Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

CAIDAS WüNLP

ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Constraint Satisfaction

Prof. Dr. Goran Glavaš

18.1.2024

Content

- Constraint Satisfaction Problems
- Backtracking Algorithm
- Example Problems

Recap: State Space Search

- We will denote the set of all states (state space) with **S**
  - The state space is commonly **so large** that we can't iteratively list all states
  - All states in the space are not really „known" in advance
  - When in state s, we typically only then compute the set of possible next states

State space search

A state space search problem is defined with a triple (s_0, *succ*, goal) where $s_0 \in S$ is the **initial state**, *succ*: **S** → ℘(**S**) is the **successor function** that for some state s returns a set of states that we can **transition to** from s, and *goal*: **S** → {True, False} is a **predicate** (function that returns a boolean value) that for a given state s determines if s is a **goal state** or not (there can be multiple states that satisfy the goal predicate). A state space search (typically) ends as soon as any goal state is found.

Discrete (Constrained) Optimization

Discrete Constrained Optimization Problems

In **discrete constrained optimization**, we search for an **optimal state** in large space of possible states. Each state **X** can be seen as consisting of n variables **X** = $x_1, x_2, \ldots, x_n$, each with a corresponding domain $\mathbf{D_1}, \mathbf{D_2}, \ldots, \mathbf{D_n} \subseteq \mathbb{Z}$ (whole numbers). The optimal state is the one that maximizes/minimizes the **objective function** f: $\mathbf{D_1} \times \cdots \times \mathbf{D_n} \rightarrow \mathbb{R}$. Finally, the constraints $\mathbf{C_1}, \ldots, \mathbf{C_m}$, with $\mathbf{C_i} \subseteq \mathbf{D_1}, \mathbf{D_2}, \ldots, \mathbf{D_n}$ define the subsets of the state space that encompass **valid solutions** to the problem

- Optimal state (or the state with the best f that was found) is the **solution**
- No path between start and goal state – often there isn't a clear start state
- We're not making moves like in classic **SSS** problems, just **searching for the best possible solution** over a very large space of candidate solutions

Recap: State Space Search & Discrete Optimization

- **State Space Search**
  - **Goal states** represent a very small portion of the states in the search space
  - Only **paths** that reach one of goal states are (candidate) **solutions**
  - Explicit transitions between the states (*succ* function)
  - **Problem**: how to get to a goal state with **minimal cost/maximal gain**

- **Discrete Constrained Optimization**
  - **Every state** represents one candidate solution to the problem
  - Each state (candidate solution) has a measure of **"quality"** – **the objective function** f – *a*ssigned to it
  - No explicit „start state" nor „state transitions" – instead **neighbourhood** (or distance) between states (but no in the sense of transition cost)
  - **Problem**: how to find the state with minimal/maximal value of the objective f

Recap: Heuristics

- If we have a (vague) **idea in which direction to look** for the solution, why not use this information to improve the search?

- **Heuristics =** problem-specific rules („vague ideas") about the nature of the problem
  - **Purpose**: direct the search towards the goal so it becomes more efficient

Heuristic function

Heuristic function h: **S** $\rightarrow \mathbb{R}^{+}$ assigns to each state $s \in$ **S** an **estimate** of the **distance between that state and the goal state**

Recap: Metaheuristics

- **Metaheuristics strategies guide** the search process
  - **Direct** the search (selection of next states to evaluate) so that the chances of finding a good (or near-optimal) state increase

- They are approximate – no guarantee of finding an **optimal solution**

- Most commonly, they are also **non-deterministic** (and most often **stochastic**) – there is randomness involved

- **Metaheuristics** are **problem-agnostic**, but may use problem-specific heuristics as part of the strategy (but as „black boxes", without caring what they are)

Constraint Satisfaction Problem: Example

Map Coloring Problem

We're given a **map** consisting of N **regions**, we need to **color each region** with one of M colors but so that **neighbouring regions always have different colors**

- We can represent the map as a graph – one **region**, one **node**
- If regions are **neighbours** – establish an **edge** between the corresponding graph nodes

