Chapter 5

Search: Constraint Satisfaction Problems

Introduction to Artificial Intelligence

Chandra Gummaluru
University of Toronto

Version W22.1

- The following is based on material developed by many individuals, including (but not limited to):
    - Sheila McIlraith
    - Fahiem Bacchus
    - Sonya Allin
    - Craig Boutilier
    - Hojjat Ghaderi
    - Rich Zemel
    - Elliot Creager

- In many search problems, we only care about finding a goal state, and not a path to get there. Such problems are called **constraint satisfaction problems** (CSPs).

- In many search problems, we only care about finding a goal state, and not a path to get there. Such problems are called **constraint satisfaction problems** (CSPs).
- **Example**: The N-Queens Puzzle versus the Slider Puzzle
  - In the N-Queens puzzle, we want to find the goal, but do not care how to get there.
  - In the Slider puzzle, we already know the goal, but want to find to get there.

- In many search problems, we only care about finding a goal state, and not a path to get there. Such problems are called **constraint satisfaction problems** (CSPs).

- **Example**: The N-Queens Puzzle versus the Slider Puzzle

  - In the N-Queens puzzle, we want to find the goal, but do not care how to get there.
  - In the Slider puzzle, we already know the goal, but want to find to get there.

- The approaches discussed thus far still work, but are inefficient for several reasons.

- In many search problems, we only care about finding a goal state, and not a path to get there. Such problems are called **constraint satisfaction problems** (CSPs).
- **Example**: The N-Queens Puzzle versus the Slider Puzzle
  - In the N-Queens puzzle, we want to find the goal, but do not care how to get there.
  - In the Slider puzzle, we already know the goal, but want to find to get there.
- The approaches discussed thus far still work, but are inefficient for several reasons.
- To see why, we need to formalized the notion of a CSP.

- In particular, we define a CSP as a search problem in which:

- In particular, we define a CSP as a search problem in which:
  - the states can be defined through a fixed set of variables, $\mathcal{V} = \left\{ V_1, \ldots, V_{|\mathcal{V}|} \right\}$.

- In particular, we define a CSP as a search problem in which:
  - the states can be defined through a fixed set of variables, $\mathcal{V} = \{ V_1, \ldots, V_{|\mathcal{V}|} \}$.
  - the goals can be defined through a fixed set of constraints, $\mathcal{C} = \{ C_1, \ldots, C_{|\mathcal{C}|} \}$.

- Each state, $s \in \mathcal{S}$, is represented by assigning each variable, $V \in \mathcal{V}$, a unique value, $v \in \mathrm{dom}\, V$, which we represent using a set $\{V = v, V \in \mathcal{V}\}$.

- Each state, $s \in \mathcal{S}$, is represented by assigning each variable, $V \in \mathcal{V}$, a unique value, $v \in \operatorname{dom} V$, which we represent using a set $\{V = v, V \in \mathcal{V}\}$.

- The set of all possible assignments is

$$\operatorname{dom}\mathcal{V} := \prod_{V \in \mathcal{V}} \operatorname{dom} V.$$

- Each state, $s \in \mathcal{S}$, is represented by assigning each variable, $V \in \mathcal{V}$, a unique value, $v \in \mathrm{dom}\, V$, which we represent using a set $\{V = v, V \in \mathcal{V}\}$.

- The set of all possible assignments is

$$\mathrm{dom}\mathcal{V} := \prod_{V \in \mathcal{V}} \mathrm{dom}\, V.$$

- Every state must be representable as an assignment of the variables, but not every assignment needs to represent a state, i.e., $\mathcal{S}$ is a subset of $\mathrm{dom}\mathcal{V}$.

- Each constraint, $C \in \mathcal{C}$, operates on a fixed subset of the variables, $\mathcal{V}_C \subseteq \mathcal{V}$, called its **scope**.

