Fachbereich Informatik der Universität Hamburg

Vogt-Kölln-Str. 30 ◊ D - 22527 Hamburg / Germany

University of Hamburg - Computer Science Department

Mitteilung Nr. 244

Lebende Blöcke beim Go

Ein formaler Ansatz unter Verwendung von Petri-Netzen

Matthias Müller-Prove

Arbeitsbereich „Wissens- und Sprachverarbeitung“

FBI - HH - M - 244 / 95

März 1995

Lebende Blöcke beim Go

Ein formaler Ansatz unter Verwendung von Petri-Netzen

Matthias Müller-Prove

~~mprove~~ mprove@acm.org ~~urg.de~~

Betreuung durch Prof. Ph.D. Christian Freksa
Universität Hamburg, Fachbereich Informatik
Arbeitsbereich Wissens- und Sprachverarbeitung

Inhalt

Go ist ein Strategiespiel für 2 Spieler. Es wurde vor etwa 4000 Jahren in China erfunden. Das Ziel des Spiels besteht darin, möglichst viel Gebiet auf dem Spielbrett mit seinen Steinen zu umzäunen. Die wenigen und einfachen Regeln führen allerdings zu einem nuancenreichen und komplexen Spielverlauf. Nebeneinander liegende Steine einer Farbe bilden Blöcke. Eine von einem Block umschlossene Freiheit heißt Auge. Blöcke, die keine leeren Nachbarfelder mehr haben, werden vom Brett als Gefangene entfernt. Für einen Spieler ist es also spielentscheidend, daß seine Blöcke einerseits nicht dieses Schicksal erleiden müssen und daß sie andererseits möglichst viel Gebiet umschließen.

Diese Arbeit befaßt sich mit der Bewertung spezieller Spielsituationen beim japanischen Go. Ein Block mit 2 Augen lebt, da der Gegner wegen des Selbstmord-Verbots nicht beide Augen gleichzeitig besetzen kann. Es gibt aber noch weitaus komplexere Gruppierungen von Blöcken die leben, obwohl die einzelnen Blöcke dabei nichteinmal ein Auge haben. Die Bedingungen, die für solche Blöcke gelten, werden untersucht und formalisiert.

1 Einleitung

Schachprogramme spielen heute auf Großmeisterniveau. Die existierenden Go-Programme hingegen stellen nur für den Anfänger eine Herausforderung dar. Trotz der einfachen Regeln besitzt Go einen Variantenreichtum, der den des Schachs bei weitem übertrifft. Bei jedem Zug gibt es beim Go zwischen 200 und 300 Möglichkeiten; beim Schach gibt es nur etwa 30. Außerdem umfaßt eine Go-Partie etwa 200 Züge, eine Schach-Partie kommt mit 100 Halbzügen aus.[1] Die möglichen Züge beim Schachspiel werden von [Shannon 1950] mit 10^{120} angegeben. Die möglichen Züge beim Go liegen laut [Zobrist 1970] bei 10^{761}. Schon dieser Vergleich zeigt, daß eine Brute-Force Methode beim Go zum Scheitern verurteilt ist.

Eine Go-Partie läßt sich in drei Phasen einteilen. Eröffnung, Mittelspiel und Endspiel. Bibliotheken reichen in der Eröffnungsphase nicht sehr weit, so daß bald Konzepte wie Einfluß und Stärke der Go-Steine relevant werden. Beides sind schwer zu fassende Begriffe, die daher in existierenden Go-Programmen nur unzureichend implementiert sind.

[1] 100 Halbzüge sind 50 Züge. Beim Schach wird das Setzen von Weiß und Schwarz als 1 Zug gezählt; beim Go wird jeder Zug gezählt.

Im Mittelspiel werden Konzepte wie Gebiet, Leben und Tod zunehmend wichtiger. Die Frage, ob ein Bereich des Go-Bretts dem weißen oder dem schwarzen Spieler gehört, hängt von der Frage ab, ob die Blöcke, die das Gebiet abgrenzen geschlagen werden können oder nicht. In [Benson 1976] wurden erstmalig lebende Blöcke beim Go mathematisch untersucht. Sein Formalismus wird in dieser Arbeit skizziert. Auch [Kraszek 1988] stellt einen Leben und Tod Algorithmus vor, der in seinem Programm „Star of Poland" erfolgreich eingesetzt wird.
[Nievergelt 1994] schreibt : »Es hat sich gezeigt, daß Go im Endspiel eine Struktur aufweist, die der kombinatorischen Spieltheorie von Berlekamp, Conway und Guy [Berlekamp 1982] nahezu entspricht.« In [Berlekamp 1994] ist die Anwendung der Spieltheorie auf das Go-Endspiel dargestellt. Als Voraussetzung gilt jedoch, daß alle beteiligten Blöcke leben, bzw. nicht geschlagen werden können.

Dem Aspekt von Leben und Tod von Blöcken, der im Mittelspiel so wichtig ist, und der eine Voraussetzung für das spieltheoretisch untersuchte Endspiel ist, widmet sich diese Arbeit.

1.1 Gliederung

In Kapitel 1 „Einleitung" werden – neben dieser Gliederung – die Regeln des Go-Spiels vorgestellt. Als Basis für diese Arbeit werden die Begriffe Stein, Block, Nachbarschaft und Umgebung definiert.
Der letzte Teil der Einleitung schildert das Problem bei der Bewertung lebender Blöcke.

Das Kapitel 2 „Ketten" beschäftigt sich mit einem eingeschränkten Problem. Es werden nur Block-Gruppen untersucht, die die Linearitätsanforderung erfüllen. Blöcke dürfen nur wie die Perlen einer Kette aneinander gereiht werden. Demnach darf jeder Block nur mit höchstens zwei Nachbarblöcken über gemeinsame Augen verbunden sein. Für solche Block-Gruppen wird mit Hilfe einer Typ-3 Grammatik entschieden, ob die Blöcke leben oder nicht. Die Ergebnisse dieses Kapitels liefern Hinweise für den allgemeineren Ansatz in Kapitel 3.

Kapitel 3 „Go-Netze" – Die untersuchten Block-Gruppen des vorigen Kapitels decken nur einen kleinen Teil der real vorkommenden Spielsituationen auf dem Go-Brett ab. Zur Beschreibung komplexerer Block-Gruppierungen werden in diesem Abschnitt die statischen Askete der Petri-Netze herangezogen. Für die nun beschreibbar gewordenen Gruppen wird ein Algorithmus vorgestellt, der ermittelt, ob die Blöcke leben.
Es wird gezeigt, daß das Verfahren für Ketten die gleichen Resultate liefert, wie die Grammatik aus Kapitel 2. Die Korrektheit des Algorithmus wird bewiesen, indem die Äquivalenz zu dem Formalismus von Bensons gezeigt wird. Ein Prototypen des Verfahrens wurde in PROLOG implementiert. Das Programm liefert effizient und effektiv die Bewertungen für vorgegebene Go-Netze.

Kapitel 4 „Augen“ – Für den Begriff des Auges gibt es bisher keine einheitliche Definition. Da er aber für lebende Blöcke eine zentrale Bedeutung einnimmt, muß er präziser formuliert werden. In dieser Untersuchung gibt es folglich mehrere Varianten für den Augen-Begriff. Hier eine Übersicht aller vorkommenden Augen-Definitionen:

Definition 6 „eigenes und gemeinsames Auge“
Definition 11 „vitales gemeinsames Auge“
Definition 12 „effektives Auge“
Definition 19 „Auge (nach Bozulich)“
Definition 20 „Auge“
Definition 21 „großes Auge“
Definition 22 „Eye & Joiner (nach Benson)“

Der Augenbegriff aus Definition 6 ist vorläufig für die Definitionen 20 und 21. Definition 11 und Definition 12 postulieren spezielle Eigenschaften von Augen, die bei den Algorithmen zur Bewertung lebender Blöcke wichtig sind. In Definition 19 und Definition 22 werden Richard Bozulich und David Benson mit ihren Augendefinitionen zitiert, die sich von dem hier benutzen Augenbegriff unterscheiden. [Bozulich 1987] ist ein Go-Lehrbuch, das sich an den Anfänger richtet.

In dem Kapitel 4 „Augen“ wird außerdem eine Struktur für Formen vorgestellt, mit der große Augen typisiert werden können. Große Augen, die sich in ihrer Form völlig unterscheiden, werden dadurch in gemeinsame Klassen eingeteilt. Alle großen Augen einer Klasse haben in bezug auf das Leben des sie umgebenden Blocks die gleiche Bedeutung. Es genügt nun, einmalig die Bedeutungen der Klassen zu bestimmen; aufwendige Einzelfalluntersuchungen werden so vermieden.

1.2 Spielregeln

Go ist ein ostasiatisches Brettspiel für zwei Personen. Gespielt wird mit beliebig vielen schwarzen und weißen gleichförmigen Steinen auf einem anfangs leeren Brett mit 19 mal 19 sich rechtwinkelig kreuzenden Linien. Die Spieler setzen abwechselnd einen ihrer Steine auf irgendeinen unbesetzten Schnittpunkt. Es besteht kein Zugzwang.

Die hier aufgeführten Regeln sind [Hamann 1985] entnommen.

Regel 1

Jeder Spieler versucht mit Steinen seiner Farbe möglichst viele leere Punkte (*Gebiet*) zu umzäunen, wobei der Brettrand als Grenze zählt und nicht besetzt werden muß. Jeder Gebietspunkt zählt eine Einheit.

Regel 2

Eingeschlossene Steine des Gegners werden vom Brett entfernt und zählen als *Gefangene* ebenfalls eine Einheit.

Ko-Regel

Ein Stein, der soeben einen Stein des Gegners geschlagen hat, darf von diesem nicht sofort wieder geschlagen werden; es muß mindestens ein Zug an anderer Stelle erfolgen.

Selbstmord-Verbot

Das Setzen eines Steines auf einen Platz, an dem er keine Freiheit hat, ist verboten, es sei denn, daß mit dem Zug Steine des Gegners geschlagen werden.

Das Spiel endet, wenn beide Spieler gepaßt haben. Gewonnen hat derjenige Spieler mit den meisten Gebietspunkten und den meisten Gefangenen.

1.3 Begriffsdefinitionen

Definition 1 Ban & Feld

Das Spielbrett, genannt *Ban*, besteht aus einem 19 mal 19 Gitterraster. Die Schnittpunkte des Rasters sind die *Spielfelder*, auf die die Spielsteine gesetzt werden.

Definition 2 Nachbarschaft

Zwei Felder heißen *benachbart*, falls sie auf derselben Linie oder Reihe liegen und nur ein Feld voneinander entfernt sind.

Definition 3 Umgebung

Die *Umgebung* einer Menge von Feldern A umfaßt diejenigen Felder, die ein Nachbarfeld in A haben, aber nicht selbst in A liegen. Ein Menge von Feldern A ist *umgeben* von einer Menge von Feldern B, falls die Umgebung von A in B enthalten ist.

Sei A die Menge von Feldern, die in Bild 1 mit schwarzen Steinen besetzt sind. Die Umgebung von A ist durch ⊗ markiert. Jede Menge von Feldern B, die die ⊗-Felder enthält, umgibt die schwarzen Steine.

Definition 4 Zusammenhang

Eine Menge von Feldern M heißt *zusammenhängend*, gdw. es von jedem Feld der Menge zu jedem anderen Feld der Menge eine Folge paarweise benachbarter Felder gibt, die alle in M liegen.

