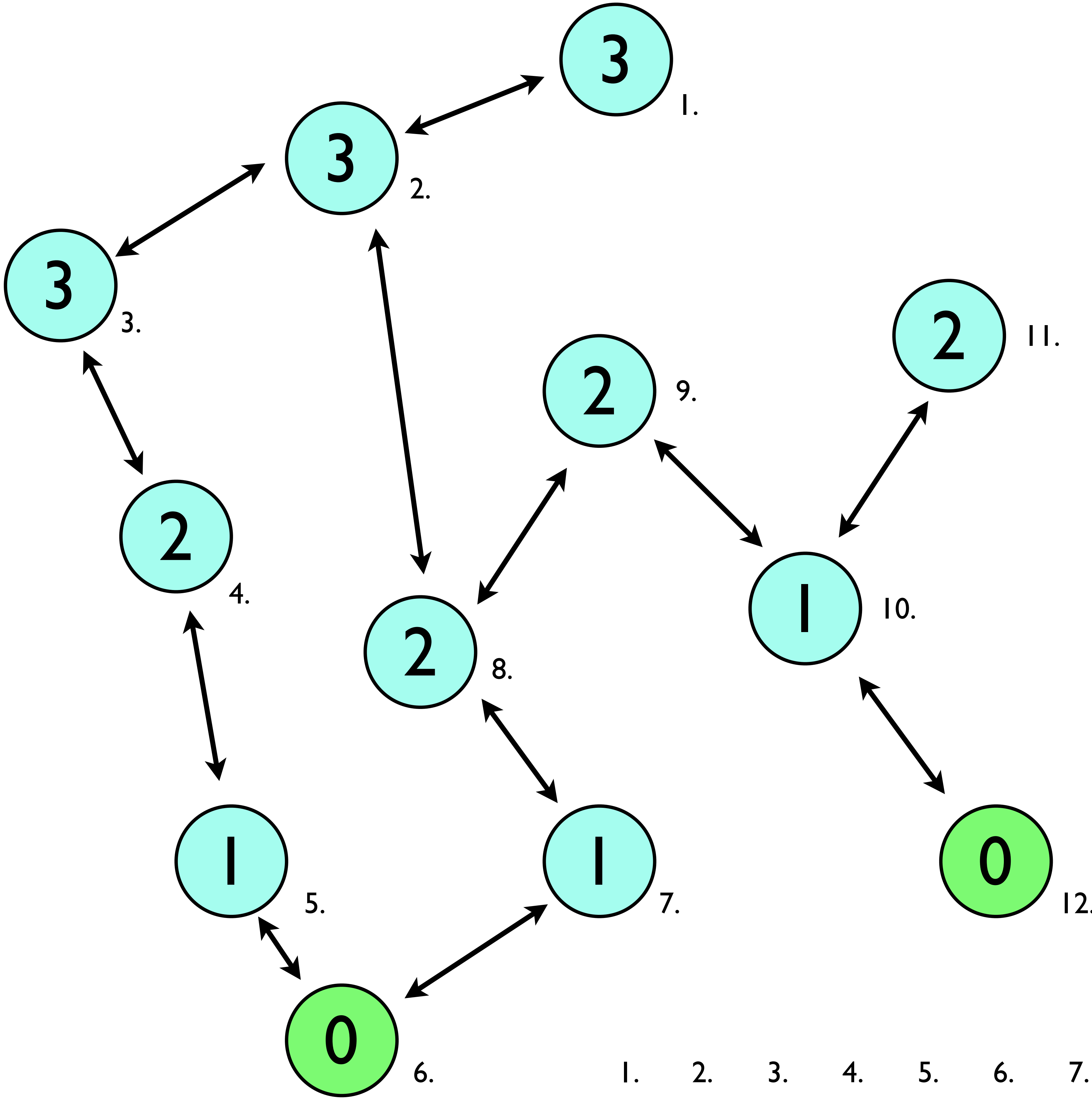

S
S
S
S
S
S
S
S
S
S
G
G



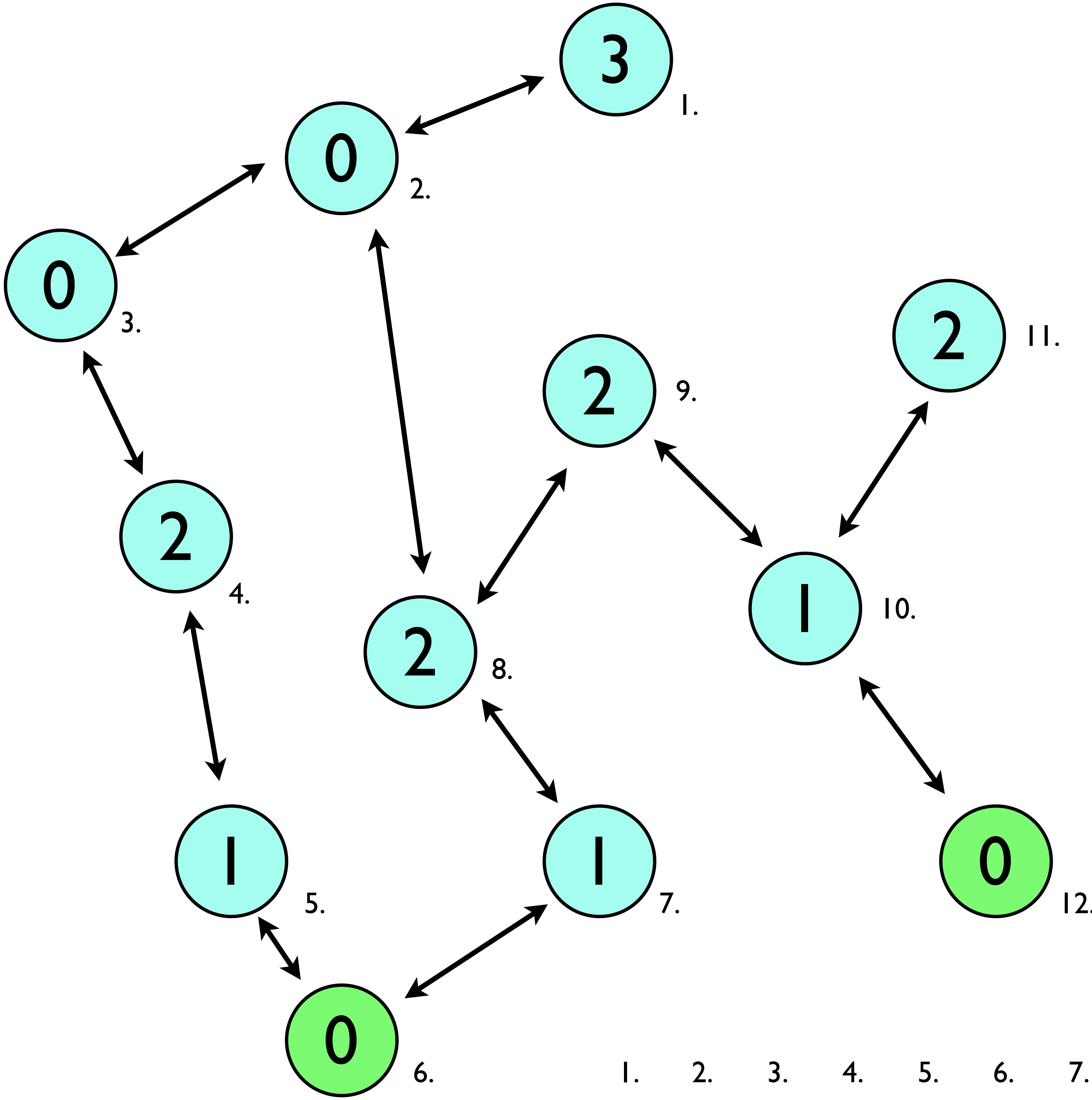
1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
3	3	3	3	3	0	3	3	3	3	3	0



