Artificial Intelligence: Foundations & Applications

Solving Constraint Satisfaction Problem

Prof. Partha P. Chakrabarti & Arijit Mondal
Indian Institute of Technology Kharagpur

Solution overview

- **CSP graph creation**
  - **Create a** *node* **for** *every variable*. **All possible** *domain values* **are initially** *assigned* **to the variable**
  - **Draw** *edges* **between nodes if there is a** *binary Constraint*. **Otherwise draw a** *hyper-edge* **between nodes with constraints involving more than two variables**
- **Constraint propagation**
  - **Reduce the** *valid domains* **of each** *variable* **by applying node consistency, arc / edge Consistency, K-Consistency, till no further reduction is possible. If a solution is found or the problem found to have no consistent solution, then terminate**
- **Search for solution**
  - **Apply** *search algorithms* **to find solutions**
  - **There are interesting properties of CSP graphs which lead of efficient algorithms in some cases:** *Trees, Perfect Graphs, Interval Graphs, etc.*
  - **Issues for Search: Backtracking Scheme, Ordering of Children, Forward Checking (Look-Ahead) using Dynamic Constraint Propagation**
  - **Solving by converting to** *satisfiability (SAT)* **problems**

Search formulation of CSP

- **Standard *search* formulation of CSP**
  - **Initial state: all unassigned variables**
  - **State: partial assignment of the variables**
  - **Successor function: assign a value to unassigned variables**
  - **Goal state: all variables are assigned and satisfies all constraints**
  - **Path cost: uniform path cost**

Constraint propagation

- Constraints
  - Unary constraints or node constraints (eg. $x_i \neq 9$)
  - Binary constraints or edge between nodes (eg. $x_i \neq x_j$)
  - Higher order or hyper-edge between nodes (eg. $x_1 + x_2 = x_3$)
- Node consistency
  - For every variable V_i, remove all elements of D_i that do not satisfy the unary constraints for the variable
  - First step is to reduce the domains using node consistency
- Arc consistency
  - For every element x_{ij} of D_i, for every edge from V_i to V_j, remove x_{ij} if it has no consistent value(s) in other domains satisfying the Constraints
  - Continue to iterate using arc consistency till no further reduction happens.
- Path consistency
  - For every element y_{ij} of D_i, choose a path of length L with L variables, use a consistency checking method similar to above to reduce domains if possible

Arc consistency check (AC-3)

AC-3(*csp*) // inputs - CSP with variables, domains, constraints

1. *queue* ← local variable initialized to all arcs in csp
2. **while** *queue* is not empty **do**
3. $(X_i, X_j) \leftarrow$ pop(queue)
4. **if** Revise(*csp*, X_i, X_j) **then**
5. **if** size of D_i = 0 **then return** *false*
6. **for each** X_k **in** X_i.neighbors-$\{X_j\}$ **do**
7. add (X_k, X_i) to *queue*
8. **return** *true*

Revise(*csp*, X_i, X_j)

1. *revised* ← *false*
2. **for each** x **in** D_i **do**
3. **if** no value y in D_j allows (x, y) to satisfy constraint between X_i and X_j **then**
4. delete x from D_i
5. *revised* ← *true*
6. **return** *revised*

Arc consistency check (AC-3)

AC-3(*csp*) // inputs - CSP with variables, domains, constraints

1. *queue* ← local variable initialized to all arcs in csp
2. **while** *queue* is not empty **do**
3. (X_i, X_j) ← pop(queue)
4. **if** Revise(*csp*, X_i, X_j) **then**
5. **if** size of D_i = 0 **then return** *false*
6. **for each** X_k **in** X_i.neighbors-{X_j} **do**
7. add (X_k, X_i) to *queue*
8. **return** *true*

Revise(*csp*, X_i, X_j)

1. *revised* ← *false*
2. **for each** x **in** D_i **do**
3. **if** no value y in D_j allows (x, y) to satisfy constraint between X_i and X_j **then**
4. delete x from D_i
5. *revised* ← *true*
6. **return** *revised*

Complexity?