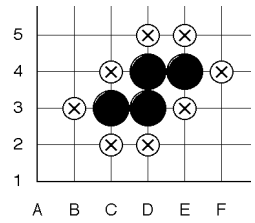


Bild 1 Umgebung eines Blocks

Definition 5 Block, Freiheit und Leben

Eine zusammenhängende Mengen von Steinen einer Farbe heißt *Block*. Die unbesetzten Nachbarfelder eines Blocks sind seine *Freiheiten*. Besitzt ein Block keine Freiheiten mehr, ist er geschlagen und wird vom Ban entfernt. Ein Block *lebt*, wenn der Gegner ihn unmöglich schlagen kann.

Definition 6 eigenes und gemeinsames Auge

Ein *eigenes Auge* ist eine Freiheit eines Blocks, die von diesem Block umgeben ist. (zur Abkürzung: eigA)

Ein *gemeinsames Auge* ist eine Freiheit mehrerer Blöcke gleicher Farbe, die von diesen Blöcken umgeben ist. An einem gemeinsamen Auge können 2 bis 4 Blöcke beteiligt sein. (zur Abkürzung: gemA)

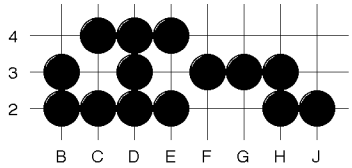


Bild 2 Zwei Blöcke, verbunden über ein gemeinsames Auge

In Bild 2 sind zwei Blöcke abgebildet, von denen der linke Block ein Auge besitzt (auf C3). Der rechte Block hat kein eigenes Auge. Zusammen formen beide Blöcke aber ein gemeinsames Auge auf E3.

1.4 Das Problem

Weiß darf in Bild 3 wegen des Selbstmord-Verbots keinen Stein auf C3 und keinen auf E3 stellen. Er kann aber alle Felder um den schwarzen Block herum besetzen. Da Weiß danach mit einem Zug aber nur einen Stein setzen kann, ist es ihm nicht möglich, beide Augen gleichzeitig zu besetzten und damit den schwarzen Block zu schlagen. Der schwarze Block lebt.

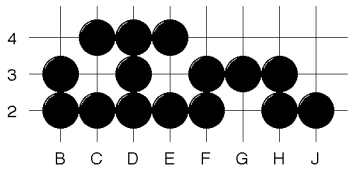


Bild 3 Ein Block mit zwei Augen

Auch in Bild 2 darf Weiß keine Steine auf C3 und E3 stellen. Weiß kann aber dem rechten Block alle Freiheiten bis auf E3 nehmen und ihn dann auf E3 schlagen.

Obwohl sich Bild 3 nur in einem Stein von der Situation in Bild 2 unterscheidet, sind die Blöcke völlig anders zu bewerten. Gesucht werden Kriterien, die die beiden Situationen aus Bild 2 und Bild 3 unterscheiden und die über diesen Spezialfall hinaus auch im allgemeinen Gruppen von Blöcken bewerten können.

2 Ketten

Dieser Abschnitt behandelt einen vereinfachten Fall bei der Beantwortung der Frage, ob Blöcke leben. Jedes gemeinsame Auge darf nur zwei Blöcke miteinander verbinden, und jeder Block darf höchstens 2 gemeinsame Augen haben.

Definition 7 Linearität

Ein Block heißt *linear*, wenn er höchstens 2 gemeinsame Augen hat; ein gemeinsames Auge heißt *linear*, wenn es genau 2 Blöcke miteinander verbindet.

Definition 8 Kette

Mehrere lineare Blöcke, die durch lineare gemeinsame Augen miteinander verbunden sind, bilden eine *Kette*. Ein einzelner Block bildet auch schon eine Kette.

Definition 9 offene und geschlossene Ketten

Eine Kette, bei der jeder Block genau 2 gemeinsame Augen hat, heißt *geschlossen*. Sonst heißt die Kette *offen*.

Eine geschlossene Kette bildet einen Kreis.

2.1 Grammatik für lebende offene Ketten

In diesem Abschnitt wird eine Abbildung von Ketten in Zeichenketten eingeführt. So lassen sich die Bedingungen für lebende Ketten durch formale Sprachen ausdrücken.

Alle lebenden offenen Ketten lassen sich durch folgende Grammatik generieren:

$$G_1 = < V_1, T_1, P_1, S >$$

Die Grammatik besteht aus dem Vokabular V_1, den Terminalsymbolen T_1, den Produktionsregeln P_1 und einem Startsymbol S. Die Terminalsymbole sind 0, 1 und 2.

$$T_1 = \{ 0, 1, 2 \}$$

Die 0 steht dabei für einen Block ohne Auge, die 1 für einen Block mit einem Auge, und die 2 steht für einen Block mit mindestens zwei Augen.

Das Produkt zweier Terminalsymbole symbolisiert zwei Blöcke, die über ein gemeinsames Auge miteinander verbunden sind. So lassen sich lange Ketten aus Blöcken bilden. Durch die Terminalsymbole kann z.B. die Konstellation der beiden Blöcke aus Bild 2 auf Seite 11 als das Wort 10 dargestellt werden.

Die Abbildung von Blöcken in Worte aus den Terminalsymbolen ist nicht eindeutig, da auch die Umkehrung der Worte die gleiche Situation auf dem Go-Brett beschreibt. D.h., es gibt keinen qualitativen Unterschied zwischen 10 und 01. Eine Kette kann aber auch deutlich mehr als zwei Blöcke umfassen, z.B 2100021110 bzw. 0111200012.

Damit alle Blöcke in einer Kette leben, müssen zwei Bedingungen erfüllt sein:

1. Besteht die Kette nur aus einem Block, so muß er zwei eigene Augen haben.
2. Der erste und der letzte Block brauchen mindestens ein eigenes Auge.

Der rechte Block in Bild 2 hat kein eigenes Auge. Er kann von Weiß geschlagen werden. Mit einem eigenen Auge hätte er zwei unbesetztbare Freiheiten und würde somit leben.

Für die Grammatik G_1 werden noch die Nonterminale A und B benötigt:

$V_1 = \{ S, A, B, 0, 1, 2 \}$

Die Produktionsregeln P_1 erfüllen die beiden Bedingungen.

P_1:

S	$\rightarrow$ 2 | 1A | 2A	2 oder nicht mit 0 beginnend …
A	$\rightarrow$ 0A | 1A | 2A | B	… beliebig viele Blöcke, aber am Ende …
B	$\rightarrow$ 1 | 2	… einen Block mit mindestens einem Auge

G_1 ist eine Typ-3 Grammatik, die generierte Sprache ist regulär.

2.2 Grammatik für lebende geschlossene Ketten

Zur Darstellung geschlossener Ketten wird obiger Formalismus erweitert.

$T_2 = \{ 2, 1, 0, (,) \}$

Eine eingeklammerte Zeichenkette aus Ziffern bedeutet, daß der erste und letzte Block der Kette über ein gemeinsames Auge miteinander verbunden sind. Die Kette wird dadurch geschlossen. In Bild 4 ist als Beispiel die geschlossene Kette (100) dargestellt.

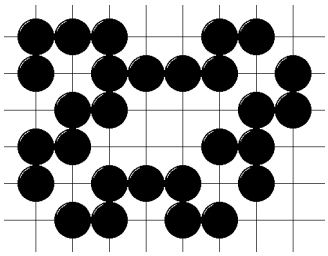


Bild 4 Drei gemeinsame Augen verbinden drei Blöcke zu einer geschlossenen Kette

Alle Blöcke einer geschlossenen Ketten leben, egal wieviele eigenen Augen die Blöcke haben, da jeder Block ein gemeinsames Auge mit seinem Vorgänger und eins mit seinem Nachfolger hat. Die Grammatik G_2 beschreibt lebende geschlossene Ketten:

$G_2 = < V_2, T_2, P_2, S >$

$T_2 = \{ (,), 2, 1, 0 \}$

$V_2 = \{ S, A, B \} \cup T_2$

P_2:

S	$\rightarrow$	(0A | (1A | (2A	geschl. Ketten enthalten mind. zwei Blöcke
A	$\rightarrow$	0A | 1A | 2A | B	beliebig viele 0, 1, 2
B	$\rightarrow$	0) | 1) | 2)	ein beliebiger Block

Auch G_2 ist eine Typ-3 Grammatik.

G_1 generiert offene, aber keine geschlossnen Ketten. G_2 generiert ausschließlich geschlossene Ketten. Leider ist es nicht möglich, eine geschlossene Kette mit einer offenen Kette zu verbinden. Dazu müßte die Linearität aufgegeben werden; d.h., gemeinsame Augen dürften dann mehr als zwei Blöcke haben und Blöcke dürften mehr als zwei gemeinsame Augen haben.

3 Go-Netze

In diesem Abschnitt werden die bisherigen Beschränkungen für die Zusammenstellung von Blöcken und gemeinsamen Augen fallengelassen. Trotzdem werden diese Netze noch auf Lebendigkeit überprüfbar sein.

Definition 10 Go-Netz

Eine *Go-Netz* N ist ein Paar (B,A) aus einer Blockmenge B und einer Menge gemeinsamer Augen A. Die Blöcke sind über die gemeinsamen Augen miteinander verbunden.

Definition 11 vitales gemeinsames Auge

Ein gemeinsames Auge ist *vital*, wenn für alle an dem gemeinsamen Auge beteiligten Blöcke gilt, daß sie

i) mindestens ein Auge haben oder

ii) weitere gemeinsame Augen haben, von denen mindestens eines vital ist.

Andernfalls ist das gemeiname Auge *nicht vital*.

Die Definition 11 ist rekursiv.

Definition 12 effektives Auge

Eigene Augen und vitale gemeiname Augen sind *effektive Augen* des betroffenen Blocks. Zur Abkürzung: eff.Auge oder effA

3.1 Beschreibungsformalismus

Go-Netze können als Petri-Netze dargestellt werden. Dabei sind hier nur die statischen Aspekte der Petri-Netze von Bedeutung. Die Blöcke sind die Stellen der Petri-Netze. Sie werden mit einem Namen des Blocks und der Zahl seiner eigenen Augen beschriftet. 2 bedeutet wieder, wie in der Grammatik G_1, daß der Block mindestens zwei eigene Augen besitzt. Da die Zahl der gemeinsamen Augen, die ein Block haben kann, nur durch die Größe des Spielbretts beschränkt ist, können beliebig viele Kanten von einem Block-Knoten abgehen. Die gemeinsamen Augen sind die Transitionen des Netzes. Sind zwei Blöcke über ein gemeinsames Auge miteinander verbunden, werden auch die beiden Block-Knoten mit demselben gemeinsamen-Augen-Knoten verbunden. Da die Verbindungen zwischen Blöcken über gemeinsame Augen nicht gerichtet sind, muß zu einer Kante (s,t) im Petri-Netz auch die Kante (t,s) existieren und umgekehrt. Die Zahl der von einer Transition ausgehenden Kanten liegt, bedingt durch die Topologie des Go-Bretts, zwischen 2 und 4.

Z.B. kann die schwarze Stellung aus Bild 5, die aus 7 Blöcken besteht, durch das Go-Netz aus Bild 6 dargestellt werden.

