APPLIED ALGORITHMS

SOLVING PROBLEM BY SEARCHING

Lecture 4: Constraint satisfaction search

T.S Ha Minh Hoang
Th.S Nguyen Minh Anh

Phenikaa University

Last Update: 20th February 2023

Outline

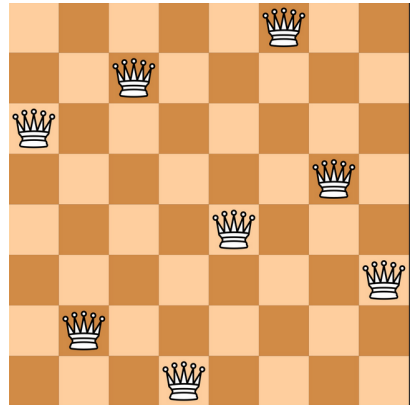
Constraint satisfaction problem (CSP)

Objective:

- **Find a configuration** satisfying goal conditions
- Constraint satisfaction problem (CSP) is a **configuration search problem** where:
  - A **state** is defined by **a set of variables and their domains (possible values)**
  - **Goal condition** is represented by a **set constraints that restrict the values that the variables can take**
- A solution to a CSP is an assignment of all variables such that every constraint is satisfied

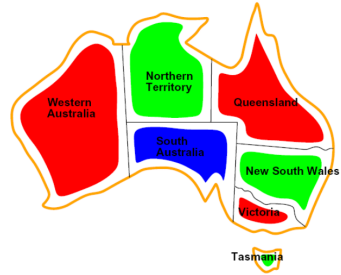
CSP example: N-queens

- **Goal**: n queens placed in non-attacking positions on the board
- **Variable**:
  - Represent queens, one for each column:
    - Q_1, Q_2, Q_3, Q_4
  - Domain (possible values):
    - Row placement of each queen on the board
    $\{1,2,3,4\}$
- **Constraints**:
  - $Q_i \neq Q_j$: Two queens not in the same row
  - $|Q_i - Q_j| \neq |i - j|$: Two queens not on the same diagonal



CSP example: Map coloring

- **Goal**: Color a map using k different colors such that no adjacent countries have the same color
- **Variable**:
  - Represent countries
    - A, B, C, D, E
  - Domain (possible values):
    - k different colors {Red, Blue, Green,...}
- **Constraints**: $A \neq$
  - $Q_i \neq Q_j$: Two queens not in the same row
  - $|Q_i - Q_j| \neq |i - j|$: Two queens not on the same diagonal



CSP example: Sudoku 9x9

- **Goal**: fill the square with numbers from 1 - 9 such that the numbers are different in every row, every column, and every 3x3 grid
- **Variable**:
  - Represent the value of a cell
    - 81 variables
  - Domain (possible values):
    - a fixed value for those cells that are already filled in, the values {1-9} for those cells that are empty
- **Constraints**:
  - no cell in the same column can have the same value.
  - no cell in the same row can have the same value.
  - no cell in the same sub-square can have the same value.

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

CSP example: Timetable Scheduling

Given a set of classes and the teachers and rooms that are available, find a timetable such that no two classes are scheduled at the same time and in the same room.

- **Goal**:
- **Variable**:
  - Represent ...
    - ... variables
  - Domain (possible values):
    - ...
- **Constraints**:
  -

CSP example: Exam Scheduling Problem

- **Goal**: schedule a set of exams E to a time slot and space while satisfying certain constraints (e.g. no student is scheduled to take more than one final at the same time).
- **Variable**: The exams to be scheduled
  - Represent the time slot, the room in which a exam takes place
    - $|E|$ variables
  - Domain (possible values):
    - room id and time slot
- **Constraints**:
  - No two exams can be scheduled at the same time, same place
  - Exams for the same course must be scheduled in a block
  - Student availability (students may have classes or other exams)
  - Availability of rooms and resources

CSP example: Student Course Registration

- **Goal**: Given many classes offered for each course. Pick classes for all N courses.
- **Variable**:
  - Represent the class of a course to be selected
    - N variables
  - Domain (possible values):
    - The set of available classes of a course.
- **Constraints**:
  - Time conflicts: This constraint ensures that students are not scheduled for two classes at the same time.
  - Course preferences: This constraint takes into account the preferences of students for particular courses.

