Programmierung von Strategiespielen

Ivo Blöchliger & Ulrich Ultes-Nitsche

Departement für Informatik
Universität Freiburg

13. September 2013

Zeitplan

Morgen

- Theorie
- Kaffeepause
- Theorie

Nachmittag

Praktische Übungen
Vertiefung des Stoffes

Dank

SVIA

Schweizerischer Verein für Informatik in der Ausbildung

http://www.svia-ssie-ssii.ch/

Departement für Informatik der Universität Freiburg

http://diuf.unifr.ch/

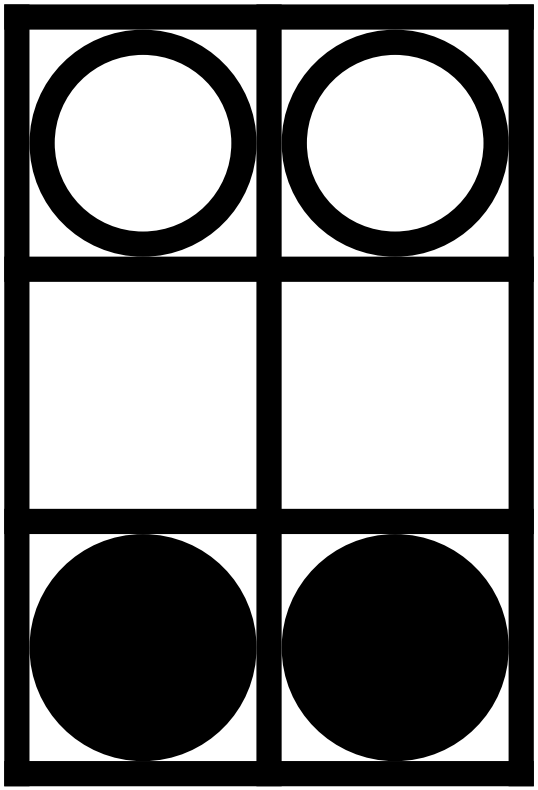
Inhalt

Miniaturschach

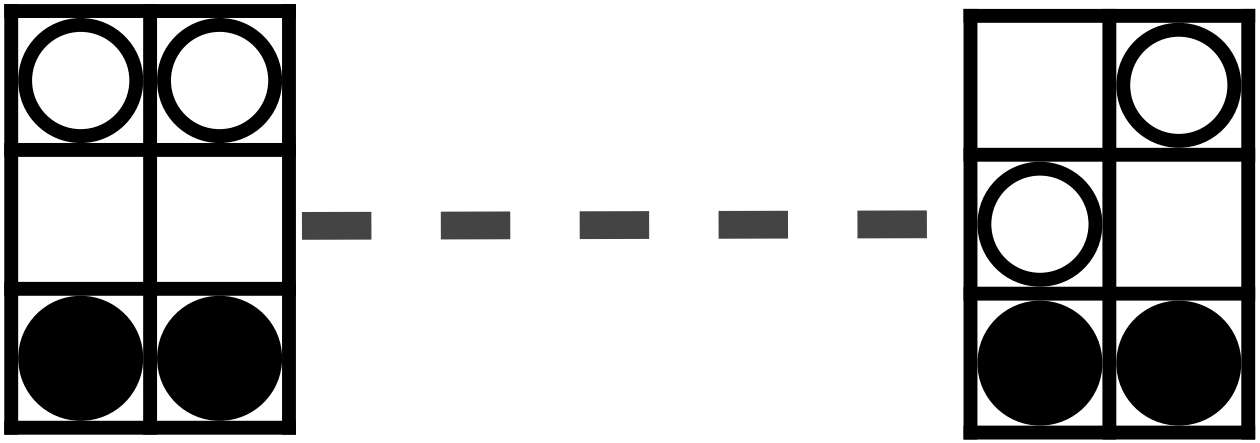
Regeln

- Brett reduzierter Grösse
- Ausschliesslich Bauern
- Ein Feld geradeaus ziehen oder diagonal schlagen
- Gewinn, wenn man die andere Seite erreicht
- Verlust, wenn man nicht mehr ziehen kann