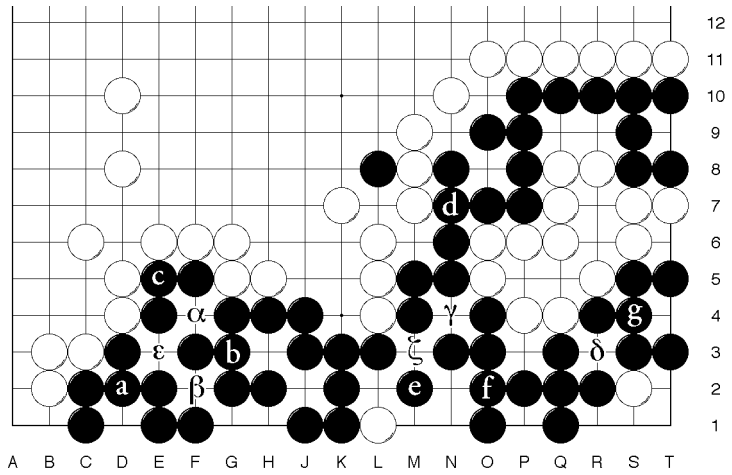


Bild 5 Schwarze Stellung mit den Blöcken a bis g und den gemeinsamen Augen α bis ζ

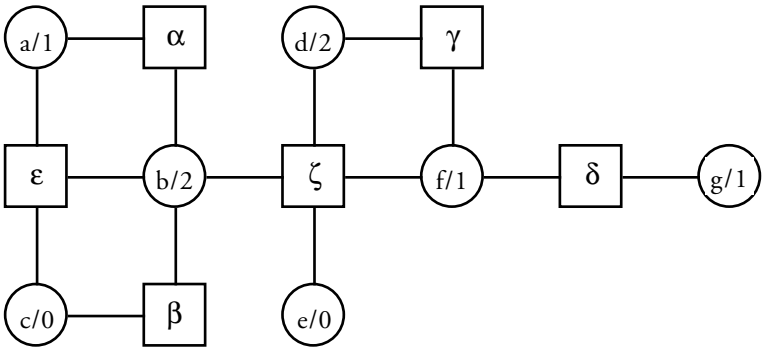


Bild 6 Go-Netz für die schwarze Stellung aus Bild 5

Eine textuelle Beschreibung eines Go-Netzes besteht aus einer Liste aller beteiligten Blöcke und einer Liste aller gemeinsamen Augen des Go-Netzes. Das Prädikat für die Blöcke block/3 hat die Komponenten

i) Name des Blocks (als kleiner lateinischer Buchstabe),
ii) Zahl der eigenen Augen und
iii) Zahl der gemeinsamen Augen.

Das Prädikat für die gemeinsamen Augen gemA/3 besitzt als Werte

i) den Namen des gemeinsamen Auges (als kleinen griechischen Buchstaben),
ii) den Typ des gemeinsamen Auges und
iii) eine Liste der Namen der Blöcke, die das gemeinsame Auge miteinander verbindet.

Für das Go-Netz in Bild 6 ergibt sich:

```
block( a,1,2 ), block( b,2,4 ), block( c,0,2 ), block( d,2,2 ),
block( e,0,1 ), block( f,1,3 ), block( g,1,1 )
gemA( α,2,[a,b] ), gemA( β,2,[b,c] ), gemA( γ,2,[d,f] ),
gemA( δ,2,[f,g] ), gemA( ε,3,[a,b,c] ), gemA( ζ,4,[b,d,e,f] )
```

3.2 Algorithmus für lebende Blöcke

In diesem Abschnitt wird ein Algorithmus vorgestellt, mit dem die Blöcke eines Go-Netzes auf Leben überprüft werden können. Ein gemeinsames Auge ist vital, wenn alle daran beteiligten Blöcke mindestens ein effektives Auge haben. Nur durch effektive gemeinsame Augen können sich Blöcke aneinander festhalten. Die anderen sind noch vom Gegner zu erobern. Blöcke leben, wenn sie nach Beendigung des Algorithmus mindestens zwei effektive Augen haben. Die übrigen Blöcke leben nicht.

Die Notation, in der der Algorithmus verfaßt ist, lehnt sich an Pascal, die Prädikatenlogik und die Mengenlehre an. Dieses Sprachkondensat wird benutzt, um das Verfahren konzentriert festzuhalten. Es folgt eine Erläuterung, in der auf die vergebenen Marken Bezug genommen wird.

aBlock ist ein Feld, das alle Blöcke enthält. Die Zahl der Augen (iAuge) und die Zahl der gemeinsamen Augen (iGemA) eines Blocks sind für alle Blöcke vordefiniert. Der Algorithmus berechnet für jeden Block die Zahl seiner effektiven Augen.

Das Feld aGemA enthält die gemeinsamen Augen. Jedes gemeinsame Auge kann einen der Werte vital-λ, vital-YES und vital-NO annehmen.

◊A – Für alle Blöcke wird zunächst die Zahl ihrer effektiven Augen mit der Zahl ihrer eigenen Augen initialisiert, da die eigenen Augen als effektive Augen zählen.

◊B – Dann werden alle gemeinsamen Augen untersucht. ◊C – Wenn alle Blöcke, die an dem untersuchten gemeinsamen Auge liegen, mindestens ein effektives Auge haben, ist das gemeinsame Auge vital. ◊D – Es zählt ab jetzt als effektives Auge für seine Blökke. Andernfalls bekommt das gemeinsame Auge den Wert vital-λ zugewiesen, der ausdrückt, daß das gemeinsame Auge weiter untersucht werden muß.

◊E – Für alle gemeinsamen Augen, die den Wert vital-λ haben, wird die Funktion PrüfeVerbindung aufgerufen. Der zweite Parameter für einen Block bleibt an dieser Stelle unbestimmt.

◊F – PrüfeVerbindung erhält ein gemeinsames Auge α und einen Block A als Argumente. Für das gemeinsame Auge wird getestet, ob es über seine Nachbarblöcke vital ist. Der Block A wird dabei nicht berücksichtigt.

```
     Eingabe     aBlock:    Definitionen der Blöcke
                            (Zahl der Augen, Zahl der gemA)
                 agemA :    Definitionen der gem.Augen
                            (Liste der verbundenen Blöcke)

     TYPE        TBlock:    RECORD
                                iAuge, iGemA, iEffA: INTEGER;
                            END;
                 TgemA :    ( vital-λ, vital-YES, vital-NO );

     VAR         aBlock:    ARRAY[ 0.. ] OF TBlock;
                 agemA :    ARRAY[ 0.. ] OF TgemA;
```

```
     BEGIN
◊ A        ∀ ( B:= aBlock[i] ) DO
                B.iEffA:= B.iAuge

◊ B        ∀ ( α:= aGemA[i] ) DO
◊ C             IF ( ∀ Blöcke B an α : B.iEffA > 0 ) THEN
                     α:= vital-YES
◊ D                  ∀ ( Blöcke B an α ) DO
                         B.iEffA:= B.iEffA + 1
                ELSE
                     α:= vital-λ

◊ E        ∀ ( α:= aGemA[i] | α = vital-λ ) DO
                PrüfeVerbindung( α, _ )
     END


◊ F  PROCEDURE PrüfeVerbindung( α: TgemA, A: TBlock ): BOOLEAN;
     BEGIN
◊ G        M:= { Blöcke B |
                     B liegt an α AND
                     B.iEffA = 0 AND
◊ H                  B ≠ A
               }

◊ I        IF ( (M = {}) OR
◊ J             (( ∀ B ∈ M: B.iGemA > 1 ) AND
◊ K              ( ∀ B ∈ M  ∃ gemA β von B : β ≠ α:
◊ L                  PrüfeVerbindung( β, [B]) )))
              )
◊ M             α:= vital-YES
                ∀ ( Blöcke B an α ) DO
                     B.iEffA:= B.iEffA + 1
                RETURN TRUE
           ELSE
◊ N             α:= vital-NO
                RETURN FALSE
     END PrüfeVerbindung
```

Programm 1 Berechnung lebender Blöcke

Definition 13 armer Nachbar

Ein Block, der keine effektiven Augen, aber ein gemeinsames Auge hat, heißt *armer Nachbar* des gemeinsamen Auges.

◊G – Es wird die Menge der armen Nachbarn des zu untersuchenden gemeinsamen Auges α erstellt. ◊H – Der Block A wird nicht in die Menge der armen Nachbarn aufgenommen.

◊I – Das untersuchte gemeinsame Auge α ist vital, wenn eine der beiden folgenden Bedingungen erfüllt ist. Entweder gibt es keine armen Nachbarn oder – ◊J – alle armen Nachbarn haben weitere gemeinsame Augen außer α – ◊K – und für jeden armen Nachbarn existiert ein gemeinsames Auge außer α – ◊L –, für das die Funktion PrüfeVerbindung TRUE liefert. Der Funktion wird auch der arme Nachbar übergeben, um Rückkopplungen zu vermeiden.

◊M – Ein vitales gemeinsames Auge zählt als effektives Auge für seine Blöcke. Die Funktion wird mit TRUE verlassen.

◊N – Ist keine der beiden Bedingungen unter Marke ◊I erfüllt, so ist das gemeinsame Auge α nicht vital. Die Funktion wird mit FALSE beendet.

Die Funktion PrüfeVerbindung testet für ein gemeinsames Auge, ob es vital ist oder nicht. Dabei kann es nötig werden, für weitere gemeinsame Augen PrüfeVerbindung aufzurufen. Die Funktion ist also rekursiv.

3.3 Der Algorithmus angewendet auf Ketten

Ketten sind spezielle Go-Netze. In diesem Abschnitt wird nachgeprüft, ob der Algorithmus für Ketten dieselben Ergebnisse liefert, wie die beiden Grammatiken G_1 für lebende offene Ketten und G_2 für lebende geschlossene Ketten aus Kapitel 2 . Es wird sich zeigen, daß der Algorithmus für geschlossene Ketten modifiziert werden muß.

3.3.1 Der Algorithmus angewendet auf offene Ketten

Zu zeigen ist:

i) Alle Blöcke jeder Kette, die von der Grammatik G_1 erzeugt werden können, werden auch von dem Algorithmus als lebend erkannt.
ii) In jeder nicht lebenden Kette wird von dem Algorithmus ein Block gefunden, der weniger als zwei effektive Augen hat.

Beweis

i) und ii) für Ketten mit einem Block:
Bei Marke ◊A wird die Zahl der effektiven Augen auf die Zahl der realen Augen gesetzt. Der Algorithmus ist damit beendet, da es keine gemeinsamen Augen gibt.

Blöcke die mindestens 2 Augen haben leben, Blöcke mit weniger Augen leben nicht. Für Ketten, die aus einem Block bestehen, liefern also der Algorithmus und die Grammatik G_1 dasselbe Resultat.

i) für Ketten mit mehr als einem Block:
Für jeden Block einer Kette gilt, daß er entweder mindestens ein Auge hat und ein gemeinsames Auge oder, daß er zwei gemeinsame Augen hat. Wenn alle gemeinsamen Augen vital sind, leben alle Blöcke der Kette. Die Funktion PrüfeVerbindung untersucht, ob ein gemeinsames Auge vital ist. Wegen der Linearität gilt, daß jedes gemeinsame Auge φ zwei Blöcke x und y verbindet. Jeder dieser beiden Blöcke kann mindestens ein eigenes Auge haben, oder er kann – wenn er kein eigenes Auge hat – ein weiteres gemeinsames Auge ψ haben. Über dieses Auge ψ ist der Block mit einem nächsten Block z verbunden. Da die beiden Blöcke am Anfang und Ende der Kette mindestens ein Auge haben, sind alle gemeinsamen Augen vital.

