Backtracking and Branch-and-Bound Search

Georgy Gimel'farb

(with basic contributions by Michael J. Dinneen)

COMPSCI 369 Computational Science

Learning outcomes:

- Understand that heuristic optimisation strategies must be used when no good exact algorithm is known

Recommended reading:

- R. E. Neapolitan, K. Naimipour: *Foundations of Algorithms Using C++ Pseudocode*. Sudbury, Mass., USA : Jones and Bartlett, 2004

Additional sources:
R. Bartak: Systematic search algorithms: http://ktiml.mff.cuni.cz/ bartak/constraints/backtrack.html
http://www.academic.marist.edu/ jzbv/algorithms/Backtracking.htm
http://www.academic.marist.edu/ jzbv/algorithms/Branch and Bound.htm
http://en.wikipedia.org/wiki/Branch_and_bound
http://www.ifors.ms.unimelb.edu.au/tutorial/knapsack/introduction.html

Systematic Search Through Possible Solutions

Potential solution: an assignment of values to a vector of variables $\mathbf{x} = [x_1, \ldots, x_m]$, such that satisfy given constraints

- m variables $(x_1, x_2, \ldots, x_m)$ with k values each:
  $\rightarrow$ k^m possible assignments
- m variables $(x_1, x_2, \ldots, x_m)$; $x_i \in \mathbb{X}_i = \{\alpha_{i:1}, \ldots, \alpha_{i:K_i}\}$
  $\rightarrow$ $\prod_{i=1}^{m} |\mathbb{X}_i| \equiv \prod_{i=1}^{m} K_i$ possible assignments

Partial solution: a vector $[x_1, \ldots, x_i]$ of $i < m$ elements, which have been assigned

Example: n Queens Problem

Place n non-attacking queens on an $n \times n$ chess board

How quickly can we find a solution? E.g. for $n = 8$

"Brute Force", or "Generate-and-Test" Search

Full exhaustion of, or searching through all the possible assignments of values from $\mathbb{X}_i$ to variables x_i; $i = 1, \ldots, K$

1. Systematically guess (generate) one of the remaining solutions
2. Test whether this solution is correct
3. Terminate if the test is successful

Disadvantage: The inefficiency of generating new assignments with no account of earlier rejected ones

- This method considers $\prod_{i=1}^{m} |\mathbb{X}_i| \equiv \prod_{i=1}^{m} K_i$ combinations (the size of the Cartesian product $\mathbb{X}_1 \times \cdots \times \mathbb{X}_m$ of all the variable domains)
- For better efficiency, the validity of each constraint is to be tested as soon as its respective variable is instantiated
  - The latter idea is used by the *backtracking* method

The n-Queens Problem: No Mutual Attacks!

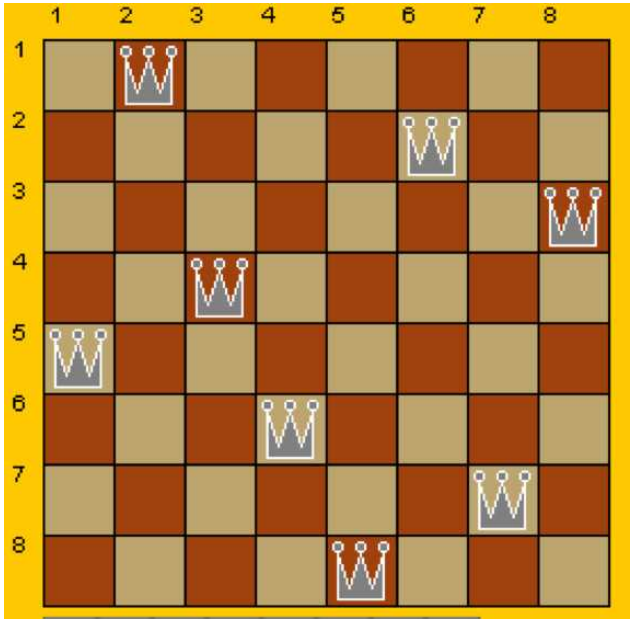
Place n queens on a square $n \times n$ chess board in a way that no two queens would be able to attack each other

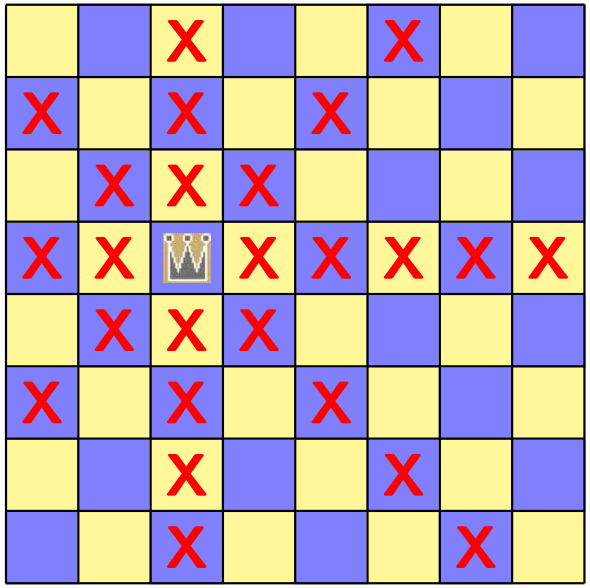
- Two queens can attack eacn others on the same row, column, or diagonal
- The board has n rows, n columns, and $4n - 2$ diagonals
- **Distinct solution**: may differ by symmetric operations (rotation and reflection of the board)
  - Eight queens ($n = 8$) – 92 distinct solutions
- **Unique solution**: symmetric configurations are the same
  - Eight queens ($n = 8$) – 12 unique solutions
- Exponential complexity
  - The world record for $n = 25$ – over 50 years of the cumulated computing time (a heterogeneous computer cluster over the world)

http://drdobbs.com/go-parallel/article/showArticle.jhtml?articleID=214303519

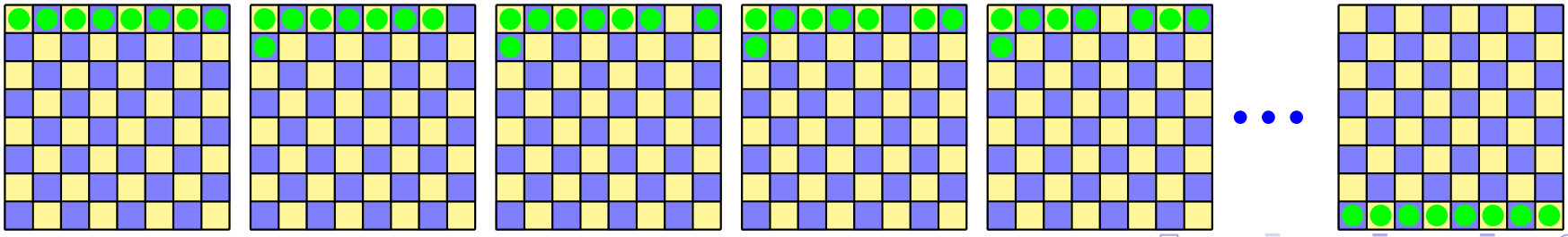
Eight Queens Problem

Solution – one queen per row, column or diagonal:





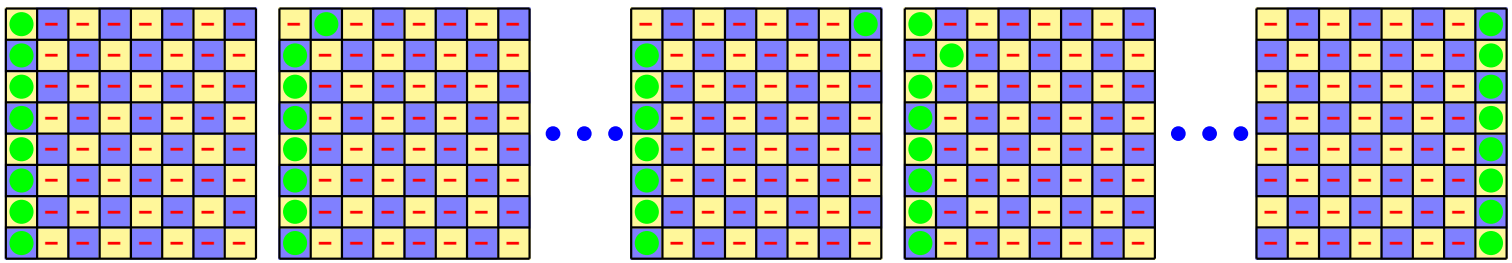
Full exhaustion: $n = \binom{64}{8} = 4,426,165,368$ assignments!!!



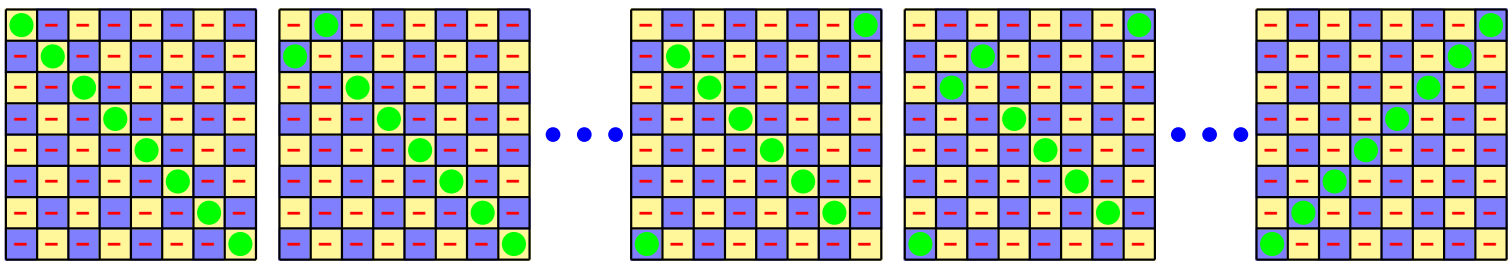
Faster Search by Constraints Propagation

A vast majority of the assignments do not match constraints

- The brute-force exhaustion takes no account of constraints
- Faster search if each next assigned value is consistent with the current partial solution
- "One per row" constraint: $n = 8^8 = 16,777,216 \ll \binom{64}{8}$



- "One per row and column": $n = 8! = 40,320 \ll 8^8$



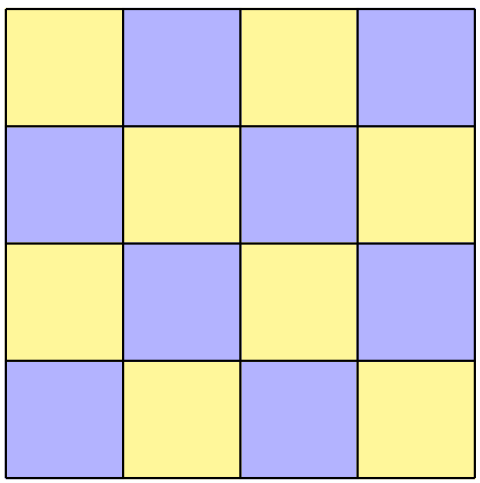
Backtracking

The most common algorithm for systematic search through all the possible combinations within a search space

- Incrementally attempts to extend a partial solution toward a complete solution
  - Partial solution: consistent values for some of the variables
  - Extension by repeatedly choosing a value for another variable consistent with the values in the current partial solution
- A merge of generate and test phases from the full exhaustion
  1. Instantiate the variables sequentially
  2. As soon as all the variables relevant to a constraint are instantiated, test the validity of the constraint
  3. Whenever the test fails, backtrack to the most recently instantiated variable that still has alternatives available
  4. Terminate when all the variables are successfully instantiated

Example: 4 Queens Problem

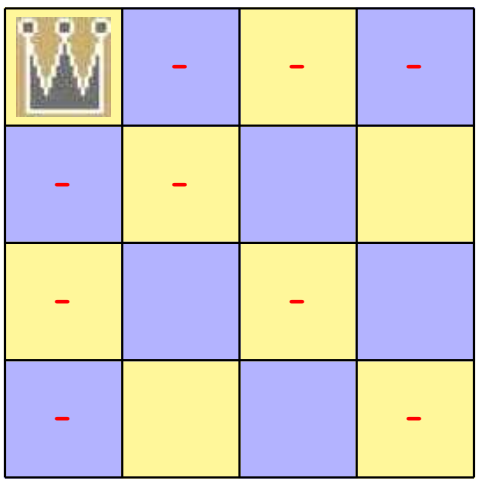
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

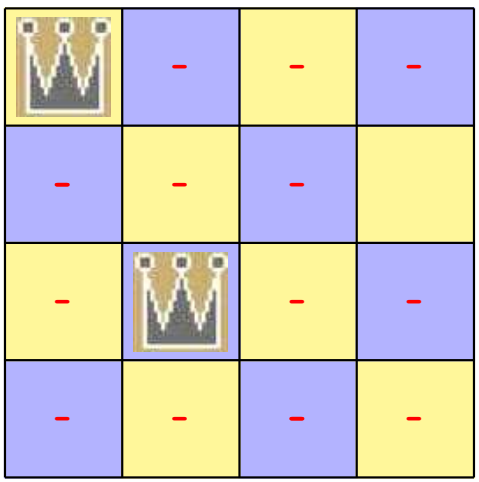
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