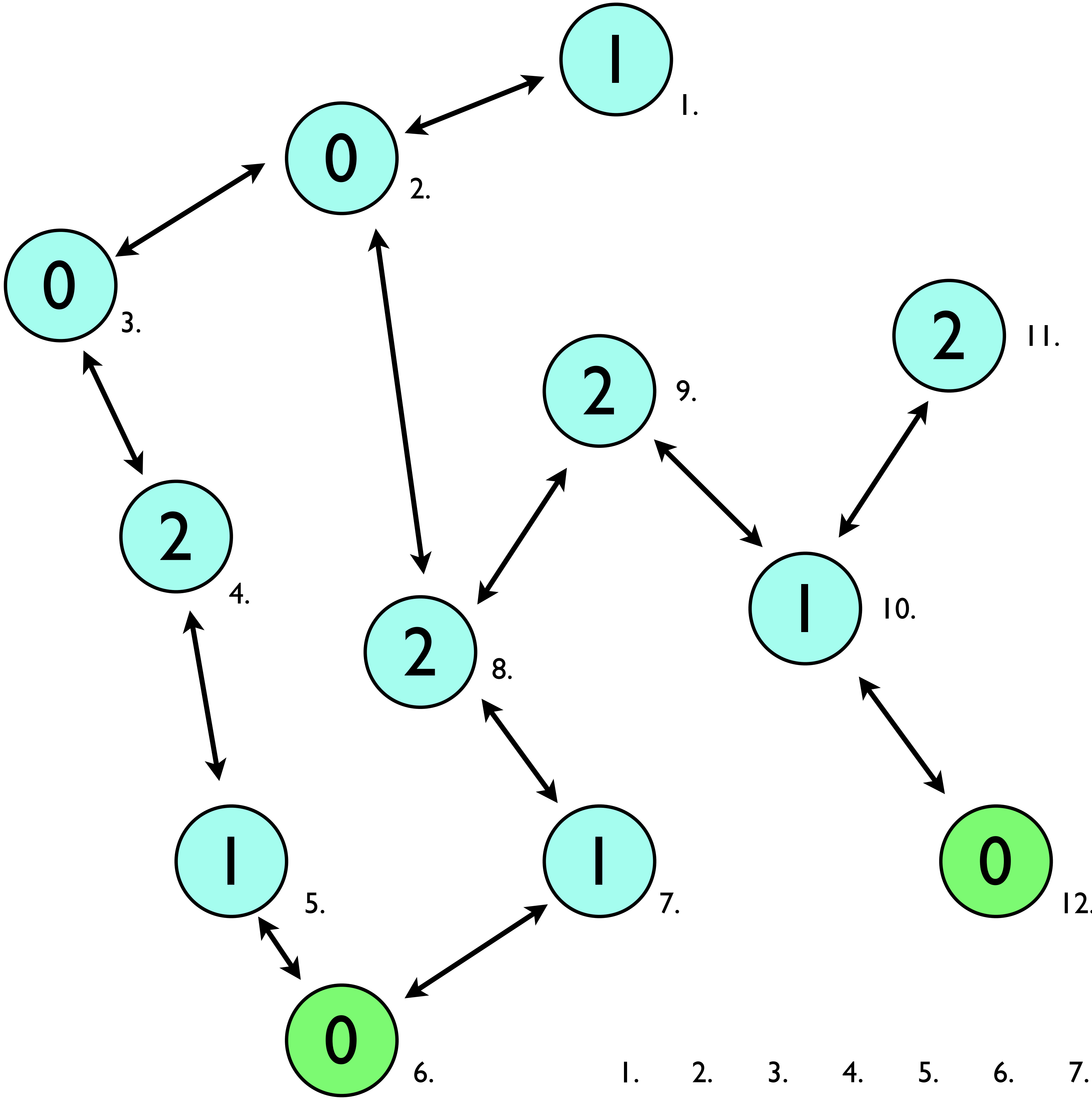
1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
3	3	3	3	1	0	1	3	3	1	3	0



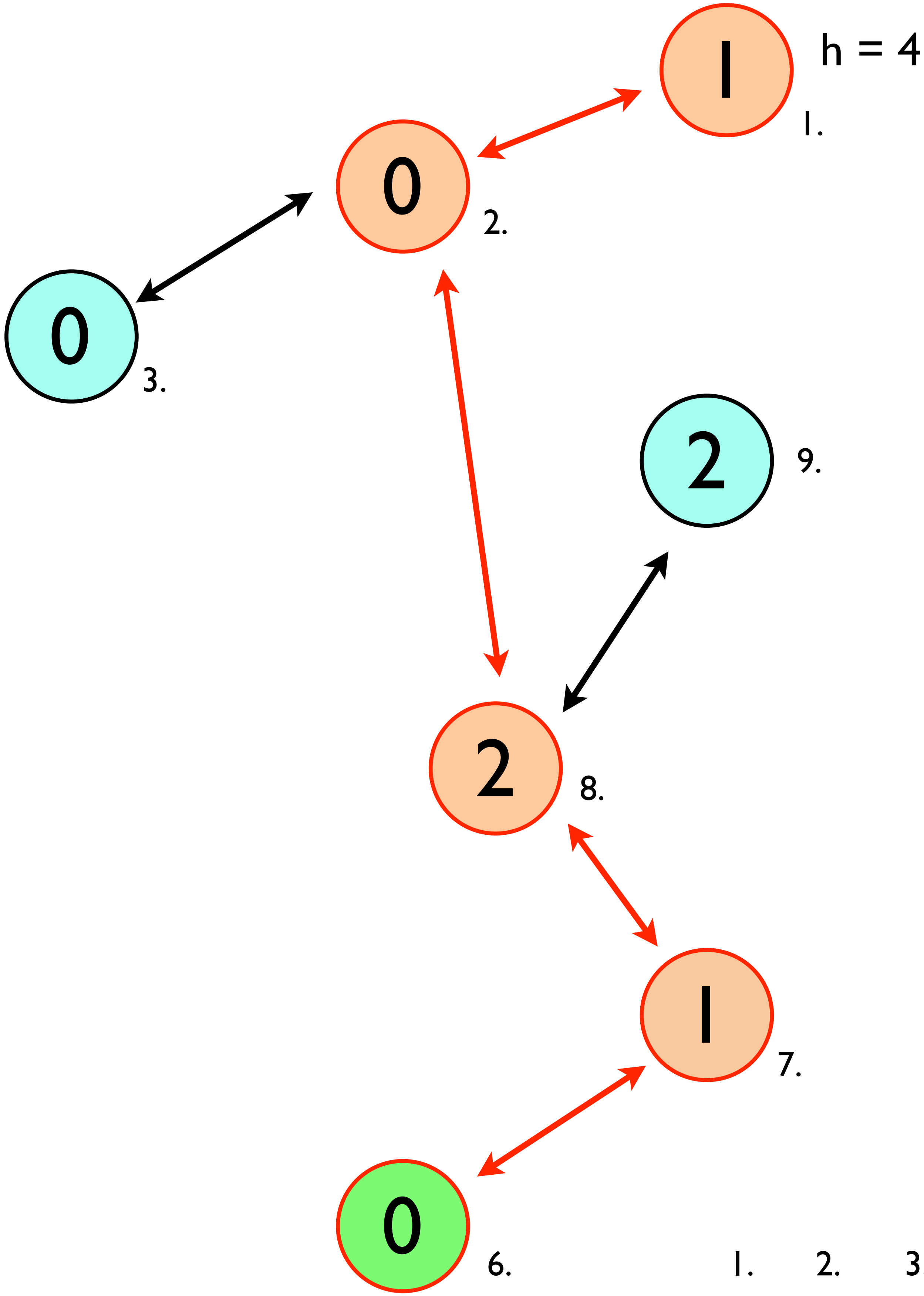
1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
3	3	3	2	1	0	1	2	2	1	2	0