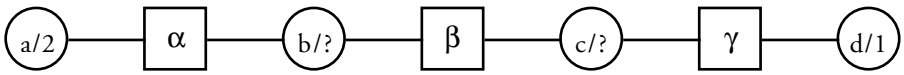


Bild 7 eine lebende Kette in Go-Netz-Darstellung

ii) für Ketten mit mehr als einem Block:
Sei K eine offene Kette mit mindestens 2 Blöcken, die nicht von der Grammatik G_1 erzeugt werden kann. Dann gibt es einen Block a am Ende der Kette, der kein Auge besitzt. Das gemeinsame Auge α neben diesem Block hat also den armen Nachbarn a, der keine weiteren gemeinsamen Augen besitzt. Damit ist das gemeinsame Auge α nicht vital und der Block a kann keine effektiven Augen erhalten.

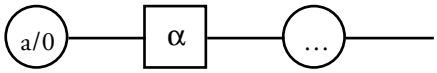
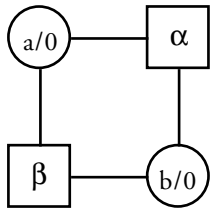


Bild 8 Anfang einer nicht-lebende Kette mit mehreren Blöcken

3.3.2 Der Algorithmus angewendet auf geschlossene Ketten

Schon der einfachste Fall einer geschlossenen Kette – (00) – überfordert den Algorithmus.



	Augen	gemA	effA
a	0	2	0
b	0	2	0

Bild 9 Go-Netz für zwei Blöcke, die über zwei gemeinsame Augen miteinander verbunden sind

Weder das gemeinsame Auge α noch das gemeinsame Auge β werden in den ersten Schritten des Verfahrens als vital erkannt, da kein Block effektive Augen hat. Deshalb wird zuerst (Œ) PrüfeVerbindung(a, _) aufgerufen.[2] Die armen Nachbarn von α sind die Blöcke a und b. Beide haben noch andere gemeinsame Augen, nämlich a hat β und b hat β. Es muß geprüft werden, ob diese vital sind: PrüfeVerbindung(β, a). Der arme Nachbar von β ist b und der hat noch gemeinsame Augen, nämlich α. Œ PrüfeVerbindung(α, b), Œ . . . . . etc.

Um diese Endlosschleife zu vermeiden, werden der Funktion PrüfeVerbindung alle Blöcke übergeben, die schon auf dem bisherigen Untersuchungsweg lagen. Der Algorithmus ändert sich bei den Marken ◊F, ◊H und ◊L.

```
# F   PROCEDURE PrüfeVerbindung( α: TgemA, L: Menge aus TBlock´s ): BOOLEAN;
      BEGIN
◊ G         M:= { Blöcke B |
                        B liegt an α AND
                        B.iEffA = 0 AND
# H                     B ∉ L
                  }

◊ I         IF ( (M = {}) OR
◊ J                (( ∀ B ∈ M: B.iGemA > 1 ) AND
◊ K                 ( ∀ B ∈ M  ∃ gemA β von B : β ≠ α:
# L                       PrüfeVerbindung( β, L ∪ {B} )))
               )
◊ M               α:= vital-YES
                  ∀ ( Blöcke B an α ) DO
                        B.iEffA:= B.iEffA + 1
                  RETURN TRUE
            ELSE
◊ N               α:= vital-NO
                  RETURN FALSE
      END PrüfeVerbindung
```

Programm 2 Modifikation der Funktion PrüfeVerbindung

[2] PrüfeVerbindung(b, _) führt zu dem gleichen Resultat.

3.4 Der Algorithmus angewendet auf Go-Netze

Hier wird gezeigt, daß alle Blöcke, die von dem Algorithmus als lebend erkannt werden, auch in Bensons Formalismus leben [Benson 1976]. Dazu wird zunächst ein Einblick in Bensons Konzepte gegeben.

3.4.1 Bensons Formalismus – Unconditional Life

Definition 14 small x-enclosed region

Eine *small x-enclosed region* R ist eine Menge zusammenhängender Felder, für die gilt:

(1) Kein Stein der Farbe x darf in R liegen, $x \notin a(R)$. [3])

(2) R ist von Steinen der Farbe x umgeben, $a(Ext(R)) = \{x\}$. [4])

(3) Jedes Feld der Region hat einen Nachbarstein der Farbe x
oder es ist von einem Stein der Farbe $\bar{x}$ besetzt, $a(Int(R)) \subseteq \{\bar{x}\}$. [5])

Definition 15 healthy region

Eine Region R ist *healthy* für einen Block b, H(R, b), wenn R eine small x-enclosed region ist und wenn jedes unbesetzte Feld von R eine Freiheit von b ist.

Definition 16 NB

NB(R) ist die Menge der Nachbarblöcke von R.

Definition 17 vital region

Sei X eine Menge von Blöcken der Farbe x mit $b \in X$. Eine Region R ist $vital_{Benson}$ für den Block b, V(R, b, X), wenn

(1) R healthy für b ist, H(R, b), und

(2) alle Nachbarblöcke von R in X enthalten sind, $NB(R) \subseteq X$

Definition 18 Unconditional Life

Sei X eine Menge von Blöcken der Farbe x. Wenn alle Blöcke aus X zwei unabhängige $vitale_{Benson}$ Regionen haben, so ist X *unconditionally alive*,

$\forall\, b \in X\ \exists\, R_1, R_2, R_1 \neq R_2 :\ V(R_i, b, X), i= 1,2$

Benson führt den Beweis, daß kein Block einer Menge X, die unconditionally alive ist, in irgendeiner Zugfolge vom Gegenspieler – bei passivem Spiel des x-Spielers – geschlagen werden kann.

[3] a ist ein Abbildung von der Menge der Felder in die Menge der Farben {black, white, empty}. Für eine spezielle Feldermenge liefert sie alle Farben der Steine, die auf den Feldern stehen.

[4] Ext ist eine Abbildung, die die Umgebung zu einer Mengen von Feldern liefert.

[5] Int ist eine Abbildung, die zu einer Region diejenigen Felder bestimmt, die keine Nachbarn der Umgebung sind.

3.4.2 Korrektheit des Algorithmus

Zum Beweis, daß alle Blöcke, die der Algorithmus als lebend erkennt, wirklich leben, wird gezeigt, daß die Menge der schwarzen lebende Blöcke und die Menge der weißen lebenden Blöcke unconditionally alive sind.

Beweis

Ein eigenes Auge ist ein unbesetztes Feld, das von einem Block umgeben wird. Damit ist es eine healthy Region für seinen Block, da die Bedingungen aus Definition 14 und Definition 15 erfüllt sind.

Ein gemeinsames Auge ist ein unbesetztes Feld, das von mindestens 2 Blöcken gleicher Farbe umgeben ist. Es ist eine small x-enclosed Region, und für jeden Block des gemeinsamen Auges ist es eine healthy Region.

Die Blöcke, die der Algorithmus als lebend erkannt hat, werden in zwei Mengen eingeteilt. X_B ist die Menge der lebenden schwarzen Blöcke und X_W ist die Menge der lebenden weißen Blöcke. Œ wird im folgenden nur mit der Menge X_B argumentiert, da der Beweisschluß für X_W analog verläuft.

Sei b ein lebender schwarzer Block, $b \in X_B$. Wenn b ein eigenes Auge hat, dann ist es eine $\text{vitale}_{\text{Benson}}$ Region für den Block b, denn es ist healthy, und alle Nachbarblöcke des Auges, nämlich nur b, sind in X_B enthalten. Wenn b ein vitales gemeinsames Auge α hat, dann ist auch das eine $\text{vitale}_{\text{Benson}}$ Region für den Block b. α ist healthy, und alle Nachbarblöcke von α sind in X_B. Angenommen das vitale gemeinsame Auge hätte einen Nachbarblock c, der nicht in X_B läge, $c \notin X_B$. Der Block c wäre nicht lebendig. Dann könnte aber auch das gemeinsame Auge nicht vital sein.

Eigene Augen und vitale gemeinsame Augen sind effektive Augen für einen Block. Jeder Block aus X_B hat mindestens 2 effektive Augen. Da sowohl eigene Augen als auch vitale gemeinsame Augen $\text{vitale}_{\text{Benson}}$ Regionen für ihre Nachbarblöcke sind, haben alle vom Algorithmus als lebend erkannten Blöcke 2 unabhängige $\text{vitale}_{\text{Benson}}$ Regionen. D.h., die Menge X_B ist unconditionally alive.

3.5 Prototyp

Im Rahmen dieser Arbeit wurde ein funktionaler Prototyp implementiert. Die verwendeten Programmiersprachen sind LPA-Prolog auf Apple-Macintosh und Quintus-Prolog auf VAX/VMS.

Das Listing des Prolog-Programms für den in Abschnitt 3.2 skizzierten Algorithmus ist als Anhang A dieser Arbeit angefügt. Als Beispiel ist dort das Go-Netz aus Bild 6 eingetragen (ab Zeile 40). Statt der Beschreibung des Go-Netzes über die beiden Prädikate

block/3 und gemA/2 wurden die Prädikate data_block/2 und data_gemA/2 verwendet, die nicht je einen Block bzw. ein gemeinsames Auge beschreiben, sondern die Beziehung zwischen Block, Zahl der eigenen Augen und Zahl der gemeinsamen Augen des Blocks bzw. zwischen einem gemeinsamen Auge und der Liste aller Blöcke an dem gemeinsamen Auge. Der Unterschied besteht darin, daß die data-Prädikate den Block bzw. das gemeinsame Auge als Argument nehmen, statt nur den Blocknamen und den Namen des gemeinsamen Auges.

Gestartet wird das Programm durch das Prädikat go/0 (Zeile 134). Zuerst wird dann das dynamische Prädikat effA/2 für jeden Block auf die Zahl der eigenen Augen gesetzt und das dynamische Prädikat modus_gemA/2 für jedes gemeinsame Auge auf lambda. Im go_vorlauf/0 (Zeile 161) werden dann alle gemeinsamen Augen, für die gilt, daß alle anliegenden Blöcke mindestenst ein effektives Auge haben, auf den Moduswert vital gesetzt. Immer, wenn ein gemeinsames Auge vital wird, wird auch den anliegenden Blöcken ein effektives Auge angerechnet (set_vital/1, Zeile 333). Das Prädikat go_hauptschleife/0 (Zeile 195) startet für alle gemeinsamen Augen mit dem Moduswert lambda pruefe_vital/2. Ob das Prädikat erfolgreich ist oder fehlschlägt ist an dieser Stelle ohne Bedeutung. Pruefe_vital/2 (Zeile 222) wird sofort erfolgreich beendet, falls das gemeinsame Auge schon vital ist[6]. Andernfalls berechnet das Prädikat die armen Nachbarn des gemeinsamen Auges. Es können drei Fälle auftreten.
1) Es gibt keine armen Nachbarblöcke, so wird das gemeinsame Auge vital gemacht und das Prädikat wird erfolgreich verlassen.
2) Es gibt arme Nachbarn, so werden für diese zusätzliche Bedingungen überprüft: Alle armen Nachbarn müssen weitere gemeinsame Augen haben (Zeile 283) und von diesen gemeinsamen Augen muß pro armer Nachbar mindestens eines vital sein, also erfolgt hier ein rekursiver Aufruf auf pruefe_vital/2 (ab Zeile 300). Das dritte Argument für pruefe_vital/3 ist die Liste mit der pruefe_vital/3 aufgerufen wurde plus des Blocks, der gerade untersucht wird. Dies entspricht der Änderung bei Marke #L aus Abschnitt 3.3.2. Ist diese komplexe Bedingung erfüllt, so wird, wie im ersten Fall, das gemeinsame Auge vital gemacht.
3) Im dritten Fall gibt es arme Nachbarn, die aber diese komplexe Forderung nicht erfüllen. Das gemeinsame Auge wird dann nonvital gemacht und das Prädikat pruefe_vital/3 schlägt fehl.
Wenn es kein gemeinsames Auge mit dem Moduswert lambda mehr gibt, ist die go_hauptschleife/0 beendet. Alle Blöcke mit 2 effektiven Augen leben. Die übrigen können noch vom Spielgegner geschlagen werden.[7]

[6] Das kann aber nur der Fall sein, wenn man von einem gemeinsamen Auge ausgehend durch Rekursion wieder zu diesem gemeinsamen Auge kommt. Mit diesem gemeinsamen Auge als Argument wird dann zweimal set_vital/1 aufgerufen, einmal am Ende der Rekursionskette und einmal zu Beginn.

Startet man das Programm mit den Daten des Go-Netzes aus Bild 6, so ergibt sich folgende Ausgabe:

```
 : go
go-init
lambdaGemA_[ga, gb, gc, gd, ge, gf]
lebende_bloecke_[kb, kd]
-----
go-vorlauf
[set_vital, ga]
[set_vital, gc]
[set_vital, gd]
vitaleGemA_[ga, gc, gd]
lambdaGemA_[gb, ge, gf]
lebende_bloecke_[kb, kd, ka, kf, kg]
-----
go-hauptschleife
[pruefe_vital, gb, []]
[pruefe_vital, ge, [kc]]
[set_vital, ge]
[set_vital, gb]
[pruefe_vital, gf, []]
[set_nonvital, gf]
vitaleGemA_[ga, gc, gd, ge, gb]
nichtvitaleGemA_[gf]
lebende_bloecke_[kb, kd, ka, kf, kg, kc]
-----
Yes
No more solutions
```

Es wird richtig erkannt, daß die Blöcke b, d, a, f, g und c leben. Block e hat nach Aufruf des Prädikats go/0 kein effektives Auge; d.h, Block e lebt nicht. Alle gemeinsamen Augen, bis auf gf=ζ, werden als vital erkannt. Auch das ist richtig.

Das Go-Netz aus Bild 9 wird so dargestellt:

```
data_block( ka,0,2 ).
data_block( kb,0,2 ).

data_gemA( ga, [ka,kb] ).
data_gemA( gb, [ka,kb] ).
```

Startet man das Programm mit diesen Daten, so erhält man folgendes Ergebnis:

```
 : go
go-init
lambdaGemA_[ga, gb]
lebende_bloecke_keine
-----
go-vorlauf
lambdaGemA_[ga, gb]
lebende_bloecke_keine
-----
go-hauptschleife
```

[7] Randbemerkung: Das Prädikat block/3 enthält an dritter Stelle nur die Anzahl der gemeinsamen Augen, nicht aber eine Referenz auf diese. Der Grund dafür liegt in dem eben vorgestellten Algorithmus. Es wird dort fast immer über die gemeinsamen Augen quantifiziert, die dann ihrerseits auf die beteiligten Blöcke verweisen. Einzige Ausnahme ist unter Marke ◊K im Algorithmus. Für arme Nachbarn (also für Blöcke) werden ihre gemeinsamen Augen untersucht. In der Prolog-Implementation ist das Prädikat bedingung2b/3 (ab Zeile 295 im Anhang A) gezwungen, für alle gemeinsamen Augen zu prüfen, ob sie gemeinsame Augen für einen bestimmten Block sind.

```
[pruefe_vital, ga, []]
[pruefe_vital, gb, [ka]]
[pruefe_vital, ga, [kb, ka]]
[set_vital, ga]
[set_vital, gb]
vitaleGemA_[ga, gb]
lebende_bloecke_[ka, kb]
-----
Yes
No more solutions
```

Beide Blöcke leben, beide gemeinsamen Augen sind vital und das Programm terminiert.

4 Augen

Hier wird die Frage behandelt, inwieweit die Begriffe des eigenen und gemeinsamen Auges aus den Kapiteln 1 bis 3 verallgemeinert werden können. Welche Kriterien müssen Augen erfüllen, so daß das vorgestellte Verfahren noch korrekt funktioniert? Daß eine genauere Untersuchung der Begriffe nötig ist, zeigt folgendes Beispiel:

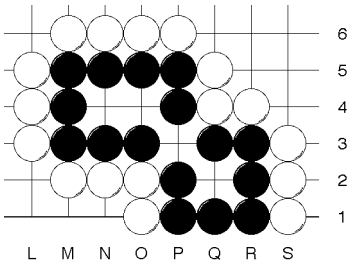


Bild 10 Zwei lebende schwarze Blöcke

Den beiden schwarzen Blöcken entspricht im bisherigen Formalismus 01, d.h., ein Block ohne eigenes Auge[8] verbunden über ein gemeinsames Auge auf P3 mit einem Block mit einem eigenen Auge. Der Algorithmus ermittelt dafür, daß das gemeinsame Auge nicht vital ist und daß die schwarzen Blöcke nicht leben. Aber trotzdem kann Weiß die Blöcke nicht erobern. Auf Q2 und P3 darf Weiß wegen des Selbstmord-Verbots keinen Stein setzen. Ein weißer Stein auf N4 ist möglich; damit wird aber der nächste Zug auf O4 wegen des Selbstmord-Verbots ausgeschlossen.

4.1 Begriffsbildung

Nach Definition 6 hat der schwarze Block aus Bild 11a ein eigenes Auge. Die 3 Blöcke aus Bild 11b haben ein gemeinsames Auge.

Der entscheidende Punkt ist, daß der Spielgegner alle seine Freiheiten verliert, wenn er das Auge besetzt. Die gleiche Eigenschaft hat aber auch das Auge in Bild 11c, ohne ein eigenes oder gemeinsames Auge zu sein.[9]

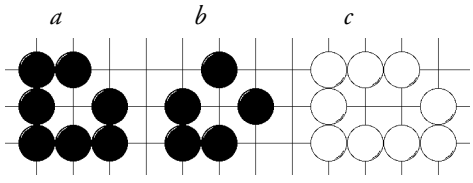


Bild 11 a) ein Block mit einem eigenen Augen, b) 3 Blöcke und ein gemeinsames Auge, c) ein Block mit…

[8] Das Auge ist im Sinne der Definition 6 kein eigenes Auge, da es zwei Felder groß ist.
[9] vgl. Bild 10 links-oben

In einem Einführungsbuch in das Go-Spiel von Richard Bozulich [Bozulich 1987] wird der Begriff „Auge“ folgendermaßen erklärt:

Definition 19 Auge (nach Bozulich)

Ein Auge ist ein leeres Feld, das von Steinen gleicher Farbe umgeben ist.

Ein *echtes Auge* ist ein Auge, das niemals zerstört werden kann, i.e., i) man kann niemals gezwungen werden das echte Auge selbst zu besetzen, und ii) der Spielgegner kann das Auge nur besetzen, wenn er alle an dem Auge beteiligten Blöcke auch schlägt.

Ein *falsches Auge* ist ein Auge, das kein echtes Auge ist.

Ein *großes Auge* ist eine Menge zusammenhängender Freiheiten, die von Blöcken einer Farbe umgeben sind.

Angenommen, daß in [Bozulich 1987] derselbe Umgebungsbegriff benutzt wird, wie er hier eingeführt wurde, so sind die Begriffe „eigenes Auge“ und „gemeinsames Auge“ aus Kapitel 1 Verfeinerungen von Bozulichs Augen-Begriff. Ein Auge ist entweder ein eigenes Auge oder ein gemeinsames Auge. Ein echtes Auge kann den Blöcken, die es umgeben, nicht genommen werden. Das trifft auf eigene Augen und auf vitale gemeinsame Augen zu. Ein nicht vitales gemeinsames Auge ist ein falsches Auge. Die Termini „echtes Auge“ und „falsches Auge“ sind also zunächst nur auf Augen der Größe 1 anwendbar.[10] Die Bemerkungen zu den großen Augen seien noch einen Moment aufgeschoben.

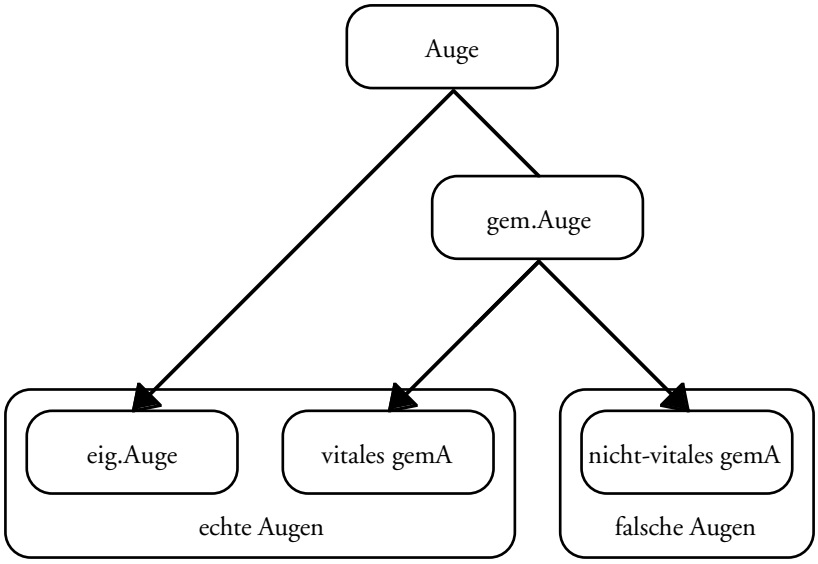


Bild 12 Augen der Größe 1

[10] Die *Größe* eines Auges ist die Zahl der Felder im Auge. Die Größe einer Region wird in Abschnitt 4.2 definiert.

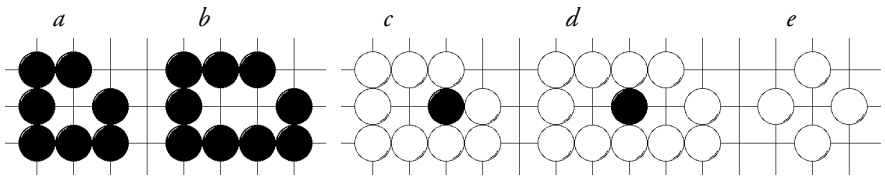


Bild 13 Blöcke mit Augen

Der Block in Bild 13a hat ein Auge, genau wie die Blöcke aus Bild 13e ein Auge haben. Der Block in Bild 13b hat ein großes Auge.

Die weißen Blöcke in Bild 13c und in Bild 13d haben kein Auge, nichteinmal ein großes, da die eingeschlossenen Freiheiten nicht vollständig von weißen Steinen umgeben sind.