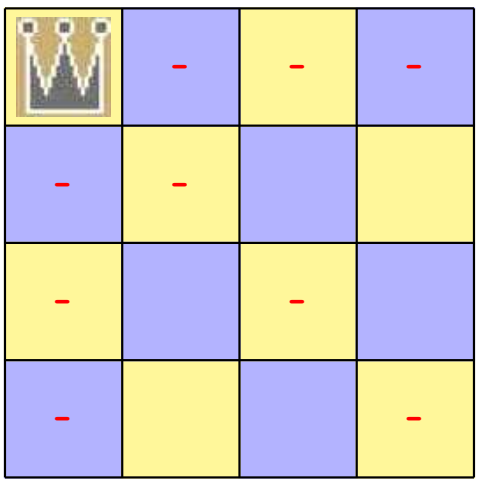
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

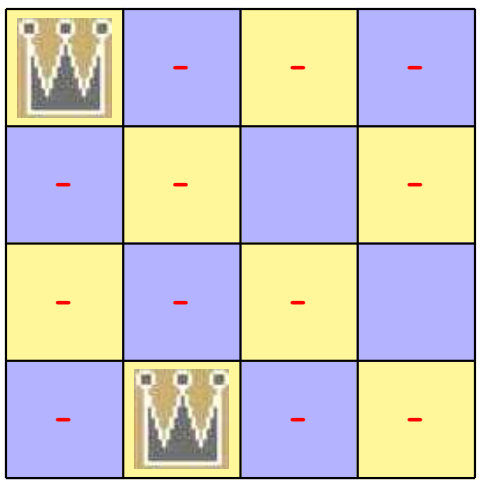
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

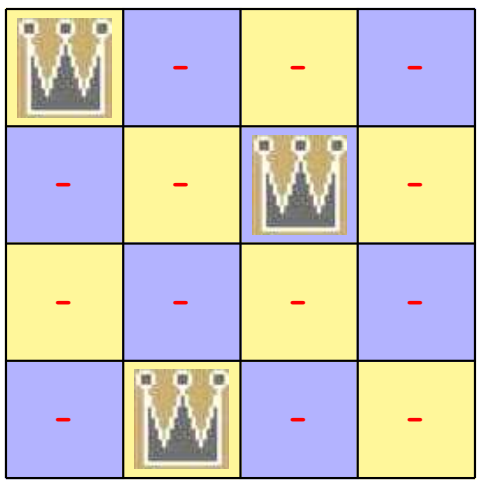
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

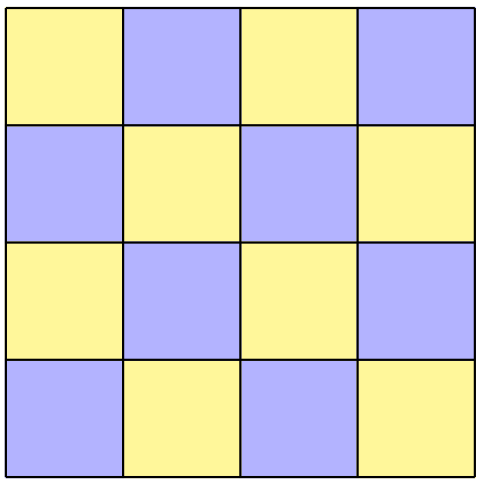
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

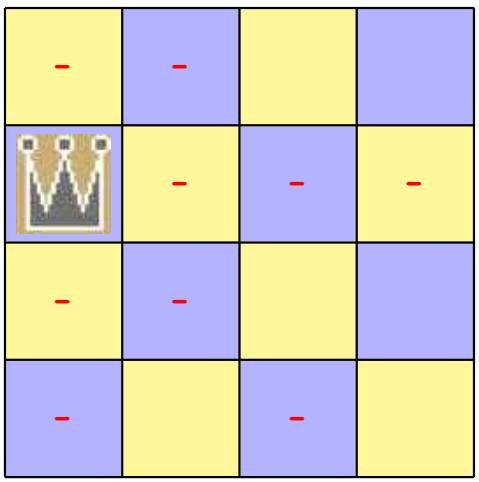
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

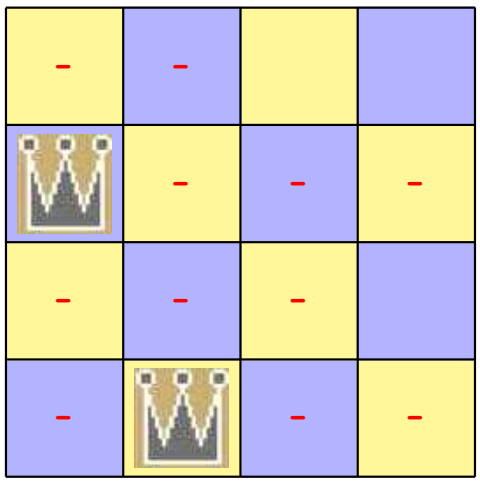
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

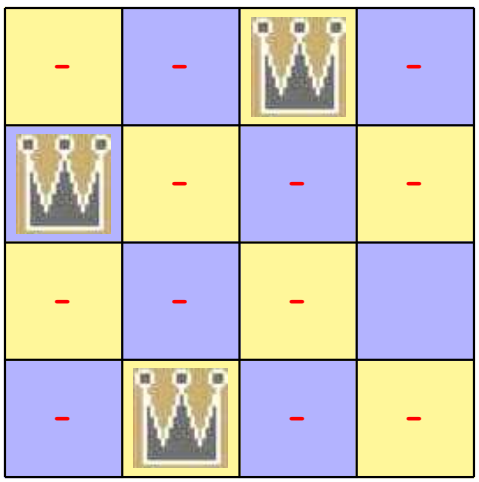
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

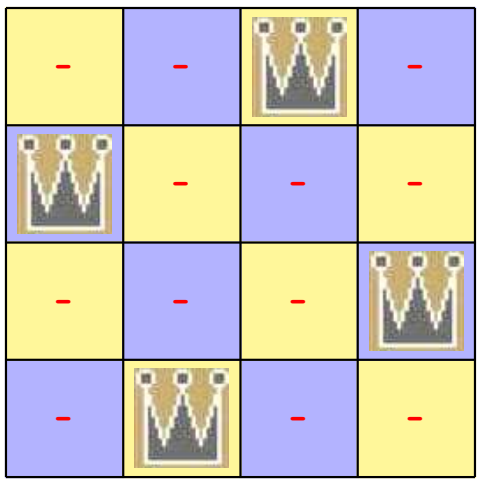
Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Example: 4 Queens Problem

Brute-force: $\binom{16}{4} = 43,680$; "1 in row": $4^4 = 256$; "1 in row/column": $4! = 24$



- Step 1: Instantiate Queen 1 position (1^{st} out of 4)
- Step 2: Instantiate Queen 2 position (1^{st} out of 2)
- Step 3: Backtracking (no position for Queen 3)
- Step 4: Instantiate Queen 2 position (2^{nd} out of 2)
- Step 5: Instantiate Queen 3 position (1^{st} out of 1)
- Step 6: Backtracking (no position for Queens 4-2)
- Step 7: Instantiate Queen 1 position (2^{nd} out of 4)
- Step 8: Instantiate Queen 2 position (1^{st} out of 1)
- Step 9: Instantiate Queen 3 position (1^{st} out of 1)
- Step 10: Instantiate Queen 4 position (1^{st} out of 1)

Backtracking Exemplified by the n Queens Problem

Algoritm	Positions to check for $n = 8$
Brute-force	$\binom{64}{8} = 4,426,165,368$
"One per row"	$8^8 = 16,777,216$
"One per row and column"	$8! = 40,320$
Backtracking	2,057

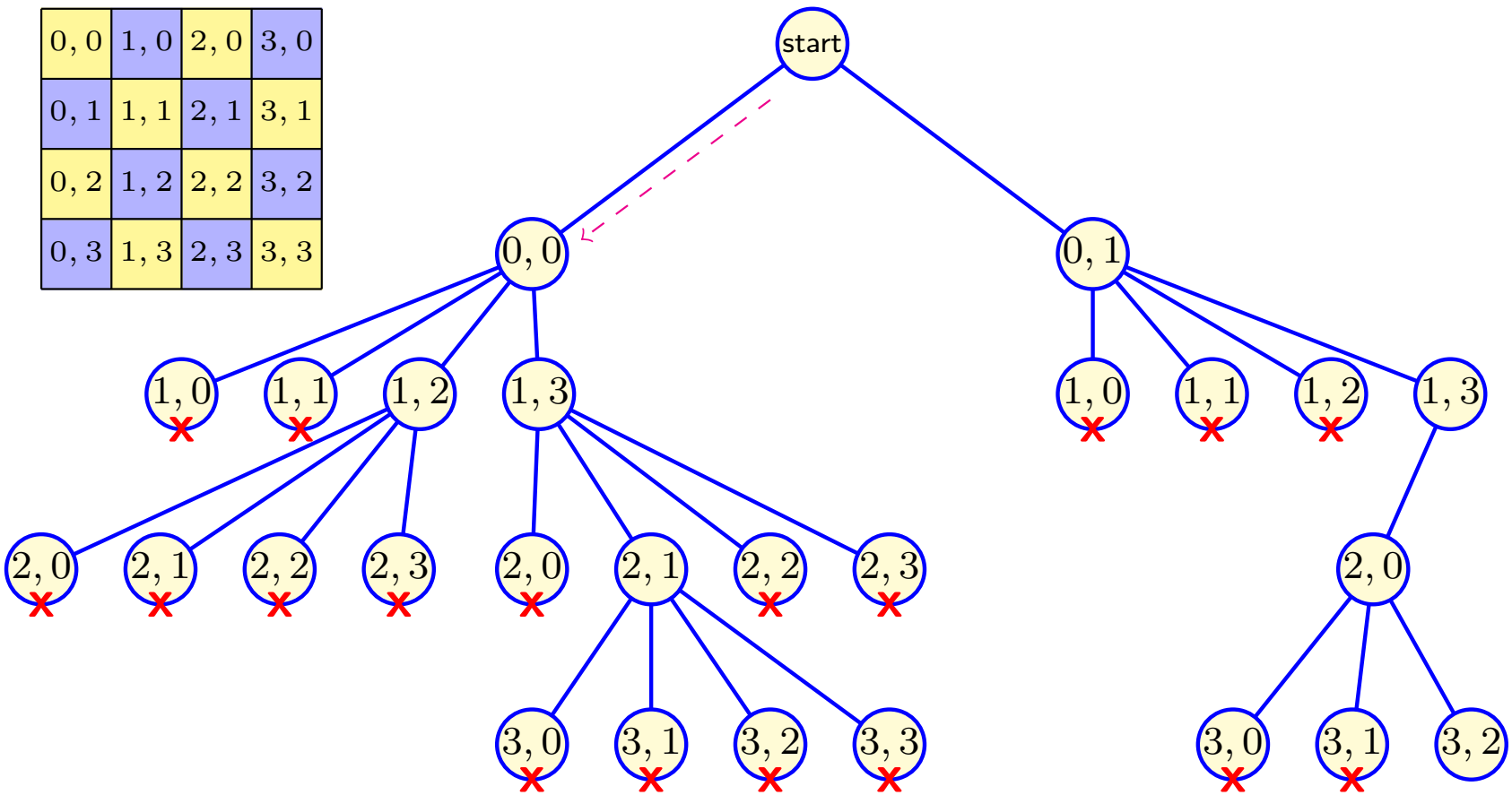
n	4	5	6	7	8	9	10	11	12	13
unique	1	2	1	6	12	46	92	341	1,787	92,333
distinct	2	10	4	40	92	352	724	2,680	14,200	73,712

- Backtracking is just **depth-first search** (DFS) on an implicit graph of configurations
- Can easily iterate through all subsets or permutations of a set
- Ensures correctness by enumerating all possibilities
- However, to be efficient, it must prune the search space

Backtracking as DFS (for the 4 Queens Example)