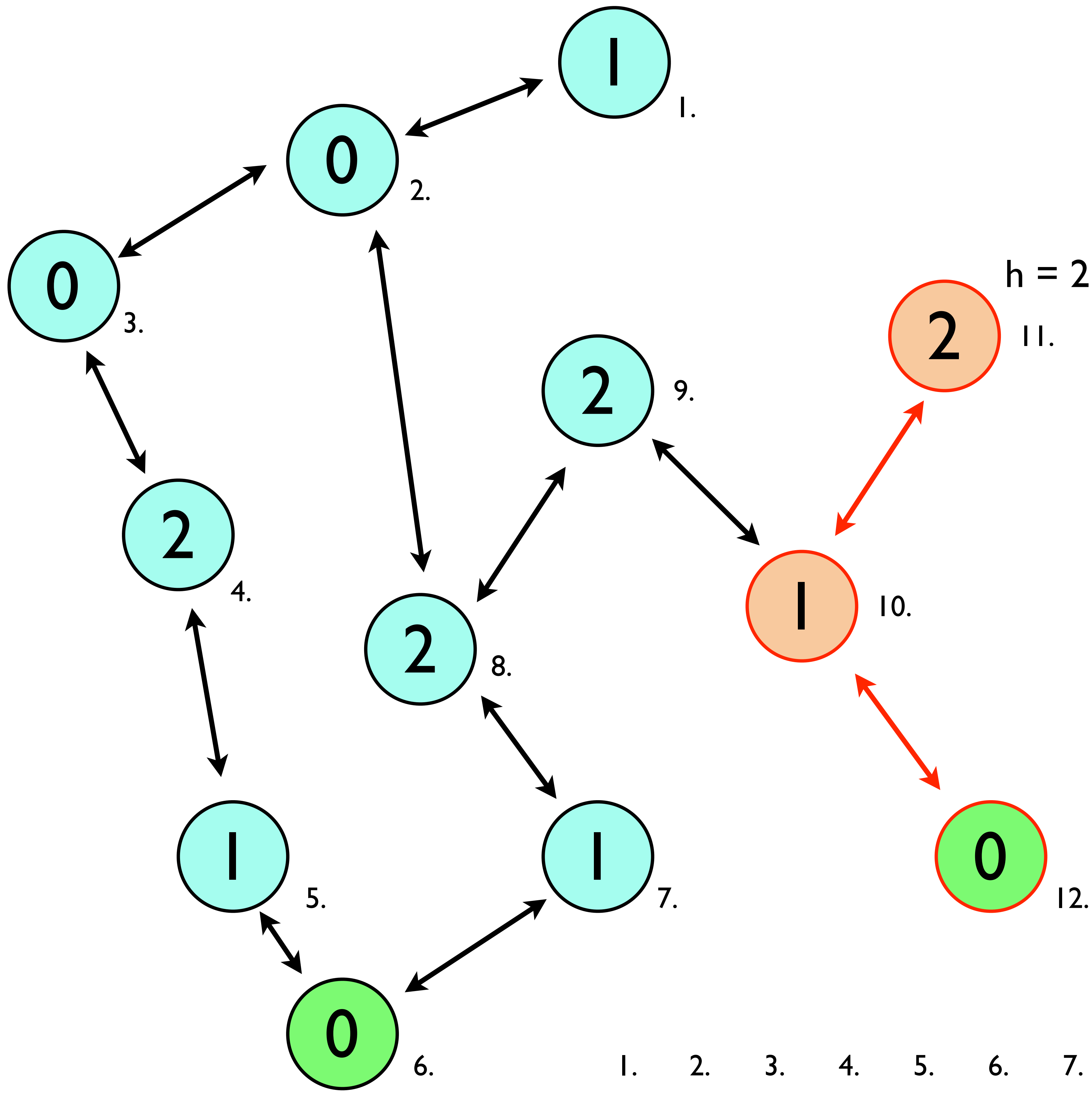
1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
3	0	0	2	1	0	1	2	2	1	2	0



