Backtracking algorithm

Backtracking search system

- Backtracking is the search system where
  - global workspace:
    - contains one path from the start node to the current node with all untested outgoing arcs from the nodes of this path
      - initially this path contains only the start node
      - the search terminates: either the current node is the goal, or the outgoing arcs of the start node are completely tested
  - searching rules:
    - append a new untested outgoing arc driving from the current node to the end of the current path
    - remove the last arc of the current path (backtrack)
  - control strategy: applying the backtracking in last case

Conditions of the backtracking

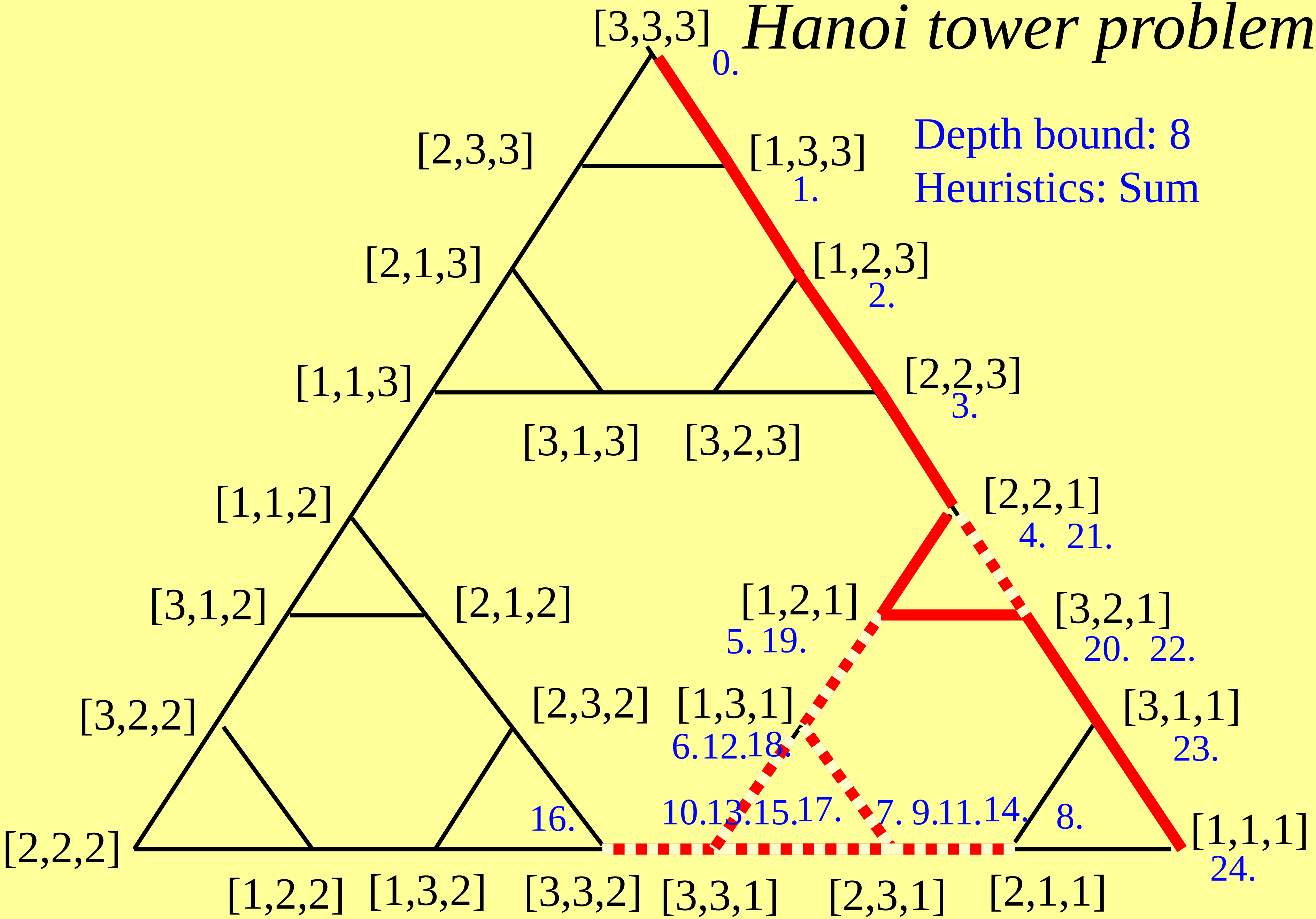
- dead end: the current node has no outgoing arcs
- checked crossroads: the current node has no untested outgoing arcs (all outgoing arcs drive to a dead end)
- cycle: a node is repeated in the current path (the current node is in the rest of the current path)
- depth bound: the length of the current path is equal to a given limit

Refinements of the control strategy

- The general control strategy might be extended with secondary strategies which may be model dependent control strategies (derived from the particularity of the model of the problem) or heuristic control strategies (derived from the knowledge about the problem domain)
- The secondary strategy may be
  - ordinal strategy : that ranks the outgoing arcs of the current node, and tries them to apply in order of their preference
  - cutting strategy : that ignores some untested outgoing arcs of the current node

Hanoi tower problem

Depth bound: 8
Heuristics: Sum



First version: *BT1*

- The first version of the backtracking algorithm (*BT1*) observes only the first two conditions of the backtracking: “dead end” and “checked crossroads”.
- ***In a finite acyclic directed graph the BT1 always terminates, and if there exists a solution path, then it finds one***.
- It can be implemented with a recursive procedure
  - Starting: *solution* := *BT1*(*start*)

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

BT1

A ~ set of arcs
A^* ~ set of finite sequences of arcs

N ~ nodes

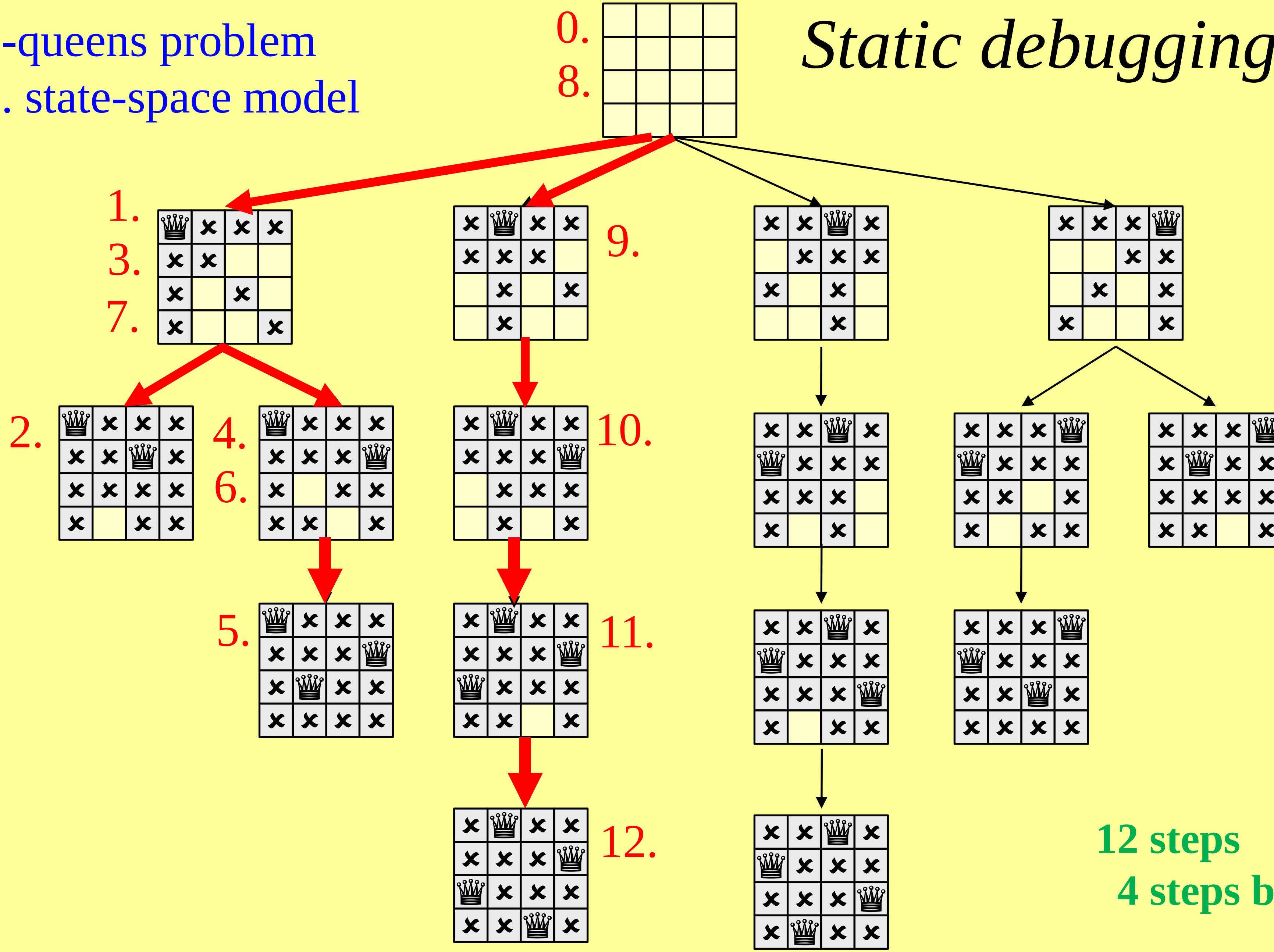
recursive procedure $BT1(current : N)$ **return** $(A^*; fail)$
1. **if** $goal(current)$ **then** **return**(nil) **endif**
2. **for** $\forall new \in \Gamma(current)$ **loop**
3. $solution := BT1(new)$
4. **if** $solution \neq fail$ **then**
5. **return**$(concat((current, new), solution))$ **endif**
6. **endloop**
7. **return**$(fail)$