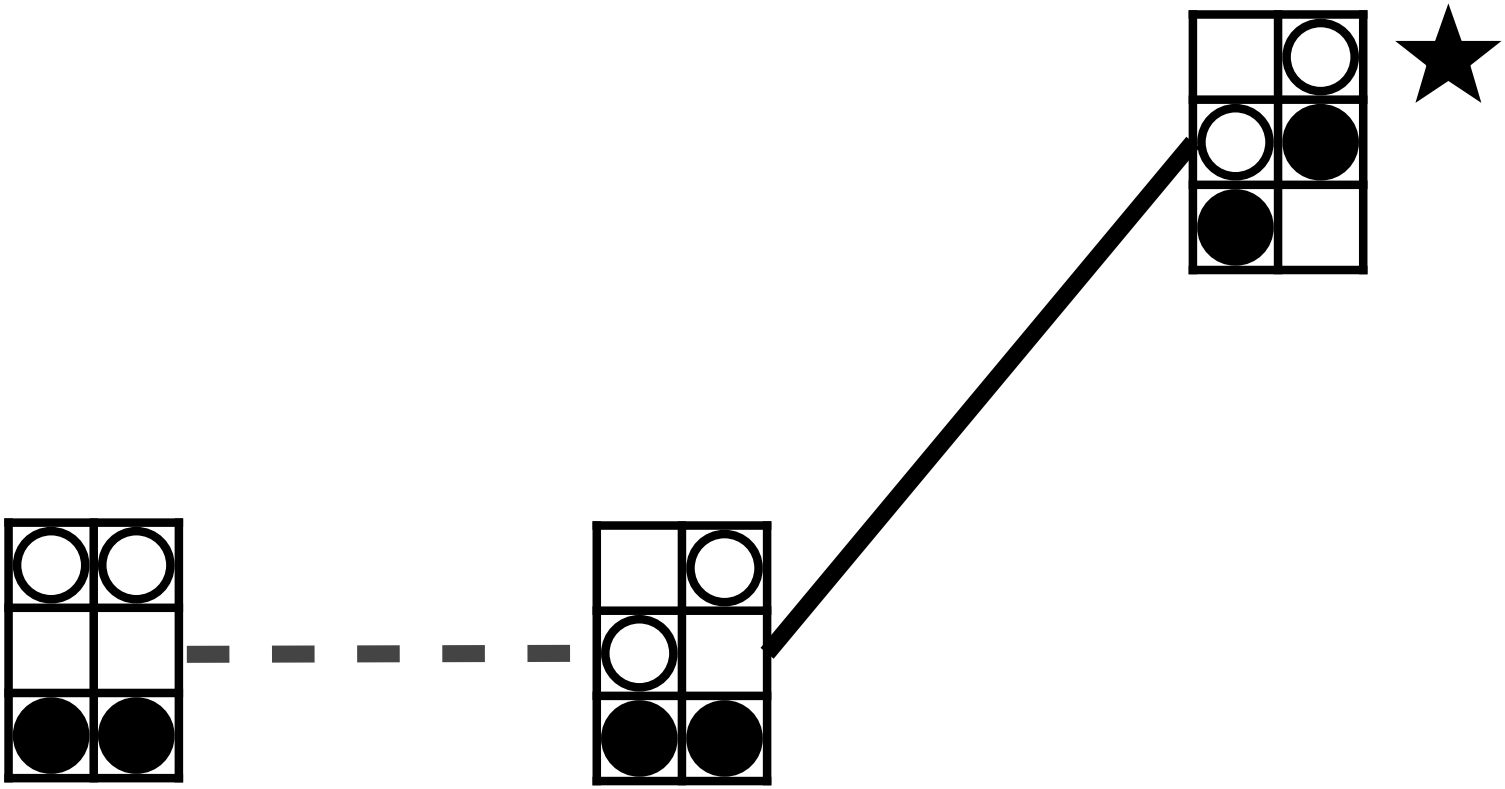
Miniaturschach



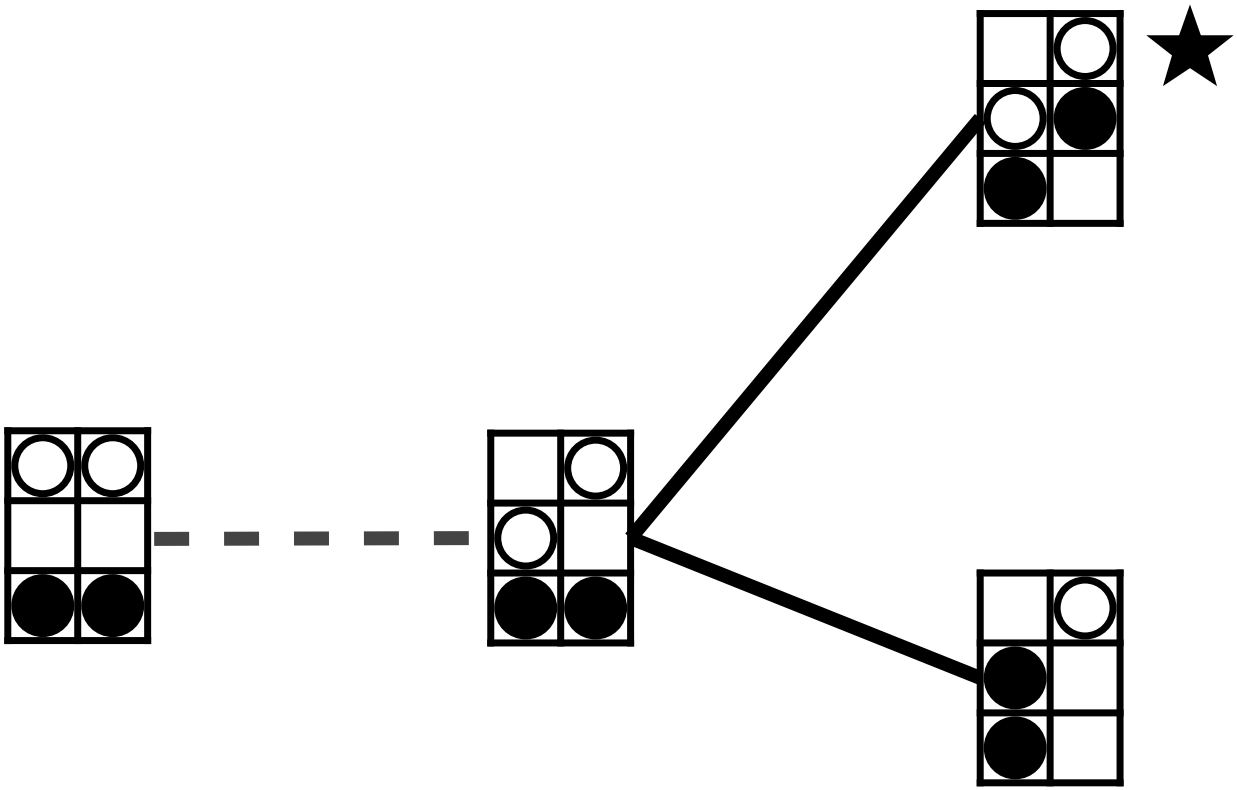
Miniaturschach



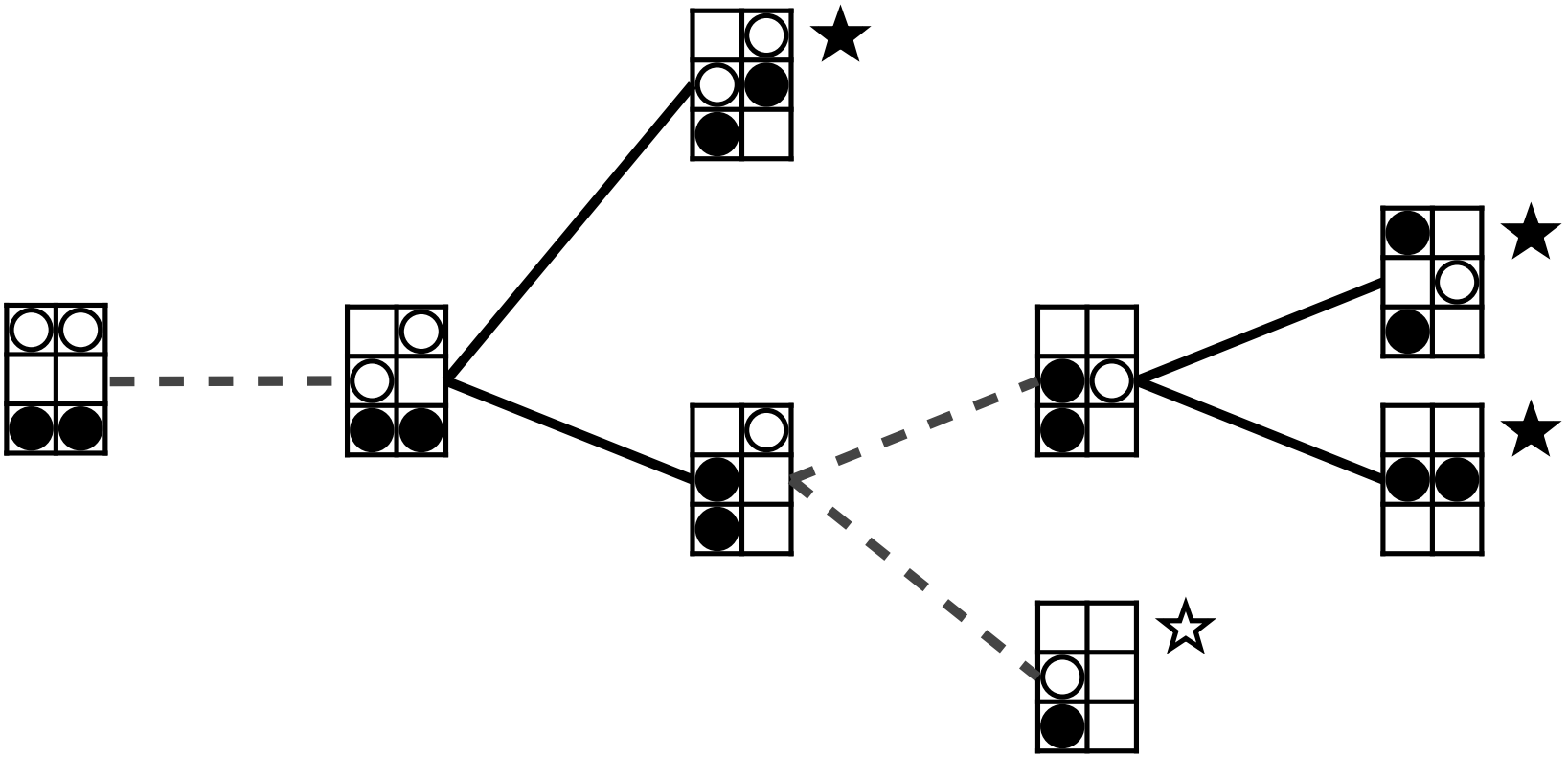
Miniaturschach



Miniaturschach



Miniaturschach



Allgemeine Situation

Situation

- Zwei Gegner
- Kein Zufall
- Abwechselnde Züge
- Bei jeder Runde : endlich viele Züge möglich
- Vollständige Information ist verfügbar (Spielzustand)
- Das Spiel endet mit einem Sieg oder Unentschieden

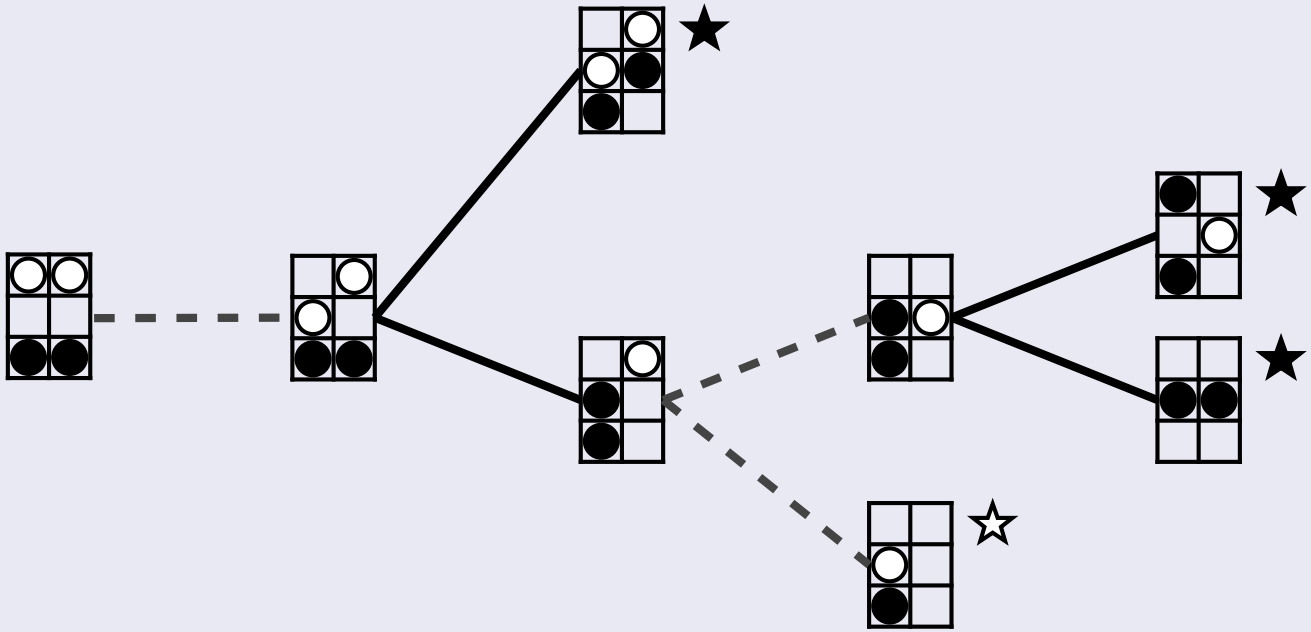
Beispiele

Ein paar Spiele gemäss dieser Definition

- Schach
- Vier gewinnt
- Dame
- Mühle
- Go
- Quoridor

Entscheidungsbaum

- Wurzel : Anfangszustand des Spiels
- Nachfolger : mögliche Nachfolgezustände
- Blätter : Endzustände des Spiels



