Early work on PDBs

Ariel Felner
SISE Department
Ben-Gurion University
ISRAEL
felner@bgu.ac.il

Fifteen Puzzle (1869)

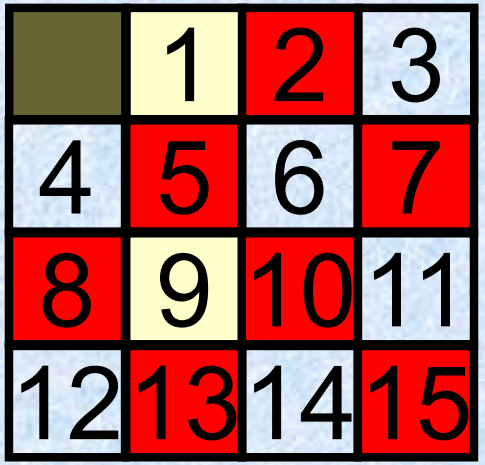
	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- Easy to manipulate and encode
- Has 10^{13} states
- Has long history of serving as testbed for heuristic search
- Baseline heuristic: **Manhattan distance**
- Solved optimally with IDA* [Korf 1985]

Domains

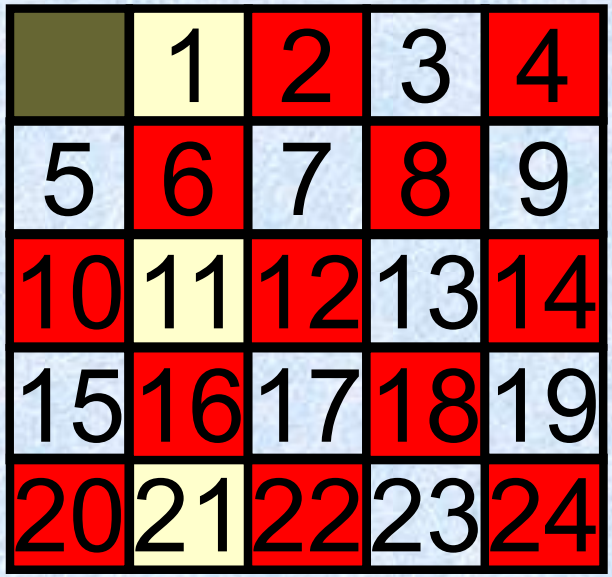
15 puzzle

- 10^13 states
- First solved by [Korf 85] with IDA* and Manhattan distance
- Takes 53 seconds

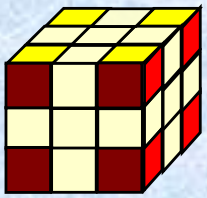


24 puzzle

- 10^24 states
- First solved by [Korf 96]
- Takes two days



B	x	x	3
x	x	x	7
x	x	x	11
12	13	14	15



The year 1999

Ariel

Richard Korf

How do we find strong heuristics for the Tile Puzzle?

How can we combine knowledge from different PDBs?

Three ideas

1) Disjoint additive PDBs [AIJ 2002]

Statically-partitioned PDBs (2000)

Dynamically-partitioned-PDBs [JAIR 2004]

Vertex-cover table [never published] (2001)

2) Compressed PDBs

Entry compression (2004-2007)

Value compression (2017)

Bloom filters (2014)

Learning PDBs (2008)

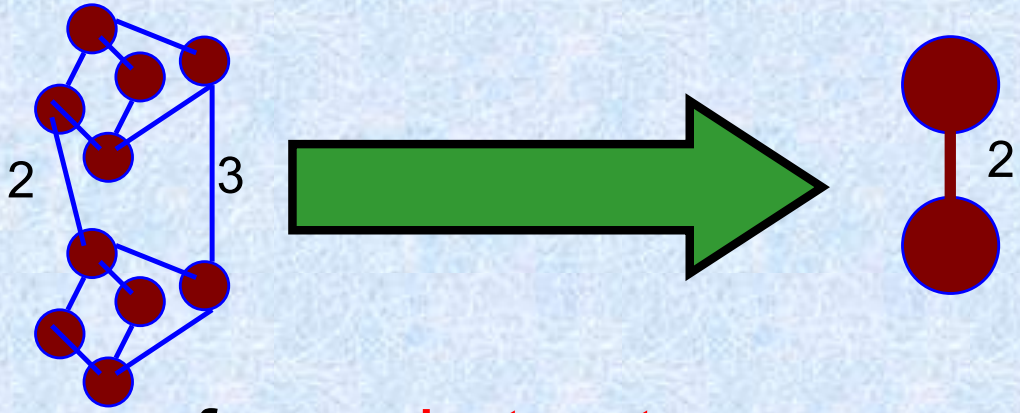
3) Dual-lookups in PDBs and Inconsistency (2005)

Older work on PDBS

In the nineties

Homomorphic Abstractions

- Many search spaces can be abstracted by merging nodes into abstract nodes



- Distances from abstract spaces are lower bounds for the original problem

1) PDBs for the 15-puzzle

[Culberson and Schaeffer 96]

- Many problems can be abstracted into subproblems that must be also solved.
- A solution to the subproblem is a lower bound on the entire problem.

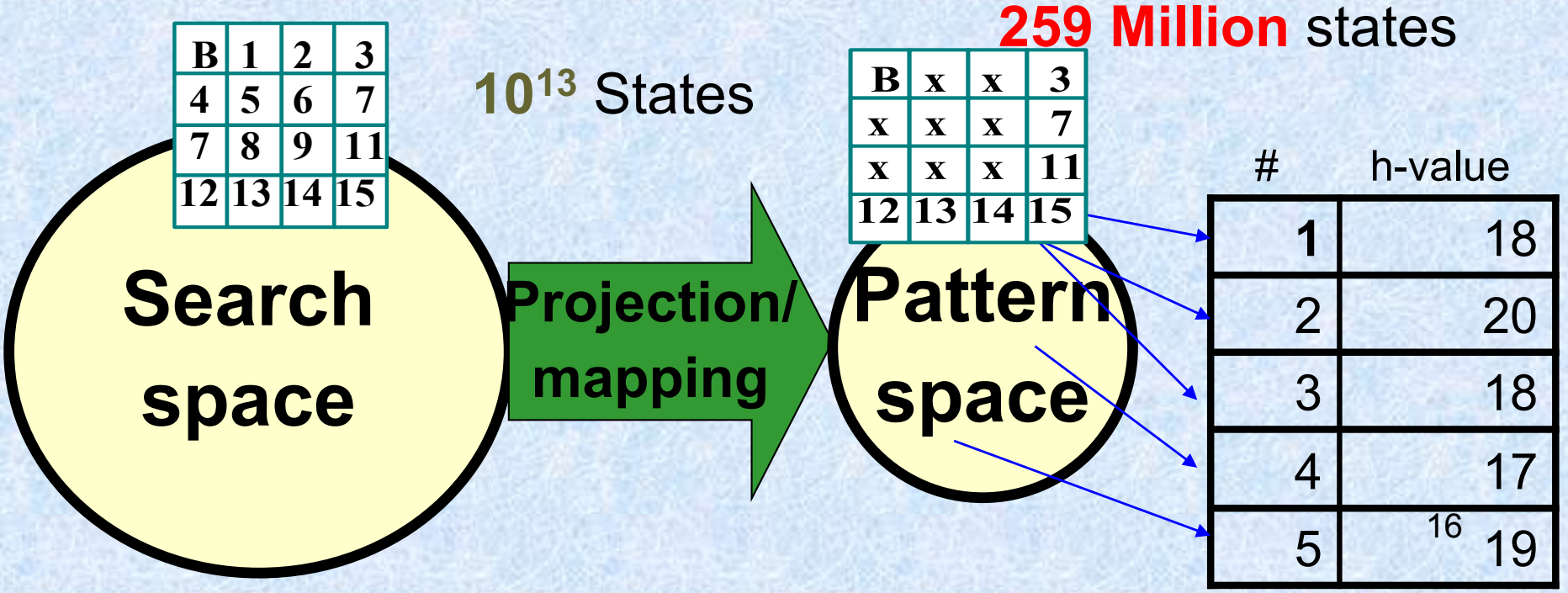
B	x	x	3
x	x	x	7
x	x	x	11
12	13	14	15

13	x	B	x
	x	x	x

Idea was inspired by endgame databases from Checkers

Pattern Databases heuristics

- A **pattern database (PDB)** is a **lookup table** that stores solutions to all configurations of the **sub-problem (patterns)**



- This PDB is used as the h-value of the f=g+h of nodes during the search

More than one PDBs

- If you have ***k*** of admissible heuristics (e.g. PDBs), their ***max*** is also admissible,
- ***h=max(h1,h2… hk)***

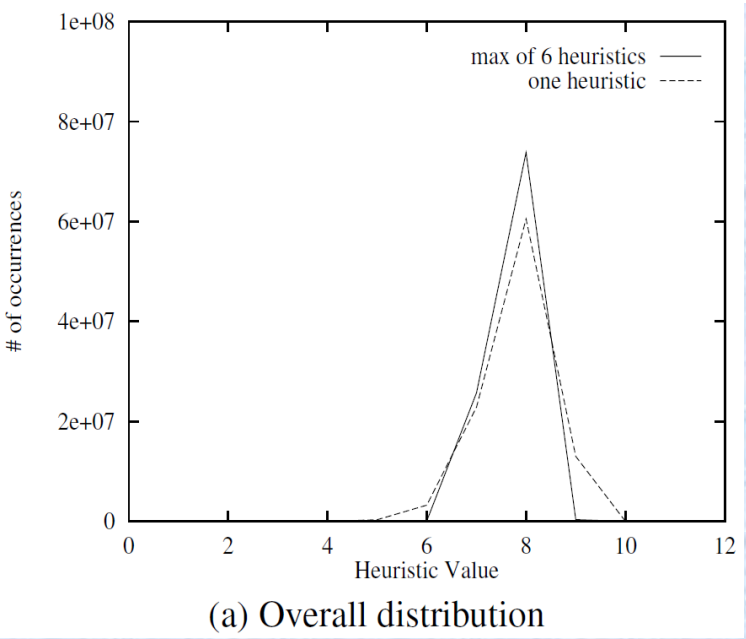
B	x	x	3
x	x	x	7
x	x	x	11
12	13	14	15

B	x	x	x
x	x	x	x
8	9	10	11
12	13	14	15

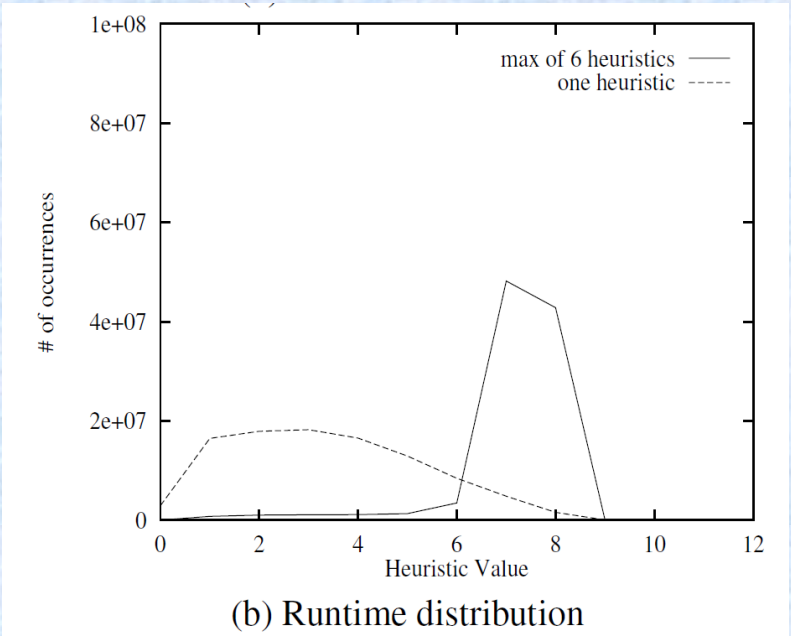
One large of a few small ones?

- Given 1 giga byte of memory, how do we best use it with PDBs?
- It is better to use many small PDBs and take their maximum instead of one large PDB? [Holte, Newton, Felner, Meshulam and Furcy, ICAPS-2004]
- It is better to be more precise on the low h-values than on the large h-values

$$\sum_{i=0}^{d} N(i) \cdot P(d-i)$$



(a) Overall distribution



(b) Runtime distribution

2) Rubik's Cube [Korf 1997]

Finding Optimal Solutions to Rubik's Cube Using Pattern Databases

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90095
korf@cs.ucla.edu

Abstract

We have found the first optimal solutions to random instances of Rubik's Cube. The median optimal solution length appears to be 18 moves. The algorithm used is iterative-deepening-A* (IDA*), with a lower-bound heuristic function based on large memory-based lookup tables, or "pattern databases" (Culberson and Schaeffer 1996). These tables store the exact number of moves required to solve various subgoals of the problem, in this case subsets of the individual movable cubies. We characterize the effectiveness of an admissible heuristic function by its expected value, and hypothesize that the overall performance of the program obeys a relation in which the product of the time and space used equals the size of the state space. Thus, the speed of the program increases linearly with the amount of memory available. As computer memories become larger and cheaper, we believe that this approach will become increasingly cost-effective.