- Each constraint, $C \in \mathcal{C}$, operates on a fixed subset of the variables, $\mathcal{V}_C \subseteq \mathcal{V}$, called its **scope**.
- Given an assignment, $\{V = v, V \in \mathcal{V}_C\}$, the constraint returns either `true` or `false`, indicating whether it was satisfied or not.

- Each constraint, $C \in \mathcal{C}$, operates on a fixed subset of the variables, $\mathcal{V}_C \subseteq \mathcal{V}$, called its **scope**.
- Given an assignment, $\{V = v, V \in \mathcal{V}_C\}$, the constraint returns either `true` or `false`, indicating whether it was satisfied or not.
- Clearly, we need not necessarily assign every variable in $\mathcal{V}$ to check a particular constraint. Assignments where at least one variable is unassigned is called a **partial** assignment.

- Each constraint, $C \in \mathcal{C}$, operates on a fixed subset of the variables, $\mathcal{V}_C \subseteq \mathcal{V}$, called its **scope**.
- Given an assignment, $\{V = v, V \in \mathcal{V}_C\}$, the constraint returns either `true` or `false`, indicating whether it was satisfied or not.
- Clearly, we need not necessarily assign every variable in $\mathcal{V}$ to check a particular constraint. Assignments where at least one variable is unassigned is called a **partial** assignment.
- As we will see later, the ability to form partial assignments is critical to developing efficient algorithms for CSPs.

- **Example** The N-Queens Puzzle as a CSP

- **Example** The N-Queens Puzzle as a CSP
    - Assume there is exactly one queen per row. Let $V_r \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$ denote the column in which the queen on the r^{th} row is located.

- **Example** The N-Queens Puzzle as a CSP
  - Assume there is exactly one queen per row. Let $V_r \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$ denote the column in which the queen on the r^{th} row is located.
  - The constraints are:

- **Example** The N-Queens Puzzle as a CSP
  - Assume there is exactly one queen per row. Let $V_r \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$ denote the column in which the queen on the r^{th} row is located.
  - The constraints are:
    - **row/column constraints**: $V_i \neq V_j, \forall i \neq j$

- **Example** The N-Queens Puzzle as a CSP
  - Assume there is exactly one queen per row. Let $V_r \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$ denote the column in which the queen on the r^{th} row is located.
  - The constraints are:
    - **row/column constraints**: $V_i \neq V_j, \forall i \neq j$
    - **diagonal constraints**: $|n(V_i) - n(V_j)| \neq |i - j|, \forall i \neq j$, where $n(V)$ denotes the alphabetic position of V (e.g., $n(b) = 2$).

- We can now express the CSP in terms of a GSP.

- We can now express the CSP in terms of a GSP.
  - The initial state is some arbitrary assignment, $\left\{V = v^{(0)}, V \in \mathcal{V}\right\}$.

- We can now express the CSP in terms of a GSP.
  - The initial state is some arbitrary assignment, $\left\{V = v^{(0)}, V \in \mathcal{V}\right\}$.
  - The feasible actions involve modifying the value of any variable.

- We can now express the CSP in terms of a GSP.
  - The initial state is some arbitrary assignment, $\left\{V = v^{(0)}, V \in \mathcal{V}\right\}$.
  - The feasible actions involve modifying the value of any variable.
  - The goal test function checks that the assignment satisfies all the constraints.

- We can now express the CSP in terms of a GSP.
  - The initial state is some arbitrary assignment, $\left\{V = v^{(0)}, V \in \mathcal{V}\right\}$.
  - The feasible actions involve modifying the value of any variable.
  - The goal test function checks that the assignment satisfies all the constraints.
- As stated before, this is inefficient. There are two main reasons why.

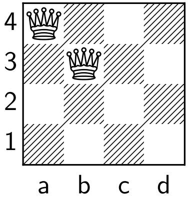
1. We must assign all variables simultaneously before checking the constraints, but constraints are often violated much earlier.

1. We must assign all variables simultaneously before checking the constraints, but constraints are often violated much earlier.