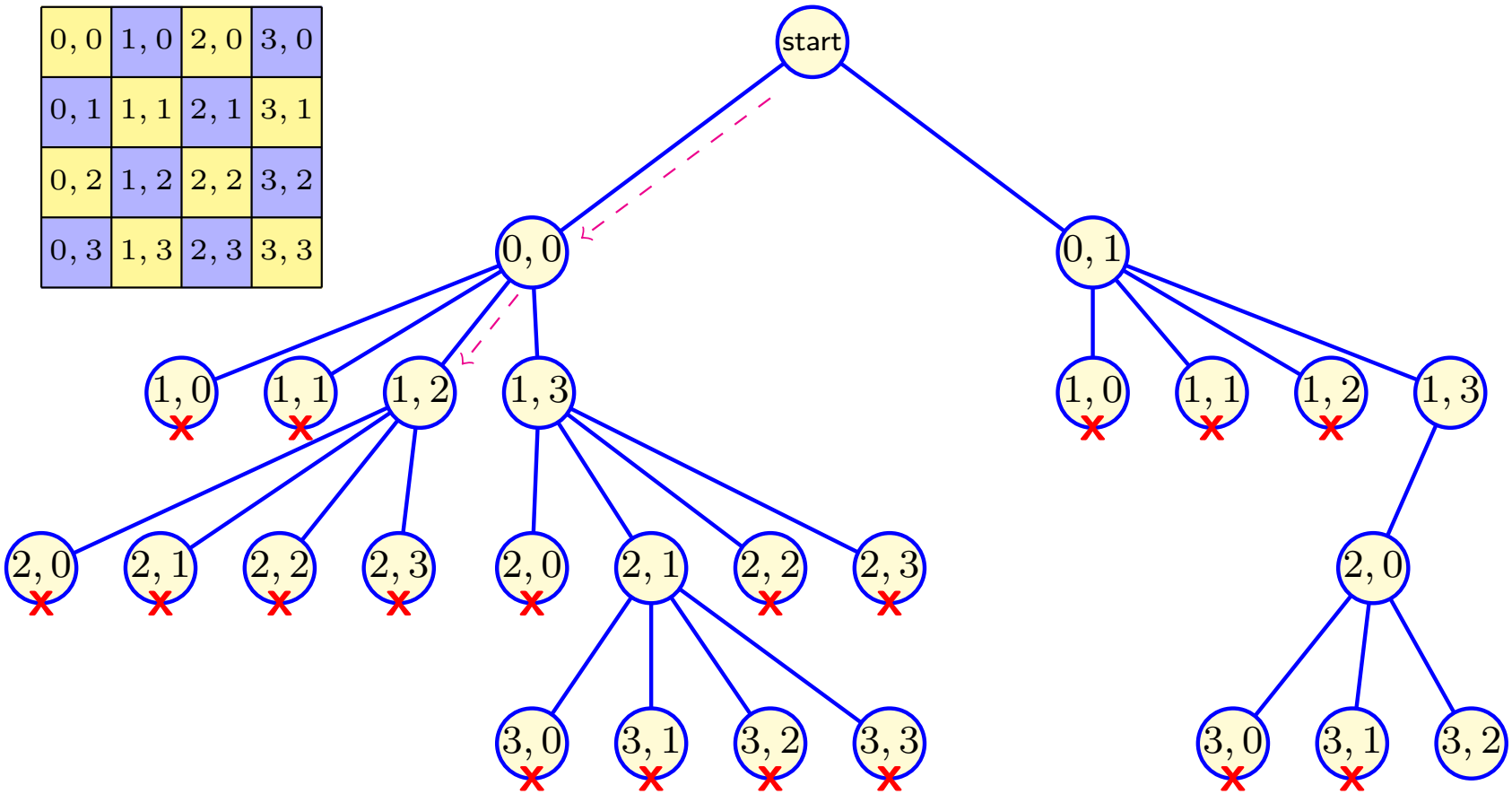
Step:	1
Position:	0,0

0, 0	1, 0	2, 0	3, 0
0, 1	1, 1	2, 1	3, 1
0, 2	1, 2	2, 2	3, 2
0, 3	1, 3	2, 3	3, 3



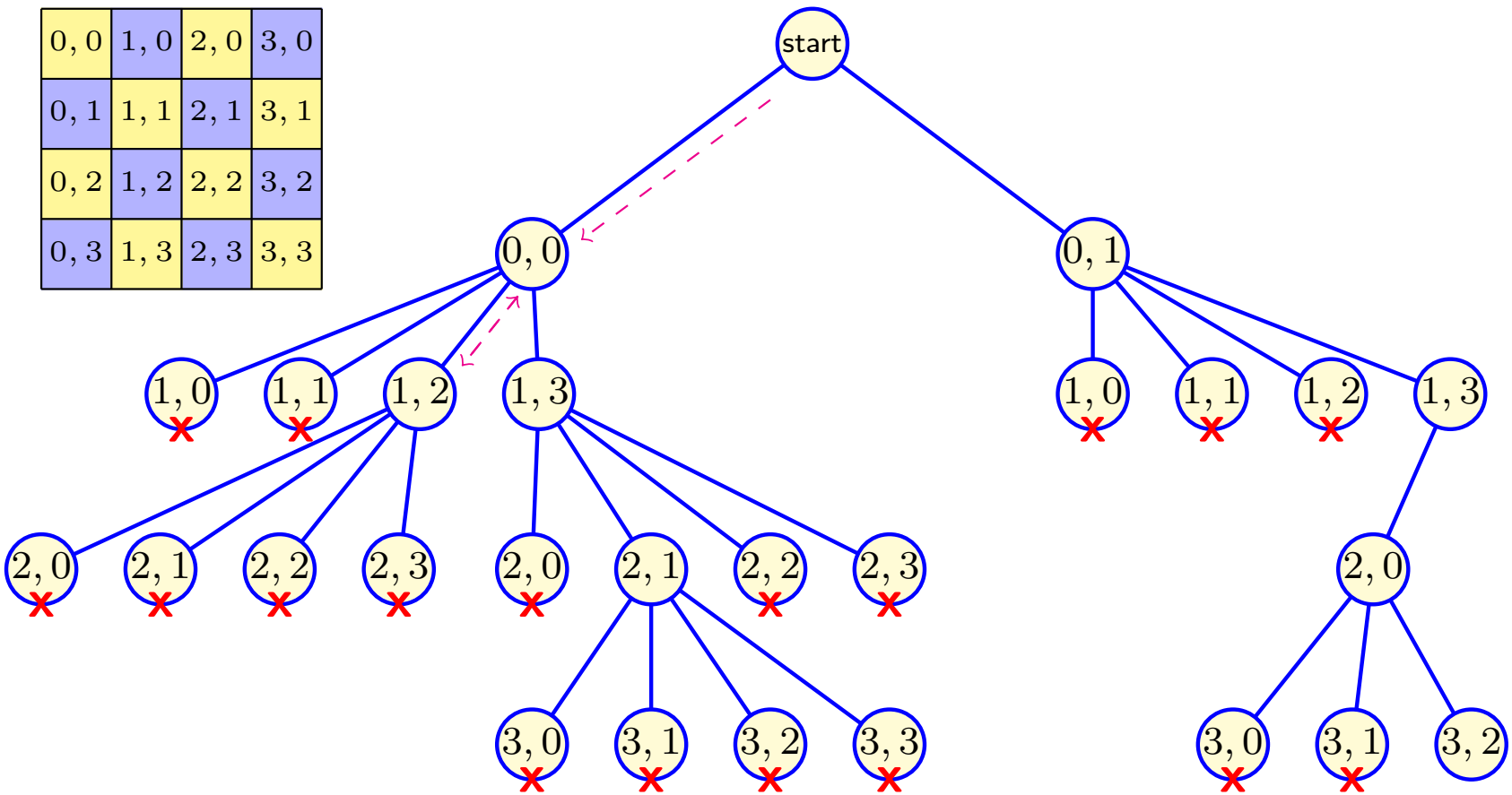
Backtracking as DFS (for the 4 Queens Example)

Step:	1	2
Position:	0,0	1,2



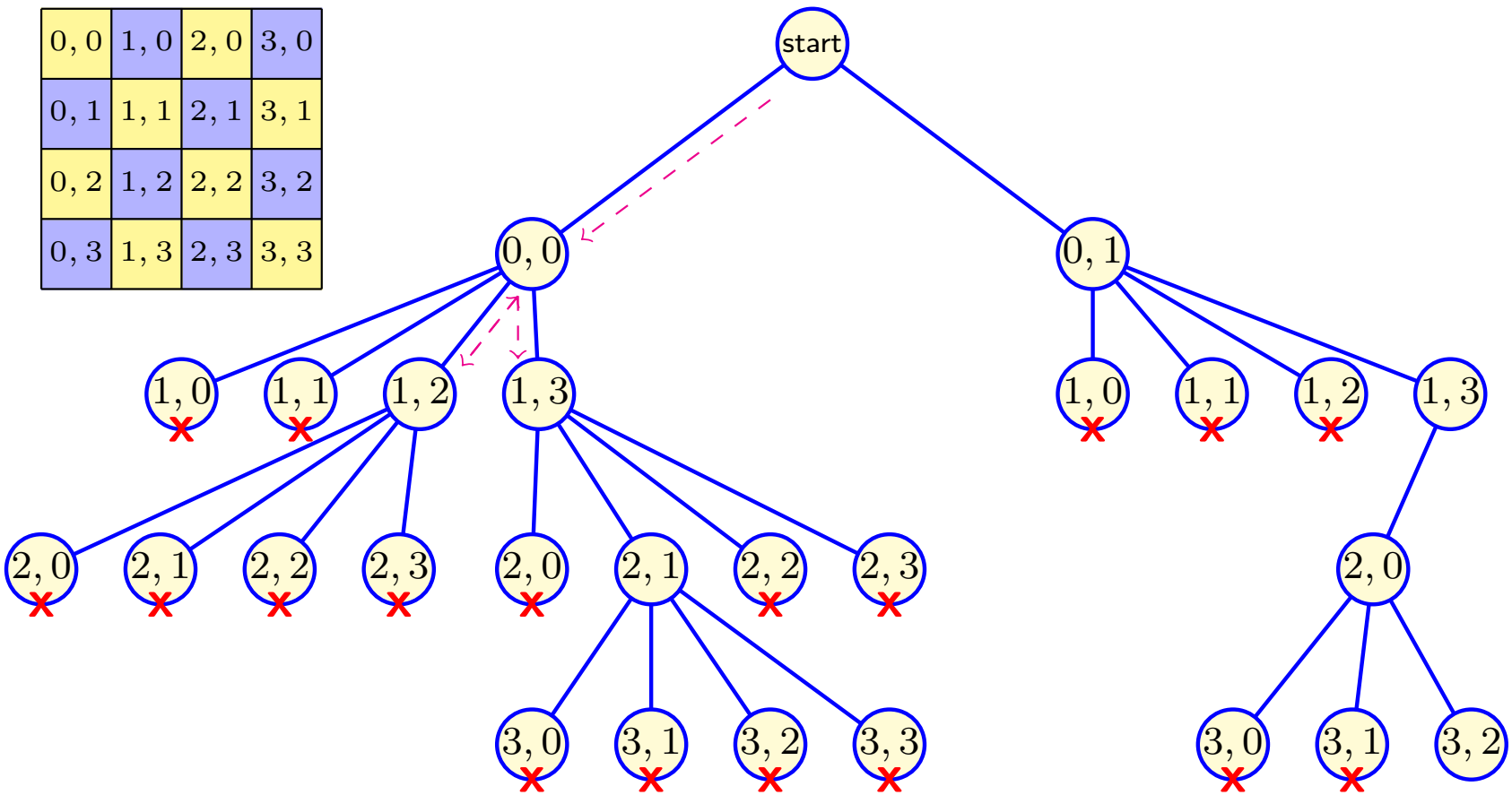
Backtracking as DFS (for the 4 Queens Example)

Step:	1	2	3
Position:	0,0	1,2	**BT**



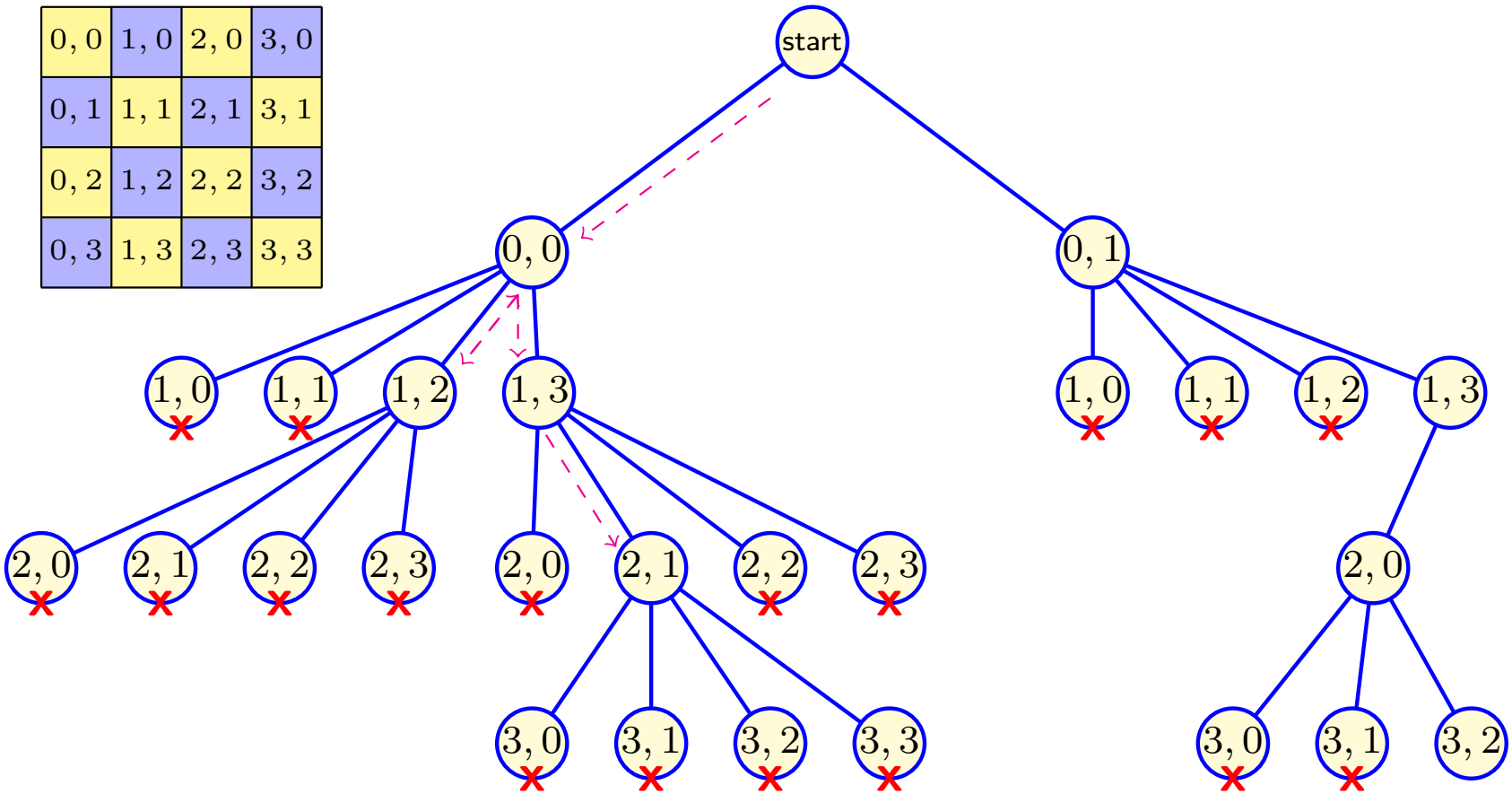
Backtracking as DFS (for the 4 Queens Example)