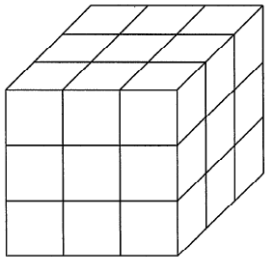


Figure 1: Rubik's Cube

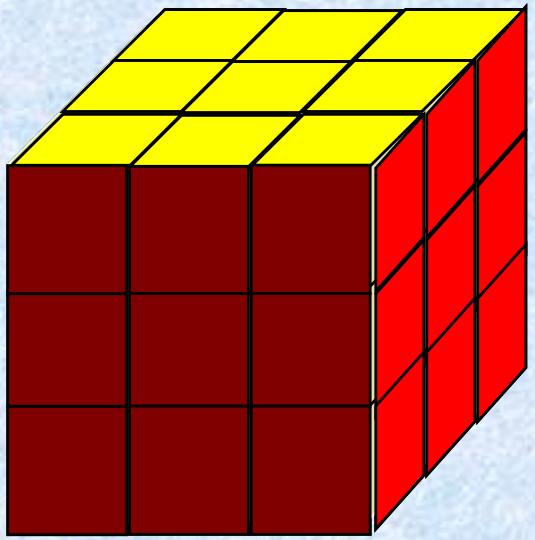
rotations of the entire cube as operators.

The idea of using large tables of precomputed optimal solution lengths for subparts of a combinatorial problem was first proposed by (Culberson and Schaeffer 1996). They studied these pattern databases in the context of the 15-puzzle, achieving significant performance improvements over simple heuristics like manhattan distance. This paper can be viewed as simply applying their idea to Rubik's Cube.

Large tables of heuristic values originated in the area of two-player games, where such tables are used to store the exact value (win, lose, or draw) of endgame positions. This technique has been used to great effect by (Schaeffer et al. 1992) in the game of checkers.

Problem: → 3x3x3 10^19 states

Subproblem: → 2x2x2 88 Million states.



The max of 6-6-8 PDB

3) All pairwise heuristics

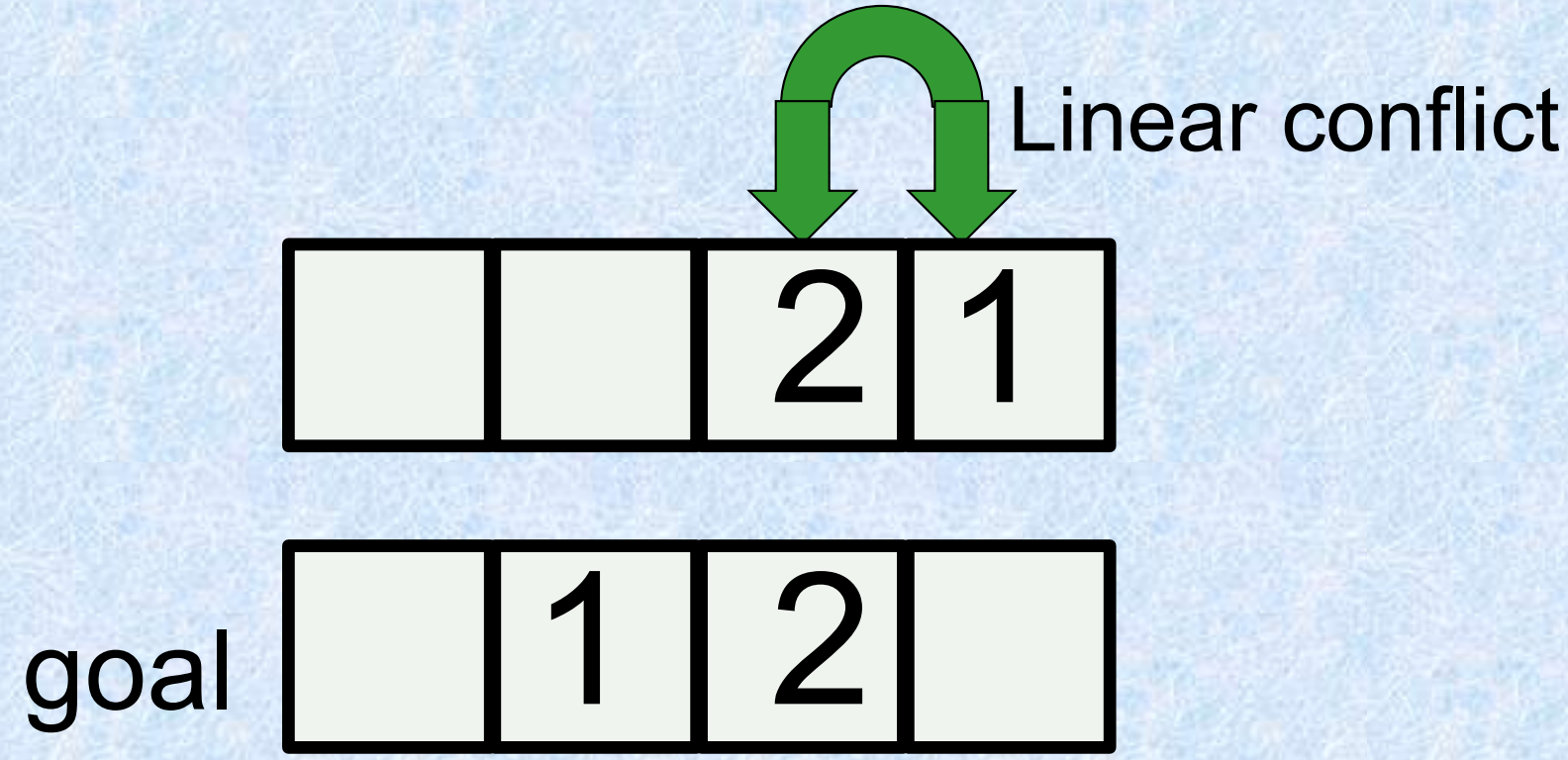
Finding Optimal Solutions to the Twenty-Four Puzzle

Richard E. Korf and **Larry A. Taylor**
Computer Science Department
University of California, Los Angeles
Los Angeles, Ca. 90024
korf@cs.ucla.edu, ltaylor@cs.ucla.edu

AAAI-96

To see this problem more clearly, represent a state as a graph with a node for each tile, and an edge between each pair of tiles, labelled with their pairwise distance. We need to select a set of edges from this graph, so that no two edges are connected to a common node, and the sum of the labels of the selected edges is maximized. This problem is called the maximum weighted matching problem, and can be solved in $O(n^3)$ time, where n is the number of nodes (Papadimitriou and Steiglitz

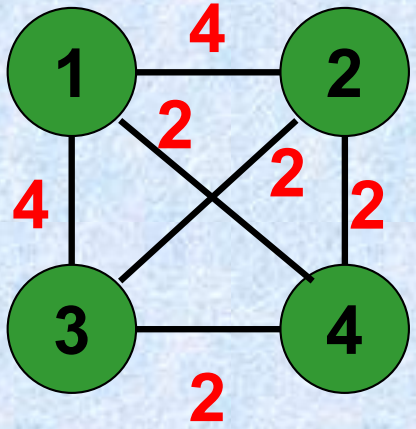
Pairwise distance



- Manhattan distance =2
- Pairwise distance = 4
- Pairwise distance over MD = 2

i) Maximal Matching on the pairwise graph

- Mutual cost graph (MCG)
  - Vertices: tiles
  - Edges: pairs of tiles.
  - Weights of edges: pairwise cost (above MD)



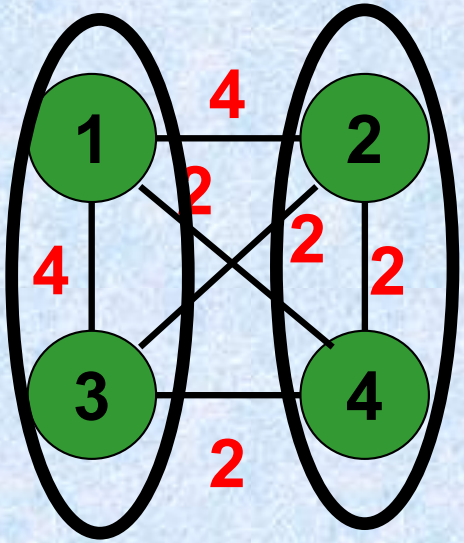
We can add 4+2=6 to MD

Algorithm

- Partition to disjoint pairs
- Perform Maximal Matching on the MCG

Maximal Matching on the pairwise graph

- Mutual cost graph (MCG)
  - Vertices: tiles
  - Edges: pairs of tiles.
  - Weights of edges: pairwise cost (above MD)



We can add 4+2=6 to MD

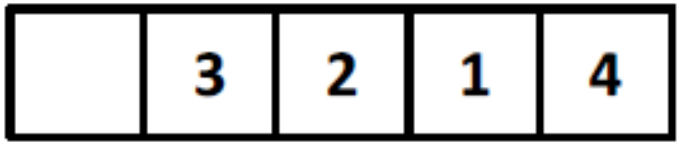
Algorithm

- Dynamic partition to disjoint pairs
- Pe

Very related to "systematic partitioning"

My first contribution: 1999

Weighted Vertex Cover



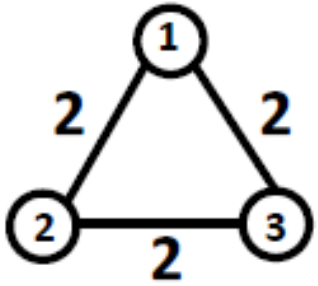


Figure 2: Mutual cost graph for 3-way linear conflict

- MM gives 2
- However, each edge (x,y) is a constraint in the form: **X+Y≥2**
- Yields a heuristic of 3

Weighted vertex cover

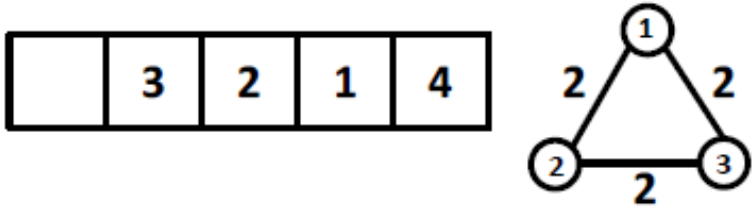


Figure 2: Mutual cost graph for 3-way linear conflict

The h^{PHO} heuristic (Pommerening, Roger, and Helmert 2013) uses linear programming to solve a similar set of constraints

We aim to work with integer values only.

- This yields a heuristic of 4

From pairs to triples and quadruples

- For triples {x,y,z}=4 we write

$$(X \geq 2 \ and \ Y \geq 2) \ or$$
$$(Y \geq 2 \ and \ Z \geq 2) \ or$$
$$(X \geq 2 \ and \ Z \geq 2) \ or$$
$$(X \geq 4) \ or \ (Y \geq 4) \ or \ (Z \geq 4)$$

- Now want the minimal assignments to tiles that satisfy these constraints.

- This is called ***weighted vertex-cover*** (ordinary vertex cover is a special case)

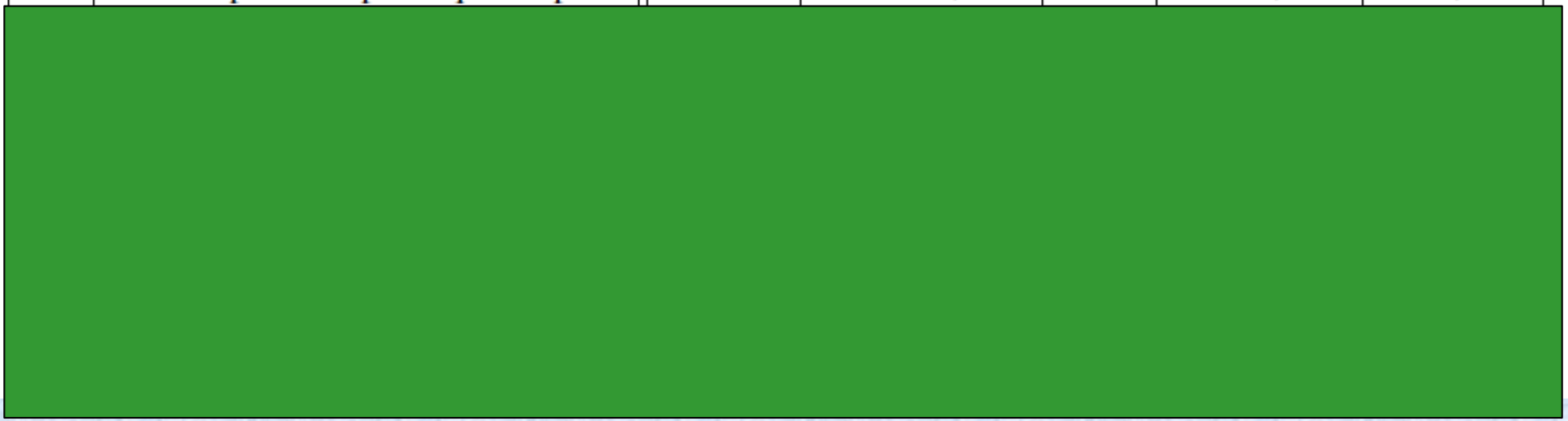
Weighted vertex cover

- WVC is NP-complete but…for up to triples
  - We can only store additions above MD
  - A number of disjoint connected components
  - Can be solved incrementally while moving from parent to child.