Es stellt sich die Frage nach dem Unterschied zwischen einem großen Auge, das nur aus unbesetzten Feldern besteht, und einem ebensolchen großen Auge, in dem aber Steine des Spielgegners stehen. Hat der weiße Block in Bild 13d ein großes Auge der Größe 3 mit einem schwarzen Stein in der Mitte, oder ist es ein Block ohne Augen, wie es bis jetzt alle Definitionen beschreiben? Bei Bozulich gibt es in den Ausgangspositionen keine Steine des Spielgegners in großen Augen. Erst der Kampf entscheidet dann, ob aus dem großen Auge ein oder zwei echte Augen gemacht werden können. Die Position in Bild 13d wäre also als Kampf um ein großes Auge zu betrachten.

Definition 20 Auge

Ein *Auge* ist ein unbesetztes Feld, das von einem oder mehreren Blöcken einer Farbe umgeben ist.

Definition 21 großes Auge

Ein *großes Auge* ist eine Menge zusammenhängender Felder, für die gilt:

i) Die Felder sind leer oder von Steinen des Spielgegners besetzt.
ii) Die Felder sind von einem oder mehreren Blöcken einer Farbe umgeben.
iii) Die Felder enthalten mindestens eine Freiheit eines umgebenden Blocks.

Nach dieser Definition haben alle Blöcke aus Bild 13 ein Auge bzw. ein großes Auge.[11]

Es gibt bisher keinen einheitlichen Begriff von Augen und großen Augen. Häufig wird einfach beides als Auge bezeichnet.

Der Unterschied zwischen eigenen und gemeinsamen Augen läßt sich auch auf große Augen übertragen, so daß sich folgendes Schema ergibt:

[11] Die beiden schwarzen Steine aus Bild 13c und Bild 13d haben natürlich keine Augen; es sei denn, die weißen Blöcke wären ihrerseits außerhalb des weißen Auges von schwarzen Blöcken umgeben.

Größe / Blöcke	1	>1
1	eigenes Auge	großes eigenes Auge
>1	gemeinsames Auge	großes gemeinsames Auge

Tab. 1 Zusammenhang zwischen der Zahl der umliegenden Blöcke und der Zahl der Felder von Augen

Es sind jetzt große eigene Augen und große gemeinsame Augen eingeführt worden. Der Algorithmus braucht aber präzisere Begriffe für große Augen, um korrekte Ergebnisse zu liefern.

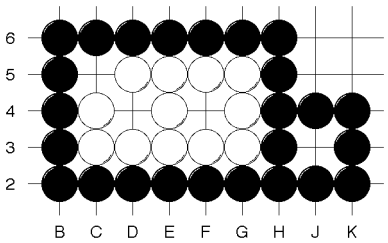


Bild 14 Schwarzer Block mit 2 eigenen Augen

Das Beispiel aus Bild 14 zeigt einen schwarzen Block mit 2 Augen, einem eigenen Auge auf J3 und einem großen eigenen Auge in dem ein weißer Block lebt! Das große Auge ist offensichtlich zu groß, als das es noch seine Funktion im Algorithmus übernehmen könnte. Es muß für große Augen also noch Einschränkungen geben. Diese werden in den nächsten beiden Abschnitten untersucht.

Abschließend sei noch die technisch formale Definition von Benson aus [Benson 1976] erwähnt:

Definition 22 Eye & Joiner (nach Benson)

Sei die Region R healthy für den Block b. Wenn die Zahl der Nachbarblöcke von R gleich 1 ist, dann ist R ein *Eye*. Sonst ist R ein *Joiner*.

4.2 Regionen, Formen und Typen

Dieser Abschnitt beschreibt die Formen, die zusammenhängende Mengen von Feldern bilden können. Es wird eine Typisierung der Formen vorgestellt, die speziell für die Beurteilung der Formen als Augen hilfreich ist.

Definition 23 Region

Eine zusammenhängende Menge von Feldern heißt *Region*.

Die *Größe* einer Region ist die Anzahl der Felder in der Region.

Definition 24 Form, Typ

Eine *Form* ist eine von Ort und Orientierung abstrahierte Region; die Form einer Region wird als rechtwinkliges Kantenmuster dargestellt.

Der *Typ* einer Form ist ein Quadrupel (a,b,c,d) $\in \mathbb{N}^4$, wobei a die Zahl der Felder ist, die 4 Nachbarn innerhalb der Form haben. b ist die Zahl der Felder mit 3 Nachbarn, c die Zahl der Felder mit 2 Nachbarn und d die Zahl der Felder mit einem Nachbarn in der Form. Falls a,b,c und d alle einstellig sind, kann statt (a,b,c,d) auch Typ abcd geschrieben werden.

Beispielsweise belegt das Auge in Bild 15 die Region {(a,2),(b,2),(b,1)}. Die Form des Auges ist daneben abgebildet. Der Typ des Auges ist 0012.

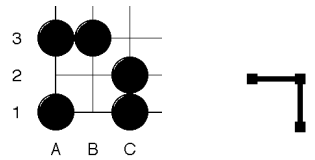


Bild 15 Region und Form

Es folgt eine Aufstellung aller möglichen Formen bis zur Größe 6. Zusätzlich werden sie nach ihren Typen klassifiziert.

Größe 1: Ein einzelnes Feld ist die einzige Form der Größe 1. Der Typ dieses Feldes ist 0000.

Größe 2: Es gibt auch nur eine Form aus 2 Feldern. Der Typ ist 0002, weil es 2 Felder gibt, die einen Nachbarn in der Form haben.

Größe 3: Hier gibt es 2 Formen, die aber beide den gleichen Typ haben, nämlich 3 Felder in einer Reihe und 3 Felder im Dreieck, beide mit Typ 0012.

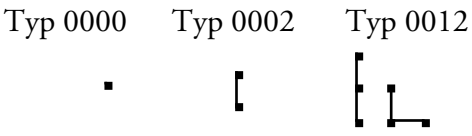


Bild 16 Formen der Größe 1 bis 3 und ihre Typen

Größe 4: Es gibt 5 Formen der Größe 4, die sich in 3 Typen aufteilen. Typ 0022 enthält 4 Felder in einer Reihe, 4 Felder in Gestalt eines „L“ und einen Zick-Zack. 0103 ist der Typ für 4 Felder in Gestalt eines „T“ und das Quadrat hat den Typ 0040.

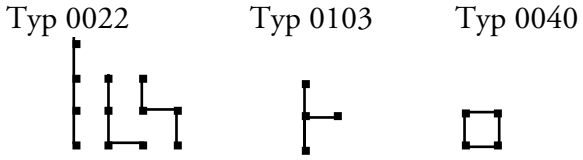


Bild 17 Formen der Größe 4 und ihre Typen

Größe 5: Es gibt 12 Formen in 4 Typen

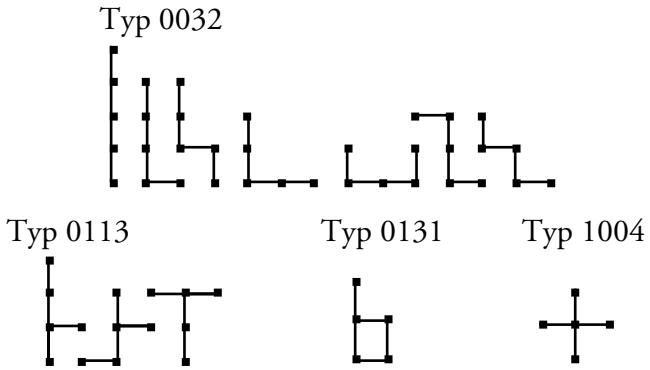
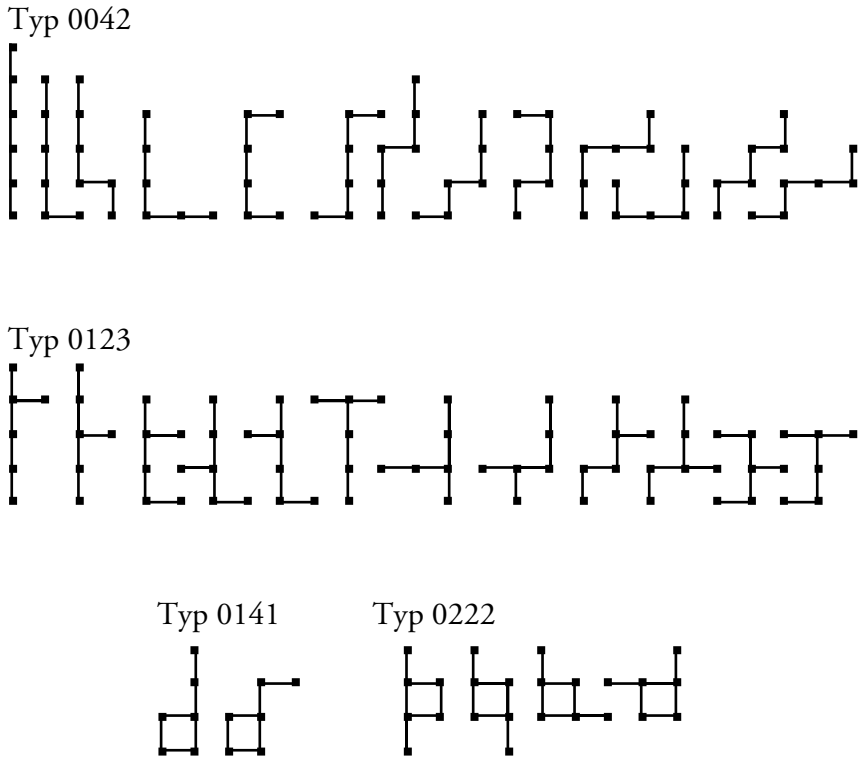


Bild 18 Formen der Größe 5 und ihre Typen

Größe 6: Es gibt 35 Formen, die sich in 8 Typen einteilen.



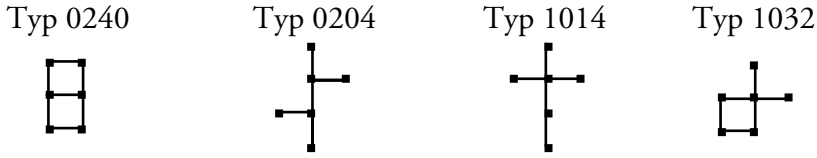


Bild 19 Formen der Größe 6 und ihre Typen

Daß diese Aufstellung vollständig ist, mag der Vergleich mit [Conway 1976, p120] verdeutlichen. Conway hat alle Domino-Positionen mit höchstens 6 Knoten zusammengestellt und spieltheoretisch bewertet.

Mit Ausnahme der Form der Größe 1, ist die Quersumme des Typs mit des Größe der Form identisch. Der Typ einer Form ist die nochmalige Abstraktion von Regionen des Go-Bretts.

4.3 Leben und Tod

Die Formen werden jetzt als große eigene Augen von Blöcken interpretiert. Als Randbedingung wird gefordert, daß das Auge nicht am Spielfeldrand liegt.

Dieser Abschnitt untersucht die Frage, wieviele effektive Augen ein großes eigenen Auge für den Block wert ist. Sicherlich hat ein Block mit einem großen Auge ein effektives Auge. Kann er aber auch sicher zwei haben? Mit dieser Frage wird der Bereich der statischen Bewertung der Blöcke verlassen, da Verteidigungszüge mit in die Betrachtung aufgenommen werden.

Unter Umständen ist es entscheidend, wer am Zug ist. Ist das der Angreifer, also der Spieler, der das Ziel hat den Block mit dem Auge zu schlagen, so wird das durch ein nachgestelltes „↑“ markiert. Der Verteidiger ist derjenige, dem der Block und das Auge gehören. Ist der Verteidiger am Zug, so markiert das ein „↓“.

0012↑

bezeichnet die Situation, daß ein Block ein großes eigenes Auge mit dem Typ 0012 besitzt und das der Gegner des Blocks am Zug ist.

Ein „K“ wie „Klingel“ bedeutet, daß der Verteidiger erst dann reagieren muß, wenn der Spielgegner in das Auge hineingesetzt hat. Geschieht dies nicht, so geht die Eigenschaft, die jeweils links vom „K“ steht verloren. Ist das „K“ eingeklammert, so ist es nicht nötig sofort zu reagieren.

2 eigA | K

bedeutet, daß der Block 2 eigene Augen sicher hat – unter der Bedingung, daß er den richtigen Verteidigungszug macht, wenn das Auge angegriffen wird.

Größe	Typ	Entsprechung
1	0000	1 eigA
2	0002	1 eigA
3	0012↓	2 eigA
	0012↑	1 eigA
4	0022	2 eigA | K
	0103↓	3 eigA
	0103↑	1 eigA
	0040	1 eigA
5	0032	2 eigA | (K)
	0113	2 eigA | (K)
	1004↓	4 eigA
	1004↑	1 eigA
	0131↓	2 eigA
	0131↑	1 eigA
6	0042	2 eigA | (K)
	0123	2 eigA | (K)
	0222	2 eigA | (K)
	0141	2 eigA | (K)
	0204	2 eigA | (K)
	1014	2 eigA | (K)
	0240↓	0032↑
	0240↑	2 eigA | K
	1032↑	1 eigA
	1032↓	2 eigA | 0012
		Typ 0113

Tab. 2 Auflistung der großen eigenen Augen und ihrer Bedeutung für den Algorithmus

Ein Beispiel soll zeigen, welche Bedeutung große eigene Augen für einen Block haben können.

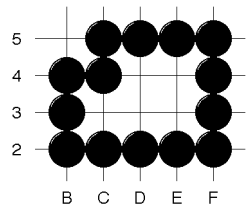


Bild 20 Block mit einem großen eigenen Auge des Typs 0131

Der schwarze Block in Bild 20 hat ein eigenes großes Auge. Für ein Auge mit dem Typ 0131 gilt laut Tab. 2 die Entspechung 2 eigA wenn Schwarz am Zug ist und 1 eigA, wenn Weiß am Zug ist. Darf Schwarz zuerst einen Stein setzen, so wählt er D3. Damit hat er sein großes Auge in ein eigenes Augen und in ein großes Auge vom Typ 0012 geteilt. Das neu entstande große Auge entspricht aber mindestens einem eigenen Auge. Diese beiden effektiven Augen reichen dem Block gemäß des Algorithmus zum Leben. Wenn Weiß am Zug ist, setzt er einen Stein auf D3. Damit sind die Chancen für Schwarz verbaut, zwei Augen zu bekommen.

Wenn ein Block ein großes eigenes Auge hat, können ihm zu Beginn des Algorithmus der Auflistung entsprechend virtuelle eigene Augen zugerechnet werden, die sich genau wie die bisherigen eigenen Augen auf das Leben des Blocks und seiner Nachbarn auswirken.

Die Bedeutung, die große gemeinsame Augen für ihre Blöcke haben, wird hier nicht betrachtet.

5 Résumé

Die vorgestellten Verfahren sind alle formaler Art. Sie haben nicht den Anspruch, in irgendeiner Weise kognitiv adäquat zu sein oder ein Modell für das menschliche Denken beim Go-Spiel zu liefern.

Es ist möglich, den Algorithmus als Basis eines Agenten innerhalb eines Go-Programms einzusetzen, der Zugvorschläge in bezug auf Angriff und Verteidigung von Blöcken macht. Potentielle Züge können so auf ihren Wirkung untersucht werden. Dabei ist es nicht nötig, einen Spielbaum abzuarbeiten, da die Bewertung – abgesehen vom Spieler der am Zug ist und den damit verbundenen Klingeln – statisch ist.

Es lassen sich leicht Folgerungen für das Go-Spiel ableiten, wenn man Blöcke als lebend erkannt hat. Solche Blöcken können nicht mehr geschlagen werden und üben damit einen großen Einfluß auf ihre Umgebung aus. Durch die Typisierung der großen Augen und der damit eindeutig verbundenen Funktion, die sie für ihren Block haben, ist es möglich, Blöcke als lebend anzusehen, die noch keine 2 effektiven Augen haben. Dadurch wird Zeit gespart, die für Züge an anderer Stelle benutzt werden kann.
Die Tabelle, die den Typ des großen eigenen Auges in einen effektiven Augen-Wert übersetzt, kann man als Regelwerk ansehen, in dem Wissen über große eigene Augen gespeichert ist.
Welche Bedeutung die Typisierung für große gemeinsame Augen hat, ist noch nicht näher untersucht worden.

Der Algorithmus ist korrekt. Er kann Block-Gruppen bewerten, die über gemeinsame Augen miteinander verbunden sind. Die Algorithmen von Benson und Kraszek können in speziellen Situationen mehr Blöcke als lebend erkennen, da es ihnen gelingt, große Augen, in denen einzelne gegnerische Steine stehen, und Blöcke, die nicht über gemeinsame Augen miteinander verbunden sind, in ihre Verfahren zu integrieren.

