2. Backtracking algorithm

Backtracking search system

- The backtracking is the search system where
  - global workspace:
    - contains one path from the start node to the current node and all untested outgoing arcs from its nodes
      - initially this path contains only the start node
      - it terminates: either the current node is the goal or it is the start node with fully tested outgoing arcs
  - rules:
    - expand the path with a new arc that is an untested outgoing arc from the current node
    - delete the last arc out form the path (backtracking step)
  - control strategy: applying the backtracking step at the last case only

Condition of the backtracking step

- dead end: the current node has not got outgoing arc
- checked crossroads: the current node has not got untested outgoing arcs
- circle: the current node is repeated in the current path
- depth bound: the length of the current path is equal to a given limit

Heuristics

- ordering heuristics:
  - gives an order on the outgoing arcs of the current node
- cutting heuristics :
  - cuts the untested outgoing arcs without checking them

First version

- The first version of backtracking (*BT1*) implements the first two conditions of the backtracking step: “dead end” and “checked crossroads”.
- ***In a finite acyclic directed graphs*** (not δ-graph) ***the BT1 always terminates, and if there exists a solution path, then it finds one.***
- It can be implemented with a recursive algorithm
  - Starting: *solution := BT1*(*start*)

DATA := *initial value*
while ¬ *termination condition*(DATA) **loop**
    SELECT R FROM *rules that can be applied*
    DATA := R(DATA)
endloop

BT1

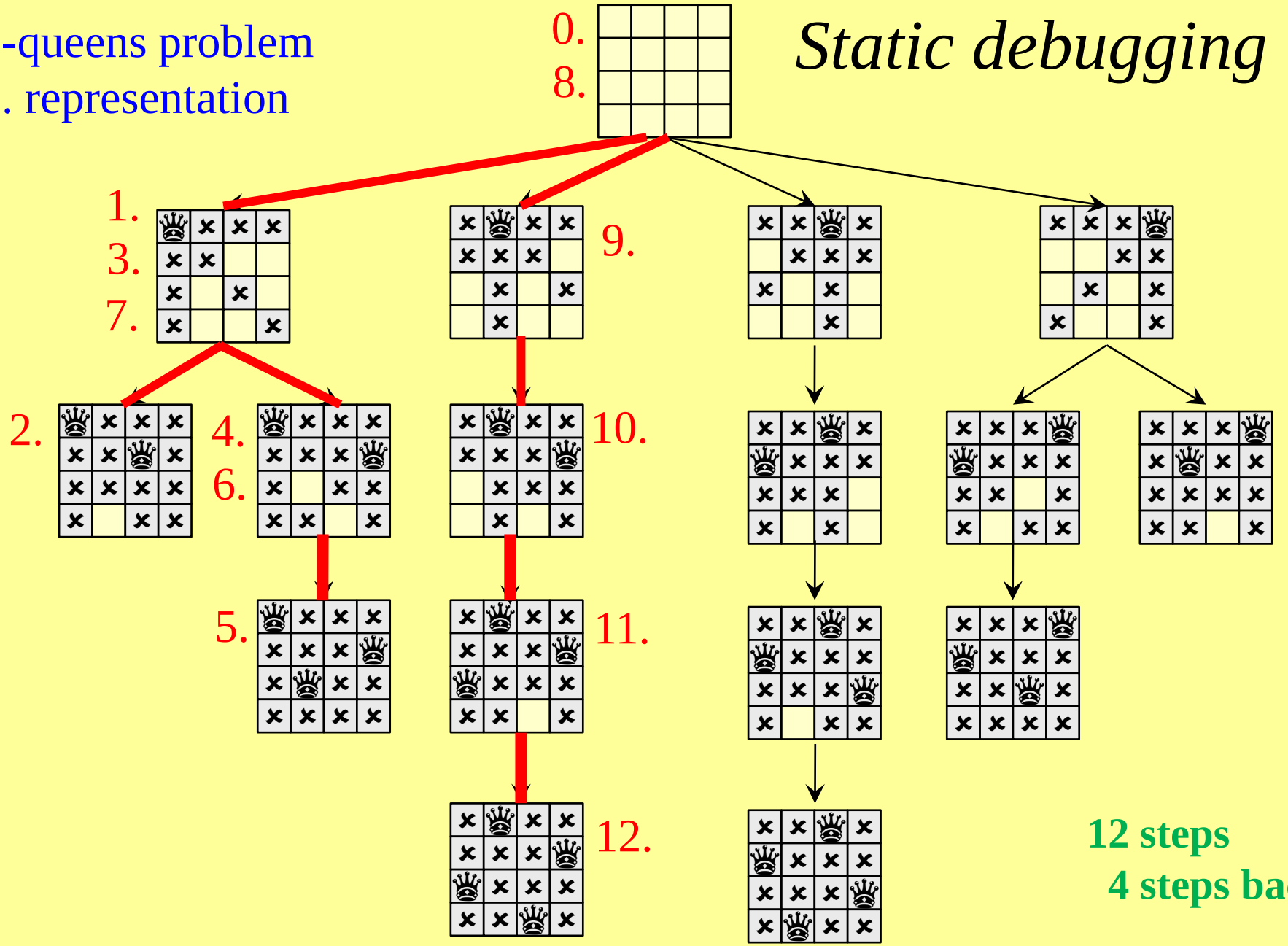
Recursive procedure *BT1*(*current*) **return** *solution*