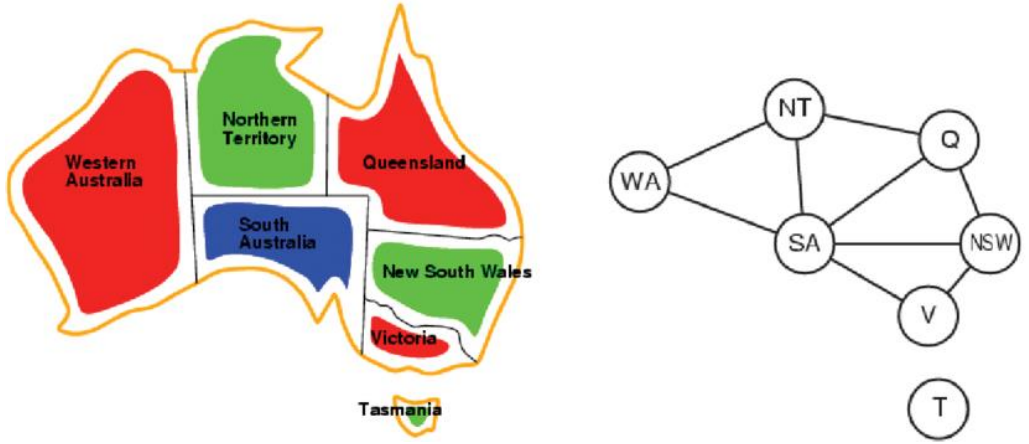


Image from: https://www.researchgate.net/figure/An-example-of-graph-coloring-problem_fig2_325808704

Constraint Satisfaction Problem: Example

Map Coloring Problem

We're given a **map** consisting of N **regions**, we need to **color each region** with one of M colors but so that **neighbouring regions always have different colors**

- One **state** (potential solution): one (any) **coloring** of the graph
- **Many (most)** of all possible colorings will violate the constraint(s)
  - **Key**: No point in further searching from those **partial** states that violate constraints

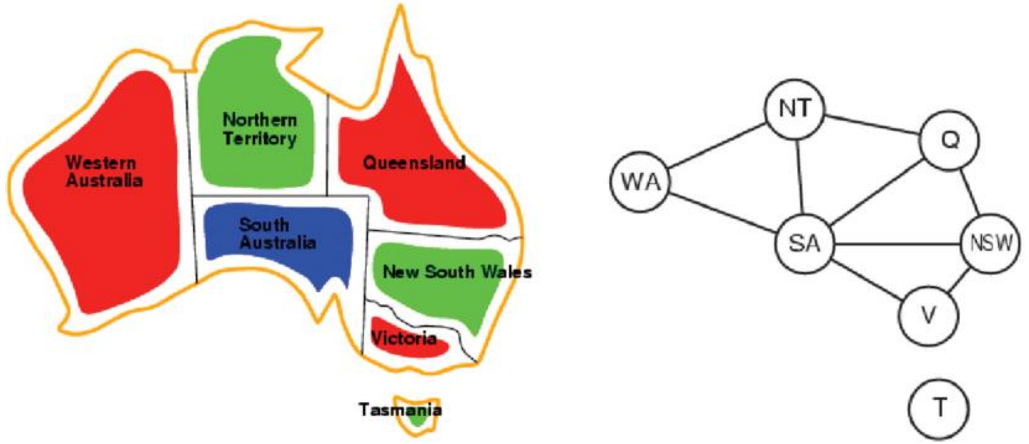


Image from: https://www.researchgate.net/figure/An-example-of-graph-coloring-problem_fig2_325808704

Constraint Satisfaction Problem

Constraint Satisfaction Problems

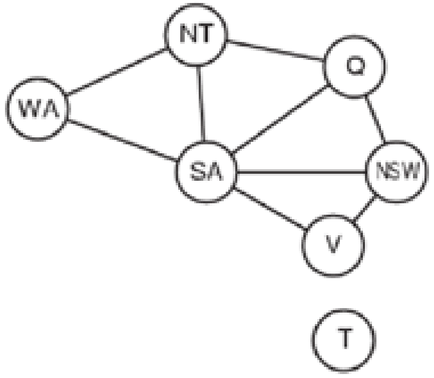
Constraint satisfaction problems (CSP) are search problems where we search for a **state** **X** that can be **factored** into n variables **X** = x_1, x_2, ..., x_n, each with a corresponding domain **D_1**, **D_2**, ..., **D_n** $\subseteq \mathbb{Z}$ (whole numbers), which **satisifes** the (set of) **constraints** **C**. Unlike in discrete constrained optimization, in CSP we search through the states that represent **partial solutions** to the problem, that is, where only a subset of the variables x_1, x_2, ..., x_n has been assigned a value. The **key property of the CSP**s is that a partial solution that violates the constraints cannot be part of the goal state/solution. This allows to simply discard **large portions of the state space during the search**.

- In a sense, we're incrementally constructing a solution and **backtrack** everytime the partial solution we built violates the constraints.

