Constraint Satisfaction Problems (Backtracking Search)

- Chapter 6
  - 6.1: Formalism
  - 6.2: Constraint Propagation
  - 6.3: Backtracking Search for CSP
  - 6.4 is about local search which is a very useful idea but we won't cover it in class.

Constraint Satisfaction Problems

- As we've discussed search problems, we've designed problem specific state representations for our search routines to use.
- We've also generally been concerned not only to arrive at goal states, but to determine paths for our agents to follow.
- We might, however, care less about paths and more about final goal states or configurations.
- We might also prefer general state representations that can take advantage of specialized search algorithms.
- We call search problems that can be formalized in this specialized way **CSP**s, or Constraint Satisfaction problems.
- CPSs are search problems with a uniform, simple state representation that allows design of more efficient algorithms.
- Techniques for solving CSPs have many practical applications in industry.

Constraint Satisfaction Problems

State Representation:

Factored into variables that can take on values

Algorithms:

General purpose, enforce constraints on factors

Main Idea:

Eliminate chances of search space by identifying value assignments for variables that **violate constraints**.

Representing States with Feature Vectors

- For each problem we have designed a new state representation (and designed the sub-routines called by search based on this representation).
- **Feature vectors** provide a general state representation that is useful for many different problems.
- Feature vectors are also used in many other areas of AI, particularly Machine Learning, Reasoning under Uncertainty, Computer Vision, etc.

Feature Vectors

- **State Representation:** states are vectors of feature values

- We have
  - A set of k **features** (or **variables**)
  - Each variable has a **domain** of different values.
  - A **state** is specified by an assignment of a value for each variable.
    - height = {short, average, tall},
    - weight = {light, average, heavy}
  - A **partial state** is specified by an assignment of a value to some of the variables.

- In CSPs, the problem is to search for a set of values for the features (variables) so that the values satisfy some conditions (constraints).
  - I.e., a goal state specified as conditions on the vector of features values.

Example: Sudoku

	2							
			6					3
	7	4		8				
					3			2
	8			4			1	
6			5					
				1		7	8	
5					9			
							4	

1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

Example: Sudoku

- 81 **variables**, each representing the value of a cell.
- **Domain of Values**: a single value for those cells that are already filled in, the set {1, …9} for those cells that are empty.
- **State:** any completed board given by specifying the value in each cell (1-9, or blank).
- **Partial State:** some incomplete filling out of the board.

Example: 8-Puzzle

2	**3**	**7**
6	4	8
5	1	

- **Variables:** 9 variables $Cell_{1,1}$, $Cell_{1,2}$, …, $Cell_{3,3}$
- **Values:** {'B', 1, 2, …, 8}
- **State:** Each "$Cell_{i,j}$" variable specifies what is in that position of the tile.
  - If we specify a value for each cell we have completely specified a state.

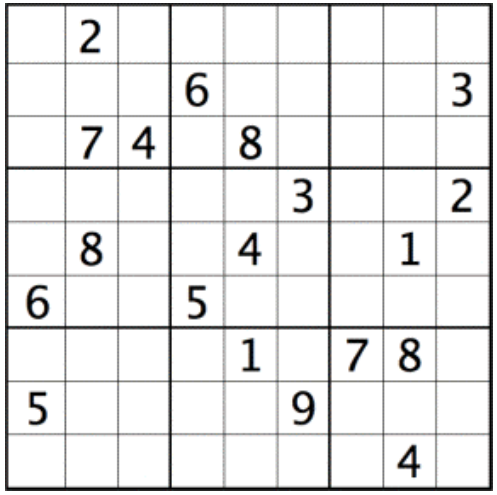
This is only one of many ways to specify the state.

Constraint Satisfaction Problems

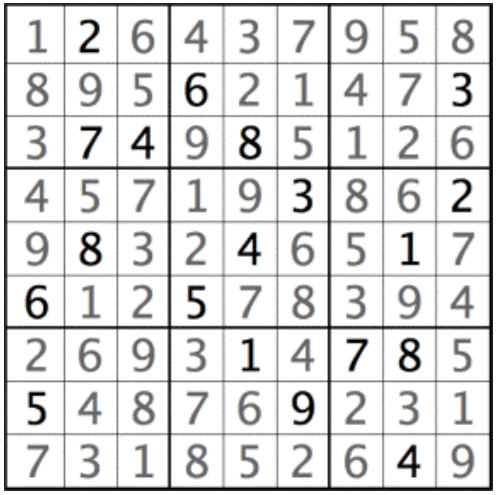
- Notice that in these problems some settings of the variables are **illegal.**
  - In Sudoku, we can't have the same number in any column, row, or subsquare.
  - In the 8 puzzle each variable must have a distinct value (same tile can't be in two places)

Constraint Satisfaction Problems

- In many practical problems finding which setting of the feature variables yields a legal state is difficult.



	2							
			6					3
	7	4		8				
					3			2
	8			4			1	
6			5					
				1		7	8	
5					9			
							4	



1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

- We want to find a state (setting of the variables) that satisfies certain constraints.

Constraint Satisfaction Problems

- In Suduko: The variables that form
  - a column must be distinct
  - a row must be distinct
  - a sub-square must be distinct.

	2							
			6					3
	7	4		8				
					3			2
	8			4			1	
6			5					
				1		7	8	
5					9			
							4	

1	2	6	4	3	7	9	5	8
8	9	5	6	2	1	4	7	3
3	7	4	9	8	5	1	2	6
4	5	7	1	9	3	8	6	2
9	8	3	2	4	6	5	1	7
6	1	2	5	7	8	3	9	4
2	6	9	3	1	4	7	8	5
5	4	8	7	6	9	2	3	1
7	3	1	8	5	2	6	4	9

Constraint Satisfaction Problems

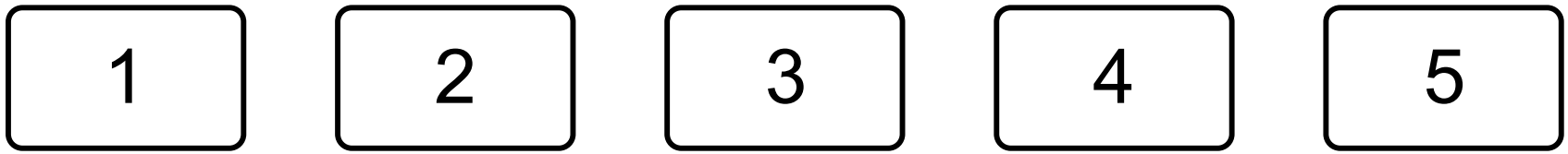
- In these problems we **do not care** about the sequence of moves needed to get to a goal state.
- We only care about finding a feature vector (a setting of the variables) that satisfies the goal.
  - A setting of the variables that satisfies some constraints.