1. **if** *goal*(*current*) **then** **return**(*nil*) **endif**
2. **for** $\forall new \in \Gamma(current) - \pi(current)$ **loop**
3. *solution* := *BT1*(*new*)
4. **if** *solution* ≠ *fail* **then**
5. **return**(*concat*((*current*, *new*), *solution*) **endif**
6. **endloop**
7. **return**(*fail*)

end

n-queens problem
3. representation

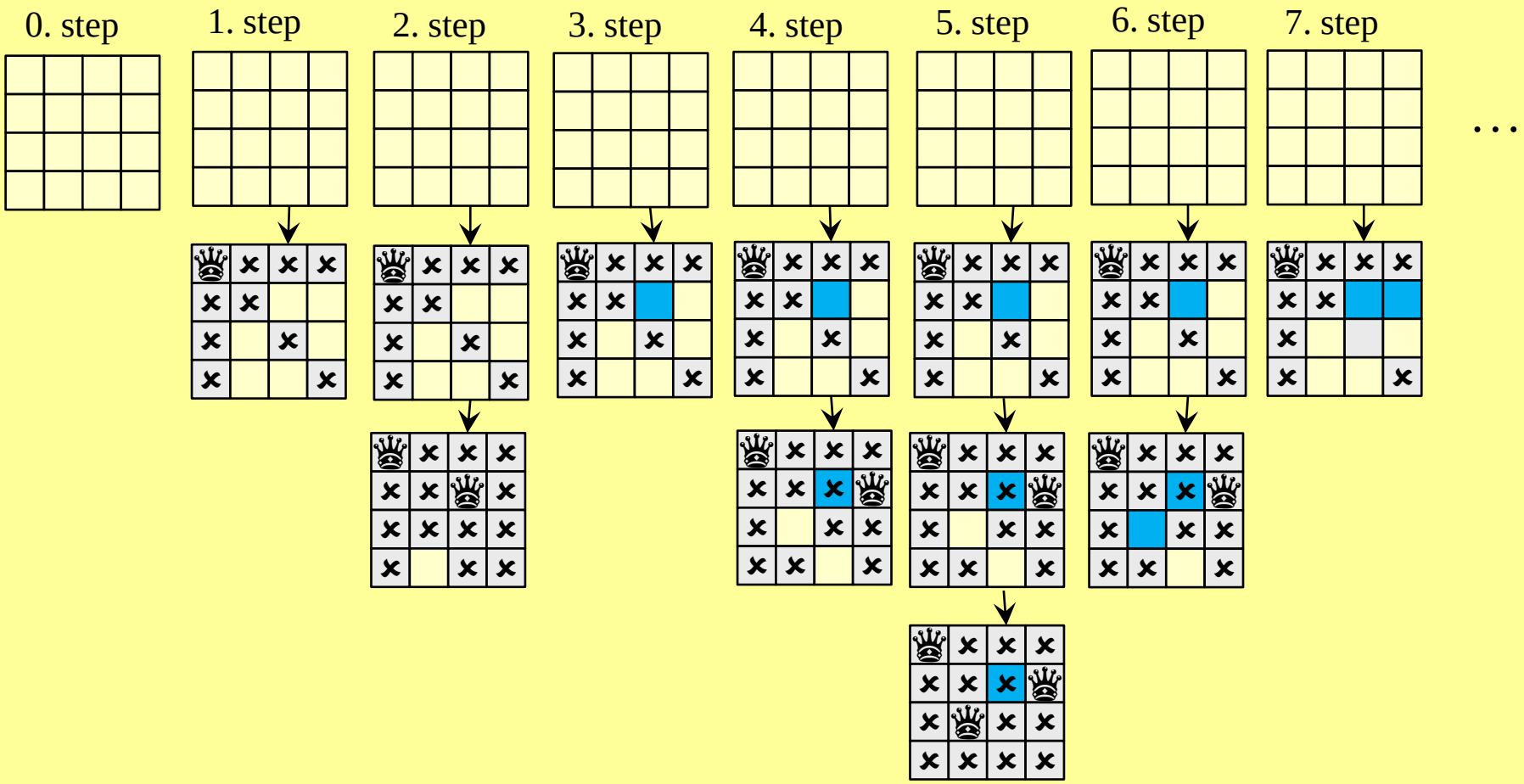
Static debugging



12 steps
4 steps back

n-queens problem
3. representation

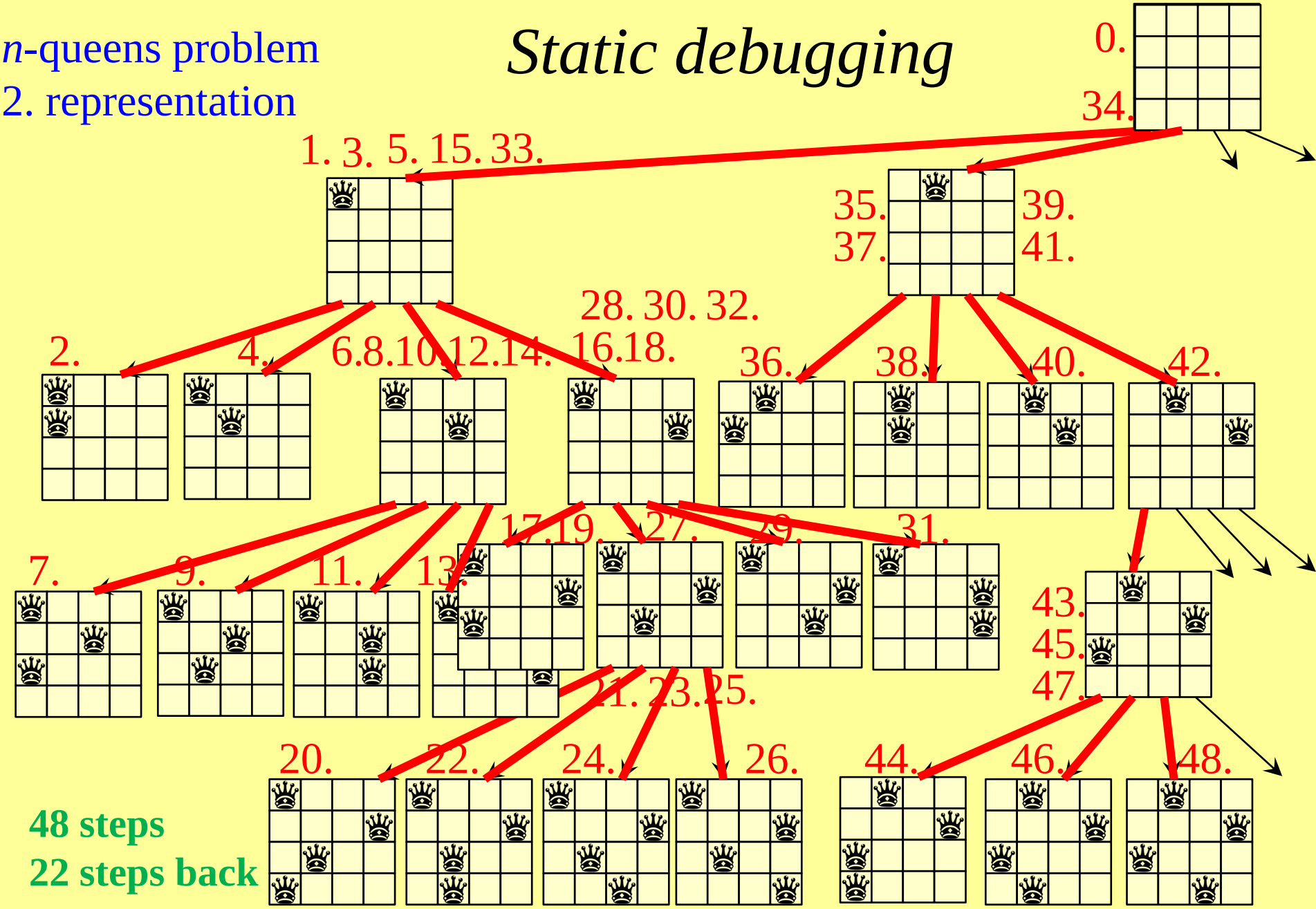
Dynamic debugging



12 steps
4 steps back

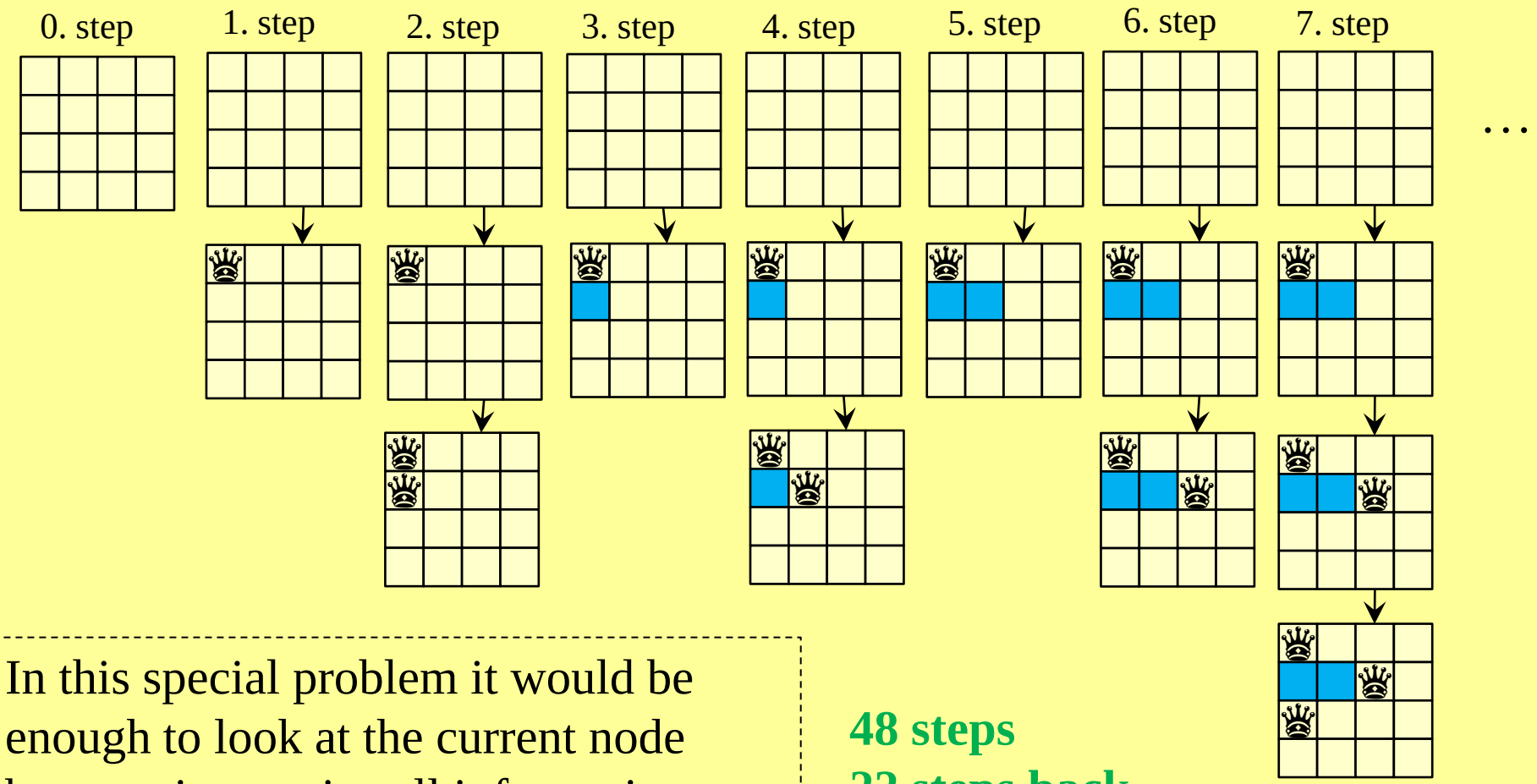
n-queens problem
2. representation

Static debugging



48 steps
22 steps back

Dynamic debugging



In this special problem it would be enough to look at the current node because it contains all information about the current path.

48 steps
22 steps back

Ordering heuristics for n-queens problem

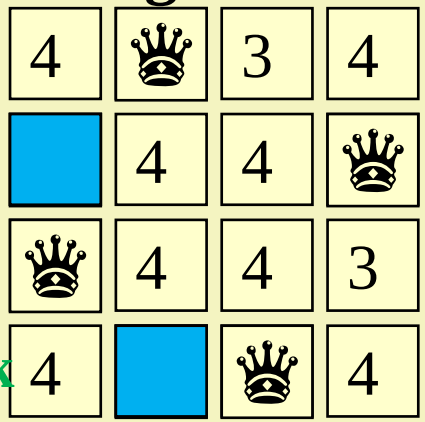
These heuristics assign a value to each square.
In each step the search selects the square that has the best value among the untested squares of the current row.

4	3	3	4
3	4	4	3
3	4	4	3
4	3	3	4

- Diagonal: the length of the longest diagonal passing through a square.
- Diagonal + odd-even: Odd-even is a secondary principle that orders the squares with the same primary value: in the odd rows from left to right, in the even rows from right to left.
- Number of free squares (that are not attacked) that remains after placing a new queen.
- Difference of the number of free squares of the state before placing a new queen and the state after that.