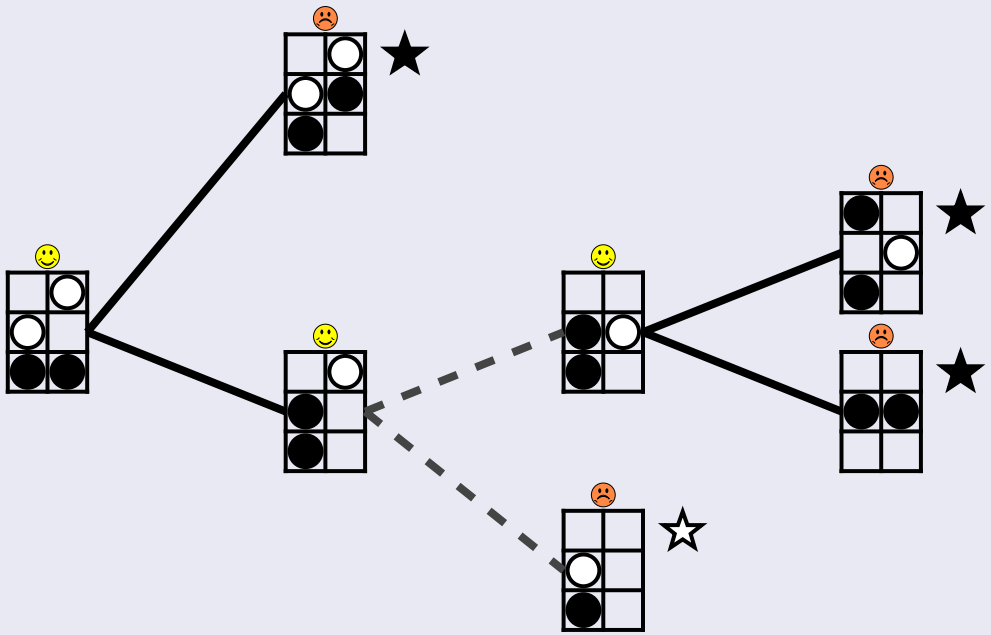
Entscheidungsbaum

Unendlich Bäume

- Zyklische Spiele
- Schach : 3 mal gleicher Spielzustand

Analyse eines Spielzustands

- Ein Zustand ist ein Gewinnzustand für einen Spieler, wenn es einen Nachfolgezustand gibt, der für den Gegner ein Verlustzustand ist.



Schlussfolgerung

- Bei endlichem Baum ist das Spiel bekannt (von den Blättern ausgehend).
- Das Spiel ist bekannt, wenn der Spieler, der zuerst am Zug ist, gewinnt oder verliert, wenn beide Spieler *perfekt* spielen.
- Problem : Baumgrösse, benötigte Rechenzeit (der Speicher ist nicht das Problem)

Diese Funktion liefert $+1$, wenn es ein Gewinnzustand für Spieler p ist :

Algorithmus

function evaluateState(state s, player $p \in \{1, 2\}$)
    if Game over **then**
        return -1 (or 1) if player p has lost (or won)
    end if
    for all moves m for player p at state s **do**
        $s' \leftarrow m$ applied to s
        if evaluateState(s', $3 - p$) $= -1$ **then**
            return 1 ▷ Optionally return winning move
        end if
    end for
    return -1
end function

Problem gelöst

- Für einen endlichen Baum liefert der Algorithmus eine optimale Gewinnstrategie.
- Durchläuft nicht notwendigerweise den ganzen Baum (stoppt, wenn ein Gewinnzug gefunden ist).
- Der Baum kann riesig werden.
- $O(m^d)$ Blätter, m : Anzahl der Spielzüge pro Zustand, d : Anzahl halber Spielrunden

Miniaturschach, 3×3

- $m \approx 3$, $c \approx 6$
- $3^6 = 729$ (tatsächlich 252 Zustände)
- Mit cutoff : 49 Zustände

Ein paar Beispiele

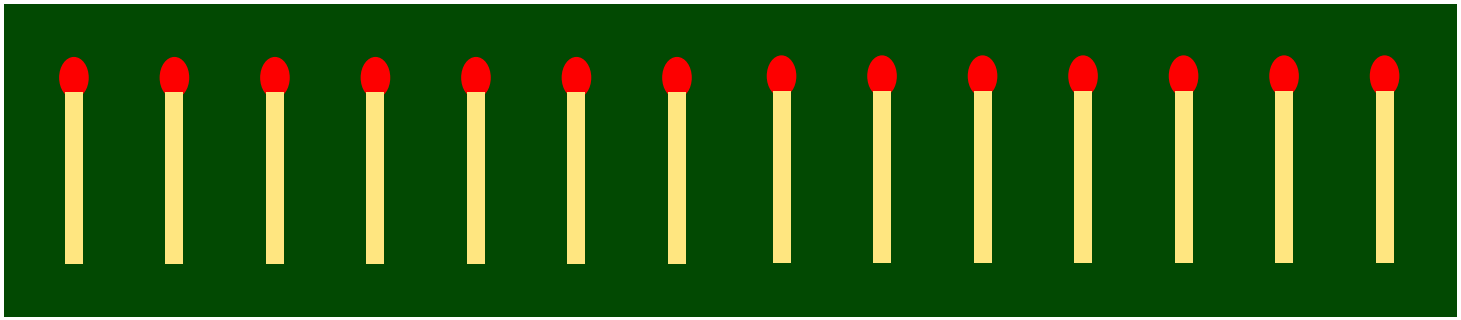
Bäume für 3×2, 4×2 et 3×3

- Vollständige Bäume
- Bäume, bei denen Züge zu Verlustzuständen abgeschnitten sind
- Siehe http ://bit.ly/1a6jyHh

Zugreihenfolge

- Wenn per Zufall immer der Gewinnzug zuerst getestet wird, ist es, als ob die Baumtiefe durch zwei geteilt wird.
- Wenn es keinen Gewinnzug gibt, müssen alle Züge ausprobiert werden.
- An Stelle von n Zuständen werden $\sqrt{n}$ Zustände bewertet.
- Heuristik, um den vielversprechendsten Zug auszuwählen.

Steichholzspiel



Steichholzspiel

- Gegeben sind n Streichhölzer.
- Die Spieler können 1, 2 oder 3 Streichhölzer auswählen.
- Wer das letzte nimmt, gewinnt.

Analyse des Steichholzspiels

Steichholzspiel

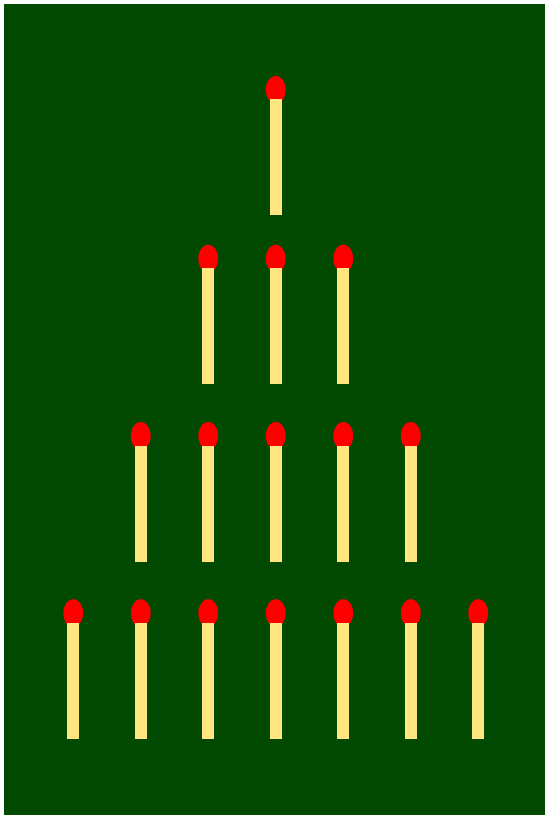
- Wenn 0 Streichhölzer übrig sind, hat man verloren.
- Wenn 1, 2 oder 3 Streichhölzer übrig sind, hat man gewonnen.
- Wenn also 4 übrig sind, verliert man.
- Wenn also 5, 6 oder 7 übrig sind, gewinnt man.
- Man verliert, wenn 8 übrig sind
- ...
- Man gewinnt, wenn man $4m$ Streichhölzer übrig lässt, $m \in \mathbb{N}$

Optimale Strategie

Invariante

- Grösse des Baums $\approx 3^{\frac{n}{2}}$.
- Direkte Berechnung, ob eine Gewinnsituation vorliegt, für beliebiges n.

