Fifth Chapter

Back on Recursion

- Avoiding Recursion
  - Non-Recursive Form of Tail Recursion
  - Transformation to Tail Recursion
  - Generic Algorithm Using a Stack
- Back-tracking
- Conclusion on Recursion

Combinatorial Search and Optimization

Large class of Problems with similar algorithmic approach

- Solutions are really numerous; A set of constraints make some solution invalids
- Combinatorial Search $\rightsquigarrow$ look for any valid solution
  Combinatorial Optimization $\rightsquigarrow$ look for the solution maximizing a function

Examples

- Open the lock: Find the right 4-digits combination out of 10000
- Knapsac: Ali-Baba searches object set fitting in bag maximizing the value
- Minimum Spanning Tree of a given graph
- Traveling Salesman: visit n cities in order minimizing the total distance

First Resolution Approach: Exhaustive Search

- Study *every* solutions
  $\rightsquigarrow$ Test all lock combinations
  $\rightsquigarrow$ Enumerate all possible knapsack contents + get max value
- This often reveals to be exponential and thus infeasible

Better Approach?

Guessing the right number can become difficult that way

- 0001 $\rightsquigarrow$ *no*; 0002 $\rightsquigarrow$ *no*; 0003 $\rightsquigarrow$ *no*; 0004 $\rightsquigarrow$ *no*; 0005 $\rightsquigarrow$ *no*; Booooring
- Let's more information: length of correct suffix instead of yes/no answers
  0001 $\rightsquigarrow$ 0; 0002 $\rightsquigarrow$ 0; 0004 $\rightsquigarrow$ 1; 0024 $\rightsquigarrow$ 2; 0424 $\rightsquigarrow$ 3; 5424 $\rightsquigarrow$ 4, *bingo*

Better Approach?

Guessing the right number can become difficult that way

- 0001 $\rightsquigarrow$ *no*; 0002 $\rightsquigarrow$ *no*; 0003 $\rightsquigarrow$ *no*; 0004 $\rightsquigarrow$ *no*; 0005 $\rightsquigarrow$ *no*; Booooring
- Let's more information: length of correct suffix instead of yes/no answers
  0001 $\rightsquigarrow$ 0; 0002 $\rightsquigarrow$ 0; 0004 $\rightsquigarrow$ 1; 0024 $\rightsquigarrow$ 2; 0424 $\rightsquigarrow$ 3; 5424 $\rightsquigarrow$ 4, *bingo*

This leads to a much more efficient algorithm:

- Guess each position by testing every digit in that pos until response increases

Better Approach?

Guessing the right number can become difficult that way

- 0001 $\rightsquigarrow$ *no*; 0002 $\rightsquigarrow$ *no*; 0003 $\rightsquigarrow$ *no*; 0004 $\rightsquigarrow$ *no*; 0005 $\rightsquigarrow$ *no*; Booooring
- Let's more information: length of correct suffix instead of yes/no answers
  0001 $\rightsquigarrow$ 0; 0002 $\rightsquigarrow$ 0; 0004 $\rightsquigarrow$ 1; 0024 $\rightsquigarrow$ 2; 0424 $\rightsquigarrow$ 3; 5424 $\rightsquigarrow$ 4, *bingo*

This leads to a much more efficient algorithm:

- Guess each position by testing every digit in that pos until response increases
- That's even easy to write by mixing recursion with a for loop:

```
search(current,pos,len): // initial values: search({0,0,0,0}, 0, 0)
  for n ∈ [0; 9] do
    put n into current at position pos
    if try(current) > len then search(current,pos+1, try(current))
                          else // no luck. Let's test the next value of n
```

Better Approach?

Guessing the right number can become difficult that way

- 0001 $\leadsto$ *no*; 0002 $\leadsto$ *no*; 0003 $\leadsto$ *no*; 0004 $\leadsto$ *no*; 0005 $\leadsto$ *no*; Booooring
- Let's more information: length of correct suffix instead of yes/no answers
  0001 $\leadsto$ 0; 0002 $\leadsto$ 0; 0004 $\leadsto$ 1; 0024 $\leadsto$ 2; 0424 $\leadsto$ 3; 5424 $\leadsto$ 4, *bingo*

This leads to a much more efficient algorithm:

- Guess each position by testing every digit in that pos until response increases
- That's even easy to write by mixing recursion with a for loop:

```
search(current,pos,len): // initial values: search({0,0,0,0}, 0, 0)
  for n ∈ [0;9] do
    put n into current at position pos
    if try(current) > len then search(current,pos+1, try(current))
                           else // no luck. Let's test the next value of n
```

This is Backtracking

- Tentative choices + cut branches leading to invalid solutions (backtrack)
- Restrict study to valid solutions only $\leadsto$ if bag is full, don't stuff something else
- Also factorize computations $\leadsto$ only sum up once the N first objects' value

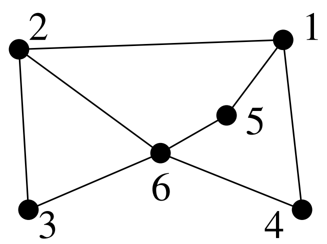
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



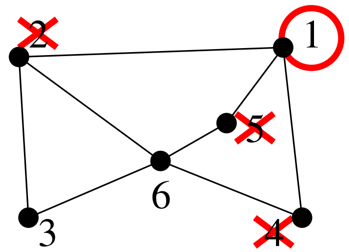
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$

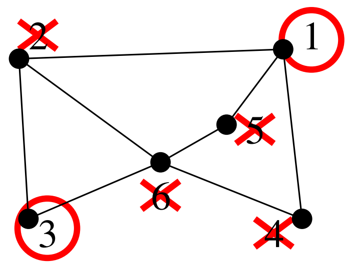
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1, 3\}$

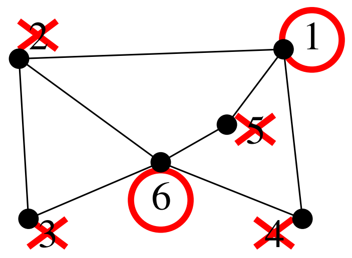
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$.

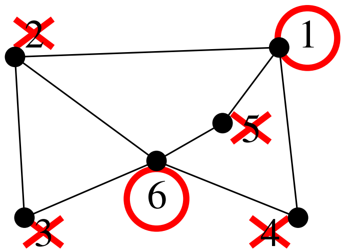
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.

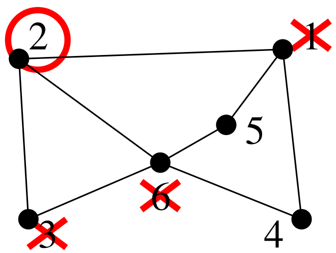
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$

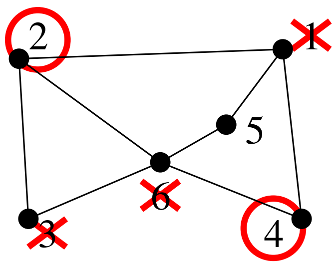
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1, 3\}$. Stuck. Remove 3. $\{1, 6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$, $\{2, 4\}$

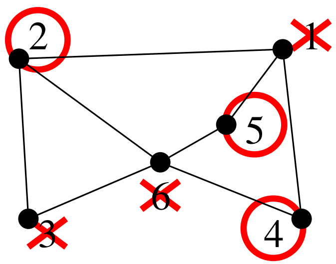
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$, $\{2,4\}$, $\{2,4,5\}$

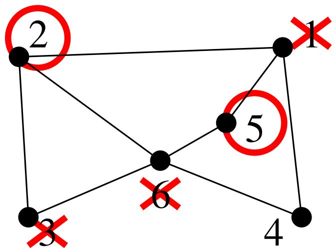
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$, $\{2,4\}$, $\{2,4,5\}$ (Stuck; remove 5 then 4) $\{2,5\}$