WVC for our purposes was solved rather fast!

Experimental Results:15 puzzle

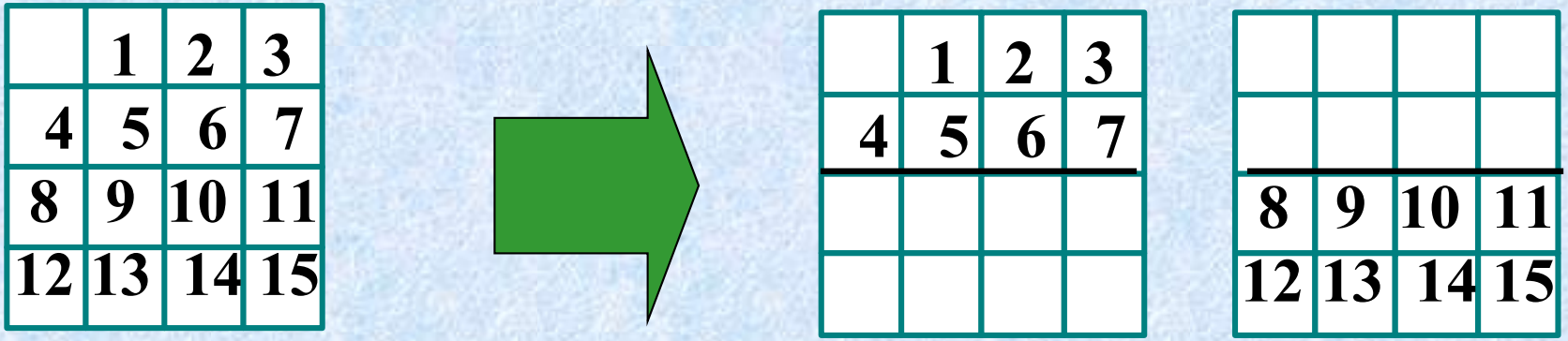
#	Heuristic Function	Value	Nodes	Sec.	Nodes/sec	Memory
1	Manhattan	36.940	401,189,630	53	7,509,527	0
3	MM: pairs	39.411	21,211,091	13	1,581,848	1,000
4	MM: pairs+triples	41.801	2,877,328	8	351,173	2,300
5	WVC: pairs	40.432	9,983,886	10	959,896	1,000
6	WVC: pairs+triples	42.792	707,476	5	139,376	2,300
7	WVC: pairs+triples+quadruples	43.990	110,394	9	11,901	78,800



1) Disjoint Additive PDBs

- The 7-tile and 8-tile subproblems are ***disjoint:***

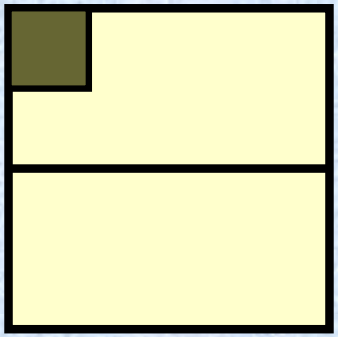
→ each operator belongs to one subproblem only



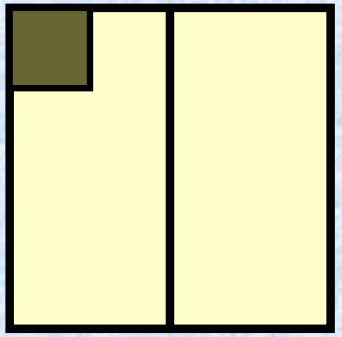
1. It is worth mentioning here that the way we treated the blank tile for the statically partitioned pattern databases is not trivial. For the backwards breadth-first search we treated this tile as a distinct tile that can move to any adjacent location. If that location was occupied by a real tile then that real tile was moved to the former blank location and we add one to the length of the path to that node. However, for the pattern database tables we have only considered the locations of the real tiles as indexes into the tables. We did not preserve the location of the blank and stored the minimum among all possible blank locations in order to save memory. In a sense, we have compressed the pattern databases according to the location of the blank (see (Felner, Meshulam, Holte, & Korf, 2004) about compressing pattern databases).

Enhancements

- Symmetry: Reflection about the main diagonal



7-8



7-8

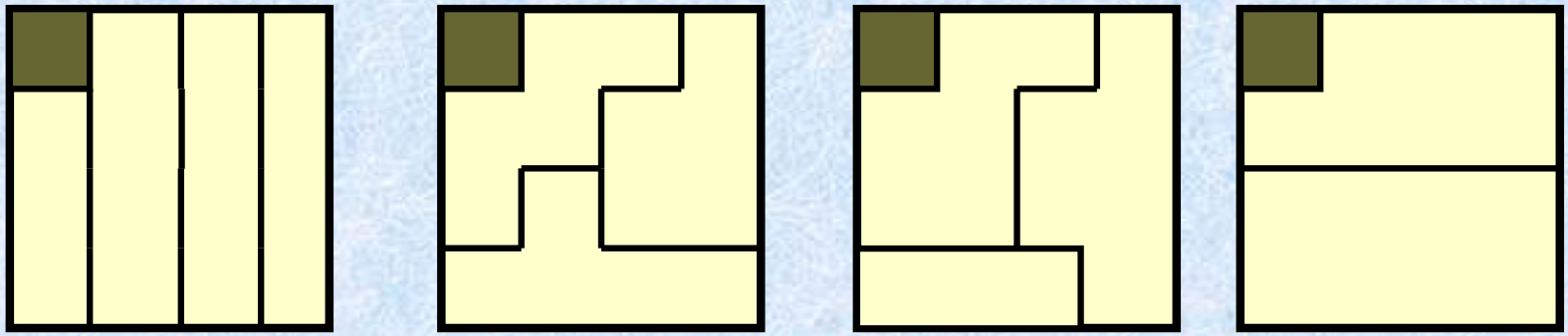
- One can store just the addition over MD in the PDB
- These additions come in units of 2.
- This was called Delta heuristics [sturtevant & Felner 2017]

Very related to "interesting patterns" by [Pommerening, Roger, and Helmert 2013]

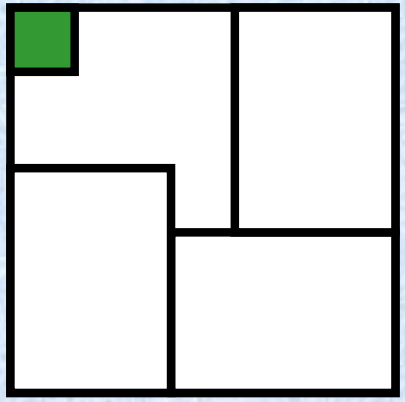
Disjoint PDBs

Also called: ***Statically partitioned PDBs***

15 puzzle



24 puzzle



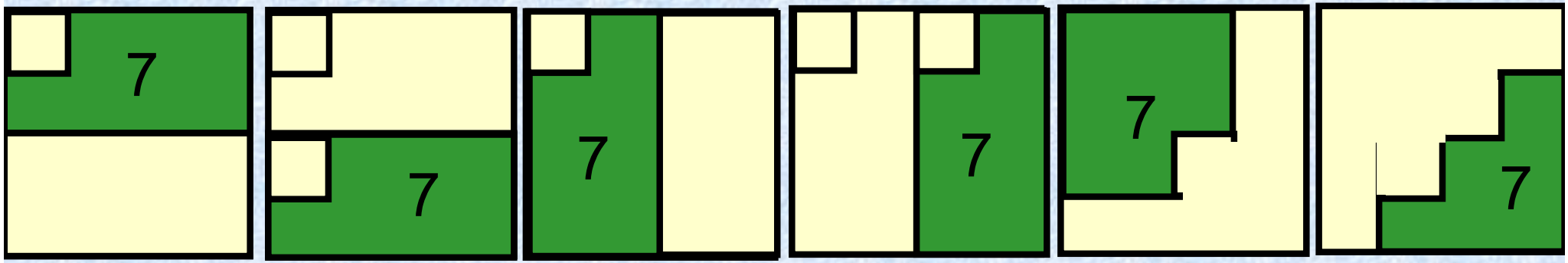
Experimental Results:15 puzzle

#	Heuristic Function	Value	Nodes	Sec.	Nodes/sec	Memory
1	Manhattan	36.940	401,189,630	53	7,509,527	0
3	MM: pairs	39.411	21,211,091	13	1,581,848	1,000
4	MM: pairs+triples	41.801	2,877,328	8	351,173	2,300
5	WVC: pairs	40.432	9,983,886	10	959,896	1,000
6	WVC: pairs+triples	42.792	707,476	5	139,376	2,300
7	WVC: pairs+triples+quadruples	43.990	110,394	9	11,901	78,800
8	PA: 5-5-5	41.560	3,090,405	.540	5,722,922	3,145
9	PA: 6-6-3	42.924	617,555	.163	3,788,680	33,554
10	PA: 7-7-1	44.586	116,985	.047	2,489,042	268,437
15	PA: 7-8	45.630	36,710	.028	1,377,630	576,575

3) Vertex-cover tables

- We store a small number of rather large overlapping PDBS

For example, we store 6 different 7-tile PDB



- Here we cannot solve WVC on the fly

3) Vertex-cover tables

- Instead we do it in a preprocessing phase
- Assume we have 6 different PDBs each with q different values
  - T-tile PDB {0,2,4,6,8,10,12}
- We have 6^q combinations
- We build a table with 6^q rows
- Each stores WVC for the corresponding entry.

P1	P2	P3	P4	P5	P6	Val
2	2	4	6	2	8	**14**
2	4	0	2	4	6	**12**
4	0	8	4	6	0	**10**

3) Vertex-cover tables

- **In the preprocessing phase:**
  1. Fill the VCT table with the correct values
- **For each node in the main search**
  1. Retrieve the PDB values for all stored groups
  2. Use these values as indexes and lookup the relevant entry in the VCT to retrieve the value of WVC.
  3. Add this value to MD.

3) Vertex-cover tables: generality

1) Number of entries is smaller than the number of expected nodes.

2) Table can be built lazily

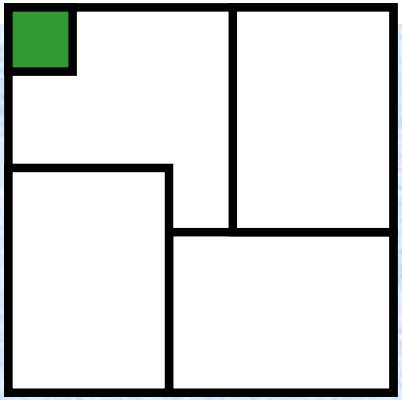
3) Other problems too.