Nim



Nim

- Aus einer Reihe wird eine zufällige Anzahl von Streichhölzern ausgewählt.
- Wer das letzte nimmt, gewinnt.
- http ://bit.ly/1dGqxec

Geschichte

Geschichte des Spiels Nim

- Bekannt seit dem 16. Jahrhundert.
- Analysiert und gelöst 1901 durch Charles L. Bouton
- Implementiert 1939, Nimatron in New York.

Analyse von Nim

Analyse

- Ungefähre Knotenanzahl im Baum ist $\approx m^d \approx 8^8 \approx 2.4 \cdot 10^7$.
- Tatsächlich sind es 10'292'376 Knoten.
- Aber nur $2 \cdot 4 \cdot 6 \cdot 8 = 384$ Situationen sind möglich.

Dynamische Programmierung

Ansatz der Dynamischen Programmierung

- $f(s)$: wahr/falsch, wenn der Zustand s gewinnt/verliert.
- $N(s)$: Menge aller erreichbaren Zustände nach s
- $f(\emptyset) =$ falsch
- $f(s) = \bigvee_{s' \in N(s)} \neg f(s')$

Implementierung

- Speichere alle Zwischenresultate
- Betrachte jede Konfiguration als eine Zahl in gemischter Basis
- Programmiert in Ruby

Ergebnis des Programms

Output

```
Number of nodes: 10292376
Number of configs: 384
Loosing positions:
[0, 0, 0, 0] [0, 0, 1, 1] [0, 0, 2, 2] [0, 0, 3, 3]
[0, 0, 4, 4] [0, 0, 5, 5] [0, 1, 2, 3] [0, 1, 4, 5]
[0, 2, 4, 6] [0, 2, 5, 7] [0, 3, 4, 7] [0, 3, 5, 6]
[1, 1, 1, 1] [1, 1, 2, 2] [1, 1, 3, 3] [1, 1, 4, 4]
[1, 1, 5, 5] [1, 3, 5, 7]
```

Strategie für Nim

Strategie

- Binärdarstellung der Anzahl der Streichhölzer pro Reihe n_r
- Verlustsituation, wenn
  $g(s) = n_1 \text{ XOR } n_2 \text{ XOR } \ldots \text{ XOR } n_r = 0$

Beweisskizze

- Keine Streichhölzer : Verlustsituation, $g(\emptyset) = 0$, ok.
- $g(s) = 0 \Rightarrow g(s') \neq 0 \quad \forall s' \in N(s)$
- $g(s) \neq 0 \Rightarrow \exists s' \in N(s)$ mit $g(s') = 0$

Beweis der Strategie für Nim

Eigenschaften der Operation XOR

- Kommutativ
- a XOR $b = 0 \quad \Leftrightarrow \quad a = b$
- 0 XOR $a = a \quad \forall a$

$g(s) = 0 \Rightarrow g(s') \neq 0 \quad \forall s' \in N(s)$

- Sei $s' \in N(s)$ und r die geänderte Reihe.
- Dann gilt $g(s') = g(s)$ XOR n_r XOR $n'_r = n_r$ XOR $n'_r \neq 0$ mit $n_r \neq n'_r$.

Beweis der Strategie für Nim

$g(s) \neq 0 \Rightarrow \exists s' \in N(s)$ mit $g(s') = 0$

- Sei $g_r = g(s)$ XOR n_r (Summe ohne Reihe r).
- Es gibt immer $g_r < n_r$. Reduziere dann Reihe r auf g_r.

Widerspruchsbeweis unter Annahme, dass $g(s)$ XOR $n_r > n_r \, \forall r$

- Sei x die Position des höchstwertigen Bits von $g(s) \neq 0$.
- $g(s)$ XOR n_r ändert Bit x in n_r (und andere geringer wertige Bits).
- Aus $g(s)$ XOR $n_r > n_r$ folgt, dass Bit x in n_r 0 ist, für alle r.
- Wenn für alle r das Bit x in n_r 0 ist, dann ist Bit x in $g(s)$ 0.
- Widerspruch !

Der optimale Algorithmus für Nim

Algorithmus

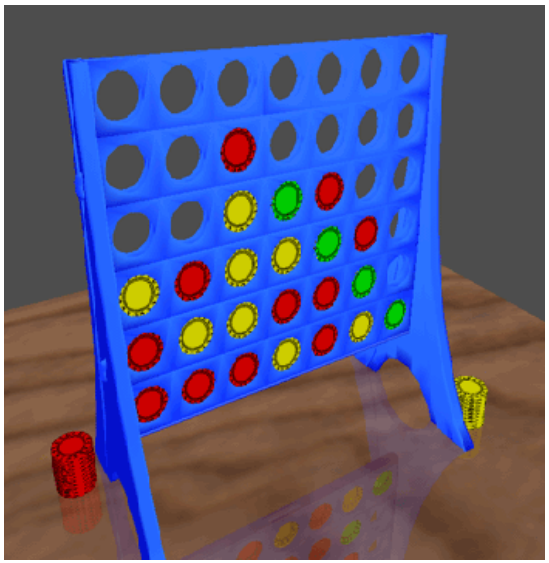
Algorithmus, um einen Zug in Nim zu berechnen

Compute $g(s) = n_1$ XOR n_2 XOR ... XOR n_r
if g(s) = 0 **then**
    Make random move
else
    Pick r s.t. $g_r = g(s)$ XOR $n_r < n_r$ and reduce line r to g_r.
end if

Schlussfolgerung

- Gibt es wenig Zustände, sollten Zwischenergebnisse gespeichert werden.
- In seltenen Fällen, ist eine direkte Berechnung möglich.
- Verwendung der Seite Nim mit ?cheat=true

Komplexere Spiele



Zum Beispiel “Vier gewinnt”

- Spielfeld mit 7×6 Fächern
- Spielsteine fallen von oben nach unten
- Wer 4 Spielsteine in einer Reihe positioniert, gewinnt

Geschichte

Gelöst 1988 : Weiss gewinnt, wenn Weiss in der Mitte beginnt, Schwarz kann ein Unentschieden erreichen, wenn Weiss an einer Position neben der Mitte beginnt, ansonsten gewinnt Schwarz.

Gegenwärtig

Heute kann das Spiel in wenigen Minuten gelöst werden.

Erste Analyse

Analyse

- 7 Möglichkeiten für einen Spieler
- Maximal 42 Züge pro Spiel
- Baum mit $\approx 7^{42} \approx 3 \cdot 10^{36}$ Knoten
- Maximal $3^{42} \approx 10^{21}$ mögliche Positionen
- Besser : $(1 + 2 + 4 + 8 + \ldots + 64)^7 \approx (2^7)^7 = 2^{49} \approx 5 \cdot 10^{15}$ mögliche Positionen
- Tatsächlich : $4531985219092 \approx 5 \cdot 10^{13}$ mögliche Positionen

Schwierigkeiten

Schwierigkeiten

- Baum zu gross
- Speicherung / Vergleich von Positionen
- Effizientes Erkennen von Gewinnpositionen

Tricks

- Heuristik, um eine Position zu bewerten
- Beschränkung der Baumtiefe
- α, β-Pruning
- Bit-Felder um Positionen zu speichern

Bewertungsheuristik

Bewertungsheuristik $h(s, p)$

- $h(s, p)$ liefert -1, wenn die Spielsituation s für Spieler p eine Verlustsituation ist, ($+1$, wenn es eine Gewinnsituation ist).
- Sonst liefert $h(s, p)$ einen Wert in $[-1, +1]$.
- Ein Wert nahe bei 1 zeigt eine günstige Situation an.
- $h(s_1, p) > h(s_2, p)$ zeigt an, dass s_1 günstiger für p ist als s_2.
- $h(s, p) = -h(s, 3 - p)$ (Vorzeichenumkehr für anderen Spieler)

Beschränkung der Baumtiefe

- Falls das Spiel bei einer Baumtiefe $d_{\max}$ nicht beendet ist, verwenden wir h, um die Nachfolgesituationen zu bewerten.

Bewertung mit Hilfe einer Heuristik

```
function evaluateState(state s, player p, depth d, d_max)
    if d = d_max or Game over then
        return h(s, p)
    else
        b ← −1                          ▷ Best evaluation the current player can get so far
        for all moves m for player p at state s do
            s' ← m applied to s
            e ← evaluateState(s', 3 − p, d + 1, d_max)
            if −e = 1 then
                return 1                          ▷ Optionally return winning move
            else
                if −e > b then
                    b ← −e                ▷ Optionally remember best move so far
                end if
            end if
        end for
        return b                          ▷ Optionally return best move remembered
    end if
end function
```

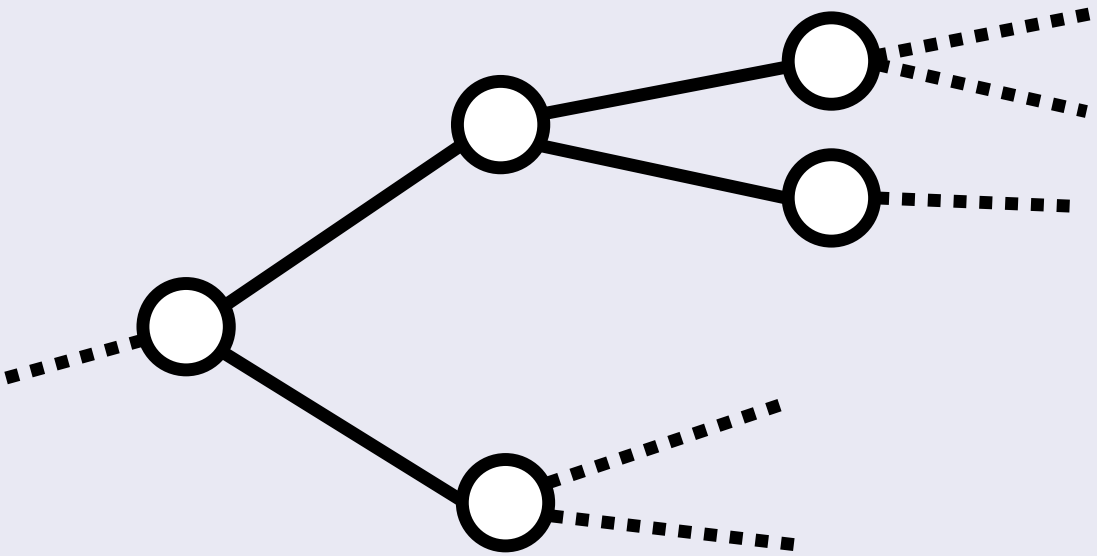
Performance des Algorithmus

Bottlenecks

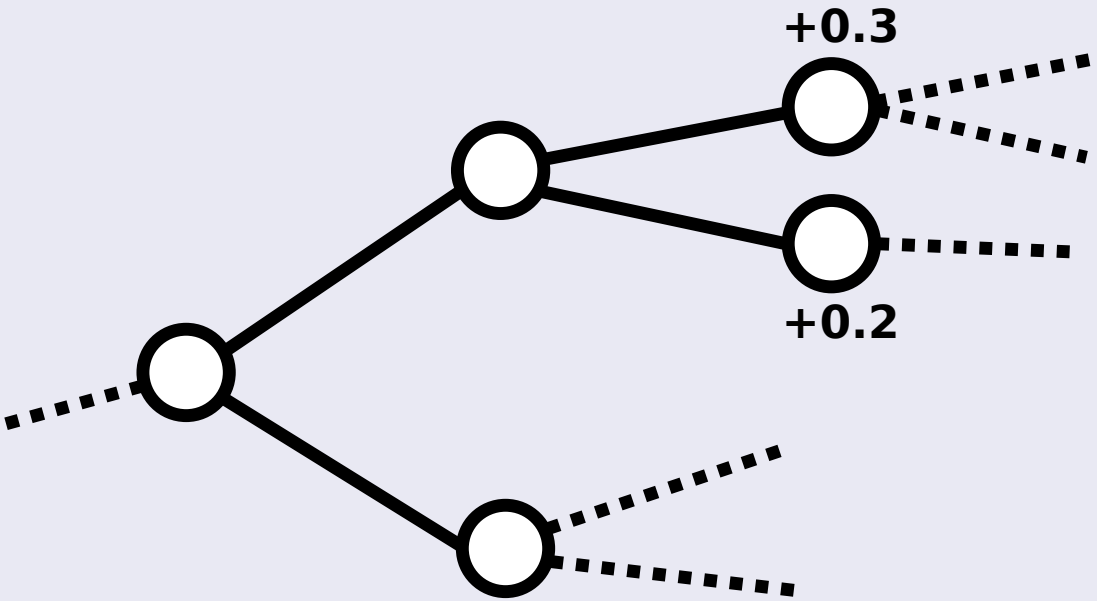
- Geschwindigkeit der Berechnung von $h(s)$
  - Bit-Fields
  - Inkrementelle Funktion
- Geschwindigkeit der Erzeugung von s'
  - Neue Datenstruktur beziehungsweise
  - Modifikation der aktuellen Struktur, dann von vorne beginnen

Übung auf Papier, Bewertung zwischen $[-10, +10]$, "Tree 6"
http ://bit.ly/1dGzF2o