Step:	1	2	3	4
Position:	0,0	1,2	**BT**	1,3



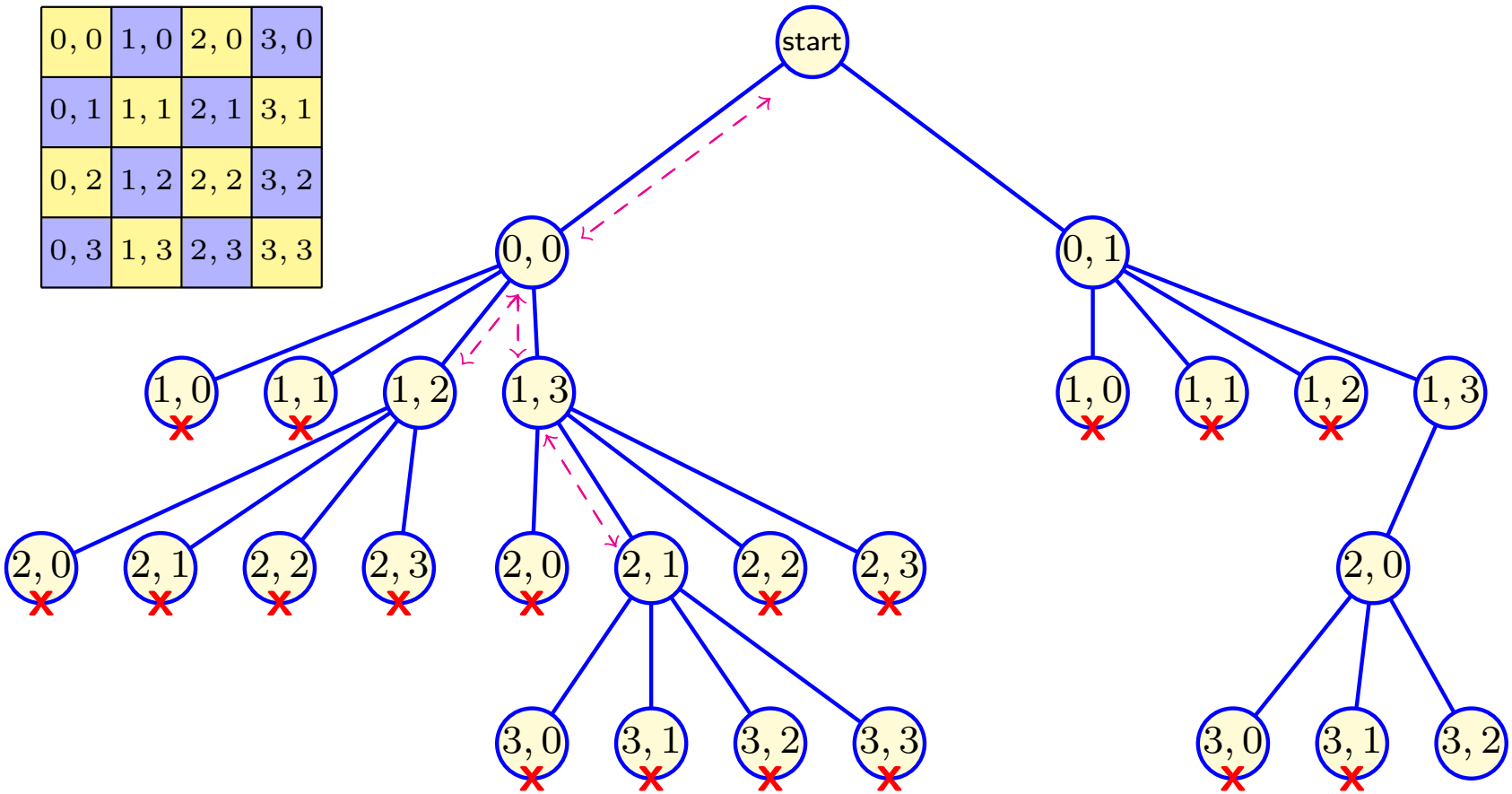
Backtracking as DFS (for the 4 Queens Example)

Step:	1	2	3	4	5
Position:	0,0	1,2	**BT**	1,3	2,1



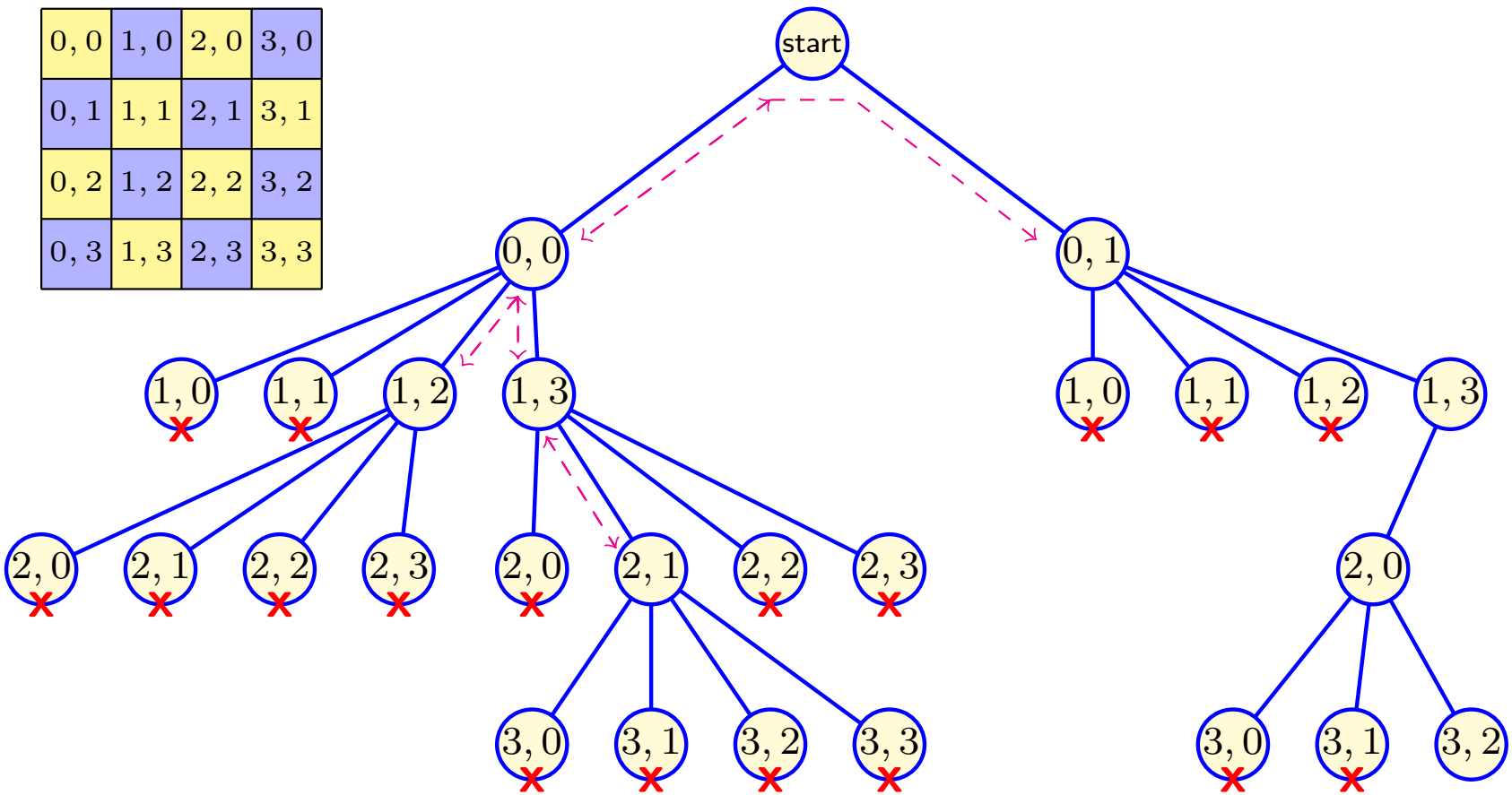
Backtracking as DFS (for the 4 Queens Example)

Step:	1	2	3	4	5	6
Position:	0,0	1,2	**BT**	1,3	2,1	**BT**



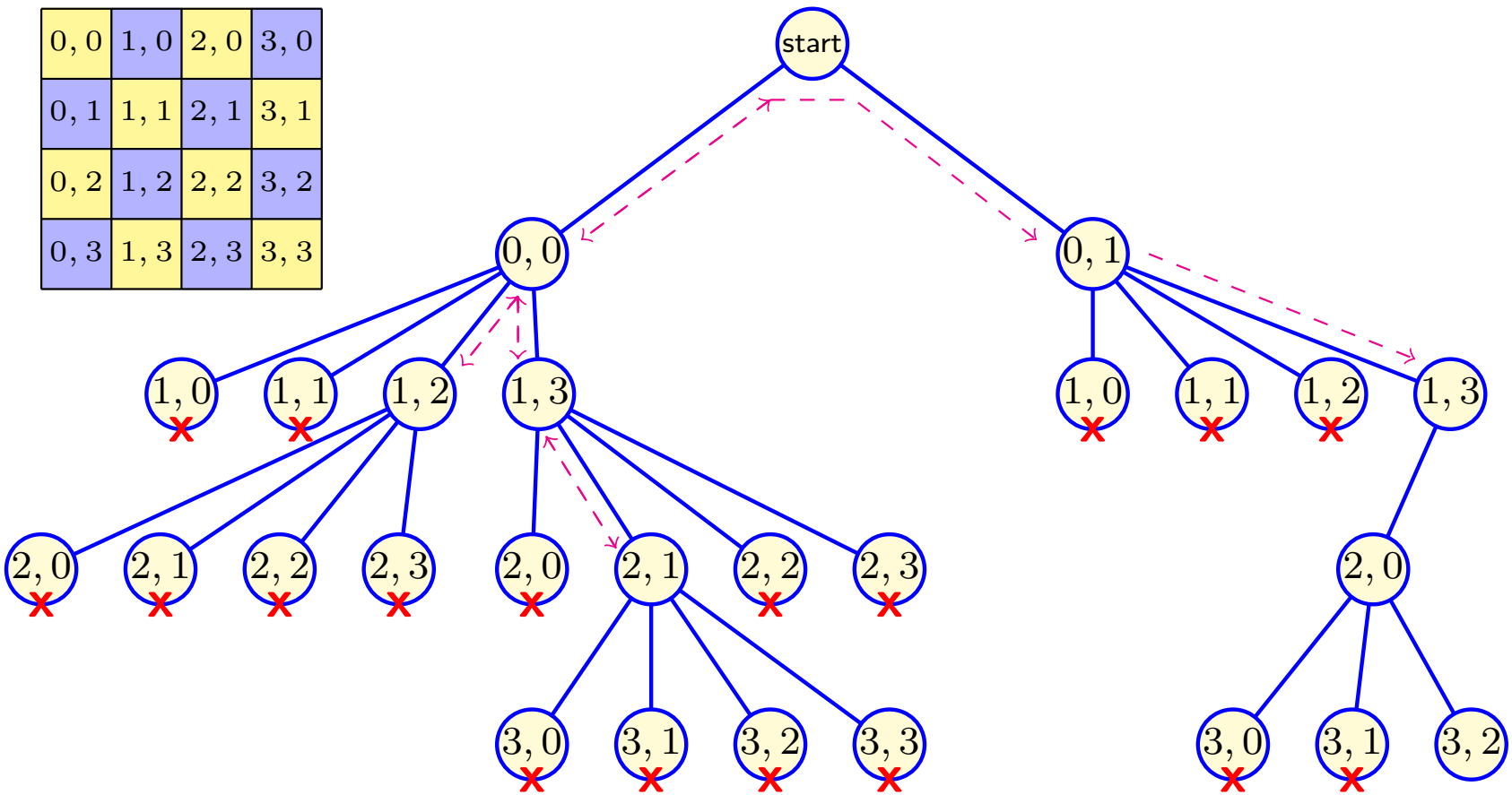
Backtracking as DFS (for the 4 Queens Example)