CSP example: Intructor Class Assignment

Given a set of classes of different subjects and a set of instructors. The goal is to assign an instructor to each class such that the following constraints are satisfied:

- No instructor is assigned to more than one class at the same time.
- Each instructor may have a list of classes that they prefer to teach.
- Only assigned to classes that take place during their available time slots.

Outline

Backtracking

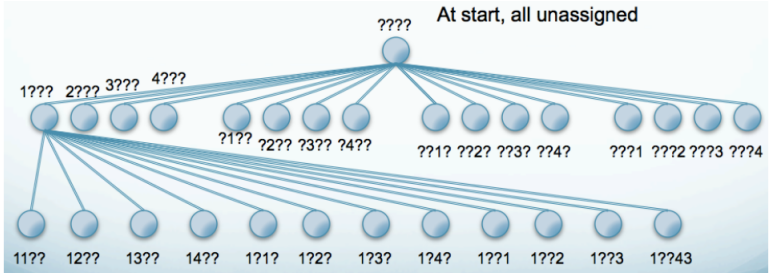
- In the previous module, we spent some time with understanding CSPs from a modeling perspective.
- In this module, we will learn an algorithm to perform inference (i.e., to solve a CSP) based on backtracking search.

Why not just do basic search algorithms from last time?

$n = 4$ variables each taking $d = 4$ values

$b = nd$

$b=(n-1)d$



Generate a search tree of $n!d^n$ but there are only d^n possible assignments!

Commutativity

- The order of assigning the variables has no effect on the final outcome
- CSPs are commutative: Regardless of the assignment order, the same partial solution is reached for a defined set of assignment values
- Don't care about path
- Only a single variable at each node in the search tree needs to be considered (can fix the order)
- d^n number of leaves in the search tree

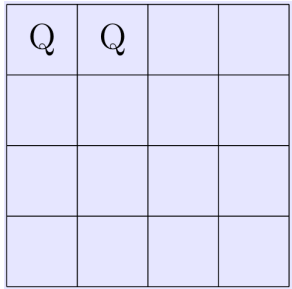
Backtracking: DFS with single variable assignments

- Only consider a single variable at each point of making decision
- Don't care about path
- Order of variable assignment doesn't matter

Constraint checking

Question: When to check the violation of constraints?

- at the end (for the leaf nodes)
- for each node of the search tree during its generation or before its expansion



Constraint checking

- **Check at leaf nodes**
  Backtracking search will treat each state as a **black box**
  - It does not know what is inside each state.
  - Backtracking search can perform two tasks:
    - test whether a state is a goal
    - generate successors of a state
- **Check during node generations**
  - Understand the internal structure of the state.
    - For example, it knows and understands the positions of the queens
  - **Ensure the constraints are satisfied at anypoint**
  - It may be able to prune more states early, then complete the search much more quickly

Question: The trade-off between two methods?

Backtracking

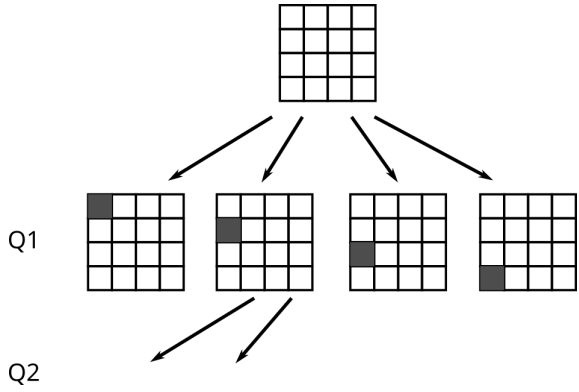
Function Solve()

return **Backtrack**({})

Function Backtrack(`assignment`)