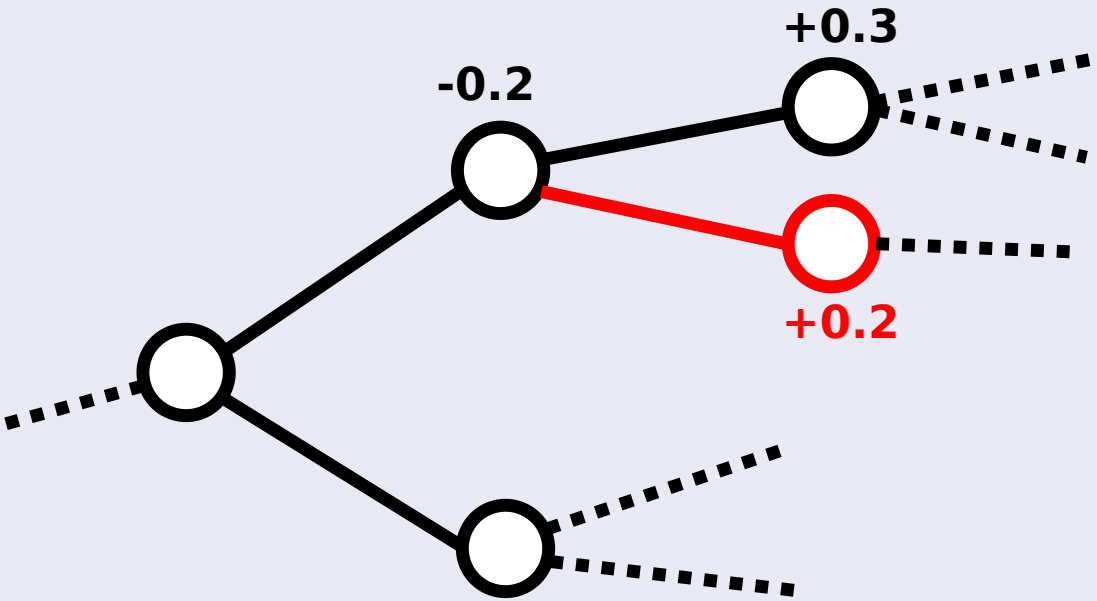
Grundidee von α, β-Pruning



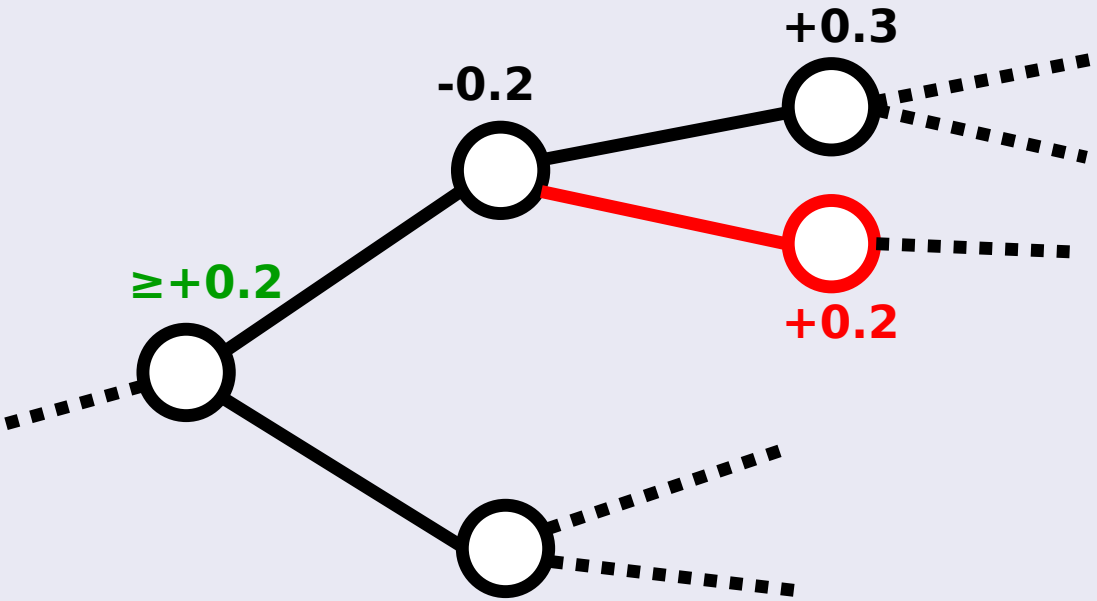
Grundidee von α, β-Pruning



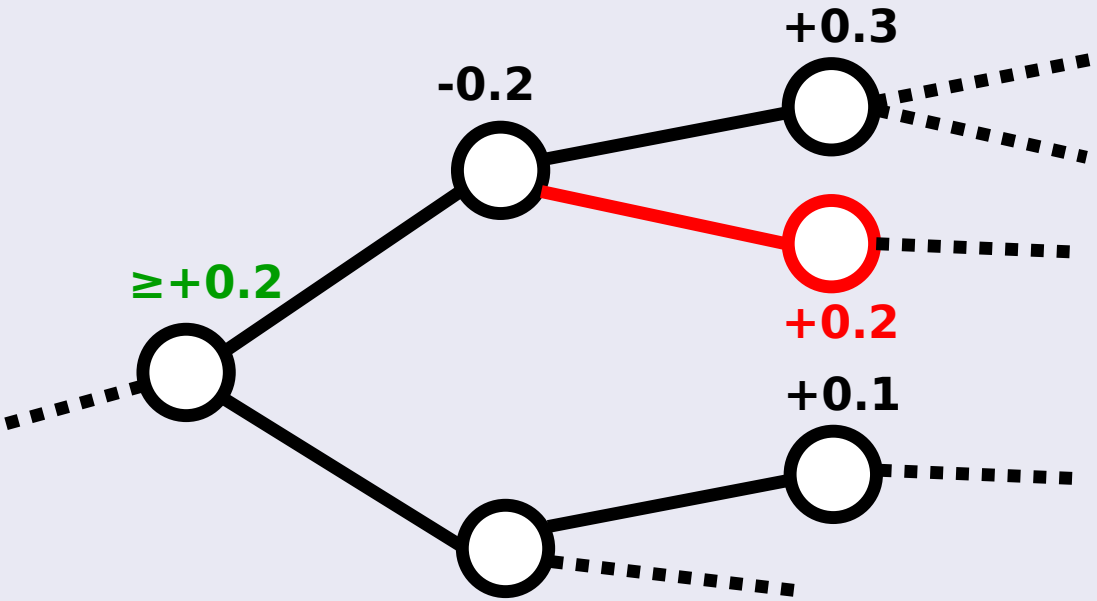
Grundidee von α, β-Pruning



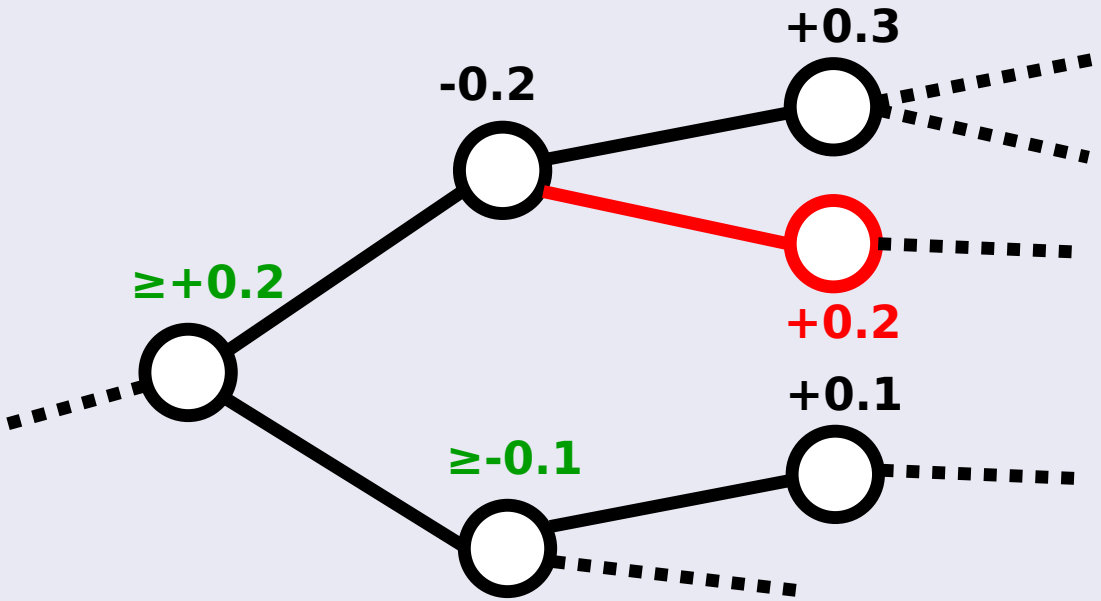
Grundidee von α, β-Pruning



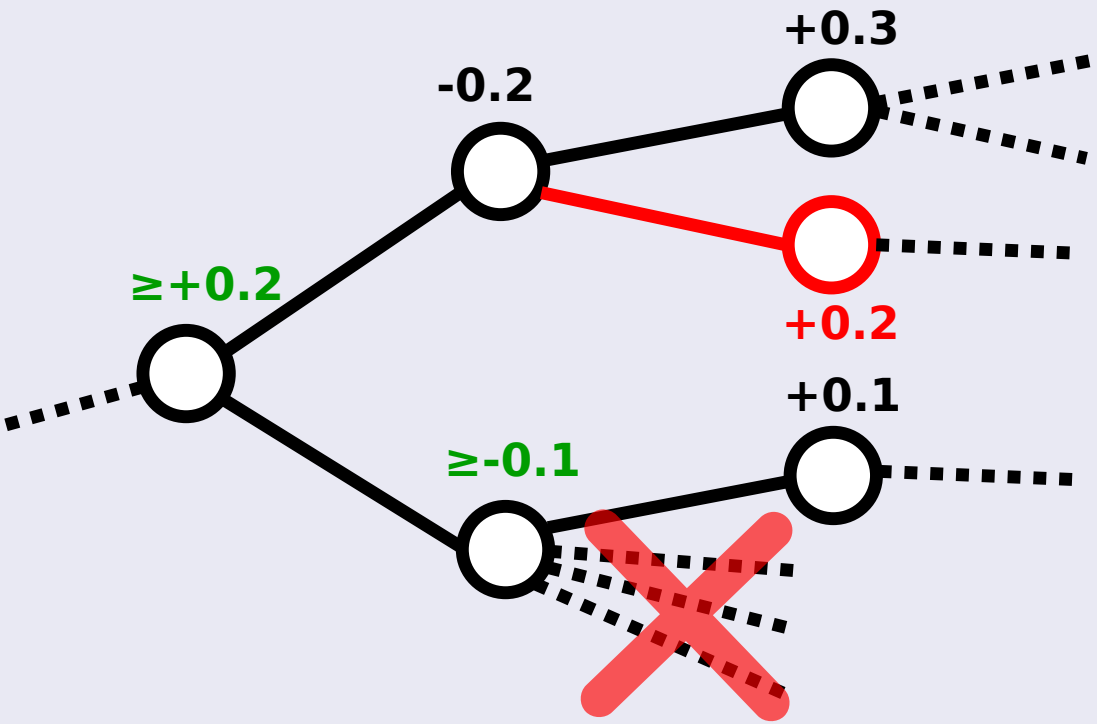
Grundidee von α, β-Pruning



Grundidee von α, β-Pruning



Grundidee von α, β-Pruning



α, β-Pruning

- Bei jedem Aufruf ist $[\alpha, \beta]$ das Intervall der interessanten Bewertungen.
- Beim Anfangaufruf : $[\alpha, \beta] = [-1, 1]$.
- $-\beta$: Beste Bewertung für den Gegner zum gegenwärtigen Zeitpunkt.
- Wenn ich einen besseren Wert als β finde, wird der Gegner diese Wahl nicht treffen. Abbruch der Suche.
- Wenn für die aktuelle Situation das Intervall $[\alpha, \beta]$ ist, ist das Nachfolgeintervall $[-\beta, -\alpha]$.
- α wird aktualisiert, wenn eine bessere Bewertung gefunden wird.

Übung auf Papier, $[\alpha, \beta] = [-10, +10]$, “Tree 42”
http ://bit.ly/1dGzF2o

α, β-Pruning

```
function evaluateState(state s, player p, depth d, d_max, α, β)
    if d = d_max or Game over then
        return h(s)
    else
        b ← −1                                   ▷ Best evaluation the current player can get so far
        for all moves m for player p at state s do
            s′ ← m applied to s
            e ← evaluateState(s′, 3 − p, d + 1, d_max, −β, −α)
            if −e = 1 then
                return 1                         ▷ Optionally return winning move
            else
                if −e > b then
                    b ← −e                       ▷ Optionally remember best move so far
                    if −e > α then
                        α ← −e
                        if α ≥ β then
                            return b             ▷ Optionally return this good enough move
                        end if
                    end if
                end if
            end if
        end for
        return b                                 ▷ Optionally return best move remembered
    end if
end function
```

Vier gewinnt

Implementationsdetails

- Darstellung durch Bit-Feld
- Bewertung einer Situation
- Speicherung von Zwischenresultaten
- Iterative Deepening

Kodierung durch Bit-Feld

Bit-Field

6	13	20	27	34	41	48
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42

Kodierung für einen Spieler

- Total 49 Bits für 42 Fälle.
- Zwei 64-Bit Zahlen für die zwei Farben c_1 und c_2.
- Die oberste Reihe ist normalerweise 0.
- $c_1 \vee c_2$ liefert ein Bitmap der besetzten Positionen.