Constraint Satisfaction Problem: Example

- **Initial state** s_0 = no colored nodes
- **States** that we „transition" to – one node colored (value fixed for one variable x_i)
  - **WA** colored red / blue / green (3 different states)
  - **NT** colored red / blue / green (3 different states)
  - **SA** colored red / blue / green (3 different states)
  - **…**

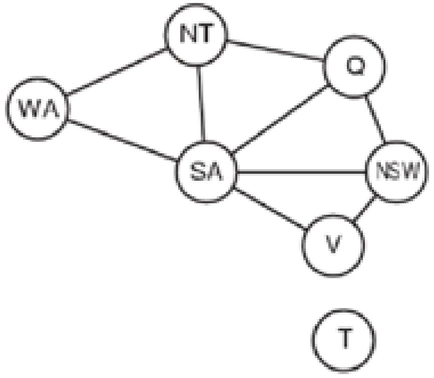
- We can just pick any to start with
  - A particular value for a single variable x_i cannot break the constraints, only in relation to values of other variables x_j



Constraint Satisfaction Problem: Example

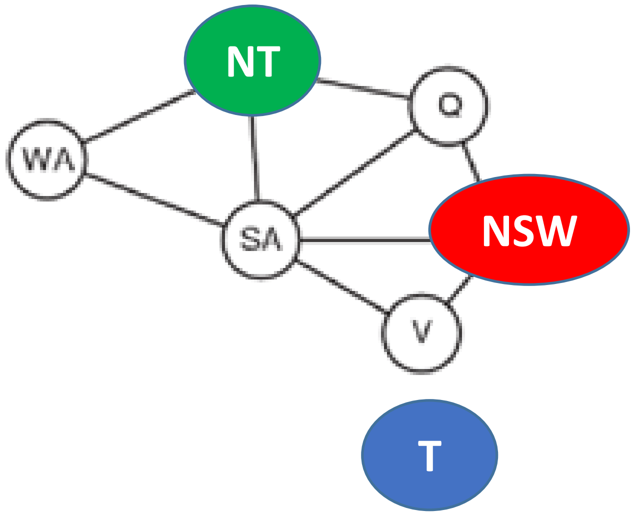
- Generally, a state s is a **partial assignment** of values to some subset of variables from **X**
  - We've assigned color to some subset of regions/nodes

- The **next possible states**: set of partial assignments with one more assigned variable
  - If we have *k* remaining unassigned variables, and *d* possible values that can, in principle, be assigned to each of them
  - Then we have a **search** with a branching factor b = k * d!!!
  - This would, in general, be intractable



Constraint Satisfaction Problem: Example

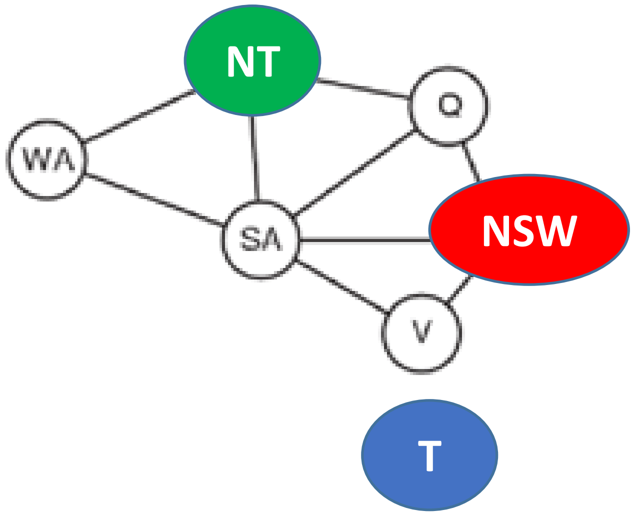
- **Current state s**: colored **NT**, **NSW**, and **T**
- **Next states**:
  - **+ WA** → violates the constraint!
  - **+ WA**
  - **+ WA**
  - **+ SA** or **+ SA** (violation!) or **+ SA** (violation)
  - **+ Q** or **+ Q** (violation!) or **+ Q** (violation)
  - **+ V** or **+ V** or **+ V** (violation!)



- If we find a state (partial solution) that violates constraints, **no need to continue from that state**
  - Subsequent assignments to remaining unassigned variables cannot fix the violation and thus **cannot lead to a goal state**

Constraint Satisfaction Problem: **Commutativity**

- Note that the **order of assignments** of **colors** (values) to **nodes** (variables) **does not matter**!
  - 1. **NT**, 2. **NSW**, 3. **T**
  - 1. **NT**, 2. **T**, 3. **NSW**
  - 1. **NSW**, 2. **NT**, 3. **T**
  - 1. **NSW**, 2. **T**, 3. **NT**
  - 1. **T**, 2. **NT**, 3. **NSW**
  - 1. **T**, 2. **NSW**, 3. **NT**