- In contrast, in the 8-puzzle, the feature vector satisfying the goal is given. We care about the sequence of moves needed to move the tiles into that configuration

Example Car Sequencing

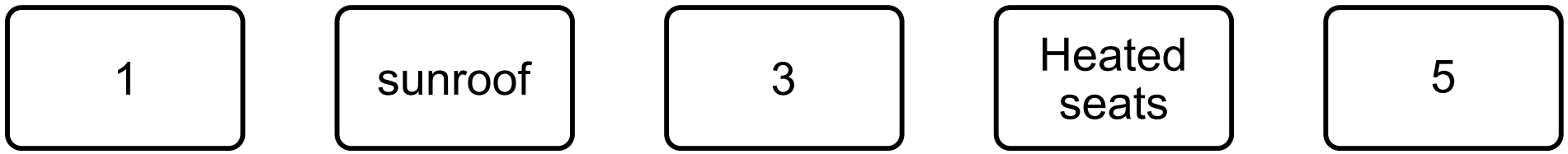
Car Factory Assembly Line—back to the days of Henry Ford

Move the items to be assembled don't move the workers



The assembly line is divided into stations. A particular task is preformed at each station.

Example Car Sequencing



Some stations install optional items…not every car in the assembly line is worked on in that station.

As a result the factory is designed to have lower capacity in those stations.

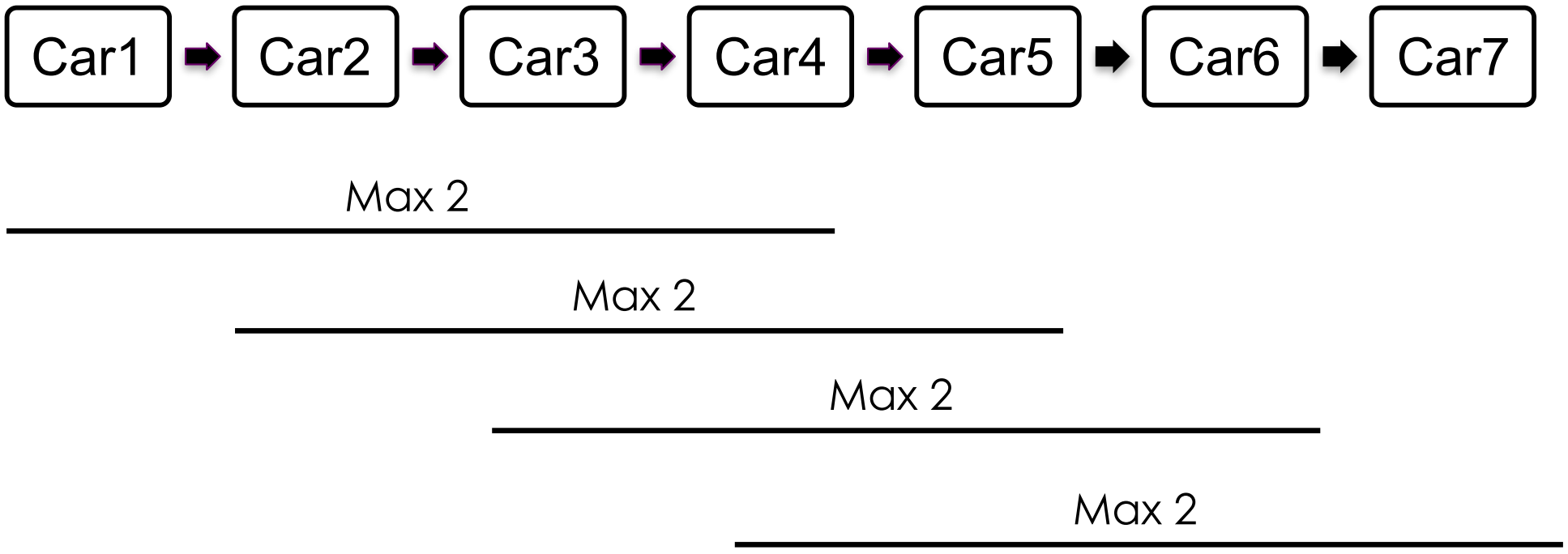
Example Car Sequencing

Cars move through the factory on an assembly line which is broken up into slots.

The stations might be able to process only a limited number of slots out of some group of slots that is passing through the station at any time.

E.g., the sunroof station might accommodate 4 slots, but only has capacity to process 2 slots out of the 4 at any one time.

Example Car Sequencing



Example Car Sequencing

Each car to be assembled has a list of required options.

We want to assign each car to be assembled to a slot on the line.

But we want to ensure that no sequence of 4 slots has more than 2 cars assigned that require a sun roof.

Finding a feasible assignment of cars with different options to slots without violating the capacity constraints of the different stations is hard.

Formalization of a CSP

- A CSP consists of
  - A set of **variables** $V_1, \ldots, V_n$
  - For each variable a (finite) **domain** of possible values $Dom[V_i]$.
  - A set of **constraints** $C_1, \ldots, C_m$.
  - A **solution** to a CSP is an assignment of a value to all of the variables such that ***every constraint is satisfied***.
  - A CSP is unsatisfiable if no solution exists.

Formalization of a CSP

- Each variable can be assigned any value from its domain.
  - Vi = d where d ∈ Dom[Vi]

- Each constraint C
  - Has a set of variables it is over, called its **scope**
    - E.g., C(V1, V2, V4) is a constraint over the variables V1, V2, and V4. Its scope is {V1, V2, V4}
  - Given an assignment to its variables the constraint returns:
    - **True**—this assignment satisfies the constraint
    - **False**—this assignment falsifies the constraint.

Formalization of a CSP

- **Unary** Constraints (over one variable)
  - e.g. C(X):X=2; C(Y): Y>5
- **Binary** Constraints (over two variables)
  - e.g. C(X,Y): X+Y<6
- **Higher-order** constraints: over 3 or more variables.