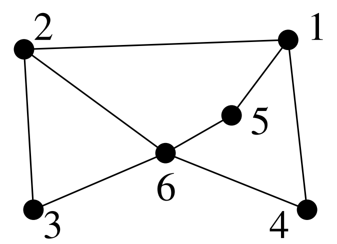
Back-tracking

Characterization

- Search for a solution in given space:
  - Choice of a (valid) partial solution
  - Recursive call for the rest of the solution
- Some built solutions are dead-ends
  (no way to build a valid solution with choices made so far)
- Backtracking then mandatory for *another* choice
- General Schema: **Recursive Call within an Iteration**

First example: Independent Sets

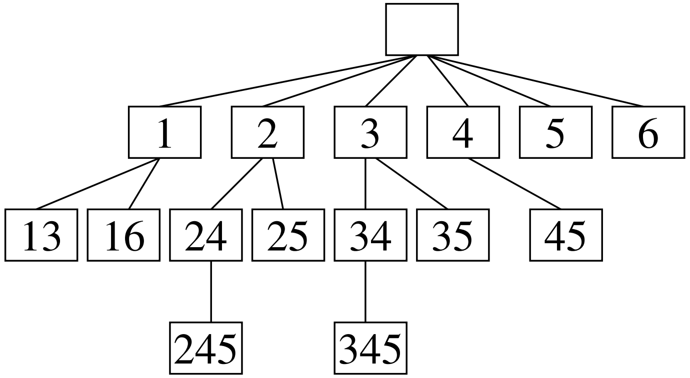
- Sets of vertices not interconnected by any graph edge
- Init: set of 1 element; Algo: increase size as much as possible then backtrack



- $\{1\}$, $\{1,3\}$. Stuck. Remove 3. $\{1,6\}$. Stuck.
  Removing 6 is not enough, remove everything.
- $\{2\}$, $\{2,4\}$, $\{2,4,5\}$ (Stuck; remove 5 then 4) $\{2,5\}$
- $\{3\}$, $\{3,4\}$, $\{3,4,5\}$, $\{3,5\}$; $\{4\}$, $\{4,5\}$; $\{5\}$, $\{6\}$

Algorithm Computation Time

Solution Tree of this Algorithm



- Traverse every nodes (without building it explicitly)
- Amount of algorithm steps = amount of solutions
- Let n be amount of nodes

Amount of solutions for a given graph?

- Empty Graph (no edge) $\rightsquigarrow I_n = 2^n$ independent sets
- Full Graph (every edges) $\rightsquigarrow I_n = n + 1$ independent sets
- On average $\rightsquigarrow I_n = \sum_{k=0}^{n} \binom{k}{n} 2^{-k(k-1)/2}$

n	2	3	4	5	10	15	20	30	40
I_n	3,5	5,6	8,5	12,3	52	149,8	350,6	1342,5	3862,9
2^n	4	8	16	32	1024	32768	1048576	1073741824	1099511627776

- Backtracking algorithm traverses I_n nodes on average
- An exhaustive search traverses 2^n nodes

Other example: n queens puzzle

Goal:

- Put n queens on a $n \times n$ board so than none of them can capture any other

Algorithm:

- Put a queen on first line
  There is n choices, any implying constraints for the following
- Recursive call for next line

Other example: n queens puzzle

Goal:

- Put n queens on a $n \times n$ board so than none of them can capture any other

Algorithm:

- Put a queen on first line
  There is n choices, any implying constraints for the following
- Recursive call for next line

Pseudo-code `put_queens(int line, board)`

If *line* $>$ *line_count*, return `board` (success)

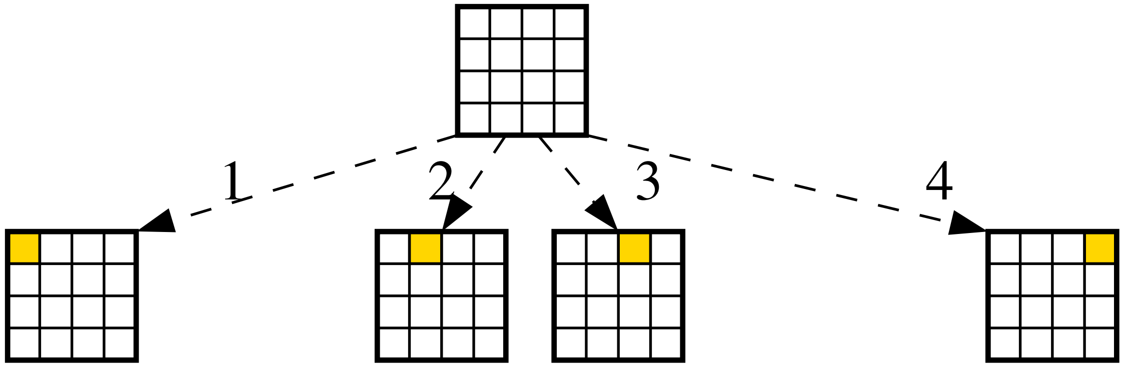
$\forall$ *cell* $\in$ *line*,

- Put a queen at position *cell* $\times$ *line* of `board`
- If conflict, then return (stopping descent – failure)
- (else) call `put_queens(ligne+1, board` $\cap$ $\{cell, line\}$)

$\Rightarrow$ Recursive Call within a Loop

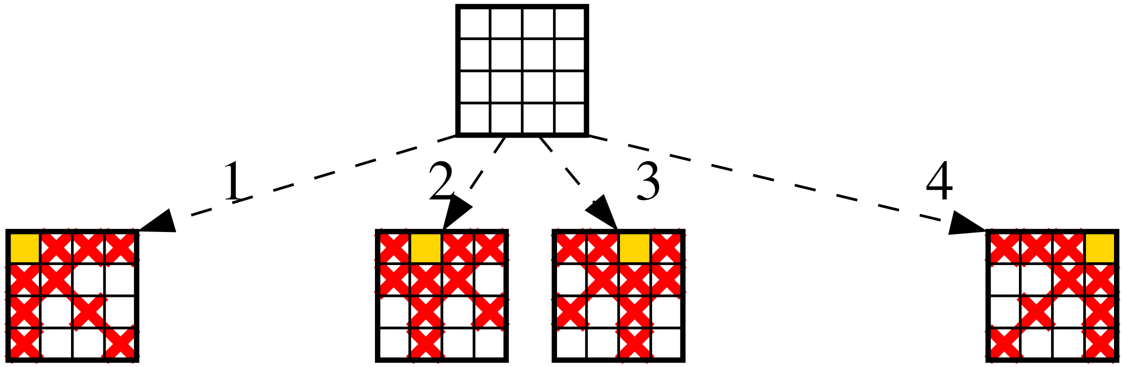
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions



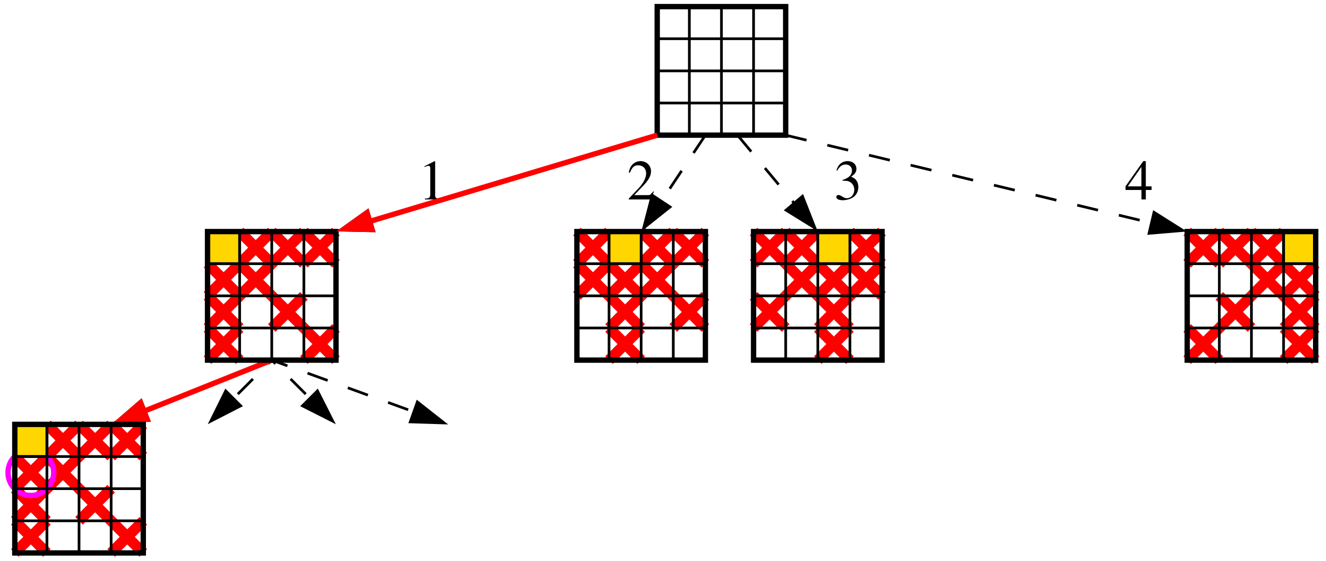
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following