Kodierung durch Bit-Feld

Bit-Field

6	13	20	27	34	41	48
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	**7**	**14**	**21**	**28**	**35**	**42**

Kodierung einer Situation

- Sei c_b die Zahl, bei der die tiefsten Bits auf 1 gesetzt sind.
- $c_1 \vee c_2 + c_b$ liefert die Einhüllende der besetzten Positionen.
- $(c_1 \vee c_2 + c_b) \vee c_1$ kodiert die Position eindeutig mit 49 Bits.
- Alle Operationen $\vee$ sind disjunkt und können durch $+$ ersetzt werden.
- $2c_1 + c_2 + c_b$ ist das Gleiche.

Gewinnerkennung

Bit-Feld

6	13	20	27	34	41	48
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42

Vertikale Gewinnerkennung

- $d \leftarrow c_1 \wedge (c_1 >> 1)$
- d markiert die Positionen mit zwei aufeinanderfolgenden Spielsteinen.
- $e \leftarrow d \wedge (d >> 2)$ markiert die Positionen mit vier aufeinanderfolgenden Spielsteinen.
- Wenn $e \neq 0$ gewinnt Spieler 1.
- Die leere Linie oben ist nützlich.

Gewinnerkennung

Bit-Feld

6	13	20	27	34	41	48
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42

Horizontalen Gewinnerkennung

- $d \leftarrow c_1 \wedge (c_1 >> 7)$
- d markiert die Positionen mit zwei aufeinanderfolgenden Spielsteinen.
- $e \leftarrow d \wedge (d >> 14)$ markiert die Positionen mit vier aufeinanderfolgenden Spielsteinen.
- Wenn $e \neq 0$ gewinnt Spieler 1.
- Die Bits 49 bis 54 müssen Null sein.

Gewinnerkennung

Bit-Feld

6	13	20	27	34	41	48
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42

Diagonalen Gewinnerkennung ↗

- $d \leftarrow c_1 \wedge (c_1 >> 8)$
- d markiert die Positionen mit zwei aufeinanderfolgenden Spielsteinen.
- $e \leftarrow d \wedge (d >> 16)$ markiert die Positionen mit vier aufeinanderfolgenden Spielsteinen.
- Wenn $e \neq 0$ gewinnt Spieler 1.
- Die Bits 49 bis 55 müssen Null sein.

Gewinnerkennung

Bit-Feld

6	13	20	27	34	41	48
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42

Diagonalen Gewinnerkennung ↘

- $d \leftarrow c_1 \wedge (c_1 >> 6)$
- d markiert die Positionen mit zwei aufeinanderfolgenden Spielsteinen.
- $e \leftarrow d \wedge (d >> 12)$ markiert die Positionen mit vier aufeinanderfolgenden Spielsteinen.
- Wenn $e \neq 0$ gewinnt Spieler 1.

Die grösste Herausforderung

Bewertungsheuristik

- Sehr schnell (inkrementell ?)
- Präzise (? ?)

Trade off

- Rapide : grosse Tiefe möglich
  - Nützlich am Ende
  - Vollständiges Durchsuchen möglich
- Präzise : kleine Tiefe
  - Nützlich am Anfang
  - Vollständiges Durchsuchen unmöglich

Heuristik

Implementierung

- Anzahl der Möglichkeiten, mit den bereits platzierten Steinen 4 zu erreichen.
- Schnell
- Mittlere Qualität
- Implementierung mittels logischer Bitoperationen

Speicherung der Bewertungen

Hashtable

- Test, ob bereits evaluiert
- Für jeden Zustand : Speicherung der Bewertung
- Gemäss Spiel, nennenswerter Gewinn
- Kodierung mit 49 Bits, sehr gut geeignet für Hashtable
- Achtung : Bewertung kann unvollständig sein, wenn $\alpha \geq \beta$

Move ordering

- Ordnung der Züge gemäss vorheriger Bewertung
- Besseres Pruning.

Iterative deepening

Iterative deepening

- Erhöhung der Suchtiefe bis zum Ablauf einer Zeitbeschränkung
- Wenig overhead (Faktor $\approx \frac{1}{m-1}$, m : Anzahl der Züge in einer Situation)
- Verwendung der Bewertungen für das move-ordering

"Pathologische" Spiele

Spiele ohne vom Computer beherrschbare Strategie

- Go
  - 19×19
  - 150 bis 250 Züge pro Situation
- Quoridor
  - 11×11
  - 4 Richtungen und 80-180 Positionen, um eine Mauer zu platzieren.

Zusammenfassung

- Effiziente Darstellung und Bewertung einer Situation $\star$
- Bewertungsheuristik $\star$
- α, β-Pruning
- Move ordering
- Hashtable
- Iterative deepening

Wozu ist Spieltheorie noch gut ? (ausser zum Spielen)

Zum Beispiel, um Systeme zu verifizieren.

Wozu ist Spieltheorie noch gut ? (ausser zum Spielen)

Zum Beispiel, um Systeme zu verifizieren.

Der *Verifizierer* (verifier) spielt gegen das System(modell) und versucht, es in eine Fehlersituation zu bringen (der Verifizierer ist der Gewinner).

Wozu ist Spieltheorie noch gut ? (ausser zum Spielen)

Zum Beispiel, um Systeme zu verifizieren.

Der *Verifizierer* (verifier) spielt gegen das System(modell) und versucht, es in eine Fehlersituation zu bringen (der Verifizierer ist der Gewinner).

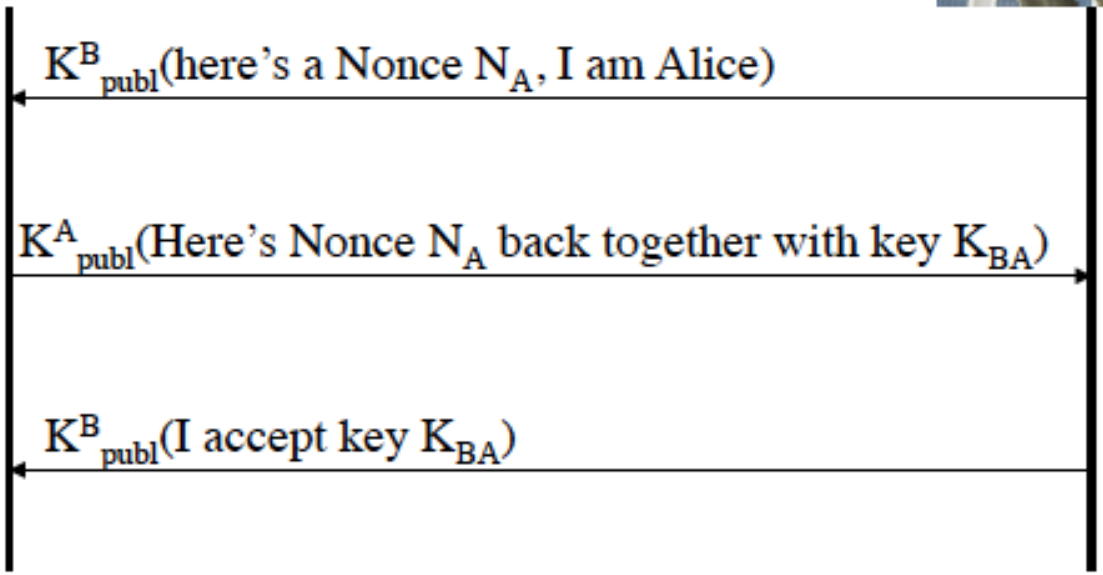
Schafft er das nicht (das System ist der Gewinner), dann wird das System als korrekt angesehen.

Konkret ?

Wir betrachten hier im folgenden nur ein Beispiel, um die Idee dieses Ansatzes zu verstehen.

Das Needham-Schroeder-Protokoll

Bob   Alice



Das Needham-Schroeder-Protokoll als Spiel : die Züge

In dem Spiel geht es darum, in sicherer Weise einen geheimen Schlüssel zwischen zwei Spielern auszutauschen. Die gemäss Protokoll möglichen Züge sind :

- Spieler 1 schickt Spieler 2 eine Nachricht, die nur Spieler 2 lesen kann, in der Spieler 1 eindeutig identifiziert wird und die zusätzlich eine grosse, frisch generierte Zufallszahl N enthält.
- Spieler 2 schickt an Spieler 1 eine Nachricht, die nur Spieler 1 lesen kann, in der die Zufallszahl N wiederholt wird und die eine weitere grosse, frisch generierte Zufallszahl K enthält.
- Spieler 1 schickt Spieler 2 eine Nachricht, die nur Spieler 2 lesen kann, in der die Zufallszahl K wiederholt wird.