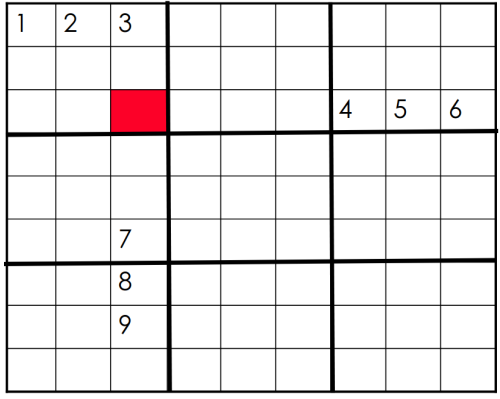
If `assignment` is complete, return `assignment`
$V_i \leftarrow$ pick unassigned variable
For each `value` in domain values of V_i:
  If `value` is consistent with `assignment`:
    Add V_i=`value` to `assignment`
    `result`$\leftarrow$ **Backtrack**(`assignment`)
    If `result`$\neq$ fail, return `result`
  Remove V_i=`value` from `assignment`
return fail

Backtracking: N-Queens Demonstration

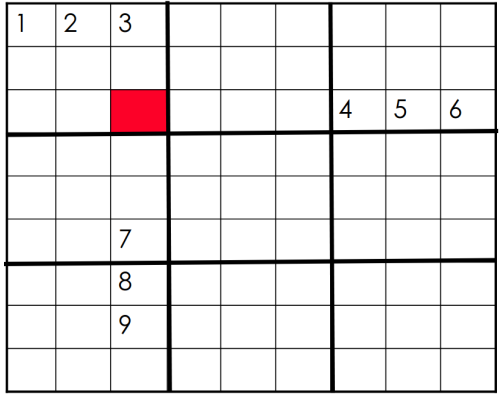


Question: Draw the search process?

The problem with plain/naive backtracking

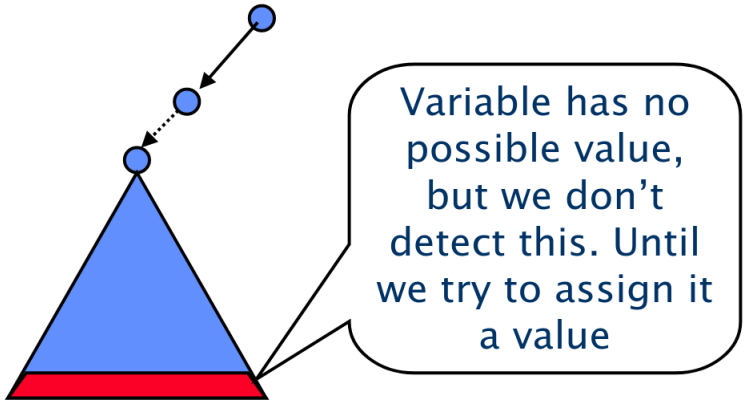


The problem with plain/naive backtracking



The 3x3 cell has no possible value. But in the backtracking search don't detect this until all variables of a row/column or sub-square constraint are assigned. So we have the following situation

The problem with plain/naive backtracking: **Late failure**



Outline

Outline

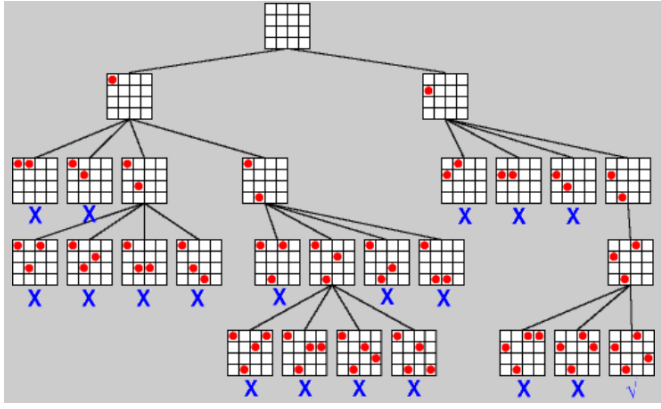
Lookahead: forward checking

Key idea: forward checking (one-step lookahead)

- After assigning a variable X_i, **eliminate inconsistent values** from the domains of unassigned X_i's neighbors.
- If detect **any domain becomes empty**, return fail.

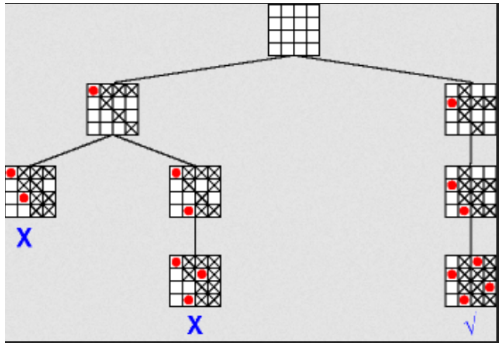
Lookahead: forward checking

From this ...



Lookahead: forward checking

... To this



Forward checking algorithm

FC(`X`, `val`)

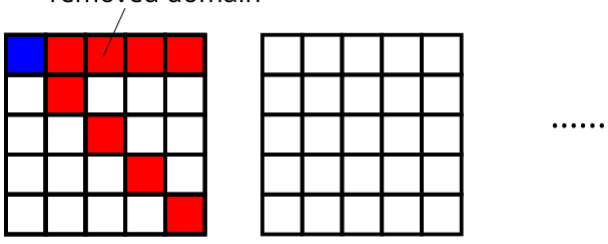
`X` $\rightarrow$ `val`
Update/prune the domains of other unassigned variables accordingly
If **detect empty domain**:
  Restore the previous domains of other unassigned variables
  return `fail`

Restoring Values/Undo your mistakes

- After we backtrack from the current assignment (in the for loop) we must **restore the values that were pruned** as a result of that assignment.
- Some bookkeeping needs to be done, as we must remember which values were pruned by which assignment

Forward checking demo

Question: perform BT with forward checking on the n-queens as follows:



Outline

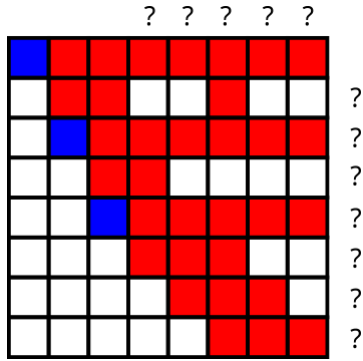
Dynamic Ordering for Backtracking

CSP searches the space in the **depth-first manner**.
Forward Checking also gives us for free two very powerful heuristics by let us choose:

- **Which variable to assign next?**
  $\rightarrow$ **Most constrained variable**: which variable is likely to become a bottleneck?
- **Which value to choose first?**
  $\rightarrow$ **Least constraining value**: which value gives us eaves most choices for the neighboring variables, more flexibility later?

These heuristic tends to produce skinny trees at the top. This means that more variables can be instantiated with fewer nodes searched, and thus more constraint propagation failures occur with less work.

Dynamic Ordering: N-queens



Question: **which column** to place the next queen? at **which row**?

Empirically

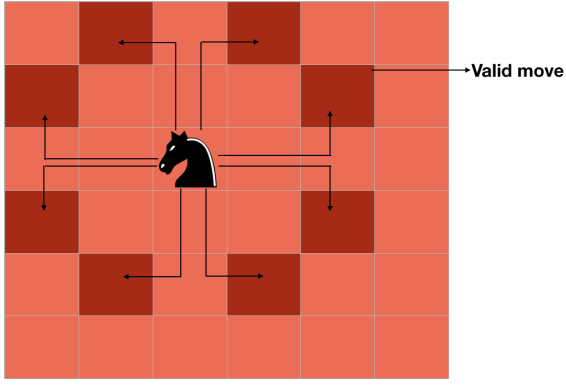
- FC often is about 100 times faster than BT
- FC with MRV (minimal remaining values) often 10000 times faster.
- But on some problems the speed up can be much greater
  - Converts problems that are not solvable to problems that are solvable.

Summary

- We have presented backtracking search for finding the maximum weight assignment in a CSP.
- Given a partial assignment, we first choose an unassigned variable X_i. For this, we use the **most constrained variable (MCV)** heuristic, which chooses the variable with the smallest domain.
- Next we order the values of X_i using the **least constrained value (LCV)** heuristic, which chooses the value that constrains the neighbors of X_i the least.
- Then we perform lookahead (forward checking) to prune down the domains, so that MCV and LCV can work on the latest information.
- Finally, we recurse with the new partial assignment.
- All of these heuristics aren't guaranteed to speed up backtracking search, but can often make a big difference in practice.

Practice Problems: Knight's Tour Problem

A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square exactly once.



1	48	31	50	33	16	63	18
30	51	46	3	62	19	14	35
47	2	49	32	15	34	17	64
52	29	4	45	20	61	36	13
5	44	25	56	9	40	21	60
28	53	8	41	24	57	12	37
43	6	55	26	39	10	59	22
54	27	42	7	58	23	38	11

Class Scheduling

Define this problem as CSP:

- 4 more required classes to graduate
  A: Algorithms B: Bayesian Learning C: Computer Programming D: Distributed Computing
- A few restrictions
  - Algorithms must be taken same semester as Distributed computing
  - Computer programming is a prereq for Distributed computing and Bayesian learning, so it must be taken in an earlier semester
  - Advanced algorithms and Bayesian Learning are always offered at the same time, so they cannot be taken the same semester
- 3 semesters (semester 1,2,3) when can take classes

Outline

Search for the optimal configuration

Constrain satisfaction problem

- **Objective**: find a configuration that satisfies all constraints

Optimal configuration problem

- **Objective**: find the best configuration that optimize (min/max) an **objective function**
- The **objective function**: reflects our preference towards each configuration (or state)

The search space

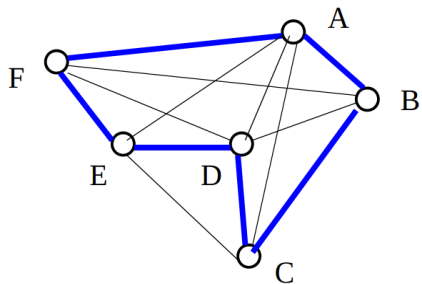
If the search space of configurations is

- **Discrete** or **finite**
  - then it is a **combinatorial optimization problem**
- **Continuous**
  - then it is a **parametric/continous optimization problem**

Example: Traveling salesman problem

Problem:

- A graph with distances
- Find tour - a path that visits every city once and returns to the start (e.g. ABCDEF)
- Goal: the shortest tour



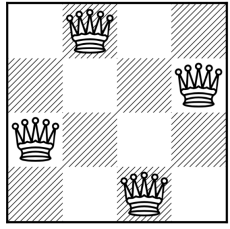
Example: Knapsack Problem

Problem:

- Given precious n items, each associated with a value
- Pick the items into your bag which has a limitation
- Goal: maximizes the overall value of items in your bag

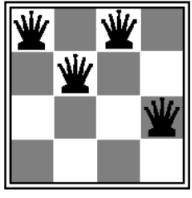
The N-Queens

- A CSP problem
- Is it possible to formulate the problem as an optimal configuration search problem ?

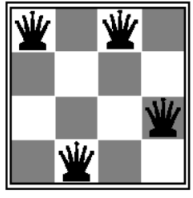


The N-Queens

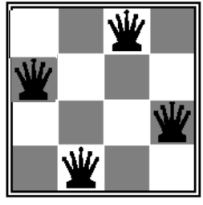
- A CSP problem
- Is it possible to formulate the problem as an optimal configuration search problem? **Yes**
- The **quality of a configuration in a CSP** can be measured by **the number of violated constraints**
- **Solving**: minimize the number of constraint violations



# of violations =3



# of violations =1



# of violations =0

In the next lecture

We are going to discuss on the methods for solving **optimization problems**