AC-3 example

- Variables: A, B, C, D
- Domain: $\{1, 2, 3\}$
- Constraints: $A \neq B$, $C < B$, $C < D$

AC-3 example

- Variables: A, B, C, D
- Domain: {1, 2, 3}
- Constraints: $A \neq B$, $C < B$, $C < D$

queue: AB, BA, BC, CB, CD, DC

AC-3 example

- Variables: A, B, C, D
- Domain: $\{1, 2, 3\}$
- Constraints: $A \neq B$, $C < B$, $C < D$

queue: AB, BA, BC, CB, CD, DC
pop(queue) // AB

AC-3 example

- Variables: A, B, C, D
- Domain: $\{1, 2, 3\}$
- Constraints: $A \neq B$, $C < B$, $C < D$

queue: AB, BA, BC, CB, CD, DC
pop(queue) // AB
No change in queue. queue=BA, BC, CB, CD, DC

AC-3 example

- Variables: A, B, C, D
- Domain: $\{1, 2, 3\}$
- Constraints: $A \neq B$, $C < B$, $C < D$

queue: AB, BA, BC, CB, CD, DC
pop(queue) // AB
No change in queue. queue=BA, BC, CB, CD, DC
pop(queue) // BA

AC-3 example

- Variables: A, B, C, D
- Domain: $\{1, 2, 3\}$
- Constraints: $A \neq B$, $C < B$, $C < D$

queue: AB, BA, BC, CB, CD, DC

pop(queue) // AB

No change in queue. queue=BA, BC, CB, CD, DC

pop(queue) // BA

No change in queue. queue=BC, CB, CD, DC

AC-3 example

- Variables: A, B, C, D
- Domain: $\{1, 2, 3\}$
- Constraints: $A \neq B$, $C < B$, $C < D$

queue: AB, BA, BC, CB, CD, DC
pop(queue) // AB
No change in queue. queue=BA, BC, CB, CD, DC
pop(queue) // BA
No change in queue. queue=BC, CB, CD, DC
pop(queue) // BC

AC-3 example

- Variables: A, B, C, D
- Domain: $\{1, 2, 3\}$
- Constraints: $A \neq B$, $C < B$, $C < D$

queue: AB, BA, BC, CB, CD, DC
pop(queue) // AB
No change in queue. queue=BA, BC, CB, CD, DC
pop(queue) // BA
No change in queue. queue=BC, CB, CD, DC
pop(queue) // BC
Remove 1. $D_B = \{2, 3\}$

AC-3 example

- Variables: A, B, C, D
- Domain: $\{1, 2, 3\}$
- Constraints: $A \neq B$, $C < B$, $C < D$

queue: AB, BA, BC, CB, CD, DC
pop(queue) // AB
No change in queue. queue=BA, BC, CB, CD, DC
pop(queue) // BA
No change in queue. queue=BC, CB, CD, DC
pop(queue) // BC
Remove 1. $D_B = \{2, 3\}$
Add AB to queue. queue=CB, CD, DC, AB
pop(queue) // CB
Remove 3. $D_C = \{1, 2\}$
No change in queue. queue=CD, DC, AB
pop(queue) // CD
No change. queue=DC, AB
pop(queue) // DC
Remove 1. $D_D = \{2, 3\}$
No change. queue=AB
pop(queue) // AB
No change in queue. queue=$\emptyset$

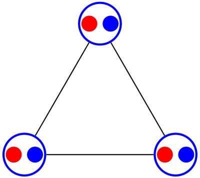
$A = \{1, 2, 3\}, B = \{2, 3\},$
$C = \{1, 2\}, D = \{2, 3\}.$

Sudoku

		3		2		6		
9			3		5			1
		1	8		6	4		
		8	1		2	9		
7					*x*			8
		6	7		8	2		
		2	6		9	5		
8			2		3			9
		5		1	*y*	3		

AC-3 limitations

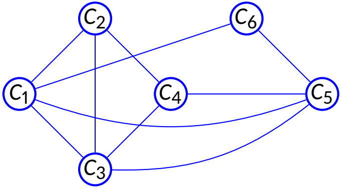
- After successful run of AC-3
  - There can be only one solution
  - There can be more than one solutions
  - There may be no solution and it fails to identify



Examination schedule

Student	Subjects
S_1	C_1, C_2, C_3
S_2	C_2, C_3, C_4
S_3	C_3, C_4
S_4	C_3, C_4, C_5
S_5	C_1, C_5, C_6

Is it possible to conduct all these exams in 3 days assuming one exam per day?