end

n-queens problem

3. state-space model

Static debugging

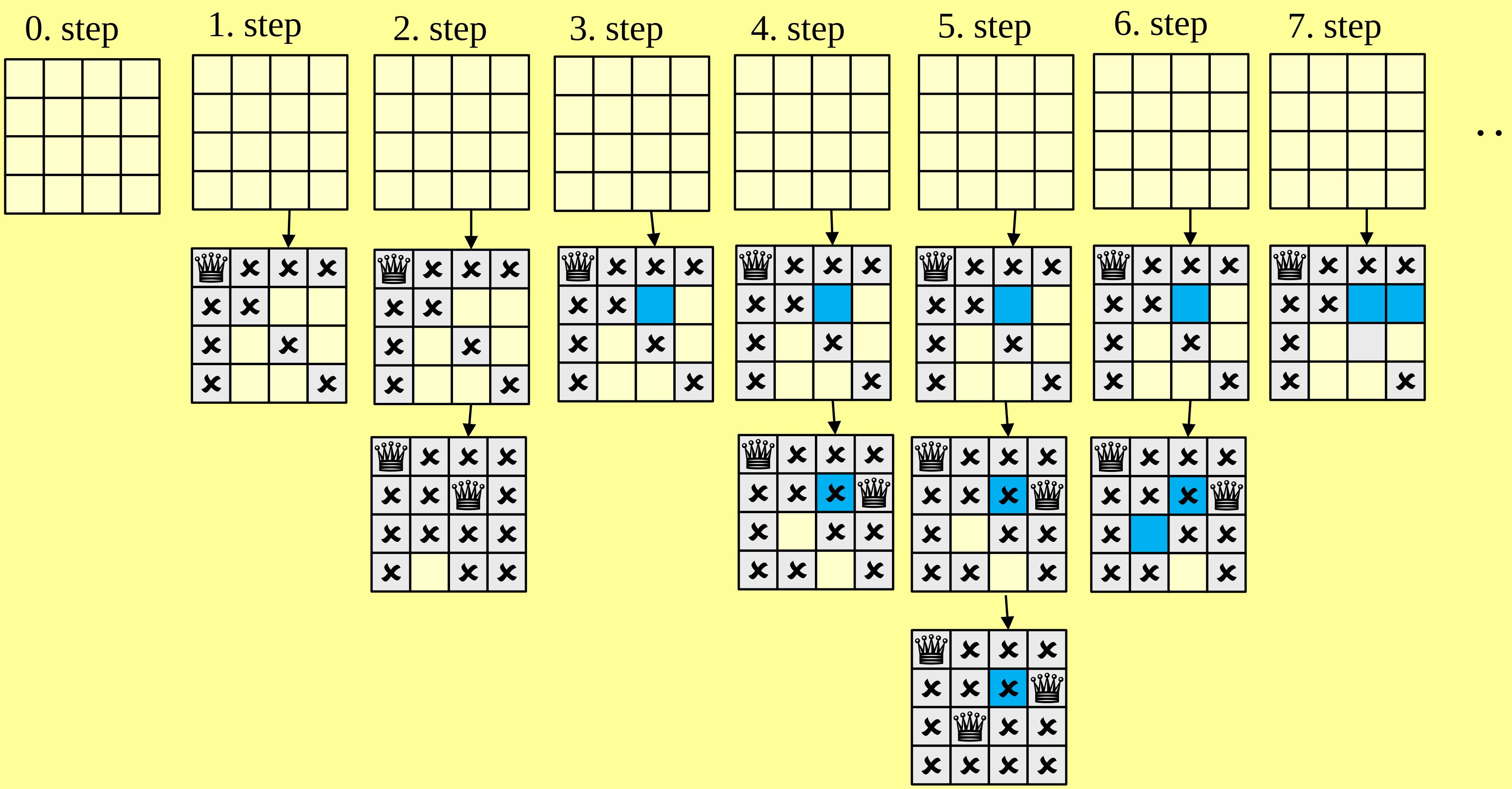


12 steps

4 steps back

n-queens problem
3. state-space model

Dynamic debugging



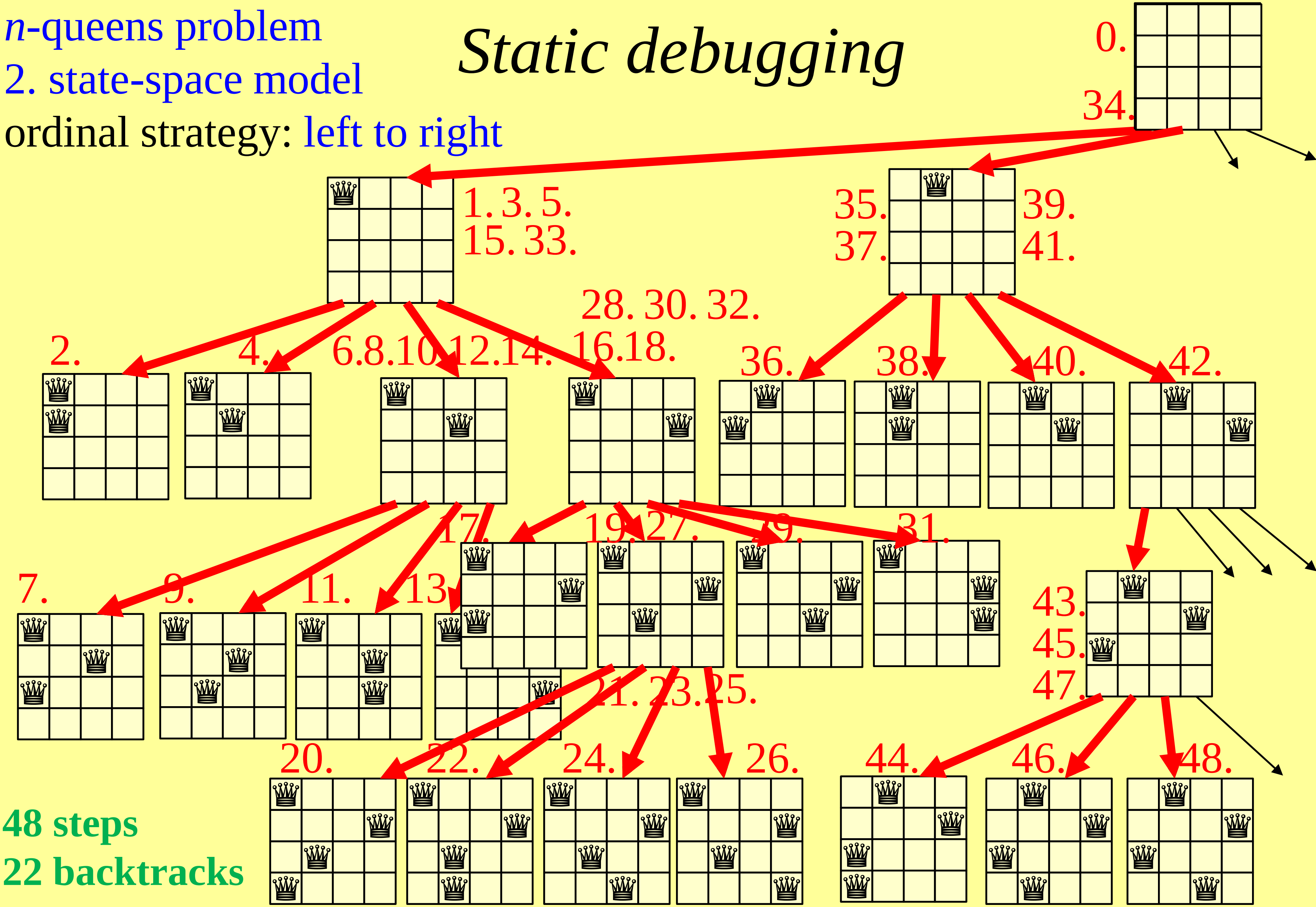
12 steps
4 steps back

n-queens problem

2. state-space model

ordinal strategy: left to right

Static debugging

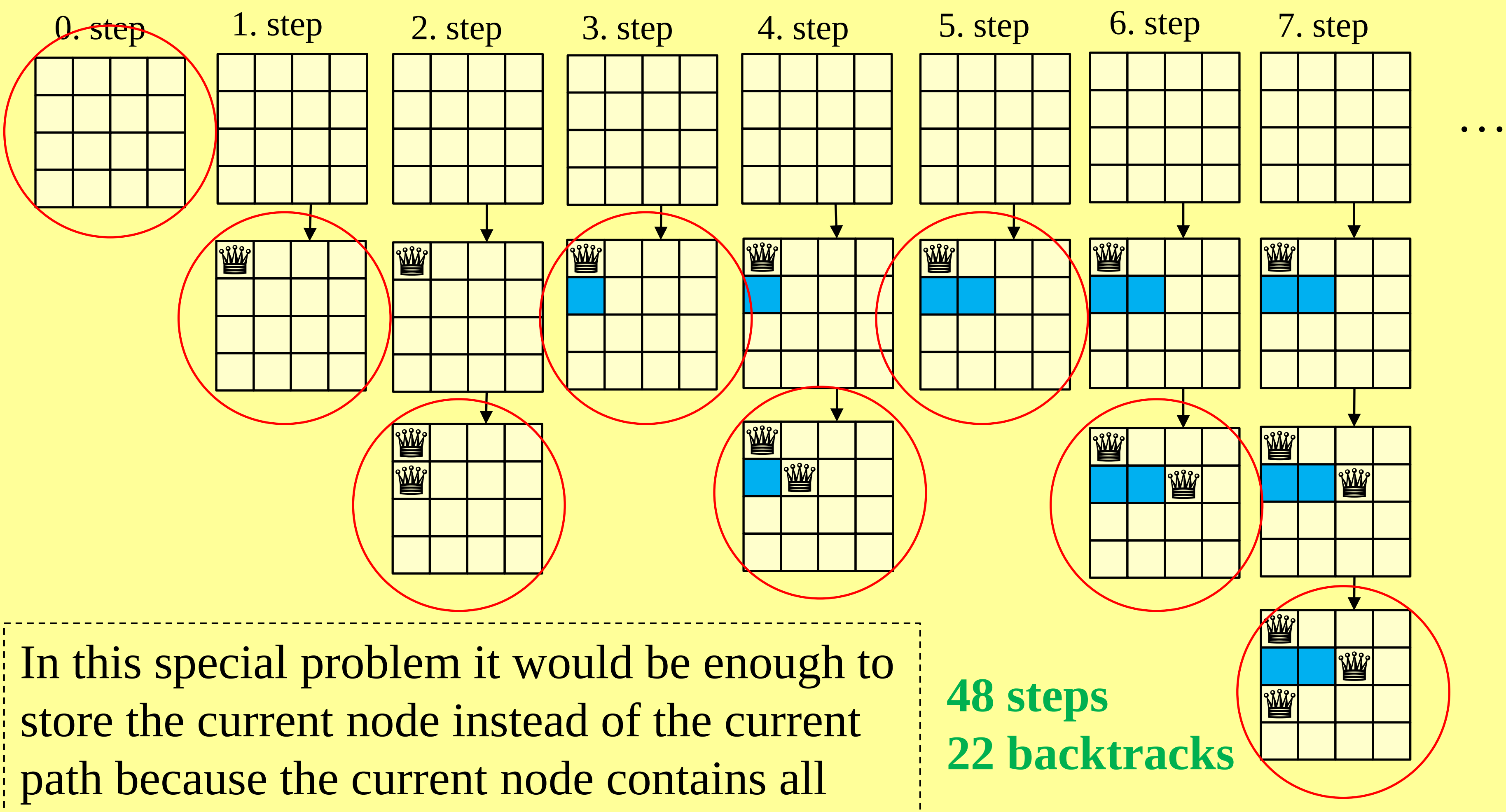


48 steps

22 backtracks

n-queens problem
2. state-space model
ordinal strategy: left to right

Dynamic debugging



In this special problem it would be enough to store the current node instead of the current path because the current node contains all information about the current path.

48 steps
22 backtracks

Ordinal heuristics for n-queens problem

The squares of the board may be ranked by heuristics.

- Diagonal heuristics assigns a square the length of the longest diagonal passing through it.

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

- Odd-even heuristics gives the squares integers from 1 up to n ordered by increasing in the odd rows, and by decreasing in the even rows.

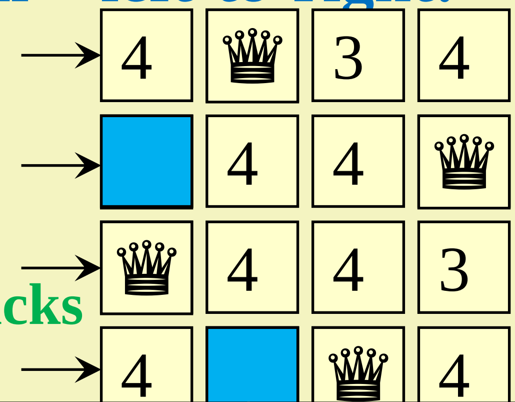
1	2	3	4
4	3	2	1
1	2	3	4
4	3	2	1

- Number of attacked squares: how many free squares of the remaining empty rows become under attack after placing a queen on a certain square.

♛	×	×	×
×	×	3	2
×		×	
×			×

Ordinal heuristics for n-queens problem

diagonal + left-to-right:



8 steps
2 backtracks

diagonal + odd-even:

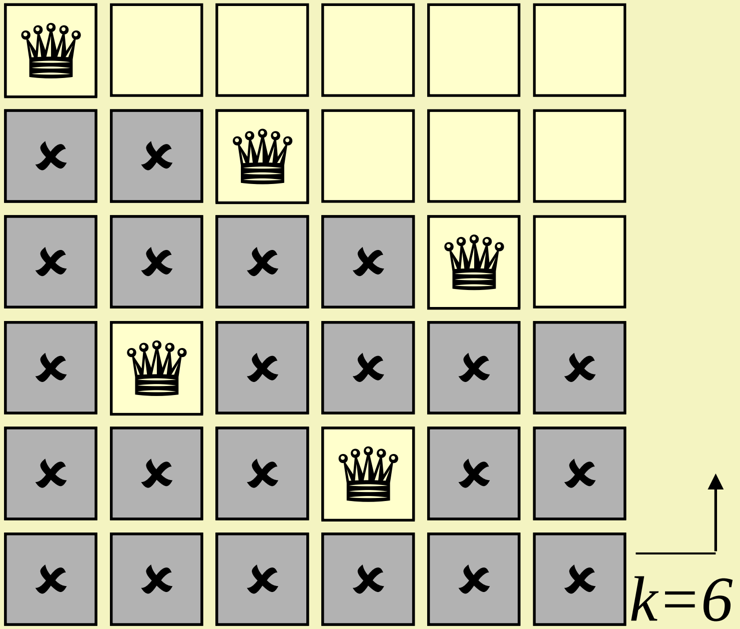
4 steps
0 backtracks

2. model	None + left-to-right	Diag + left-to-right
n = 4	22/48	2/8
n = 5	10/25	10/25
n = 6	165/336	63/132
n = 7	35/77	80/167
n = 8	868/1744	196/400

n = 4	None + left-to-right	Diag + left-to-right	Diag + odd-even
2. model	22/48	2/8	0/4
3. model	4/12	0/4	0/4

BT1 *without heuristics*

D_i ={free squares in the i^{th} row}



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

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

Remove(*i*,*k*) : removes the free squares from the set D_i which are attacked by the k^{th} queen

BT1: **if** $D_k = \emptyset$ **then** it must backtrack.

Forward Checking

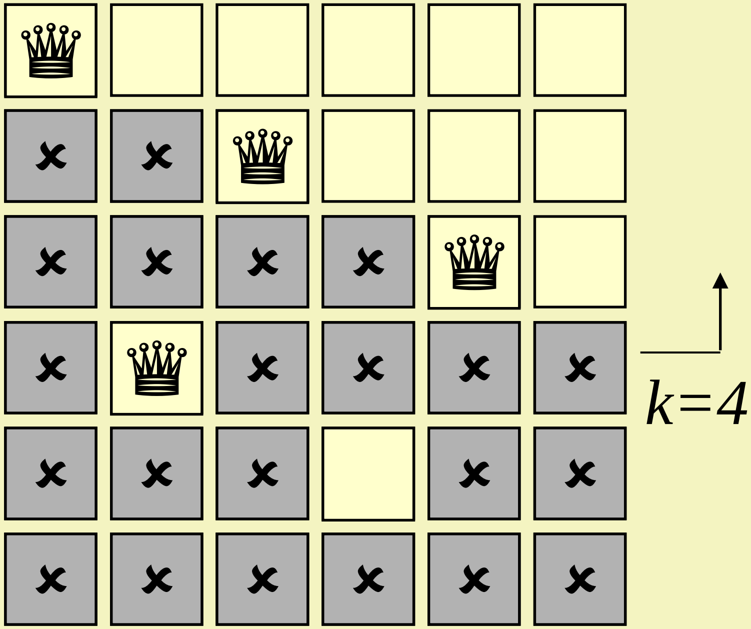
***FC* algorithm:**

BT1

+

if $\exists i \in [k+1..\ n]$: $D_i = \emptyset$

then it must backtrack.



$D_6 = \emptyset$

Partial Look Forward

***PLF* algorithm**:

BT1

\+

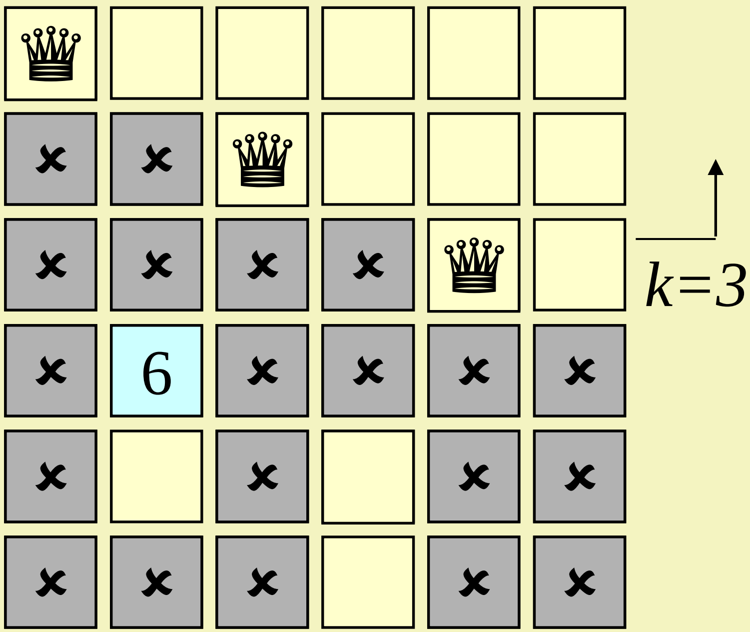
for $i=k+1$ .. n **loop**

for $j=i+1$ .. n **loop** $(i<j)$

Filter(i,j)

if $\exists i \in [k+1 .. n]$: $D_i = \emptyset$

then it must backtrack.



Filter(i,j) : removes the free square from the i^{th} row if all free squares of the j^{th} row are attacked by it

$i = 4, j = 6$ $D_4 = \emptyset$

Look Forward

***LF* algorithm**:

BT1

+

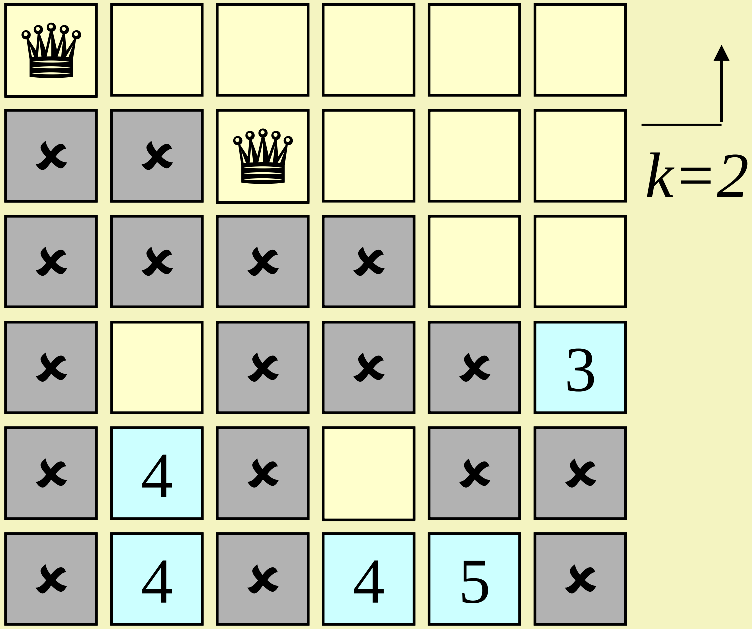
for $i=k+1 \,..\, n$ **loop**

for $j=k+1 \,..\, n$ **and** $i\neq j$ **loop**

Filter(i,j)

if $\exists i\in[k+1 .. n]$: $D_i = \emptyset$

then it must backtrack.



$i = 4,\ j = 3$

$i = 5,\ j = 4$

$i = 6,\ j = 4$

$i = 6,\ j = 5$

$D_6 = \emptyset$

Look Forward once more

***LF* algorithm**:

BT1

\+

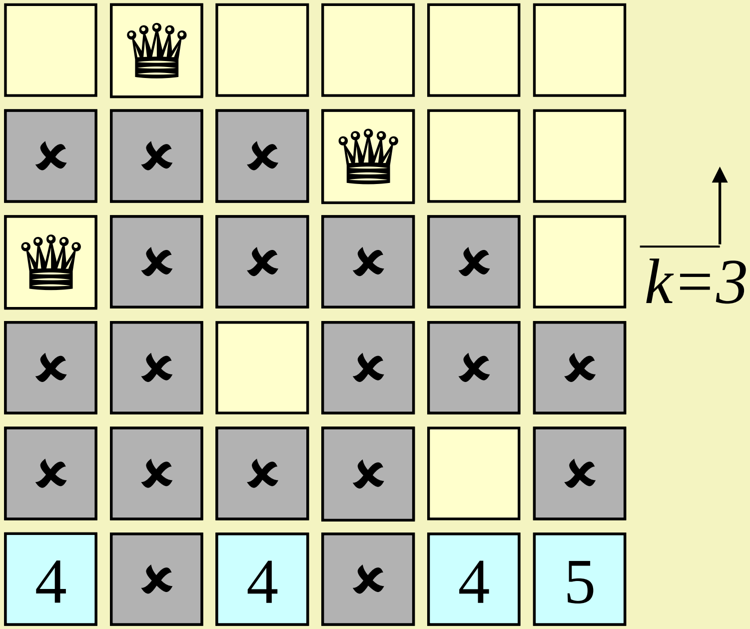
for i=k+1 .. n **loop**

for j=k+1 .. n **and** $i \neq j$ **loop**

Filter(i,j)

if $\exists i \in [k+1 .. n]: D_i = \emptyset$

then it must backtrack.



$i = 6,\ j = 4$

$i = 6,\ j = 5$

$D_6 = \emptyset$

AC1

***AC1* algorithm**:

BT1

\+

repeat

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

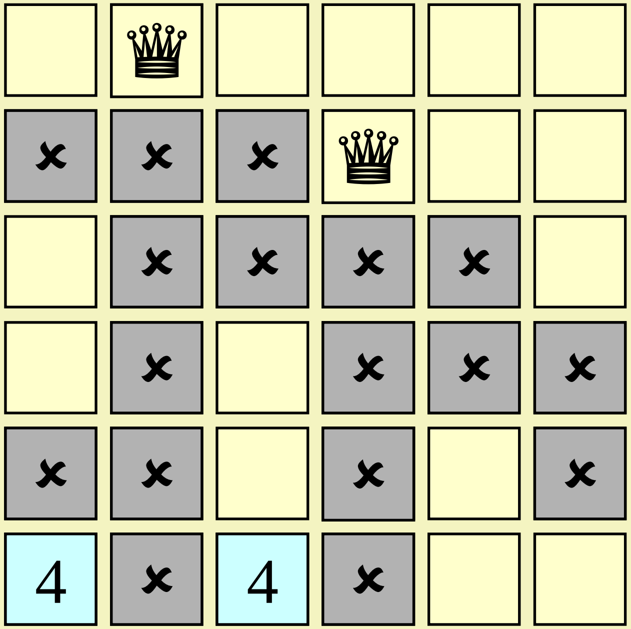
for $j=k+1$ .. n **and** $i \neq j$ **loop**

Filter(i,j)

until *there was no elimination*

if $\exists i \in [k+1 .. n]$: $D_i = \emptyset$

then it must backtrack.



$k=2$

1. turn $i = 6, \; j = 4$

AC1

***AC1* algorithm**:

BT1

+

repeat

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

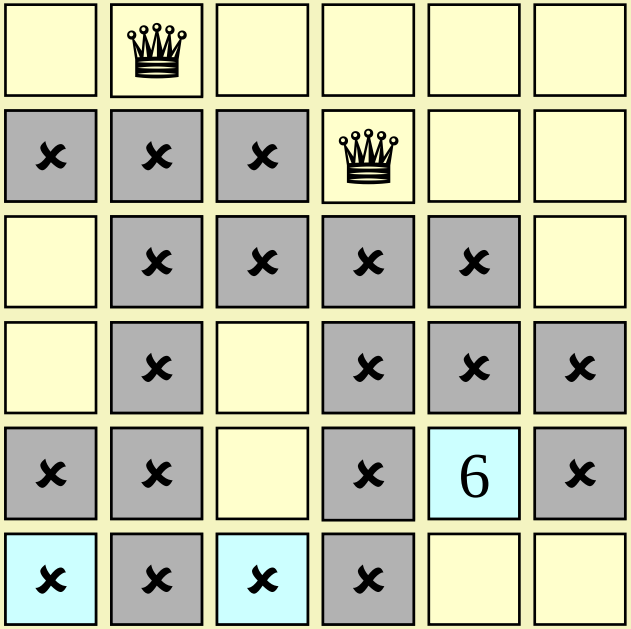
for $j=k+1$ .. n **and** $i \neq j$ **loop**

Filter(i,j)

until *there was no elimination*

if $\exists i \in [k+1 .. n]$: $D_i = \emptyset$

then it must backtrack.



$k=2$

2. turn $i = 5,\ j = 6$

AC1

***AC1* algorithm**:

BT1

+

repeat

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

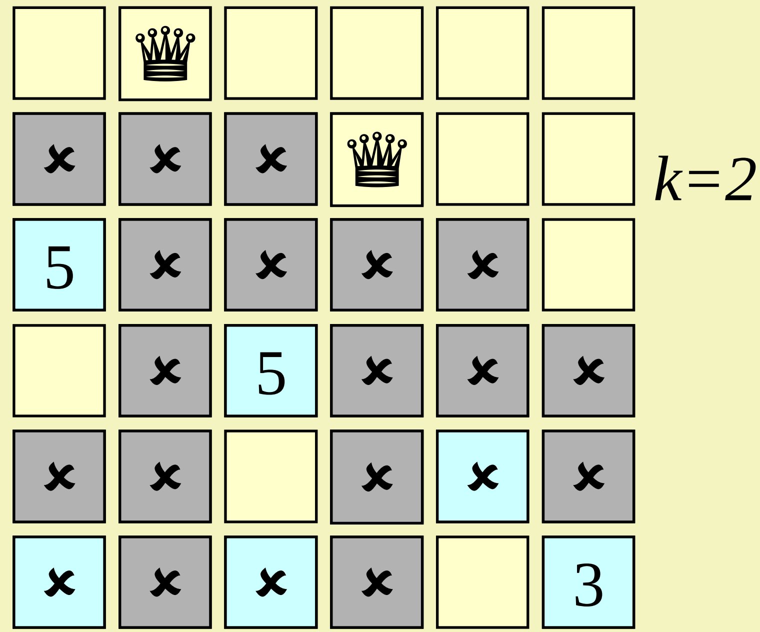
for $j=k+1$ .. n **and** $i\neq j$ **loop**

Filter(i,j)

until *there was no elimination*

if $\exists i\in[k+1..\ n]$: $D_i = \emptyset$

then it must backtrack.



3. turn

$i = 3,\ j = 5$

$i = 4,\ j = 5$

$i = 6,\ j = 3$

AC1

***AC1* algorithm**:

BT1

+

repeat

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

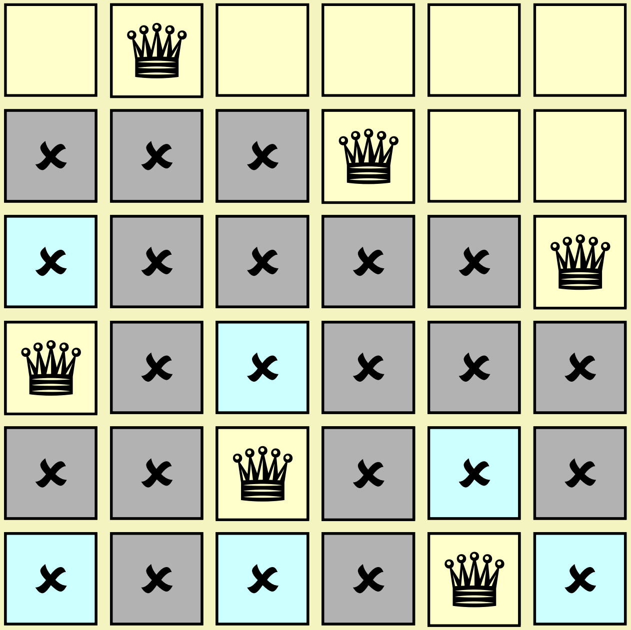
for $j=k+1$ .. n **and** $i \neq j$ **loop**

Filter(i,j)

until *there was no elimination*

if $\exists i \in [k+1 .. n]$: $D_i = \emptyset$

then it must backtrack.



4. turn

A new model of n-queens problem

- In the previous slides the model of the *n*-queens problem has been reformulated:
  - Let us consider the domains $D_1 , \ldots , D_n$ (where D_i contains the possible free squares in the i^{th} row, and initially $D_i = \{1 \ldots n\}$).
  - Find the placement $(x_1 ,\ldots, x_n) \in D_1 \times \ldots \times D_n$ (where x_i is the position of the i^{th} queen in the i^{th} row) so that
  - the placement does not contain attacks, i.e. it satisfies each following constraint (C_{ij}) for the pair (x_i,x_j) :

$$C_{ij}(x_i,x_j) \equiv (x_i \neq x_j \wedge |x_i - x_j| \neq |i-j|)$$

Binary constraint satisfaction model

- Find the n-tuples $(x_1 ,\ldots, x_n) \in D_1 \times \ldots \times D_n$ (D_i is finite) that satisfies some given binary constraints $C_{ij} \subseteq D_i \times D_j$.

  Many problems can be modeled in this way.

  1. Coupling problem: Find a good horse for each horse archer (HA)
     - $D_i=\{1,\ldots,m\}$ contains the possible horses of the i^{th} HA ($i=1..n$)
     - $x_i \in D_i$ is the candidate horse of the i^{th} HA ($i=1..n$)
     - for all i,j: $C_{ij}(x_i,x_j) \equiv x_i \neq x_j$ (a horse can belong only to one HA)
  2. Coloring problem: Color the vertices of a simple finite undirected graph so that no two adjacent vertices share the same color:
     - $D_i=\{1,\ldots,m\}$ contains the possible colors of the i^{th} vertex ($i=1..n$)
     - $x_i \in D_i$ is the candidate color of the i^{th} vertex ($i=1..n$)
     - for all edges (i, j): $C_{ij}(x_i,x_j) \equiv x_i \neq x_j$ (colors are altered)

Model dependent control strategies

- The earlier cutting methods can be redefined using the constraints of the constraint satisfaction model:
  - *Remove*(i,k) : $D_i := D_i - \{e \in D_i \mid \neg C_{ik}(e, x_k)\}$
  - *Filter*(i,j) : $D_i := D_i - \{e \in D_i \mid \forall f \in D_j : \neg C_{ij}(e, f)\}$
- It can be observed that these definitions are independent of the meaning of the constraints. Thus these methods are model dependent cutting strategies (not heuristics).
- Model dependent ordinal strategies are also useful:
  - Prefer the unfilled variable to be filled which has the smallest domain.
  - Fill two variables immediately one after another if there exists a constraint between them.

Second version: *BT2*

- 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 procedure
  - Starting: *solution := BT2*(*<start>*)

BT2

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

Recursive procedure *BT2*(*path* : N^*) **return** (A^*; *fail*)
current := *last_node*(*path*)
if *goal*(*current*) **then** **return**(*nil*)
if *length*(*path*) $\geq$ *limit* **then** **return**(*fail*)
if *current* $\in$ *remain*(*path*) **then** **return**(*fail*)
for $\forall$*new* $\in$ Γ(*current*) $-$ π(*current*) **loop**
solution := *BT2*(*concat*(*path*, *new*))
if *solution* $\neq$ *fail* **then**
return(*concat*((*curent*, *new*), *solution*) **endif**
endloop
return(*fail*)
end

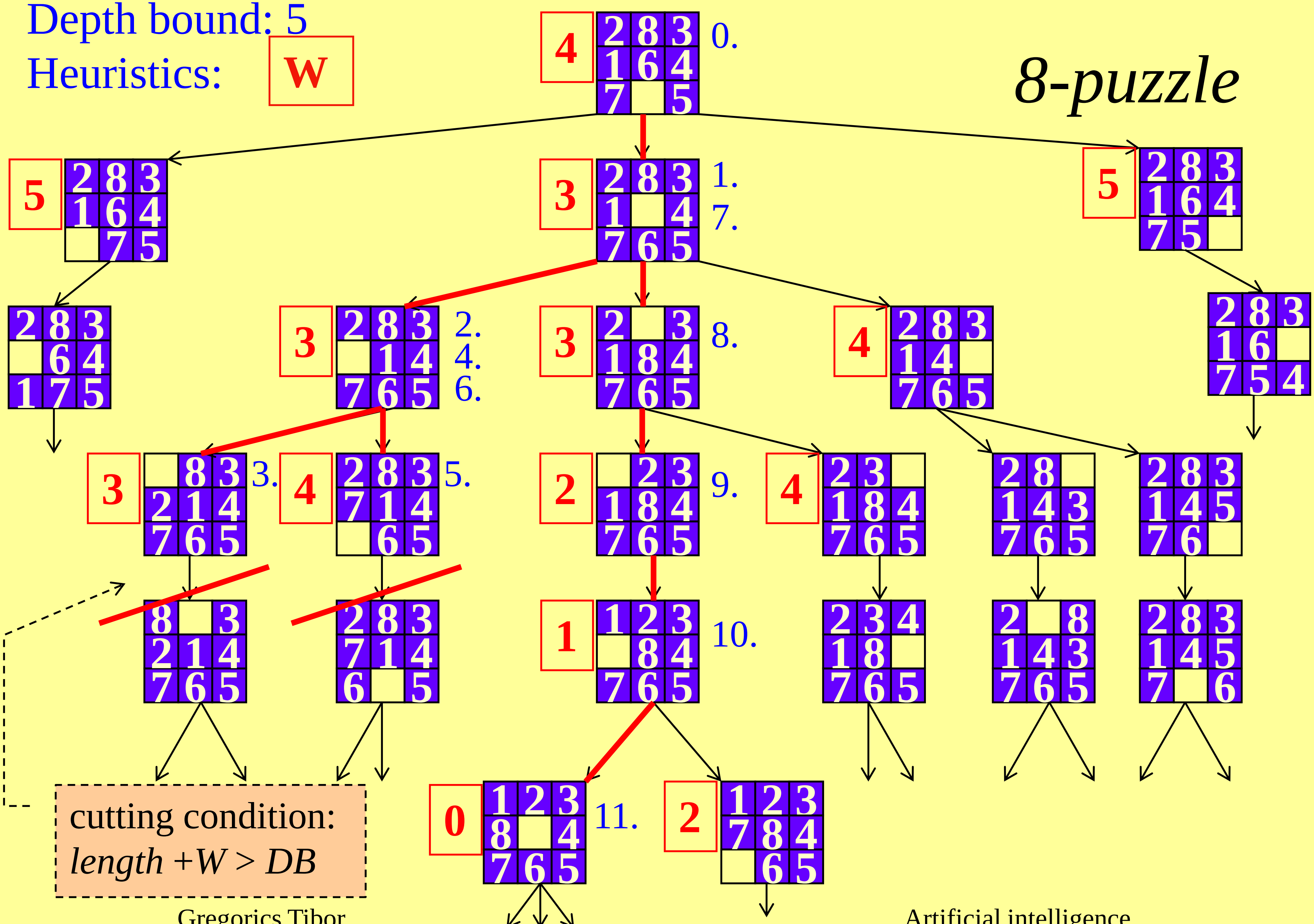
Role of the depth bound

- *BT2* can find the solution path which length is less than or equal to the depth bound.
- The checking of cycles might be ignored because the checking of the depth bound alone ensures the outcome of *BT2*.
  - This simplification can improve the efficiency if there are no short cycles in the representation graph (except the 2-length cycles since they can be eliminated easily by examining the parent node of the current one).
  - In this case it is enough to give the recursive procedure the current node, the length of the current path and the parent of the current node instead of the whole current path.

8-puzzle

Depth bound: 5

Heuristics: W

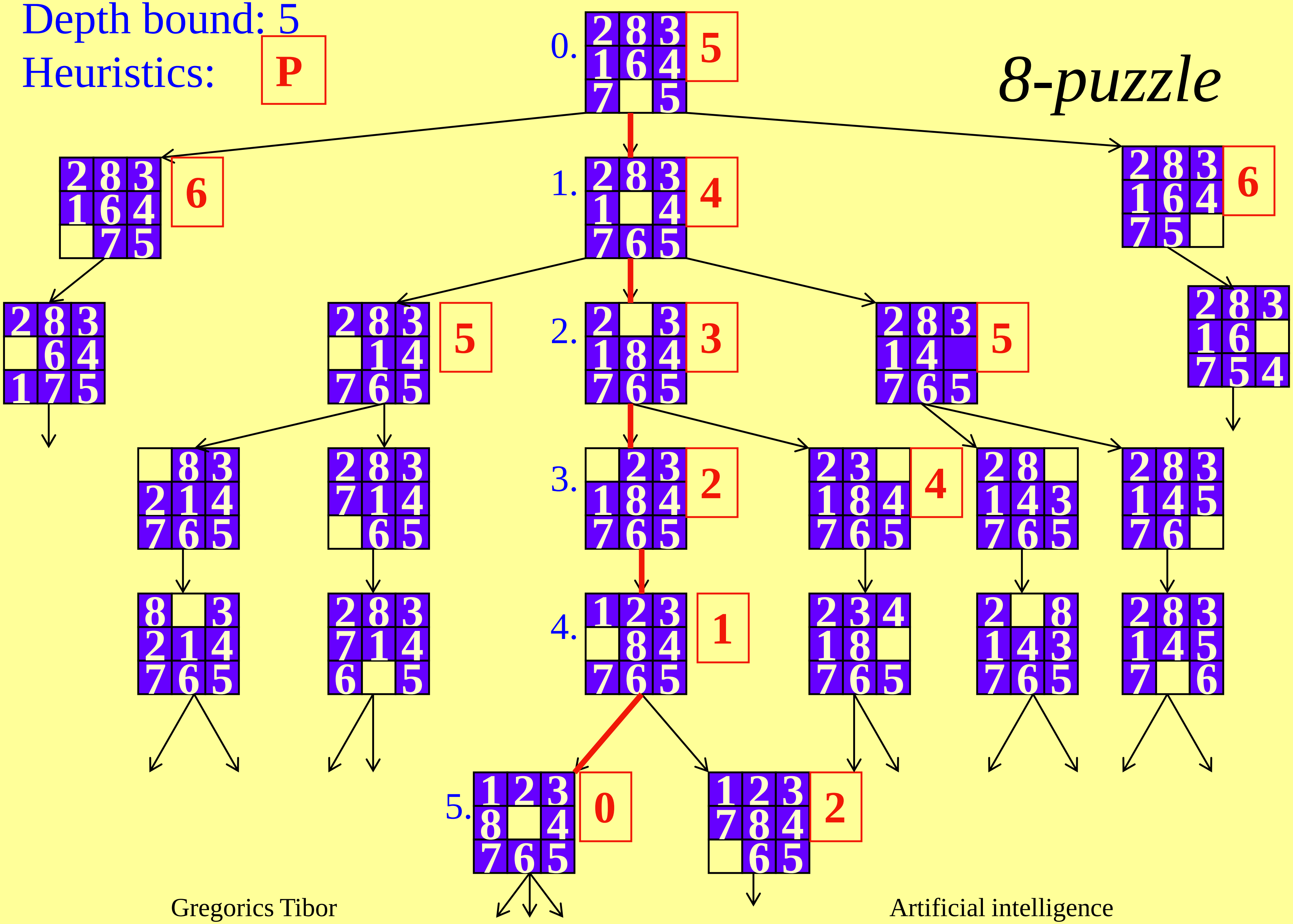


cutting condition:
length +W > DB

8-puzzle

Depth bound: 5

Heuristics: P



Conclusions

- Advantages
  - always terminates, and finds solution (inside the depth bound)
  - 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