Step:	1	2	3	4	5	6	7
Position:	0,0	1,2	**BT**	1,3	2,1	**BT**	0,1



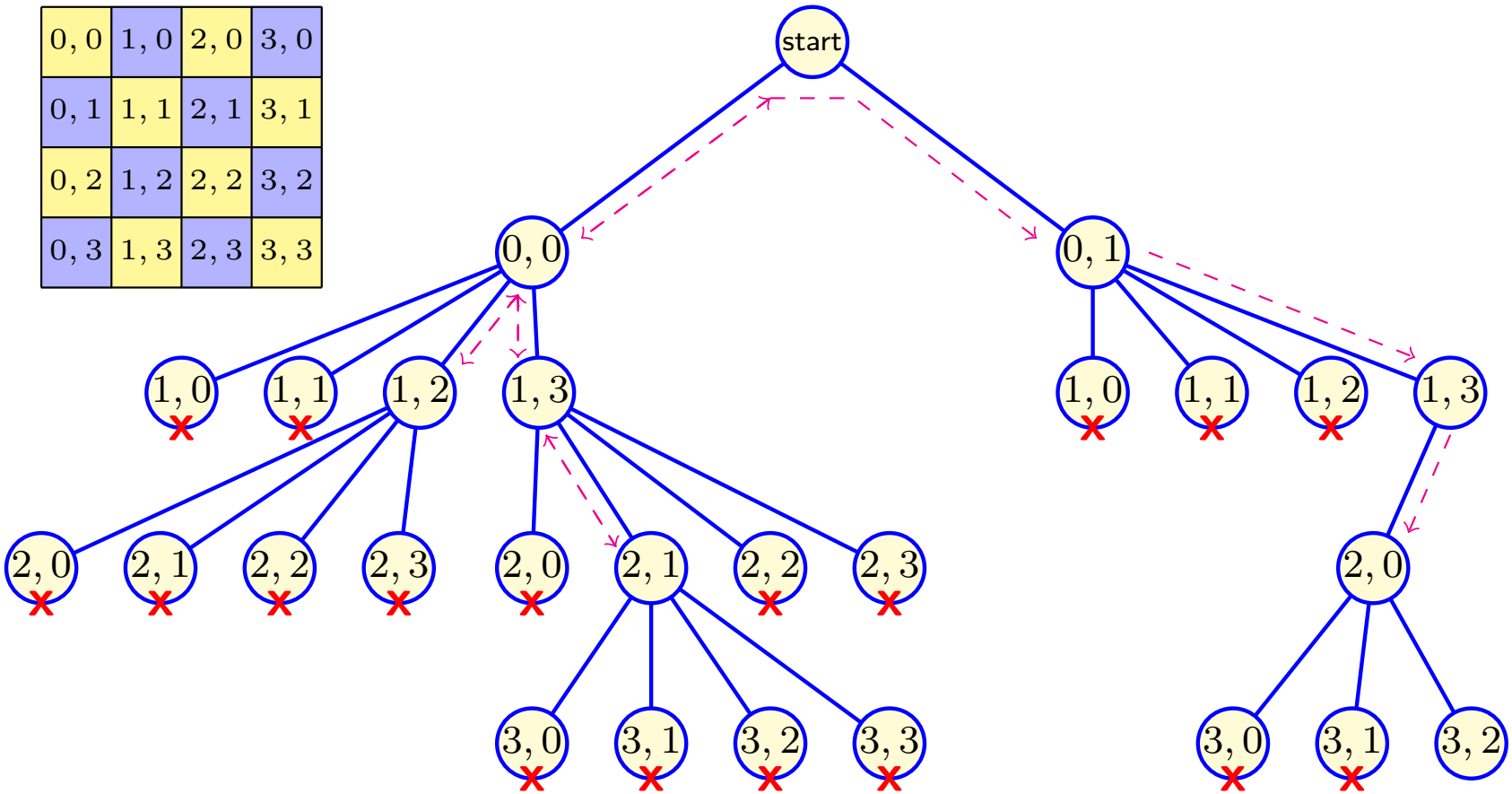
Backtracking as DFS (for the 4 Queens Example)

Step:	1	2	3	4	5	6	7	8
Position:	0,0	1,2	**BT**	1,3	2,1	**BT**	0,1	1,3



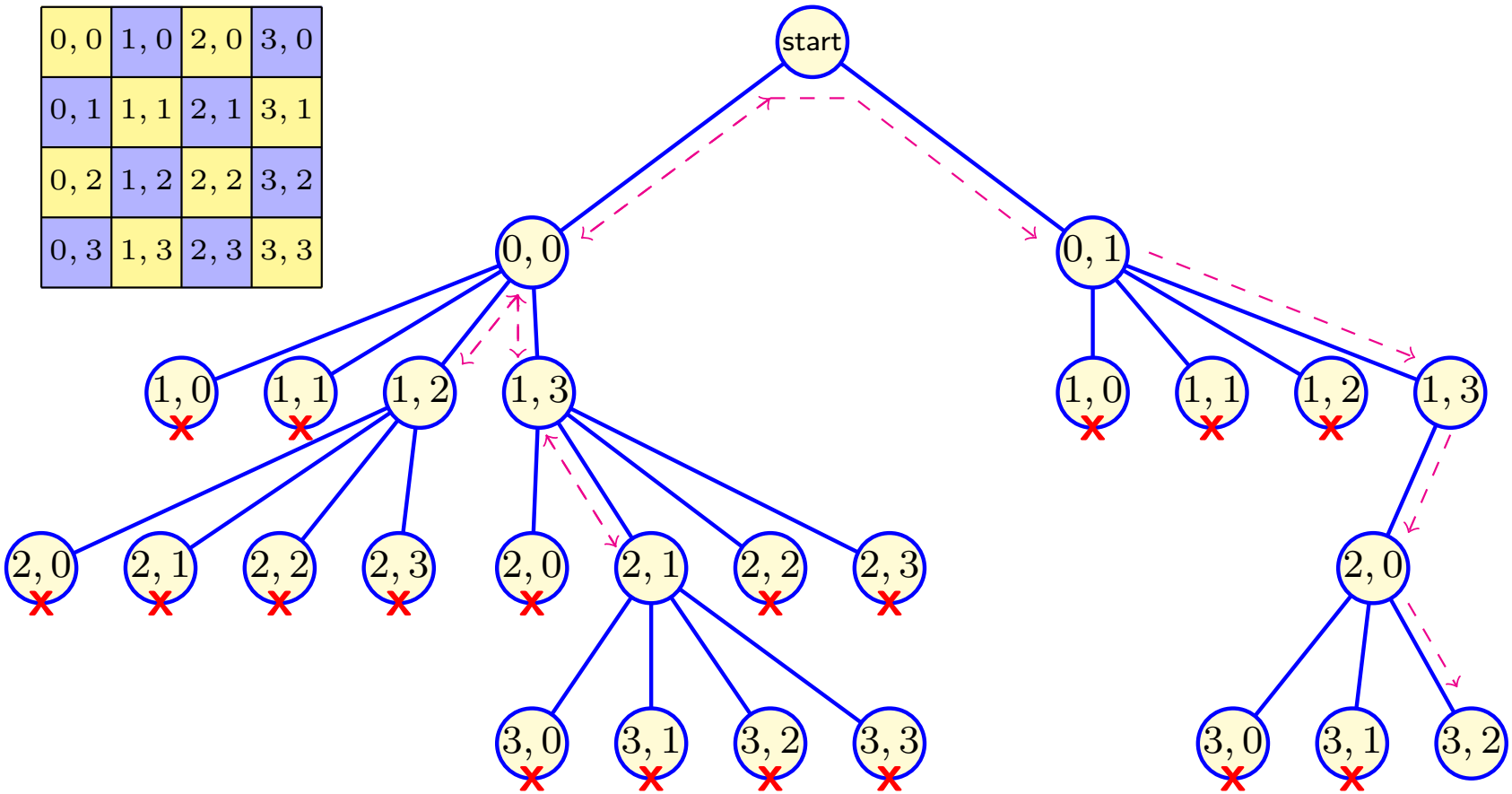
Backtracking as DFS (for the 4 Queens Example)

Step:	1	2	3	4	5	6	7	8	9
Position:	0,0	1,2	**BT**	1,3	2,1	**BT**	0,1	1,3	2,0



Backtracking as DFS (for the 4 Queens Example)

Step:	1	2	3	4	5	6	7	8	9	10
Position:	0,0	1,2	**BT**	1,3	2,1	**BT**	0,1	1,3	2,0	3,2



Standard Backtracking: Major Drawbacks

Thrashing (repeated failure due to the same reason)

- Without identification of conflicting variables, search in different parts of the space keeps failing for the same reason
- **Intelligent backtracking**: returning directly to the variable that caused the failure

Redundant work

- Even if the conflicting values of variables are identified during the intelligent backtracking, they are not remembered for subsequent immediate conflict detection
- **Dependency-directed backtracking** eliminates both above drawbacks

Too late detection of the conflict

- Applying consistency techniques to **forward check** the possible future conflicts

Backtracking vs. Branch-and-Bound

- **Backtracking**: Depth-first search (DFS) with pruning for systematic enumeration of the space of all candidate solutions
- **Branch-and-bound**: Breadth-first search (BFS) with pruning
  - Performs better for many optimisation problems
  - Example: $\min_{\mathbf{x}\in\mathbb{S}} f(\mathbf{x})$ – minimise an objective function $f(\mathbf{x})$ of variables $\mathbf{x} = (x_1, \ldots, x_n)$ over a region of feasible solutions $\mathbb{S}$
    - Typical constraints: discrete $x_i \in \mathbb{X} = \{\alpha_1, \ldots, \alpha_K\}$; $\mathbb{S} = \mathbb{X}^n$
  - DFS through K^n solutions $(\alpha_1, \alpha_1, \ldots, \alpha_1)$, $(\alpha_2, \alpha_1, \ldots, \alpha_1)$, $\ldots$, $(\alpha_K, \alpha_K, \ldots, \alpha_K)$
  - DFS: Pruning and backtracking if the subsequence $(x_1, \ldots, x_k)$ cannot be continued due to the problem constraints
  - BFS: Looking first through all $\alpha_k \in \mathbb{X}$ for x_1, then for x_2, etc.
  - BFS: Pruning by finding at each level i bounds for the objective function, which do not depend on the values assigned to the remaining variables $(x_{i+1}, \ldots, x_n)$

Branch-and-Bound (B&B): Basic Principles

B&B is an algorithm paradigm to be filled out for each problem

Splitting (branching) and **bounding** to discard large subsets of fruitless candidates

- **Branching** divides a set $\mathbb{S}$ of candidates into two or more smaller sets such that their union covers $\mathbb{S}$
- Recursive branching defines the search tree whose nodes are the subsets of $\mathbb{S}$
- **Bounding** computes upper and lower bounds for the objectivel function

Pruning for minimisation: if the lower bound for some tree node (set of candidates) q_i is greater than the upper bound for some other node q_k, then q_i and all its descendants may be discarded

Branch-and-Bound (B&B): Basic Principles

B&B searches through a dynamically built search tree

- Root node – the initial problem to be solved, e.g. $\min_{\mathbf{x}\in\mathbb{S}} f(\mathbf{x})$
- Each other node – a subproblem of the initial problem
  - Children of a node q are subproblems derived from q through imposing a new constraint for each subproblem
  - Descendants of the node q – subproblems satisfying the same constraints as q and additionally a number of others
  - Leaves – the feasible solutions (the exponential number of the leaves for an NP-hard problem)
- Bounding function $b(\mathbf{x})$ associates the bound to each node:

$$\begin{array}{lcll} b(q_i) & \leq & f(q_i) & \text{for all nodes } q_i \text{ in the tree} \\ b(q_i) & = & f(q_i) & \text{for all leaves in the tree} \\ b(q_i) & \geq & b(q_j) & \text{if } q_j \text{ is the parent of } q_i \end{array}$$

By J. Clausen: Branch and bound algorithms: Principles and examples. Copenhagen, 1999

NP-Hard Integer "0–1" Knapsack Problem

Maximise Total Value of Objects in a Knapsack

Given:

1. a knapsack of a limited size S and
2. a set $\Omega = \{\omega_1, \ldots, \omega_n\}$ of n objects of size s_i and value v_i each; $i = 1, \ldots, n$,

Select a most valuable subset of the objects to place into the knapsack:

$$V^* = \max_{(x_1,\ldots,x_n)\in\{0,1\}^n} \left\{ \sum_{i=1}^{n} x_i v_i \;\middle|\; \sum_{i=1}^{n} x_i s_i \leq S \right\}$$