Experimental Results:15 puzzle

#	Heuristic Function	Value	Nodes	Sec.	Nodes/sec	Memory
1	Manhattan	36.940	401,189,630	53	7,509,527	0
3	MM: pairs	39.411	21,211,091	13	1,581,848	1,000
4	MM: pairs+triples	41.801	2,877,328	8	351,173	2,300
5	WVC: pairs	40.432	9,983,886	10	959,896	1,000
6	WVC: pairs+triples	42.792	707,476	5	139,376	2,300
7	WVC: pairs+triples+quadruples	43.990	110,394	9	11,901	78,800
8	PA: 5-5-5	41.560	3,090,405	.540	5,722,922	3,145
9	PA: 6-6-3	42.924	617,555	.163	3,788,680	33,554
10	PA: 7-7-1	44.586	116,985	.047	2,489,042	268,437
11	VCT: 7 6-tile PDBs	43.211	397,107	.134	2,963,485	34,377
12	VCT: 10 6-tile PDBs	43.485	242,186	.115	2,105,965	332,806
13	VCT: 5 7-tile PDBs	44.563	97,730	.044	2,221,136	402,669
14	VCT: 6 7-tile PDBs	44.531	76,634	.037	2,071,189	419,548
15	PA: 7-8	45.630	36,710	.028	1,377,630	576,575

Experimental Results:24 puzzle

		Nodes			Time (seconds)		
#	Sol	WVC	PA	VCT	WVC	PA	VCT
1	95	**306,958,148**	2,031,102,635	1,377,159,819	1,757	1,446	**1,063**
2	96	**65,125,210,009**	211,884,984,525	158,889,554,781	692,829	147,493	**123,018**
3	97	52,906,797,645	21,148,144,928	**14,448,309,001**	524,603	14,972	**11,294**
4	98	**8,465,759,895**	10,991,471,966	9,262,519,107	72,911	7,809	**7,016**
5	100	**715,535,336**	2,899,007,625	2,480,350,516	3,922	2,024	**1,894**
6	101	**10,415,838,041**	103,460,814,368	86,134,496,298	151,083	74,100	**65,252**
7	104	**46,196,984,340**	106,321,592,792	85,774,231,083	717,454	76,522	**66,491**
8	108	**15,377,764,962**	116,202,273,788	83,209,058,152	82,180	81,643	**64,424**
9	113	**135,129,533,132**	1,818,055,616,606	1,476,665,302,180	747,443	3,831,042	**3,222,608**
10	114	**726,455,970,727**	1,519,052,821,943	1,331,681,205,551	4,214,591	**3,320,098**	3,390,445
Avg	102.6	**106,109,635,224**	391,204,783,118	309,119,152,126	720,877	752,698	**695,351**



6-6-6-6

General Additive PDBS

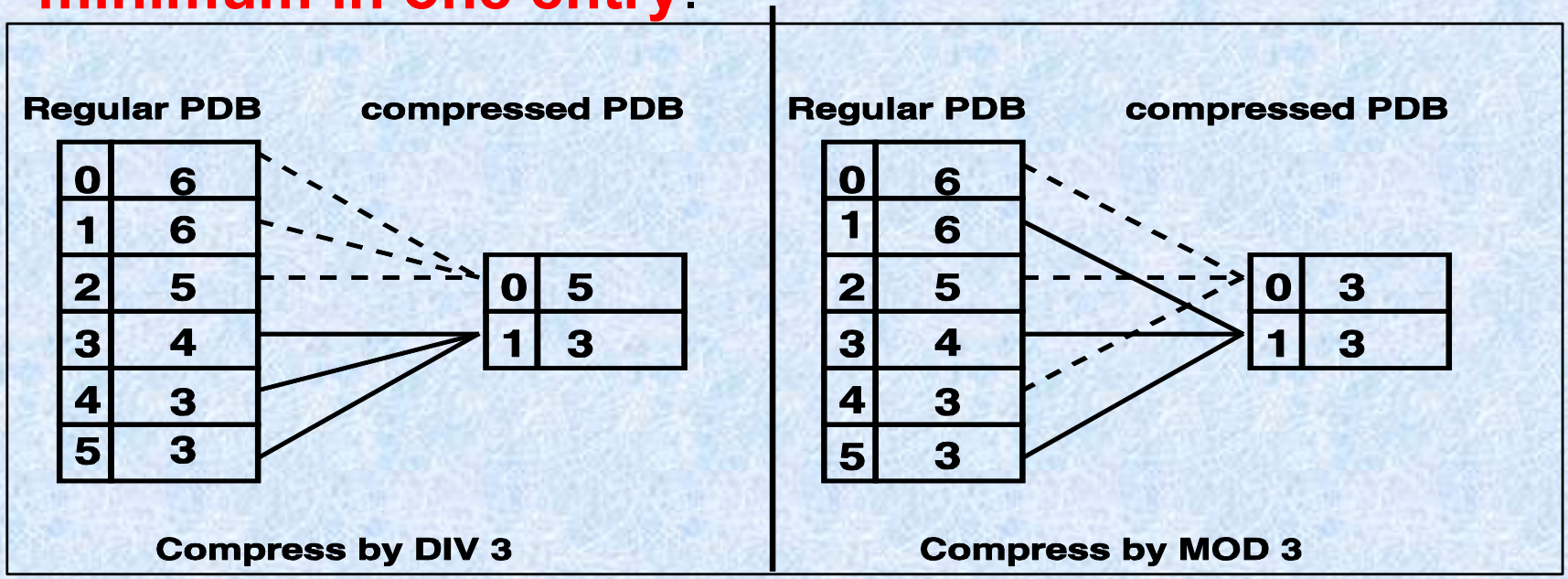
- What about problems where each operator moves more than one object?
- This was addressed by [Yang, Holte, Culberson, Zahavi, Felner JAIR 2008]
- **Cost splitting** – split the costs among the different (non-disjoint) sub-problems.

- **Location based costs** – we only charge the pattern that moved into a *special* location.
- Additivity was also used in **planning.**
  - **[pommerening et al.]**

Compressing pattern database

[Felner et al AAAI-04, JAIR-2007]

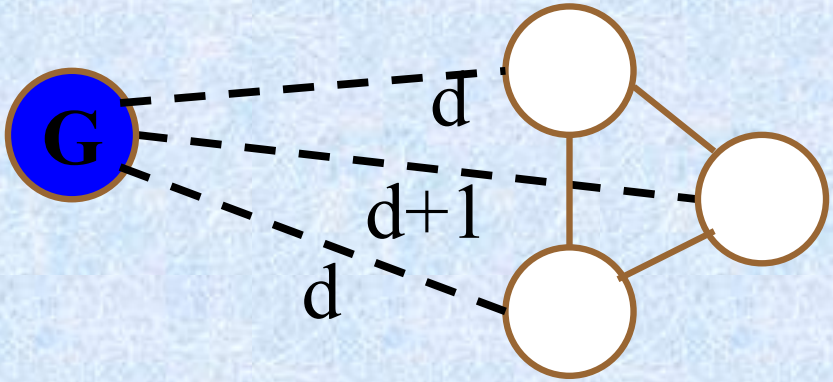
- **Entry Compression:**
  - **Nearby entries in PDBs are highly correlated !!**
- **We can compress nearby entries by storing their minimum in one entry.**



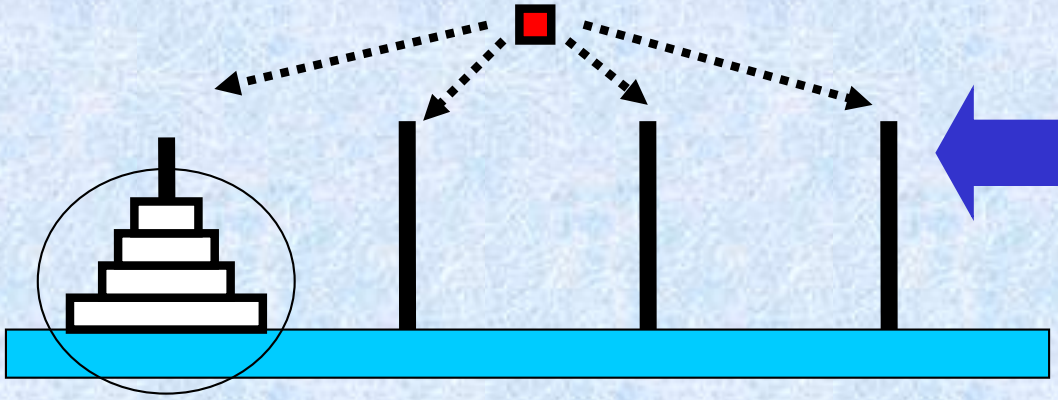
- We show that → **most of the knowledge is preserved**
- Consequences: Memory is saved, larger patterns can be used → speedup in search is obtained.

Cliques in the pattern space

- The values in a PDB for a clique are ***d*** or ***d+1***
- In permutation puzzles cliques exist when only one object moves to another location.



- Usually they have nearby entries in the PDB
- A[4][4][4][4]**[4]**



A clique in TOH4

Compressing cliques

- Assume a clique of size ***K*** with values ***d*** or ***d+1***
- Store only one entry (instead of ***K***) for the clique with the minimum ***d***. Lose at most ***1***.
  - A[4][4][4][4]**[4]** ⇨ A[4][4][4][4]**[1]**
  - Instead of 4^p we need only 4^(p-1) entries.

- This can be generalized to a set of nodes with diameter D. (for cliques D=1)
  - A[4][4][4]**[4][4]** A[4][4][4]**[1][1]**

⇨

- In general: compressing by k disks reduces memory requirements from *4^p* to *4^(p-k)*

TOH4 results: 16 disks (14+2)

PDB	H(s)	Avg H	D	Nodes	Time	Mem MB
14/0 + 2	116	87.03	0	36,479,151	14.34	256
14/1 + 2	115	86.48	1	37,964,227	14.69	64
14/2 + 2	113	85.67	3	40,055,436	15.41	16
14/3 + 2	111	84.44	5	44,996,743	16.94	4
14/4 + 2	107	82.73	9	45,808,328	17.36	1
14/5 + 2	103	80.84	13	61,132,726	23.78	0.256

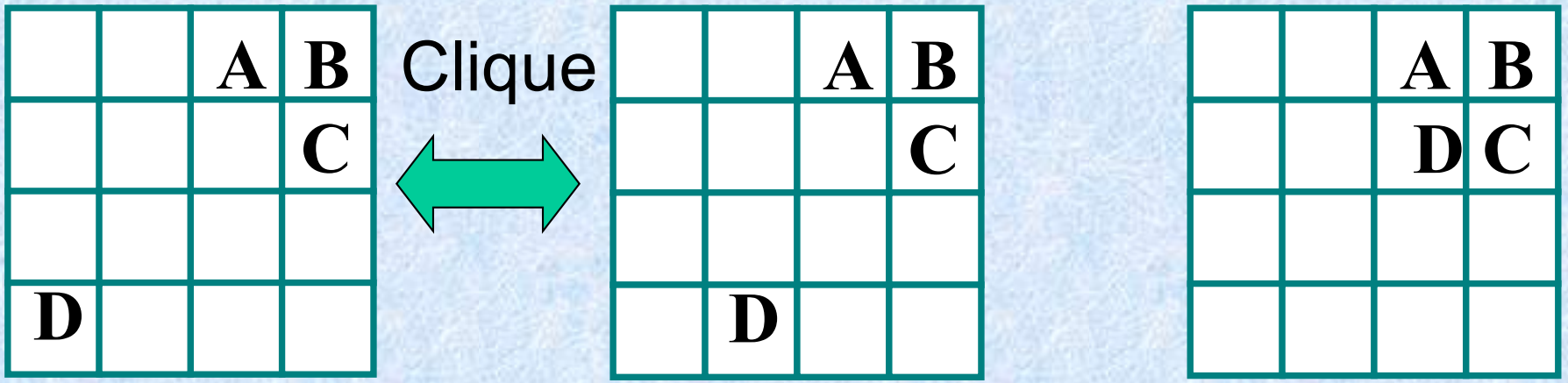
- Memory was reduced by a factor of 1000!!! at a cost of only a factor of 2 in the search effort.

TOH4: larger versions

size	PDB	Type	Avg H	Nodes	Time	Mem
17	14/0 + 3	static	81.5	>393,887,923	>421	256
17	14/0 + 3	dynamic	87.0	238,561,590	2,501	256
17	15/1 + 2	static	103.7	155,737,832	83	256
17	16/2 + 1	static	123.8	17,293,603	7	256
18	16/2 + 2	static	123.8	380,117,836	463	256

- For the 17 disks problem a speed up of 3 orders of magnitude is obtained!!!
- The 18 disks problem can be solved in 5 minutes!!

Tile Puzzles



- **Storing PDBs for the tile puzzle**
- (Simple mapping) A multi dimensional array →

  A[16][16][16][16][16] size=1.04Mb
- (Packed mapping) One dimensional array → A[16*15*14*13*12] size = 0.52Mb.
- Time versus memory tradeoff !!

15 puzzle results

- A clique in the tile puzzle is of size 2.
- We compressed the last index by two →

A[16][16][16][16][8]

PDB	Type	compress	Nodes	Time	Mem	Avg H
1 7-8	packed	No	136,288	0.081	576,575	44.75
1+ 7-8	packed	No	36,710	**0.034**	576,575	45.63
1 7-7-1	packed	No	464,977	0.232	57,657	43.64
1 7-7-1	simple	No	464,977	0.058	536,870	43.64
1 7-7-1	simple	Yes	565,881	0.069	268,435	43.02
2 7-7-1	simple	Yes	147,336	0.021	536,870	43.98
2+ 7-7-1	simple	Yes	66,692	**0.016**	536,870	44.92

Value Range Compression

[Sturtevant and Felner, AAAI-2017]

- **Value compression**:
  - When you have a large range of values partition them into disjoint regions.
  - Store a value for the entire region

- For example for numbers 0…99
- Partition to [0..9] [10..19] …. [90..99]
- Store, 0, 10, 20 … 90 for these regions.