1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
0	0	0	2	1	0	1	2	2	1	2	0

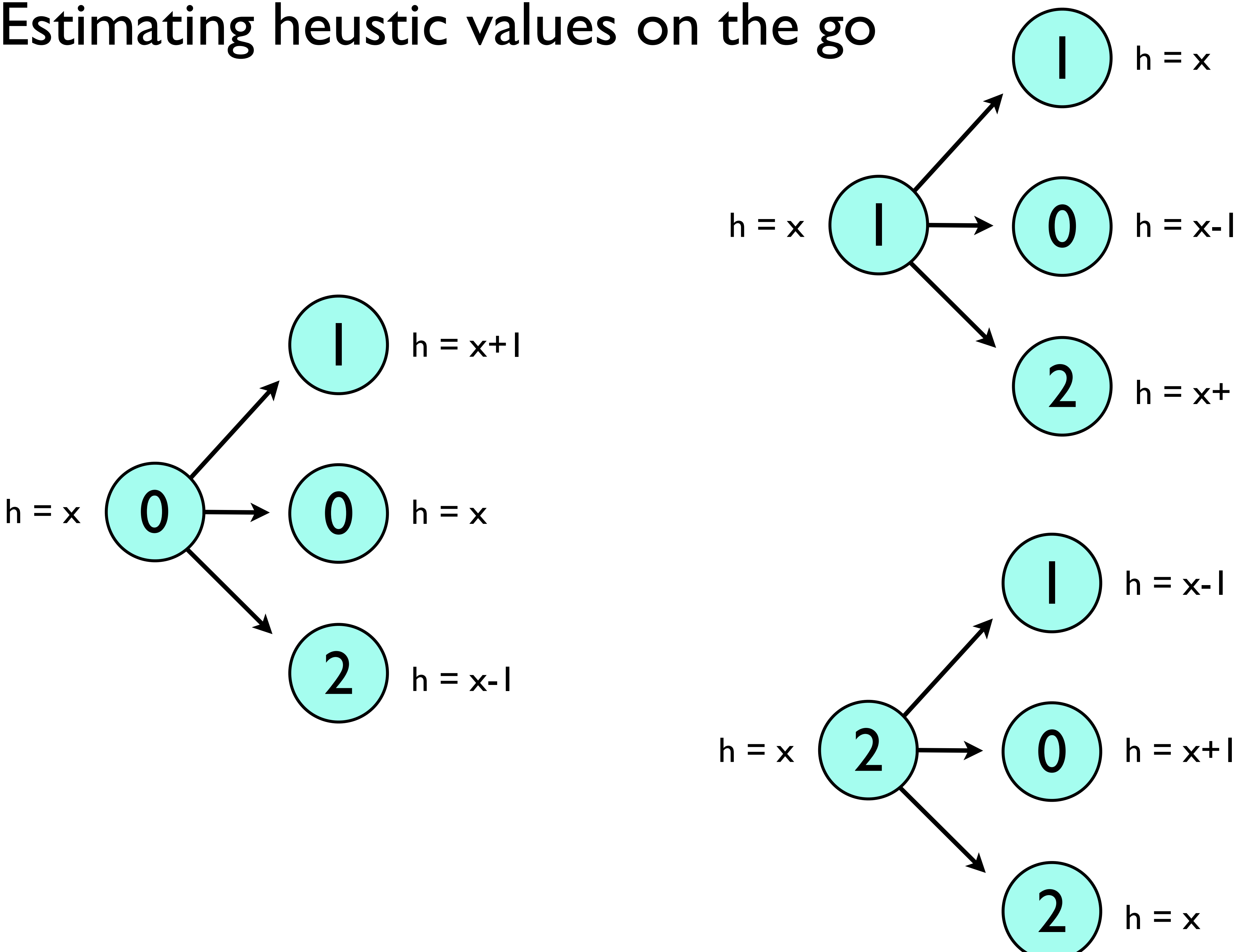


1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
0	0	0	2	1	0	1	2	2	1	2	0



1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.
0	0	0	2	1	0	1	2	2	1	2	0

Estimating heustic values on the go



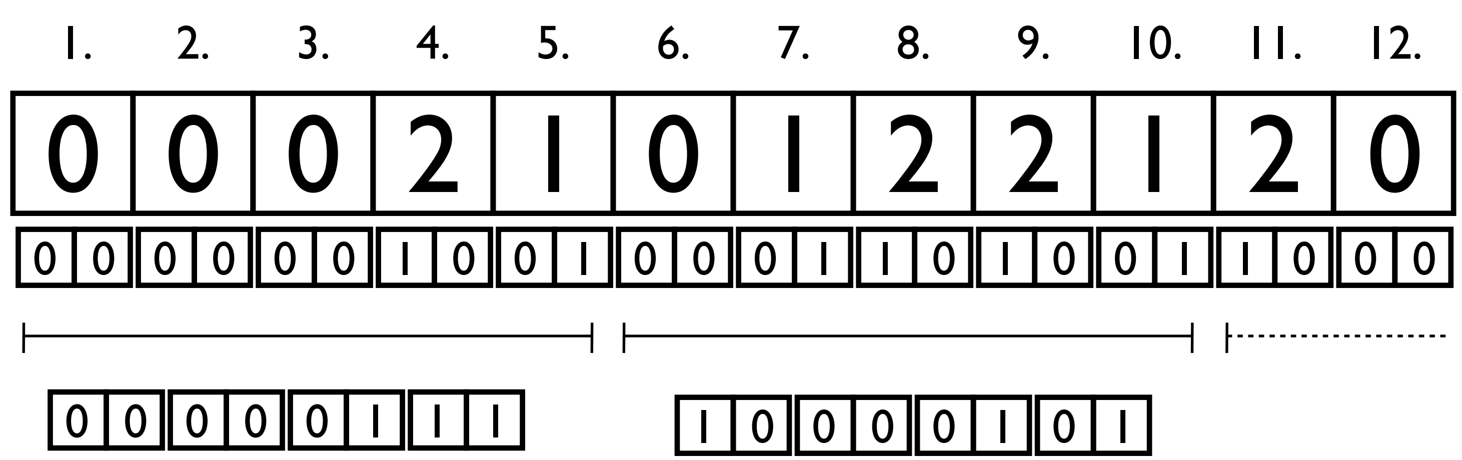
1.6 Bit Pattern Databases

Its possible to compress the two bit database even further, because only three values are required for storing the heuristic values modulo three.

We do this by fitting as many pattern as possible in one byte.

Since 1 byte equals to 256 different possible values, the largest base 3 number that can be stored inside a byte is 243, which is 3^5. This means, we can fit 5 modulo 3 values in one byte.

Thus, we compress our two bit database even further by a factor of five and use now only 1.6 bits per pattern.



1.6 Bit Pattern Databases

Disadvantages:

Slightly more exprensive for lookup (Needs integer division by 5 and modulo operator instead of shift and bitwise operator needed for 2 Bit databases)

We need a seperate table for checking which patterns have been generated.

Theoretically we would only need $(n*\log_2 3)/8$ bits, this means we would use 1.58 bits per state saved. The problem is, the lookup will get too expensive and the resulting gain in memory is marginal. The whole PDB would be stored as one large number and we would have to extract our heuristic estimates from that number.

Results - (17,4) Top Spin Puzzle

#	Comp.	$h(s)$	Generated	Time	Size
1	None	10.53	43,607,741	12.25	247
2	Two Bit	10.53	43,607,741	12.83	123
3	Mod 2	10.16	55,244,961	16.10	123
4	1.6 Bit	10.53	43,607,741	13.57	99
5	Mod 2.5	9.97	62,266,443	18.46	99

Table 1: Solving the (17,4) Top-Spin puzzle using a 9-token PDB

#	Comp.	Heur.	$h(s)$	Generated	Time	Size
1	Two Bit	8r+0d	12.37	11,103	0.016	990
2	None	4r+4d+c	11.53	76,932	0.080	247
3	Two Bit	4r+4d+c	12.39	10,188	0.631	990