- **How does naive BFS & DFS perform?**

Backtracking search

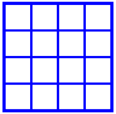
- Backtracking is a basic search methodology for solving CSP
- Basic steps:
  - Assign one variable at a time
    - Fix ordering of variables (eg. $C_1 = 1, C_2 = 3$ is same as $C_2 = 3, C_1 = 1$)
  - Check constraint
    - Check with previously assigned variables

Backtracking search

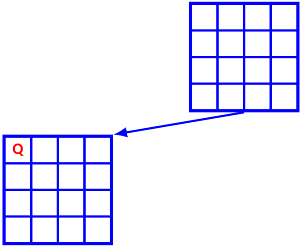
```
Backtrack(assignment)
    if assignment is complete then return success, assignment
    var ← Choose-unassigned-variable()
    for each value of Domain(var) do
        if value is consistent with the assignment then
            add var = value to assignment
            result = Backtrack(assignment)
            if result ≠ failure return result, assignment
    return failure
```

- **Choices:**
  - **Variable to be assigned next**
  - **Value to be assigned to the variable next**
  - **Early detection of failure**

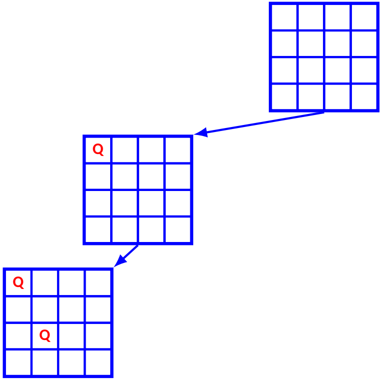
4 Queens



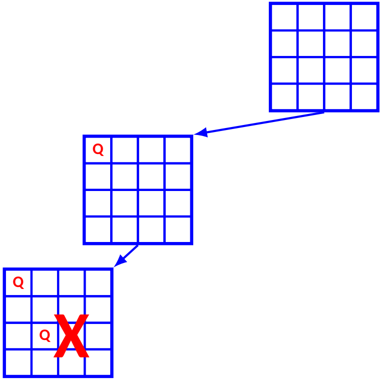
4 Queens