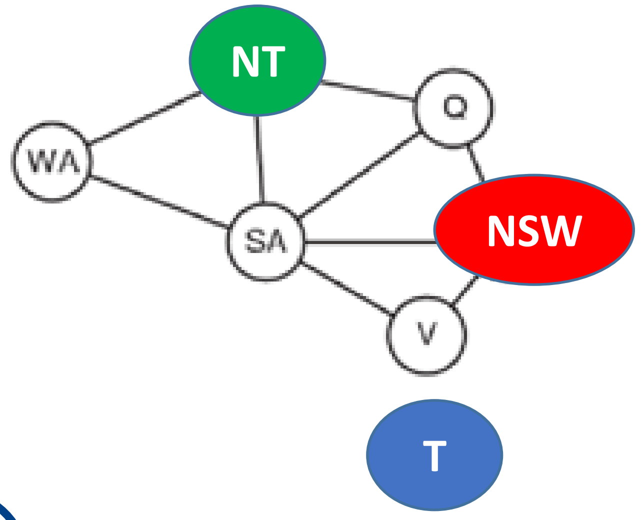


- We care about whether the state we're in (partial solution) violates the constraints or not, **not** how we got to that state
- **Path** doesn't matter → **different** from **state space search**

Constraint Satisfaction Problem: **Commutativity**

- Note that the **order of assignments** of **colors** (values) to **nodes** (variables) **does not matter**!

- **CSPs** are **commutative**!



Commutative (Search) Problems

A (search) problem is **commutative** if the **order** of application of any given set of actions (operations) has **no effect** on the **outcome**.

Constraint Satisfaction Problem: **Commutativity**

Commutative (Search) Problems

A (search) problem is **commutative** if the **order** of application of any given set of actions (operations) has **no effect** on the **outcome**.

- **CSPs** are **commutative**!
- This means that we actually have d^n possible assignments to all variables
  - n – the number of variables (x_1 to x_n) (nodes)
  - d – as the number of values that can be assigned to each of them (colors)
- We typically need to find **only one** (**any**) that doesn’t violate the constraints

Different Search Problems: SSS vs. DCO vs. CSP

- **State space search** (example: jigsaw puzzle):
  - **Optimal path problems: many** possible paths from initial to goal state, need to find the one with minimal cost / maximal gain
  - Complex, **non-factorable** states
  - Explicit state transitions defined by the nature of the problem
- **Discrete (Constrained) Optimization** (example: travelling salesman)
  - **Optimal state problems**: very **many** solutions satisfy the constraints, find the **optimal**
  - **Factorable states** = problem is a set of value assignments to variables
  - No explicit state transitions, need to define **neighborhoods**
- **Constraint Satisfaction** (example: graph coloring or sudoku)
  - **Few (if any)** solutions that satisfy the constraints, find **any** (all equally good)
  - **Factorable states** = problem is a set of value assignments to variables
  - Transitions: from a state with k assigned variables, to those with k+1 assigned variables

Content

- Constraint Satisfaction Problems
- Backtracking Algorithm
- Example Problems

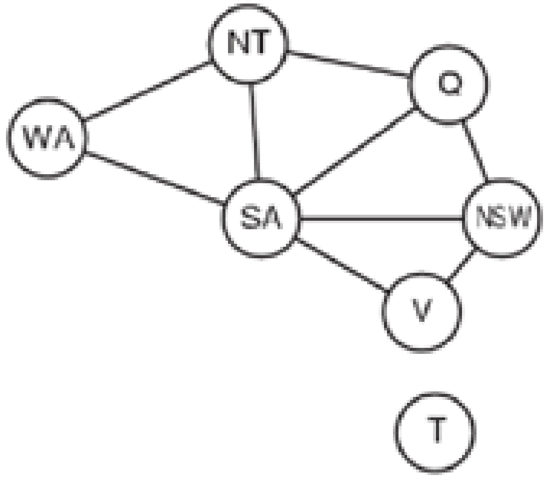
Backtracking

- **Backtracking** is a **„brute force”** algorithm for finding solutions to CSPs, exploiting two key properties of CSPs:
  - **Commutativity**
  - Unsatisfying **partial solutions** (those that violate constraints) cannot lead to a satisfying solution

- Essentially, a **depth-first search (DFS)** that chooses values for one variable x_i at a time and then **backtracks** when a variable cannot be assigned any value due to constraint violation

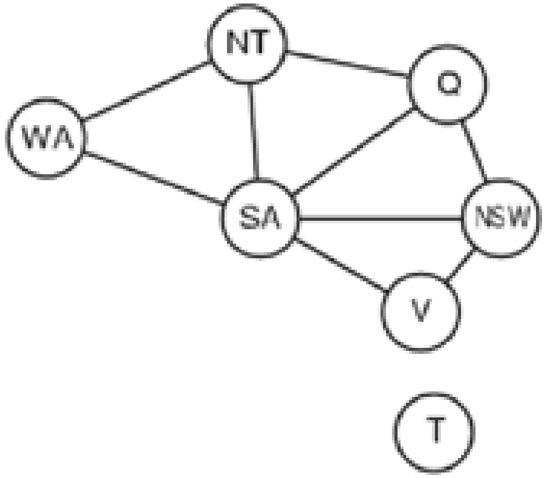
Backtracking: Example

- **Assumption:** there's an **order** of values for each variable in which we try to assign them
  - For example: first green, then blue, then red
- Since the CSPs problems are **commutative**, we can start assigning from any variable
- Also, in **any subsequent step**, we can pick any of the remaining variables that haven't been assigned
- Better strategies for „selection of next variable to assign" and „order of values to try to assign to" can lead to **more efficient search**
  - These are called – surprise – **heuristics** ☺