Formalization of a CSP

- We can specify the constraint with a table
- C(V1, V2, V4) with Dom[V1] = {1,2,3} and Dom[V2] = Dom[V4] = {1, 2}

V1	V2	V4	C(V1,V2,V4)
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	**1**	**1**	**TRUE**
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	**1**	**2**	**TRUE**
3	**2**	**1**	**TRUE**
3	2	2	False

Formalization of a CSP

- Often we can specify the constraint more compactly with an expression:
  C(V1, V2, V4) = (V1 = V2+V4)

V1	V2	V4	C(V1,V2,V4)
1	1	1	False
1	1	2	False
1	2	1	False
1	2	2	False
2	1	1	**TRUE**
2	1	2	False
2	2	1	False
2	2	2	False
3	1	1	False
3	1	2	**TRUE**
3	2	1	**TRUE**
3	2	2	False

Example: Sudoku

- **Variables**: V_{11}, V_{12}, …, V_{21}, V_{22}, …, V_{91}, …, V_{99}
- **Domains**:
  - Dom[V_{ij}] = {1-9} for empty cells
  - Dom[V_{ij}] = {k} a fixed value k for filled cells.
- **Constraints**:
  - Row constraints:
    - All-Diff(V_{11}, V_{12}, V_{13}, …, V_{19})
    - All-Diff(V_{21}, V_{22}, V_{23}, …, V_{29})
    - …., All-Diff(V_{91}, V_{92}, …, V_{99})
  - Column Constraints:
    - All-Diff(V_{11}, V_{21}, V_{31}, …, V_{91})
    - All-Diff(V_{21}, V_{22}, V_{13}, …, V_{92})
    - …., All-Diff(V_{19}, V_{29}, …, V_{99})
  - Sub-Square Constraints:
    - All-Diff(V_{11}, V_{12}, V_{13}, V_{21}, V_{22}, V_{23}, V_{31}, V_{32}, V_{33})
    - All-Diff(V_{14}, V_{15}, V_{16},…, V_{34}, V_{35}, V_{36})

Example: Sudoku

- Each of these constraints is over 9 variables, and they are all the same constraint:
  - Any assignment to these 9 variables such that each variable has a different value satisfies the constraint.
  - Any assignment where two or more variables have the same value falsifies the constraint.

- This is a special kind of constraint called an ALL-DIFF constraint.
  - ALL-Diff(V1, .., Vn) could also be encoded as a set of binary not-equal constraints between all possible pairs of variables: V1 ≠ V2, V1 ≠ V3, …, V2 ≠ V1, …, Vn ≠ V1, …, Vn ≠ Vn-1

Example: Sudoku

- Thus Sudoku has 3x9 ALL-DIFF constraints, one over each set of variables in the same row, one over each set of variables in the same column, and one over each set of variables in the same sub-square.

Solving CSPs

- Because CSPs do not require finding a paths (to a goal), it is best solved by a specialized version of depth-first search.

- Key intuitions:
  - We can build up to a solution by searching through the space of partial assignments.
  - Order in which we assign the variables does not matter – eventually they all have to be assigned. We can decide on a suitable value for one variable at a time!

  ➔ This is the key idea of backtracking search.

  - If we falsify a constraint during the process of building up a solution, we can immediately reject the current partial assignment:
    - All extensions of this partial assignment will falsify that constraint, and thus none can be solutions.

CSP as a Search Problem

A CSP could be viewed as a more traditional search problem

- Initial state: empty assignment
- Successor function: a value is assigned to any unassigned variable, which does not cause any constraint to return false.
- Goal test: the assignment is complete

Backtracking Search: The Algorithm BT

- In the slides that follow, we present a generic backtracking algorithm:
  - We pick a variable, assign it a value, test the constraints that we can. If a constraint is unsatisfied we **backtrack**, otherwise we set another variable. When all variables are set, we're done.
- Later in this unit, we refine this algorithm to include some constraint propagation (inference). More on this later!
- There are clever ways to pick what variable to assign and also what value to assign to it! Can you think of any? Think about what you do when you play Sudoku!

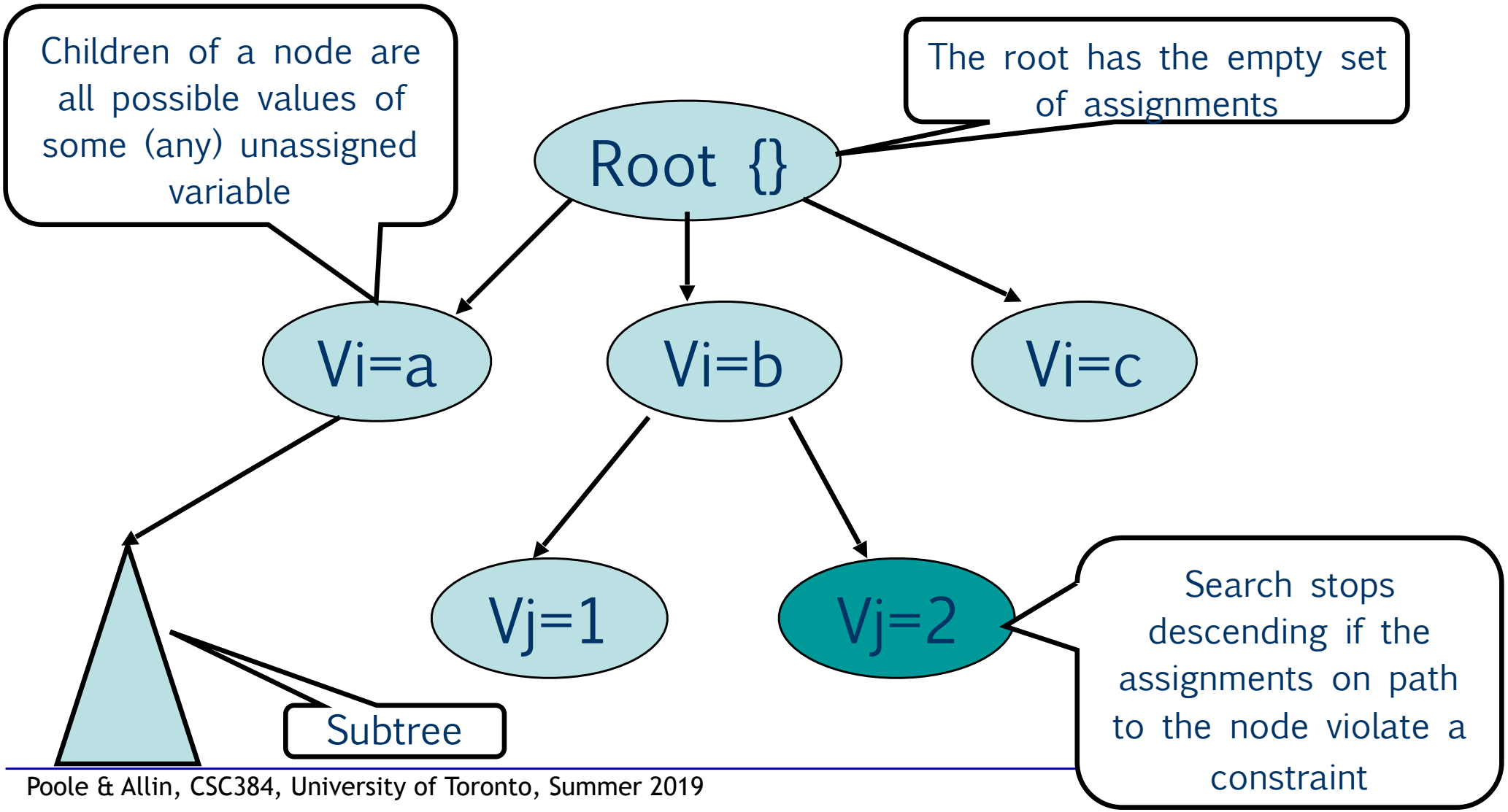
Backtracking Search: The Algorithm BT

```
BT(Level)
  If all variables assigned
       PRINT Value of each Variable
       RETURN or EXIT (RETURN for more solutions)
                      (EXIT for only one solution)
  V := PickUnassignedVariable()
  Assigned[V] := TRUE
  for d := each member of Domain(V) (the domain values of V)
       Value[V] := d
       ConstraintsOK = TRUE
       for each constraint C such that
              a) V is a variable of C and
              b) all other variables of C are assigned:
           IF C is not satisfied by the set of current
              assignments:
              ConstraintsOK = FALSE
       If ConstraintsOk == TRUE:
           BT(Level+1)

  Assigned[V] := FALSE //UNDO as we have tried all of V's values
  return
```