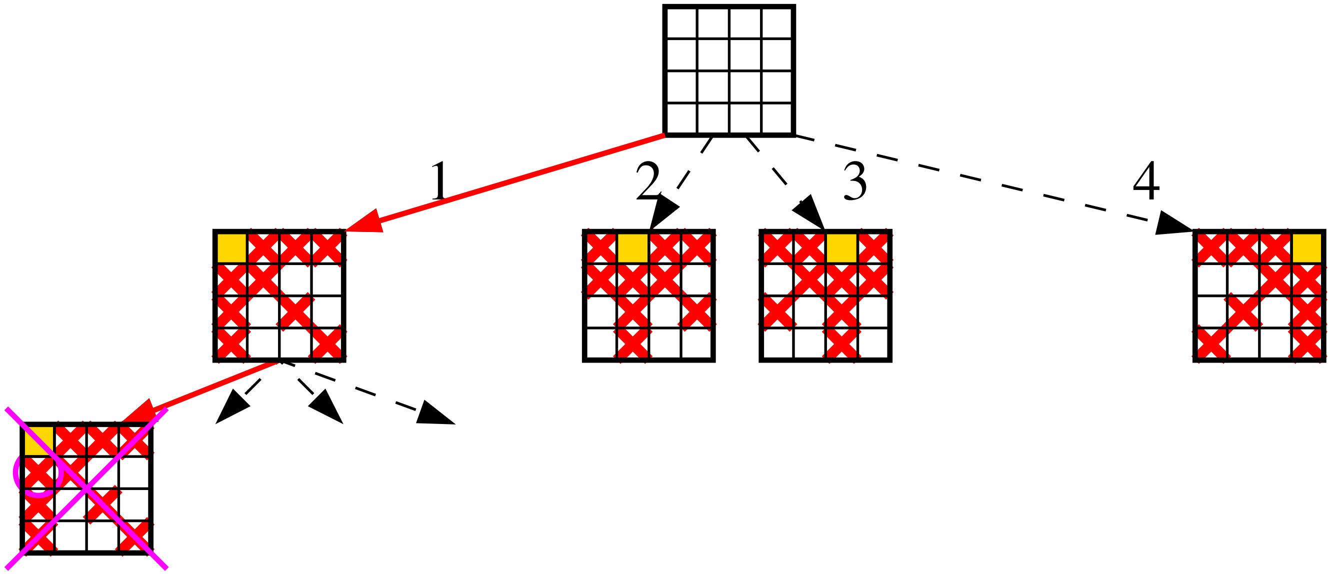
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent



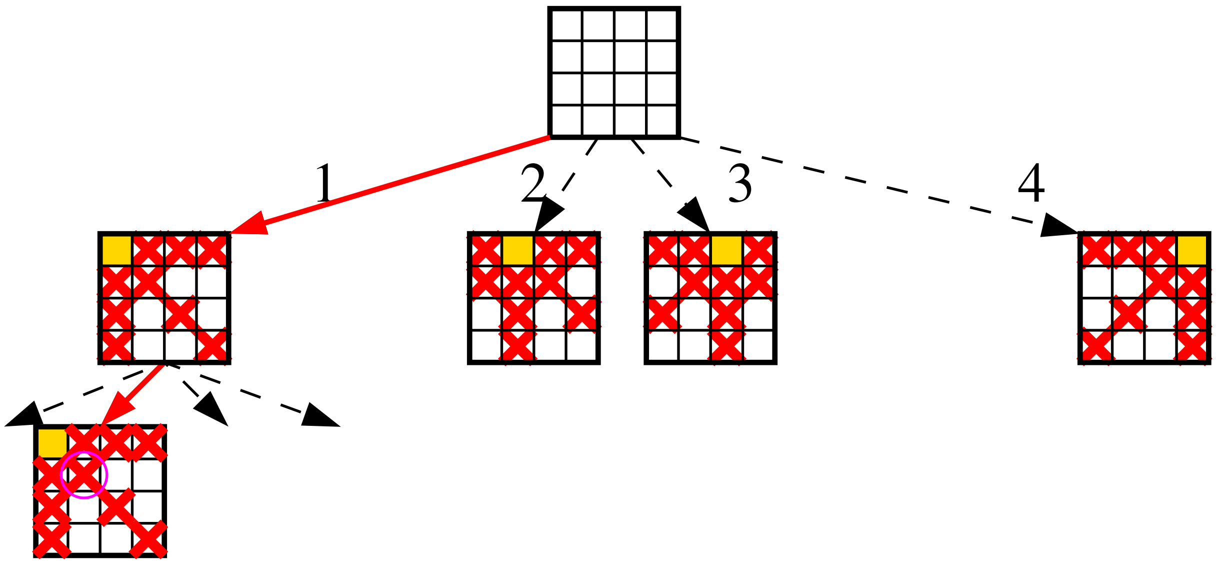
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back



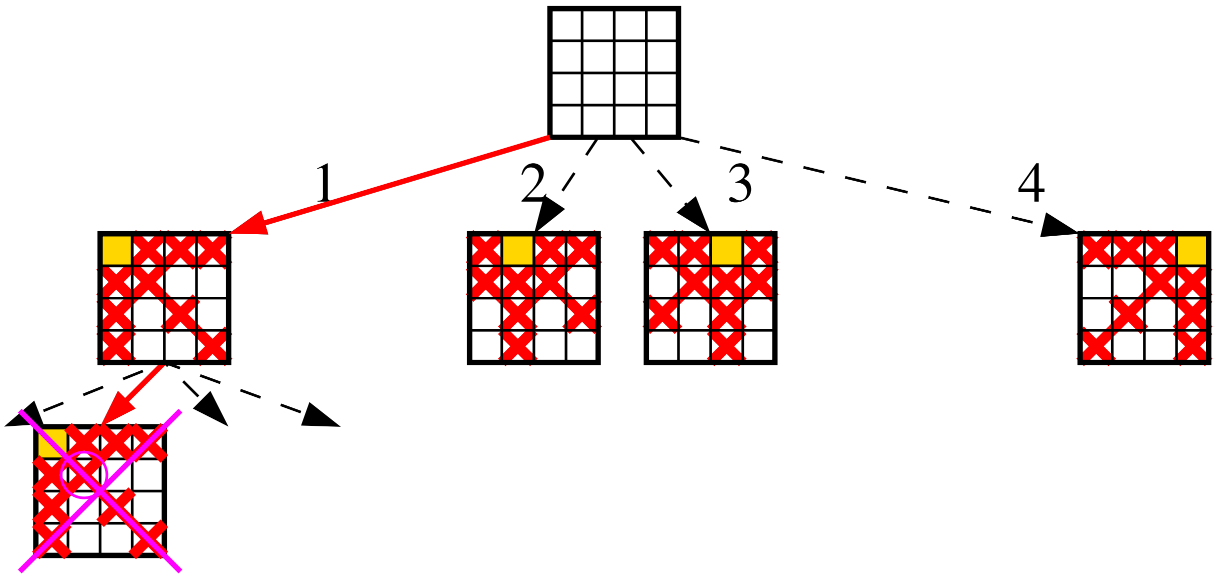
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



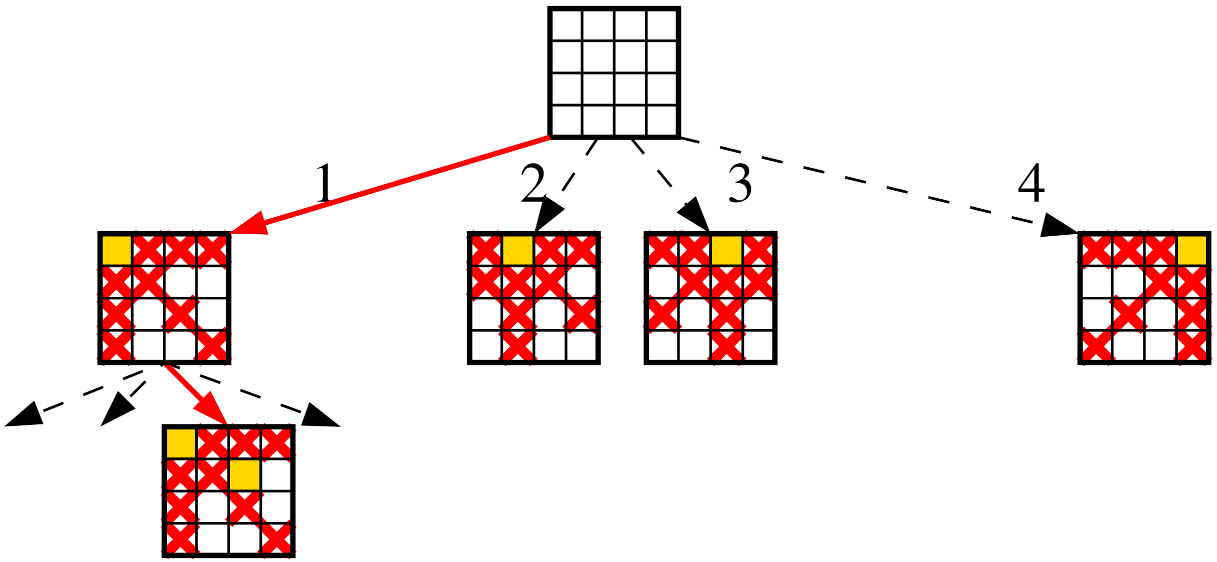
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



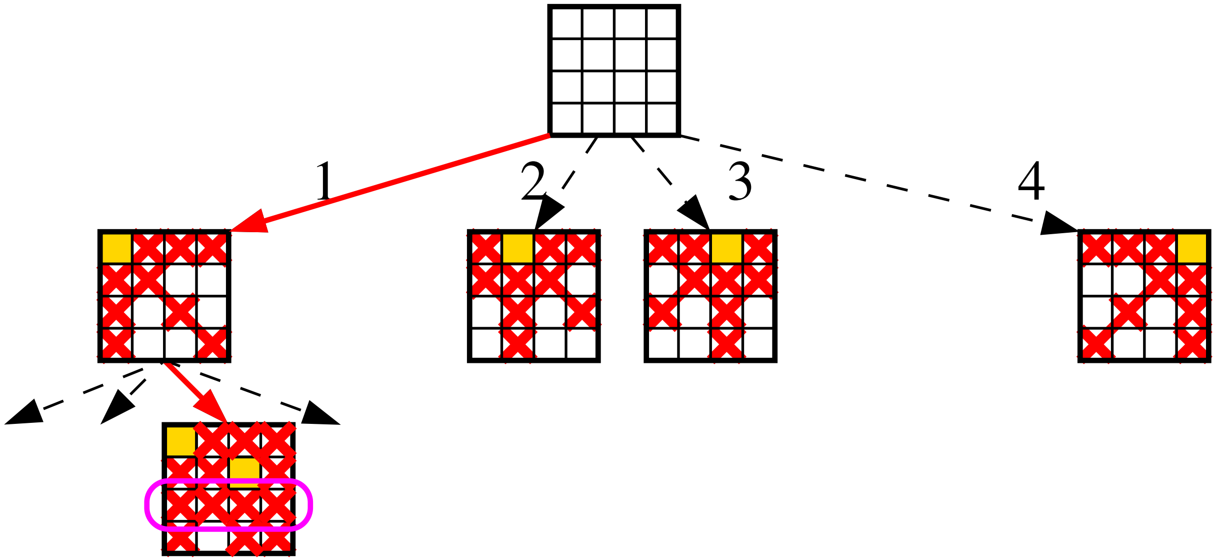
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



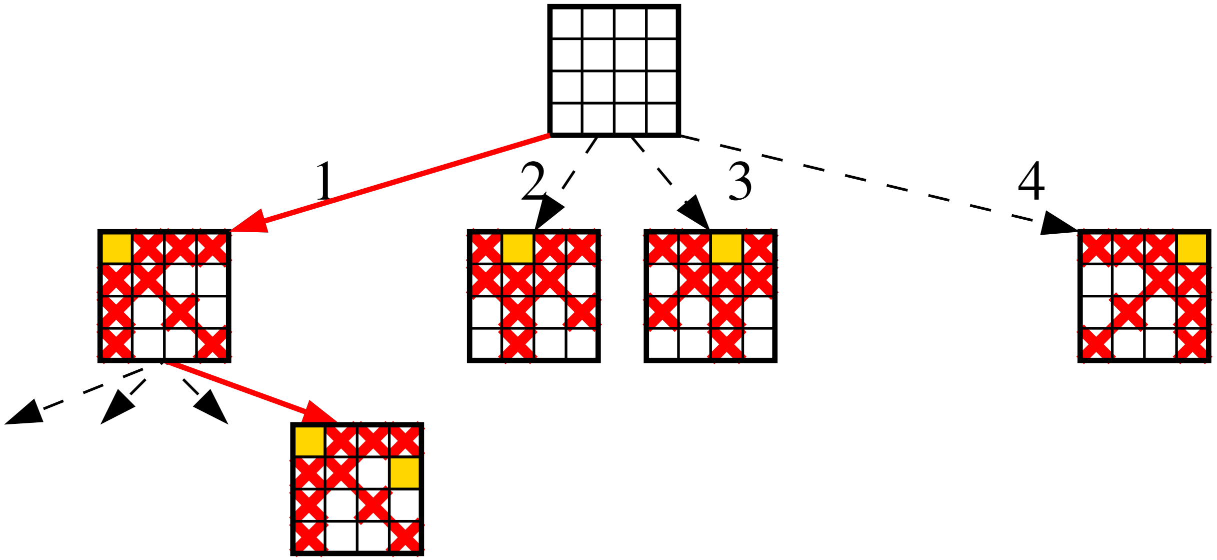
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



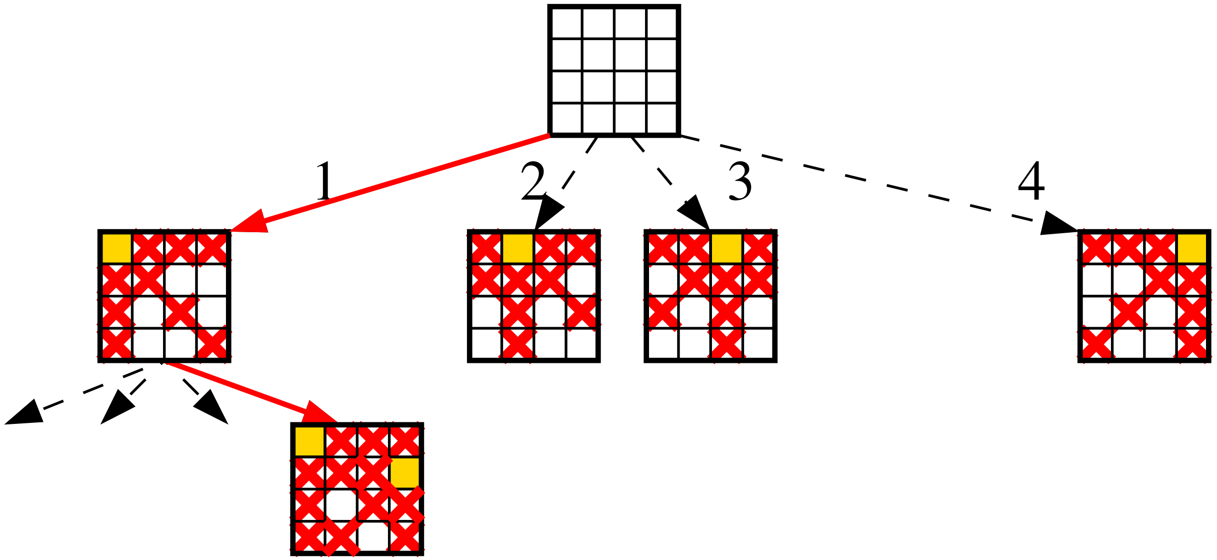
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



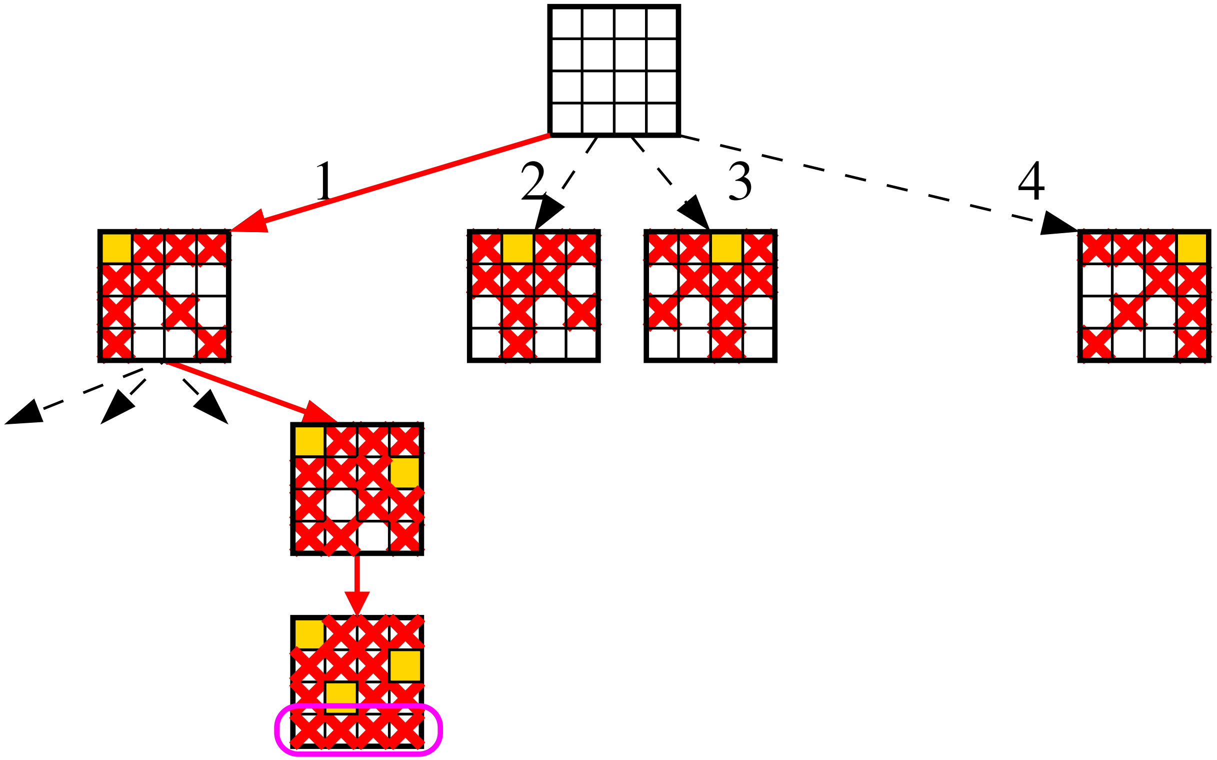
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



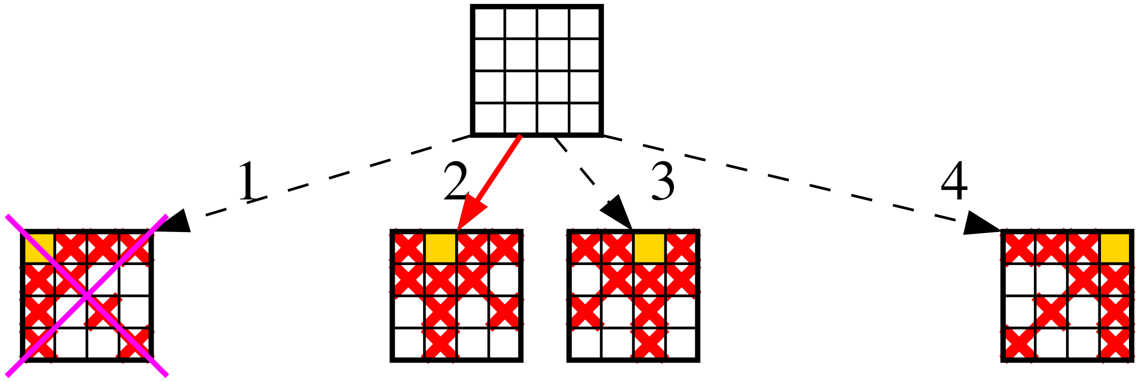
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



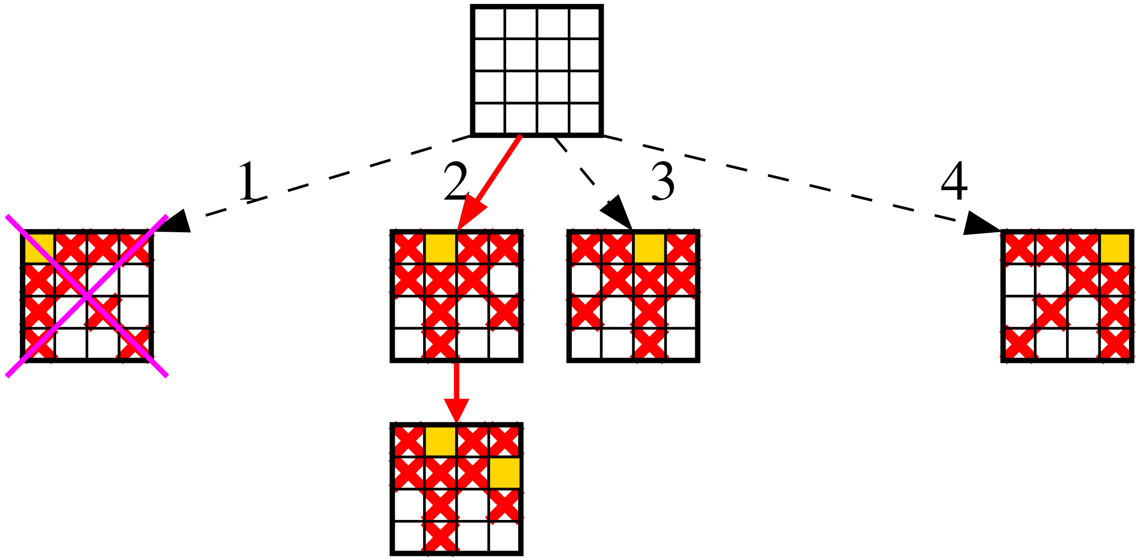
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



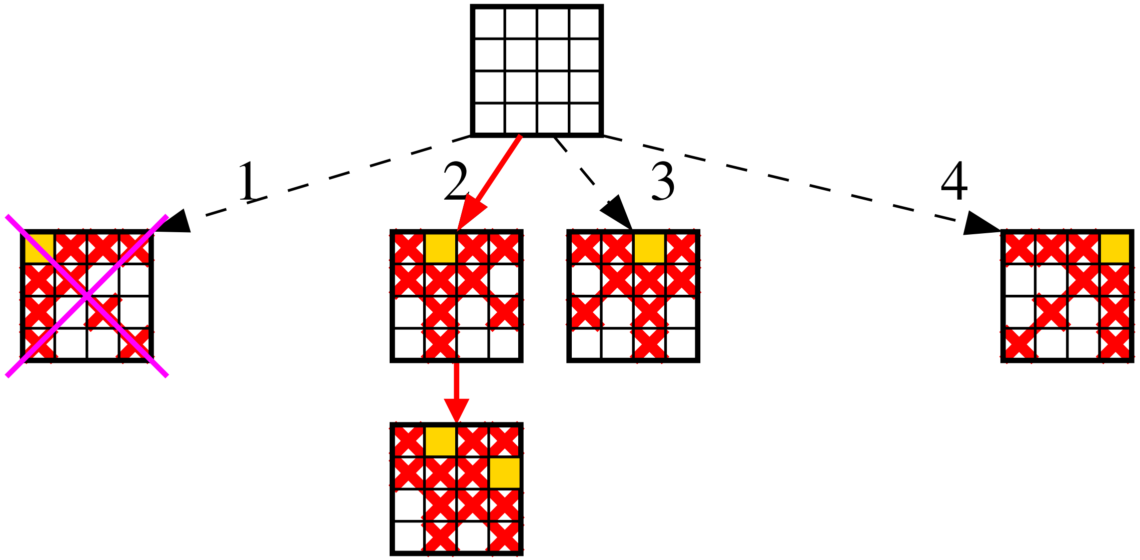
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



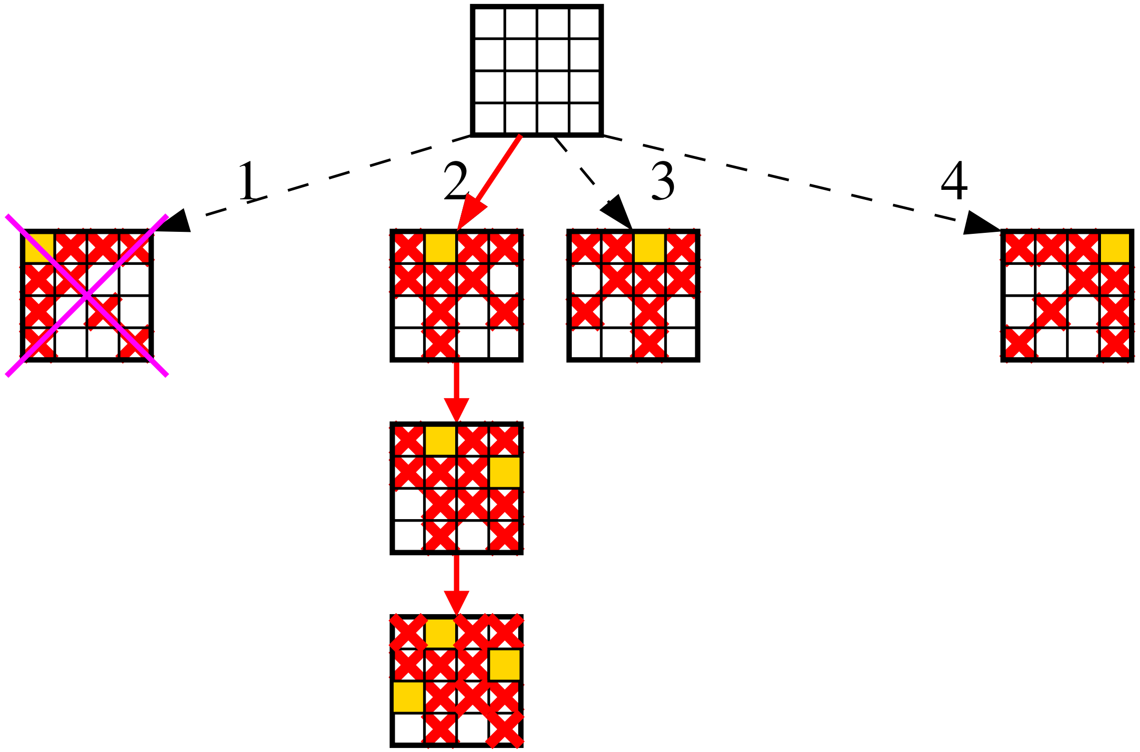
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



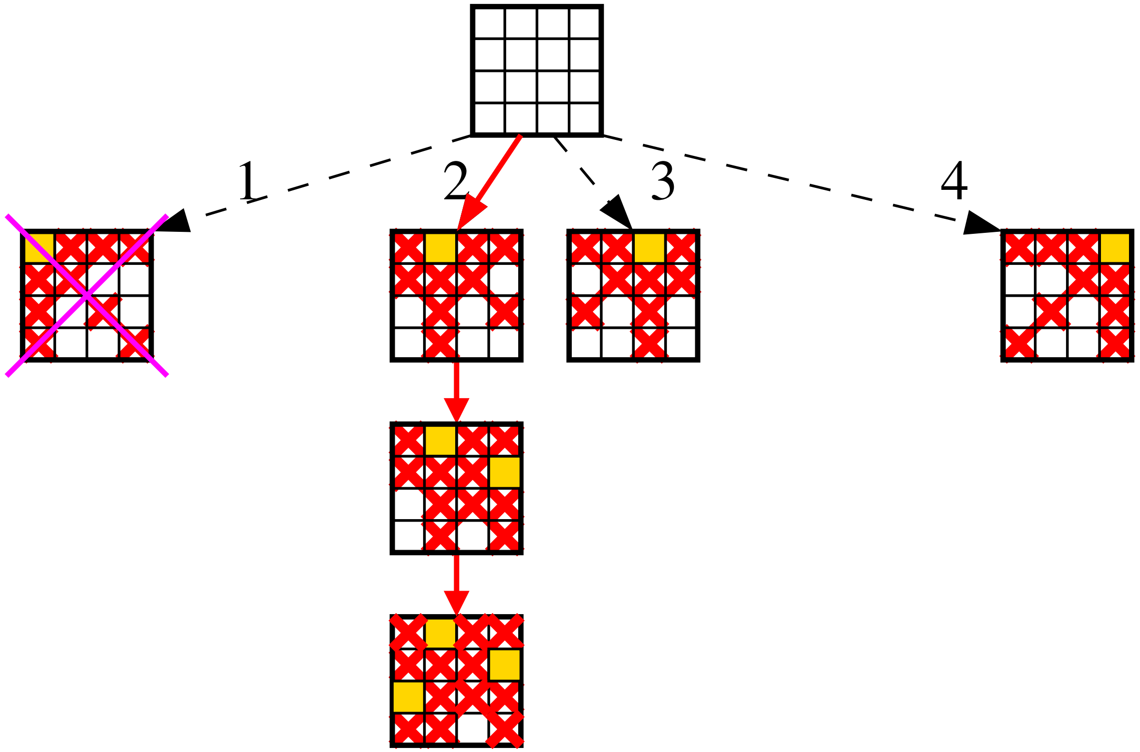
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



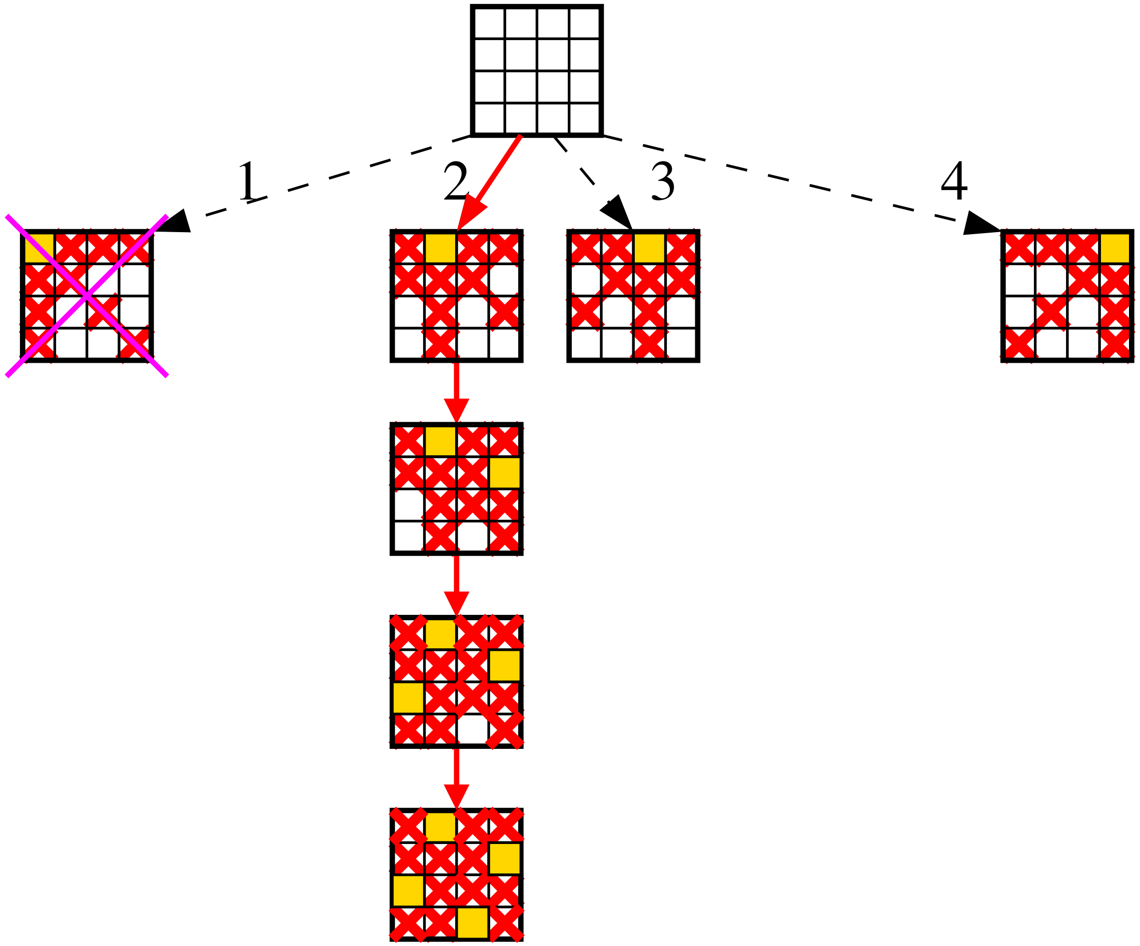
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



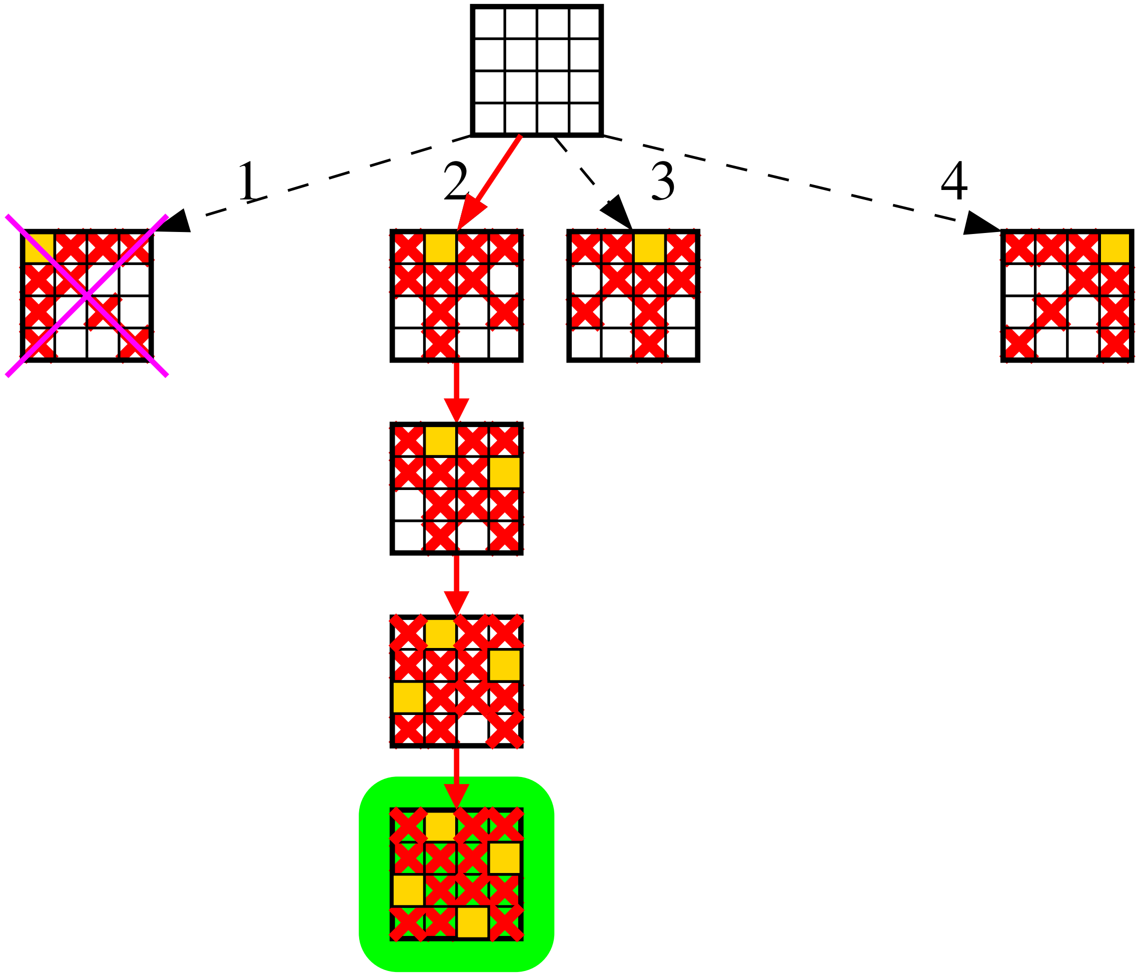
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)