Backtracking: Example

- **(Naive) backtracking**
  - Randomly selecting the next variable to be assigned
  - Trying to assign values in random (or same) order
    - In example: first **green**, then **blue**, then **red**



{NT}

{NT, WA} X (violation)

{NT, WA}

{NT, WA, V}

{NT, WA, V, NSW} X

{NT, WA, V, NSW}

{NT, WA, V, NSW, SA} X

{NT, WA, V, NSW, SA} X

{NT, WA, V, NSW, SA}

{NT, WA, V, NSW, SA, …}

- **Q:** Does this problem have a solution at all?
  - Add one edge so that it doesn’t have a solution!

Backtracking

- **CSP** is „described" in the data structure `csp`
  - `csp.vars` contains the states of the variables (assigned/unassigned and the *value* if assigned)
  - `csp.violates` is a predicate that indicates if the current partial assignment violates the constraints of the CSP

```
backtracking_search(csp)
   return backtrack({}, csp)


backtrack(s, csp)
  if complete(s) return s
  v = select_unassigned_var(csp.vars)

  for val in order-values(v, s, csp)
    if not csp.violates(s ∪ (v, val))
      csp.vars[v] = val
      res = backtrack(s ∪ (v, val), csp)
      if res ≠ null
         return res

  csp.vars[v] = null
  return null
```

Inference (or Constraint Propagation)

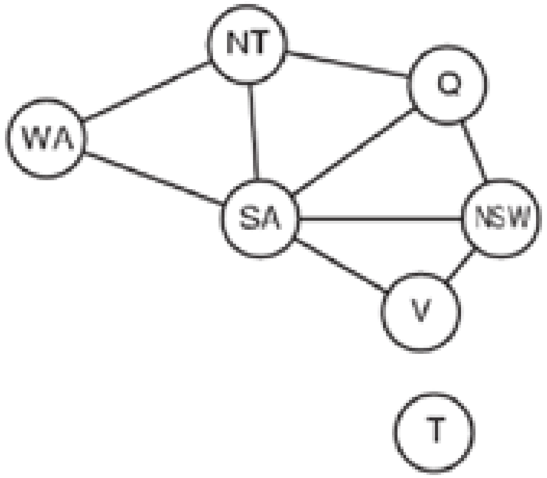
- The previous variant of backtracking is somewhat **naive**
  - It **(1) assigns a value** to a variable and only then
    (2) checks whether with this new assignment, we violate constraints
- **Q:** Can we know in advance that certain values for certain variables lead to constraint violation, **before** those variables are assigned
  - So that we don't even try to assign such values to those variables?
  - This would **improve efficiency**!
- **Inference** (or **constraint propagation**)**:** upon assignment of a value to a variable, try to **„reduce"** the domains – sets of still allowed values – for **all remaining unassigned variables** using the **constraints**

Inference (or Constraint Propagation)

- **Inference** or **constraint propagation:** upon assignment of a value to a variable, try to **„reduce"** the domains (sets of allowed values) for **all remaining unassigned variables** using the **constraints**

- Graph coloring: initially, each of the three colors (values) may be assigned to each of the nodes (variables)

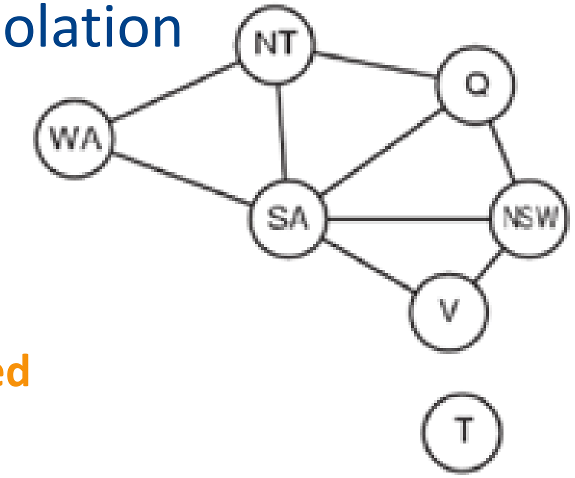
  - **WA →** {green, blue, red}
  - **NT →** {green, blue, red}
  - **SA →** {green, blue, red}
  - **Q →** {green, blue, red}
  - **NSW →** {green, blue, red}
  - **V →** {green, blue, red}
  - **T →** {green, blue, red}



Inference (or Constraint Propagation)

- **Inference** or **constraint propagation:** upon assignment of a value to a variable, try to **„reduce”** the domains (sets of allowed values) for **all remaining unassigned variables** using the **constraints**

- Graph coloring: But when we choose the color for some node, this reduces the number of colors assignable to neighbors without violation
    - When we set **NT** to green, **WA**, **SA**, and **Q** cannot be green
    - No need to try those variants only to detect violation



{**NT**}

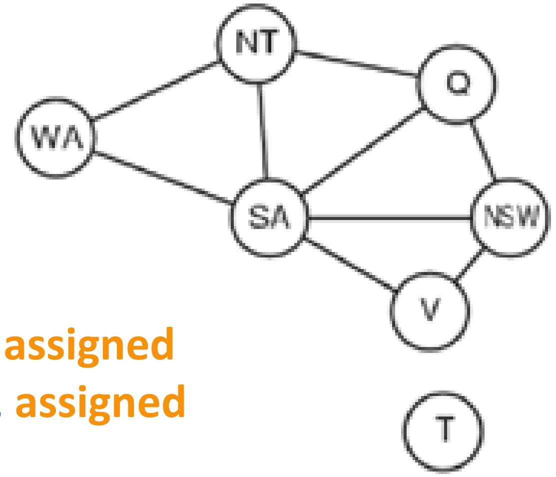
WA → {blue, red}
NT → {green}, **assigned**
SA → {blue, red}
Q → {blue, red}
NSW → {green, blue, red}
V → {green, blue, red}
T → {green, blue, red}

Inference (or Constraint Propagation)

- **Inference** or **constraint propagation:** upon assignment of a value to a variable, try to **„reduce”** the domains (sets of allowed values) for **all remaining unassigned variables** using the **constraints**

- Graph coloring: But when we choose the color for some node, this reduces the number of colors assignable to neighbors
  - When in the next step, we set **WA** to **blue**, that color needs to be removed for **SA**
  - No need to try those variants to determine violation

{**NT**} → {**NT**, **WA**} →



WA → {blue}, **assigned**
NT → {green}, **assigned**
SA → {red}
Q → {blue}
NSW → {green}
V → {blue}
T → {green, blue, red}

- **Q:** What variable does it make **most sense** to choose next for the assignment?

Backtracking with **Inference**

- After each variable assignment, we perform **inference**
- To **limit** the remaining possibilities for **unassigned variables** based on constraints
  - To **speed up** the search
- Function `csp.inference` adjusts/reduces the domains of unassigned vars
  - Can (implicitly) detect violation – if a variable remains with **empty domain**
- Function `csp.remove` removes inferences, that is, returns removed values to their domains
- But when we **backtrack**, we have to also remove all inferences made based on the backtracked assignment

```
backtracking_search(csp)
   return backtrack({}, csp)

backtrack(s, csp)
  if complete(s) return s
  v = select_unassigned_var(csp.vars)
  for val in order_values(v, s, csp)
    if not csp.violates(s ∪ (v, val))
      csp.vars[v] = val
      infs = csp.inference(v, val)
      if infs = null # violation
        continue
      res = backtrack(s ∪ (v, val), csp)
      if res ≠ null
         return res
      csp.remove(infs)
  csp.vars[v] = null
  return null
```

Backtracking with Inference

- **Efficiency** of backtracking depends on implementation of
  - `select_unassigned_var`
  - `order_values`
- **Q: problem-specific** or **problem-agnostic** selection strategies?
  - **Heuristics** or **metaheuristics**? ☺
- CSPs can be solved efficiently **without** problem-specific knowledge

```
backtracking_search(csp)
   return backtrack({}, csp)

backtrack(s, csp)
  if complete(s) return s
  v = select_unassigned_var(csp.vars)
  for val in order_values(v, s, csp)
    if not csp.violates(s ∪ (v, val))
      csp.vars[v] = val
      infs = csp.inference(v, val)
      if infs = null # violation
        continue
      res = backtrack(s ∪ (v, val), csp)
      if res ≠ null
         return res
      csp.remove(infs)
  csp.vars[v] = null
  return null
```

Efficient Search Strategies for CSP

- `select_unassigned_var`
- **Mininum-remaining-values (MRV)**
  - (Meta)Heuristic also known as „most constrained variable" or „fail first"
  - Select next the **unassigned variable** with **least** remaining allowed values
- **Degree heuristic**
  - Select the variable setting the value of which will **constrain** the domains of the largest number of unassigned variables
  - **Graph coloring**: node with the largest number of (outgoing) edges