Example: Invalid Partial Assignment in *N*-Queens

- Suppose we place queens on a4 and b3. Then, no matter where we place the remaining queens, the resulting board configuration will be invalid.

- Still, because we must assign all variables simultaneously, we need to check all 16 board configurations that include queens on a4 and b3.

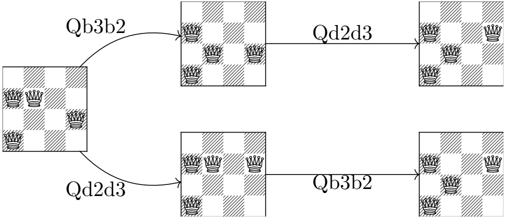


2. Different action sequences can yield the same state, but since we do not care about the paths, this results in searching more nodes than necessary.

2. Different action sequences can yield the same state, but since we do not care about the paths, this results in searching more nodes than necessary.

Example: Invalid Partial Assignment in *N*-Queens

- In the tree below, the order in which the moves Qb3b2 and Qd2d3 are played is irrelevant, as both result in the same board configuration.



- To address these issues, we can use the notion of a partial assignment.

- To address these issues, we can use the notion of a partial assignment.
- Given a partial assignment, we check the **bound** constraints, i.e., those whose scopes are fully assigned (non-bound constraints will not be violated).

- To address these issues, we can use the notion of a partial assignment.
- Given a partial assignment, we check the **bound** constraints, i.e., those whose scopes are fully assigned (non-bound constraints will not be violated).
- **Procedure**: Backtracking Search

- To address these issues, we can use the notion of a partial assignment.
- Given a partial assignment, we check the **bound** constraints, i.e., those whose scopes are fully assigned (non-bound constraints will not be violated).
- **Procedure**: Backtracking Search

    1. Start with the empty assignment, $\{\}$.

- To address these issues, we can use the notion of a partial assignment.
- Given a partial assignment, we check the **bound** constraints, i.e., those whose scopes are fully assigned (non-bound constraints will not be violated).
- **Procedure**: Backtracking Search

1. Start with the empty assignment, $\{\}$.
2. If all variables are assigned, a solution has been found. Otherwise, pick any unassigned variable, $V \in \mathcal{V}$.

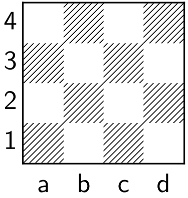
- To address these issues, we can use the notion of a partial assignment.
- Given a partial assignment, we check the **bound** constraints, i.e., those whose scopes are fully assigned (non-bound constraints will not be violated).
- **Procedure**: Backtracking Search
  1. Start with the empty assignment, $\{\}$.
  2. If all variables are assigned, a solution has been found. Otherwise, pick any unassigned variable, $V \in \mathcal{V}$.
  3. For each value, $v \in \operatorname{dom} V$: If every bound constraint is satisfied, continue searching recursively. If every value in $\operatorname{dom} V$ has been exhausted and none satisfied the bound constraints, backtrack.

Backtracking Search: Pseudo-code

```
procedure Search()
    if Assigned(V_1, ..., V_|𝒱|) then                          ▷ all variables are assigned
        𝒢.Append(([V_1], ..., [V_|𝒱|]))
    else
        V ← SelectUnassigned(𝒱)                              ▷ choose an unassigned variable
        for v ∈ dom V do
            Assign(v, V)
            γ ← false                                        ▷ flag for if constraints are violated
            for C ∈ 𝒞 : Bound(C) do                          ▷ for each bound constraint
                if Violated(C) then
                    γ ← true                                 ▷ the constraint is violated
            if γ = false then
                Search()                                     ▷ search extensions
        Unassign(V)
```

- In this example, we perform backtracking search on the 4-Queens puzzle.

