

Entwurf von Algorithmen: Backtracking bzw. das Backtrack-Prinzip

Dozent: Prof. Dr. Michael Eichberg

Kontakt: michael.eichberg@dhw.de, Raum 149B

Version: 2.0.3

Folien: https://delors.github.io/theo-algo-back_tracking/folien.de.rst.html

https://delors.github.io/theo-algo-back_tracking/folien.de.rst.html.pdf

Kontrollfragen: https://delors.github.io/theo-algo-back_tracking/kontrollfragen.de.rst.html

Fehler melden: <https://github.com/Delors/delors.github.io/issues>

Backtracking

Neben der dynamischen Programmierung ist das Backtrack-Prinzip ein weiteres grundlegendes Verfahren zur Lösung von Problemen.

- Backtracking ist ein Verfahren, das in vielen Algorithmen zur Anwendung kommt. Insbesondere, wenn kein effizienterer Algorithmus bekannt ist, als *alle möglichen Lösungen auszuprobieren*.
 - Backtracking ist eine systematische Methode, um alle möglichen Lösungen eines Problems zu finden. Es ist eine Art von rekursivem Durchsuchen, bei dem Teillösungen zu Gesamtlösungen erweitert werden.
 - Backtracking erlaubt ggf. Heuristiken, um die Suche zu beschleunigen.
Weder die Komplexitätsklasse noch die Korrektheit ändert sich dadurch.
 - Viele NP-harte Probleme werden mit Backtracking gelöst.[\[1\]](#)
-

Backtracking führt eine erschöpfende Suche durch, um eine Lösung zu finden. Kann aber auch direkt genutzt werden, um ggf. alle Lösungen zu finden.

Backtracking ist in Prolog inherent vorhanden, da Prolog auf dem Prinzip des Backtrackings basiert, weswegen Prolog für die Lösung solcher Probleme gut geeignet ist.

-
- [1] NP-harte und NP-vollständige Probleme sind solche Probleme, die schwierig sind zu lösen. Hier reicht es zu wissen, dass es Probleme gibt, die nicht in polynomieller („vernünftiger“) Zeit gelöst werden können.

Beispiel: Das 4-Damen Problem (konzeptuell)

```
1 // i: Spalte; j: Zeile
2 procedure findeStellung(i : integer)
3   j := 0
4   repeat
5     { wähle nächste Zeile j }
6     if Dame an Position i / j bedroht
7       keine bisher platzierte Dame then
8       { platziere Dame in Feld i / j }
9       if i = 4 then
10         { Lösung gefunden }
11         { Ausgabe der Lösung }
12       else
13         findeStellung(i + 1) // rek. Aufruf
14       { entferne Dame aus Spalte i und Zeile j } // zurücksetzen des Zustands
15   until { alle Zeilen j getestet }
```

Ziel ist es vier Damen auf einem Schachbrett so zu platzieren, dass keine Dame eine andere Dame schlagen kann.[\[2\]](#) Eine Lösung:

	1	2	3	4
1			D	
2	D			
3				D
4		D		

Wesentliche Elemente:

- Die Lösung ist endlich.
- Die Lösung wird iterativ aufgebaut. Es ist jederzeit möglich zu testen, ob die bisherige Lösung noch gültig ist (Zeile 6, 7).
- Ist eine Lösung nicht mehr möglich, wird die Teillösung auch nicht weiter verfolgt.
- Wurde eine Lösung gefunden, wird sie ausgegeben (Zeile 10, 11).
- Die Methode wird rekursiv aufgerufen, um die Lösung zu vervollständigen (Zeile 13).

-
- [2] Es gibt eine geschlossene Lösung für das Problem. Backtracking wird hier nur als Beispiel für das Verfahren verwendet.

Backtracking - Allgemein

Voraussetzungen für Backtracking

1. Die Lösung ist als Vektor $a[1], a[2], \dots$ endlicher Länge darstellbar.
2. Jedes Element $a[i]$ hat eine endliche Anzahl von möglichen Werten $A[i]$.
D. h. die Menge der möglichen Werte pro $a[i]$ kann unterschiedlich sein.
3. Es gibt einen effizienten Test, ob eine Teillösung $a[1], a[2], \dots, a[k]$ zu einer gültigen Lösung führen kann.

Verfahren

Start: Wähle eine Teillösung $a[1]$.

Allgemein: Ist eine Teillösung basierend auf $a[1], a[2], \dots, a[k-1]$ noch keine Gesamtlösung, dann erweitere sie mit dem nächsten nicht ausgeschlossenen Wert $a[k]$ aus $A[k]$ zur neuen Teillösung $a[1], a[2], \dots, a[k]$.

Falls noch nicht alle Elemente von $A[K]$, die zu keiner inkonsistenten Lösungen führen, ausgeschöpft sind, dann gehe zurück (backtrack) und wähle $a[k]$ neu. Ggf. gehe zu $a[k-1]$ usw. zurück.

Es wird hier nicht gefordert, dass alle Element den gleichen Wertebereich haben.