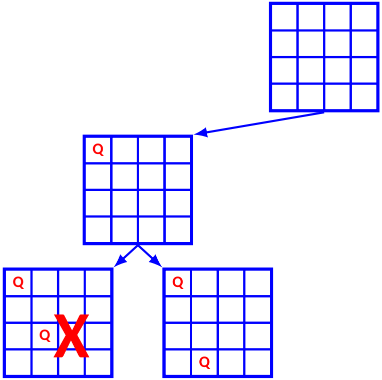
4 Queens



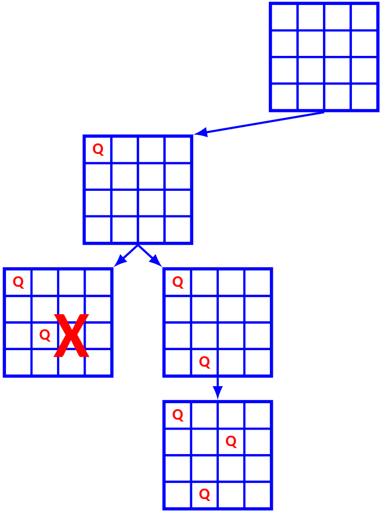
4 Queens



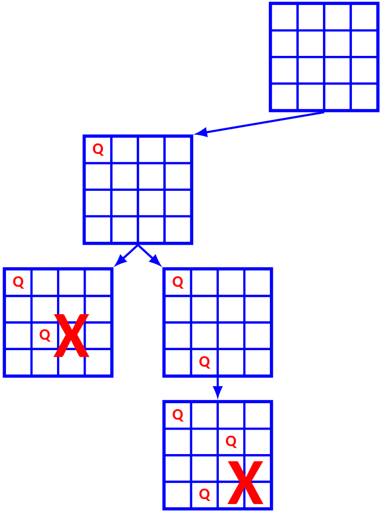
4 Queens



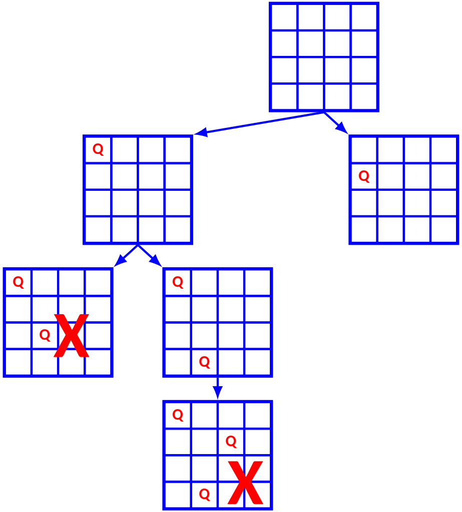
4 Queens



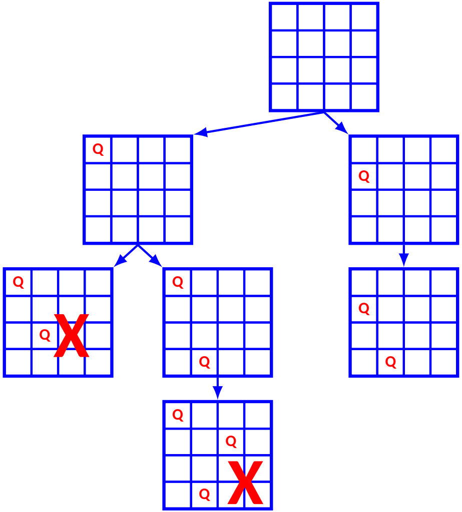
4 Queens



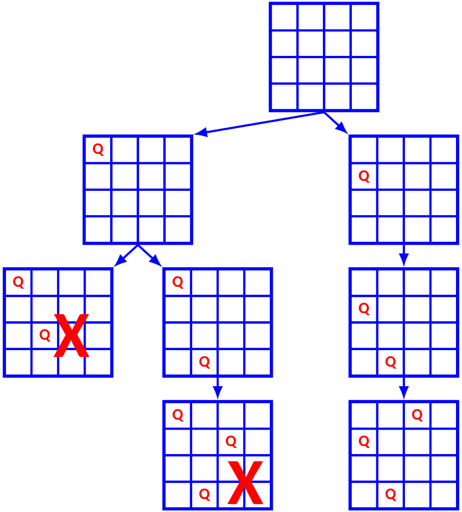
4 Queens



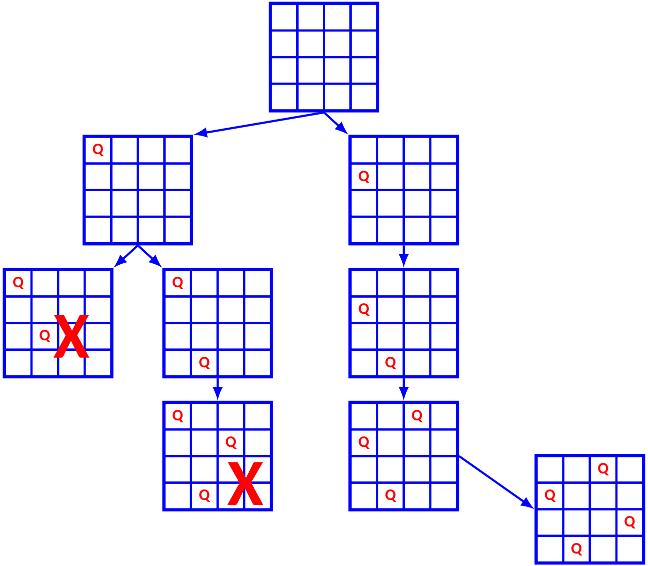
4 Queens



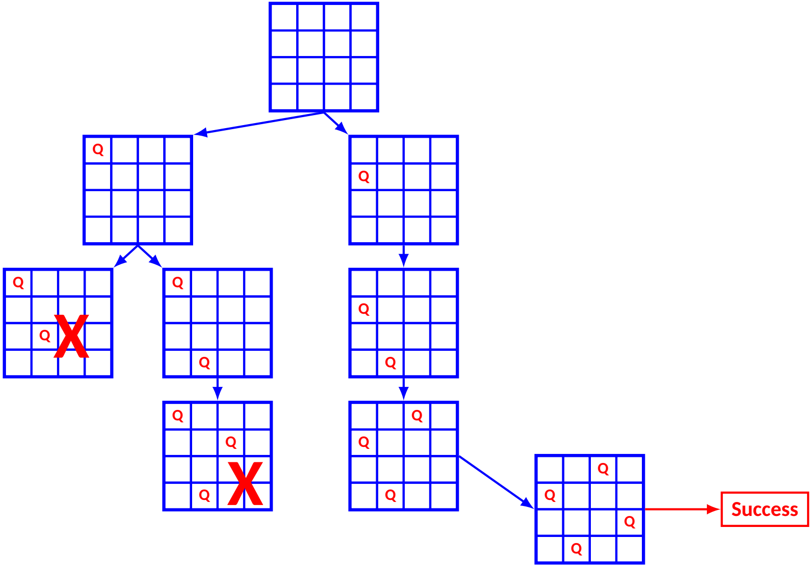
4 Queens



4 Queens



4 Queens

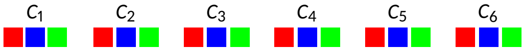


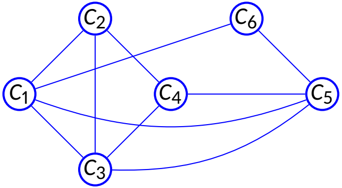
Heuristic strategy

- Variable ordering
  - Static or random
  - Minimum remaining values
    - Variable with fewest legal values (also known as most constrained variable)
  - Degree heuristic
    - Variable with the largest number of constraints on other unassigned variables
- Choice of value
  - Least constraining value
    - Value that leaves most choices for the neighboring variables in the constraint graph

Forward checking

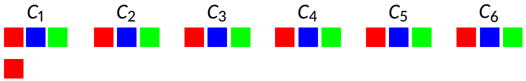
- **Forward checking propagates information from assigned to unassigned variables**