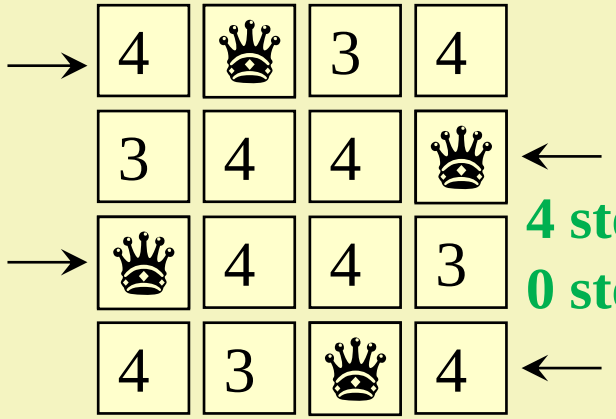
Ordering heuristics for n-queens problem

Diagonal:



8 steps
2 steps back

Diagonal + odd-even



4 steps
0 steps back

n = 4	None	Diag	Diag+odd-even
2. repr.	22/48	2/8	0/4
3. repr.	4/12	0/4	0/4

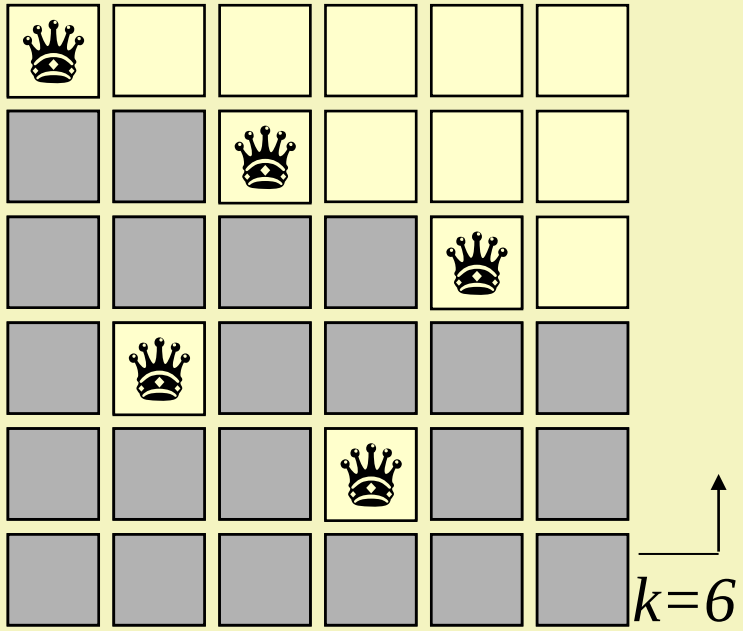
2. repr.	None	Diag
n = 4	22/48	2/8
n = 5	0/5	2/9
n = 6	25/56	3/12
n = 10	92/194	103/216

Finding cutting strategies for n-queens problem

In each step after placing the k^{th} queen, free squares of the remaining rows may be reduced

for *i=k+1 .. n* ***loop***
Mark(i,k)

Mark(*i*,*j*) : deletes the free squares from the i^{th} row if they are attacked by the k^{th} queen



$Square_6 = \emptyset$

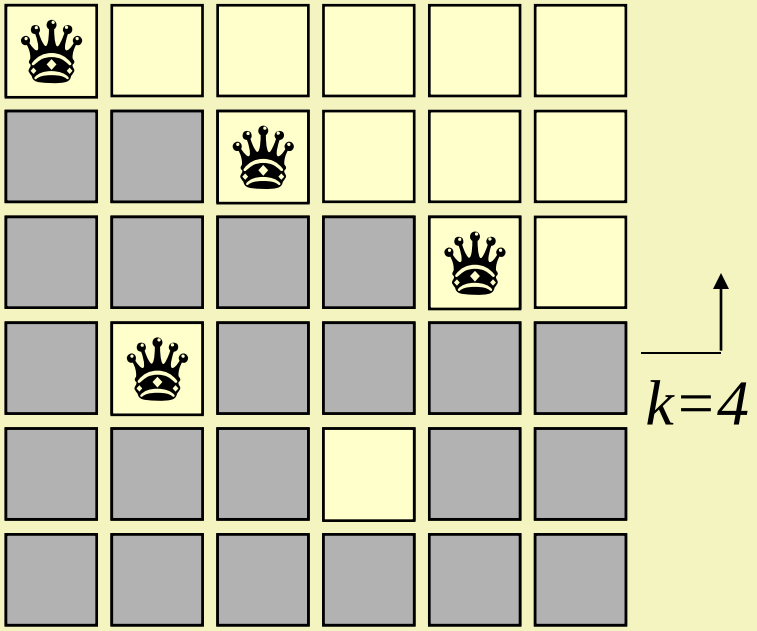
$Square_i$ = { free squares in the i^{th} row }

This is the standard backtracking method

If $Square_k = \emptyset$ then step back.

Forward Checking

***FC* algorithm:**
BT1 +
if there is no free square
in some of the remaining rows
then the algorithm steps back.



Partial Look Forward

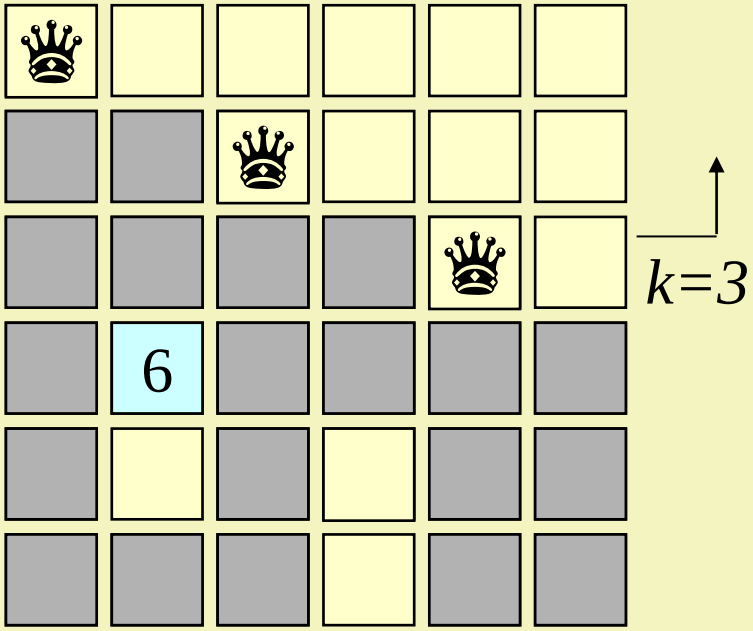
***PLF* algorithm**:

