

# 1 Constraint Satisfaction Problems (CSPs)

Constraint Satisfaction Problems (CSPs) stellen einen Paradigmenwechsel gegenüber der klassischen Pfadsuche dar. Während bei Planungsproblemen der Pfad zum Ziel entscheidend ist, interessiert bei CSPs nur der Zielzustand selbst (Identifikationsproblem). Der Zustand ist dabei nicht mehr atomar ("Blackbox"), sondern faktorisiert, d. h. er besteht aus Variablen und Werten.

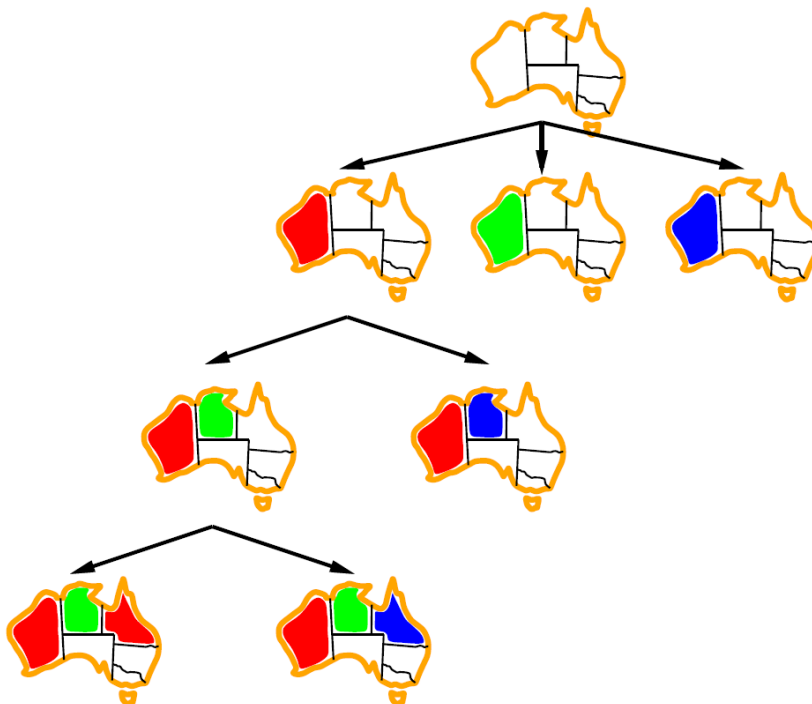
## 1.1 Definition und Komponenten

Ein CSP wird durch drei Komponenten definiert:

1. **Variablen**  $X = \{X_1, X_2, \dots, X_n\}$ : Die Objekte, denen Werte zugewiesen werden müssen.
2. **Domänen** (Wertebereiche)  $D = \{D_1, D_2, \dots, D_n\}$ : Für jede Variable  $X_i$  gibt es eine Menge  $D_i$  von möglichen Werten.
3. **Constraints** (Beschränkungen)  $C$ : Eine Menge von Regeln, die spezifizieren, welche Kombinationen von Werten für Teilmengen der Variablen zulässig sind.

### Zustandszuweisungen

- **Partielle Zuweisung**: Nur einigen Variablen wurden Werte zugewiesen (z. B.  $X_1 = v_1$ ).
- **Konsistente (legale) Zuweisung**: Eine Zuweisung, die keinen Constraint verletzt.
- **Vollständige Zuweisung**: Jeder Variablen ist ein Wert zugewiesen.
- **Lösung**: Eine vollständige und konsistente Zuweisung.



Knoten = Variablen (Regionen), Kanten = Constraints (benachbart)

## 1.2 Arten von Constraints

Constraints schränken die möglichen Belegungen der Variablen ein:

- **Unäre Constraints:** Betreffen eine einzelne Variable (z. B.  $SA \neq \text{green}$ ).
- **Binäre Constraints:** Betreffen Paare von Variablen (z. B.  $SA \neq WA$ ). Dies ist die häufigste Form und kann durch einen **Constraint-Graphen** dargestellt werden.
- **Constraints höherer Ordnung:** Betreffen 3 oder mehr Variablen (z. B. bei Kryptogrammen wie  $TWO + TWO = FOUR$ ).
- **Soft Constraints (Präferenzen):** Dienen der Optimierung (z. B. "Rot ist besser als Grün"). Diese wandeln das CSP oft in ein *Constrained Optimization Problem* um.