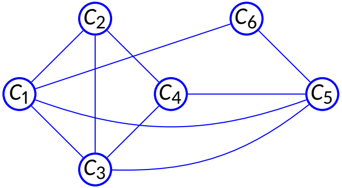




Forward checking

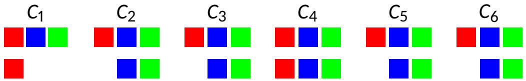
- **Forward checking propagates information from assigned to unassigned variables**

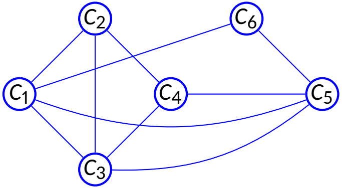




Forward checking

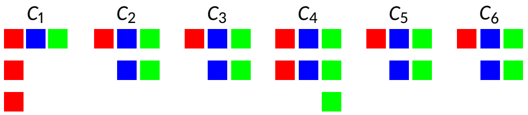
- **Forward checking propagates information from assigned to unassigned variables**

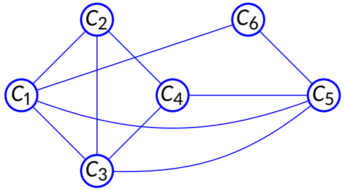




Forward checking

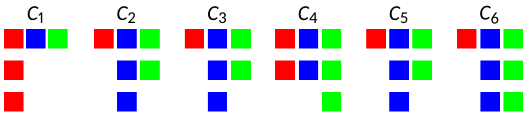
- **Forward checking propagates information from assigned to unassigned variables**

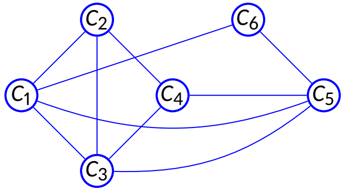




Forward checking

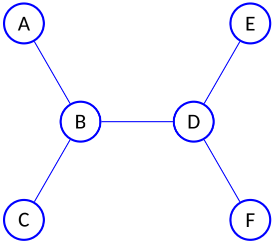
- **Forward checking propagates information from assigned to unassigned variables**





Special cases

- **General CSP problem is NP-Complete**
- **For** *perfect graphs*, *chordal graphs*, *interval graphs*, **the graph coloring problem can be solved in polynomial time**
- **Tree structured CSP can be solved in polynomial time**



Thank you!