FC +

for *i=k+1 .. n* **loop**

for *j=i+1 .. n* **loop**

Filter(i,j)



i = 4, j = 6

$Square_4 = \emptyset$

Filter(i,j) : deletes the free square from the i^{th} row if it is attacked by every free square in the j^{th} row

Look Forward

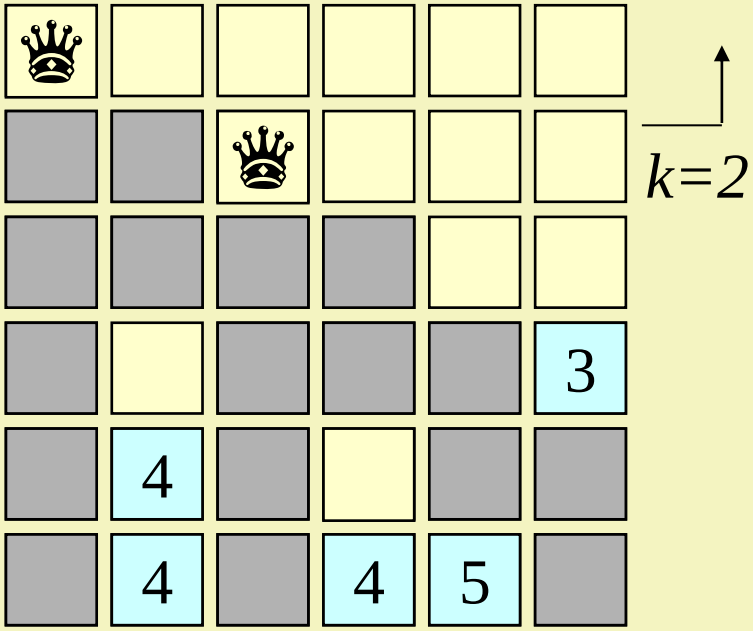
***LF* algorithm**:

FC +

for *i=k+1 .. n* **loop**

for *j=k+1 .. n* és *i≠j* **loop**

Filter(i,j)



i = 4, j = 3

i = 5, j = 4

i = 6, j = 4

i = 6, j = 5

$Square_6 = \emptyset$

LF once more

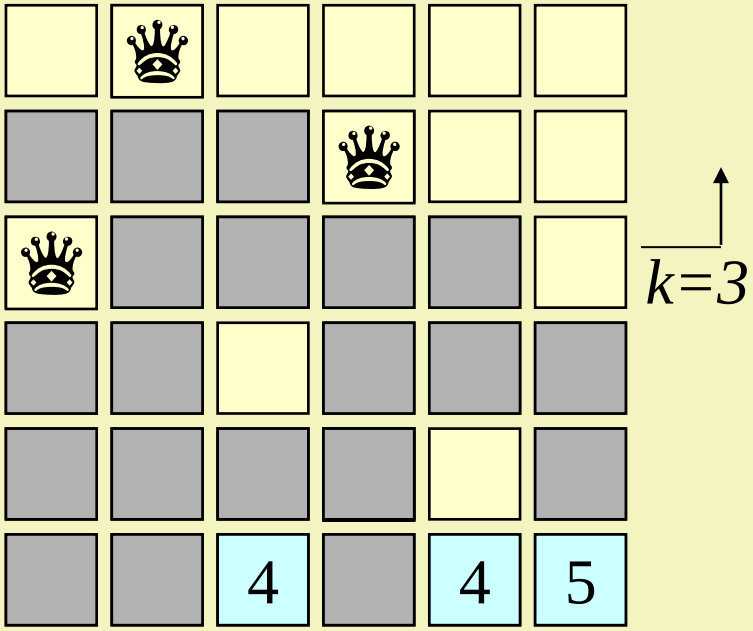
***LF* algorithm**:

FC +

for *i=k+1 .. n* **loop**

for *j=k+1 .. n* és *i≠j* **loop**

Filter(i,j)



$i = 6,\ j = 4$

$i = 6,\ j = 5$

$Square_6 = \emptyset$

AC1

***AC1* algorithm**:

FC +

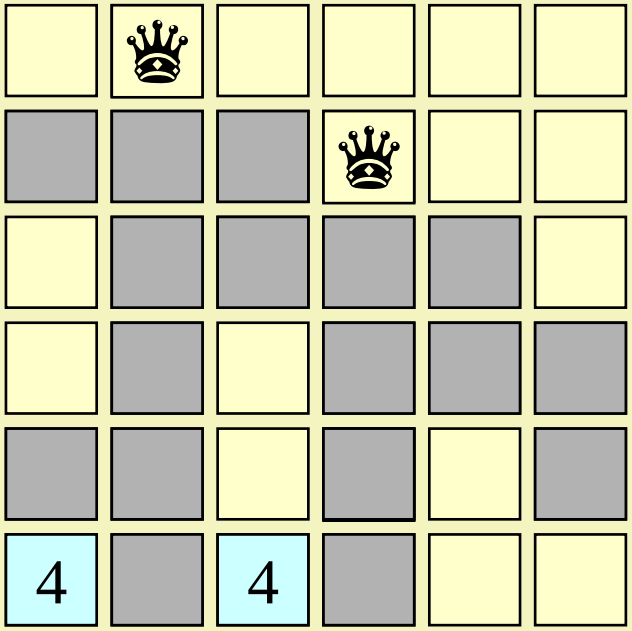
repeat

for *i=k+1 .. n* **loop**

for *j=k+1 .. n* és *i≠j* **loop**

Filter(i,j)