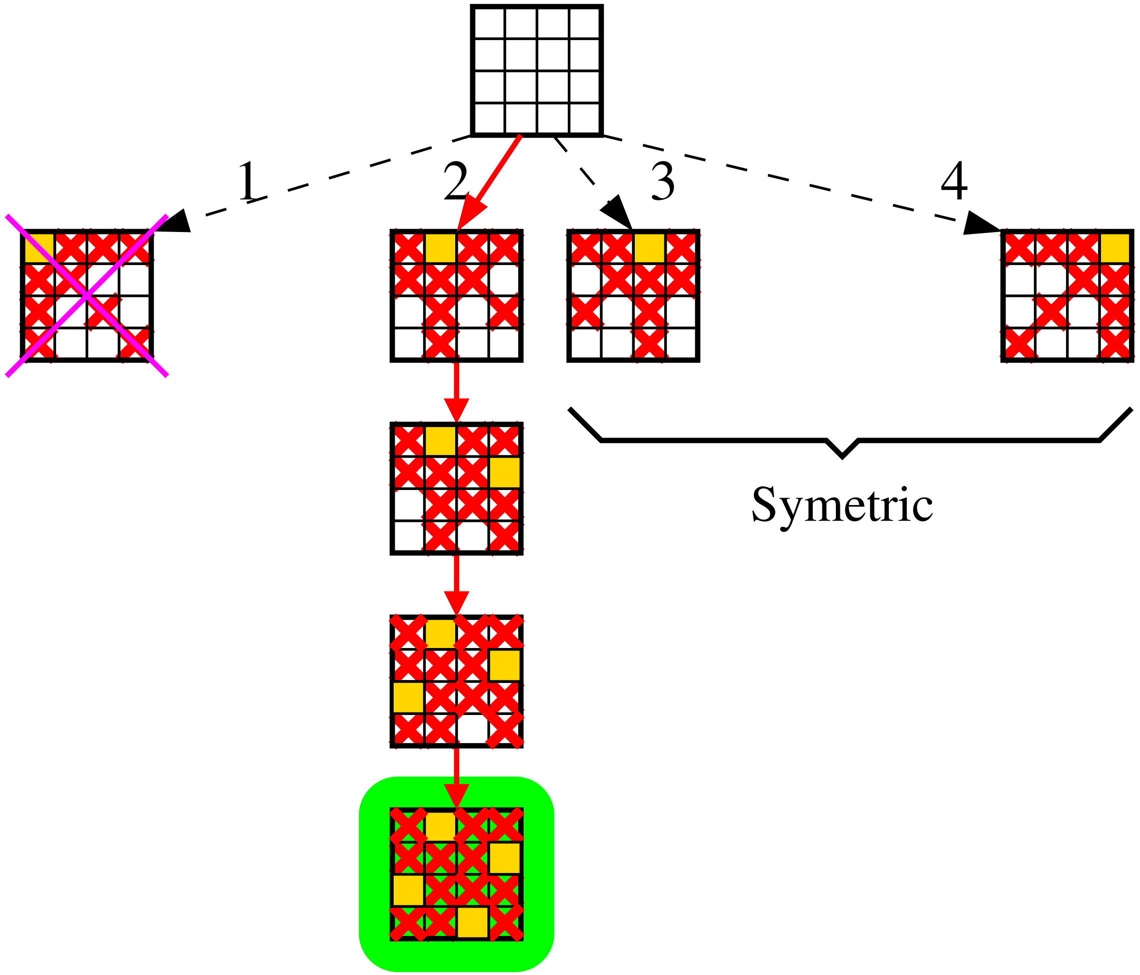
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)
- Until we find a solution (or not)



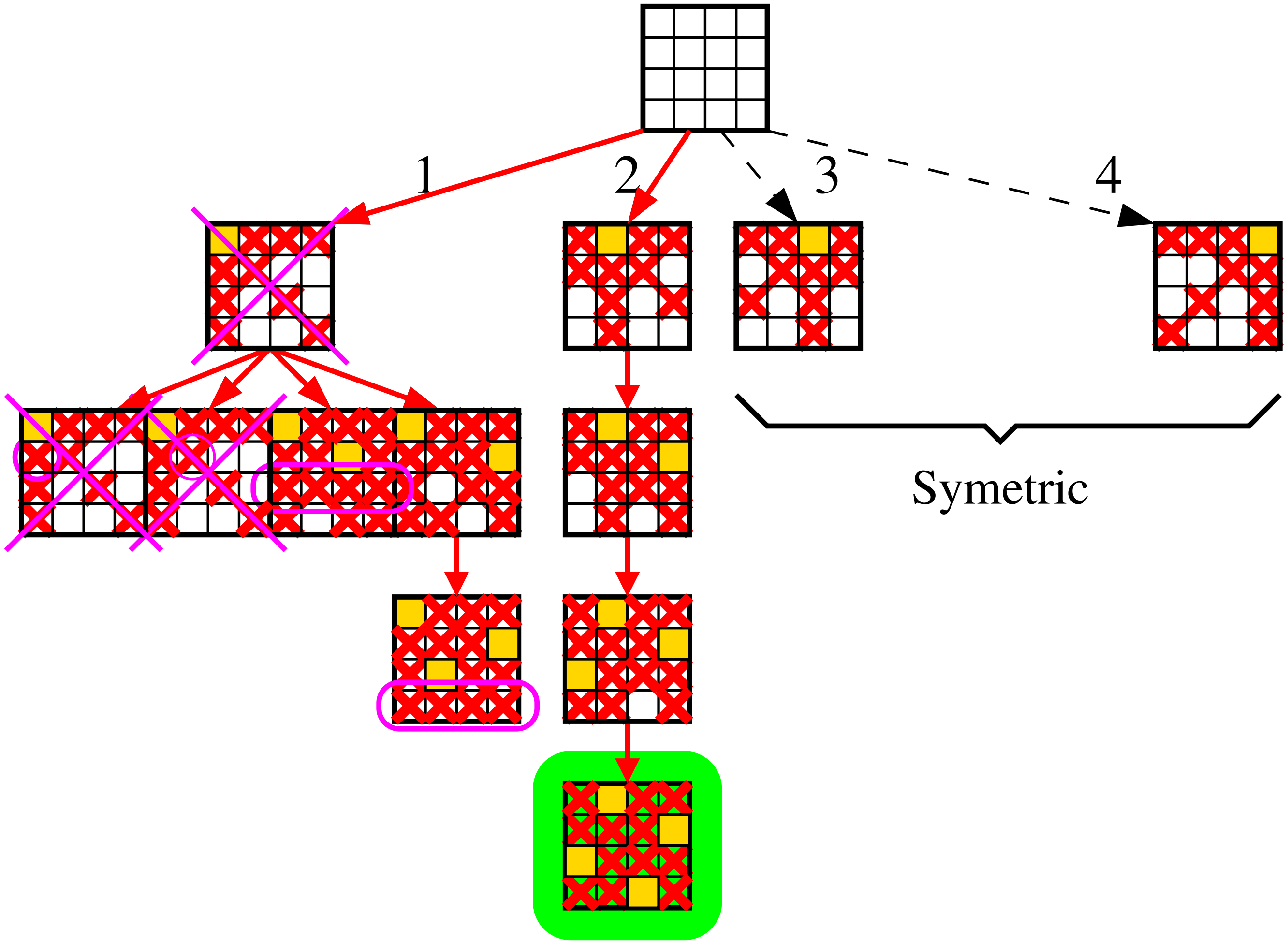
Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)
- Until we find a solution (or not)



Solving the 4 queens puzzle

- At each step of recursion, iterate on differing solutions
- Each choice induces impossibilities for the following
- For each iteration, one descent
- When stuck, climb back (and descent in following iteration)
- Until we find a solution (or not)



Scala implementation of n queens puzzle

```
def Solution(board:Array[Array[Boolean]], line:Int) {
  if (line >= board.length) // Base Case
     return true;

  for (col <- 0 to board.length - 1) {  // loop on possibilities
     if (validPlacement(board, line, col)) {
       putQueen(board, line, col);
       if (Solution(plateau, line + 1)) // Recursive Call
           return true; // Let solution climb back
       removeQueen(board, line, col);
     }
  }
  return false;
}
```

Some Principles on Backtracking

- Study “depth first” of solution tree
- On backtracking, restore state as before last choice
  Trivial here (parameters copied on recursive call), harder in iterative
- Strategy on branch ordering can improve things
- Progressive Construction of boolean function
- If function returns false, there is no solution

Some Principles on Backtracking

- Study "depth first" of solution tree
- On backtracking, restore state as before last choice
  Trivial here (parameters copied on recursive call), harder in iterative
- Strategy on branch ordering can improve things
- Progressive Construction of boolean function
- If function returns false, there is no solution

- Probable Combinatorial Explosion (4^4 boards)
  $\Rightarrow$ Need for heuristics to limit amount of tries

Conclusion on Recursion

Essential Tool for Algorithms

- Recursion in Computer Science, induction in Mathematics
- Recursive Algorithms are frequent because easier to understand ...
  (and thus easier to maintain)
  ...but maybe slightly more difficult to write (that's a practice to get)
- Recursive programs maybe slightly less efficient...
  ...but always possible to transform a code to non-recursive form
  (and compilers do it)
- Classical Functions: Factorial, gcd, Fibonacci, Ackerman, Hanoï, Syracuse, ...
- Sorting Functions: MergeSort and QuickSort are amongst the most used
  (because efficient)
- BackTracking: exhaustive search in space of *valid* solutions
- Data Structure module: several recursive datatypes with associated algorithms
- Recursion is the root of computation since it trades description for time.
  – "Epigrams in Programming", by Alan J. Perlis of Yale University.