Itr.	V_1	V_2	V_3	V_4
0				
1	a			
2	a	a		
3	a	b		
4	a	c		
5	a	c	a	
6	a	c	b	
7	a	c	c	
8	a	c	d	
9	a	d		
10	a	d	a	
11	a	d	b	
12	a	d	b	a
13	a	d	b	b
14	a	d	b	c
15	a	d	b	d
16	a	d	c	
17	a	d	d	
18	b			
19	b	a		
20	b	b		
21	b	c		
22	b	d		
23	b	d	a	
24	b	d	a	a
25	b	d	a	b
26	b	d	a	c



Heuristics for Variable Selection: Most Constraints

- The order in which unassigned variables are selected is arbitrary.

- The order in which unassigned variables are selected is arbitrary.
- However, the order can have a significant impact on performance.

Heuristics for Variable Selection: Most Constraints

- The order in which unassigned variables are selected is arbitrary.
- However, the order can have a significant impact on performance.
- One heuristic for determining which variable to select next is as follows:
  - Always choose the variable that is involved in the the most number of constraints.

Heuristics for Variable Selection: Most Constraints

- The order in which unassigned variables are selected is arbitrary.
- However, the order can have a significant impact on performance.
- One heuristic for determining which variable to select next is as follows:
  - Always choose the variable that is involved in the the most number of constraints.
- If a variable is involved in more constraints, it is more likely to violate one of them, and thus should be checked early.

- The order in which unassigned variables are selected is arbitrary.
- However, the order can have a significant impact on performance.
- One heuristic for determining which variable to select next is as follows:
    - Always choose the variable that is involved in the the most number of constraints.
- If a variable is involved in more constraints, it is more likely to violate one of them, and thus should be checked early.
- This heuristic is called **most constraints**.

- The previous example seems to suggest that backtracking is still inefficient.

- The previous example seems to suggest that backtracking is still inefficient.
    - After assigning $V_1 = \mathsf{a}$, we could remove:

- The previous example seems to suggest that backtracking is still inefficient.
  - After assigning $V_1 = \text{a}$, we could remove:
    - 'a' and 'b' from dom V_2

- The previous example seems to suggest that backtracking is still inefficient.
  - After assigning $V_1 = \text{a}$, we could remove:
    - 'a' and 'b' from dom V_2
    - 'a' and 'c' from dom V_3

- The previous example seems to suggest that backtracking is still inefficient.
  - After assigning $V_1 = \text{a}$, we could remove:
    - 'a' and 'b' from dom V_2
    - 'a' and 'c' from dom V_3
    - 'a' and 'd' from dom V_4

- The previous example seems to suggest that backtracking is still inefficient.
  - After assigning $V_1 = \mathrm{a}$, we could remove:
    - 'a' and 'b' from dom V_2
    - 'a' and 'c' from dom V_3
    - 'a' and 'd' from dom V_4
  - This is because the aforementioned choices would violate at least one constraint.

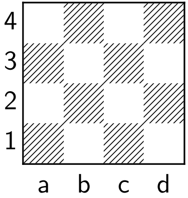
- The previous example seems to suggest that backtracking is still inefficient.
    - After assigning $V_1 = \mathrm{a}$, we could remove:
        - 'a' and 'b' from dom V_2
        - 'a' and 'c' from dom V_3
        - 'a' and 'd' from dom V_4
    - This is because the aforementioned choices would violate at least one constraint.
- After each assignment, we could iterate through each constraint, C, with exactly one unassigned variable, V, (so-called "almost bound" constraints) and prune dom V of any values that when assigned to V, violate C.

```
procedure SearchWithFC()
    if Assigned(V_1, ..., V_|𝒱|) then                          ▷ all variables are assigned
        𝒢.Append(([V_1], ..., [V_|𝒱|]))
    else
        V ← SelectUnassigned(𝒱)                               ▷ choose an unassigned variable
        for v ∈ dom V do
            Assign(v, V)
            γ ← false                                         ▷ flag for if constraints are violated
            for C ∈ 𝒞 : AlmostBound(C) do                     ▷ for each almost bound constraint
                if PruneWithFC(C) = DWO then
                    γ ← true                                  ▷ a domain wipe-out occurred
            if γ = false then
                SearchWithFC()                                ▷ search extensions
            RestorePruned()                                   ▷ restore pruned domains
        Unassign(V)
```