- You save many bits. Loss of information is small

- Combination of entry and value compression proved useful

VC2: Brute force -- 16 and 17 become 15

VC2h: Intelligent value compressing

EC2 (entry compression): two nearby entries

	1,760MB	880MB			440MB	
D	Total	$\underline{VC2}$	$VC2_{\bar{h}}$	EC2	$VC4_{\bar{h}}$	EC4
0	1	1	12	2	10,188,753	4
1	11	11		22		40
2	94	94	94	186		340
3	731	731	731	1,430		2,596
4	5,353	5,353	5,353	10,340		18,736
5	37,275	37,275	37,275	70,894		127,756
6	245,468	245,468	245,468	457,304		813,700
7	1,508,099	1,508,099	1,508,099	2,722,458		4,724,408
8	8,391,721	8,391,721	8,391,721	14,408,820		23,870,392
9	40,012,497	40,012,497	40,012,497	63,502,746	190,013,262	97,318,252
10	150,000,765	150,000,765	150,000,765	212,692,340		290,434,356
11	393,482,172	393,482,172	393,482,172	478,114,034	393,482,172	553,276,900
12	612,084,904	612,084,904	612,084,904	601,419,722	1,170,638,373	549,750,508
13	440,655,534	440,655,534	440,655,534	328,304,534		217,340,348
14	110,437,757	110,437,757	110,437,757	59,883,892		26,009,144
15	7,389,524	7,460,178	7,389,524	2,721,910		634,464
16	**70,633**		70,654	11,924		616
17	**21**			2		
Avg.	11.90	11.90	**11.90**	11.59	**11.38**	11.27

Combining both VC and EC

Memory		EC	VC	VC-bits	Nodes	Time
1		1	1	8	3.88M	15.29
0.5	(A)	1	2	4	**3.88M**	**15.32**
0.375		1	2.66	3	4.03M	15.44
0.25	(B)	1	4	2	10.39M	33.63
0.5	(A)	2	1	8	7.11M	27.70
0.25	(B)	2	2	4	**7.11M**	**27.88**
0.1875		2	2.66	3	7.37M	28.44
0.125	(C)	2	4	2	30.43M	80.04
0.25	(B)	4	1	8	13.75M	51.06
0.125	(C)	4	2	4	**13.74M**	**50.97**
0.094		4	2.66	3	14.31M	51.52
0.0625		4	4	2	30.48M	77.68

Table 2: Results for (18-4)-TopSpin

Using bloom filters

[Sturtevant and Felner, SoCS 2014]

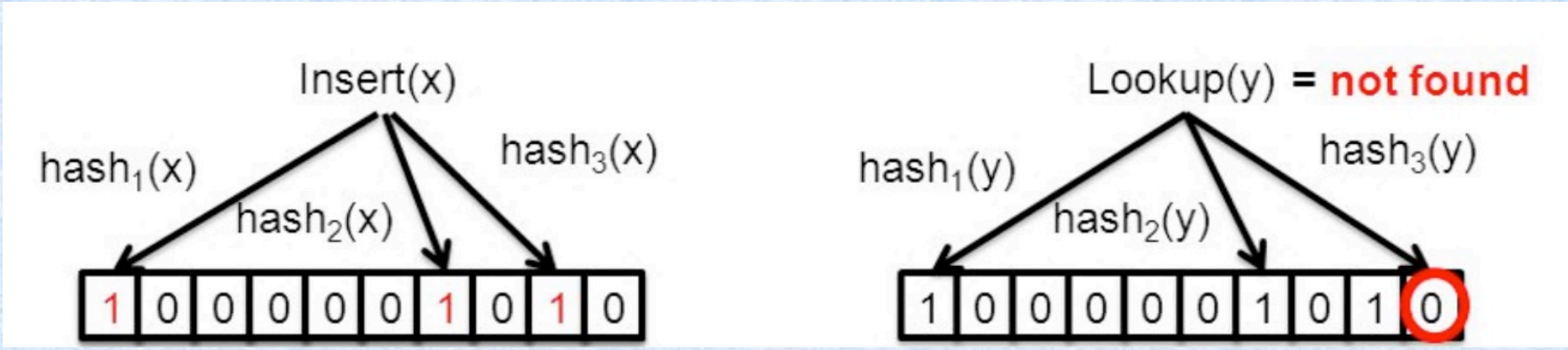
Partial Pattern Databases (PPDBs)

[Anderson, Holte, and Schaeffer 2007]:

- only store the first D levels
- Any item >D will be treated as D+1

Bloom filters [1971]: performs membership test.

- Insert: a number of hash functions, each sets a bit.
- Check: lookup all bits of an item
- Small chance for false positive



PDBs with bloom filters

Level by level bloom filters

- Build a bloom filter for each of the D levels
- Lookup:

```
Iterate on levels i from 1 to D{
    If bloom(i)=True
        set h=i
}
Set h=D+1
```

In case of a false positive h is still admissible

We can compress ranges (levels) too (like VRC)

Compressed PDB within a bloom filter

- Build one bloom filter for all values
- For each item store its value
- In case of a collision, store the minimum.

Lookup: look on all hash values and take the maximum

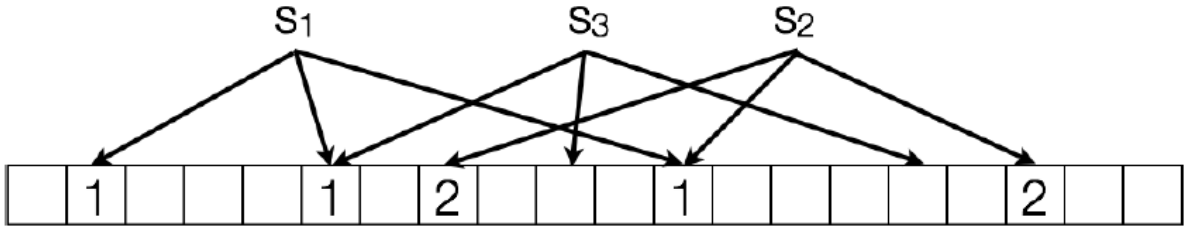
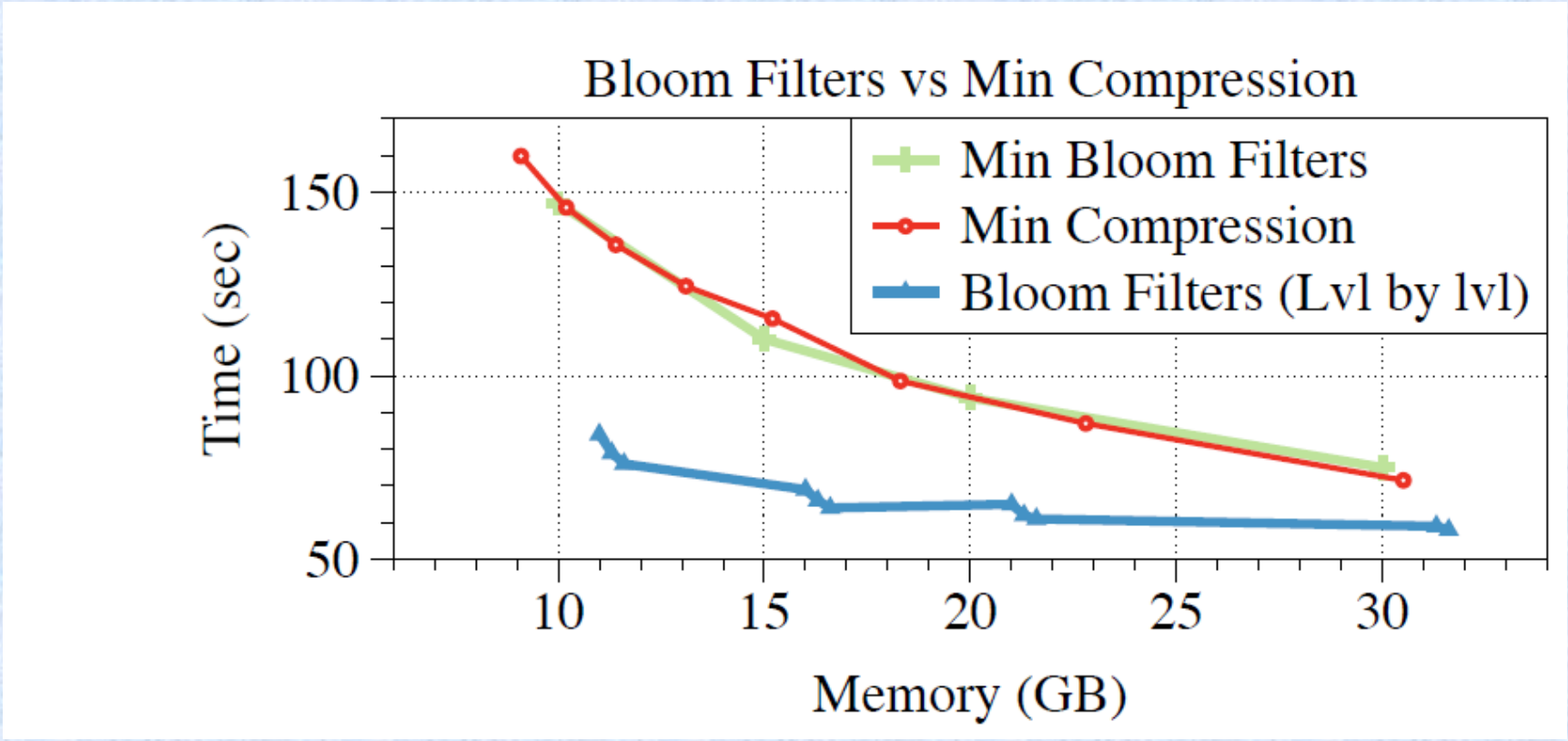


Figure 3: A min Bloom filter

Experiments with bloom filters



Learning PDBs

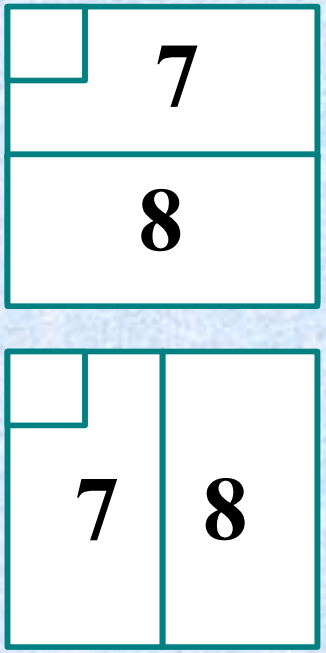
[Samadi,Felner and Holte, ECAI-2008)

- Preprocessing:
  - Build a PDB
  - Use ANN to learn it.
  - Build a side hash function that stored all the values that were overestimated.
- During Search:
  - Consult the hash table.
  - Consult the ANN
- Enhancements: use Decision Tree to classify Positive Delta

- **Dual lookups in pattern databases [Felner et al, IJCAI-05]**

Symmetries in PDBs

- **Symmetric lookups** were already performed by the first PDB paper of [Culberson & Schaeffer 96]
- **examples**
  - **Tile puzzles**: reflect the tiles about the main diagonal.
  - **Rubik's cube**: rotate the cube
- We can take the maximum among the different lookups
- These are all **geometrical** symmetries
- We suggest a new type of symmetry!!



Regular and dual representation

- **Regular** representation of a problem:
- Variables – objects (tiles, cubies etc,)
- Values – locations
- **Dual** representation:
- Variables – locations
- Values – objects