Es ist auch möglich, dass die Werte unterschiedlich sind.

Übung

0.1. Auswerten logischer Ausdrücke mittels Backtracking

Bestimmen Sie für folgenden Ausdruck C - mittels Backtracking - Wahrheitswerte für die Variablen, damit der Ausdruck als Ganzes wahr wird:

$$C = (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg C) \wedge (C \vee D) \wedge (\neg C \vee \neg D)$$

Füllen Sie dazu die folgende Tabelle aus, um alle Lösungen zu finden. In der letzten Spalte geben Sie an, ob die Zeile eine Teillösung darstellt (nicht inkonsistent), keine Lösung ist bzw. sein kann, oder eine Gesamtlösung identifiziert wurde. Die Evaluation wie vieler vollständiger Belegungen wurde eingespart, wenn die Lösung gefunden wurde?

	A	B	C	D	nicht inkonsistent (T), keine Lösung (K), vollständige Lösung (L)
1	w				T
...
16

Übung

0.2. Das Erfüllbarkeitsproblem

Konjunktive Normalform (KNF)

Ein logischer Ausdruck ist in KNF, wenn der Ausdruck nur als Konjunktion (UND-Verknüpfung) von Disjunktionen (ODER-Verknüpfungen) dargestellt wird. Die Negation darf nur auf Variablen angewendet werden.

Beispiel: $(A \vee B \vee C) \wedge (\neg C \vee D)$

Entwickeln Sie ein Programm — in einer Programmiersprache Ihrer Wahl — das in der Lage ist eine Formel in konjunktiver Normalform (KNF) auf Erfüllbarkeit zu prüfen.

Prüfen Sie Ihr Programm anhand der vorhergehenden Aufgabe.

※ Hinweis

Sollten Sie das Programm in Python oder Java implementieren wollen, dann können sie den entsprechenden Code im Anhang als Grundlage verwenden. Sie müssen dann nur noch die Methode `solve` implementieren. Der Code implementiert eine kleine Klassenhierarchie zur Repräsentation von logischen Ausdrücken und ermöglicht die Evaluation (`is_solution` bzw. `isSolution`) unter einer gegebenen Belegung.

• Python Template

```
1 from abc import abstractmethod
2
3 class Expr:
4
5     @abstractmethod
6     def is_solution(
7         self,
8         binding: dict["Var", bool]) → bool | None:
9         """True or False if this expression definitively
10            evaluates to the respective truth value with the
11            given binding or None otherwise.
12
13            Returning a truth value does not necessarily
14            require all variables to be bound to a definite
15            value.
16
17            For example, None will be returned, if the
18            truth value cannot be determined with the given
19            binding. E. g., if this expression represents a
20            variable for which the binding has no value, None
```

```

21     is returned.
22
23     An expression such as "A ∧ B" would return True
24     if A and B are both True in the
25     binding and False if at least one of them is bound
26     to False, and None otherwise.
27     """
28     raise NotImplementedError
29
30
31 class And(Expr):
32     def __init__(self, *exprs: Expr):
33         self.exprs = exprs
34
35     def is_solution(self, binding):
36         r = True
37         for expr in self.exprs:
38             e = expr.is_solution(binding)
39             if e is None:
40                 r = None
41             elif not e:
42                 return False
43         return r
44
45     def __str__(self):
46         return " ∧ ".join(str(expr) for expr in self.exprs)
47
48     def __repr__(self):
49         exprs = ", ".join(str(expr) for expr in self.exprs)
50         return "And(" + exprs + ")"
51
52
53 class Or(Expr):
54     def __init__(self, *exprs: Expr):
55         self.exprs = exprs
56
57     def is_solution(self, binding):
58         r = False
59         for expr in self.exprs:
60             e = expr.is_solution(binding)
61             if e is None:
62                 r = None
63             elif e:
64                 return True
65         return r
66
67     def __str__(self):
68         return " ∨ ".join(str(expr) for expr in self.exprs)
69
70     def __repr__(self):
71         exprs = ", ".join(repr(expr) for expr in self.exprs)
72         return "Or(" + exprs + ")"
73
74

```

```

75 class Not(Expr):
76     def __init__(self, expr: Expr):
77         self.expr = expr
78
79     def is_solution(self, binding):
80         e = self.expr.is_solution(binding)
81         if e is None:
82             return None
83         else:
84             return not e
85
86     def __str__(self):
87         return f"¬{self.expr}"
88
89     def __repr__(self):
90         return "Not(" + repr(self.expr) + ")"
91
92
93 class Var(Expr):
94     def __init__(self, name: str):
95         self.name = name
96
97     def is_solution(self, binding):
98         """True or False if bound.
99         None if unbound (default).
100        """
101        if self not in binding:
102            return None
103        else:
104            return binding[self]
105
106    def __str__(self):
107        return self.name
108
109    def __repr__(self):
110        return 'Var("' + self.name + ')'
111
112
113 A = Var("a")
114 B = Var("b")
115 C = Var("c")
116 D = Var("d")
117 vars = [A, B, C, D]
118 """ The variables are now indexed to enable iterating over
119 them in the solve function. """
120
121 expr = And(
122     Or(A, B),
123     Or(Not(A), B),
124     Or(Not(A), Not(C)),
125     Or(C, D),
126     Or(Not(C), Not(D)),
127 )
128 print("Finding solutions for: " + expr.__str__())

```

```

129
130     solution: dict[Var, bool] = {}
131     """ Stores the current solution by mapping the name of a
132         variable to its current truth value (True or False)."""
133
134
135     def solve(expr, vars):

```

Java Template

```

1 // Intended to be run using Java > 23 (Tested with Java 23 and 24)
2 // May require --enable-preview to do so.
3
4 interface Expr {
5
6     /**
7      * An Optional True or Optional False if this expression
8      * definitively evaluates to the respective truth value
9      * with the given binding or an empty Optional otherwise.
10     *
11     * Returning a truth value does not necessarily
12     * require all variables to be bound to a definite
13     * value. For example, Optional.empty will be returned,
14     * if the truth value cannot be determined with the given
15     * binding. E. g., if this expression represents a
16     * variable for which the binding has no value,
17     * Optional.empty is returned.
18     *
19     * An expression such as "A ∧ B" would return true
20     * if A and B are both bound to "true" and false
21     * if at least one of them is bound
22     * to "false", and Optional.empty otherwise.
23     */
24     Optional<Boolean> isSolution(Map<Var, Boolean> binding);
25 }
26
27 class And implements Expr {
28
29     private final Expr[] exprs;
30
31     And(Expr... exprs) {
32         this.exprs = exprs;
33     }
34
35     public Optional<Boolean> isSolution(Map<Var, Boolean> binding) {
36         Optional<Boolean> r = Optional.of(true);
37         for (var expr : this.exprs) {
38             final var e = expr.isSolution(binding);
39             if (!e.isPresent())
40                 r = Optional.empty();
41             else if (!e.get())
42                 return e;
43         }
44         return r;

```

```
45 }
46
47     public String toString() {
48         return Arrays.stream(exprs)
49             .map(Expr::toString)
50             .collect(Collectors.joining(" ^ "));
51     }
52 }
53
54 class Or implements Expr {
55
56     private final Expr[] exprs;
57
58     Or(Expr... exprs) {
59         this.exprs = exprs;
60     }
61
62     public Optional<Boolean> isSolution(Map<Var, Boolean> binding) {
63         var r = Optional.of(false);
64         for (var expr : this.exprs) {
65             final var e = expr.isSolution(binding);
66             if (!e.isPresent())
67                 r = Optional.empty();
68             else if (e.get())
69                 return e;
70         }
71         return r;
72     }
73
74     public String toString() {
75         return Arrays.stream(exprs)
76             .map(Expr::toString)
77             .collect(Collectors.joining(" v "));
78     }
79 }
80
81 class Not implements Expr {
82
83     private final Expr expr;
84
85     Not(Expr expr) {
86         this.expr = expr;
87     }
88
89     public Optional<Boolean> isSolution(Map<Var, Boolean> binding) {
90         final var r = expr.isSolution(binding).map(b → !b);
91         return r;
92     }
93
94     public String toString() {
95         return "¬" + expr;
96     }
97 }
98 }
```

```

99 class Var implements Expr {
100
101     private final String name;
102
103     Var(String name) {
104         this.name = name;
105     }
106
107     public Optional<Boolean> isSolution(Map<Var, Boolean> binding) {
108         final var r = Optional.ofNullable(binding.get(this));
109         return r;
110     }
111
112     public String toString() {
113         return name;
114     }
115
116 }
117
118 void main() {
119     final Var A = new Var("a");
120     final Var B = new Var("b");
121     final Var C = new Var("c");
122     final Var D = new Var("d");
123     Stack<Var> vars = new Stack();
124     vars.addAll(Arrays.asList(new Var[] { A, B, C, D }));
125     Expr expr = new And(
126         new Or(A, B),
127         new Or(new Not(A), B),
128         new Or(new Not(A), new Not(C)),
129         new Or(C, D),
130         new Or(new Not(C), new Not(D)));
131     IO.println("Finding solutions for: " + expr.toString());
132
133     Map<Var, Boolean> solution = new HashMap();
134     solve(expr, vars, solution);
135 }
136
137 void solve(Expr expr, Stack<Var> vars, Map<Var, Boolean> solution) {

```

Übung

0.3. Gruppenzuteilung

Finden Sie eine sehr gute Aufteilung von Personen (Studierenden) auf eine feste Anzahl an Gruppen, basierend auf den Präferenzen der Personen zueinander. Nutzen Sie dazu Backtracking.

Im Template ist eine initiale Aufgabenstellung hinterlegt, die es zu lösen gilt: Verteilung von 16 Studierenden auf 4 Gruppen inkl. Bewertungsmatrix (jeder Studierende hat jeden anderen mit Werten von 1 bis 10 bewertet):

• https://delors.github.io/theo-algo-back_tracking/code/group_assignment_template.py