until *there was deleting*



1. turn *i = 6, j = 4*

AC1

***AC1* algorithm**:

FC +

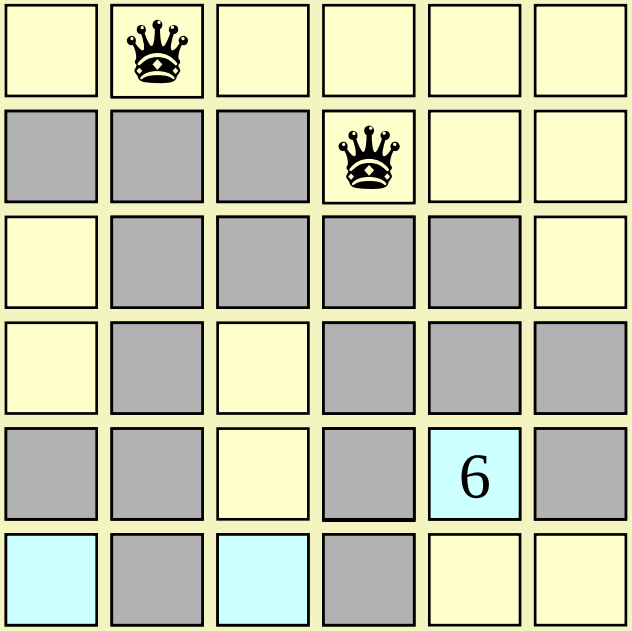
repeat

for *i=k+1 .. n* **loop**

for *j=k+1 .. n* és *i≠j* **loop**

Filter(i,j)

until *there was deleting*



2. turn *i = 5, j = 6*

AC1

***AC1* algorithm**:

FC +

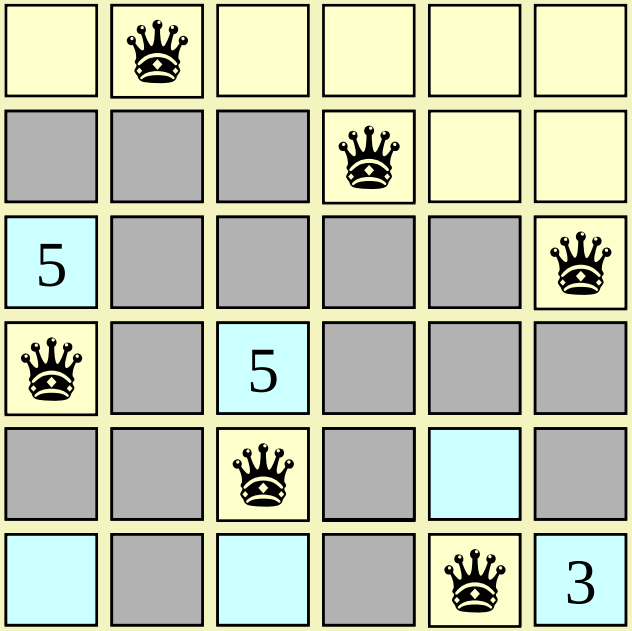
repeat

for *i=k+1 .. n* **loop**

for *j=k+1 .. n* és *i≠j* **loop**

Filter(i,j)

until *there was deleting*



3. turn

i = 3, j = 5

i = 4, j = 5

i = 6, j = 3

A new representation model: constraint satisfaction

- The *n*-queens problem can be represented in an other form:
  - Find the positions of queens $(x_1, \ldots, x_n) \in D_1 \times \ldots \times D_n$ (the i^{th} queen is on the x_i position and $D_i = Square_i$) where there is no attack between the i^{th} and j^{th} queens for all $i,j \in [1..n]$.
  - Possible squares of the i^{th} queen: $D_i = Square_i \subseteq \{1, \ldots, n\}$
  - Constraints "no attack" are binary relations: $C_{ij} \subseteq D_i \times D_j$

    $C_{ij}(x_i,x_j) \sim \neg\text{attack}((i,x_i),(j,x_j)) \equiv (x_i \neq x_j \text{ and } |x_i - x_j| \neq |i-j|)$
- The graph coloring problem (*n* vertices, *m* colors) with a similar representation:
  - Possible colors of the i^{th} vertex: $D_i \subseteq \{1, \ldots, m\}$ ($i=1..n$)
  - Constraints on the adjacent (i,j) vertices: $C_{ij}(x_i,x_j) \sim x_i \neq x_j$

Secondary control strategy

- The earlier presented cuttings can be described with the next formalization:

  $Mark(i,k)$: $D_i := D_i - \{e \in D_i \mid \neg C_{ik}(e, x_k)\}$

  $Filter(i,j)$: $D_i := D_i - \{e \in D_i \mid \forall f \in D_j : \neg C_{ij}(e,f)\}$

- These methods are independent of the meaning of the relation C_{ij} so they can apply not only in the solution of the *n*-queens problem but also in all problems that are modeled by constraint satisfaction representation.
- Thus these cuttings are not heuristics since they do not contain knowledge about the problem domains. They are based on the speciality of the constraint satisfaction model hence they are a sort of secondary control strategies.

Second version

- The second version of backtracking (*BT2*) implements all conditions of the backtracking step.
- ***In δ-graphs the BT2 always terminates, and if there exists a solution path shorter than the depth bound, then it finds a solution path.***
- It can be implemented with a recursive algorithm Starting: *solution := BT2*(<*start*>)

DATA := *initial value*
while ¬ *termination condition*(DATA) **loop**
    SELECT R FROM *rules that can be applied*
    DATA := R(DATA)
endloop