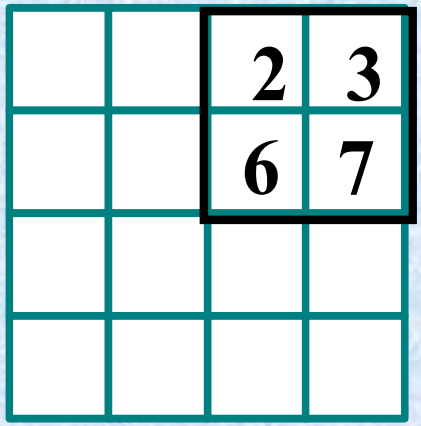
Regular vs. Dual lookups in PDBs

- **Regular question:**

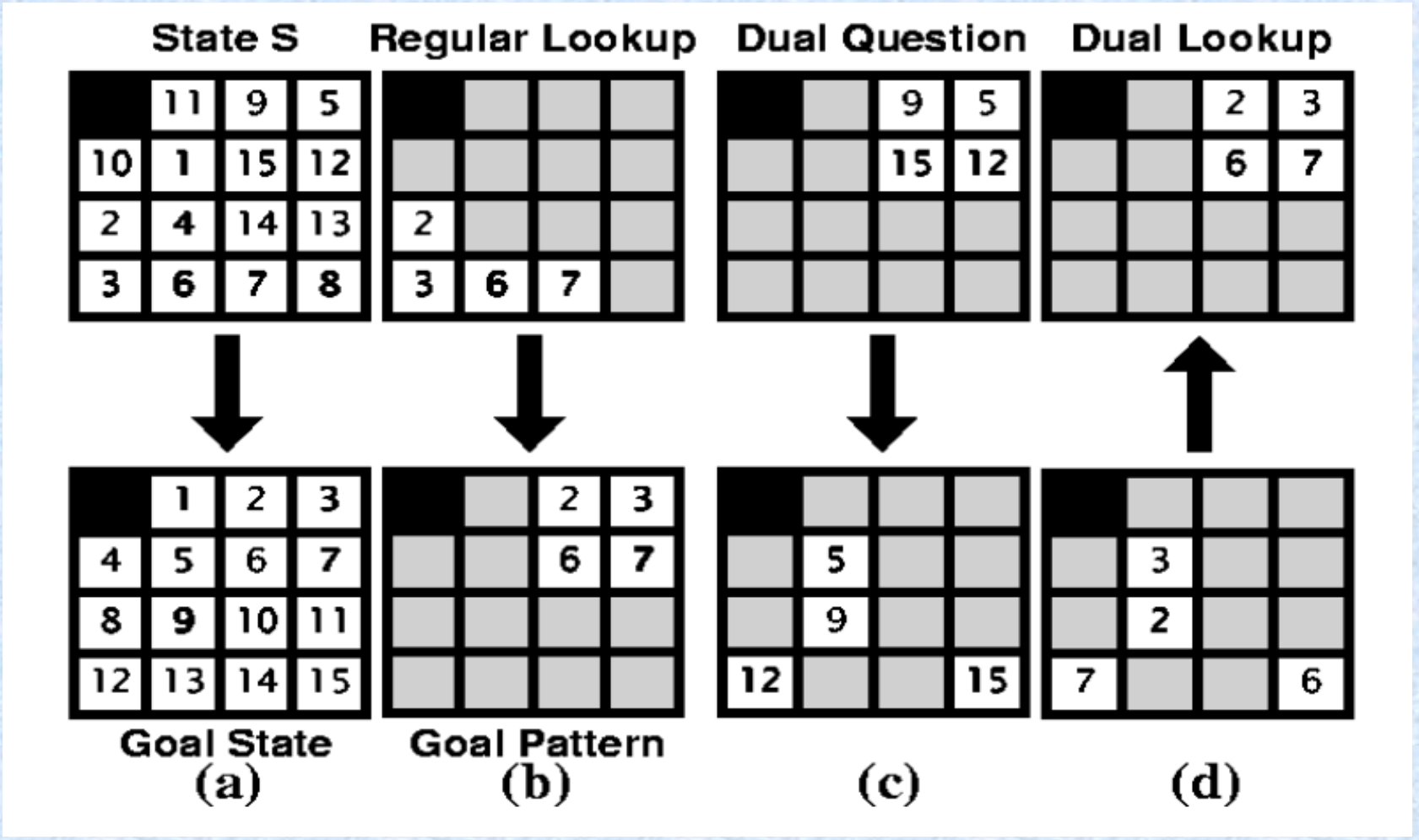
  Where are tiles {2,3,6,7} and how many moves are needed to gather them to their goal locations?

- **Dual question:**

  Who are the tiles in locations {2,3,6,7} and how many moves are needed to distribute them to their goal locations?



Regular and dual lookups



- Regular lookup: **PDB[8,12,13,14]**
- Dual lookup: **PDB[9,5,12,15]**

Regular and dual in TopSpin

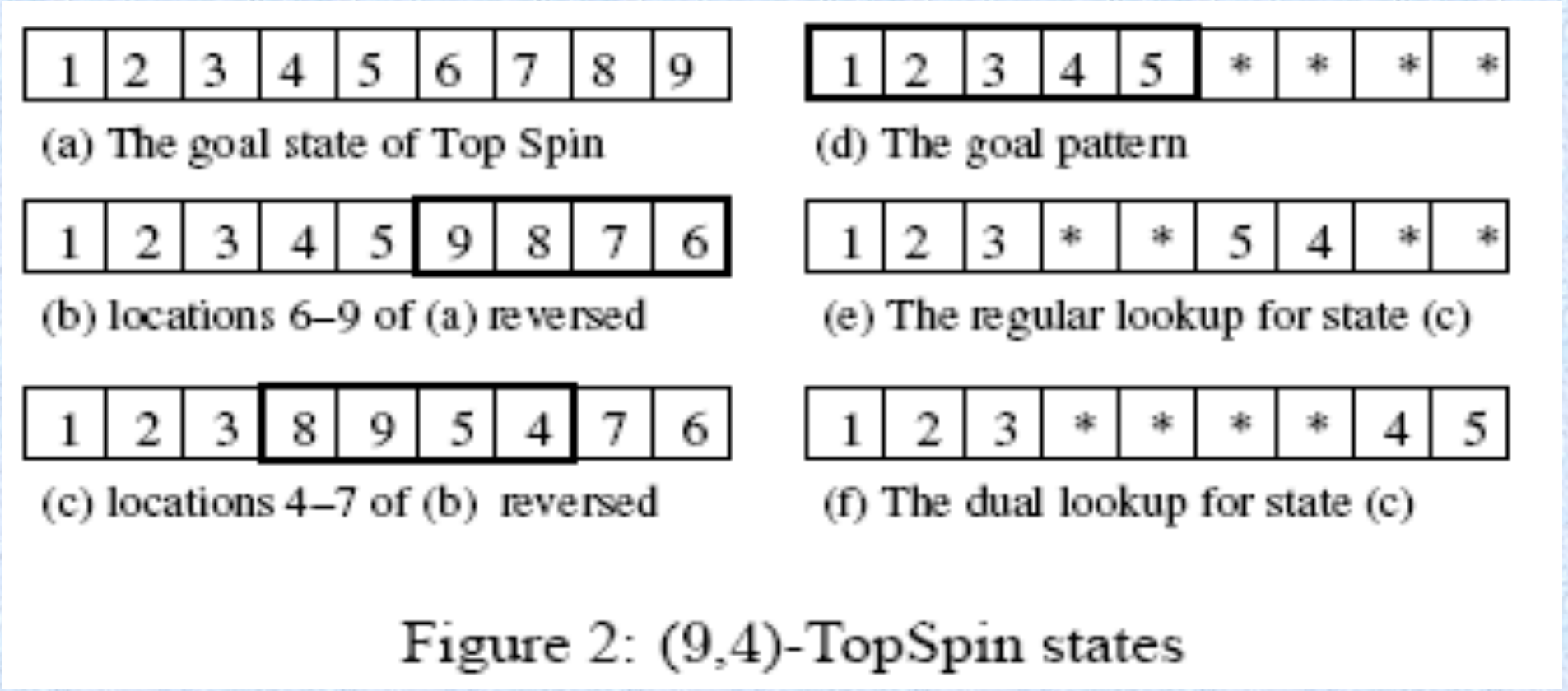


Figure 2: (9,4)-TopSpin states

- Regular lookup for C : PDB[1,2,3,7,6]
- Dual lookup for C: PDB[1,2,3,8,9]

Dual lookups

- Dual lookups are possible when there is a symmetry between locations and objects:
  - **Each object is in only one location and each location occupies only one object.**
- **Good examples**: TopSpin, Rubik's cube
- **Bad example:** Towers of Hanoi
- **Problematic example:** Tile Puzzles

1) Inconsistent heuristics

[Zahavi, et al. AAAI-2007,
Zhang et al. IJCAI 2009,
Felner et al. AIJ-2011]

Joint work with Uzi Zahavi,
Zhifu Zhang,
Nathan Sturtevant,
Robert Holte and
Jonathan Schaeffer.

Inconsistent heuristics

- Inconsistency sounds *negative*
- ***"It is hard to concoct heuristics that are admissible but are inconsistent"***
  [AI book, Russel and Norvig 2005]
- ***"Almost all admissible heuristics are consistent"*** [Korf, AAAI-2000]

Consistent heuristics

- A heuristic is ***consistent*** if for every two nodes n and m

$$h(n) \leq c(n,m) + h(m)$$

$$h(m) \leq c(m,n) + h(n)$$

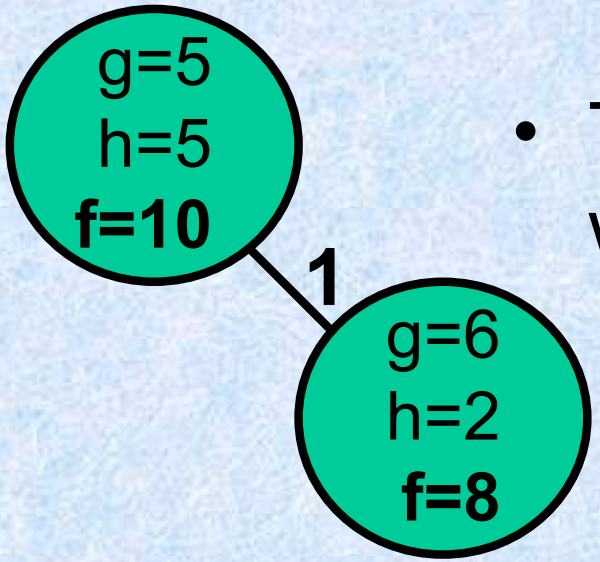
For undirected graphs: $|h(n)-h(m)| \leq c(n,m)$

- Intuition: h cannot change by more than the change of g

Inconsistent heuristics

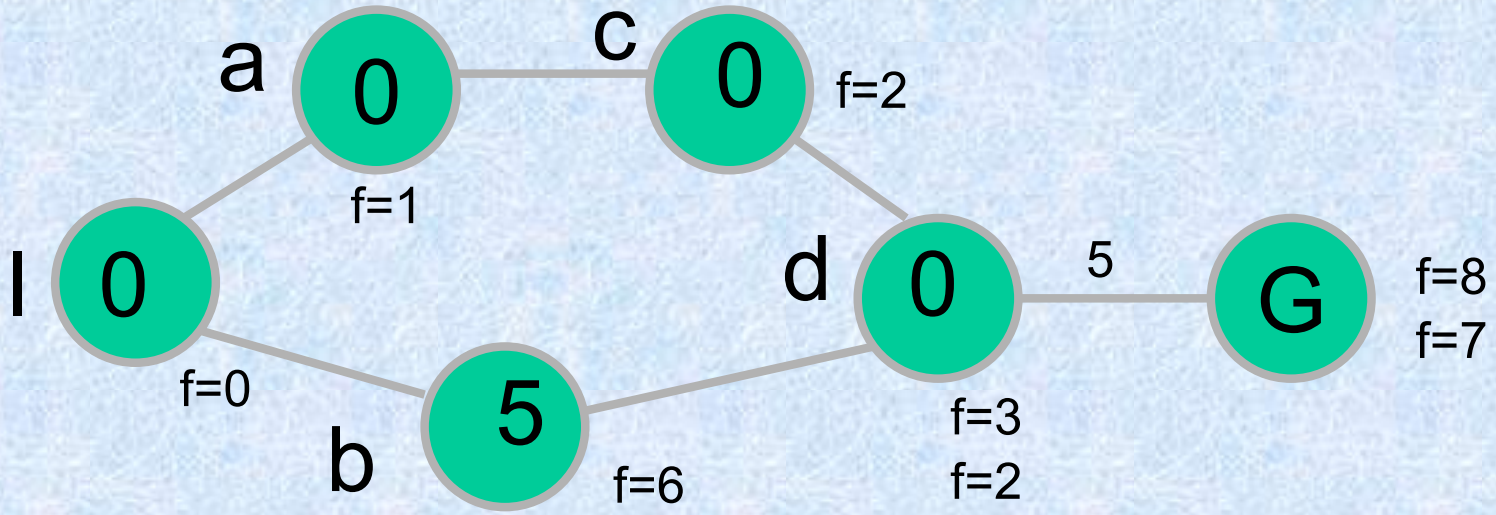
- A heuristic is *inconsistent* if for some two nodes n and m

|h(n)-h(m)| > dist(n,m)



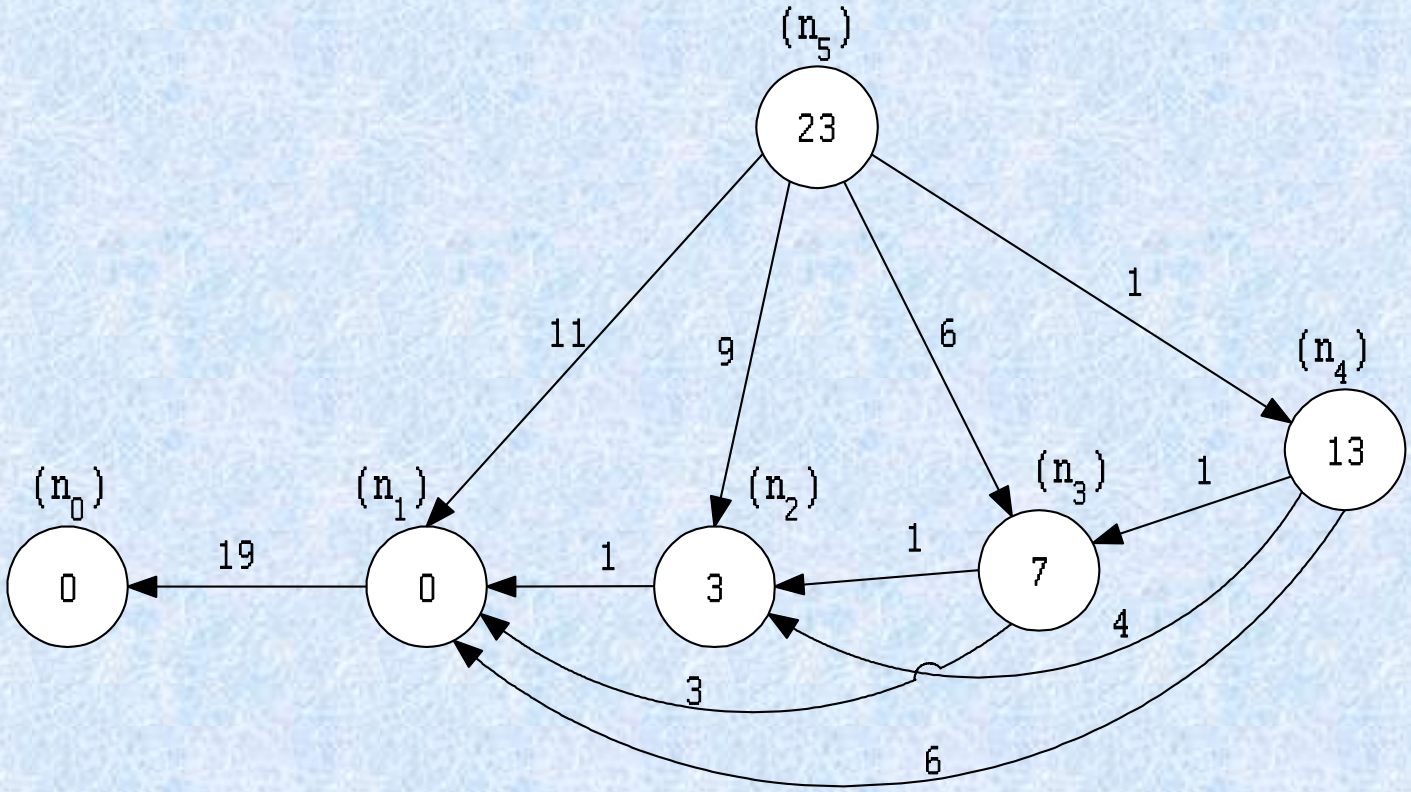
- The child is *inconsistent* with its parent

Reopening of nodes with A*



- Node ***d*** is expanded twice with A*!!

