Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Intelligente Systeme

– Constraints –

Prof. Dr. Michael Neitzke

Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.

(Eugene C. Freuder, 1997)

C1: Einführendes Beispiel

C1: Lernziele

- V1: Gestalt eines Constraint-Netzes beschreiben können
- V2: Merkmale von Problemen, die sich mit Constraints modellieren lassen, nennen können

Ein Standard-Beispiel

```
  S E N D
+ M O R E
---------
M O N E Y
```

Wie löst man dieses Problem in (Standard-) PROLOG?

Erläuterungen

- Hier handelt es sich um ein kryptoarithmetisches Rätsel. Jedes Symbol (Großbuchstabe) steht für eine Ziffer. Unterschiedliche Symbole repräsentieren unterschiedliche Ziffern.
- Prolog arbeitet mit einer Tiefensuche. Man würde eine Lösung entwerfen und sie dann testen, d.h. versuchen zu beweisen.
  - Eine Verbesserungsidee könnte darin bestehen, zunächst nur die ganz rechte Spalte zu betrachten. Die Summe von D und E müsste dann an der Einer-Position Y ergeben. Wenn dieser Test einer Teillösung fehlschlägt, wird sofort eine andere Belegung getestet und erst bei Akzeptanz wird die nächste Spalte untersucht.

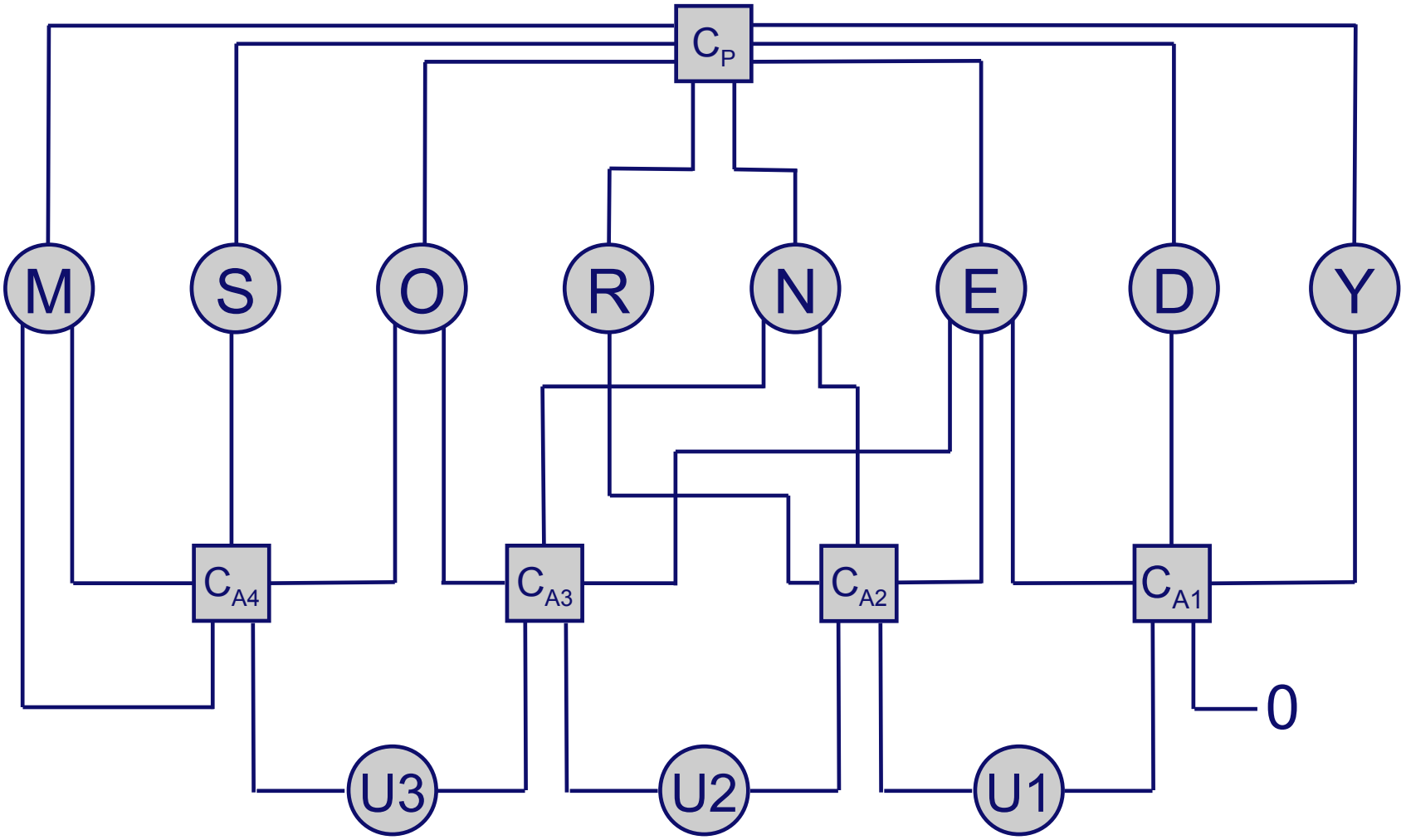
Diskutieren Sie!

- Was ist am Prolog-Programm nachteilig?
- Wie löst der Mensch dieses Problem?

Erläuterungen

- Beide Prolog-Programme haben den Nachteil, dass der Lösungsvorschlag einfach zufällig, d. h. ohne Wissen über das zu lösende Problem generiert wird. Ob die gesamte Lösung getestet wird oder zunächst nur eine Teillösung, ändert nichts am blinden Erzeugen der Lösung.
- Der Mensch macht logische Schlussfolgerungen. So kann er sehr schnell feststellen, dass M nur für die 1 stehen kann, denn durch Addition zweier vierstelliger Zahlen kann keine fünfstellige Zahl entstehen, die größer oder gleich 20000 ist. Daraus folgt dann, dass S entweder eine 8 oder eine 9 sein muss. Denn sonst könnte kein Übertrag bei der Addition von S und M und einem Übertrag aus der nächsten Spalte entstehen. O muss dann für die 0 stehen, denn die 1 ist ja schon vergeben und eine 2 kann nicht erreicht werden. Und so weiter.

Constraint-Netz für SEND + MORE = MONEY



Erläuterungen

- Variablen sind durch Kreise und Constraints durch Quadrate dargestellt. Die Ein-/Ausgänge der Constraints nennt man Pins.
- Es gibt in diesem Beispiel zwei unterschiedliche Klassen von Constraints.
  - C_P ist ein „All-different"-Constraint, der dafür sorgt, dass alle seine Variablen unterschiedliche Werte aufweisen.
  - C_A ist ein Constraint, der dafür sorgt, dass die Summe der Werte der Variablen am linken Pin, am oberen Pin und am Pin unten rechts mit den Werten der Variablen rechts und unten links übereinstimmt, wobei die Einer-Ziffer der Summe rechts und die Zehner-Ziffer unten links anliegt.

Andere Beispiele

- Konfiguration technischer Systeme
  - Fahrerlose Transportsysteme
  - Industrie-Rührer
- Planungsaufgaben
  - Produktionsplanung
  - Stundenplan-Erstellung
- Kombinatorische Optimierung
  - Transport-Optimierung

Referenzen

- Barták: "Guide To Constraint Programming"
  - http://kti.mff.cuni.cz/~bartak/constraints/
- Guesgen: "Constraints"
  - in Görz, Rollinger, Schneeberger: "Handbuch der Künstlichen Intelligenz"
- Winston: "Künstliche Intelligenz" (Waltz-Verfahren)
- http://www.cs.kuleuven.ac.be/~dtai/projects/CHR

C1: Lernziele

- V1: Gestalt eines Constraint-Netzes beschreiben können
- V2: Merkmale von Problemen, die sich mit Constraints modellieren lassen, nennen können

C2: Grundbegriffe

C2: Lernziele

- W1: Definitionen nennen können für
  - Constraint
  - Unärer Constraint
  - Binärer Constraint
  - Constraint-Netz
  - Constraint Satisfaction Problem

Was ist ein Constraint?

- Constraint **C** drückt Relation zwischen endlicher Menge von Variablen x_i aus
  - Jede Variable x_i hat eigenen Wertebereich D_i
  - endlich oder unendlich

- Relation: Teilmenge des Kreuzprodukts der betroffenen Wertebereiche

$$C \subseteq D_1 \times D_2 \times .. \times D_k$$

Unärer Constraint, binärer Constraint

- Unärer Constraint
  - einstellig
  - bezieht sich auf nur eine Variable
  - kann durch Vorverarbeitung eliminiert werden
    - Einschränkung der Wertebereiche entsprechend des unären Constraints
- Binärer Constraint
  - zweistellig

Constraint-Netz

- Constraint-Netz
  - aus p Constraints: $C_1, \ldots, C_p$
  - auf m Variablen $x_1, \ldots, x_m$
- Jeder Constraint bezieht sich auf Teilmenge der Variablen
- Das Netz entsteht durch gemeinsame Variablen in verschiedenen Constraints

Constraint Satisfaction Problem

- Ein k-stelliger Constraint C_i ist erfüllt, wenn ein Wertetupel

  $(w_1, \ldots, w_k) \in D_1 \times D_2 \times .. \times D_k$

  die Relation des Constraints erfüllt.

- Wenn alle Constraints $C_1, \ldots, C_p$ eines Constraint-Netzes durch ein Wertetupel $(w_1, \ldots, w_m)$ (bzw. durch die für den jeweiligen Constraint relevanten Werte des Wertetupels) erfüllt sind, so liegt eine Lösung des Constraint-Netzes vor.

- **Constraint Satisfaction Problem:**
  Lösung für ein Constraint-Netz finden

C2: Lernziele

- W1: Definitionen nennen können für
  - Constraint
  - Unärer Constraint
  - Binärer Constraint
  - Constraint-Netz
  - Constraint Satisfaction Problem

C3: Lösungsansätze

C3: Lernziele

- V1: Lösungsverfahren beschreiben können:
  - Generate-and-Test
  - Backtracking (Details folgen später)
  - Waltz-Filtern (Details folgen später)
- Programmcode für Algorithmen ist nicht prüfungsrelevant

Wie löst man Constraint Satisfaction Problems?

- Bereits kennengelernt: Generate-and-Test

- Ablauf:
  - Erzeugen eines vollständigen Lösungsvorschlags
  - Testen des Vorschlags
  - Wenn Test fehlschlägt, nächster Vorschlag (systematisch erzeugt)

- Eigenschaften:
  - Sehr aufwendig
  - Constraints werden zum Testen einer Lösung, aber nicht zum Erzeugen einer Lösung verwendet

Algorithmus: Generate-and-Test

```
gt(Variables,Constraints,Solution):-
    generate(Variables,Solution),
    test(Constraints,Solution).

generate([V::D|RemainingVariables],
              [V-X|PartialSolution]):-
    select_value(X,D),
    generate(RemainingVariables,PartialSolution).

generate([],[]).

test([C|RemainingConstraints],Solution):-
    test_constraint(C,Solution),
    test(RemainingConstraints,Solution).

test([],_).
```

Durch Backtracking unterschiedliche Zuweisungen

Zusatzinformation

Backtracking

- Ablauf:
  - Variablen werden nach und nach instantiiert
  - Tests, sobald möglich
  - Wenn Test scheitert, wird letzte Instantiierung zurückgezogen
    - -> „Chronologisches Backtracking“
- Eigenschaften:
  - + Teile des Suchbaums werden frühzeitig abgeschnitten
  - - Nicht die wahre Ursache, sondern die letzte Entscheidung wird zurückgezogen
  - - Keine Nutzung der Constraints zur Erzeugung der Lösung

Algorithmus: Suche mit Backtracking

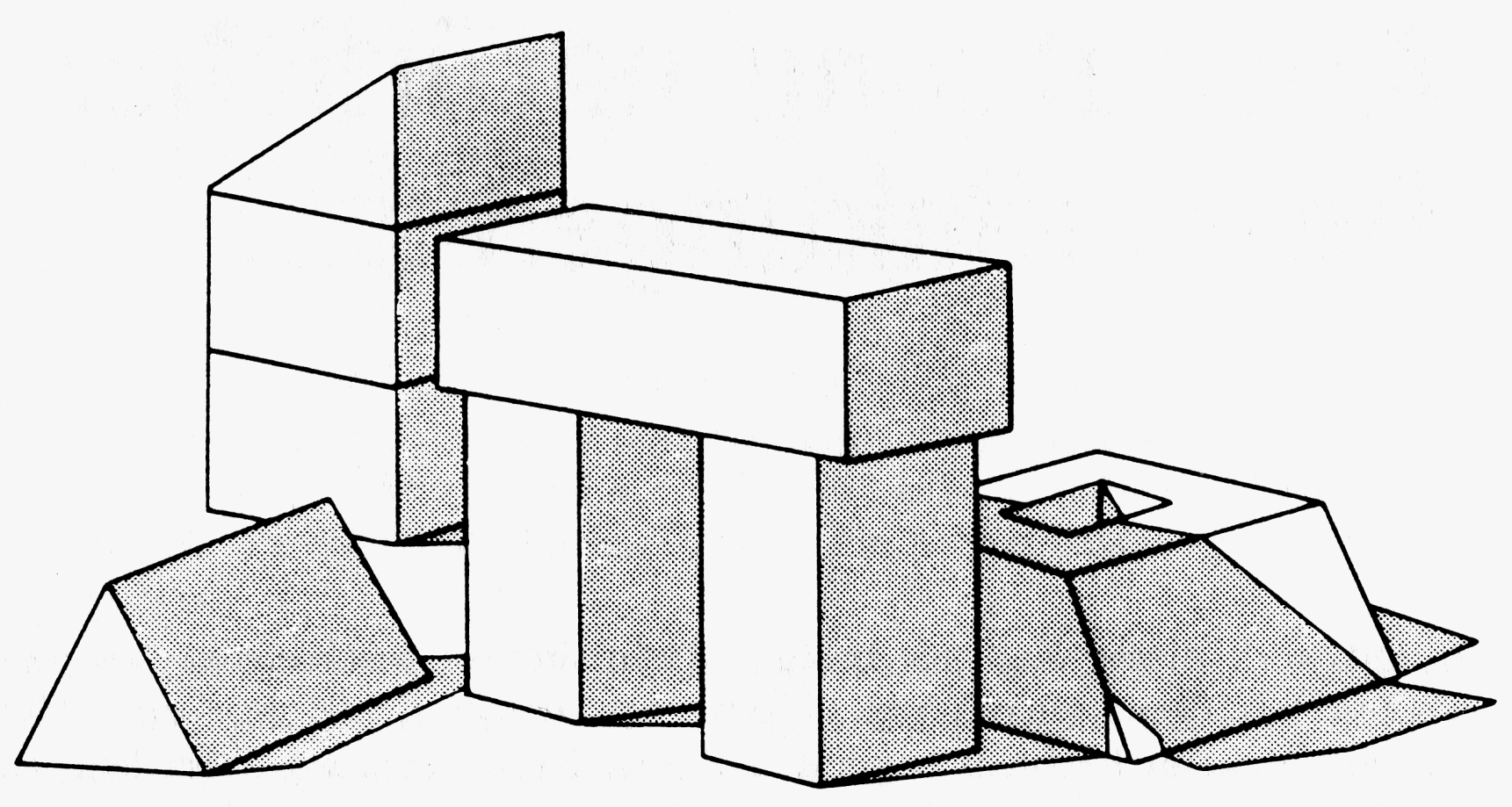
```
bt([V::D|RemainingVariables],Constraints,PartialSolution,Solution):-
  select_value(X,D),
  NewPartialSolution=[V-X|PartialSolution],
  test(Constraints,NewPartialSolution,RemainingConstraints),
  bt(RemainingVariables,RemainingConstraints,NewPartialSolution,
     Solution).

bt([],[],Solution,Solution).

test([C|RemainingConstraints],PartialSolution,NonTestedConstraints):-
  (can_be_tested(C,PartialSolution)
    -> test_constraint(C,PartialSolution),
                                   NonTestedConstraints=Constraints ;

       NonTestedConstraints=[C|Constraints]),
  test(RemainingConstraints,PartialSolution,Constraints).

test([],_,[]).

can_be_tested(Constraint,PartialSolution):-
  variables(Constraint,Variables),
  are_instantiated(Variables,PartialSolution).
```

Zusatzinformation

Erzeugung der Lösung durch Constraints

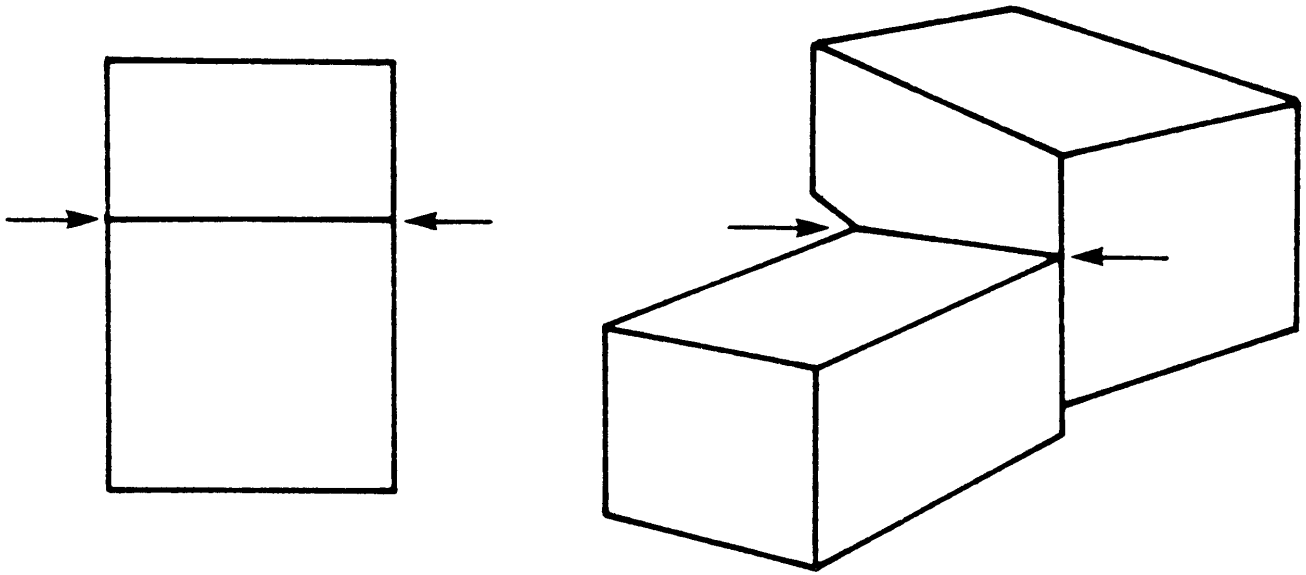
- Eine Möglichkeit für endliche Wertebereiche: **Waltz-Filtern**

Anwendungsbsp.: Problem der Linieninterpretation



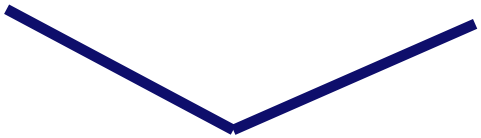
Vereinfachende Annahmen (für Erklärungszwecke)

- Keine Schatten, keine Bruchlinien
- Eckpunkte dreier Flächen (z.B. keine Pyramide)
- Allgemeiner Beobachterstandpunkt (geringe Änderungen des Beobachterstandpunktes bewirken keine Änderung der Schnittpunk` `

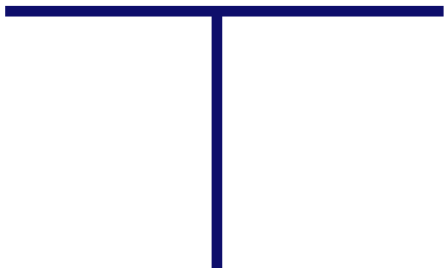


Nur vier verschiedene Arten von Schnittpunkten

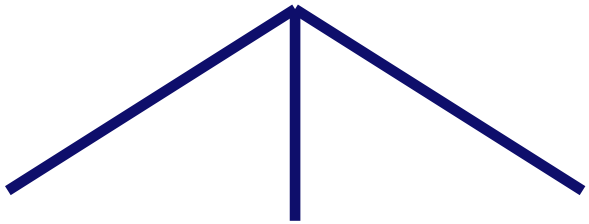
- L:



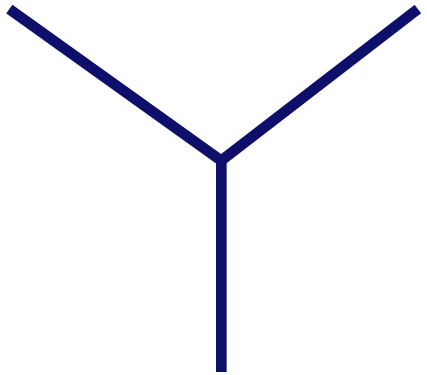
- T:



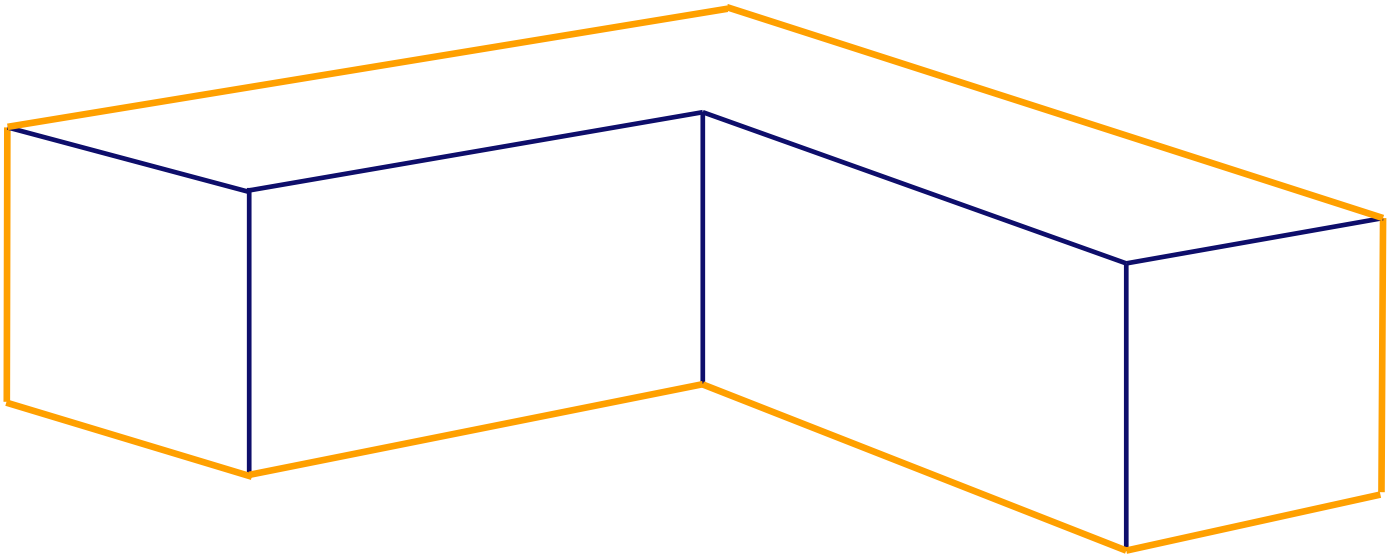
- Pfeil:



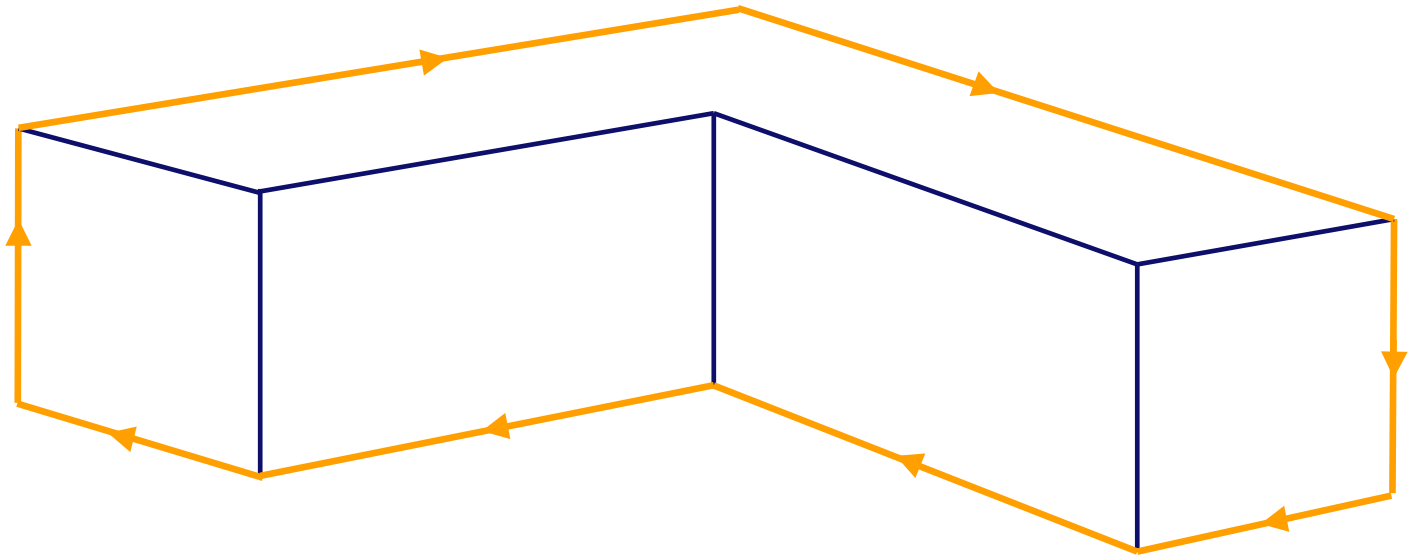
- Gabel:



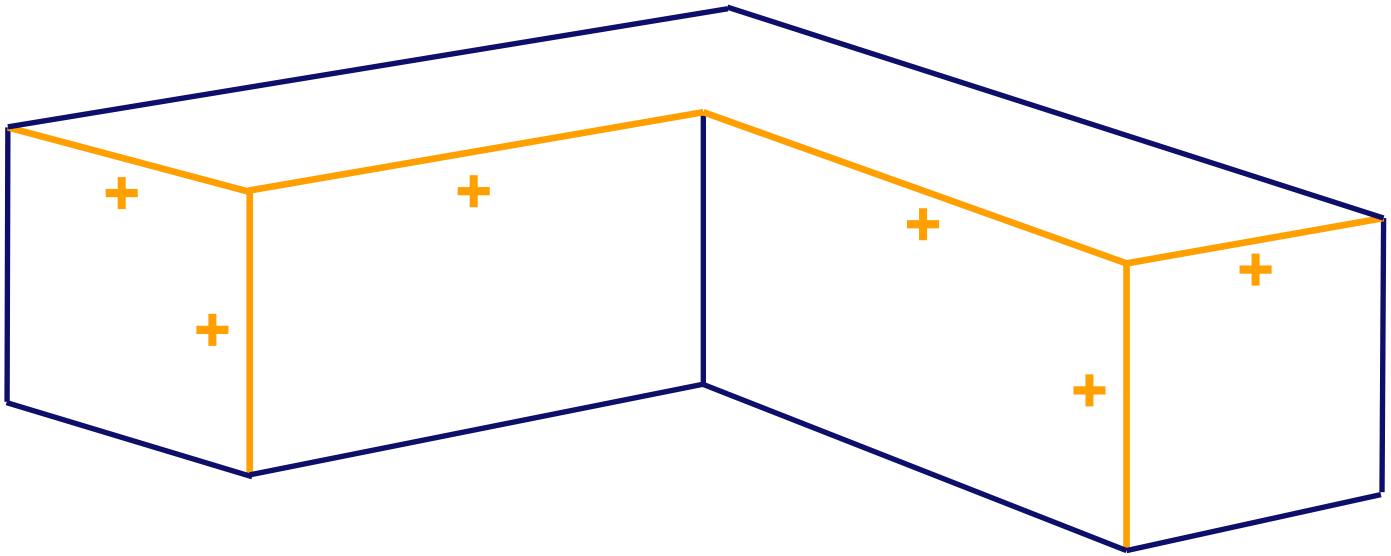
Linientypen: Begrenzungslinien



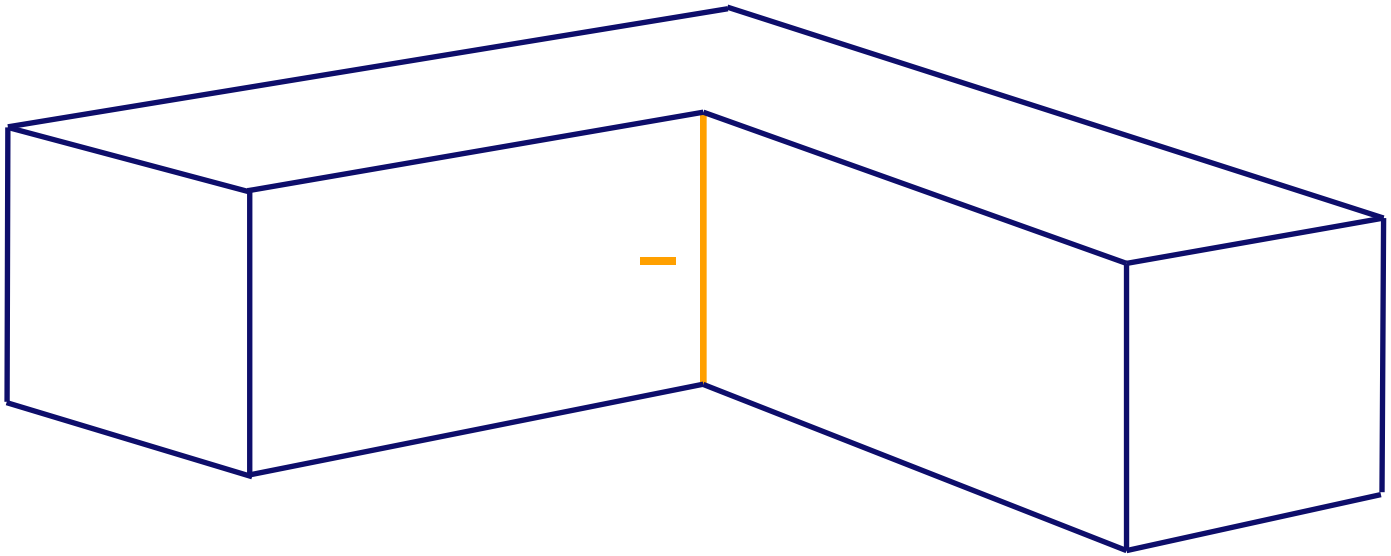
Linienmarken für Begrenzungslinien



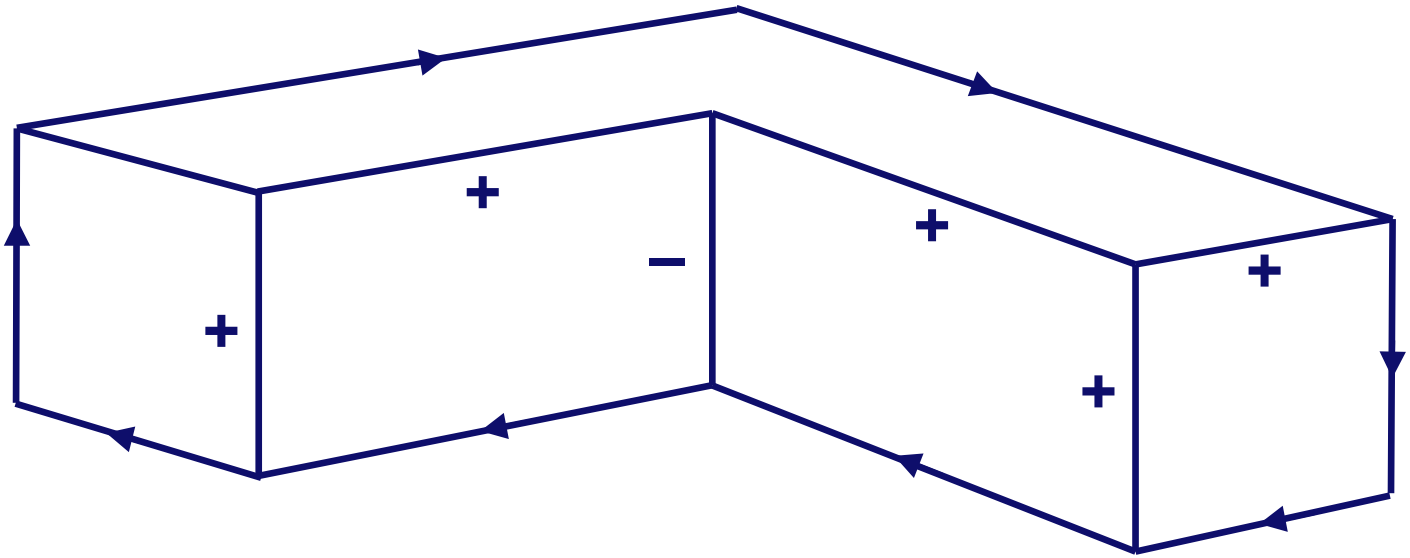
Linientypen: Konvexe Linien mit Kennzeichnung



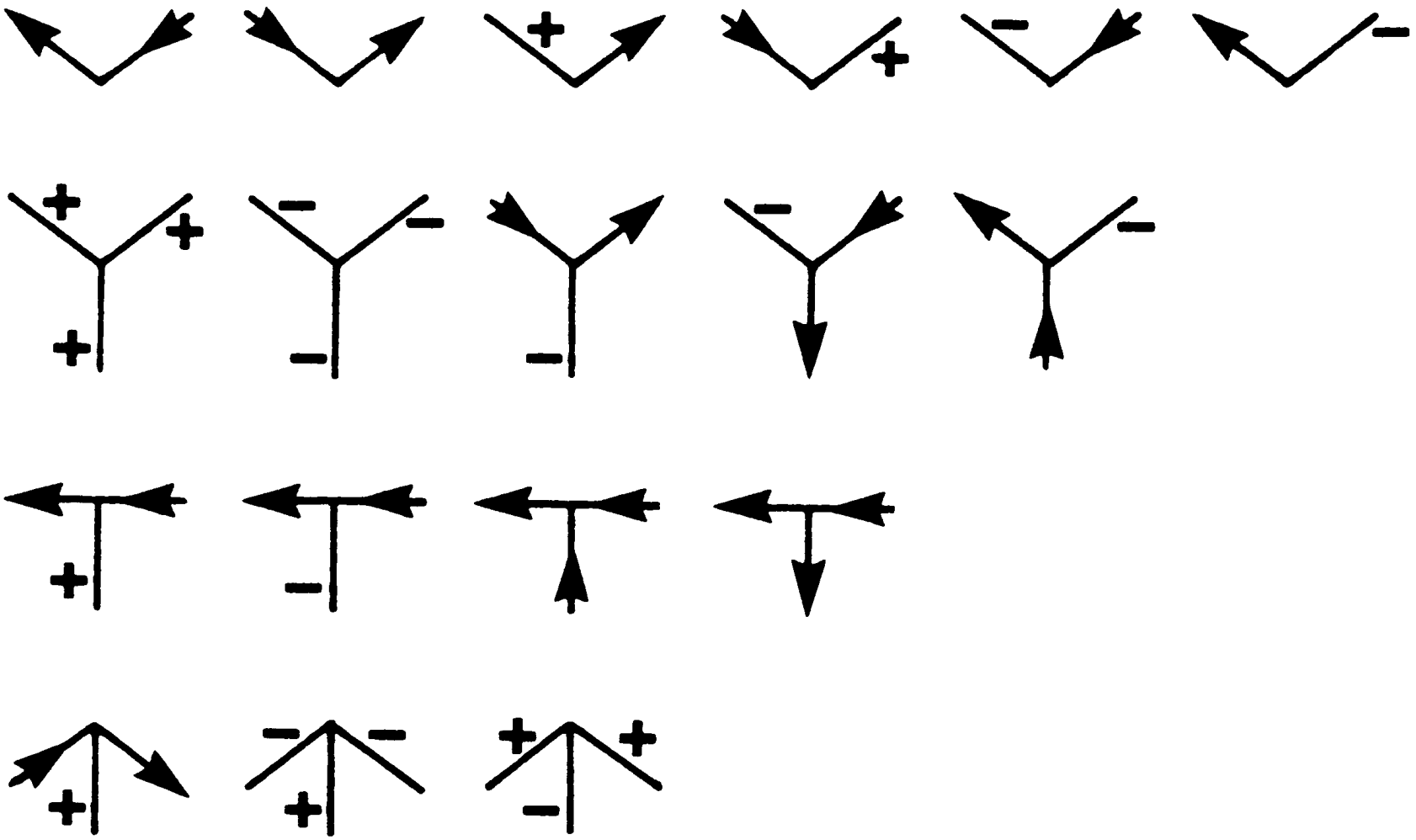
Linientypen: Konkave Linien mit Kennzeichnung



Vollständige Kennzeichnung

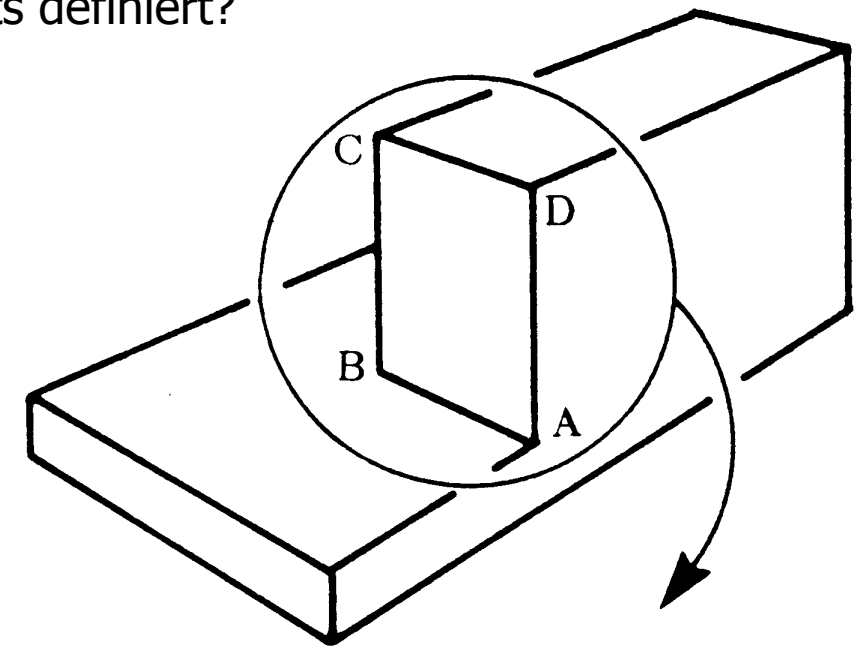


Nur 18 verschiedene Schnittpunktkonfigurationen



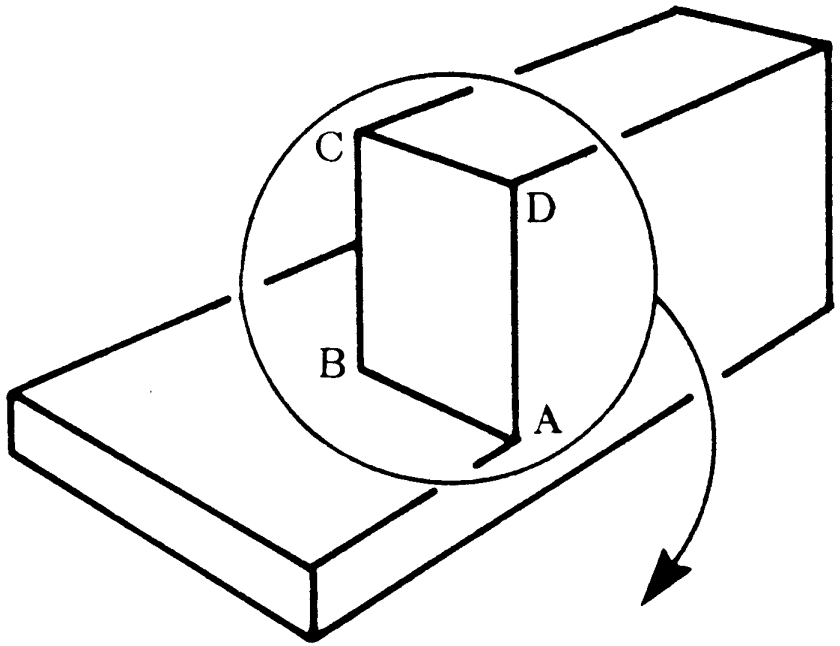
Übung (1)

- Modellieren Sie!
  - Was sind die Variablen?
  - Welche Wertebereiche haben die Variablen?
  - Was sind die Constraints?
  - Wie sind die Constraints definiert?

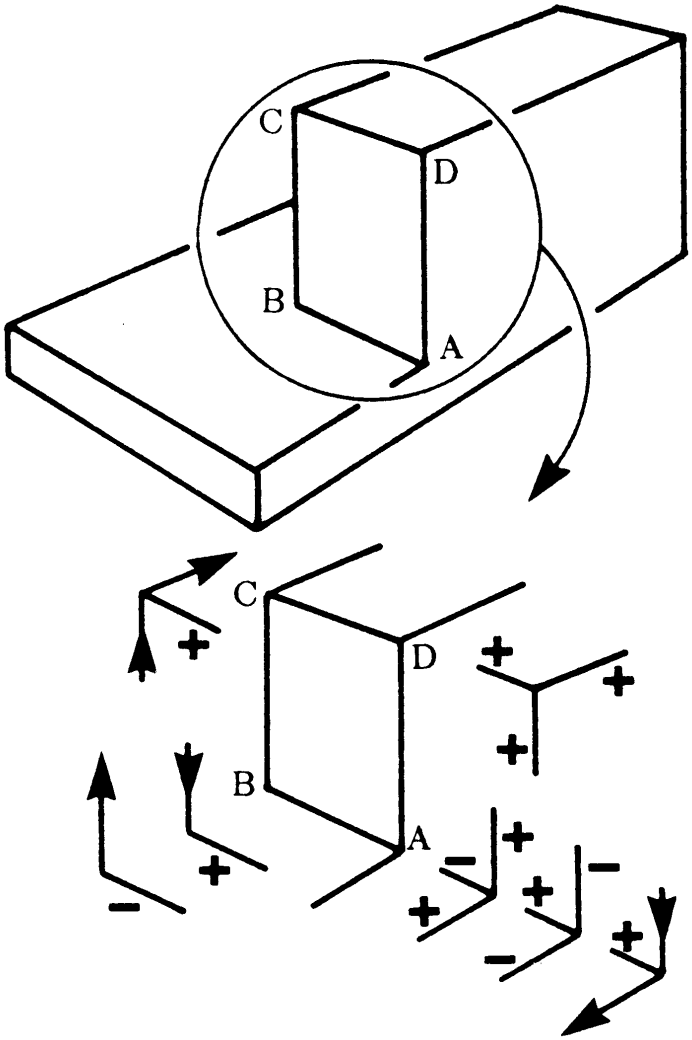


Übung (2)

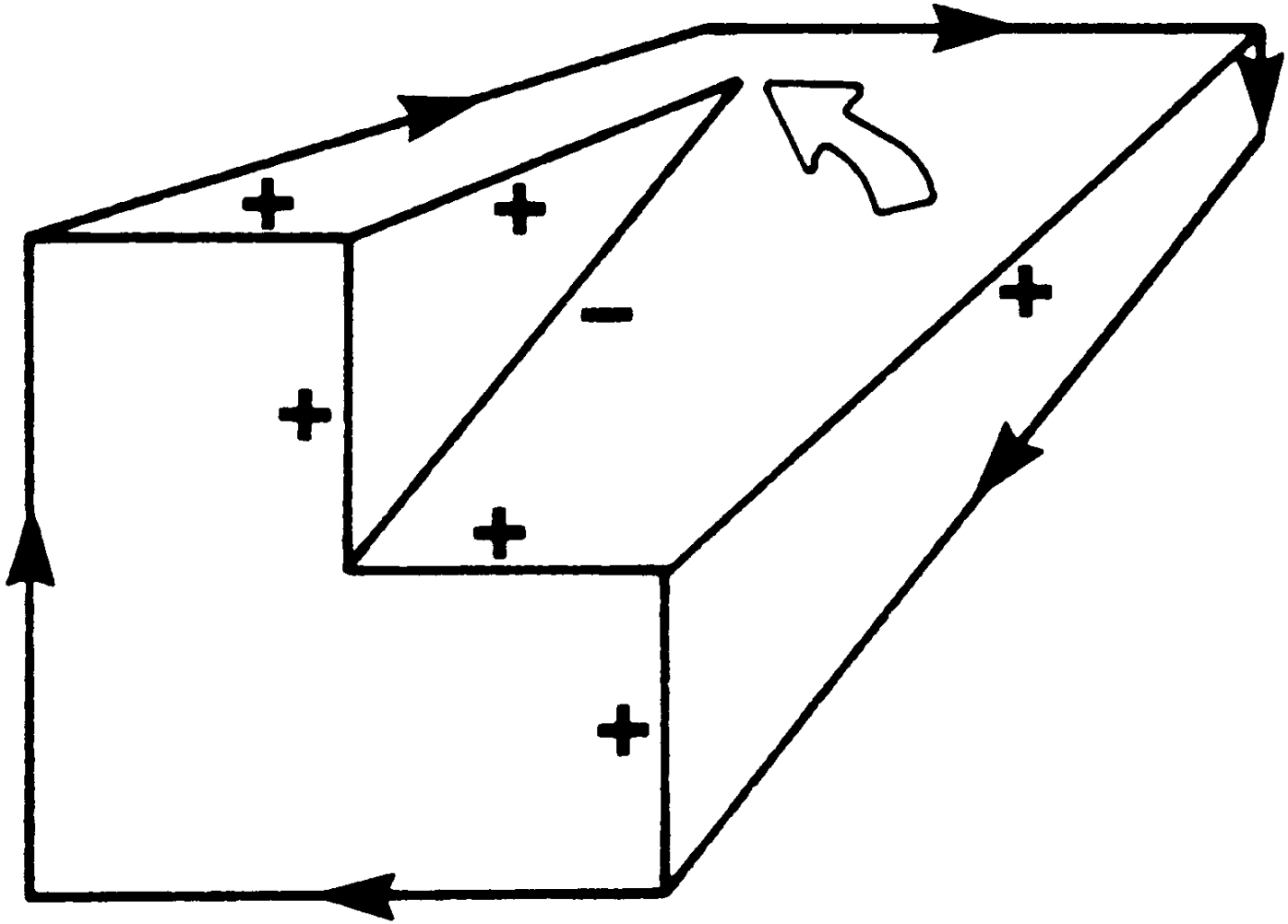
- Untersuchen Sie den Constraint beim Punkte A und stellen Sie die Auswirkungen auf die betroffenen Variablen fest.
- Tun Sie dasselbe für die anderen Punkte in sinnvoller Reihenfolge.



Waltz-Filtern: Lösungsansatz



Erkennen nicht-möglicher Körper



C3: Lernziele

- V1: Lösungsverfahren beschreiben können:
  - Generate-and-Test
  - Backtracking (Details folgen später)
  - Waltz-Filtern (Details folgen später)
- Programmcode für Algorithmen ist nicht prüfungsrelevant

C4: Verfahren zur Herstellung lokaler Konsistenz

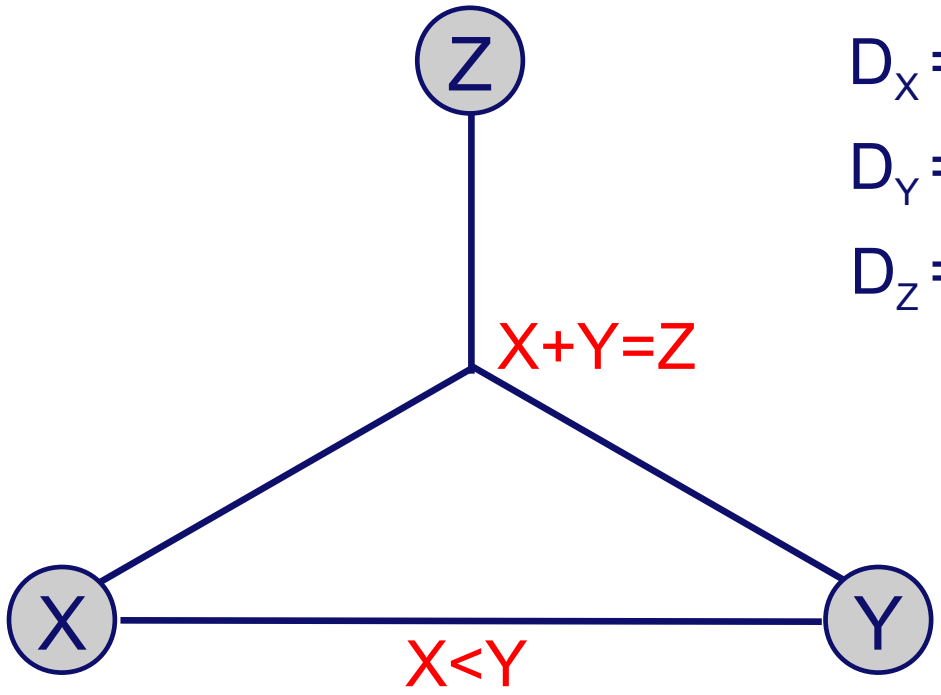
- Binarisierung
- Formen der Konsistenz
- Algorithmen zur Herstellung lokaler Konsistenz

C4: Lernziele

- V1: Motivation für Binarisierung erläutern können
- V2: Die beiden Formen der Binarisierung erläutern können
- V3: Begriffe Knoten-, Kanten und lokale Konsistenz erläutern können
- V4: Erläutern können, wie Knoten- oder Kantenkonsistenz erreicht werden kann
- V5: Algorithmen REVISE, AC-1, AC-3 und AC-4 in ihrer Arbeitsweise mit Vor- und Nachteilen erläutern können, AC4 nicht im Detail
- V6: Für REVISE, AC-1 und AC-3 Detailfragen zu konkreten Aufgabenstellungen beantworten können, für AC-4 nur bzgl. der Zähler und Abhängigkeitsmengen
- A1: AC-1 und AC-3 auf konkrete Aufgabenstellungen anwenden können (-> Teil 2 der Klausur)

Binäre Constraint-Netze

- Algorithmen zur Lösung von CSP: Meist für binäre Constraint-Netze
- Jedes Constraint-Netz ist in ein äquivalentes binäres Constraint-Netz überführbar
- Beispiel:



$D_X = \{1,2\}$

$D_Y = \{3,4\}$

$D_Z = \{5,6\}$

Binarisierung eines Constraint-Netzes

- Einführung einer zusätzlichen Variablen
  - "einschließend" / "encapsulated"
- Variable hat Kreuzprodukt als potenziellen Wertebereich:

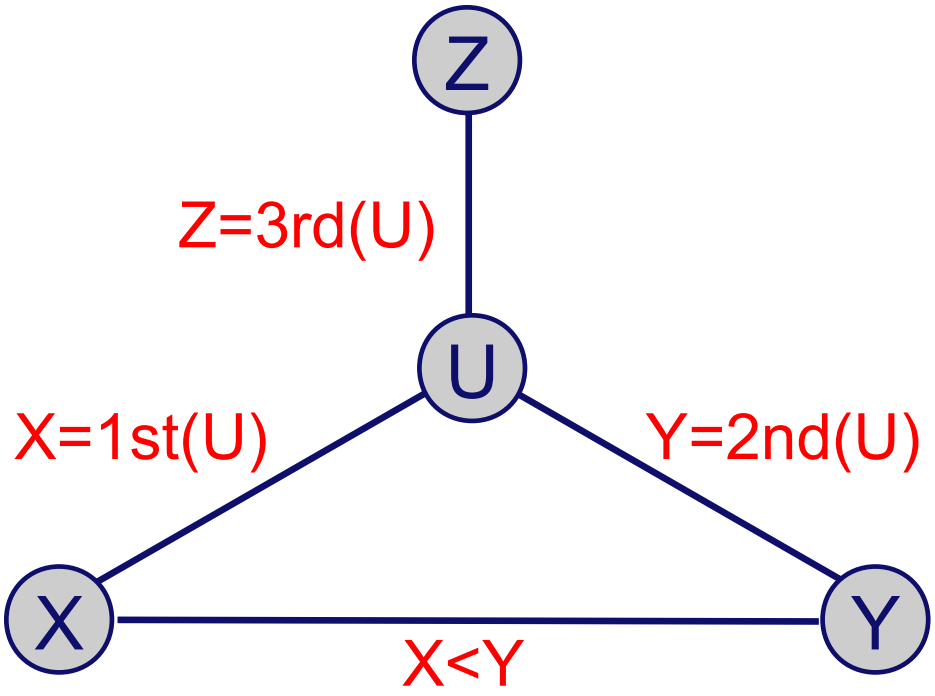
$$D_U = \{(1,3,5), (1,3,6), (1,4,5), (1,4,6), (2,3,5), (2,3,6), (2,4,5), (2,4,6)\}$$

- Zur Verwendung im Constraint-Netz wird Wertebereich auf solche Tupel eingeschränkt, die den zu ersetzenden Constraint erfüllen:

$$D_U = \{(1,4,5), (2,3,5), (2,4,6)\}$$

Zwei Möglichkeiten zur Binarisierung (1)

1. Neues Netz beinhaltet Original-Variablen (hidden variable encoding)



$D_X = \{1,2\}$

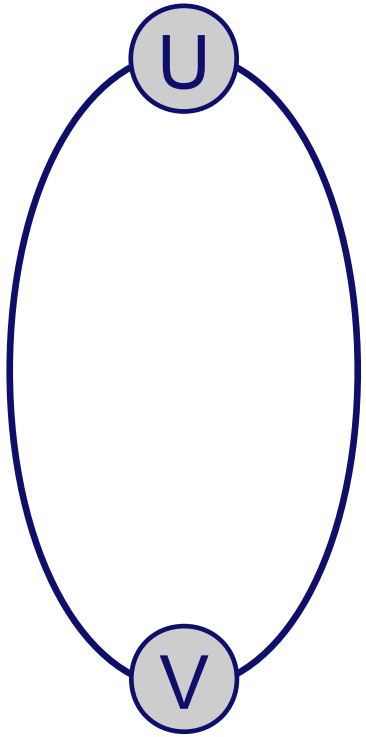
$D_Y = \{3,4\}$

$D_Z = \{5,6\}$

$D_U = \{(1,4,5), (2,3,5), (2,4,6)\}$

Zwei Möglichkeiten zur Binarisierung (2)

1. Neues Netz enthält nur einschließende Variablen (dual encoding)



$D_U = \{(1,4,5), (2,3,5), (2,4,6)\}$

$D_V = \{(1,3), (1,4), (2,3), (2,4)\}$

1st(U)=1st(V)

2nd(U)=2nd(V)

Konsistenz

- Knotenkonsistenz
  - Alle unären Constraints sind erfüllt
- Kantenkonsistenz
  - Binäres Netz
  - Jeder Constraint ist – lokal betrachtet – erfüllt
- Lokale Konsistenz
  - Jeder Constraint ist – lokal betrachtet – erfüllt

Algorithmus NC zur Knotenkonsistenz

```
procedure NC
   for each V in nodes(G)
     for each X in the domain D of V
       if any unary constraint on V
                          is inconsistent with X
        then
          delete X from D;
       endif
     endfor
   endfor
end NC
```

Algorithmus REVISE: Beitrag zur Kantenkonsistenz

```
procedure REVISE(Vi,Vj)
   DELETE <- false;
   for each X in Di do
     if there is no such Y in Dj such that
                              (X,Y) is consistent,
     then
       delete X from Di;
       DELETE <- true;
     endif;
   endfor;
   return DELETE;
end REVISE
```

D.h., ist Element der Relation, die durch den X,Y betreffenden Constraint definiert ist.

Was bewirkt REVISE?

Erklärung

- REVISE löscht nur Werte von **`Vi`**
- Und zwar diejenigen, für die es keinen den Constraint erfüllenden Wert von **`Vj`** gibt

Bewertung REVISE

- Selbst Überprüfung aller V_i, V_j auch in umgekehrter Richtung, reicht nicht für Kantenkonsistenz
  - Entfernung eines Wertes aus D_i kann Auswirkungen auf andere Constraints haben. Dann ist erneute Überprüfung bereits untersuchter Constraints notwendig

Algorithmus AC-1 zur Kantenkonsistenz

```
procedure AC-1
   Q <- {(Vi,Vj) in arcs(G),i#j};
   repeat
      CHANGE <- false;
      for each (Vi,Vj) in Q do
         CHANGE <- REVISE(Vi,Vj) or CHANGE;
      endfor
   until not(CHANGE)
end AC-1
```

ungleich

Übung

- Verdeutlichen Sie sich die Arbeitsweise des Algorithmus AC-1 an folgendem Beispiel:

  X = V
  X *2 = Z
  X < Y
  Y = Z

  Alle Wertemengen initial: **{1, 2, 3, 4}**

- Q = {(X,V), (V,X), (X,Z), (Z,X), (X,Y), (Y,X), (Y,Z), (Z,Y)}
- Was ist nachteilig?

Lösung (Reihenfolge kann auch anders sein)

- Q = {(X,V), (V,X), (X,Z), (Z,X), (X,Y), (Y,X), (Y,Z), (Z,Y)}
- Repeat-Schleife:
  - REVISE(X,V) bewirkt nichts
  - REVISE(V,X) bewirkt nichts
  - REVISE(X,Z) bewirkt X∈{1,2}
  - REVISE(Z,X) bewirkt Z∈{2,4}
  - REVISE(X,Y) bewirkt nichts
  - REVISE(Y,X) bewirkt Y∈{2,3,4}
  - REVISE(Y,Z) bewirkt Y∈{2,4}
  - REVISE(Z,Y) bewirkt nichts

Lösung (2)

- Repeat-Schleife (zweiter Durchlauf):
  - REVISE(X,V) bewirkt nichts
  - REVISE(V,X) bewirkt V∈{1,2}
  - REVISE(X,Z) bewirkt nichts
  - REVISE(Z,X) bewirkt nichts
  - REVISE(X,Y) bewirkt nichts
  - REVISE(Y,X) bewirkt nichts
  - REVISE(Y,Z) bewirkt nichts
  - REVISE(Z,Y) bewirkt nichts

Lösung (3)

- Repeat-Schleife (dritter Durchlauf):
  - REVISE(X,V) bewirkt nichts
  - REVISE(V,X) bewirkt nichts
  - REVISE(X,Z) bewirkt nichts
  - REVISE(Z,X) bewirkt nichts
  - REVISE(X,Y) bewirkt nichts
  - REVISE(Y,X) bewirkt nichts
  - REVISE(Y,Z) bewirkt nichts
  - REVISE(Z,Y) bewirkt nichts

Bewertung AC-1

- Jede Einschränkung einer Wertemenge bewirkt Überprüfen aller Kanten.
- Aber viele Kanten sind möglicherweise von der Einschränkung gar nicht betroffen.

Algorithmus AC-3 zur Kantenkonsistenz

```
procedure AC-3
    Q <- {(Vi,Vj) in arcs(G),i#j};
    while not Q empty
      select and delete any arc (Vk,Vm)
                                  from Q;
      if REVISE(Vk,Vm) then
        Q <- Q union {(Vi,Vk) such that
                (Vi,Vk) in arcs(G),i#k,i#m}
      endif
    endwhile
 end AC-3
```

REVISE schränkt nur Wertebereich von Vk ein

Diskutieren Sie!

- Welche Variablenpaare befinden sich initial in **Q** ?
- Welche Variablenpaare werden ggf. in **Q** hinzugefügt?
- Weshalb werden die Bedingung **i#k,i#m** erhoben?

Lösung

- Initial sind in Q alle Paare von zwei verschiedenen Variablen. Dabei kommt jedes Paar doppelt vor, weil jede der beiden Variablen einmal an erster und einmal an zweiter Position befindet.
- Wenn eine Variable Vk eingeschränkt wird, dann werden alle Paare (Vi,Vk) hinzugefügt. Es werden also alle Variablen Vi überprüft, die mit Vk verbunden sind (die Nachbarvariablen von Vk).
- Das Paar (Vk,Vk) muss natürlich nicht überprüft werden. Und das Paar (Vm,Vk) auch nicht, weil (Vm,Vk) ohnehin in Q vorhanden ist oder bereits überprüft wurde. Wenn (Vm,Vk) bereits überprüft wurde, kann durch die gerade erfolgte Einschränkung von Vk keine zusätzliche Einschränkung von Vm entstehen. Denn es wurden von Vk ja nur die Werte gelöscht, für die es gerade keinen Partnerwert für Vm gab.

Übung

- Verdeutlichen Sie sich die Arbeitsweise des Algorithmus AC-3 an folgendem Beispiel:

  X = V
  X *2 = Z
  X < Y
  Y = Z

- Alle Wertemengen initial: **{1, 2, 3, 4}**
- Q = {(X,V), (V,X), (X,Z), (Z,X), (X,Y), (Y,X), (Y,Z), (Z,Y)}
- LIFO für Abarbeitung von Q
  - Kluge Sortierung über Heuristiken

Lösung AC-3 (Sortierung in Q auch anders möglich)

- Q = {(X,V), (V,X), (X,Z), (Z,X), (X,Y), (Y,X), (Y,Z), (Z,Y)}
- While-Schleife:
  - REVISE(X,V) bewirkt nichts
    - Q = {(V,X), (X,Z), (Z,X), (X,Y), (Y,X), (Y,Z), (Z,Y)}
  - REVISE(V,X) bewirkt nichts
    - Q = {(X,Z), (Z,X), (X,Y), (Y,X), (Y,Z), (Z,Y)}
  - REVISE(X,Z) bewirkt X∈{1,2}
    - Q = {(V,X), (Z,X), (X,Y), (Y,X), (Y,Z), (Z,Y)}
  - REVISE(V,X) bewirkt V∈{1,2}
    - Q = {(Z,X), (X,Y), (Y,X), (Y,Z), (Z,Y)}
  - REVISE(Z,X) bewirkt Z∈{2,4}
    - Q = {(Y,Z), (X,Y), (Y,X), (Z,Y)}

Lösung (2)

- REVISE(Y,Z) bewirkt Y∈{2,4}
  - Q = {(X,Y), (Y,X), (Z,Y)}
- REVISE(X,Y) bewirkt nichts
  - Q = {(Y,X), (Z,Y)}
- REVISE(Y,X) bewirkt nichts
  - Q = {(Z,Y)}
- REVISE(Z,Y) bewirkt nichts
  - Q = {}

- Also neun statt 24 Aufrufe von REVISE

Bewertung AC-3

- Verbesserung:
  - Nur von Werteeinschränkung betroffene Kanten (= Variablenpaare) werden erneut untersucht

Diskutieren Sie!

- Wodurch entsteht im AC-3 genau der Bedarf für erneutes Testen von Constraints?
- In welchen Fällen hat sich das erneute Testen eines Constraints gelohnt? In welchen Fällen war es doch überflüssig?
- Welche Information würde helfen, um vorab zu wissen, ob sich das erneute Testen eines Constraints lohnen wird?

Lösung

- Wodurch entsteht Bedarf für erneutes Testen von Constraints?
  - Wert **b** einer Variablen V_j wurde entfernt
  - Wert **b** war Bestandteil eines konsistenten Wertepaares **(a,b)** bezüglich eines anderen, bereits vorher untersuchten Constraints (V_i, V_j)
  - Möglicherweise muss Wert **a** entfernt werden
  - Dazu muss untersucht werden, ob es noch andere Werte gibt, die mit Wert **a** konsistent sind, ihn also „unterstützen"
- **a** Wert **a** muss genau dann ebenfalls entfernt werden, wenn es keinen weiteren Wert von V_j gibt, der **a** unterstützt.
- Es würde helfen, wenn man vorab wüsste, ob es noch andere unterstützende Werte gibt
  - Oder wie viele unterstützende Werte es insgesamt gibt, so dass man dann immer herunter zählen kann, wenn ein unterstützender Wert gelöscht wird

Grundidee für Algorithmus AC-4

- Für jeden Wert jeder Variablen V_i werden die unterstützenden Werte bezüglich jeder mit V_i durch einen Constraint verbundenen Variablen V_j im Rahmen einer Vorab-Analyse des Constraint-Netzes gezählt.
- Beim Löschen von Werten werden die betroffenen Zähler um 1 dekrementiert
  - Wenn der Zähler auf 0 steht, bedeutet dies, dass der Wert gelöscht wird
- Außer den Zählern gibt es noch für jeden Wert jeder Variablen eine Menge von Wert/Variablen-Paaren, die von ihm unterstützt werden
  - Wenn ein Zähler die 0 erreicht hat, kann auf diese Weise festgestellt werden, welche Zähler um 1 vermindert werden müssen. Dies ist für den AC-4 die Constraint-Propagierung.

Algorithmus AC-4: Initialisierung

```
procedure INITIALIZE
  Q <- {}; S <- {}; % initialize each element of S
  for each (Vi,Vj) in arcs(G) do %(Vj,Vi) and (Vi,Vj) are same elements
    for each a in Di do
      total <- 0;
      for each b in Dj do
        if (a,b) is consistent according to the constraint (Vi,Vj) then
          total <- total+1;
          Sj,b <- Sj,b union {<i,a>};
        endif
      endfor;
      counter[(i,j),a] <- total;
      if counter[(i,j),a]=0 then
        delete a from Di;
        Q <- Q union {<i,a>};
      endif;
    endfor;
  endfor;
  return Q;
end INITIALIZE
```

Welche Werte werden durch $S_{j,b}$ gestützt? Also durch den Wert b der Variablen Vj.

total: Anzahl der unterstützenden Werte der Variablen V_j für Wert a der Variablen V_i

Algorithmus AC-4 zur Kantenkonsistenz

```
procedure AC-4
  Q <- INITIALIZE;
  while not Q empty
    select and delete any pair <j,b> from Q;
    for each <i,a> from Sj,b do
      counter[(i,j),a] <- counter[(i,j),a] - 1;
      if counter[(i,j),a]=0 & a is still in Di then
        delete a from Di;
        Q <- Q union {<i,a>};
      endif
    endfor
  endwhile
end AC-4
```

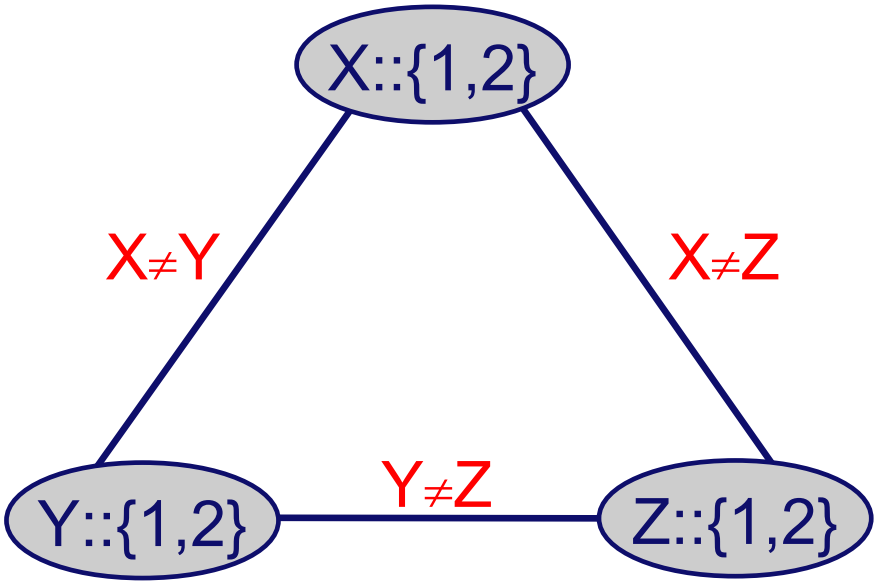
Q enthält solche Werte, die in der Initialisierung gelöscht wurden.

Bewertung Kantenkonsistenz-Algorithmen

- AC-3 und AC-4 am häufigsten verwendet
- Weitere Verbesserungen durch AC-5, AC-6, AC-7
- Höherer Verwaltungsaufwand kompensiert häufig die verminderte Redundanz in den Berechnungen
- Wahl des besten Algorithmus ist anwendungsabhängig

Übung

- Machen Sie das Netz kantenkonsistent !
- Wie lauten die Lösungen des Constraint-Netzes ?



Lösung

- Das Netz ist bereits kantenkonsistent
- Es hat aber keine Lösung
- Kantenkonsistenz garantiert also nicht, dass es eine Lösung gibt!

K-Konsistenz

- K-Konsistenz, wenn beliebiges konsistentes Wertetupel für k-1 paarweise verschiedene Variablen durch Wert einer beliebigen weiteren Variablen so erweitert werden kann, dass alle Constraints zwischen den k Variablen erfüllt sind.
- Anleitung:
  - Gegeben: Constraint-Netz
  - Man nimmt ein beliebiges konsistentes Wertetupel für k-1 (paarweise verschiedene) Variablen (beliebig heißt, es müssen alle konsistenten Wertetupel überprüft werden).
  - Man überprüft für alle übrigen Variablen, ob es einen Wert gibt, der mit dem betrachteten Wertetupel konsistent ist.

Zusatzinformation

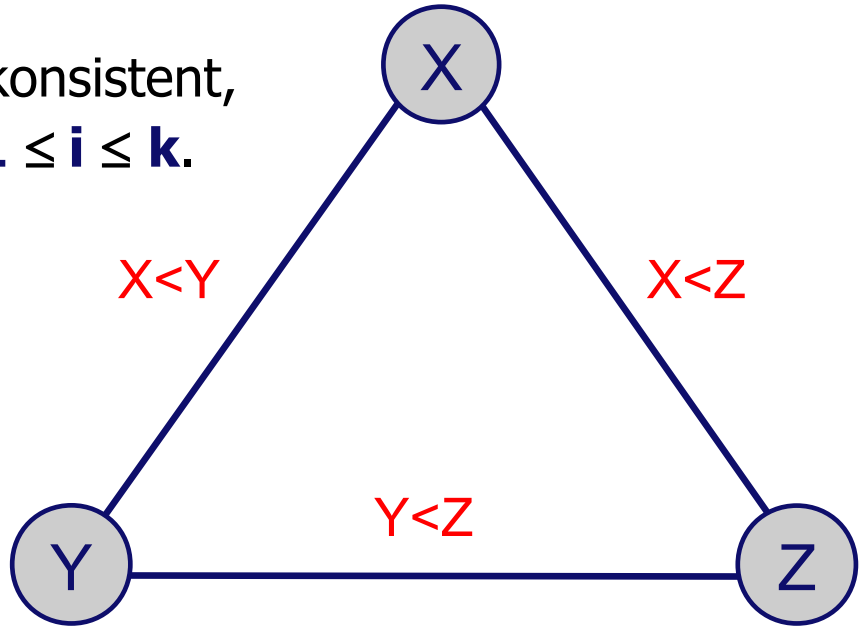
K-Konsistenz: Formale Definition

Gegeben sei ein Constraint-Netz auf den Variablen $x_1, \ldots, x_m$ mit den Wertebereichen $D_1, \ldots, D_m$. Sei $(d_{i1}, \ldots, d_{i(k-1)}) \in D_{i1} \times \ldots \times D_{i(k-1)}$ mit $i_j \in \{1, \ldots, m\}$ für $j = \{1, \ldots, k\}$ eine Zuweisung von Werten an $(k-1)$, paarweise verschiedene Variablen, die alle Constraints zwischen den Variablen erfüllt. Das Netz ist k-konsistent, wenn durch Hinzunahme einer beliebigen weiteren Variablen x_{ik}, $k \in \{1, \ldots, m\}$, die Wertezuweisung zu $(d_{i1}, \ldots, d_{ik}) \in D_{i1} \times \ldots \times D_{ik}$ so erweitert werden kann, dass alle Constraints zwischen diesen k Variablen erfüllt sind.

Zusatzinformation

Strenge k-Konsistenz, Beispiel

- Ein Constraint-Netz ist streng **k**-konsistent, wenn er **i**-konsistent ist für alle **1** ≤ **i** ≤ **k**.



x	y	z	Grad der lokalen Konsistenz
$\{1,2\}$	$\{2,3\}$	$\{3,4\}$	streng 2-konsistent, aber nicht 3-konsistent
$\{1,2\}$	$\{3,4\}$	$\{5,6\}$	streng 3-konsistent
$\{1,3\}$	$\{2\}$	$\{3\}$	3-konsistent, aber nicht 2-konsistent

Zusatzinformatio

Höhere Grade lokaler Konsistenz

- Exponentieller Aufwand (mit dem Grad der lokalen Konsistenz)
- Wenn Stelligkeit Anzahl der Variablen erreicht, liegt Lösung vor.
  - Kein Suchen mehr nötig

- Erhöhter Aufwand zur lokalen Konsistenz kann eingesparten Aufwand durch vermiedenes Ausprobieren (Suchen) überkompensieren.

Zusatzinformation

C4: Lernziele

- V1: Motivation für Binarisierung erläutern können
- V2: Die beiden Formen der Binarisierung erläutern können
- V3: Begriffe Knoten-, Kanten und lokale Konsistenz erläutern können
- V4: Erläutern können, wie Knoten- oder Kantenkonsistenz erreicht werden kann
- V5: Algorithmen REVISE, AC-1, AC-3 und AC-4 in ihrer Arbeitsweise mit Vor- und Nachteilen erläutern können, AC4 nicht im Detail
- V6: Für REVISE, AC-1 und AC-3 Detailfragen zu konkreten Aufgabenstellungen beantworten können, für AC-4 nur bzgl. der Zähler und Abhängigkeitsmengen
- A1: AC-1 und AC-3 auf konkrete Aufgabenstellungen anwenden können (-> Teil 2 der Klausur)

C5: Analytische Lösungsverfahren

- Konsistenzalgorithmen integriert mit systematischer Suche
  - Alle analytischen Lösungsverfahren bestehen aus den beiden Komponenten Kantenkonsistenz und systematische Suche
- Lösungsverfahren:
  - Suche mit (chronologischem) Backtracking
  - Forward Checking
  - (Full) Look Ahead

C5: Lernziele

- V1: Lösungsalgorithmen Backtracking, Forward Checking und Full Look Ahead im Detail erklären und vergleichen können. Pseudo-Code muss nicht im Detail beherrscht werden

Suche mit (chronologischem) Backtracking

- Variablen erhalten nach und nach testweise Werte
  - (Treffen von Annahmen)
- Nach jeder Wertezuweisung erfolgt Prüfung der Constraints
- Ggf. Zurückziehen der letzten Wertezuweisung
  - Statt dessen Ausprobieren des nächsten Werts
  - "Chronologisches Backtracking"

- Also: Generate-And-Test für Teillösungen

Algorithmus AC-3 for Backtracking

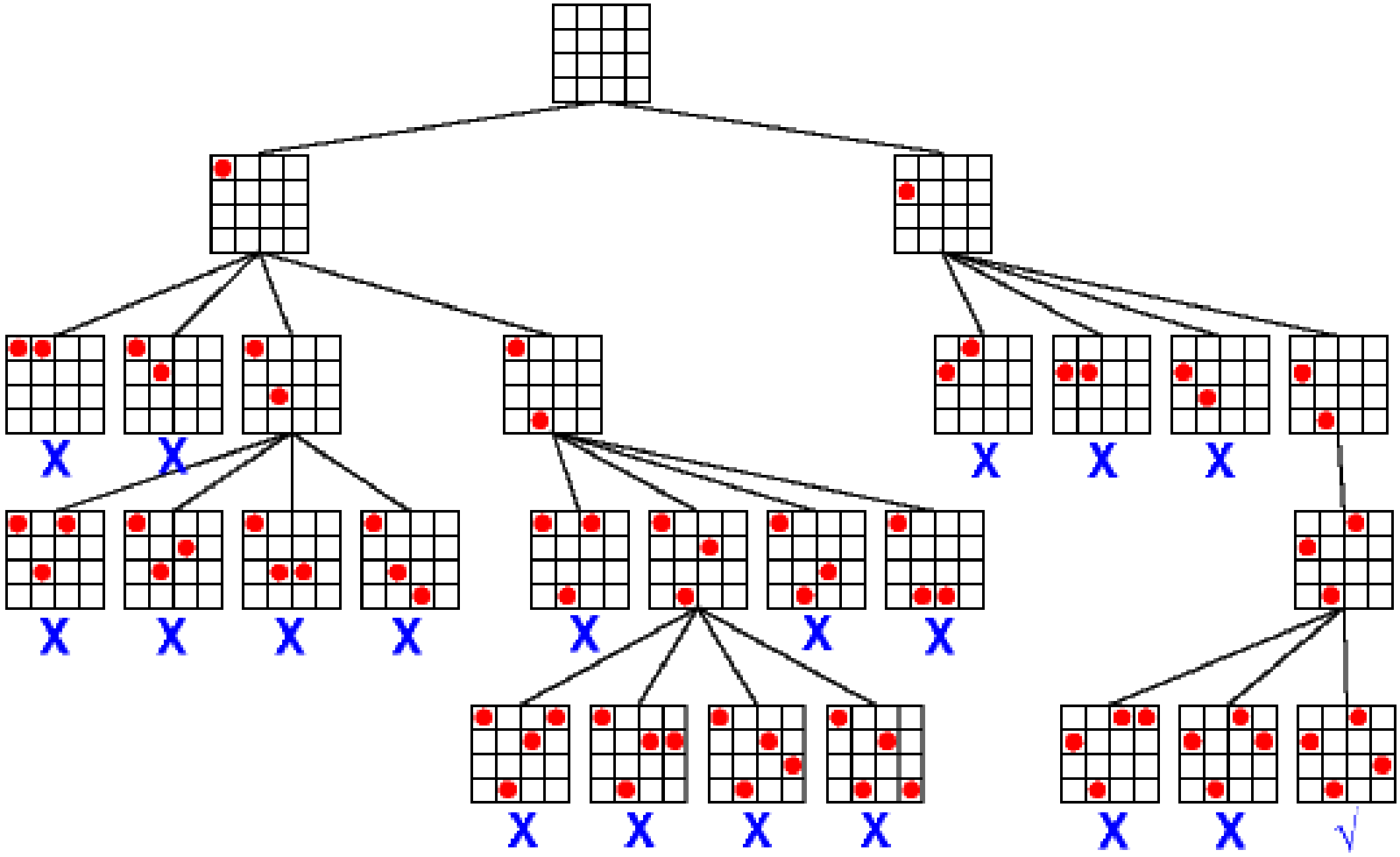
```
procedure AC3-BT(cv)
   Q <- {(Vi,Vcv) in arcs(G),i<cv};
   consistent <- true;
   while not Q empty & consistent
      select and delete any arc (Vk,Vm) from Q;
      consistent <- not REVISE(Vk,Vm)
   endwhile
   return consistent
end AC3-BT
```

Konsistenz-Prüfung für alle bereits instantiierten Variablen

Vk und Vm haben nur noch einen Wert. Wenn REVISE true liefert, wurde der Wert von Vk gelöscht, weil beide nicht konsistent sind.

(Aufruf von AC3-BT nach jeder Wertezuweisung)

Beispiel für Backtracking: 4-Damen-Problem



Erläuterungen

- Man beachte die kluge Wahl, was eine Variable ist und was ihr Wert repräsentiert. So werden die Constraints, dass nie zwei Damen untereinander stehen dürfen, automatisch erfüllt.
- Dem Pseudocode fehlt noch die übergeordnete Methode die testweise Werte setzt.

Diskutieren Sie!

- Wie bewerten Sie das Backtracking-Verfahren?

Bewertung Backtracking

- Wert für nächste Variable wird blind gesetzt
- Verbesserung: Nur solche nächsten Werte testen, die mit bereits gesetzten Werten konsistent sind

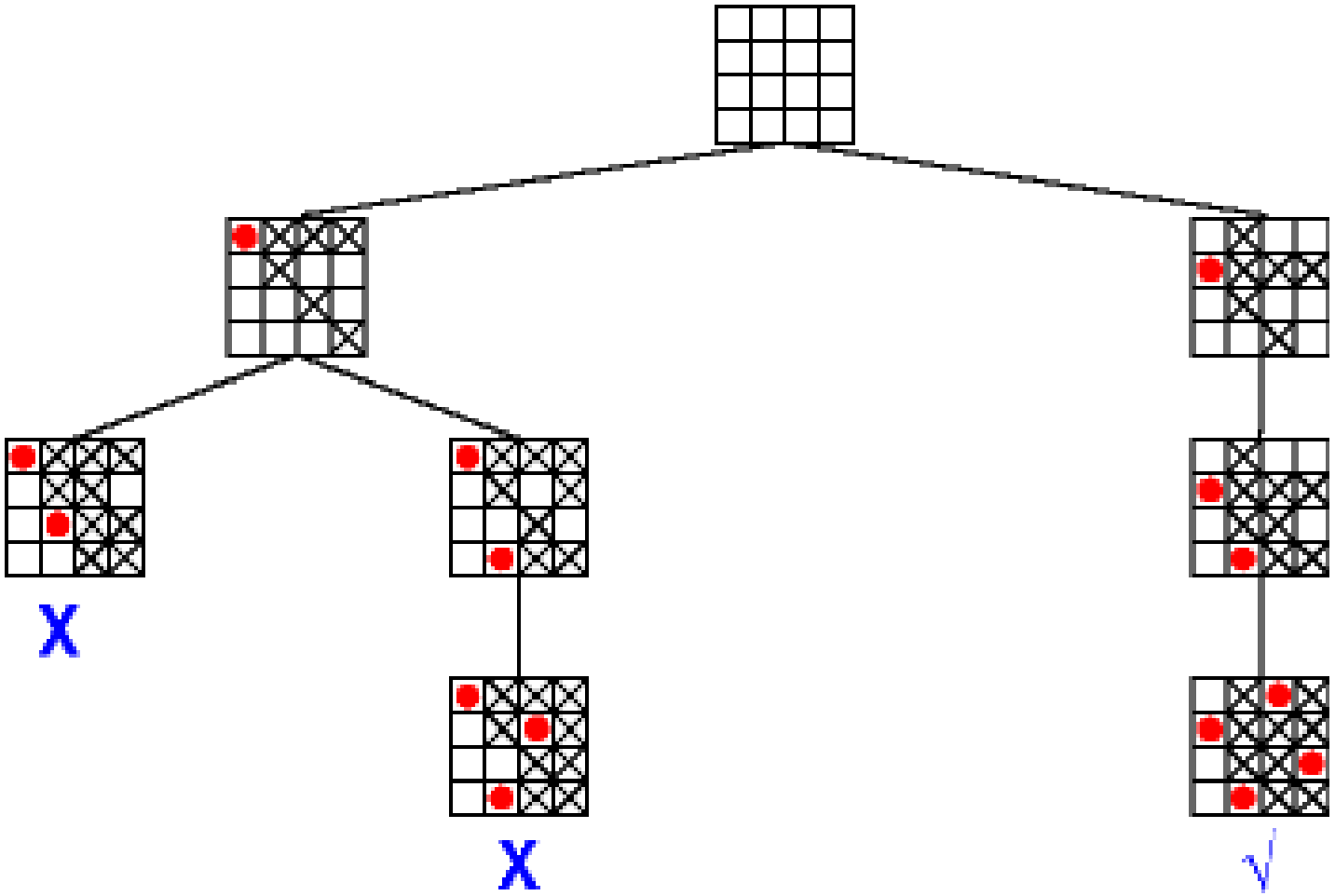
$\rightarrow$ "Forward Checking"

Algorithums AC-3 for Forward Checking

```
procedure AC3-FC(cv)
  Q <- {(Vi,Vcv) in arcs(G),i>cv};
  consistent <- true;
  while not Q empty & consistent
    select and delete any arc (Vk,Vm) from Q;
    if REVISE(Vk,Vm) then
      consistent <- not Dk empty
    endif
  endwhile
  return consistent
end AC3-FC
```

Konsistenz-Prüfung für alle freien Variablen (für bereits instantiierte Variablen nicht nötig)

Beispiel für Forward-Checking: 4-Damen-Problem



Diskutieren Sie!

- Wie bewerten Sie das Forward-Checking-Verfahren?

Bewertung Forward-Checking

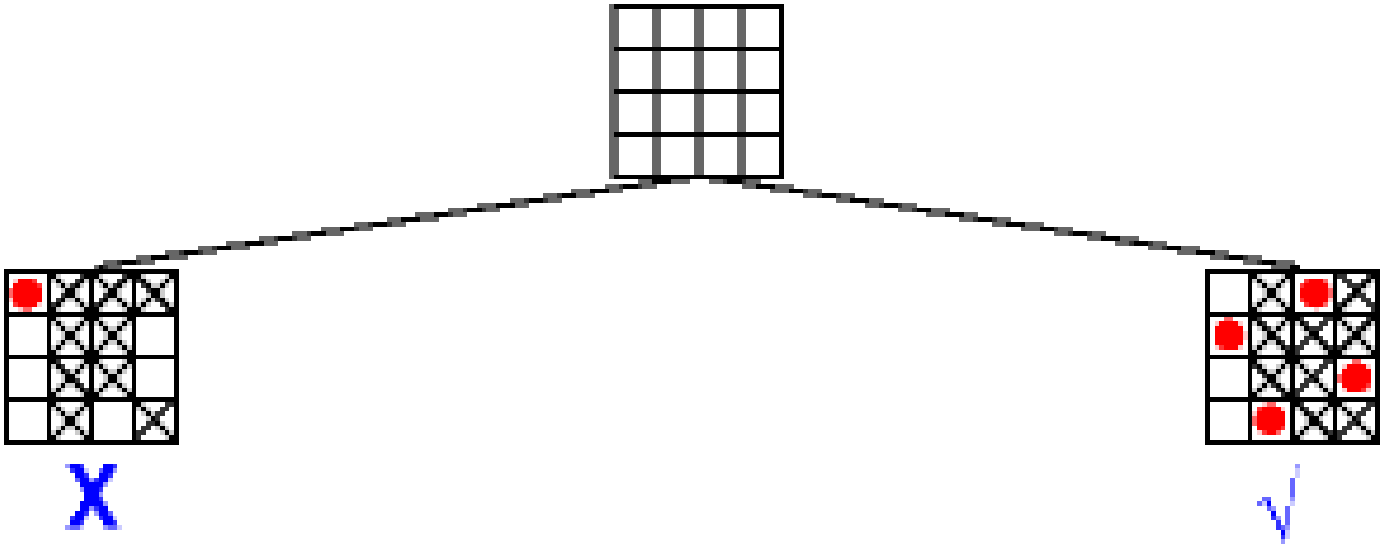
- Konsistenzprüfung nur zwischen aktueller Variablen und nachfolgenden Variablen
- Keine Prüfung zwischen nachfolgenden Variablen untereinander
- Verbesserung: Kantenkonsistenz zwischen allen nachfolgenden Variablen sicherstellen

$\rightarrow$ "(Full) Look Ahead"

Algorithums AC-3 for Look-Ahead

```
procedure AC3-LA(cv)
  Q <- {(Vi,Vcv) in arcs(G),i>cv};
  consistent <- true;
  while not Q empty & consistent
    select and delete any arc (Vk,Vm) from Q;
    if REVISE(Vk,Vm) then Q <- Q union {(Vi,Vk)
        such that (Vi,Vk) in arcs(G),i#k,i#m,i>cv}
      consistent <- not Dk empty
    endif
  endwhile
  return consistent
end AC3-LA
```

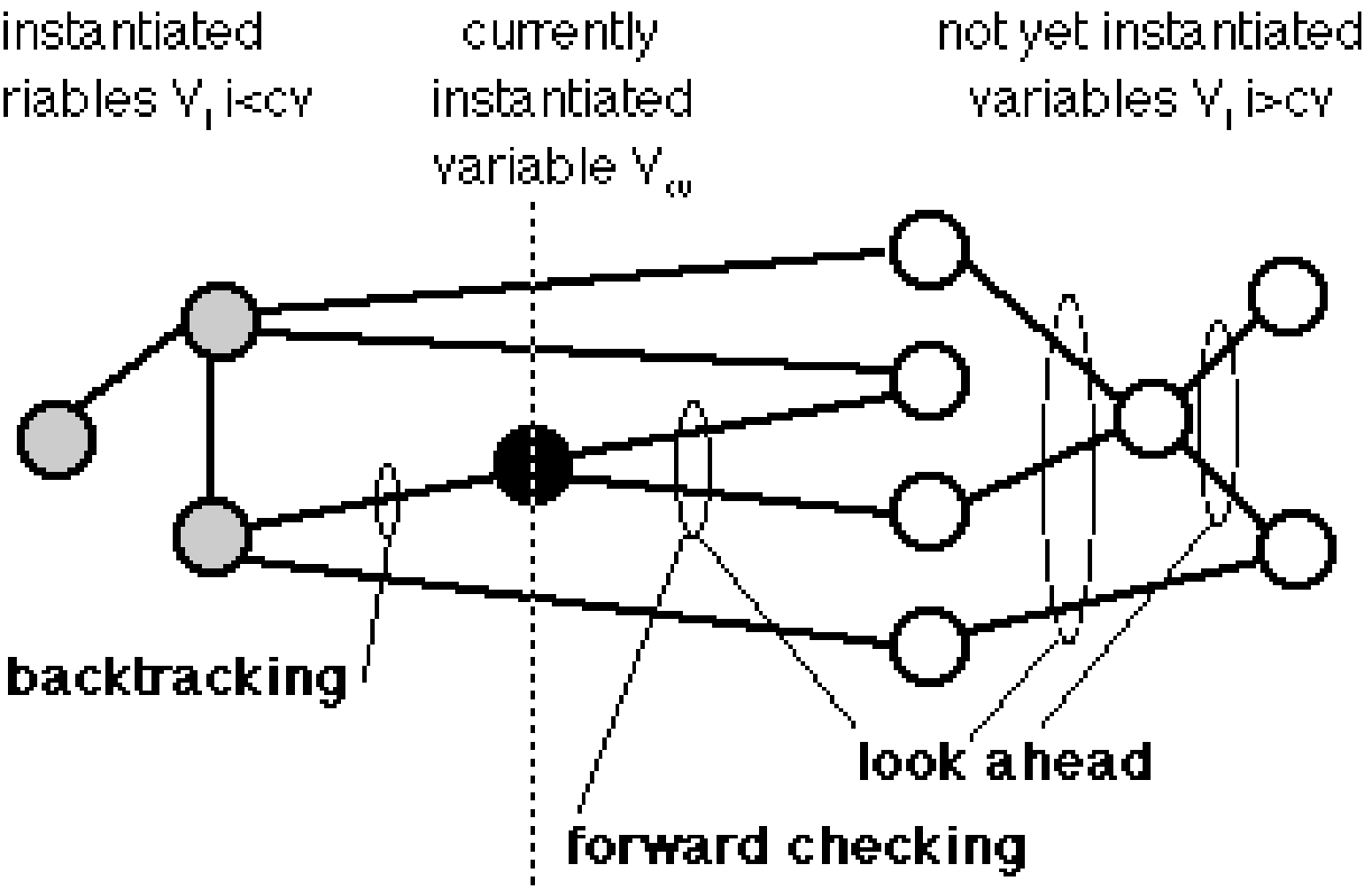
Beispiel für Look-Ahead: 4-Damen-Problem



Bewertung Look-Ahead

- Suche, also das Treffen von Annahmen, wird minimiert
- Im Prinzip liegt hier ein AC-3 im Wechselspiel mit dem Treffen von Annahmen vor
- Diese Arbeitsweise entspricht dem Waltz-Filtern !
- Da wenige Annahmen getroffen werden, ist die Gefahr von redundanten Arbeitsschritten aufgrund von Backtracking geringer. Chronologisches Backtracking ist sehr ineffizient, wenn die für die Inkonsistenz ursächliche Entscheidung sehr früh getroffen wird

Vergleich der Propagierungsverfahren



Intelligentes Backtracking

- Backmarking
  - Keine Re-Instantiierung mit Werten, die zu Konflikten geführt haben, wenn die andere am Konflikt beteiligte Variable ihren alten Wert noch aufweist
  - Keine Wiederholung von Konsistenztests, die erfolgreich waren
- Backjumping
  - Zurückziehen der Instantiierung, die einen Konflikt verursacht
- Truth Maintenance Systeme
  - Verwalten die Abhängigkeiten in Inferenzsystemen
  - Assumption Based Truth Maintenance System (ATMS, de Kleer) hilft, zwischen Annahmen-Mengen (Kontexten) umzuschalten

Zusatzinformation

C5: Lernziele

- V1: Lösungsalgorithmen Backtracking, Forward Checking und Full Look Ahead im Detail erklären und vergleichen können. Pseudo-Code muss nicht im Detail beherrscht werden

C6: Ordnungsheuristiken

- In welcher Reihenfolge sollten die Variablen (testweise) instantiiert werden?
- Welcher Wert sollte als erster probiert werden?

C6: Lernziele

- V1: Bedeutung der Reihenfolge der Variablen-Instantiierung und Reihenfolge der Werte-Auswahl erläutern können
- V2: Beispielhaft Heuristiken zur Reihenfolge der Variablen-Instantiierung und Werte-Auswahl erklären können
  - Namen der Heuristiken müssen nicht beherrscht werden
- V3: Begriff der Weite eines Constraint-Netzes und den Zusammenhang zum Backtracking erklären können (bzgl. Baumgraphen)

Reihenfolge von Variablen-Instantiierungen

- Reihenfolge, in der Variablen instantiiert werden, hat Einfluss auf Rechenaufwand
- Vorab-Berechnung ("static ordering")
- Berechnung während des Verfahrens ("dynamic ordering")
  - Nur möglich, wenn während des Verfahrens neue Informationen gewonnen werden
    - nicht möglich für Backtracking
    - möglich für Forward Checking (eingeschränkte Wertebereiche)

Diskutieren Sie!

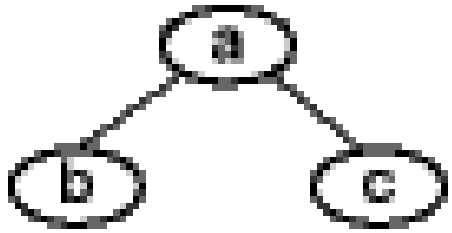
- Welche Variable würden Sie zuerst instantiieren? (Es gibt unterschiedliche Merkmale, die herangezogen werden können.)
- Begründen Sie Ihre Entscheidung!

Statische Ordnungsheuristiken für Variablen

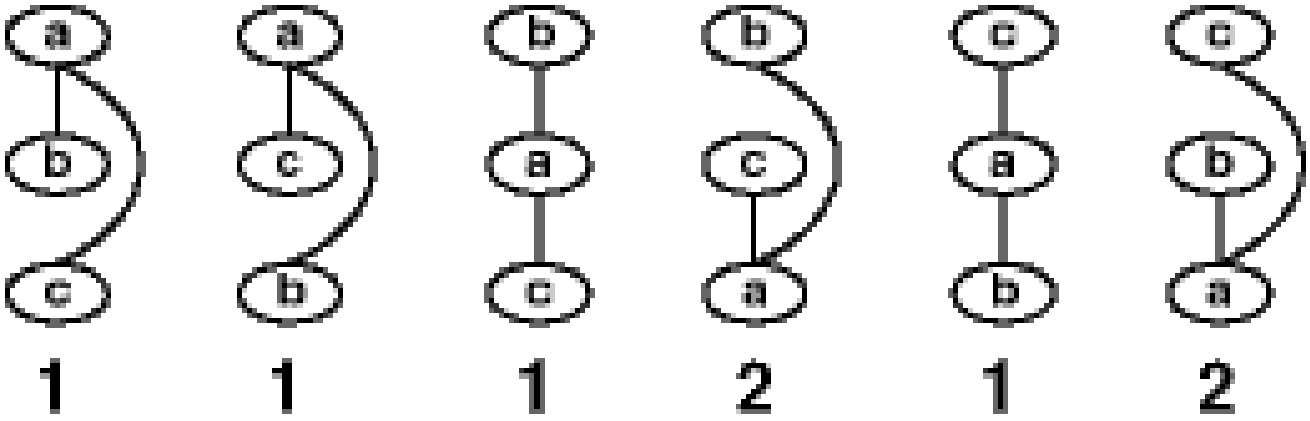
- First-Fail-Heuristik / Minimum-Remaining-Values-Heuristik
  - Variable mit höchster Wahrscheinlichkeit des Scheiterns zuerst wählen
  - Geringste Anzahl möglicher Werte
- Maximum-Degree-Heuristik
  - Ähnliche Überlegung: Variable, die an den meisten Constraints beteiligt ist
- Maximum-Cardinality-Heuristik
  - Variable mit den meisten Constraints mit bereits instantiierten Variablen
  - Geeignet für Backtracking (Bsp.: Einfärbe-Problem)
- Minimal-Width-Heuristik
  - Reihenfolge der Variablen unterstützt Weite des Constraint-Netzes

Weite eines Constraint-Graphen

- Weite bezieht sich auf eine bestimmte Sortierung der Variablen



Beispiel-Graph



Unterschiedliche Sortierungen

Weite

Erläuterungen

- Für drei Variablen gibt es 3! = 6 Sortierungen. Die Variablen sind entsprechend der Sortierung von oben nach unten aufgeführt.
- Die Weite für eine bestimmte Sortierung der Variablen ist das Maximum der Anzahl von Kanten zu Vorgängern. Man betrachtet also jede Variable und zählt die Kanten zu Vorgängern.
- Die Weite eines Constraint-Netzes ist die Weite der günstigsten Sortierung, also das Minimum der Weiten der verschiedenen Sortierungen.
- Es sollte also die Sortierung mit der geringsten Weite gewählt werden. Eine große Weite bedeutet ja, wie man am Beispiel sehen kann, dass zuerst Variablen testweise Werte bekommen, die keine direkte Verbindung haben. So wird eine Inkonsistenz erst sehr spät deutlich, nämlich dann, wenn für eine oder mehrere Variable(n), die die Verbindung zwischen den bereits gesetzten Variablen herstellt/herstellen, noch ein passender Wert gefunden werden muss.

Dynamische Ordnungsheuristiken für Variablen

- Dynamic-Search-Rearrangement-Heuristik
  - Variable mit den wenigsten mit der bisherigen Teillösung konsistenten Werten
    - Mit anderen Worten: Es wird immer die Variable als nächste genommen, die über die kleinste Anzahl möglicher Werte verfügt. So wird der Lösungsraum maximal beschnitten.

Diskutieren Sie!

- Welchen Wert würden Sie zuerst setzen?
- Begründen Sie Ihre Entscheidung!

Reihenfolge der Werte

- Wenn keine Lösung existiert oder alle Lösungen gefunden werden müssen, ist Reihenfolge egal
- Succeed-First-Heuristik
  - Wert wählen, der die geringsten zukünftigen Einschränkungen mit sich bringt
    - Wieviele Werte anderer Variablen werden unterstützt?
  - Sinnvoll, weil Scheitern lediglich bedeutet, dass weitere Alternativen geprüft werden müssen
- Auch hier gilt: Prüfen, ob Aufwand lohnt!

Vermeidung von Backtracking

- Ziel der Festlegung der Reihenfolge von Variablen:
  - Vermeidung von Backtracking
- Von Interesse: Existiert eine Variablen-Ordnung, die Backtracking komplett vermeidet?
  - Ein Constraint-Graph sei streng k-konsistent
  - w sei die Weite des Constraint-Graphen
  - Wenn k>w gilt, dann existiert eine Variablen-Ordnung, die kein Backtracking benötigt.
    - Klar, weil maximale Anzahl von Abhängigkeiten kleiner als Konsistenz-Fokus ist
  - Also reicht z. B. Kantenkonsistenz für Baum-Graphen

C6: Lernziele

- V1: Bedeutung der Reihenfolge der Variablen-Instantiierung und Reihenfolge der Werte-Auswahl erläutern können
- V2: Beispielhaft Heuristiken zur Reihenfolge der Variablen-Instantiierung und Werte-Auswahl erklären können
  - Namen der Heuristiken müssen nicht beherrscht werden
- V3: Begriff der Weite eines Constraint-Netzes und den Zusammenhang zum Backtracking erklären können (bzgl. Baumgraphen)

C7: Was gibt es sonst noch?

Stochastische Verfahren

- Start mit zufälliger Instantiierung aller Variablen ("Konfiguration")
- Sukzessive Verbesserung durch Austausch von Werten, die Konflikte verursacht haben
  - → **Suche**
  - bis Lösung gefunden
  - Ggf. Neustart mit anderer Konfiguration, um lokale Minima zu vermeiden
- Häufig sehr schnell
- Nicht vollständig
- Nicht geeignet, wenn alle Lösungen gefunden werden müssen

Overconstrained CSP

- Keine Lösung vorhanden
- Lösungsmöglichkeiten:
  - Grad der Gültigkeit eines Constraints modellieren
  - Partielle Constraint-Erfüllung
  - Harte und weiche Constraints unterscheiden

Constraint Logic Programming in SWI-Prolog

```
send([[S,E,N,D], [M,O,R,E], [M,O,N,E,Y]])  :-
              Digits   =  [S,E,N,D,M,O,R,Y],
              Carries  =  [C1,C2,C3,C4],
              Digits  in  0..9,
              Carries in  0..1,

              M                    #=                C4,
              O  +  10  *  C4  #=  M  +  S  +  C3,
              N  +  10  *  C3  #=  O  +  E  +  C2,
              E  +  10  *  C2  #=  R  +  N  +  C1,
              Y  +  10  *  C1  #=  E  +  D,

              M  #>=  1,
              S  #>=  1,
              all_different(Digits),
              label(Digits).
```