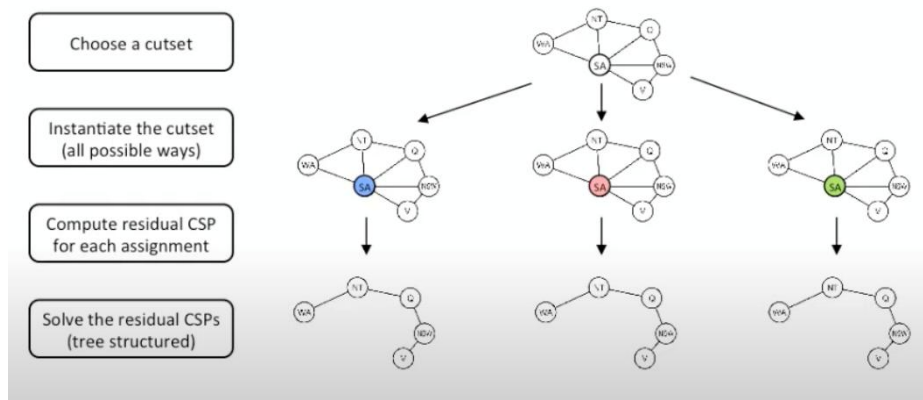
### 1.3 Lösungsansätze: Suche und Backtracking

Da die Reihenfolge der Zuweisungen bei CSPs keine Rolle spielt (Kommutativität), muss nicht der gesamte Zustandsraum als Baum mit  $n! \cdot v^n$  Blättern durchsucht werden. Es reicht aus, pro Ebene eine Variable zu betrachten. Dies reduziert den Suchraum auf  $v^n$  Blätter.

#### 1.3.1 Backtracking Search

Backtracking ist eine Tiefensuche (Depth-First Search), die für CSPs optimiert ist.

- Wähle eine unzugewiesene Variable.
- Probiere nacheinander alle Werte aus der Domäne.
- Wenn ein Wert konsistent ist: Weise ihn zu und rekursiere.
- Wenn ein Wert inkonsistent ist oder die Rekursion fehlschlägt: Mache die Zuweisung rückgängig (Backtrack) und versuche den nächsten Wert.



Ohne Heuristiken entspricht dies einer uninformierten Suche. Um die Effizienz zu steigern (z. B. Lösung des  $n$ -Damen-Problems für  $n > 25$ ), werden Heuristiken für die Auswahl von Variablen und Werten sowie Inferenzmethoden (Propagation) benötigt.

### 1.4 Heuristiken für Backtracking

Um die Suche zu beschleunigen, müssen an jedem Entscheidungspunkt drei Fragen beantwortet werden:

#### 1.4.1 1. Welche Variable soll als nächste belegt werden?

Hier gilt das Prinzip: *Fail-first* (Scheitere so früh wie möglich, um große Teile des Suchbaums abzuschneiden).

##### Variablen-Auswahl

- **Minimum Remaining Values (MRV):** Wähle die Variable mit den *wenigsten* verbleibenden legalen Werten. Dies führt am schnellsten zu einem Fehler, wenn ein Zweig keine Lösung enthält.
- **Degree Heuristic:** (Wird oft als Tie-Breaker für MRV genutzt). Wähle die Variable, die an den *meisten* Constraints mit anderen *noch nicht zugewiesenen* Variablen beteiligt ist.

## 1.4.2 2. Welchen Wert soll die Variable annehmen?

### Werte-Auswahl

**Least Constraining Value (LCV):** V  
benachbarten Variablen ausschließt. Dies

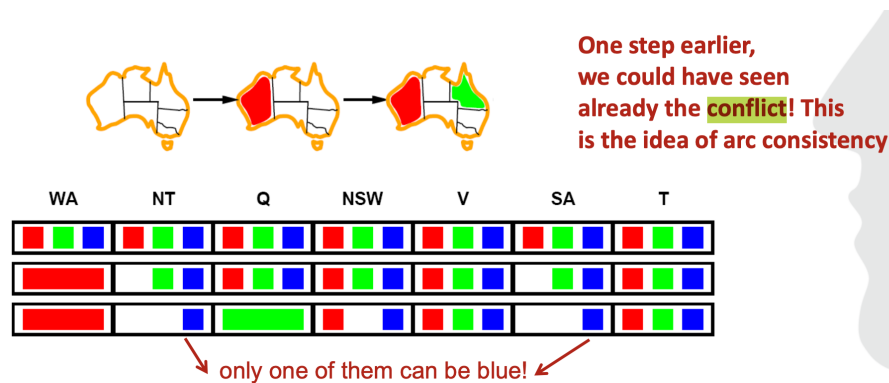
Hier gilt das Prinzip: *Fail-last* (Lasse möglichst viele Optionen für die Zukunft offen).

## 1.5 Constraint Propagation (Inferenz)

Anstatt nur zu suchen und bei Konflikten zurückzugehen, kann man durch Inferenz den Suchraum proaktiv verkleinern, indem man Werte ausschließt, die unmöglich Teil einer Lösung sein können.