- In the context of A* inconsistency was considered a bad attribute



Extreme case: exponential number of node expansions. n5(23), n1(11), n2(12), n1(10), n3(13), n1(9), n2(10),n1(8), n4(14), n1(7), n2(8), n1(6), n3(9), n1(5), n2(6), n1(4).

Inconsistency in practical graphs

- The exponential worst-case behavior only occurs if the edge weights and the heuristic values grow exponentially.

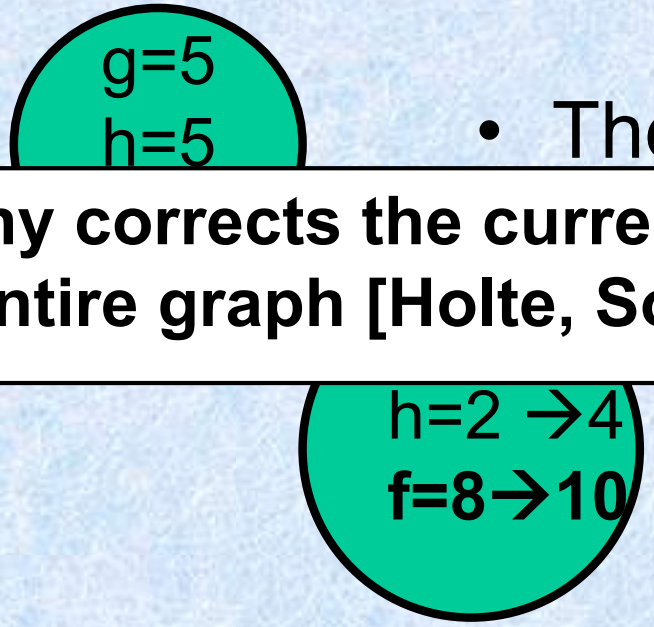
- But, if all edges weights are ≤ C then A* will expand O(N^2) states. [See the papers]

Inconsistency and IDA*

- In the context of A* inconsistency was considered a bad attribute
- Node re-opening is not a problem with IDA* because each path to a node is examined anyway!!
- No overhead for inconsistent heuristics

Pathmax

- The pathmax (PMX) method corrects inconsistent heuristics. [Mero 84,Marteli 77]

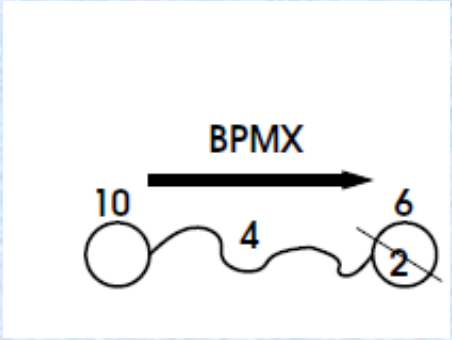


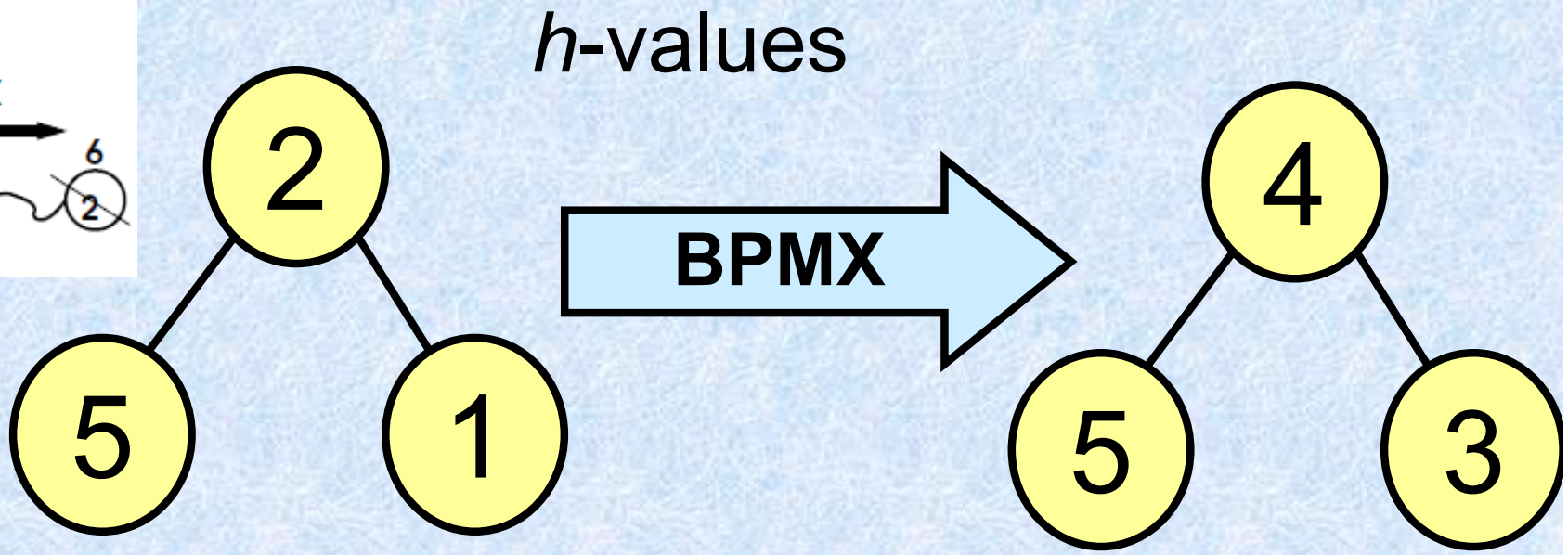
- The child inherits the f-value

Pathmax ony corrects the current path to be consistent, not the entire graph [Holte, SoCS 2010]

Bidirectional pathmax (BPMX)

[Felner, Zahavi, Schaeffer, Holte IJCAI-2005]



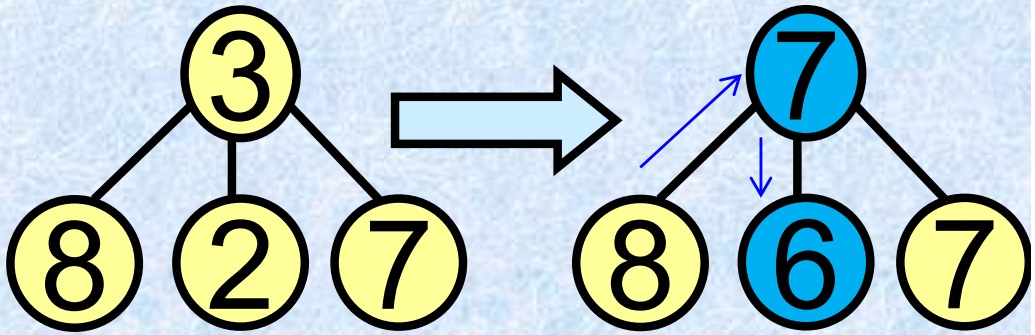


- Bidirectional pathmax: h-values are propagated in both directions decreasing by 1 in each edge.
  - If the IDA* threshold is 2 then with BPMX the right child will not even be generated!!

BPMX within A*

BMPX(1)

- We have a node p and its children n1, n2 … nk at hand.
- Let h' be the largest heuristic among the children
- For each child n we set
  - h(n)=max(h(n), h'-2)
- For the parent p we set
  - h(p)=max(h(n),h'-1)



→ Going deeper did not prove to be cost effective.

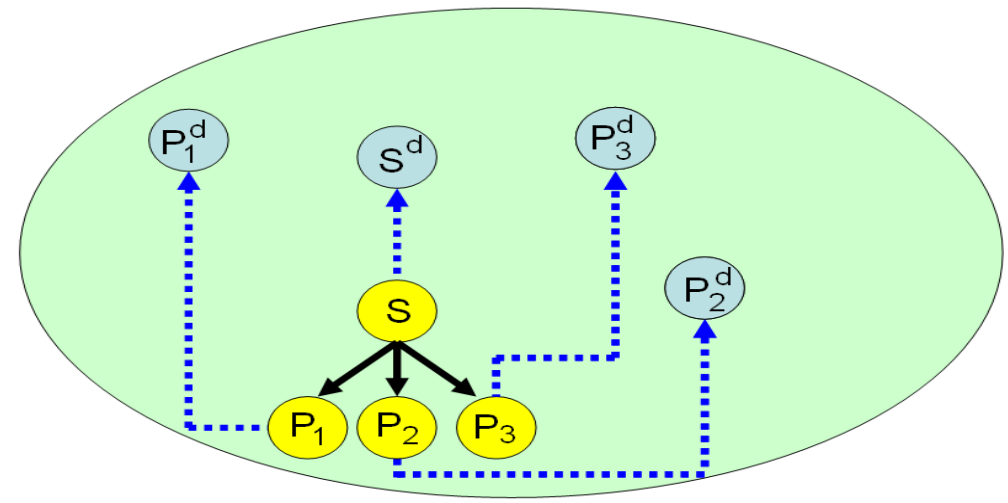
→ BPMX(∞) can be great or catastrophic. [See paper]

Achieving inconsistent heuristics

1) Random selection of heuristics (out of K)

2) Dual evaluations are inconsistent

[Zahavi et al. AAAI-2006]

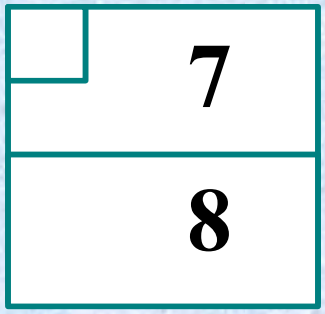


3) Compressed pattern databases

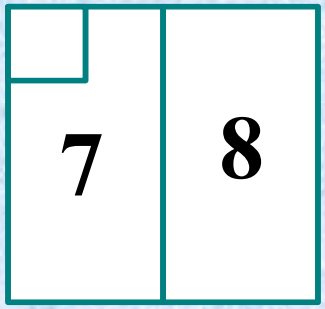
In general – any partial heuristic is inconsistent.

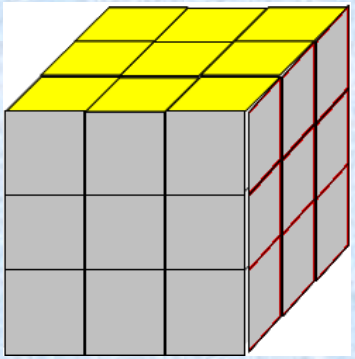
More than one heuristic

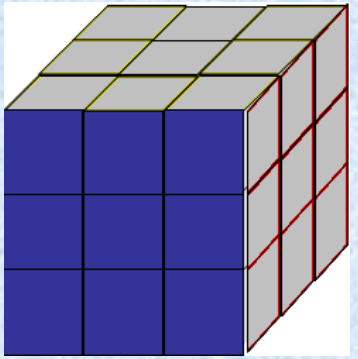
- **A munber of different PDBs**



- **Symmetric lookups**
  - **Tile puzzles**: reflect the tiles about the main diagonal.
  - **Rubik's cube**: rotate the cube







1) Randomizing a heuristic

Taking the **maximum** of K heuristics is

- Admissible
- Consistent
- Better than each of them
- Drawbacks: Overhead of K heuristics lookups
  diminishing return.

Alternatively, we can **randomize** which heuristic out of K to consult.

- Admissible
- Inconsistent
- Benefits: Only one look up. BPMX can be activated.