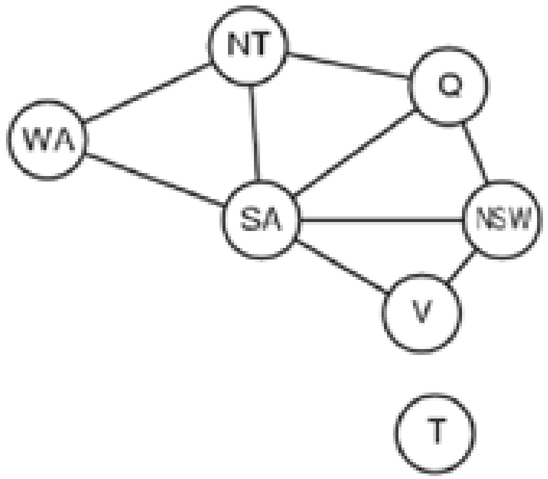
```
backtracking_search(csp)
   return backtrack({}, csp)

backtrack(s, csp)
  if complete(s) return s
  v = select_unassigned_var(csp.vars)
  for val in order_values(v, s, csp)
    if not csp.violates(s U (v, val))
      csp.vars[v] = val
      infs = csp.inference(v, val)
      if infs = null # violation
        continue
      res = backtrack(s U (v, val), csp)
      if res ≠ null
        return res
      csp.remove(infs)
  csp.vars[v] = null
  return null
```

Degree Heuristic: Example

- The node with the largest degree is **SA**: set **SA** (or any color, really)

WA → {blue, red}
NT → {blue, red}
SA → {green}, **assigned**
Q → {blue, red}
NSW → {blue, red}
V → {blue, red}
T → {green, blue, red}



- Next, any between **NT**, **Q**, and **NSW**: let's say we set **NSW** to **blue**

- With the **degree heuristic** and **inference**, we even managed to find a solution **without backtracking**!

WA → {red}
NT → {blue}
SA → {green}, **assigned**
Q → {red}
NSW → {blue}, **assigned**
V → {red}
T → {green, blue, red}

Efficient Search Strategies for CSP

order_values

- Defines order in which we try the value assignment for the selected variable
- **Least-constraining-value (LCV)**
  - Select the **value** for which the remaining unassigned variables will be **least constrained**
  - Value that rules out the fewest choices for the unassigned variables
  - Leaves the most possibilities for the unassigned variables
  - Thus has the best chance to eventually not lead to a violation

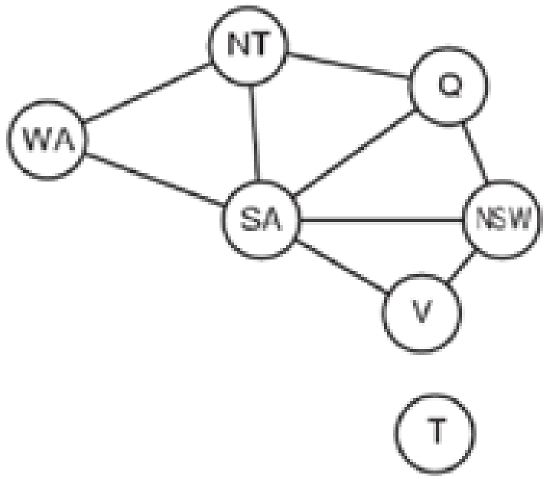
```
backtracking_search(csp)
   return backtrack({}, csp)

backtrack(s, csp)
  if complete(s) return s
  v = select_unassigned_var(csp.vars)
  for val in order_values(v, s, csp)
    if not csp.violates(s ∪ (v, val))
      csp.vars[v] = val
      infs = csp.inference(v, val)
      if infs = null # failure
        continue
      res = backtrack(s ∪ (v, val), csp)
      if res ≠ null
         return res
      csp.remove(infs)
  csp.vars[v] = null
  return null
```

Least-Constraining-Value: Example

- Assume we made a partial assignment: **NT** and **WA** and that our next node to be assigned is **Q**

  WA → {blue}, **assigned**
  NT → {green}, **assigned**
  SA → {red},
  Q → {blue, red}
  NSW → {green, blue, red}
  V → {green, blue, red}
  T → {green, blue, red}



WA → {red}
NT → {blue}
SA → {green}, **assigned**
Q → {red}
NSW → {blue}, **assigned**
V → {red}
T → {green, blue, red}

- We have two possible values for **Q**: **blue** and **red**
  - Both reduce the number of remaining values for **NSW** by 1
  - But, **red Q** also reduces the number of possibilities for **SA** (and actually leads to violation immediately), whereas **blue Q** doesn't