Backtracking Search

- The algorithm searches a tree of partial assignments.



Backtracking Search

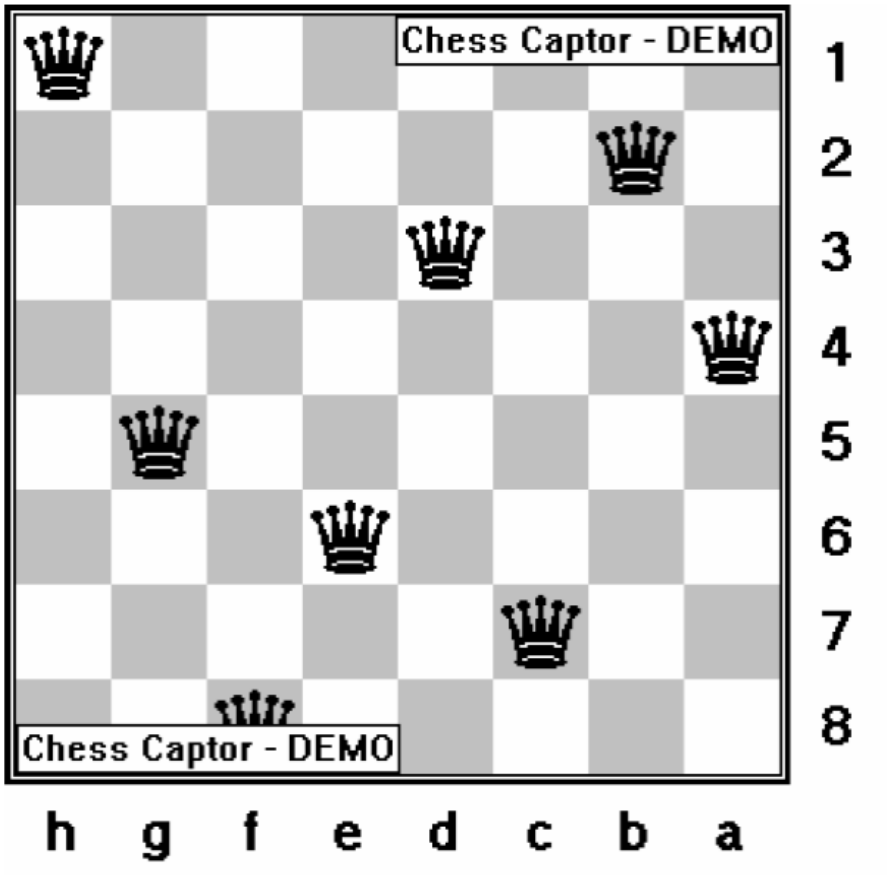
- Heuristics are used to determine
  - the order in which variables are assigned: `PickUnassignedVariable()`
  - the order of values tried for each variable.

- The choice of the next variable can vary from branch to branch, e.g.,
  - under the assignment V1=a we might choose to assign V4 next, while under V1=b we might choose to assign V5 next.

- This "dynamically" chosen variable ordering has a tremendous impact on performance.

Example: N-Queens

- Place N Queens on an N X N chess board so that no Queen can attack any other Queen.

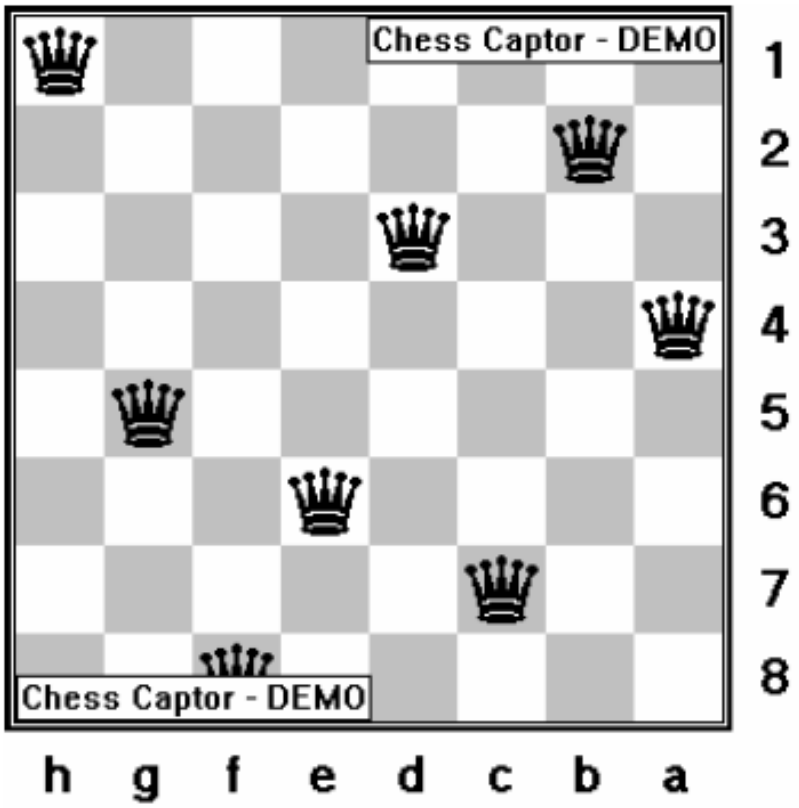


Example: N-Queens

- Problem formulation:
  - N variables (N queens)
  - N^2 values for each variable representing the positions on the chessboard
    - Value i is i'th cell counting from the top left as 1, going left to right, top to bottom.