Table 2: Solving the $(17,4)$ Top-Spin puzzle using a 10-token two-bit PDB or, a uncompressed 9-token PDB

-Limit of 2GB RAM

-Tow Bit / 1.6 Bit are the techniques presented in this paper

- Mod 2 / Mod 2.5 (applies the Modulo Function to the Hash value, only minimum heuristic value is saved)

Results - (17,4) Top Spin Puzzle

#	Comp.	$h(s)$	Generated	Time	Size
1	None	10.53	43,607,741	12.25	247
2	Two Bit	10.53	43,607,741	12.83	123
3	Mod 2	10.16	55,244,961	16.10	123
4	1.6 Bit	10.53	43,607,741	13.57	99
5	Mod 2.5	9.97	62,266,443	18.46	99

Table 1: Solving the (17,4) Top-Spin puzzle using a 9-token PDB

#	Comp.	Heur.	$h(s)$	Generated	Time	Size
1	Two Bit	8r+0d	12.37	11,103	0.016	990
2	None	4r+4d+c	11.53	76,932	0.080	247
3	Two Bit	4r+4d+c	12.39	10,188	0.631	990

Table 2: Solving the $(17, 4)$ Top-Spin puzzle using a 10-token two-bit PDB or, a uncompressed 9-token PDB

In Table 2, the two bit algorithm can use a 10-token PDB, while the uncompressed PDB has to use a 9-token pdb due to memory constraints.

1.6 Bit wasn‘t used because 11-token PDB‘s couldnt be realized by either compression variant.

Results - Rubik‘s Cube

#	Comp.	$h(s)$	Generated	Time	Size
1	None	9.1	102,891,122,415	32,457	529
2	Two-Bit	9.1	102,891,122,415	32,113	265
3	1.6-Bit	9.1	102,891,122,415	35,190	212
4	$8\text{-}8_{10}\text{-}8_{10}$	9.1	105,720,641,791	36,385	529
5	Dual	9.1	65,932,517,927	27,150	529
6	(8-7-7-7-7)	9.1	64,713,886,881	27,960	529

Table 3: Solving Korf's ten initial states of Rubik's cube using a 8-corner-cubie and two 7-edge-cubie PDBs

#	Comp.	$h(s)$	Generated	Time	Size
1	Two-Bit	9.5	26,370,698,776	11,290	1,239
2	Div 2	9.3	56,173,197,862	25,917	1,239
3	Mod 2	9.3	58,777,491,012	27,577	1,239
4	1.6-Bit	9.5	26,370,698,776	12,309	991
5	Div 2.5	9.1	68,635,164,093	33,838	991
6	Mod 2.5	9.0	77,981,222,043	35,976	991
7	Dual	9.1	65,932,517,927	27,150	529
8	Two-Bit (8-8-8-8-8)	9.7	14,095,769,007	8,667	1,239

Table 4: Solving Korf's ten initial states of Rubik's cube using a 8-corner-cubie and two 8-edge-cubie or, with dual lookups, two 7-edge-cubie PDBs

Results - Rubik‘s Cube

#	Comp.	$h(s)$	Generated	Time	Size
1	None	9.1	102,891,122,415	32,457	529
2	Two-Bit	9.1	102,891,122,415	32,113	265
3	1.6-Bit	9.1	102,891,122,415	35,190	212
4	$8\text{-}8_{10}\text{-}8_{10}$	9.1	105,720,641,791	36,385	529
5	Dual	9.1	65,932,517,927	27,150	529
6	(8-7-7-7-7)	9.1	64,713,886,881	27,960	529

Table 3: Solving Korf’s ten initial states of Rubik’s cube using a 8-corner-cubie and two 7-edge-cubie PDBs

#	Comp.	$h(s)$	Generated	Time	Size
1	Two-Bit	9.5	26,370,698,776	11,290	1,239
2	Div 2	9.3	56,173,197,862	25,917	1,239
3	Mod 2	9.3	58,777,491,012	27,577	1,239
4	1.6-Bit	9.5	26,370,698,776	12,309	991
5	Div 2.5	9.1	68,635,164,093	33,838	991
6	Mod 2.5	9.0	77,981,222,043	35,976	991
7	Dual	9.1	65,932,517,927	27,150	529
8	Two-Bit (8-8-8-8-8)	9.7	14,095,769,007	8,667	1,239

Table 4: Solving Korf’s ten initial states of Rubik’s cube using a 8-corner-cubie and two 8-edge-cubie or, with dual lookups, two 7-edge-cubie PDBs

Two bit and 1.6 bit compression generate the same amounth of nodes as the uncompressed PDB, but need much less memory, while adding little overhead.

Results - Rubik‘s Cube

#	Comp.	$h(s)$	Generated	Time	Size
1	None	9.1	102,891,122,415	32,457	529
2	Two-Bit	9.1	102,891,122,415	32,113	265
3	1.6-Bit	9.1	102,891,122,415	35,190	212
4	$8\text{-}8_{10}\text{-}8_{10}$	9.1	105,720,641,791	36,385	529
5	Dual	9.1	65,932,517,927	27,150	529
6	(8-7-7-7-7)	9.1	64,713,886,881	27,960	529

Table 3: Solving Korf’s ten initial states of Rubik’s cube using a 8-corner-cubie and two 7-edge-cubie PDBs

#	Comp.	$h(s)$	Generated	Time	Size
1	Two-Bit	9.5	26,370,698,776	11,290	1,239
2	Div 2	9.3	56,173,197,862	25,917	1,239
3	Mod 2	9.3	58,777,491,012	27,577	1,239
4	1.6-Bit	9.5	26,370,698,776	12,309	991
5	Div 2.5	9.1	68,635,164,093	33,838	991
6	Mod 2.5	9.0	77,981,222,043	35,976	991
7	Dual	9.1	65,932,517,927	27,150	529
8	Two-Bit (8-8-8-8-8)	9.7	14,095,769,007	8,667	1,239

Table 4: Solving Korf’s ten initial states of Rubik’s cube using a 8-corner-cubie and two 8-edge-cubie or, with dual lookups, two 7-edge-cubie PDBs

Table 4 shows the cababilites of two bit and 1.6 bit PDB‘s for rubik‘s cube.

If we compare them with other compression methods, we see that the other methods need to expand much more nodes due to their lossy compression.

Results - Rubik's Cube

#	Comp.	$h(s)$	Generated	Time	Size
1	None	9.1	102,891,122,415	32,457	529
2	Two-Bit	9.1	102,891,122,415	32,113	265
3	1.6-Bit	9.1	102,891,122,415	35,190	212
4	$8\text{-}8_{10}\text{-}8_{10}$	9.1	105,720,641,791	36,385	529
5	Dual	9.1	65,932,517,927	27,150	529
6	(8-7-7-7-7)	9.1	64,713,886,881	27,960	529

Table 3: Solving Korf's ten initial states of Rubik's cube using a 8-corner-cubie and two 7-edge-cubie PDBs

#	Comp.	$h(s)$	Generated	Time	Size
1	Two-Bit	9.5	26,370,698,776	11,290	1,239
2	Div 2	9.3	56,173,197,862	25,917	1,239
3	Mod 2	9.3	58,777,491,012	27,577	1,239
4	1.6-Bit	9.5	26,370,698,776	12,309	991
5	Div 2.5	9.1	68,635,164,093	33,838	991
6	Mod 2.5	9.0	77,981,222,043	35,976	991
7	Dual	9.1	65,932,517,927	27,150	529
8	Two-Bit (8-8-8-8-8)	9.7	14,095,769,007	8,667	1,239

Table 4: Solving Korf's ten initial states of Rubik's cube using a 8-corner-cubie and two 8-edge-cubie or, with dual lookups, two 7-edge-cubie PDBs

In row 7/8, we can see that our Two Bit PDB is 4 times faster than the current state of the art algorithm if we use the available memory most efficently.

The Dual solver could use only a 7-edge cubie PDB due to the 2GB memory constraint, while Two Bit could use 8 edge cubies.

Results - 18 Disk Tower of Hanoi

#	Comp.	$h(s)$	Generated	Time	Size
1	Two-Bit	164	355,856,206	333	1,024
2	16_1	163	373,045,641	355	1,024
3	1.6-Bit	164	355,856,206	336	820
4	16_2	161	400,505,833	387	256
5	16_3	159	443,154,284	443	64

Table 5: Solving the 18-disc Towers of Hanoi problem using a 16-disc PDB

-16 disk PDBs used

-16_n is lossy compression using cliques, n denotes the number of ignored smallest disks

-16_1, althought lossy, generates not that much more nodes

-16_n is capable to compress even further than 1.6 bit

Results

- The authors introduced a lossless compression which is able to store a consistent heuristics in just 2 or 1.6 bits per state.

- Improvements are largely given by the type of problem. For Rubik‘s cube and the top spin puzzle, there are large benefits for using this compression method. 4-peg towers of hanoi and sliding-tile puzzles show only marginal or no improvements at all.

-Two or 1.6 Bit Pattern Databases are useful for problems where lossy compression leads to a very high number of additional expansions. This will happen wenn we can‘t use cliques or if adjacent entries in the PDB are not highly correlated.

-The most benefit we can get from compression methods is fitting a better PDB in the same space as a uncompressed worse PDB.