procedure PruneWithFC()
    $V \leftarrow$ FindUnassigned($\mathcal{V}_C$) ▷ find an unassigned variable in C's scope
    for $v \in \text{dom}\, V$ **do**
        Assign(v, V)
        if Violated(C) **then**
            Remove($v, \text{dom}\, V$)
            if $\text{dom}\, V = \emptyset$ **then** ▷ domain wipe-out occurred
                return `DWO`
    return `true`

- In this example, we perform backtracking search with forward-checking on the 4-Queens puzzle.

Itr.	V_1	V_2	V_3	V_4	dom V_1	dom V_2	dom V_3	dom V_4
0					$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$
1	a					$\{c, d\}$	$\{b, d\}$	$\{b, c\}$
2	a	c					$\varnothing$	
3	a	d				$\{b\}$	$\{c\}$	
4	a	d	b					$\varnothing$
5	b					$\{d\}$	$\{a, c\}$	$\{a, c, d\}$
6	b	d					$\{a\}$	$\{a, c\}$
7	b	d	a					$\{c\}$
8	b	d	a	c				



- Note that domains are restored whenever a domain wipe-out occurs.

Heuristics for Variable Selection: Fewest Remaining Values

- Forward checking gives us, for free, a very powerful heuristic for which variables to try next:

- Forward checking gives us, for free, a very powerful heuristic for which variables to try next:

- Forward checking gives us, for free, a very powerful heuristic for which variables to try next:
    - Always choose the variable with the fewest values remaining in its domain.

- Forward checking gives us, for free, a very powerful heuristic for which variables to try next:
  - Always choose the variable with the fewest values remaining in its domain.
- If a variable only has one value left, that value is forced, so we should propagate its consequences immediately.

Heuristics for Variable Selection: Fewest Remaining Values

- Forward checking gives us, for free, a very powerful heuristic for which variables to try next:
  - Always choose the variable with the fewest values remaining in its domain.
- If a variable only has one value left, that value is forced, so we should propagate its consequences immediately.
- This heuristic is called **fewest remaining values** (we actually employ it in the previous example).

- In forward-checking, we only checked the constraints whose scopes had exactly one unassigned variable. In other words, we had a modest amount of constraint propagation.

- In forward-checking, we only checked the constraints whose scopes had exactly one unassigned variable. In other words, we had a modest amount of constraint propagation.
- Of course, in theory, we could check constraints whose scopes contain multiple unassigned variables...though this would take longer to enforce.

- In forward-checking, we only checked the constraints whose scopes had exactly one unassigned variable. In other words, we had a modest amount of constraint propagation.
- Of course, in theory, we could check constraints whose scopes contain multiple unassigned variables...though this would take longer to enforce.
- Alternatively, we could observe that some partial assignments will violate an unbound constraint regardless of what we assign to the other variables in its scope.

- **Example**: Pruning Inconsistent Domains

- **Example**: Pruning Inconsistent Domains
  - Let X and Y be two variables with $\operatorname{dom} X = \{1, 6, 11\}$ and $\operatorname{dom} Y = \{3, 8, 15\}$.

- **Example**: Pruning Inconsistent Domains
  - Let X and Y be two variables with $\operatorname{dom} X = \{1, 6, 11\}$ and $\operatorname{dom} Y = \{3, 8, 15\}$.
  - Consider the constraint, $X > Y$.

- **Example**: Pruning Inconsistent Domains
    - Let X and Y be two variables with $\operatorname{dom} X = \{1, 6, 11\}$ and $\operatorname{dom} Y = \{3, 8, 15\}$.
    - Consider the constraint, $X > Y$.
    - If we let $Y = 15$, the constraint is not satisfiable since there is no $x \in \operatorname{dom} X$ such that $x > 15$.