Das Needham-Schroeder-Protokoll als Spiel : die Spielregeln

Die Spielregeln lauten :

- Der Verifizierer spielt als Spieler im Spiel mit.
- Der Verifizierer darf alles, was kryptographisch möglich ist :
  - Nachrichten lesen, wenn er den passenden Schlüssel besitzt
  - Nachrichten aufhalten und zu einem anderen Ziel weiterleiten
  - Nachrichten umstrukturieren, soweit die Struktur sichtbar ist
  - Nachrichten neu verschlüsseln, wenn er den passenden Schlüssel besitzt
- Der Verifizierer gewinnt, wenn er einen Schlüssel kennt, von dem ein anderer Spieler glaubt, dass er ihn nicht kennen kann.
- Das Protokoll gewinnt, wenn der Verifizierer nicht gewinnt.

Achtung !

Die Spieler sehen sich nicht ! Sie verlassen sich nur auf die Nachrichten, die sie erhalten.

Achtung !

Die Spieler sehen sich nicht ! Sie verlassen sich nur auf die Nachrichten, die sie erhalten.

Ausserdem nehmen wir an, dass die Kryptographie perfekt funktioniert.

Die “Verifikation” des Protokolls

Wir sehen uns an der Tafel an, was passiert, wenn der Verifizierer die Rolle von Alice bzw. die Rolle von Bob im Protokollspiel einnimmt.

Die “Verifikation” des Protokolls

Wir sehen uns an der Tafel an, was passiert, wenn der Verifizierer die Rolle von Alice bzw. die Rolle von Bob im Protokollspiel einnimmt.

Das Protokoll funktioniert also, oder?

Mehrere Spieler (wie im Synchronschach)

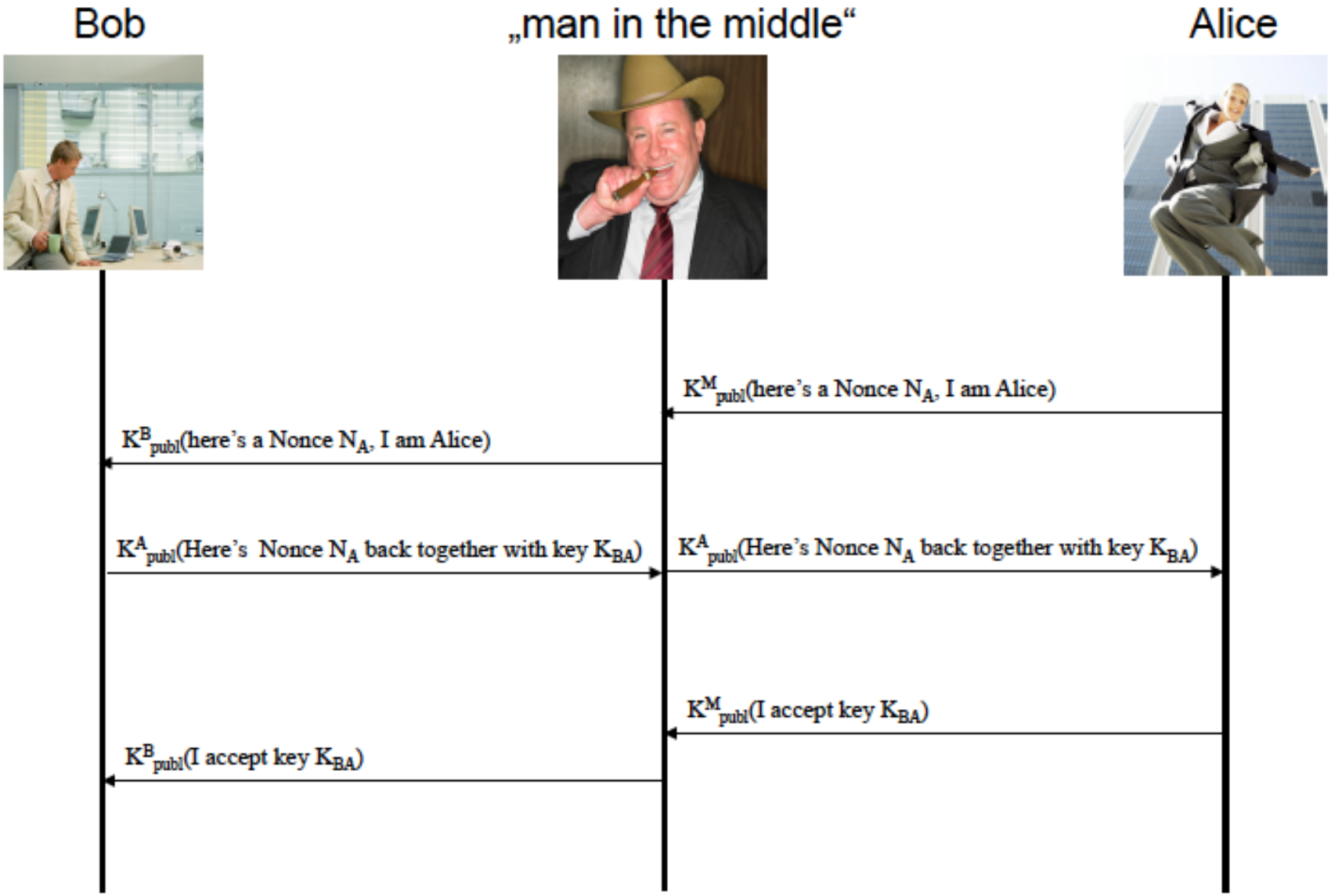
Wir sehen uns an der Tafel an, was passiert, wenn drei Spieler spielen.

Mehrere Spieler (wie im Synchronschach)

Wir sehen uns an der Tafel an, was passiert, wenn drei Spieler spielen.

Autsch !

Der Fehler im Needham-Schroeder-Protokoll



Unendliche Spiele

Es kann sein, dass ein Verifikationsspiel erst nach einer unendlichen Anzahl von Zügen gewonnen/verloren wird.

Unendliche Spiele

Es kann sein, dass ein Verifikationsspiel erst nach einer unendlichen Anzahl von Zügen gewonnen/verloren wird.

Bis dahin bin ich allerdings tot ...

Unendliche Spiele

Es kann sein, dass ein Verifikationsspiel erst nach einer unendlichen Anzahl von Zügen gewonnen/verloren wird.

Bis dahin bin ich allerdings tot ...

Was soll das also???

Gewinnen kann ich also nicht

Was möchte ich dann ?

Gewinnen kann ich also nicht

Was möchte ich dann ?

Ich möchte eine Strategie finden, die für mich jeweils einen Zug auswählt, sodass ich, wenn ich mich immer gemäss der Strategie verhalte, das Spiel gewinnen werde (eventuell erst nach unendlich vielen Zügen).

Gewinnen kann ich also nicht

Was möchte ich dann ?

Ich möchte eine Strategie finden, die für mich jeweils einen Zug auswählt, sodass ich, wenn ich mich immer gemäss der Strategie verhalte, das Spiel gewinnen werde (eventuell erst nach unendlich vielen Zügen).

Ich suche einfach nur eine *Gewinnstrategie* (winning strategy).

Wann funktioniert diese Verifikationsmethode ?

Wenn der Spielraum endlich ist.

Wann funktioniert diese Verifikationsmethode ?

Wenn der Spielraum endlich ist.

Dann lässt sich das Spiel durch einen endlichen Automaten beschreiben (mit voneinander unterschiedenen akzeptierenden und nichtakzeptierenden Zuständen).

Wann funktioniert diese Verifikationsmethode ?

Wenn der Spielraum endlich ist.

Dann lässt sich das Spiel durch einen endlichen Automaten beschreiben (mit voneinander unterschiedenen akzeptierenden und nichtakzeptierenden Zuständen).

Anhand des Automaten lässt sich algorithmisch testen, ob eine Gewinnstrategie des Verifizieres existiert.

Das war's schon zu unendlichen Spielen

Ich hoffe, die Grundidee und der Anwendungskontext (Verifikation) sind einigermassen klar geworden.

Gibt es Fragen ?

6 Werbung
- Informatikbiber
- Cybercamp 2014
- SSIE / SVIA

Informatikbiber

- http ://castor-informatique.ch/
- Online-Wettbewerb vom 11. bis 15. November 2013
- Bisherige Wettbewerbe sind verfügbar : http://concours.castor-informatique.ch/index.php?action=user_competitions

Cybercamp 2014

- http ://cybercamp.unifr.ch/
- 7. bis 11. Juli 2014
- Ein einwöchiges Informatik-Sommercamp.

SSIE / SVIA

- http ://svia-ssie-ssii.ch/
- Schweizerischer Verein für Informatik in der Ausbildung
- Austausch unter Lehrkräften
- Informationen über verschiedene Angebote
- Förderung von Informatik in Schulen

Und jetzt zur Praxis

An die Tastatur!

Fragen?

Und jetzt zur Praxis

An die Tastatur!

Fragen?

Also ...

- Software installieren
- Öffnen der Sourcecodes in Netbeans
- Erzeugung und Lesen der JavaDoc