Example: N-Queens

- Q1 = 1, Q2 = 15, Q3 = 21, Q4 = 32,
  Q5 = 34, Q6 = 44, Q7 = 54, Q8 = 59



Example: N-Queens

- This representation has $(N^2)^N$ states (different possible assignments in the search space)
  - For 8-Queens: $64^8 = 281{,}474{,}976{,}710{,}656$

- Is there a better way to represent the N-queens problem?
  - We know we cannot place two queens in a single row → we can exploit this fact in the choice of the CSP representation

Example: N-Queens

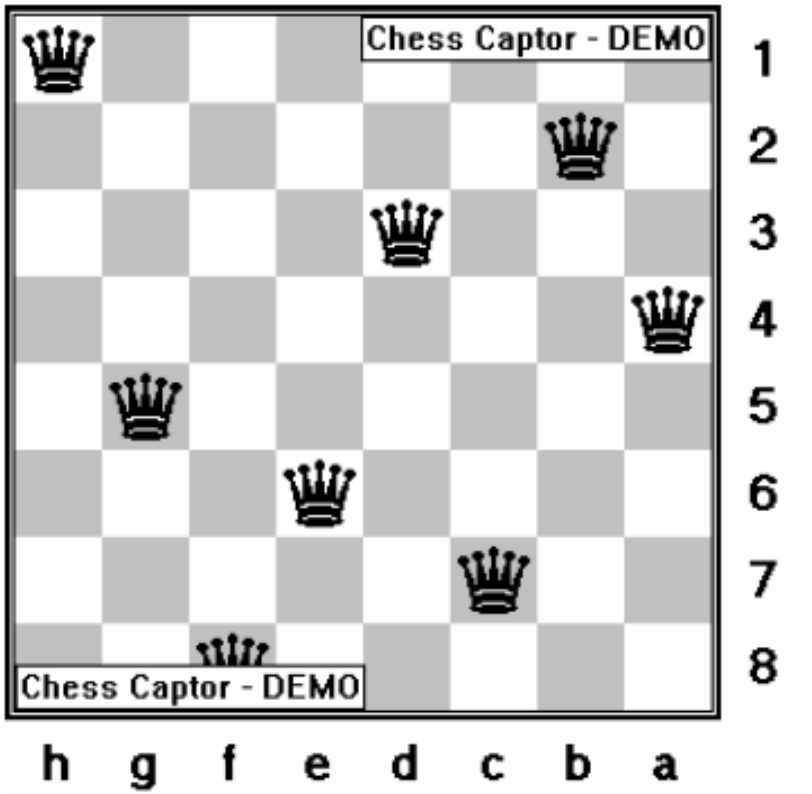
- Better Modeling:
  - N variables Qi, one per row.
  - Value of Qi is the column the Queen in row i is placed; possible values {1, …, N}.

- This representation has N^N states:
  - For 8-Queens: $8^8 = 16{,}777{,}216$

- The choice of a representation can make the problem solvable or unsolvable!