6 Anhang: Go1 Source

```
    /*3456789012345678901234567890123456789012345678901234567890123456789012345678*/
    /*       1         2         3         4         5         6         7         */
    /*****************************************************************************
    * Go1 source
5   *
    * Autor:            Matthias Mueller-Prove
    * Datum:            21.Juni 91
    * letzte Aenderung: 9.Oktober 91
    * Dokumentation:    GoMMP-2
10  * System:           LPA MacPROLOG  auf Atari/Aladin
    *                   Quintus-PROLOG auf VAX/VMS
    *
    * Dies ist die Implementation eines Algorithmus, der Bloecke beim GO auf
    * Leben pruefen soll. vgl. GoMMP-2.2
15  *****************************************************************************
    * Letzte Aenderung/ Probleme
    * 22. 6.91  gemA/3 Œ gemA2, da der gleich der Laenge der BlockListe ist.
    * 26. 6.91  MengenSource,
    *           Go-Algorithmus fertig, aber noch nicht fehlerfrei.
20  *         ! Aufraeumen, Praedikatsnamen !!
    * 27. 6.91  Adaption fuer QuintusPROLOG,VAX/VMS
    *           Import( sets ), subtract/3 statt ohne/3
    * 19. 8.91  wieder Rück-Import nach Mac-LPA-Prolog
    *  8.10.91  Das gemA gf hatte nach Aufruf von go/0 noch den Wert lambda
25  *           set_nonvital/1
    *           Klausel2 von pruefe_vital/2 hatte ich vergessen
    *           data_gemA/2 data_block/3 statt gemA/2, block/3
    *             block/3 war die Beziehung zw. Blockname, #Augen und #gemA
    *             data_block/3 ist nun die Bez. zw. Block, #Augen und #gemA
30  *****************************************************************************/

    /* Import Mengen Source */
    /*VAX*:-ensure_loaded( [ library( basics ), library( sets ) ] ). */
    /* statt dessen ist BASICS Source und SETS Source mit in GO Source mit drin */
35

    /*===========================================================================*/
    /*          Die Datenbasis: Bloecke und gemAugen                             */
    /*===========================================================================*/
40
    /*---------------------------------------------------------------------------
    * data_block/3
    * ?Block
    * ?augen, die Anzahl der (eigenen) Augen
45  * ?ZahlgemA,  die Zahl der gemA, an denen der Block teil hat
    */

    data_block( ka,1,2 ).
    data_block( kb,2,4 ).
50  data_block( kc,0,2 ).
    data_block( kd,2,2 ).
    data_block( ke,0,1 ).
    data_block( kf,1,3 ).
    data_block( kg,1,1 ).
55
    /*---------------------------------------------------------------------------
    * data_gemA/2
    * ?gemA
    * ?blockListe, eine Liste aller beteiligten Bloecke
60  */

    data_gemA( ga, [ka,kb] ).
    data_gemA( gb, [kb,kc] ).
    data_gemA( gc, [kd,kf] ).
65  data_gemA( gd, [kf,kg] ).
    data_gemA( ge, [ka,kb,kc] ).
    data_gemA( gf, [kb,kd,ke,kf] ).
```

```
/*=============================================================================*/
/*          Dynamische Praedikate effA/2, modus_gemA/2 und ihre Inits          */
/*=============================================================================*/

/*-----------------------------------------------------------------------------
* effA/2
* ?Block
* ?zahl der effAugen des Blocks
*           Dieses Prädikat gibt die Zahl der effA eines Blocks an. effA/2
*           wird von init_effA/0 und effA_anpassen/1 assertiert und
*           retractiert.
*/

/*-----------------------------------------------------------------------------
* init_effA/0
*               fuer alle Bloecke wird effA( Block, _ ) mit der Zahl der Augen
*               belegt. (Block.effA:= Augen)
*/

init_effA:-
        retractall( effA( _,_ ) ),
        fail.

init_effA:-
        data_block( Block, NAugen, _ ),    /* fur alle Bloecke: effA := Augen */
        assert( effA( Block, NAugen ) ),
        fail.

init_effA.


/*-----------------------------------------------------------------------------
* modus_gemA/2
* ?gemA
* ?modus des gemA, vital, nonvital oder lambda
*/

/*-----------------------------------------------------------------------------
* init_modus_gemA/0
*               fuer alle gemA: modus auf lambda initialisieren
*/

init_modus_gemA:-
        retractall( modus_gemA( _,_ ) ),
        fail.

init_modus_gemA:-
        data_gemA( GemA,_ ),
        assert( modus_gemA( GemA, lambda ) ),
        fail.

init_modus_gemA.


/*=============================================================================*/
/*          Der Algorithmus                                                    */
/*=============================================================================*/

/*-----------------------------------------------------------------------------
* go/0
*             go/0 ist das Herzstueck des Algorithmus.
*             go/0 ist immer erfolgreich.
*/

go:-    go_init,
        go_vorlauf,
        go_hauptschleife,
  write_netz,
        !.
```

```
140
    /*------------------------------------------------------------------------
    * go_init/0                                                      <- go/0
    *               go_init/0 ist immer erfolgreich
    */
145
    go_init:-
            write( go-init ), nl,
            init_effA,
            init_modus_gemA,
150   write_netz.                           /* Ergebnisausgabe des Init */


    /*------------------------------------------------------------------------
    * go_vorlauf/0                                                   <- go/0
155 *               alle gemA, deren angrenzende Bloecke mindestens ein effA haben
    *               sind vital, d.h, es kann kein Block vom gemA herausgebrochen
    *               werden.
    *               go_vorlauf/0 ist immer erfolgreich
    */
160
    go_vorlauf:-
            write( go-vorlauf ), nl,
            data_gemA( GemA, BlockL ),            /* ein GemA mit seinen Bloecken */
            alle_effA_gr_0( BlockL ),             /* alle effA aller Bloecke > 0 */
165         set_vital( GemA ),
            fail.

    go_vorlauf:-
      write_netz.                               /* Ergebnisausgabe des Vorlaufs */
170

    /*------------------------------------------------------------------------
    * alle_effA_gr_0/1                                          <- go_vorlauf/0
    * +BloeckListe
175 *               alle_effA_groesser_0 (<- Null) ist erfolgreich, wenn
    *               die effA aller Bloecke der BlockListe > 0 sind.
    *               Das Praedikat darf nur 1x erfolgreich werden. (Wg Backtracking
    *               in go_vorlauf/0) Dazu das cut.
    */
180
    alle_effA_gr_0( [] ).

    alle_effA_gr_0( [ Block | Rest ] ):-
            effA( Block, Zahl_effA ),
185         Zahl_effA > 0,
            !,
            alle_effA_gr_0( Rest ).


190 /*------------------------------------------------------------------------
    * go_hauptschleife/0                                             <- go/0
    *               go_hauptschleife/0 ist immer erfolgreich
    */

195 go_hauptschleife:-
            write( go-hauptschleife ), nl,
            modus_gemA( GemA, lambda ),        /* fuer alle gemA = lambda */
            pruefe_vital( GemA, [] ),
            fail.
200
    go_hauptschleife.


    /*------------------------------------------------------------------------
205 * pruefe_vital/2                                                 <- go/0
    * +gemA         : Ausgangspkt der Untersuchung
    * +MerkListe    : Liste der gemA, die schon betrachtet wurden.
    *               Ein GemA ist vital, falls (Klausel 1) es keine armen Nachbarn
    *               hat, ODER für die armen Nachbarn die Bedingung ii) erfuellt
```

```
210 *               ist:
    *               a) fuer alle armenNachbarn gilt: gemA>1
    *               b) und pro armenNachbar mindestens eines dieser gemA vital ist
    *               Klausel 2: Sonst ist es nonvital und das Praedikat schlaegt
    *               fehl.
215 *               pruefe_vital/3 darf nur 1x erfolgreich sein (wg Backtrakking
    *               ueber alle gemA mit Modus = lambda in go_hauptschleife) dashalb
    *               steht ein cut am Ende der 1. Klausel.
    *               set_(nicht)vital/1 hat Seiteneffekte auf
    *               modus_gemA/2 und effA/2.
220 */

    pruefe_vital( GemA, _ ):-                       /* das GemA ist schon vital */
            modus_gemA( GemA, vital ), !.

225 pruefe_vital( GemA, MerkListe ):-
      write( [ pruefe_vital ,GemA , MerkListe ] ), nl,
            arme_Nachbarn( GemA, MerkListe, ArmeNachbarnL ),  /* ArmeNachbarnL:= */
            ( ArmeNachbarnL = [] ;                                    /* ODER */
              ( haben_weitere_gemA( ArmeNachbarnL ),
230             bedingung2b( ArmeNachbarnL, GemA, MerkListe ) )
            ),
            set_vital( GemA ), !.

    pruefe_vital( GemA, _ ):-
235         set_nonvital( GemA ),
            !, fail.


    /*-------------------------------------------------------------------------
240 * arme_Nachbarn/3                                          <- pruefe_vital/2
    * +GemA        : die armenNachbarn dieses gemAuges sollen gefunden werden.
    * +MerkListe   : arme Nachbarn duerfen nicht in dieser Liste stehen.
    * -ArmeNachbarnListe : Rueckgabewert
    *               Arme Nachbarn des GemAuges (GemA) sind alle Bloecke, die
245 *               1) am gemA beteiligt sind
    *               2) nicht in der MerkListe stehen und
    *               3) kein effA haben (arme_Nachbarn/2)
    */

250 arme_Nachbarn( GemA, MerkListe, ArmeNachbarnL ):-
            data_gemA( GemA, NachbarL ),                      /* alle Nachbarn */
            subtract( NachbarL, MerkListe, NachbarL2 ),  /* L2 :=  NachL - MerkL */
            arme_Nachbarn( NachbarL2, ArmeNachbarnL ).

255
    /*-------------------------------------------------------------------------
    * arme_Nachbarn/2                                           <- arme_Nachbarn/3
    * +Liste1      : potentielle arme Nachbarn
    * -Liste2      : wirliche arme Nachbarn.
260 *               Ermittelt alle Bloecke aus Liste1, die kein effA haben und gibt
    *               diese in Liste 2 zurück.
    */

    arme_Nachbarn( [], [] ).
265
    arme_Nachbarn( [ Block | Rest ], [ Block | Rest2 ] ):-
            effA( Block, 0 ),
            !,
            arme_Nachbarn( Rest, Rest2 ).
270
    arme_Nachbarn( [ _Block | Rest ], L2 ):-              /* ELSE */
            arme_Nachbarn( Rest, L2 ).


275 /*-------------------------------------------------------------------------
    * haben_weitere_gemA/1                                      <- pruefe_vital/2
    * +BlockListe
    *               alle Bloecke (=armeNachbarn) der BlockListe haben mehr als 1
    *               gemAugen
280 *               (Ist die 1 wirklich konstant ? )
```

```
    */

    haben_weitere_gemA( [] ).

285 haben_weitere_gemA( [ Block | Rest ]):-
            data_block( Block, _Augen, Zahl_gemA ),
            !,
            Zahl_gemA > 1,
            haben_weitere_gemA( Rest ).
290

    /*-------------------------------------------------------------------------
    * bedingung2b/3                                              <- pruefe_vital/2
    * +BlockListe
295 * +das Ausgangs-gemA
    * +MerkListe    : der Listenparameter aus pruefe_vital/2
    *                pro Block ist mind. 1 gemA (ungleich dem Ausgangs-gemA) vital
    *                Es werden die gemA eines Blocks untersucht.
    */
300
    bedingung2b( [], _Ur_gemA, _ ).

    bedingung2b( [ Block | Rest ], Ur_gemA, MerkListe ):-
            data_gemA( GemAuge, GABlockL ),                        /* ein GemA, */
305         member( Block, GABlockL ),                /* das an dem Block haengt */
            GemAuge \== Ur_gemA,                        /* und ein Weiteres ist */
            append( [Block], MerkListe, NeueListe ),
            pruefe_vital( GemAuge, NeueListe ),                /* und vital ist */
            !,                                                  /* !Rekursion ! */
310         bedingung2b( Rest, Ur_gemA, MerkListe ).
                                              /* fuer alle Bloecke der BlockL */


    /*-------------------------------------------------------------------------
315 * set_nonvital/1                                <-  pruefe_vital/2
    * +gemA
    *               dyn.modus_gemA/2 auf nonvital setzen
    */

320 set_nonvital( GemA ):-
      write( [set_nonvital,GemA] ), nl,
            retract( modus_gemA( GemA, _ ) ),
            assert( modus_gemA( GemA, nonvital ) ).

325
    /*-------------------------------------------------------------------------
    * set_vital/1                                 <- go/0, pruefe_vital/2
    * +gemA
    *               dyn.modus_gemA/2 auf vital setzen und effA der umliegenden
330 *               Bloecke anpassen.
    */

    set_vital( GemA ):-              /* GemA ist schon vital */
            modus_gemA( GemA, vital ).
335
    set_vital( GemA ):-
      write( [set_vital,GemA] ), nl,
            retract( modus_gemA( GemA, _ ) ),
            assert( modus_gemA( GemA, vital ) ),
340         !,
            data_gemA( GemA, GBlockL ),          /* umliegende Bloecke = GBlockL */
            effA_anpassen( GBlockL ).


345 /*-------------------------------------------------------------------------
    * effA_anpassen/1                                        <- set_vital/0
    * +BlockListe, mit Bloecken, die jetzt alle ein effA mehr haben.
    */

350 effA_anpassen( [] ).
```

```
    effA_anpassen( [ Block | Rest ] ):-
            effA( Block, 2 ),               /* hat schon genug effAugen */
            !,
355         effA_anpassen( Rest ).

    effA_anpassen( [ Block | Rest ] ):-
            effA( Block, 1 ),               /* bekommt 1 effA dazu */
            retract( effA( Block, 1 ) ),
360         assert( effA( Block, 2 ) ),
            !,
            effA_anpassen( Rest ).

    effA_anpassen( [ Block | Rest ] ):-     /* bekommt auch 1 effA dazu */
365         retract( effA( Block, 0 ) ),
            assert( effA( Block, 1 ) ),
            !,
            effA_anpassen( Rest ).

370
    /*=========================================================================*/
    /*          Ergebnis: lebendige Bloecke !                                  */
    /*=========================================================================*/

375 /*-------------------------------------------------------------------------
    * lebt/1
    * ?Block
    *               Ein Block lebt gdw er 2 effA hat
    */
380
    lebt( Block ):-
            effA( Block, 2 ).


385 /*=========================================================================*/
    /*          Ausgabe                                                        */
    /*=========================================================================*/

    /*-------------------------------------------------------------------------
390 * write_netz/0
    *               Ausgabe aller lebenden Bloecke und aller Modi der gemA
    */

    write_netz:-                                            /* modus_gemA */
395         bagof( GemA, modus_gemA( GemA, vital), L1 ),
            write( vitaleGemA_ ),write( L1 ),nl,
            fail.
    write_netz:-
            bagof( GemA, modus_gemA( GemA, lambda), L2 ),
400         write( lambdaGemA_ ),write( L2 ),nl,
            fail.
    write_netz:-
            bagof( GemA, modus_gemA( GemA, nonvital), L3 ),
            write( nichtvitaleGemA_ ),write( L3 ),nl,
405         fail.

    write_netz:-                                            /* lebende Bloecke */
            write( lebende_bloecke_ ),
            bagof( K, lebt( K ), Liste ),
410         write( Liste ), nl, write( ----- ), nl.

    write_netz:-
            write( keine ), nl, write( ----- ), nl.

415 /*345678901234567890123456789012345678901234567890123456789012345678901234567*/
    /*       1         2         3         4         5         6         7       */
```

7 Literatur

[Berlekamp 1982] : Berlekamp, E., Conway, J.H., Guy, R. : Winning Ways. Academic Press, London (1982)

[Berlekamp 1994] : Berlekamp, E., Wolfe, D. : Mathmatical Go Endgames. Ishi Press (1994)

[Benson 1976] : Benson, David : Life in the Game of Go. Information Science, vol. 10 (1976), pp 17-29 (nachgedruckt in: Levy, David N.L., Computer Games II, Springer (1988))

[Bozulich 1987] : Bozulich, Richard : The second book of Go. Ishi Press (1987)

[Conway 1976] : Conway, J.H. : On Numbers And Games. Academic Press, London (1976)

[Hamann 1985] : Hamann, Christian M. : Chronologie der Programmierung des japanischen Brettspiels GO – Eine Herausforderung an die Künstliche Intelligenz. Angewandte Informatik 12/85, Jg. 27, S. 501 - 511

[Kraszek 1988] : Kraszek, Janusz : Heuristics in the Life and Death Algorithm of a Go-playing Program. Computer Go No.9 (1988/89)

[Nievergelt 1994] : Nievergelt, J. : Das Go-Spiel, Mathematik und Computer. Informatik Spektrum 17, S.106-110 (1994)

[Shannon 1950] : Shannon, Claude, E. : Programming a computer for playing chess. Philosophical Mag.41, 314 (1950), S. 256 - 275

[Zobrist 1970] : Zobrist, A.L. : Feature extraction and representation for pattern recognition and the game of go. University of Wisconsin, Ph.D. Thesis (1970)