BT2

Recursive procedure *BT2*(*path*) **return** *solution*

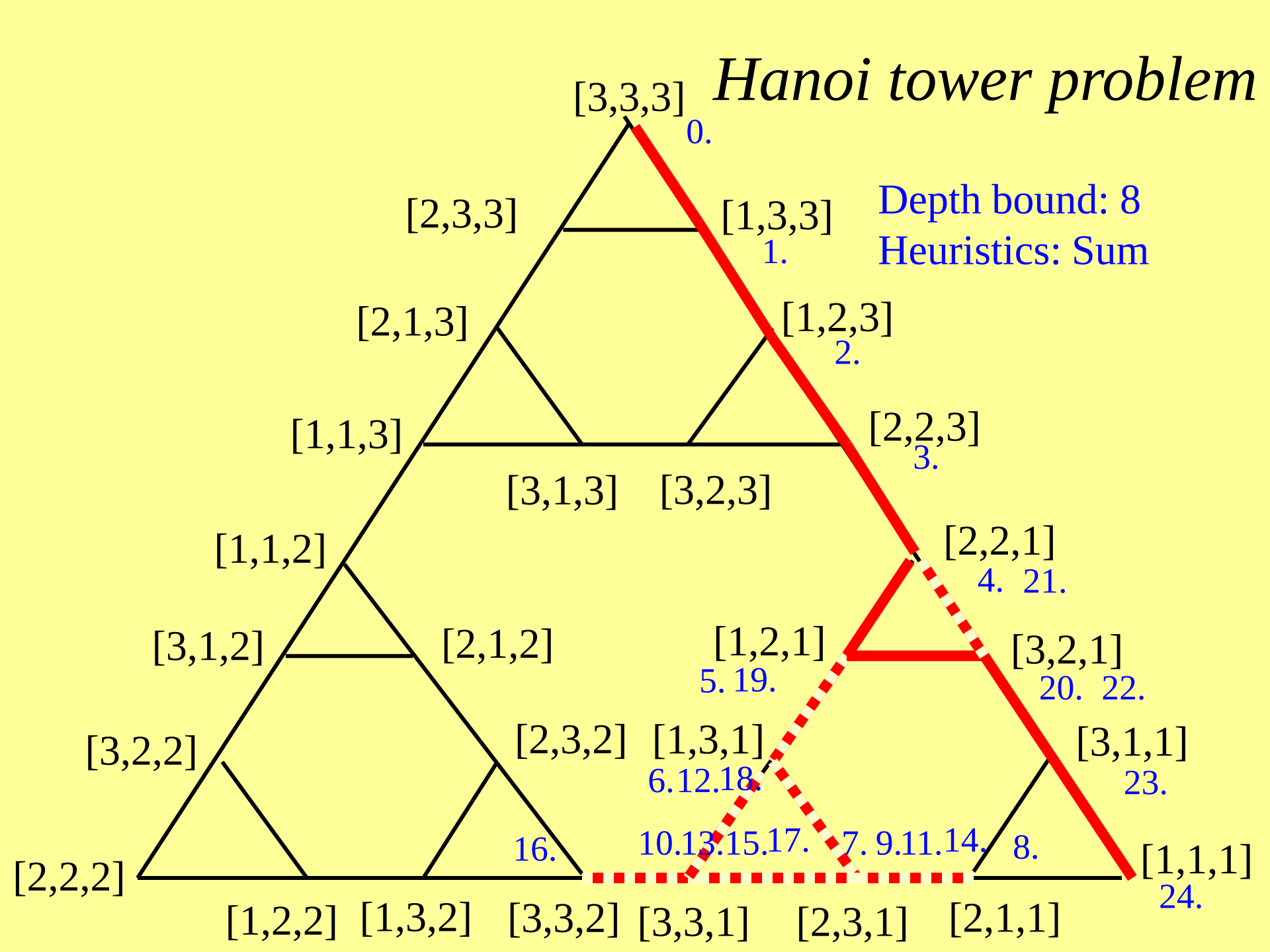
1. *current* := *last_node*(*path*)
2. **if** *goal*(*current*) **then** **return**(*nil*)
3. **if** *length*(*path*) ≥ *limit* **then** **return**(*fail*)
4. **if** *current* ∈ *remain*(*path*) **then** **return**(*fail*)
5. **for** ∀*new* ∈ Γ(*current*) −π(*current*) **loop**
6. *solution* := *BT1*(*concat*(*path*, *new*))
7. **if** *solution* ≠ *fail* **then**
8. **return**(*concat*((*curent*, *new*), *solution*) **endif**
9. **endloop**
10. **return**(*fail*)

end

Remarks

- If the length of the shortest solution path is greater than the depth bound, then *BT2* terminates without solution path.
- The observing circles can be ignored because the implementation of the depth bound alone ensures the outcome of *BT2*.
  - This simplification can mend the efficiency if there are no short circles in the representation graph (ecxept of the 2-length circles that can be avoided by the storing parent.)
  - In this case it is enough to give the recursive procedure only the length of the current path instead of the whole current path besides the current node and its parent.

Hanoi tower problem



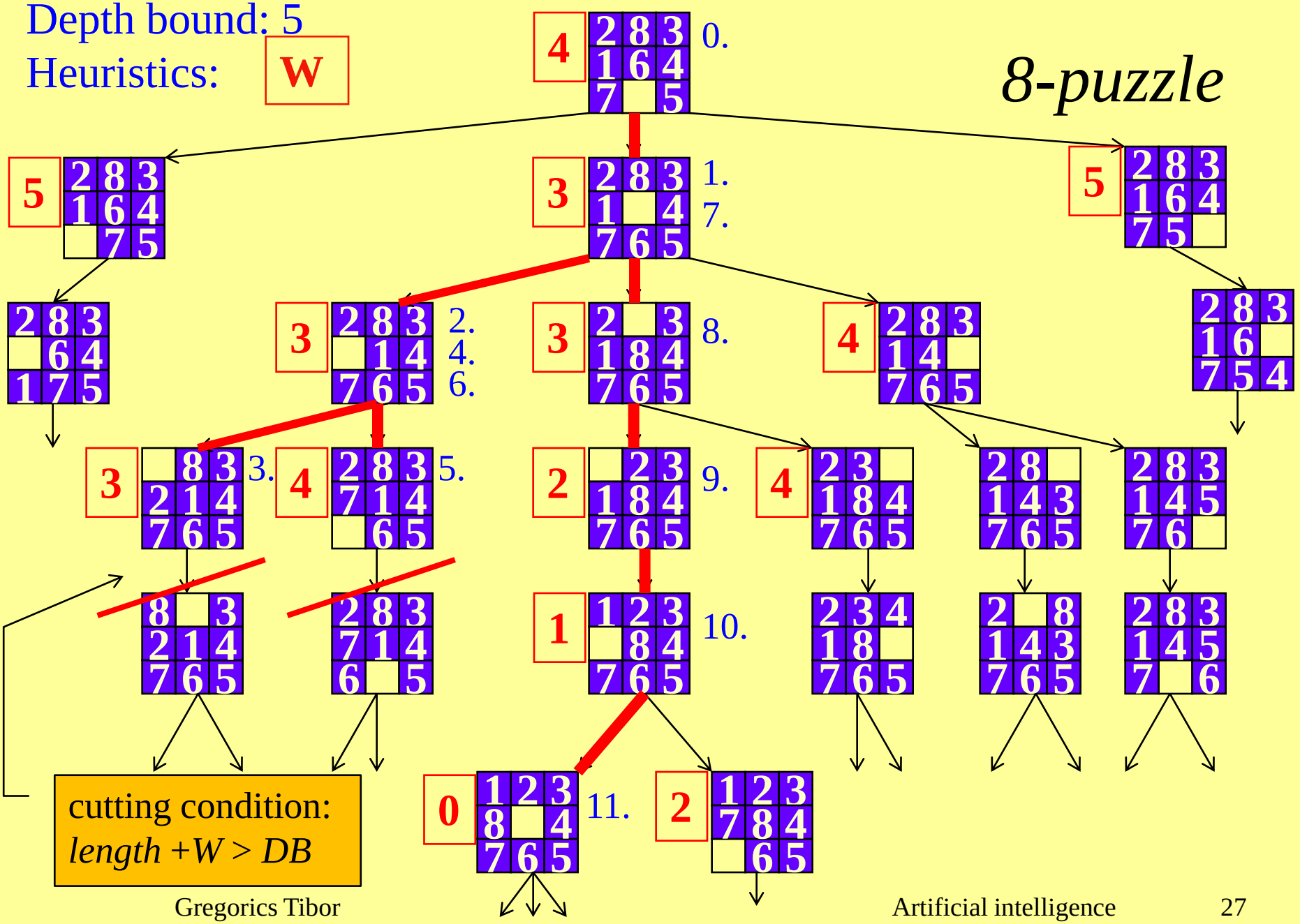
Depth bound: 8

Heuristics: Sum

Depth bound: 5

Heuristics: W

8-puzzle

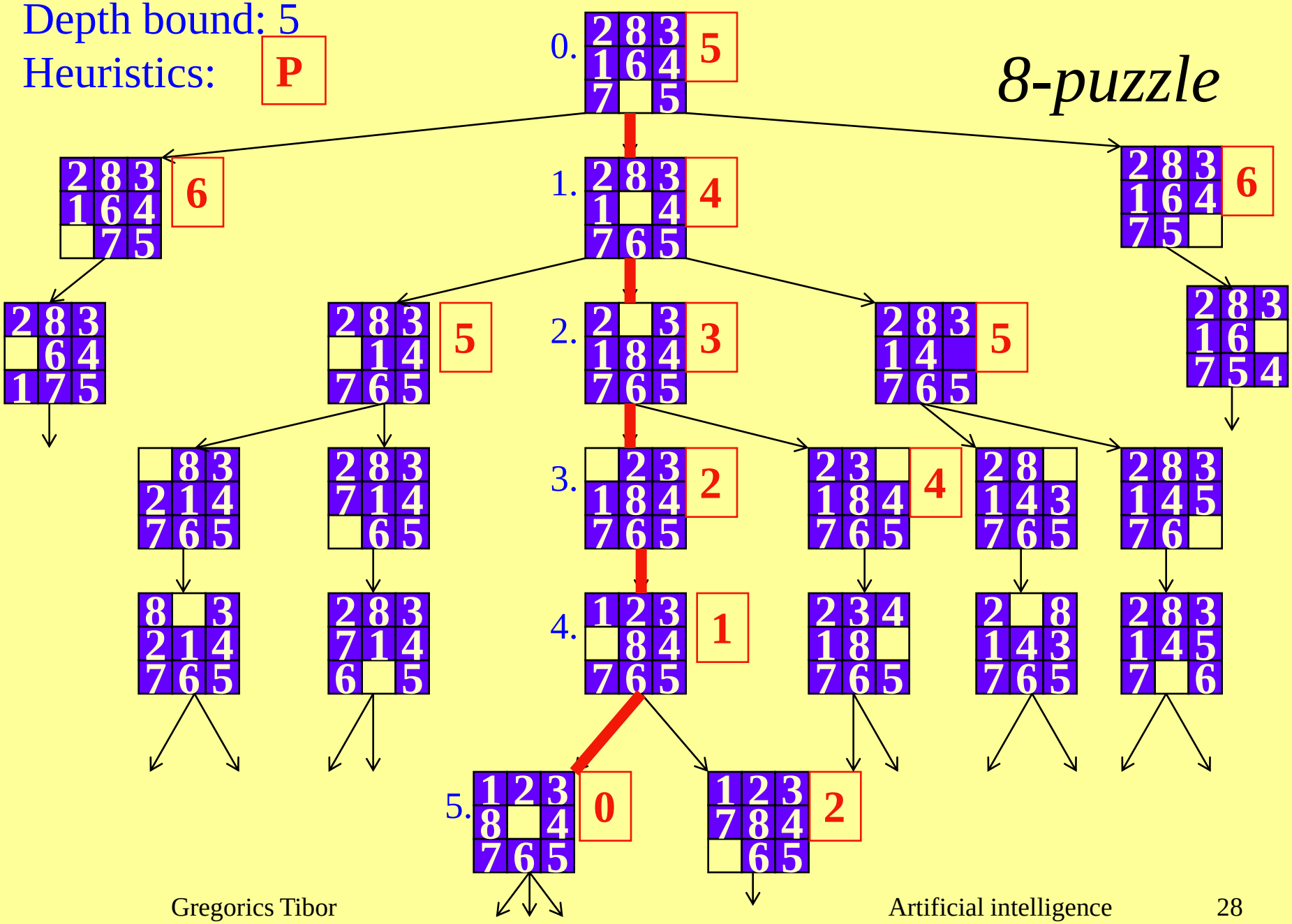


cutting condition:
length +W > DB

Depth bound: 5

Heuristics: **P**

8-puzzle



Conclusions

- Advantages
  - always terminates, finds solution
  - implementation is simple
  - small memory
- Disadvantages
  - no optimal solution
  - wrong choice at the first stage of the search can be undone only after many steps
  - the same part of the graph can be traversed many times