### 1.5.1 Konsistenzarten

- **Node Consistency (Knotenkonsistenz):** Jede Variable erfüllt ihre unären Constraints.
- **Arc Consistency (Kantenkonsistenz):** Eine Variable  $X$  ist kantenkonsistent zu einer Variable  $Y$ , wenn für *jeden* Wert  $x \in D_X$  mindestens ein Wert  $y \in D_Y$  existiert, der den binären Constraint zwischen  $X$  und  $Y$  erfüllt.
- **Path Consistency / k-Consistency:** Verallgemeinerung auf Tripel oder  $k$  Variablen. Stärkere Konsistenz prüft mehr, ist aber rechenaufwendiger.



### 1.5.2 Forward Checking vs. Arc Consistency

- **Forward Checking:** Sobald  $X$  ein Wert zugewiesen wird, werden alle inkonsistenten Werte aus den Domänen der direkten Nachbarn entfernt. Wenn eine Domäne leer wird, backtrackt der Algorithmus. *Nachteil:* Erkennt Konflikte nicht früh genug (sieht z. B. nicht, dass zwei zukünftige Variablen inkonsistent zueinander geworden sind).
- **Arc Consistency (AC-3 Algorithmus):** Propagiert Constraints durch das gesamte Netzwerk. Wenn aus  $D_X$  ein Wert entfernt wird, müssen alle Nachbarn von  $X$  erneut geprüft werden, da deren Unterstützung verloren gegangen sein könnte.

#### AC-3 Algorithmus Kernidee

Verwende eine Warteschlange (Queue) aller Kanten (Arcs). Solange die Queue nicht leer ist:

1. Entnimm Kante  $(X_i, X_j)$ .
2. Entferne alle Werte aus  $D_i$ , die keinen Partner in  $D_j$  haben.
3. Wenn Werte aus  $D_i$  entfernt wurden: Füge alle Kanten  $(X_k, X_i)$  (Nachbarn, die von  $X_i$  abhängen) wieder zur Queue hinzu.

Die Kombination aus Backtracking und AC-3 wird als **MAC** (Maintaining Arc Consistency) bezeichnet.

## 1.6 Lokale Suche für CSPs

---

Für Probleme, bei denen der Pfad egal ist, kann auch Lokale Suche (Hill Climbing, Simulated Annealing) auf *vollständigen* (aber inkonsistenten) Zuweisungen arbeiten.

### Min-Conflicts Heuristik:

- Wähle zufällig eine Variable, die einen Constraint verletzt.
- Wähle für diese Variable den Wert, der die *Anzahl der verletzten Constraints* minimiert.

Diese Methode ist erstaunlich effizient (z. B. Million-Queens in Minuten), außer in einem kritischen Bereich des Verhältnisses von Constraints zu Variablen (*Critical Ratio*).

## 1.7 Struktur von CSPs

---

Die Struktur des Constraint-Graphen beeinflusst die Komplexität der Lösung massiv.

### 1.7.1 Problem-Dekomposition

---

Wenn ein Graph in unabhängige Teilgraphen zerfällt, kann die Komplexität von  $O(d^n)$  auf  $O(n/c \cdot d^c)$  reduziert werden (lineare Skalierung in  $n$ )

### 1.7.2 Baumstrukturierte CSPs

---

Wenn der Constraint-Graph ein Baum ist (keine Zyklen), kann das CSP in linearer Zeit  $O(n \cdot d^2)$  gelöst werden. **Algorithmus für Bäume:** *Topologische Sortierung:* Ordne Variablen so, dass jeder Knoten nach seinem Elternknoten kommt (Wurzel bis Blätter). *Rückwärts-Pass (Konsistenz):* Mache von den Blättern zur Wurzel hin alle Kanten gerichtete kantenkonsistent (entferne Werte im Elternknoten, die keine Entsprechung im Kind haben). *Vorwärts-Pass (Zuweisung):* Weise von der Wurzel zu den Blättern Werte zu. Da die Konsistenz hergestellt wurde, gibt es kein Backtracking.

### 1.7.3 Fast-Baum-Strukturen (Cutset Conditioning)

---

Für Graphen, die "fast" Bäume sind: Identifiziere ein **Cycle Cutset** (Menge von Variablen, deren Entfernung den Graphen zum Baum macht). Instantiiere die Variablen im Cutset (probiere alle Kombinationen). Löse den verbleibenden Baum für jede Instantiierung ("Residual CSP"). Die Laufzeit ist exponentiell nur in der Größe des Cutsets, nicht in  $n$ .