The $0-1$ constraint $x_i \in \{0, 1\}$ for the indicator variables: only the whole object ω_i can be placed into the knapsack

Relaxed "0–1" Knapsack Problem

Relaxed, or fractional problem is much simpler:

$$V^\circ = \max_{(x_1,\ldots,x_n)\in[0,1]^n} \left\{ \sum_{i=1}^{n} x_i v_i \;\middle|\; 0 \le x_1, \ldots, x_n \le 1; \sum_{i=1}^{n} x_i s_i \le S \right\}$$

The relaxation implies that $V^\circ \ge V^*$

- The optimal solution of the relaxed problem is an upper bound for the optimal solution of the original problem with integer $x_i \in \{0, 1\}$
- For a fixed integer subsequence $x_1, \ldots, x_i$, such that $\sum_{j=1}^{i} x_i s_i < S$, the solution with the relaxed remaining variables $x_{i+1}, \ldots, x_n$ is an upper bound for the optimal solution of the original problem constrained by fixing the indicators $x_1, \ldots, x_i$

Relaxed Knapsack Problem: Greedy Algorithm

1. Sort the objects by decreasing value density $\eta_i = \frac{v_i}{s_i}$
2. Set $x_i = 1$ for the top k objects: $\sum_{i=1}^{k} s_i \le S < \sum_{i=1}^{k+1} s_i$
3. Set $x_{k+1} = \frac{1}{s_{k+1}} \left(S - \sum_{i=1}^{k} s_k \right)$
4. The solution $V^\circ = \sum_{i=1}^{k} v_i + x_{k+1} v_{k+1}$

Example: $S = 11$; $n = 5$;

Object i	1	2	3	4	5
Value v_i	1	6	18	21	35
Size s_i	1	2	3	6	7
Density $\eta_i = \frac{v_i}{s_i}$	1	3	6	3.5	5

$\Rightarrow$

3_1	5_2	4_3	2_4	1_5
18	35	21	6	1
3	7	6	2	1
6	5	3.5	3	1

Solution of the relaxed problem (the **upper bound** for the $0-1$ one):

- $\mathbf{x} = \left(x_1 = 0, x_2 = 0, x_3 = 1, x_4 = \frac{1}{6}, x_5 = 1 \right)$
- $V^\circ = 18 + 35 + \frac{1}{6} \cdot 21 = 56.5$; $S^\circ = 7 + 3 + \frac{1}{6} \cdot 6 = 11$

B&B Solution: State Space Tree

Information associated with a tree node q:

$$\begin{cases} \text{Total current value } V_{\text{c:}q} \text{ in the knapsack's contents} \\ \text{Total current size } S_{\text{c:}q} \text{ of the knapsack's contents} \\ \text{Max potential value } V_{\text{p:}q} \text{ (bound) the knapsack can hold} \end{cases}$$

- The root – the empty knapsack: $V_{c:0} = S_{c:0} = 0; V_{p:0} = V^{\circ}$
- The nodes q at level i – for the indicators $x_i = 1$ and $x_i = 0$
  - Constraint: the first i items have been already considered
  - Greedy solution $V_{\text{p:}q}$ with the already selected $x_1, \ldots, x_i$
    - $V_{\text{c:}q} = \sum_{j=1}^{i} x_i v_i; \;\; S_{\text{c:}q} = \sum_{j=1}^{i} x_i s_i$
    - Each of the remaining objects $\omega_{i+1}, \ldots, \omega_n$ is placed one-by-one until the object ω_k is too big w.r.t. the available size:

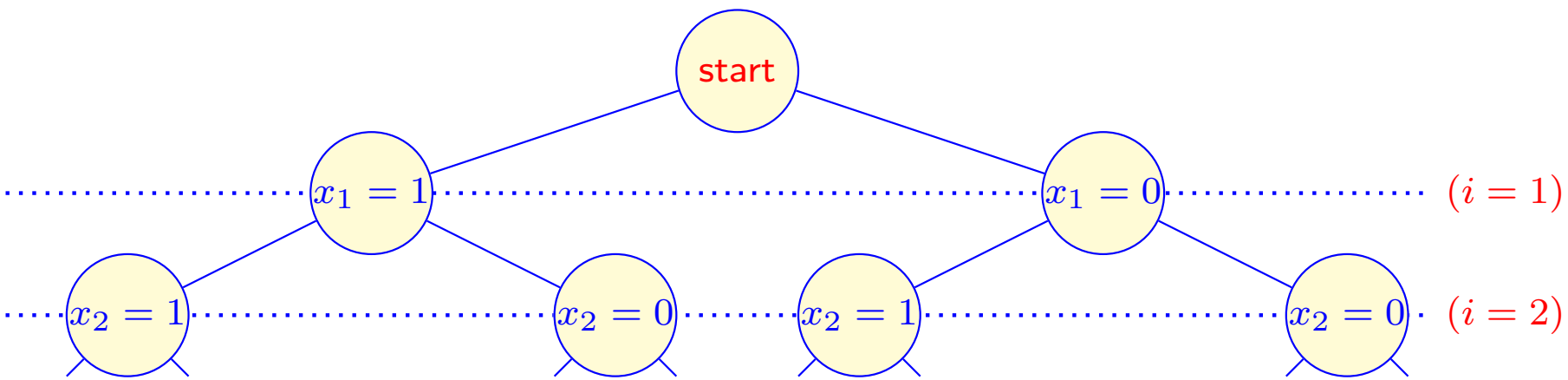
$$\sum_{j=i+1}^{k-1} s_j \leq S - S_{\text{c:}q} < \sum_{j=i+1}^{k} s_j \;\Rightarrow\; V_{\text{p:}q} = V_{\text{c:}q} + \sum_{j=i+1}^{k-1} v_j + \frac{\left(S - S_{\text{c:}q} - \sum_{j=i+1}^{k-1} s_j\right)}{s_k} v_k$$

B&B Solution: State Space Tree

Fractional amount of the first too big object, ω_k, determines the potential greedy maximum, being the upper bound of the goal one:

$$S_{\mathrm{c}:q} = \sum_{j=1}^{i} x_j s_j;\ V_{\mathrm{c}:q} = \sum_{i=1}^{i} x_j v_j;\ k \leftarrow \sum_{j=i+1}^{k-1} s_j < S - S_{\mathrm{c}:q} < \sum_{j=i+1}^{k} s_j$$

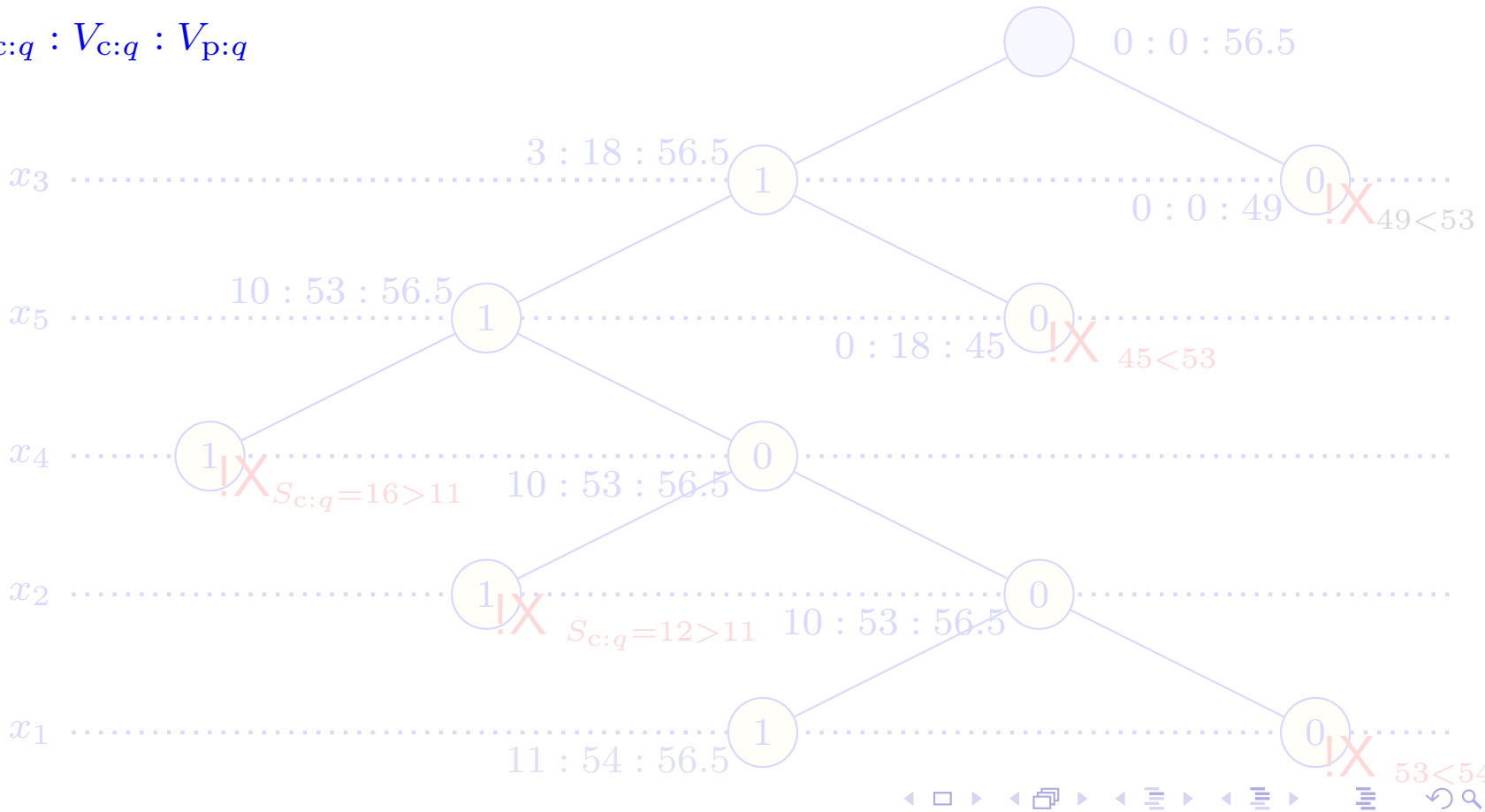
$$V_{\mathrm{p}:q} = V_{\mathrm{c}:q} + \sum_{j=i+1}^{k-1} v_i + \frac{S - S_{\mathrm{c}:q} - \sum_{j=i+1}^{k-1} s_j}{s_k} v_k$$



Example: B&B Solution for the $0-1$ Knapsack

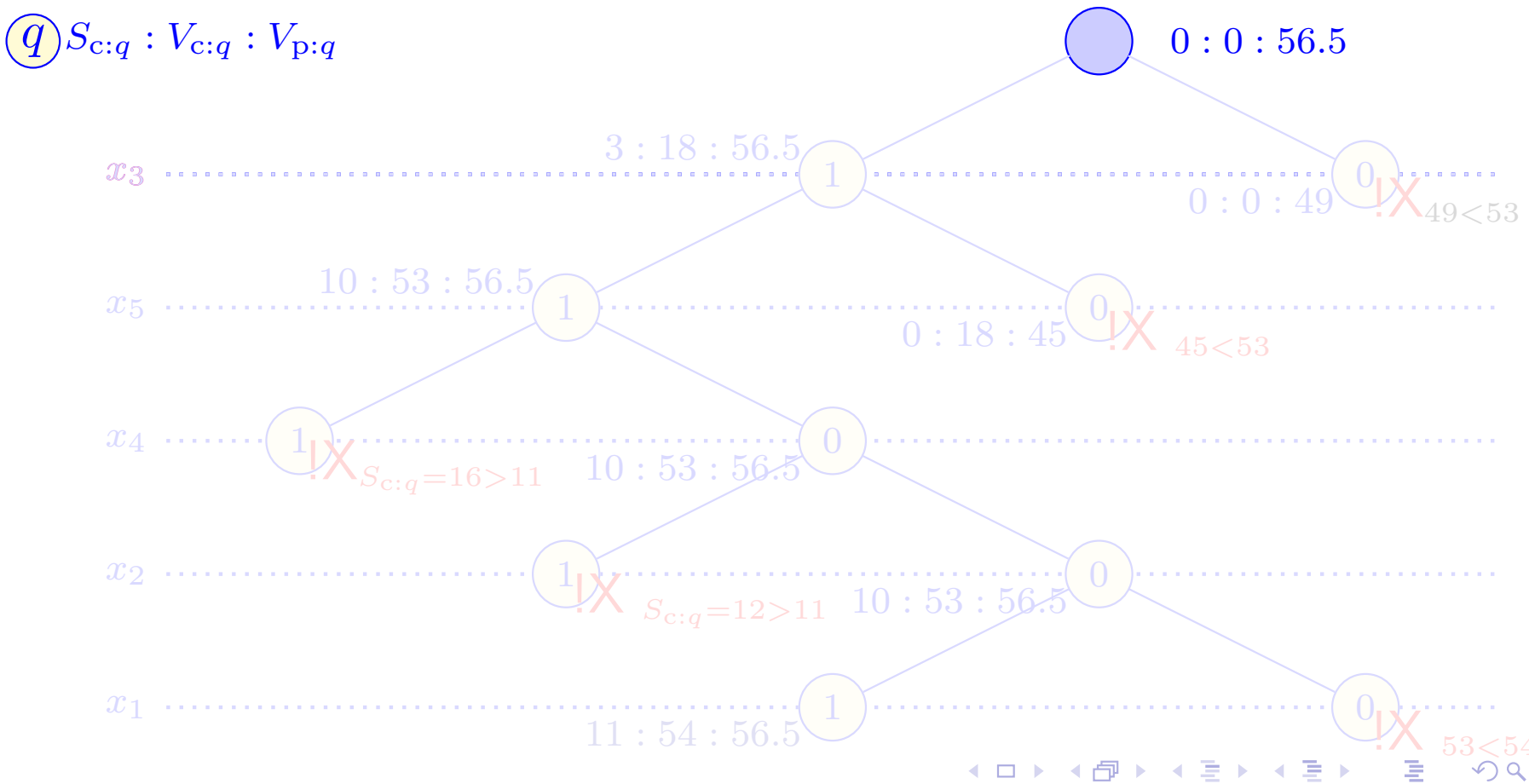
i	3_1	5_2	4_3	2_4	1_5
v_i	18	35	21	6	1
s_i	3	7	6	2	1
η_i	6	5	3.5	3	1