- **Example**: Pruning Inconsistent Domains
  - Let X and Y be two variables with $\operatorname{dom} X = \{1, 6, 11\}$ and $\operatorname{dom} Y = \{3, 8, 15\}$.
  - Consider the constraint, $X > Y$.
  - If we let $Y = 15$, the constraint is not satisfiable since there is no $x \in \operatorname{dom} X$ such that $x > 15$.
  - If we let $X = 1$, the constraint is not satisfiable since there is no $y \in \operatorname{dom} Y$ such that $1 > Y$.

- **Example**: Pruning Inconsistent Domains
  - Let X and Y be two variables with $\operatorname{dom} X = \{1, 6, 11\}$ and $\operatorname{dom} Y = \{3, 8, 15\}$.
  - Consider the constraint, $X > Y$.
  - If we let $Y = 15$, the constraint is not satisfiable since there is no $x \in \operatorname{dom} X$ such that $x > 15$.
  - If we let $X = 1$, the constraint is not satisfiable since there is no $y \in \operatorname{dom} Y$ such that $1 > Y$.
  - Thus, we can prune '1' from $\operatorname{dom} X$ and '15' from $\operatorname{dom} Y$.

Backtracking Search with Generalized Arc Consistency: Pseudo-code

```
procedure SearchWithGAC()
    if Assigned(V_1, ..., V_|𝒱|) then                          ▷ all variables are assigned
        𝒢.Append(([V_1], ..., [V_|𝒱|]))
    else
        V ← SelectUnassigned(𝒱)                                ▷ choose an unassigned variable
        for v ∈ dom V do
            Assign(v, V)
            𝒬 = ∅
            for C ∈ 𝒞 : V ∈ 𝒱_C do                ▷ for each constraint whose scope contains V
                𝒬.Append(C)
            if not PruneWithGAC(𝒬) = DWO then                   ▷ no domain wipe-out occurred
                SearchWithGAC()
            RestorePruned()                                     ▷ restore pruned domains
        Unassign(V)
```

Backtracking Search with Generalized Arc Consistency: Pseudo-code

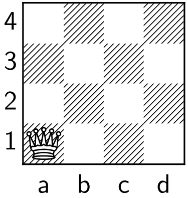
```
procedure PruneWithGAC(𝒬)
    while 𝒬 ≠ ∅ do
        C ← 𝒬.Next()
        for V ∈ 𝒱_C do
            for v ∈ dom V do
                if not Satisfiable(C, V, v) then
                    Remove(v, dom V)
                    if dom V = ∅ then
                        return DWO
                    else
                        for C ∈ 𝒞 do
                            if V ∈ 𝒱_C  and  C ∉ 𝒬 then
                                𝒬.Append(C)
```

Backtracking Search with Generalized Arc Consistency: Example

- In this example, we perform backtracking search while enforcing arc consistency on the 4-Queens puzzle.

Itr. 1:

dom V_1	dom V_2	dom V_3	dom V_4	$\mathcal{Q}$
$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{\}$
$\{a\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{C_1^2, C_1^3, C_1^4\}$
$\{a\}$	$\{c, d\}$	$\{a, b, c, d\}$	$\{a, b, c, d\}$	$\{C_1^3, C_1^4, C_2^3, C_2^4\}$
$\{a\}$	$\{c, d\}$	$\{b, d\}$	$\{a, b, c, d\}$	$\{C_1^4 C_2^3, C_2^4, C_3^4\}$
$\{a\}$	$\{c, d\}$	$\{b, d\}$	$\{b, c\}$	$\{C_2^3, C_2^4, C_3^4\}$
$\{a\}$	$\{d\}$	$\{b\}$	$\{b, c\}$	$\{C_2^4, C_3^4, C_2^1, C_2^4, C_3^1\}$
$\{a\}$	$\{d\}$	$\{b\}$	$\{c\}$	$\{C_3^4, C_2^1, C_2^4, C_3^1, C_4^1\}$
$\{a\}$	$\{d\}$	$\emptyset$	$\{c\}$	$\{C_2^1, C_2^4, C_3^1, C_4^1\}$