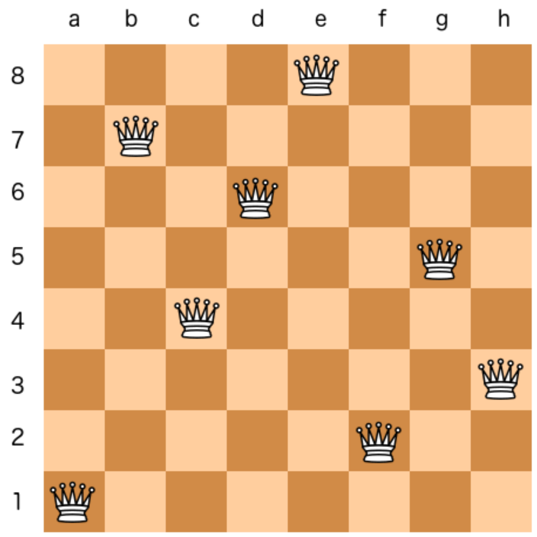
Content

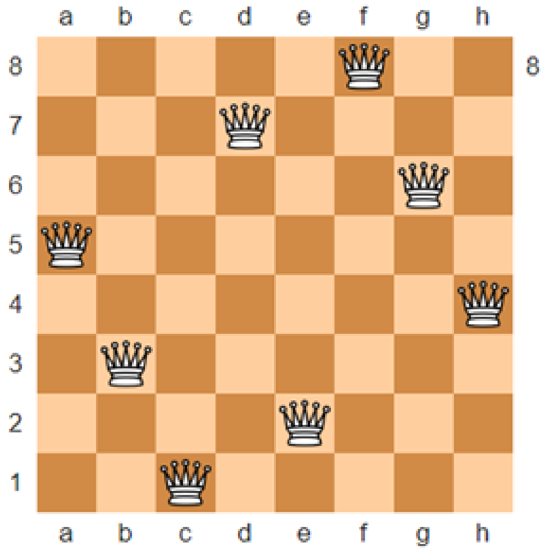
- Constraint Satisfaction Problems
- Backtracking Algorithm
- Example Problems

CSP Examples: **8-Queen Problem**

8-Queen Problem

Place 8 (N) queens on an 8 by 8 (N by N) chess board such that **none of the queens attacks any of the others**.





Images from https://stackoverflow.com/questions/63536411/how-to-rotate-a-solution-to-the-8-queens-puzzle-by-90-degrees

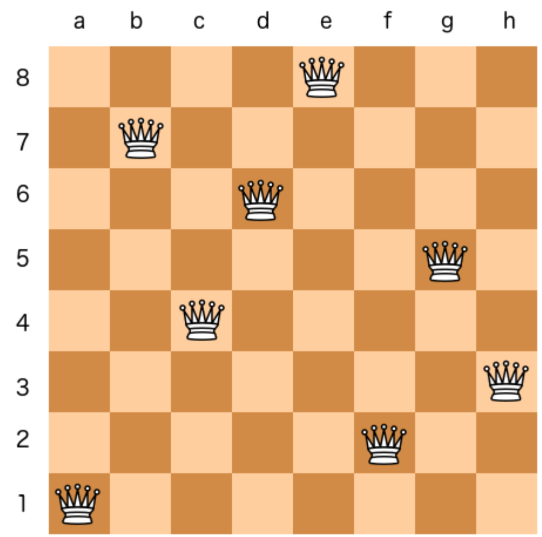
CSP Examples: **8-Queen Problem**

- All we have to do is formulate problem as **CSP**
  - Backtracking, inference, and heuristics will take care of the rest ☺

8-Queen Problem

Place 8 (N) queens on an 8 by 8 (N by N) chess board such that **none of the queens attacks any of the others**.

- **Constraint:** no two queens in the same row, column or diagonal



- We know already that one queen has to be in each **row**/column
  - **X** = $x_1, x_2, \ldots, x_8$

    x_i = **column** or the queen in **row** i

    $x_1, x_2, \ldots, x_8 \in \{a, b, c, d, e, f, g, h\}$

CSP Examples: **Sudoku**

Sudoku

A (standard) sudoku is a grid with 81 cells, some of which have been prefilled with numbers (1 to 9). The task is to **fill the empty cells** (also only with numbers 1 to 9) so that **no number repeats** in any row, any column, or any of the 9-cell (3x3) sub-grids. Put differently, we must have all numbers 1-9 in every row, column and 3x3 sub-grid.

- **X** = $x_1, \ldots, x_n$, n = number of empty cells
- $x_i \subseteq \{1, 2, \ldots, 9\}$
- **Q:** Easier with more or fewer numbers filled in at the beginning? Why?

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

Image from https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

Questions?

Awọn ibeere?
ਸਵਾਲ?
Küsimusi?
Turite klausimų?
Pitanja?
Sorusu olan?
Dúvidas?
質問は？
有问题吗？
Fragen?
¿Preguntas?
Questions?
Frågor?
Pytania?
Domande?
Питання?
Vragen?
Ερωτήσεις;
Porandukuéra?
คำถาม?
أسئلة؟