Regular and dual in TopSpin

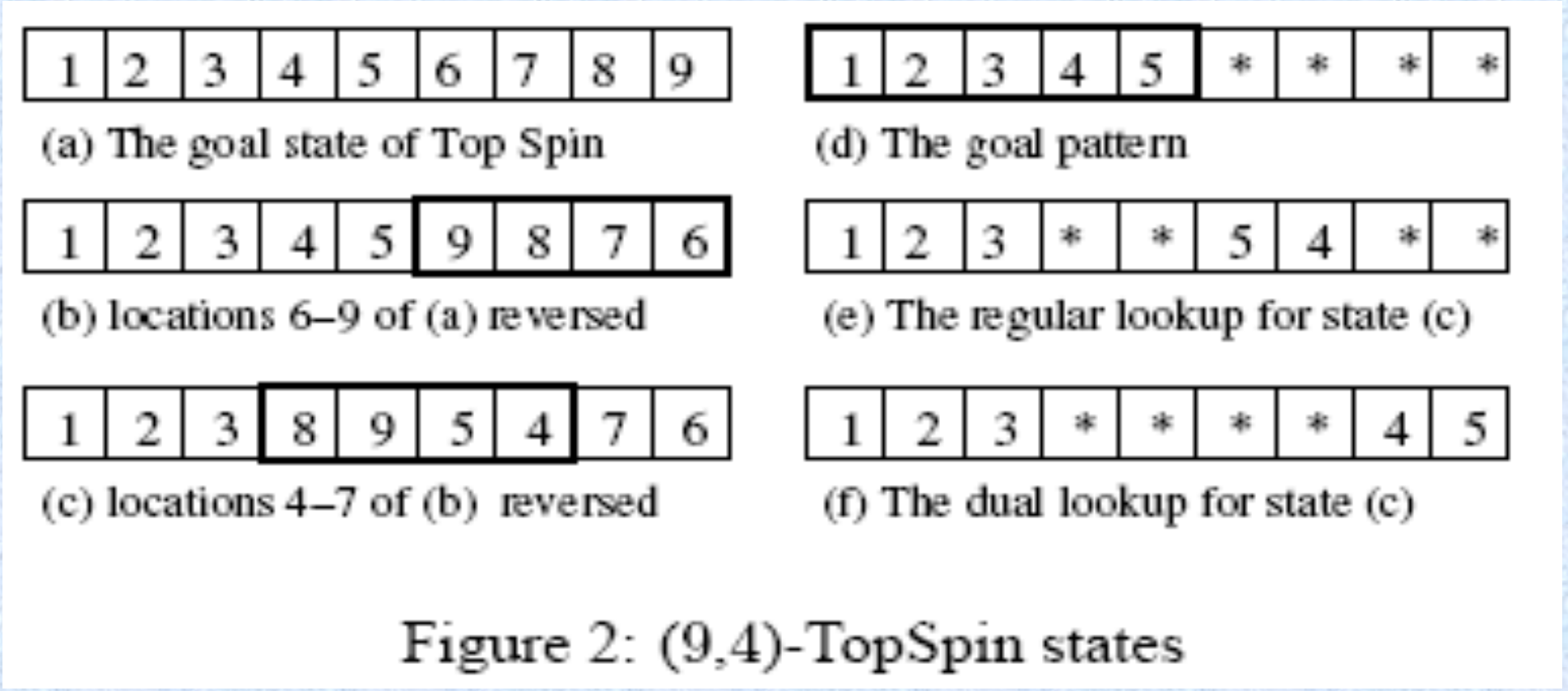


Figure 2: (9,4)-TopSpin states

- Regular lookup for C : PDB[1,2,3,7,6]
- Dual lookup for C: PDB[1,2,3,8,9]

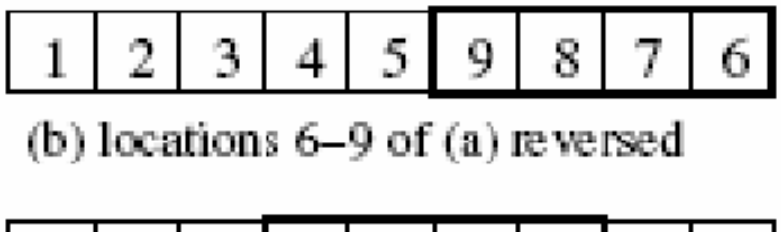
Inconsistency of Dual lookups

Consistency of heuristics:

|h(a)-h(b)| <= c(a,b)

Example: Top-Spin

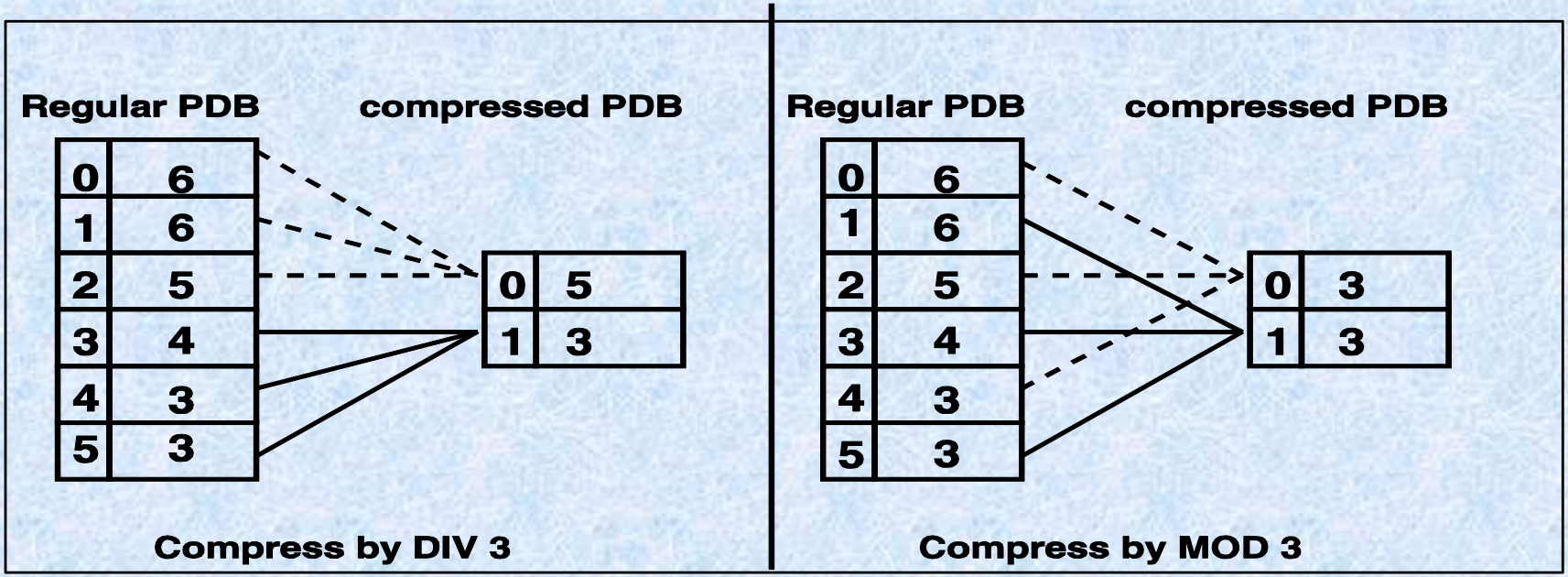
c(b,c)=1



- Both lookups for B
  PDB[1,2,3,4,5]=0
- Regular lookup for C
  PDB[1,2,3,7,6]=1
- Dual lookup for C
  PDB[1,2,3,8,9]=2

	Regular	**Dual**
b	0	**0**
c	1	**2**

Compressing pattern database are inconsistent



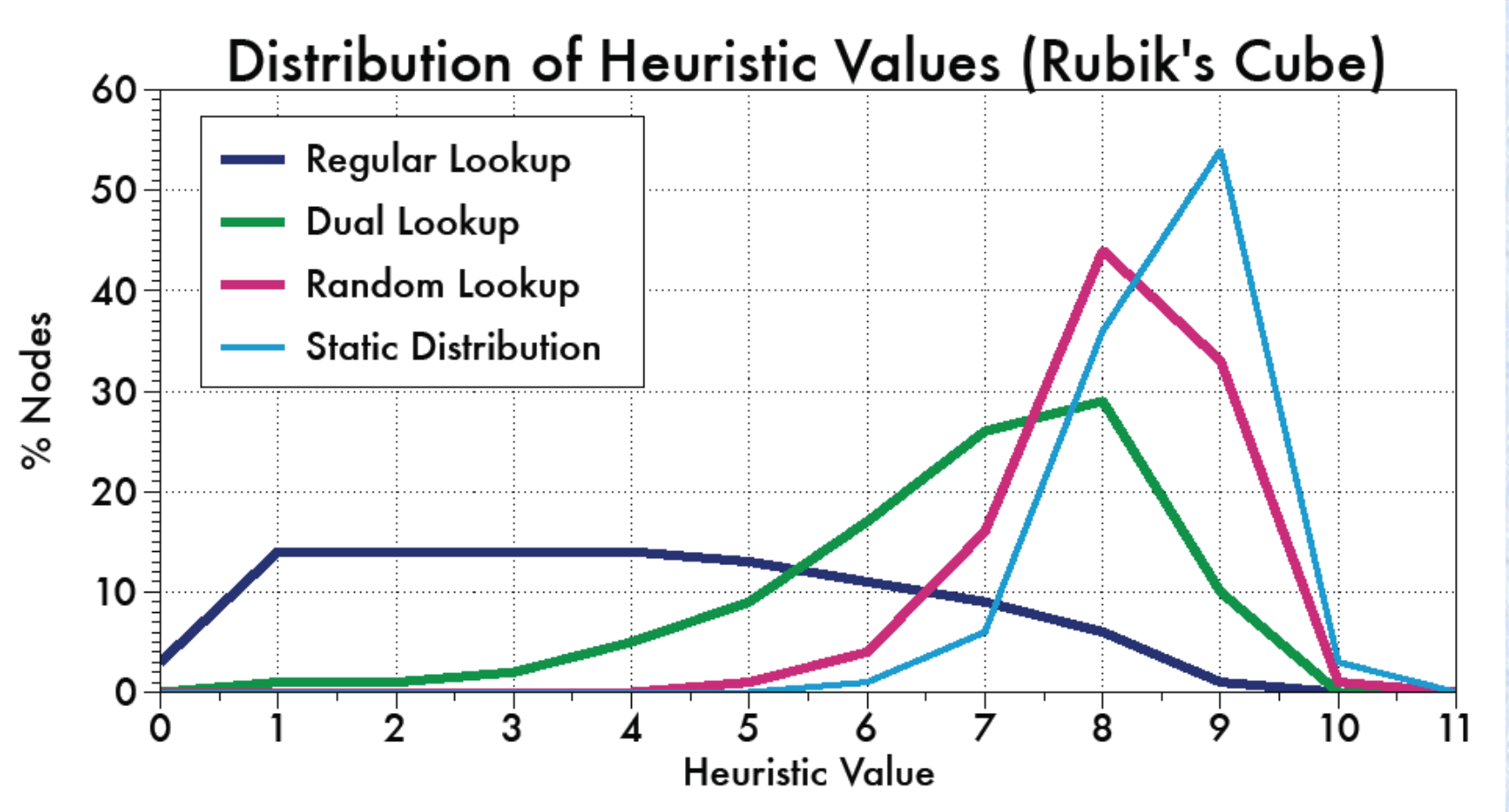
- 3 and 5 are inconsistent

Rubik's cube results

No	Lookups	Nodes	Time
1	Regular	90,930,662	28.18
1	Dual	19,653,386	7.38
1	Dual+BPMX	8,315,116	3.24
1	Random	9,652,138	3.30
1	Random+BPMX	3,828,138	1.25
2	Regular	13,380,154	7.85
4	Regular	10,574,180	11.60

7-edges PDB over 1000 instances of depth 14

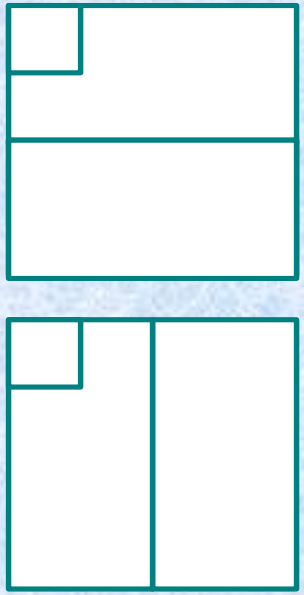
Heuristic value distribution



- Notice that all these heuristics have the same average value

Tile puzzle results

No	Lookups	Nodes	Time
1	Regular	136,289	0.081
1	Dual+BPMX	247,299	0.139
1	Random+BPMX	44,829	0.029
2	Regular+reflected	36,130	0.034
2	2 Random+BPMX	26,862	0.025
3	3 Random+BPMX	21,425	0.026
4	All 4	18,601	0.022



7-8 additive PDB over 1000 instances

Summary

- PDBs are very exciting
- Still ongoing direction
- Many future ideas