q $S_{c:q} : V_{c:q} : V_{p:q}$



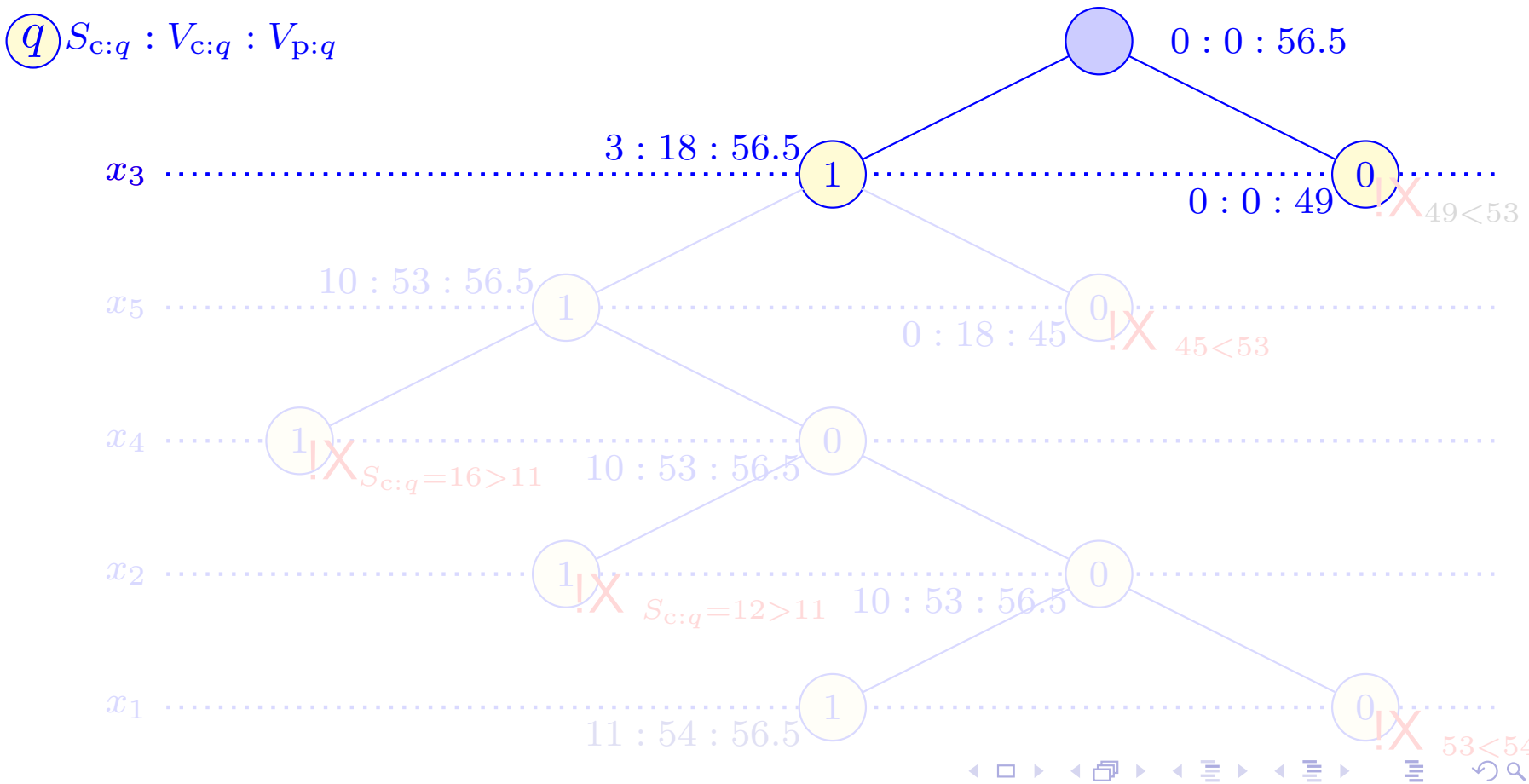
Example: B&B Solution for the $0-1$ Knapsack

i	3_1	5_2	4_3	2_4	1_5
v_i	18	35	21	6	1
s_i	3	7	6	2	1
η_i	6	5	3.5	3	1



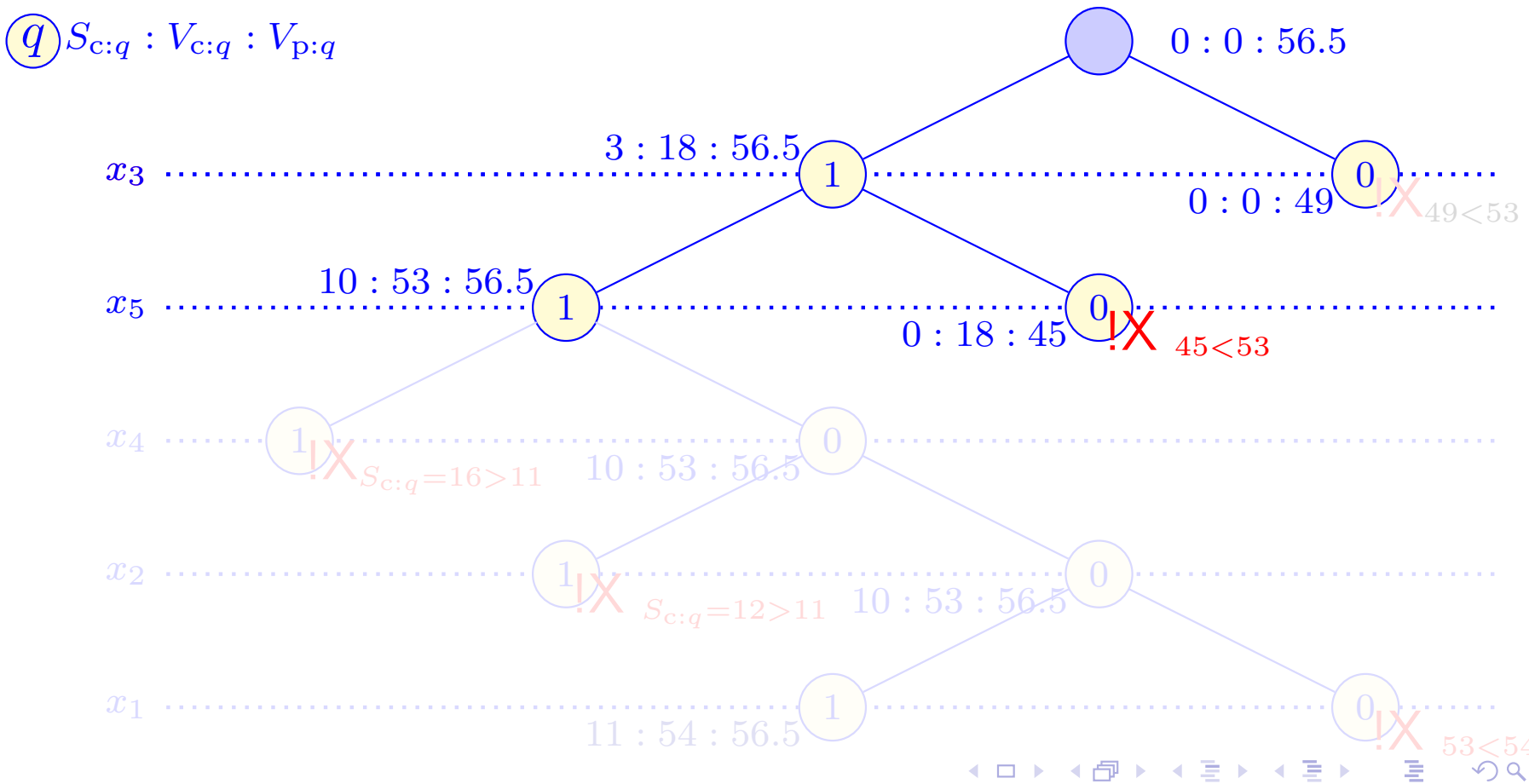
Example: B&B Solution for the $0-1$ Knapsack

i	3_1	5_2	4_3	2_4	1_5
v_i	18	35	21	6	1
s_i	3	7	6	2	1
η_i	6	5	3.5	3	1



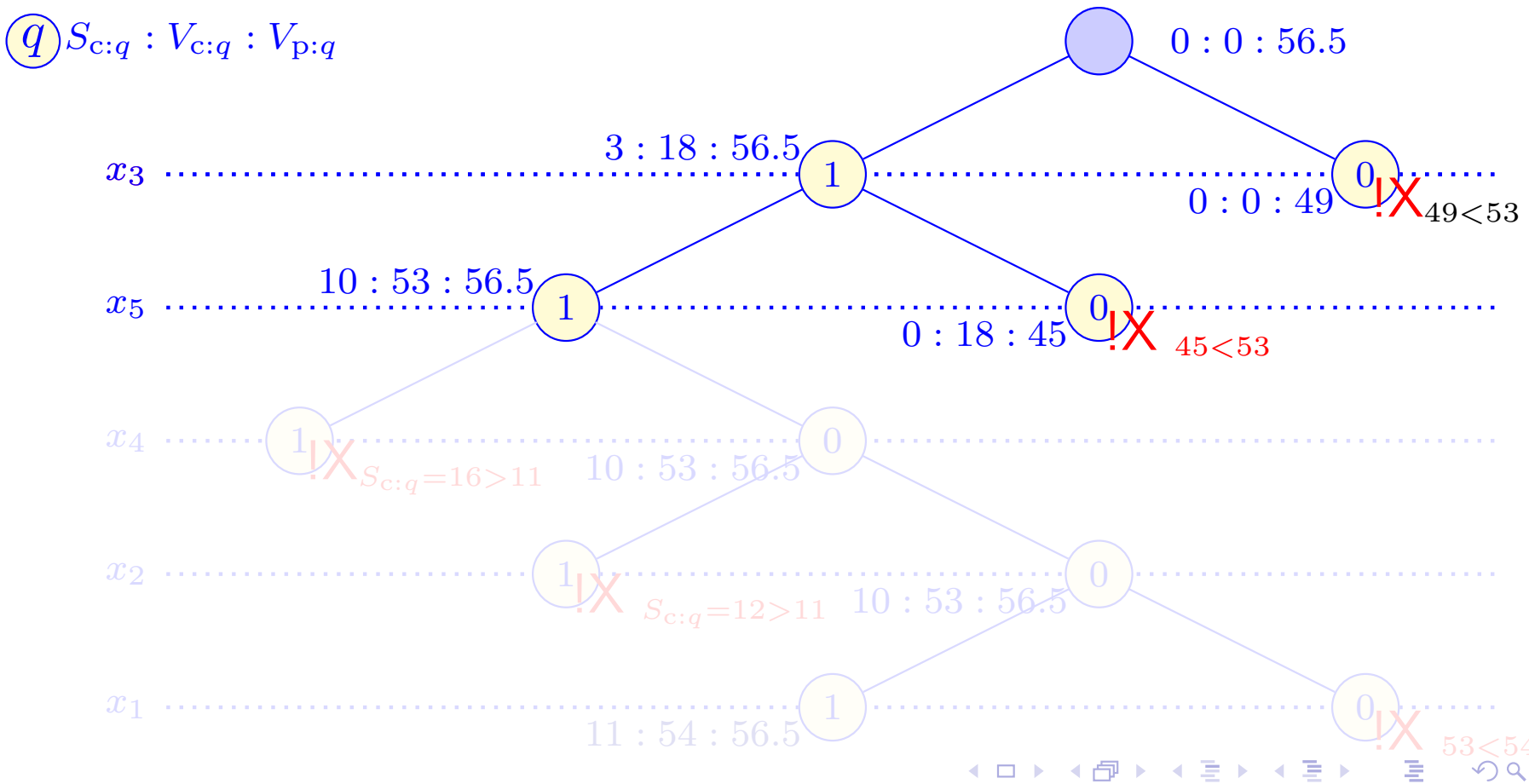
Example: B&B Solution for the $0-1$ Knapsack

i	3_1	5_2	4_3	2_4	1_5
v_i	18	35	21	6	1
s_i	3	7	6	2	1
η_i	6	5	3.5	3	1



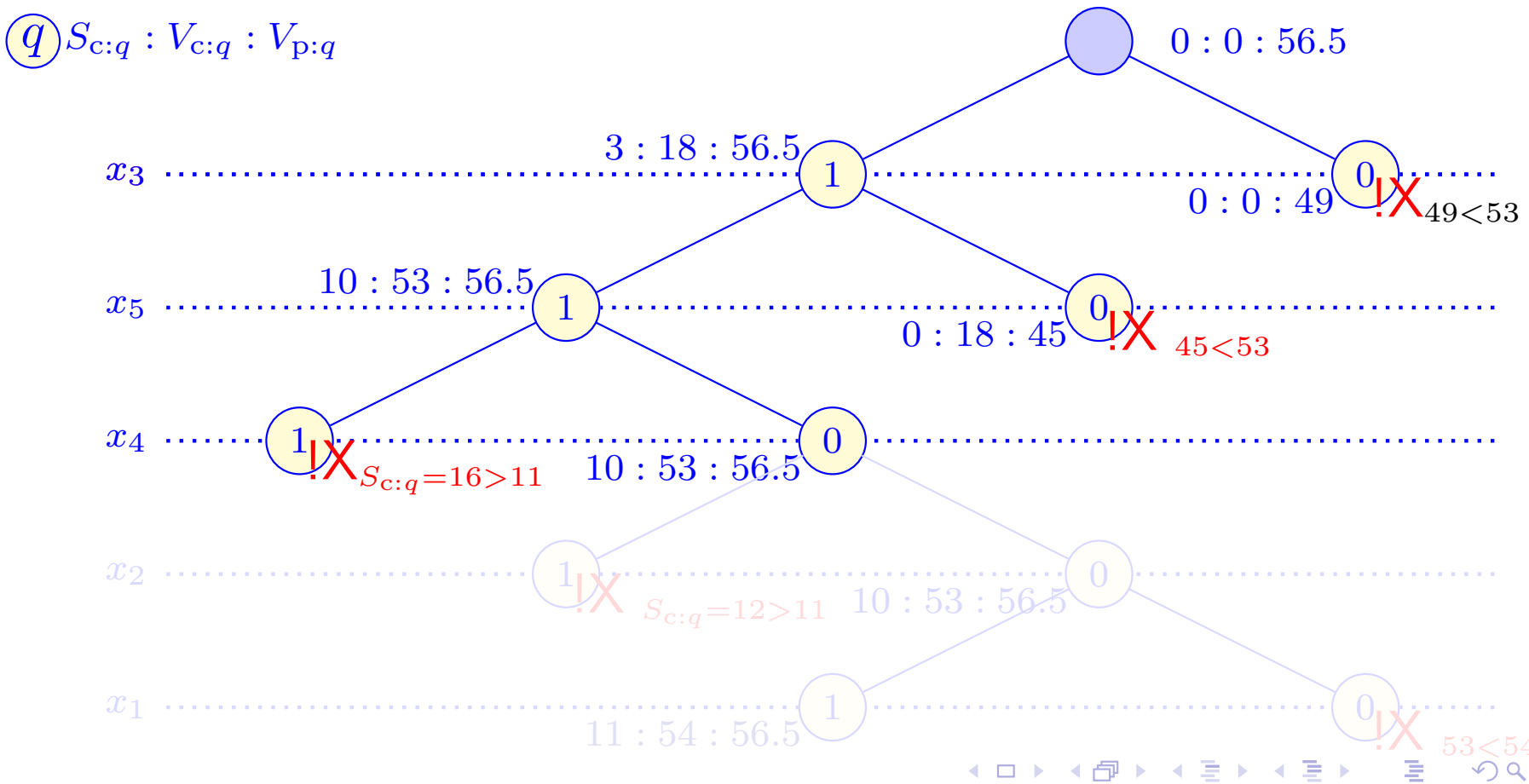
Example: B&B Solution for the $0-1$ Knapsack

i	3_1	5_2	4_3	2_4	1_5
v_i	18	35	21	6	1
s_i	3	7	6	2	1
η_i	6	5	3.5	3	1



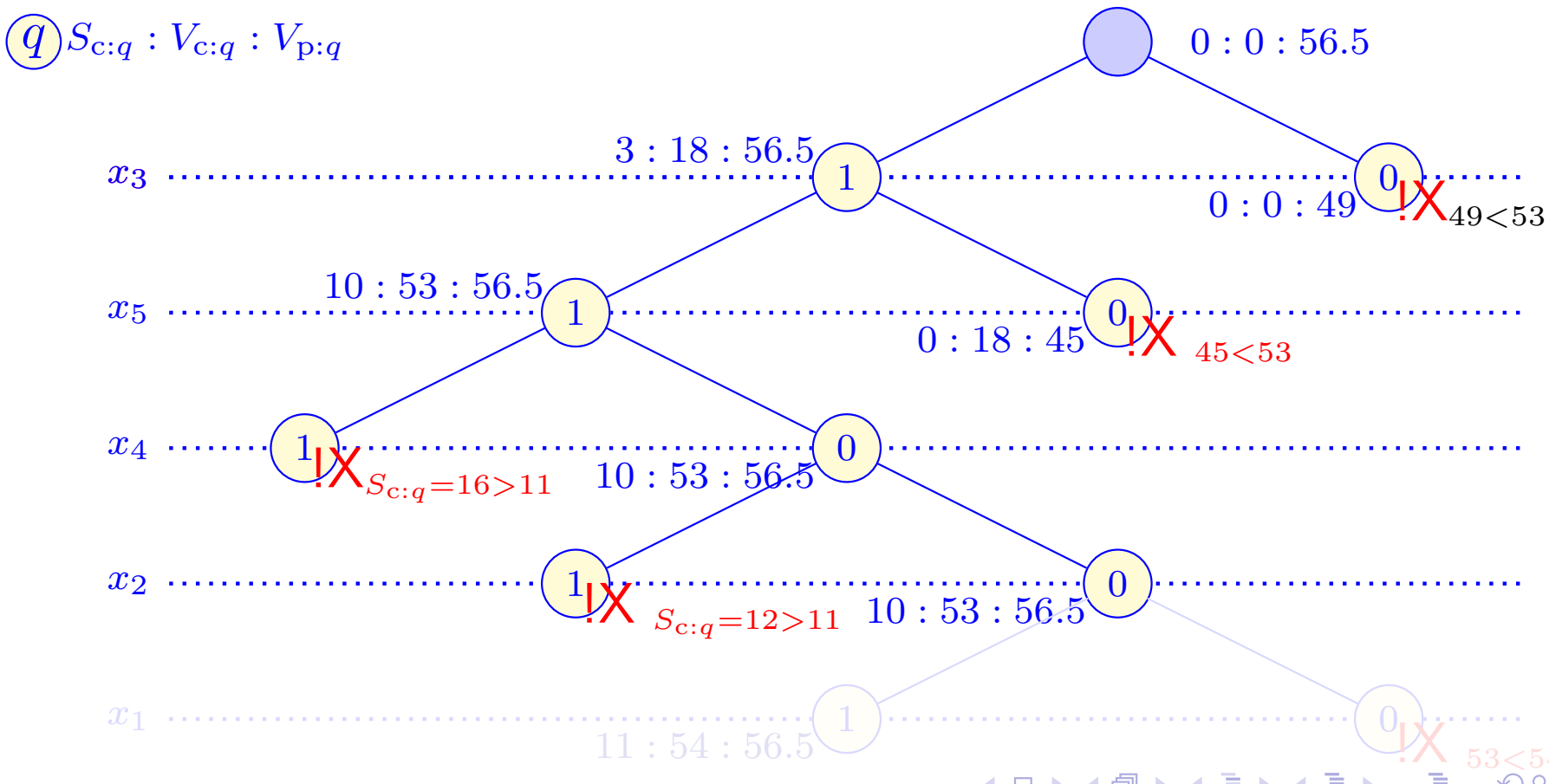
Example: B&B Solution for the $0-1$ Knapsack

i	3_1	5_2	4_3	2_4	1_5
v_i	18	35	21	6	1
s_i	3	7	6	2	1
η_i	6	5	3.5	3	1



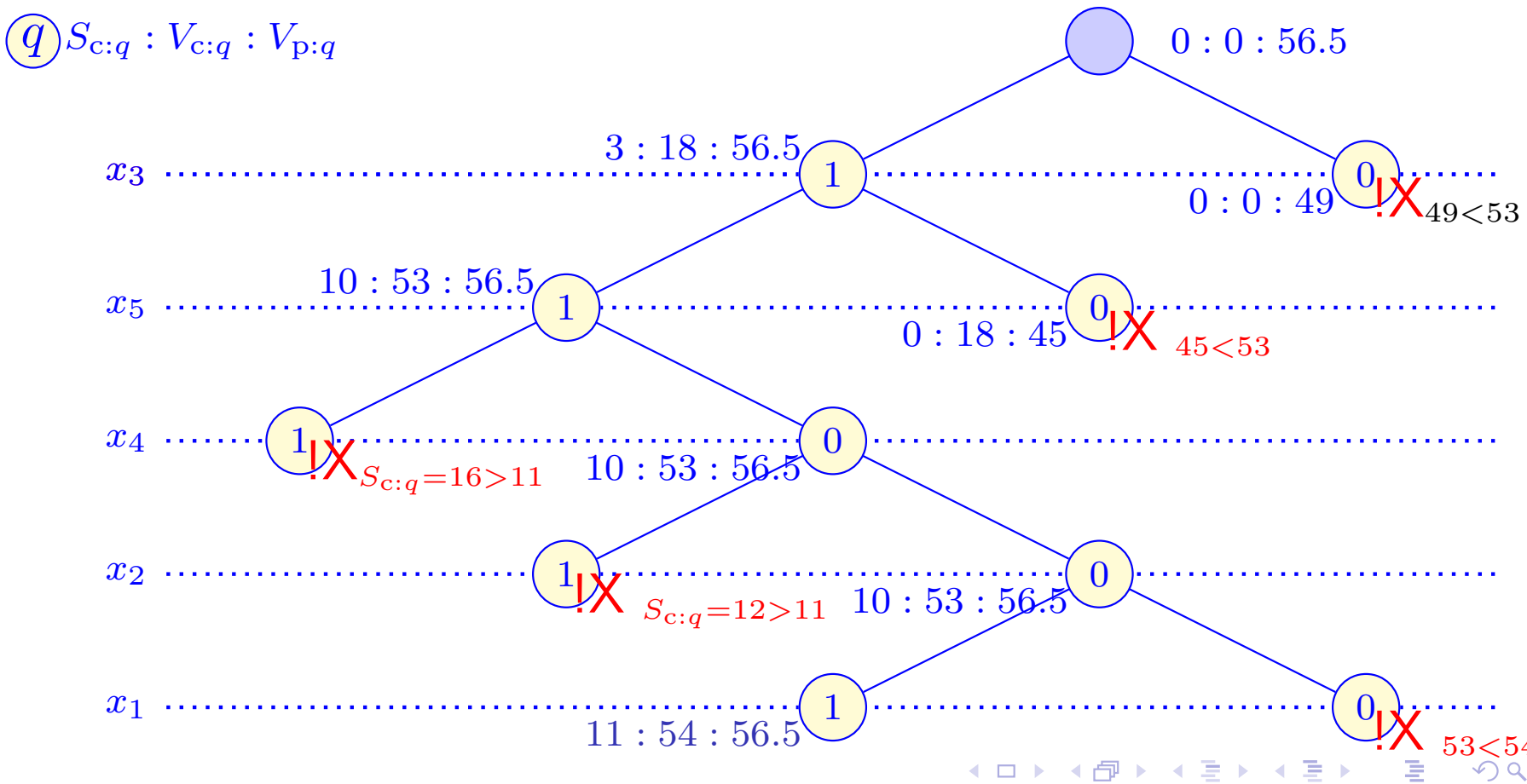
Example: B&B Solution for the $0-1$ Knapsack

i	3_1	5_2	4_3	2_4	1_5
v_i	18	35	21	6	1
s_i	3	7	6	2	1
η_i	6	5	3.5	3	1



Example: B&B Solution for the $0-1$ Knapsack

i	3_1	5_2	4_3	2_4	1_5
v_i	18	35	21	6	1
s_i	3	7	6	2	1
η_i	6	5	3.5	3	1



"0–1" Knapsack Example: B&B Vs. Backtracking

Backtracking uses the DFS with the same state space tree as B&B

- Pruning of non-promising nodes in backtracking: if the search cannot proceed to one of the node's children without exceeding the maximum knapsack size S
- Backtracking in the example: analysing 49 nodes out of 63 nodes
  - 63 tree nodes in total $(2^6 - 1)$
  - Pruning: 14 nodes (by the constraint $\sum_{i=1}^{5} x_i s_i < 11$)
- Branch-and-boundin the example: analysing only 11 nodes out of 63 nodes

Although both algorithms have exponential complexity, the B&B search is faster due to more diverse pruning