Example: N-Queens

- Q1 = 1, Q2 = 7, Q3 = 5, Q4 = 8,
  Q5 = 2, Q6 = 4, Q7 = 6, Q8 = 3



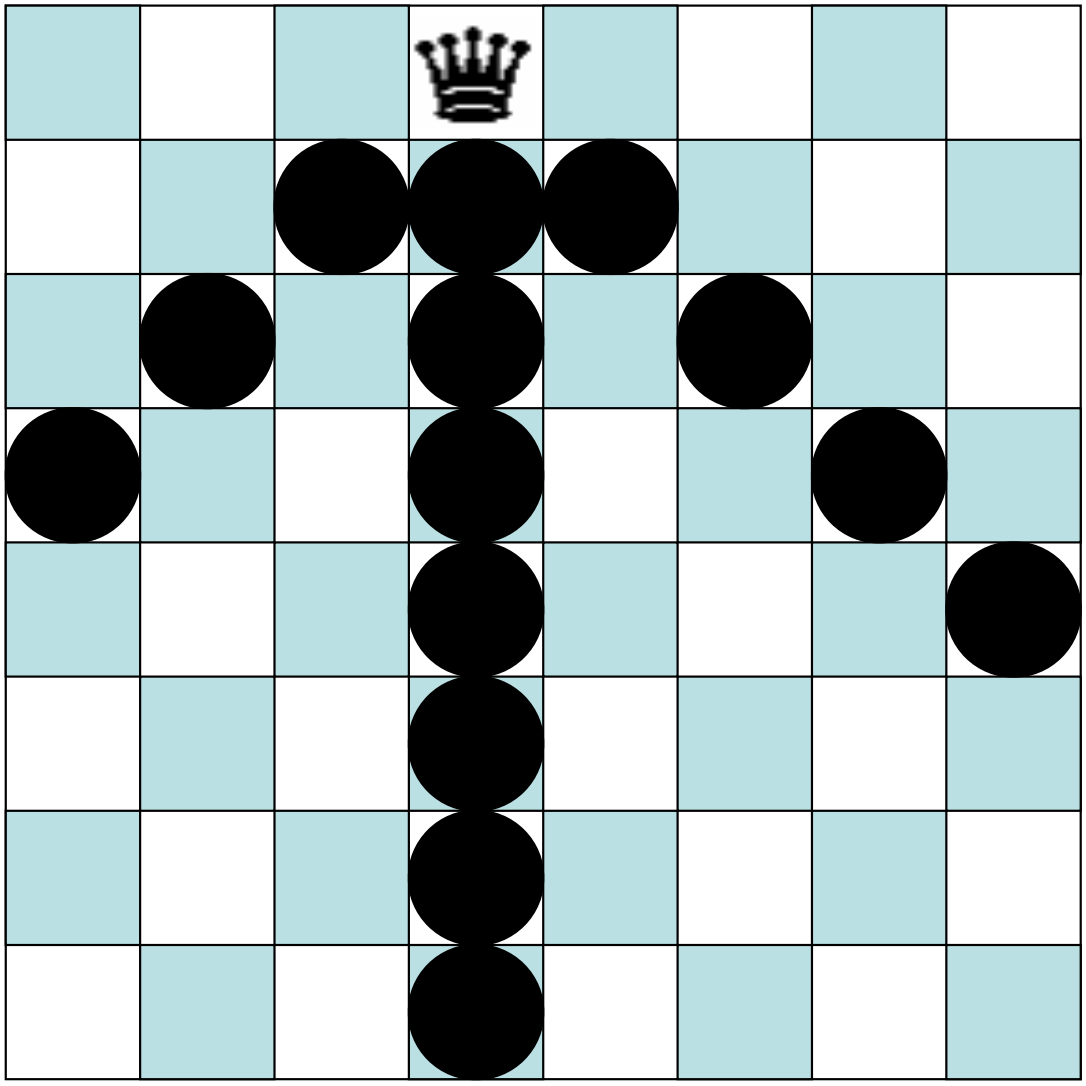
Example: N-Queens

- Constraints:
  - Can't put two Queens in same column

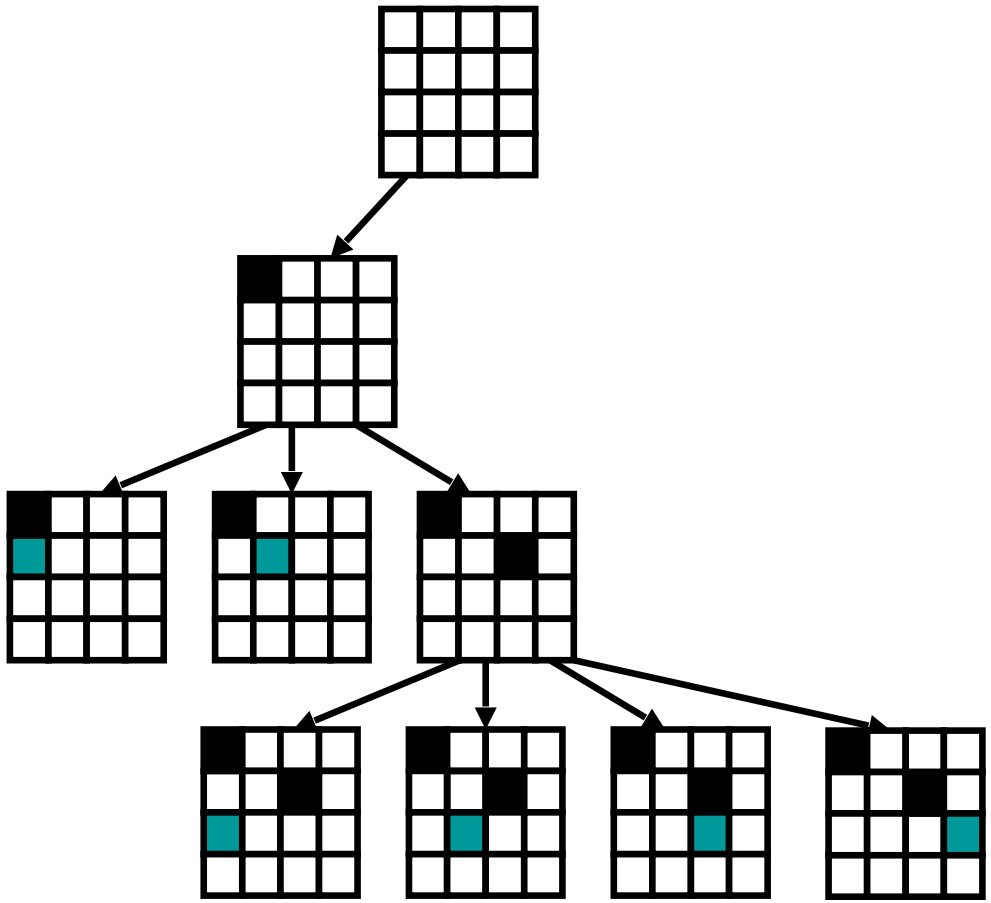
    $Qi \neq Qj$ for all $i \neq j$
  - Diagonal constraints

    abs(Qi-Qj**)** ≠ **abs(**i-j**)**
    - i.e., the difference in the values assigned to Qi and Qj can't be equal to the difference between i and j.