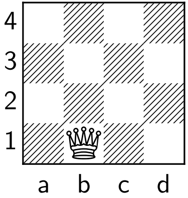


Backtracking Search with Generalized Arc Consistency: Example

- In this example, we perform backtracking search while enforcing arc consistency on the 4-Queens puzzle.

Itr. 2:

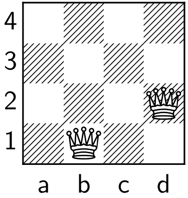
dom V_1	dom V_2	dom V_3	dom V_4	$\mathcal{Q}$
$\{b\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{C_1^2, C_1^3, C_1^4\}$
$\{b\}$	$\{d\}$	$\{a,b,c,d\}$	$\{a,b,c,d\}$	$\{C_1^3, C_1^4, C_2^3, C_2^4\}$
$\{b\}$	$\{d\}$	$\{a,c\}$	$\{a,b,c,d\}$	$\{C_1^4, C_2^3, C_2^4, C_3^4\}$
$\{b\}$	$\{d\}$	$\{a,c\}$	$\{a,c,d\}$	$\{C_2^3, C_2^4, C_3^4\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{a,c,d\}$	$\{C_2^4, C_3^4, C_2^1\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{a,c\}$	$\{C_3^4, C_2^1, C_4^1\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{C_2^1, C_4^1, C_4^2\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{C_4^1, C_4^2\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{C_4^2\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{\}$



- In this example, we perform backtracking search while enforcing arc consistency on the 4-Queens puzzle.

Itr. 3:

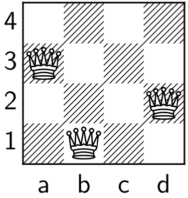
dom V_1	dom V_2	dom V_3	dom V_4	$\mathcal{Q}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{C_2^1, C_2^3, C_2^4\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{C_2^3, C_2^4\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{C_2^4\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{\}$



- In this example, we perform backtracking search while enforcing arc consistency on the 4-Queens puzzle.

Itr. 4:

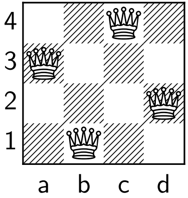
dom V_1	dom V_2	dom V_3	dom V_4	$\mathcal{Q}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{C_3^1, C_3^2, C_3^4\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{C_3^2, C_3^4\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{C_3^4\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{\}$



- In this example, we perform backtracking search while enforcing arc consistency on the 4-Queens puzzle.

Itr. 5:

dom V_1	dom V_2	dom V_3	dom V_4	$\mathcal{Q}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\left\{C_4^1, C_4^2, C_4^3\right\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\left\{C_4^2, C_4^3\right\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\left\{C_4^3\right\}$
$\{b\}$	$\{d\}$	$\{a\}$	$\{c\}$	$\{\}$



- In the 4-Queens example, we observed the following:

- In the 4-Queens example, we observed the following:
    - Backtracking search takes many iterations.

- In the 4-Queens example, we observed the following:
  - Backtracking search takes many iterations.
  - Using forward checking reduced the number of iterations, but each one takes longer.

- In the 4-Queens example, we observed the following:
    - Backtracking search takes many iterations.
    - Using forward checking reduced the number of iterations, but each one takes longer.
    - Enforcing arc consistency reduces the number of iterations even further, but each one takes even longer still.

- In the 4-Queens example, we observed the following:
  - Backtracking search takes many iterations.
  - Using forward checking reduced the number of iterations, but each one takes longer.
  - Enforcing arc consistency reduces the number of iterations even further, but each one takes even longer still.
- This is true in general.