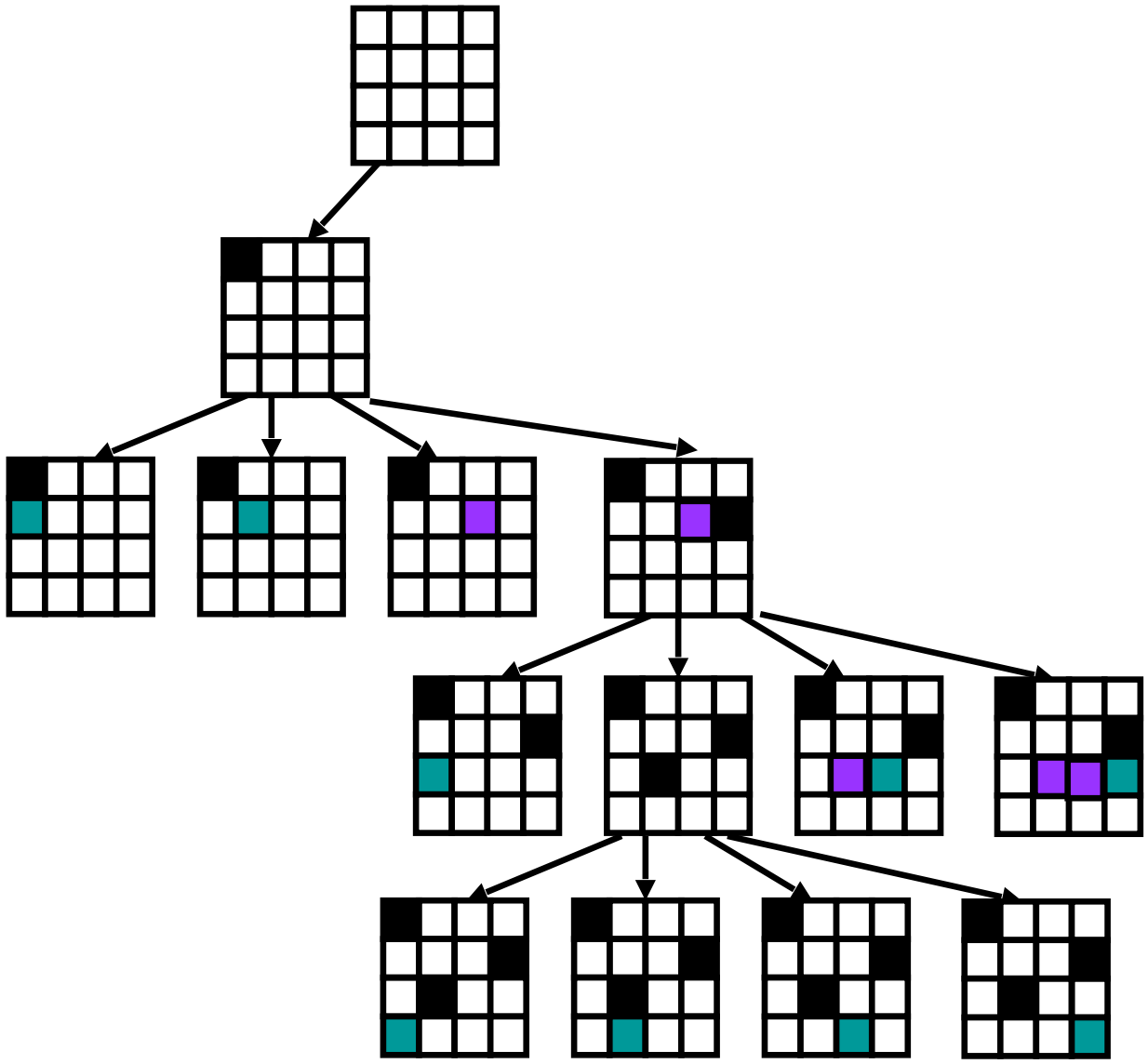
Example: N-Queens



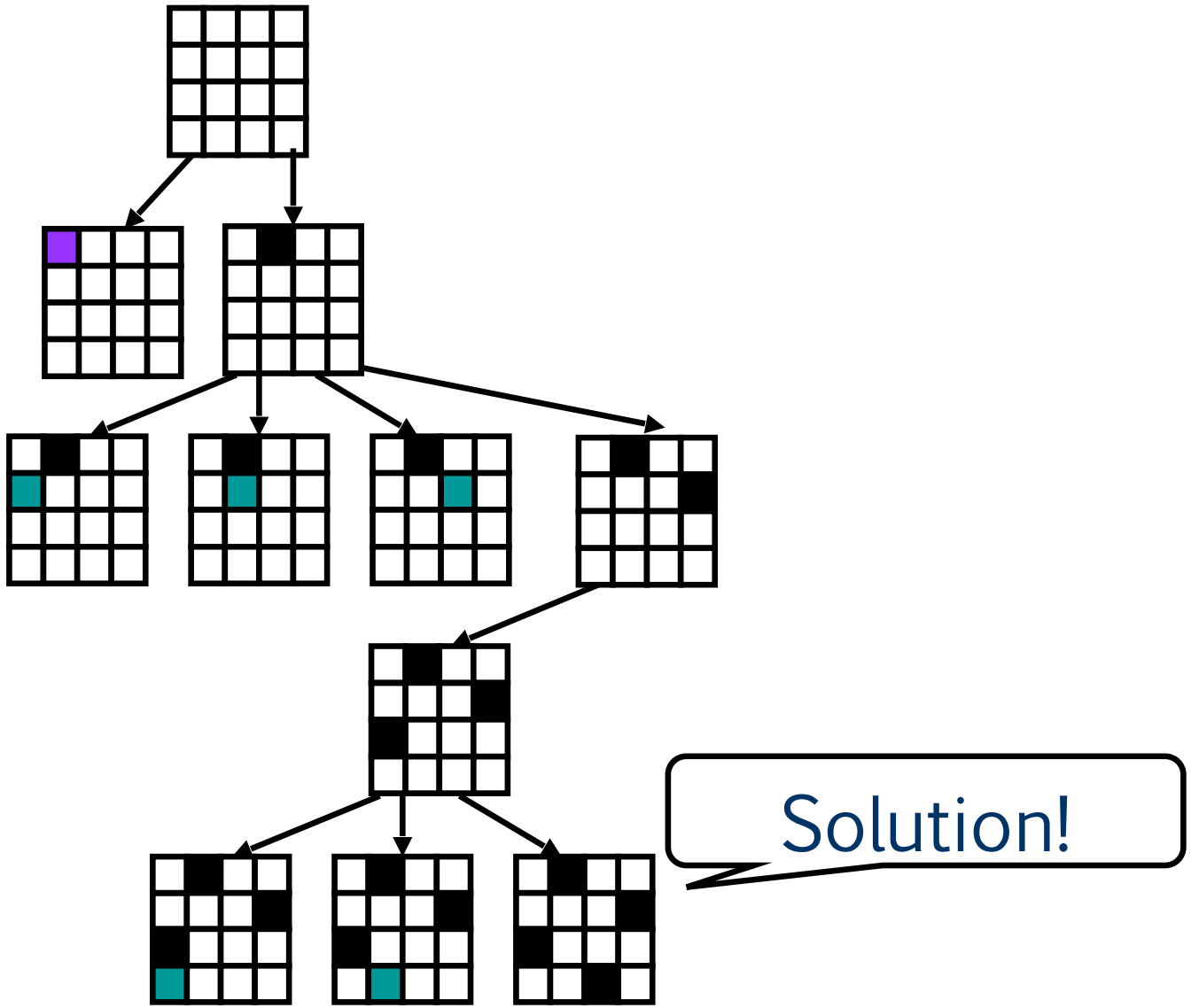
Example: N-Queens



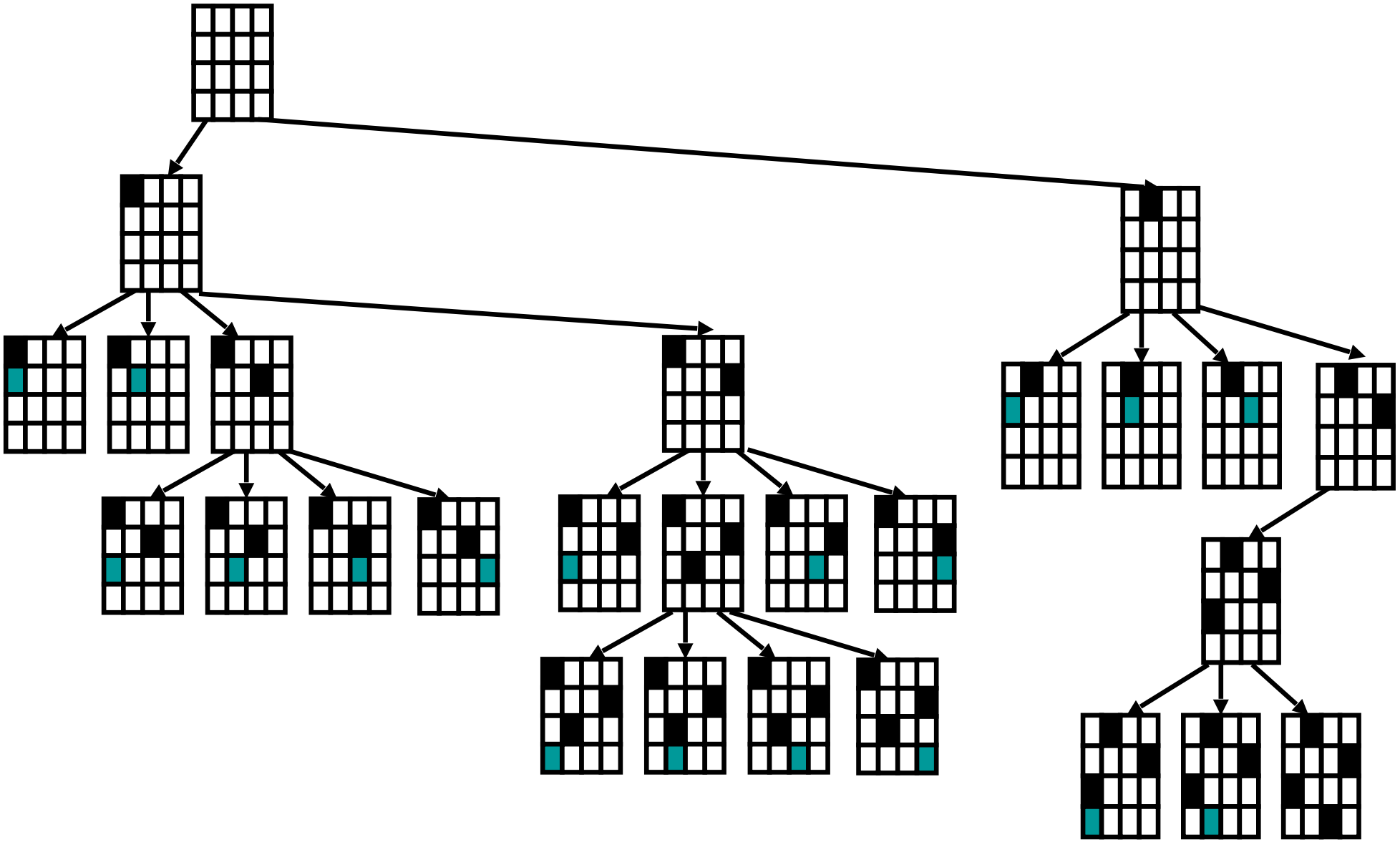
Example: N-Queens



Example: N-Queens

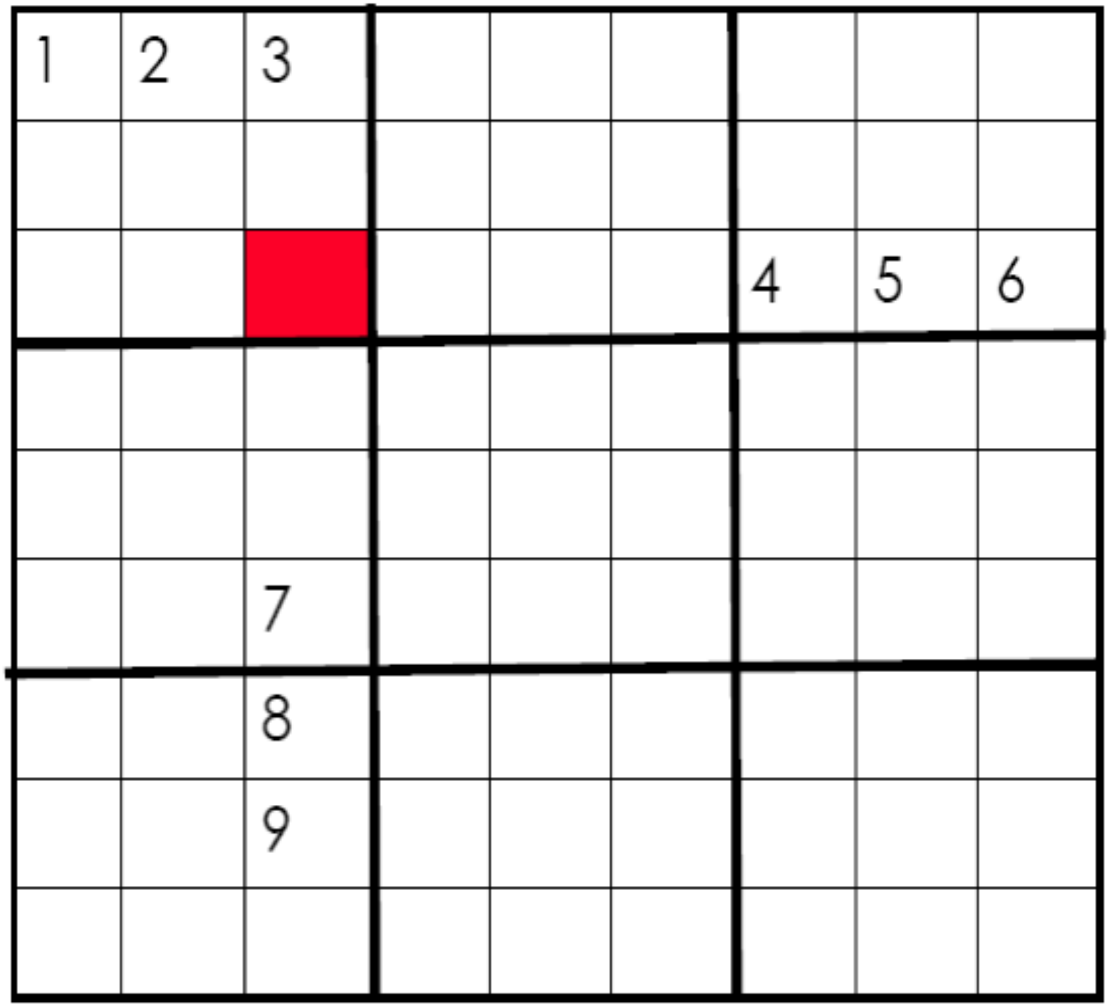


Example: N-Queens Backtracking Search Space



Problems with Plain Backtracking

Sudoku: The 3,3 cell has no possible value.



Problems with Plain Backtracking

- In the backtracking search we won't detect that the (3,3) cell has no possible value until all variables of the row/column (involving row or column 3) or the sub-square constraint (first sub-square) are assigned.
  So we have the following situation: