Die Entwicklung der Spielprogrammierung: Von John von Neumann bis zu den hochparallelen Schachmaschinen

Alexander Reinefeld
Zuse Institut Berlin
Humboldt-Universität zu Berlin

„Themen der Informatik im historischen Kontext“
Ringvorlesung an der HU Berlin, 02.06.2005

Spiel n. `nicht auf Nutzen ausgerichtete, vergnügliche, mit Ernst betriebene Tätigkeit, Zeitvertreib, Vergnügen, Wettkampf.' ... Das Substantiv (speel, spil, spel) ist nur kontinentalwestgerm. bezeugt.9. Jh.

Etym. Wörterbuch des Deutschen, DTV, 2000.

- das Spiel verderben
- die Hand mit im Spiel haben
- ins Spiel bringen
- abspielen
- zuspielen
- sich aufspielen
- Anspielung
- Spielraum

Outline

- **Games**

- **Searching Game Trees**
    - Minimax, Negamax, Alphabeta,

- **History**
    - From Minimax to Alphabeta

- **Searching Game Trees (cont.)**
    - NegaScout, SSS*, ...

- **Analysis**
    - empirical
    - all leaf permutations
    - strict dominance

- **Summary**

Why Games?

Games are

1. well defined by a set of simple rules,
2. scalable to a large complexity,
3. their outcome is measurable,

and they are fun.

„It is not that the games and mathematical problems are chosen because they are clear and simple; rather it is that they give us, for the smallest initial structures, the greatest complexity“

M. Minsky, 1968

Two-Person Zero-Sum Game

TWO-PERSON

2 parties involved, each trying to maximize his outcome

ZERO-SUM

the gain of the one player is the loss of the other

GAME

simple and well-defined rules

scalable to larger complexity

finite time (necessary to determine the outcome)

JOHN VON NEUMANN
1903–1957

M
Neu

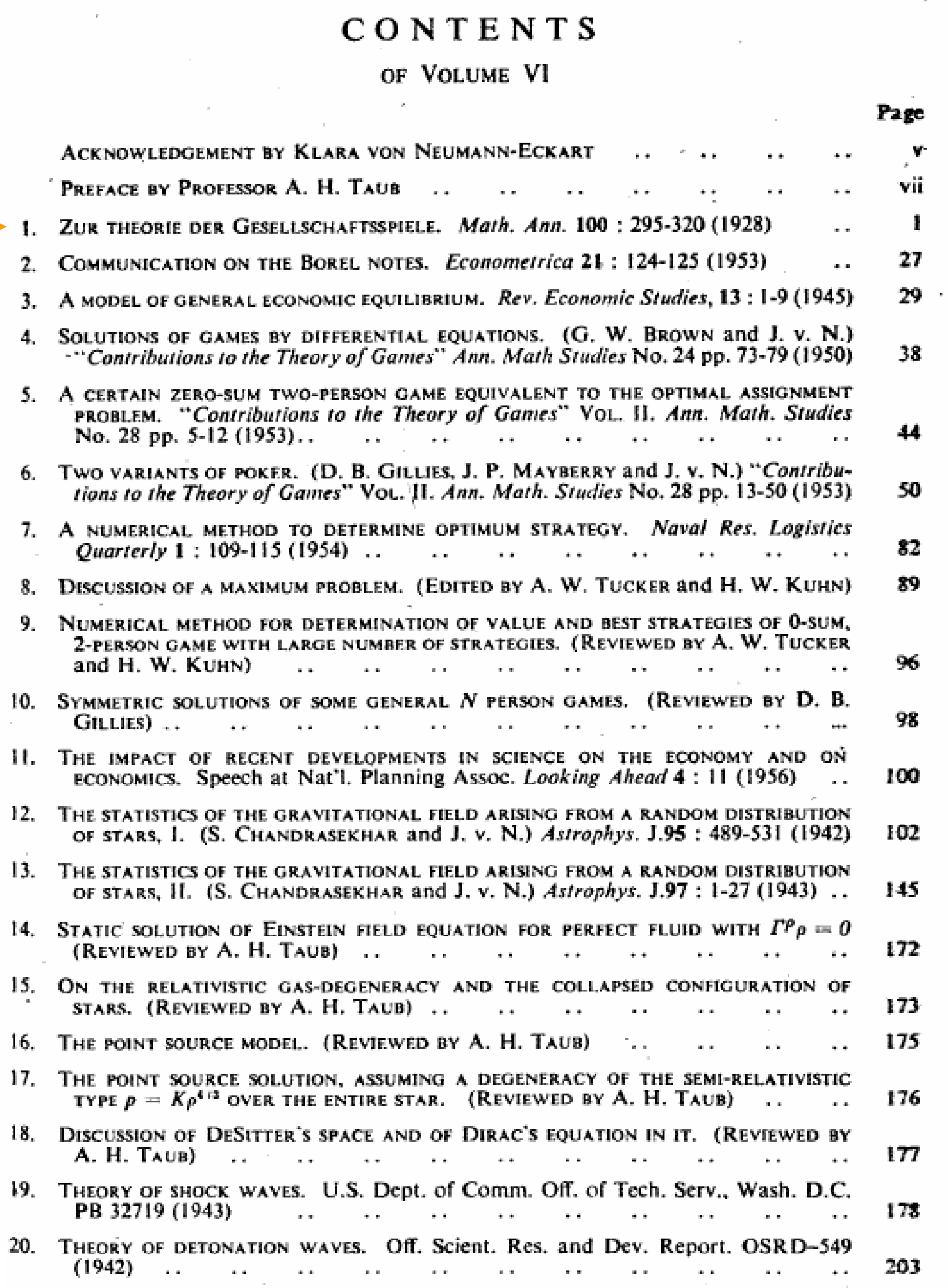
John von Neumann
COLLECTED WORKS

GENERAL EDITOR
A. H. TAUB
RESEARCH PROFESSOR OF APPLIED MATHEMATICS
DIGITAL COMPUTER LABORATORY
UNIVERSITY OF ILLINOIS

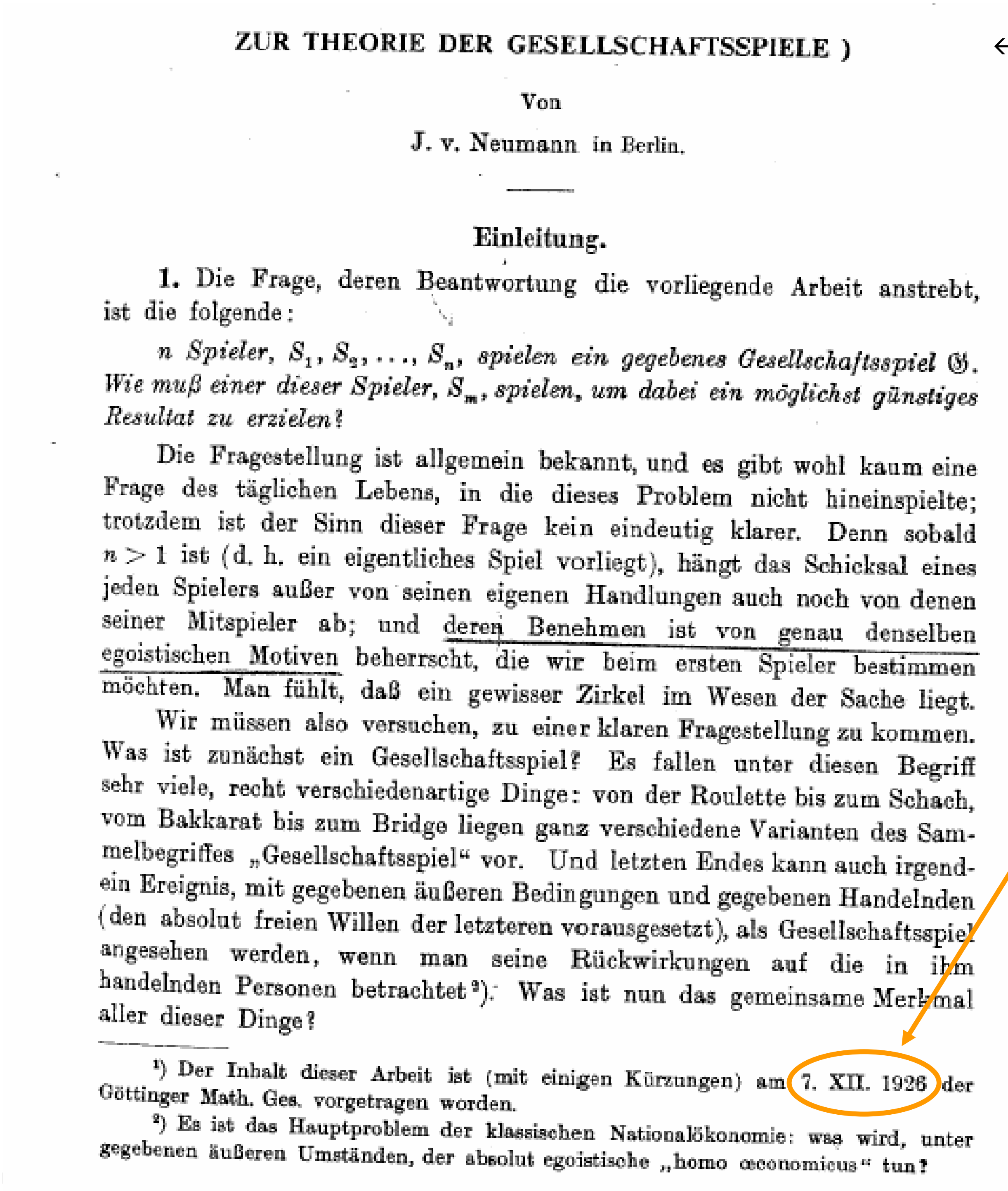
Volume VI
THEORY OF GAMES,
ASTROPHYSICS, HYDRODYNAMICS
AND METEOROLOGY

PERGAMON PRESS

JOHN VON NEUMANN
1903–1957


CONTENTS
OF VOLUME VI

Alexander Reinefeld



ZUR THEORIE DER GESELLSCHAFTSSPIELE)

Von

J. v. Neumann in Berlin.

Einleitung.

1. Die Frage, deren Beantwortung die vorliegende Arbeit anstrebt, ist die folgende:

n Spieler, $S_1, S_2, \ldots, S_n$, spielen ein gegebenes Gesellschaftsspiel $\mathfrak{G}$. Wie muß einer dieser Spieler, S_m, spielen, um dabei ein möglichst günstiges Resultat zu erzielen?

Die Fragestellung ist allgemein bekannt, und es gibt wohl kaum eine Frage des täglichen Lebens, in die dieses Problem nicht hineinspielte; trotzdem ist der Sinn dieser Frage kein eindeutig klarer. Denn sobald $n > 1$ ist (d. h. ein eigentliches Spiel vorliegt), hängt das Schicksal eines jeden Spielers außer von seinen eigenen Handlungen auch noch von denen seiner Mitspieler ab; und deren Benehmen ist von genau denselben egoistischen Motiven beherrscht, die wir beim ersten Spieler bestimmen möchten. Man fühlt, daß ein gewisser Zirkel im Wesen der Sache liegt.

Wir müssen also versuchen, zu einer klaren Fragestellung zu kommen. Was ist zunächst ein Gesellschaftsspiel? Es fallen unter diesen Begriff sehr viele, recht verschiedenartige Dinge: von der Roulette bis zum Schach, vom Bakkarat bis zum Bridge liegen ganz verschiedene Varianten des Sammelbegriffes „Gesellschaftsspiel" vor. Und letzten Endes kann auch irgendein Ereignis, mit gegebenen äußeren Bedingungen und gegebenen Handelnden (den absolut freien Willen der letzteren vorausgesetzt), als Gesellschaftsspiel angesehen werden, wenn man seine Rückwirkungen auf die in ihm handelnden Personen betrachtet[2]). Was ist nun das gemeinsame Merkmal aller dieser Dinge?

[1]) Der Inhalt dieser Arbeit ist (mit einigen Kürzungen) am 7. XII. 1926 der Göttinger Math. Ges. vorgetragen worden.

[2]) Es ist das Hauptproblem der klassischen Nationalökonomie: was wird, unter gegebenen äußeren Umständen, der absolut egoistische „homo œconomicus" tun?

← erschienen in: Mathematische Annalen **100**:295-320, 1928

J.v.N was only 23 years old.

Häufiger zitiert, aber viel später veröffentlicht:

J. v. Neumann, O. Morgenstern.

The Theory of Games and Economic Behavior,

Princeton Univ. Press, 1944.

Man darf wohl annehmen, daß es dieses ist:

Ein Gesellschaftsspiel besteht aus einer bestimmten Reihe von Ereignissen, deren jedes auf endlich viele verschiedene Arten ausfallen kann. Bei gewissen unter diesen Ereignissen hängt der Ausfall vom Zufall ab, d. h.: es ist bekannt, mit welchen Wahrscheinlichkeiten die einzelnen möglichen Resultate eintreten werden, aber niemand vermag sie zu beeinflussen. Die übrigen Ereignisse aber hängen vom Willen der einzelnen Spieler $S_1, S_2, \ldots, S_n$ *ab. D. h.: es ist bei jedem dieser Ereignisse bekannt, welcher Spieler* S_m *seinen Ausfall bestimmt, und von den Resultaten welcher anderer („früherer") Ereignisse er im Moment seiner Entscheidung bereits Kenntnis hat. Nachdem der Ausfall aller Ereignisse bereits bekannt ist, kann nach einer festen Regel berechnet werden, welche Zahlungen die Spieler* $S_1, S_2, \ldots, S_n$ *aneinander zu leisten haben.*

Es ist leicht, diese mehr qualitative Erklärung in die Form einer exakten Definition zu bringen. Diese Definition des Gesellschaftsspieles würde so lauten:

Um ein Gesellschaftsspiel $\mathfrak{G}$ *vollständig zu beschreiben, sind die folgenden Angaben notwendig, die zusammen die „Spielregel" ergeben:*

α) Es muß angegeben werden, wie viele vom Zufall abhängige Ereignisse oder „Ziehungen" und wieviel vom Willen der einzelnen Spieler abhängige Ereignisse oder „Schritte" erfolgen. Diese Anzahlen seien z *bzw.* s, *die „Ziehungen" bezeichnen wir mit* $E_1, E_2, \ldots, E_z$, *die „Schritte" mit* $F_1, F_2, \ldots, F_s$.

β) Es muß angegeben werden, auf wie viele Arten jede „Ziehung" E_μ *und jeder „Schritt"* F_ν *ausfallen kann. Diese Anzahlen seien* M_μ *bzw.* N_ν $(\mu = 1, 2, \ldots, z,\ \nu = 1, 2, \ldots, s)$. *Wir bezeichnen die betreffenden Resultate kurz mit ihren Nummern* $1, 2, \ldots, M_\mu$ *bzw.* $1, 2, \ldots, N_\nu$.

γ) Bei jeder „Ziehung" E_μ *müssen die Wahrscheinlichkeiten* $\alpha_\mu^{(1)}, \alpha_\mu^{(2)}, \ldots, \alpha_\mu^{(M_\mu)}$ *der einzelnen Resultate* $1, 2, \ldots, M_\mu$ *gegeben sein. Natürlich ist*

$$\alpha_\mu^{(1)} \geqq 0,\ \alpha_\mu^{(2)} \geqq 0, \ldots, \alpha_\mu^{(M_\mu)} \geqq 0,$$
$$\alpha_\mu^{(1)} + \alpha_\mu^{(2)} + \ldots + \alpha_\mu^{(M_\mu)} = 1.$$

δ) Bei jedem „Schritt" F_ν *muß erstens derjenige Spieler* S_m *angegeben sein, der den Ausfall dieses „Schrittes" bestimmt („dessen Schritt"* F_ν *ist):* $S_{(F_\nu)}$. *Ferner müssen die Nummern aller „Ziehungen" und „Schritte" angegeben sein, über deren Ausfall er im Momente seiner Entscheidung über* F_ν *Kenntnis hat. (Diese „Ziehungen" und „Schritte" nennen wir „früher" als* F_ν.)

Damit die ganze Sache möglich ist und zeitlich-kausal vorstellbar

ε) Schließlich müssen n *Funktionen* $f_1, f_2, \ldots, f_n$ *gegeben sein. Jede von ihnen ist abhängig von* $z + s$ *Variablen, die bzw. die Werte*

$$1, 2, \ldots, M_1;\quad 1, 2, \ldots, M_2;\quad \ldots;\quad 1, 2, \ldots, M_z;$$
$$1, 2, \ldots, N_1;\quad 1, 2, \ldots, N_2;\quad \ldots;\quad 1, 2, \ldots, N_s$$

durchlaufen. Diese Funktionen haben reelle Zahlen als Werte, und es gilt identisch

$$f_1 + f_2 + \ldots + f_n \equiv 0.$$

Wenn nun im Laufe einer zu Ende gespielten Partie die Resultate der z *„Ziehungen" und der* s *„Schritte" bzw.* $x_1, x_2, \ldots, x_z$, $y_1, y_2, \ldots, y_s$ $(x_\mu = 1, 2, \ldots, M_\mu,\ y_\nu = 1, 2, \ldots, N_\nu;\ \mu = 1, 2, \ldots, z,\ \nu = 1, 2, \ldots, s)$ *waren, so erhalten die Spieler* $S_1, S_2, \ldots, S_n$ *voneinander* [3]) *die Summen*

$$f_1(x_1, \ldots, x_z,\ y_1, \ldots, y_s),\quad f_2(x_1, \ldots, x_z,\ y_1, \ldots, y_s),\quad \ldots,$$
$$f_n(x_1, \ldots, x_z,\ y_1, \ldots, y_s).$$

(Trotz der etwas langatmigen Beschreibung handelt es sich hier, wenn man genau zusieht, um recht einfache und klare Dinge. Übrigens hätten wir die Definition in mehreren Beziehungen etwas allgemeiner fassen können: so hätten wir z. B. zulassen können, daß die M_μ, N_ν und $\alpha_1^{(\mu)}, \alpha_2^{(\mu)}, \ldots, \alpha_{M_\mu}^{(\mu)}$ von den Resultaten der „früheren" „Ziehungen" und „Schritte" abhängen, u. ä.; indessen überzeugt man sich leicht davon, daß dabei nichts wesentlich Neues herauskommt.)

2. Mit dieser Definition ist der Begriff des Gesellschaftsspieles genau umschrieben. Es tritt aber auch ganz klar in Erscheinung, was wir bereits am Anfang von 1. berührten, daß nämlich die Ausdrucksweise: „S_m sucht ein möglichst günstiges Resultat zu erzielen" eine recht unklare ist. Ein für den Spieler S_m möglichst günstiges Resultat ist offenbar ein möglichst großer Wert von f_m, aber wie soll überhaupt irgendein Wert von f_m durch S_m „erzielt" werden? S_m ist ja allein gar nicht in der Lage, den Wert von f_m festzulegen! f_m hängt von den Variablen $x_1, \ldots, x_z$, $y_1, \ldots, y_s$ ab, und von diesen wird nur ein Teil durch den Willen von S_m bestimmt (nämlich diejenigen y_ν, für die S_m den „Schritt" F_ν hat, d. h. $S_{(F_\nu)} = S_m$ ist); die übrigen Variablen hängen vom Willen der Mitspieler (nämlich alle übrigen y_ν) oder vom Zufall (nämlich alle x_μ) ab.

In unserem Falle ist der „unvoraussehbare" Zufall noch der leichter zu beherrschende Faktor. In der Tat: nehmen wir an, ein f_m hinge außer von jenen y_ν, die S_m bestimmt ($S_{(F_\nu)} = S_m$), nur von den x_μ (die vom

[3]) Die Identität

$$f_1 + f_2 + \ldots + f_n \equiv 0$$

$(x_m = 1, 2, \dots, \Sigma_m,\ m = 1, 2, \dots, n)$, *so erhalten sie bzw. die folgenden Summen:*

$$g_1(x_1, \dots, x_n),\quad g_2(x_1, \dots, x_n),\quad \dots,\quad g_n(x_1, \dots, x_n).$$

(*Dabei ist identisch* $g_1 + g_2 + \dots + g_n \equiv 0$.)

Damit ist diejenige Form der Spielregel erreicht, die (trotzdem sie, wie wir soeben zeigten, im wesentlichen nichts an Allgemeinheit verloren hat), nur noch die für uns wesentlichen Merkmale des Gesellschaftsspieles zeigt. Vom „Glücksspiel“ ist nichts mehr da: die Handlungen aller Spieler bestimmen das Resultat restlos (weil ja so operiert wird, als ob es ein jeder von ihnen nur auf den Erwartungswert abgesehen hätte). Aber dafür tritt das am Ende der Einleitung hervorgehende Prinzip in voller Schärfe in Erscheinung: jedes g_m hängt von allen $x_1, x_2, \dots, x_n$ ab.

Der aus der Wahrscheinlichkeitsrechnung bekannte Fall: g_m hängt nur nur von x_m ab (was natürlich nicht für alle m eintreten kann), erscheint nun als vollkommen trivial.

II. Der Fall $n = 2$.

1. Weiter können wir zunächst in der bisherigen Allgemeinheit nicht kommen, es erweist sich vielmehr als zweckmäßig, jetzt den einfachsten Fall für n zu betrachten. Der Fall $n = 0$ ist sinnlos, der Fall $n = 1$ (wegen $g_1 + \dots + g_n \equiv 0$) ebenfalls, beidemal ist kein eigentliches Gesellschaftsspiel vorhanden. Es ist also der Fall $n = 2$, der nun in Frage kommt. ?

Da $g_1 + g_2 \equiv 0$ ist, kann $g_1 = g$, $g_2 = -g$ gesetzt werden. Dann lautet die Beschreibung des allgemeinen 2-Personen-Spieles so:

Die Spieler S_1, S_2 *wählen irgendwelche der Zahlen* $1, 2, \dots, \Sigma_1$ *bzw.* $1, 2, \dots, \Sigma_2$ *und zwar jeder ohne die Wahl des anderen zu kennen. Wenn sie die Zahlen* x *bzw.* y *gewählt haben, so erhalten sie die Summen* $g(x, y)$ *bzw.* $-g(x, y)$.

Dabei kann nun $g(x, y)$ jede beliebige Funktion (definiert für $x = 1, 2, \dots, \Sigma_1$, $y = 1, 2, \dots, \Sigma_2$!) sein.

Es ist leicht, sich ein Bild von den Tendenzen zu machen, die in einem solchen 2-Personen-Spiele miteinander kämpfen: Es wird von zwei Seiten am Werte von $g(x, y)$ hin und her gezerrt, nämlich durch S_1, der ihn möglichst groß, und durch S_2, der ihn möglichst klein machen will. S_1 gebietet über die Variable x, und S_2 über die Variable y. Was wird geschehen?

jedenfalls $\geqq \mathrm{Min}_y\, g(x, y)$. Und diese untere Grenze kann durch geeignete Wahl von x gleich $\mathrm{Max}_x\, \mathrm{Min}_y\, g(x, y)$ (und nicht größer!) gemacht werden. D. h. wenn S_1 es will, so kann er $g(x, y)$ (unabhängig von S_2!) jedenfalls

$$\geqq \mathrm{Max}_x\, \mathrm{Min}_y\, g(x, y)$$

machen. Ebenso zeigt man: wenn S_2 es will, so kann er $g(x, y)$ (unabhängig von S_1!) jedenfalls

$$\leqq \mathrm{Min}_y\, \mathrm{Max}_x\, g(x, y)$$

machen.

Wenn nun

$$\mathrm{Max}_x\, \mathrm{Min}_y\, g(x, y) = \mathrm{Min}_y\, \mathrm{Max}_x\, g(x, y) = M$$

ist, so folgt aus dem Obigen, sowie daraus, daß S_1 das $g(x, y)$ möglichst groß und S_2 es möglichst klein machen will, daß $g(x, y)$ den Wert M haben wird. Denn S_1 hat das Interesse, es groß zu machen, und kann verhindern, daß es kleiner als M wird; S_2 hingegen hat das Interesse, es klein zu machen und kann verhindern, daß es größer als M wird. Folglich wird es den Wert M haben.

Nun ist zwar allgemein

$$\mathrm{Max}_x\, \mathrm{Min}_y\, g(x, y) \leqq \mathrm{Min}_y\, \mathrm{Max}_x\, g(x, y),$$

aber es besteht keineswegs stets das $=$-Zeichen. Es ist vielmehr leicht, solche $g(x, y)$ anzugeben, bei denen das $<$-Zeichen gilt, wo also diese Überlegung versagt. Das einfachste derartige Beispiel ist das folgende:

$$\Sigma_1 = \Sigma_2 = 2,\qquad \begin{aligned} g(1, 1) &= 1, & g(1, 2) &= -1,\\ g(2, 1) &= -1, & g(2, 2) &= 1. \end{aligned}$$

(Es ist offenbar Max Min $= -1$ und Min Max $= 1$.)

Ein anderes Beispiel ist die sog. „Morra“[8]):

$$\Sigma_1 = \Sigma_2 = 3,\quad \begin{aligned} g(1, 1) &= 0, & g(1, 2) &= 1, & g(1, 3) &= -1,\\ g(2, 1) &= -1, & g(2, 2) &= 0, & g(2, 3) &= 1,\\ g(3, 1) &= 1, & g(3, 3) &= -1, & g(3, 3) &= 0. \end{aligned}$$

(Auch hier ist Max Min $= -1$ und Min Max $= 1$.)

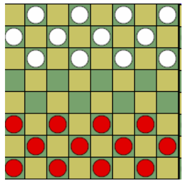
Daß diese Schwierigkeit auftritt, kann man sich auch so klarmachen: $\mathrm{Max}_x\, \mathrm{Min}_y\, g(x, y)$ ist das beste Resultat, das S_1 erzielen kann, wenn ihn S_2 vollkommen durchschaut: wenn S_2, sooft S_1 x spielt, ein solches y

Properties

The complete status (all choices and all consequences) of the game is given.

Just by knowing the rules, any novice is able to play an optimal game.

So what’s the problem? **The size of the search space!**



Checkers: ~10^{78}

[A.L. Samuel, “Some studies in machine learning using the game of checkers, 1963]

Chess: ~10^{120}

[C.E. Shannon, “Programming a computer for playing chess”, 1950]

Go: ~10^{170}

AND Trees

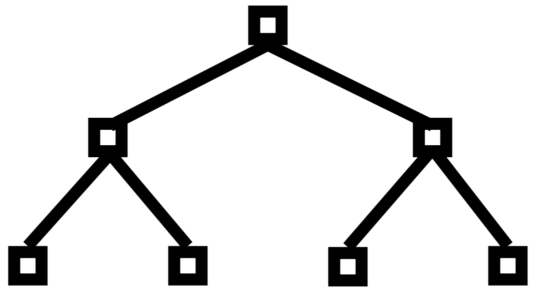
To solve a problem, **all sub-problems** must be solved, i.e. all nodes must be explored.

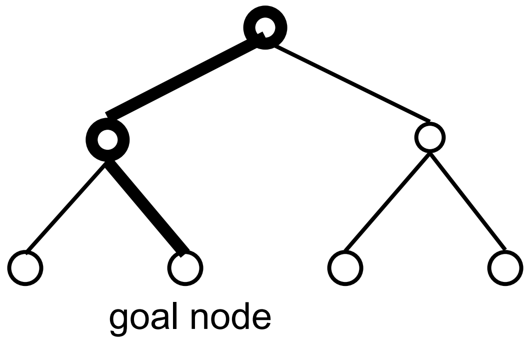
OR Trees

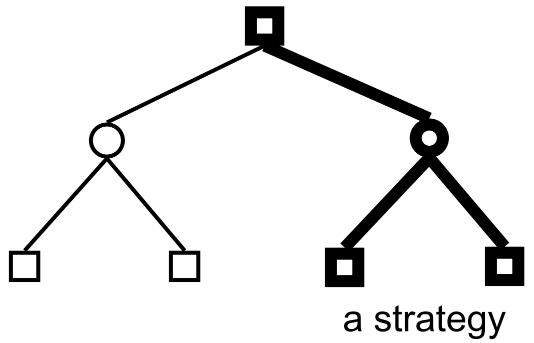
To solve a problem, **one solution** (path) must be found.

AND / OR Trees

To solve a problem, an optimal **strategy** must be found







Examples

- problem decomposition

Algorithms

- divide and conquer paradigm (no search)

Examples

- combinatorial optim.
- constraint satisfaction
- robotics, ...

Algorithms

- A*, IDA*
- Branch&Bound

Examples

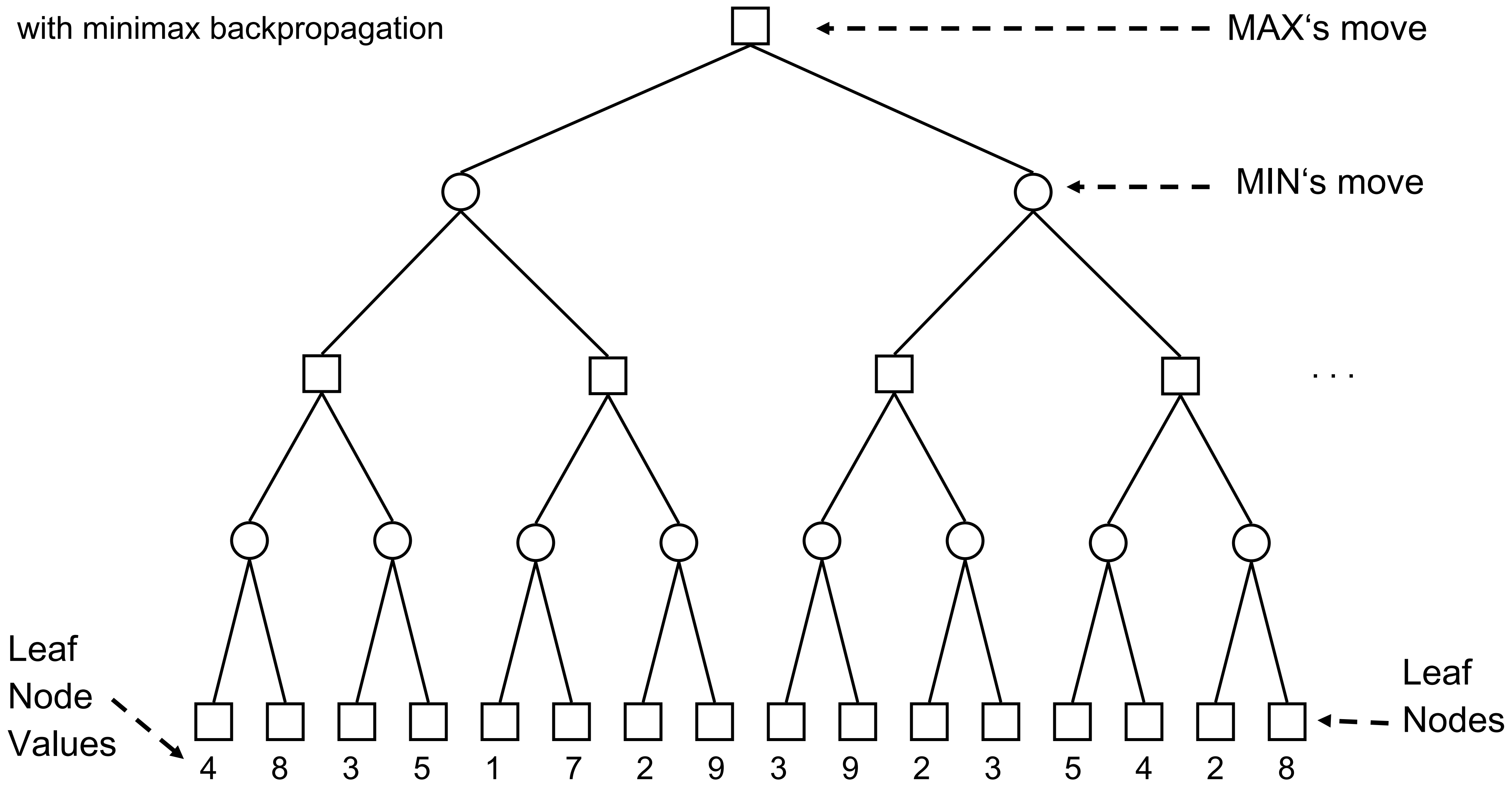
- theorem proving

Algorithms

- AO*
- SSS*, Dual*
- AlphaBeta, NegaScout

A Game Tree

A Game Tree is a AND/OR tree with minimax backpropagation



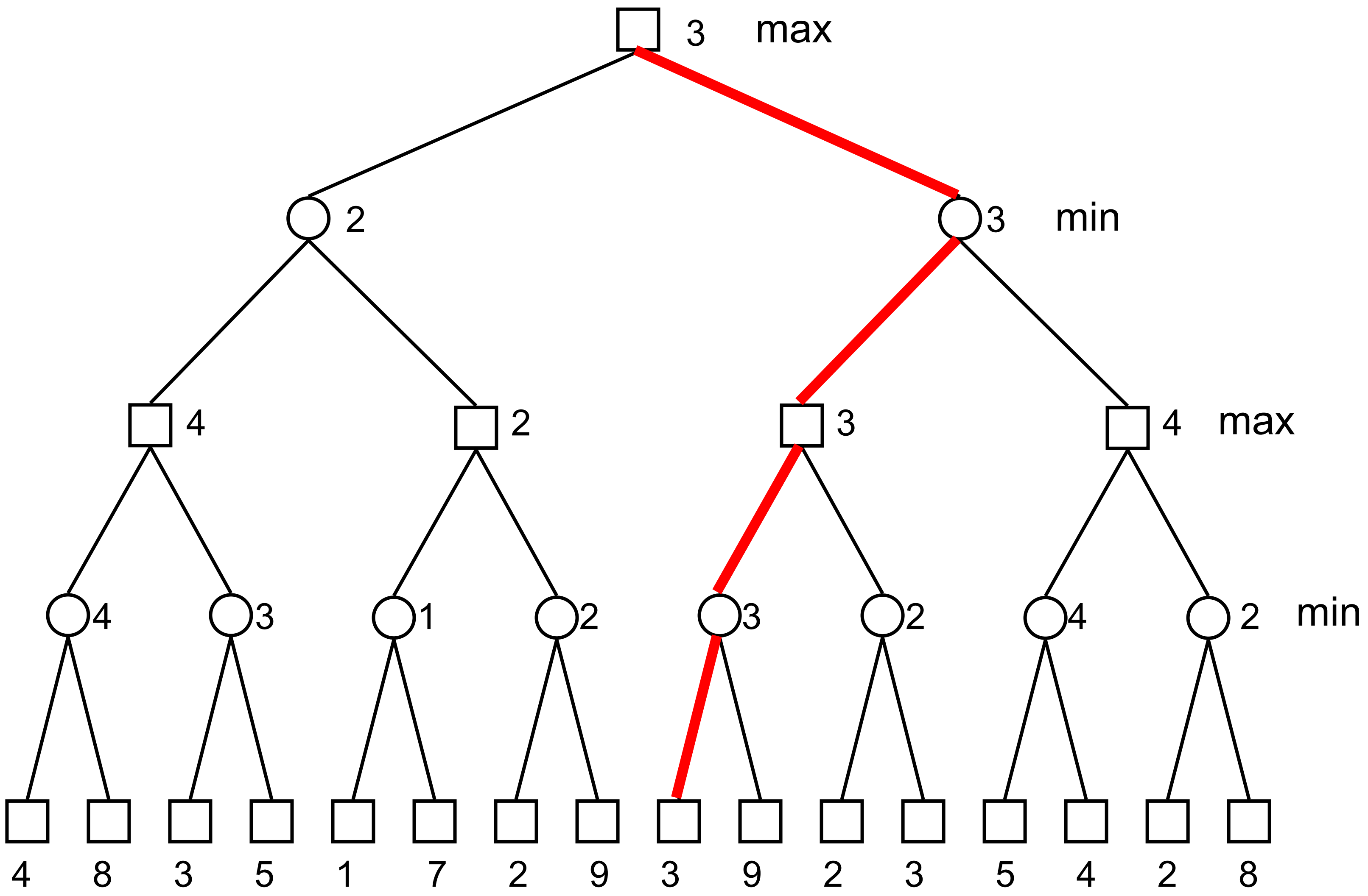
Note: We use a binary tree for illustration only. A typical chess tree has a branching factor of b = 35

Minimax Value v(J)

$$v(J) = \begin{cases} f(J) & \text{if J is a leaf node} \\ \max \{ v(J.j) \mid 1 \leq j \leq w \} & \text{if J is an interior MAX node} \\ \min \{ v(J.j) \mid 1 \leq j \leq w \} & \text{if J is an interior MIN node} \end{cases}$$

$f(J)$ = leaf value

Minimax



Where is the principal variation?

Negamax Value u(J)

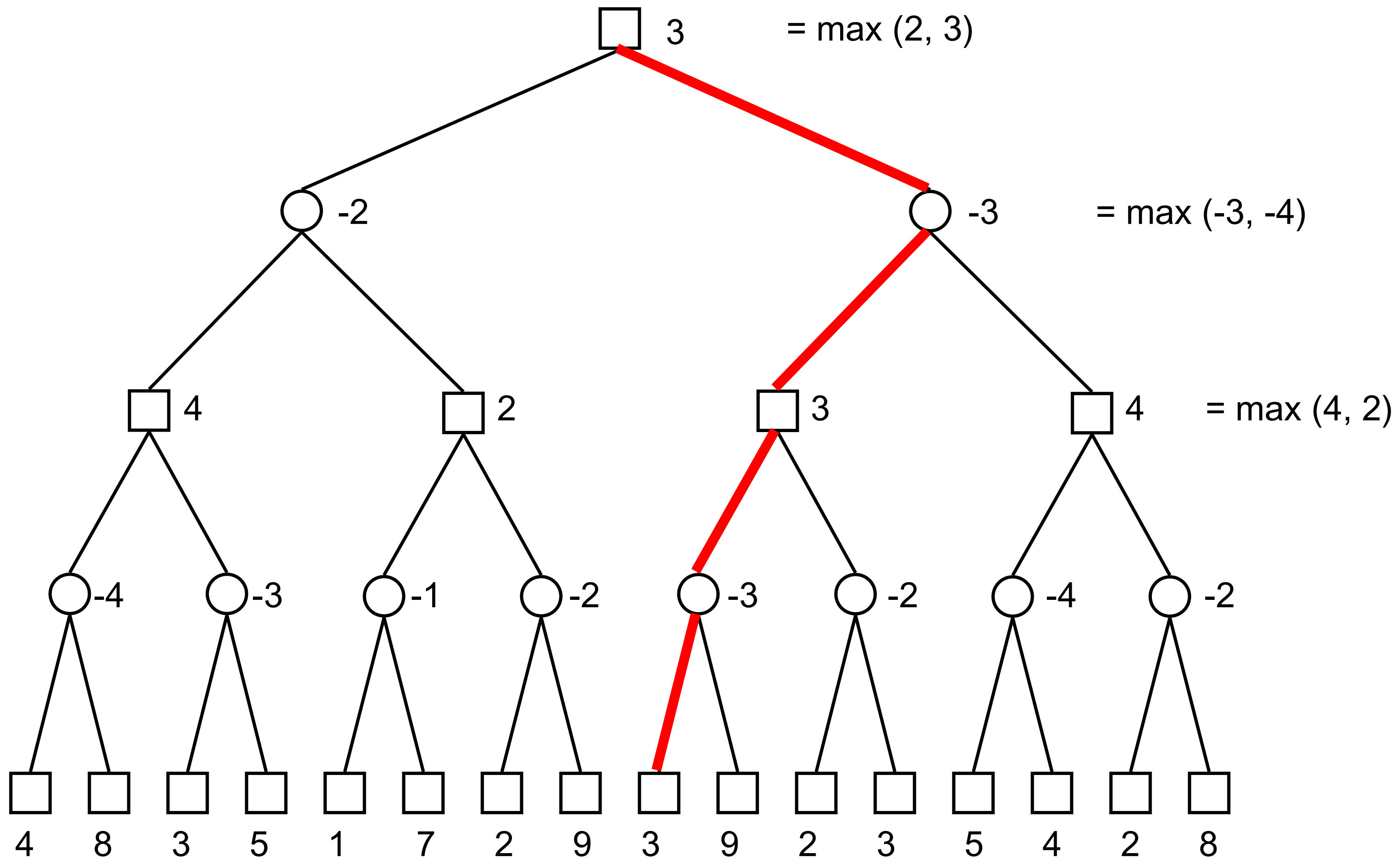
By observing that *min { a, b } = - max { -a, -b }* we can simplify the minimax function.

$$u(J) = \begin{cases} g(J) & \text{if } J \text{ is a leaf node} \\ \max \{ -v(J.j) \mid 1 \leq j \leq w \} & \text{if } J \text{ is an interior node} \end{cases}$$

$$g(J) = \begin{cases} f(J) & \text{if } J \text{ is a MAX leaf node} \\ -f(J) & \text{if } J \text{ is a MIN leaf node} \end{cases}$$

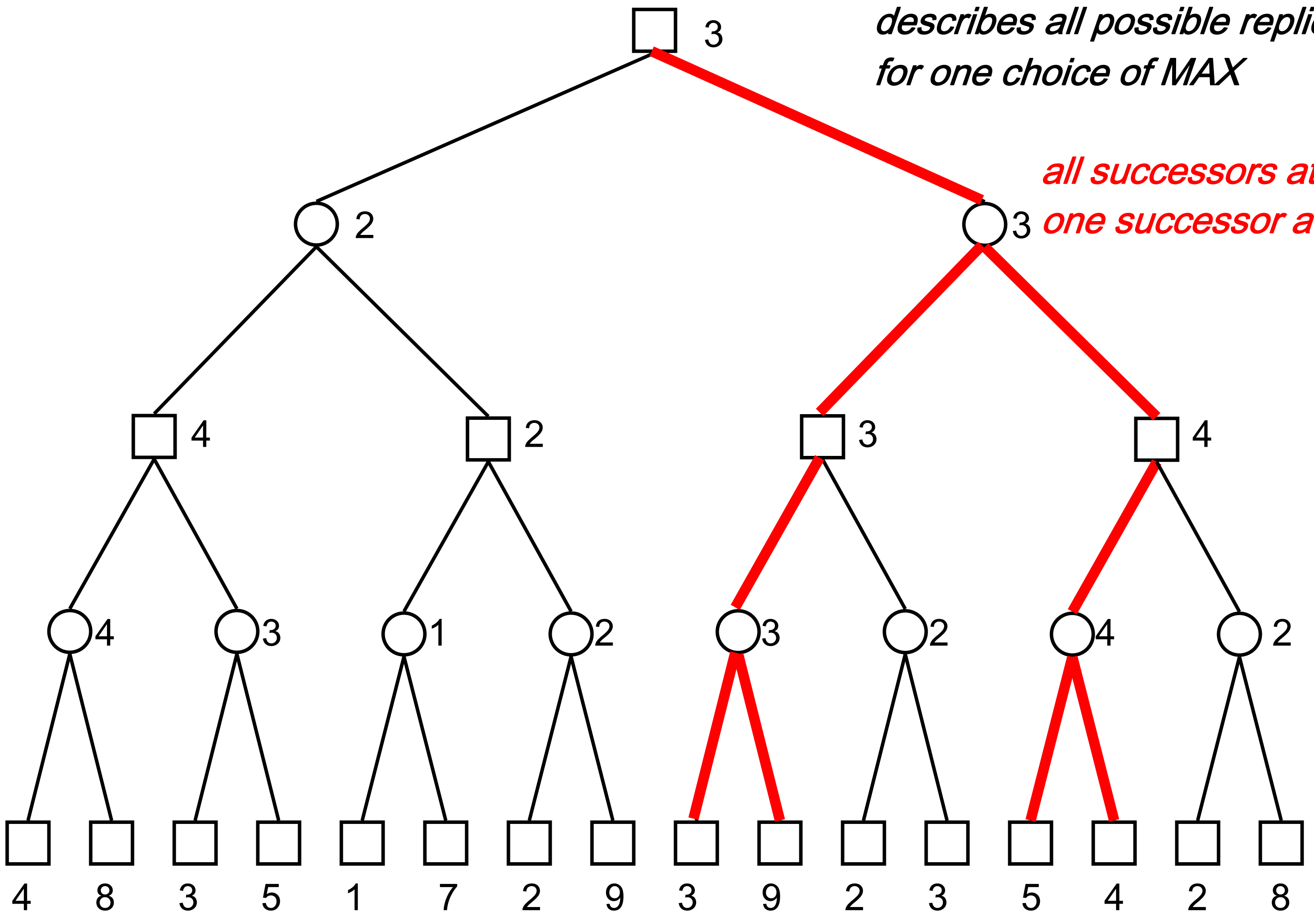
Negamax

computes the maximum of the negated successor values



principal variation

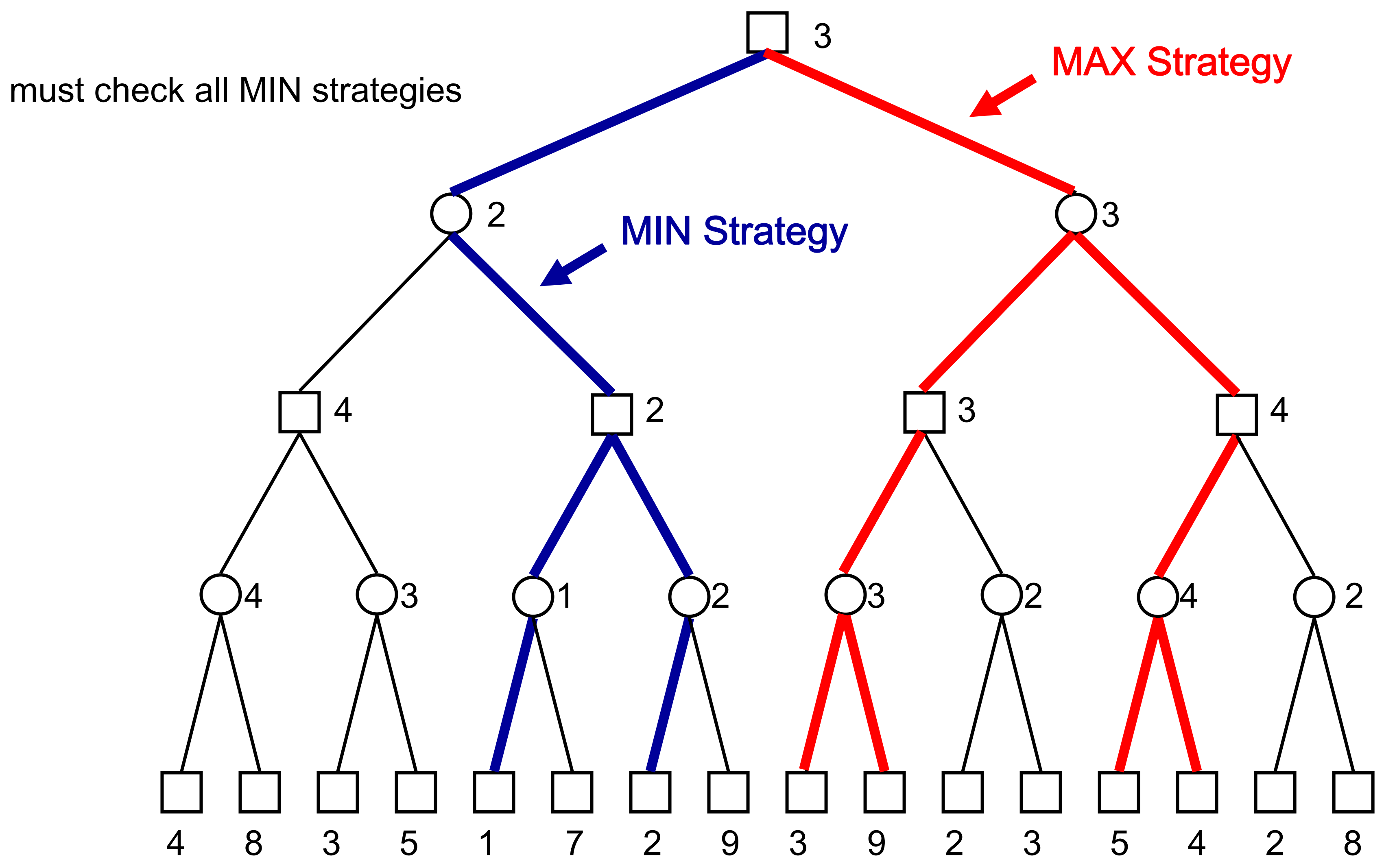
A MAX Strategy



describes all possible replies of MIN for one choice of MAX

all successors at MIN nodes
one successor at MAX nodes

Proof that the MAX Strategy is Optimal



Four ways to „solve“ a Game Tree

- compute the **minimax** (resp. negamax) **value** ← Minimax, Alphabeta, NegaScout
- enumerate all **MAX strategies** and chose the best one ← SSS*, DUAL*
- determine „maximum word“ in the corresponding **context free grammar**
- solve the **disjunctive normal form**
- resistor/capacitor network
- maximal flow problem
- … (many more)

Alphabeta

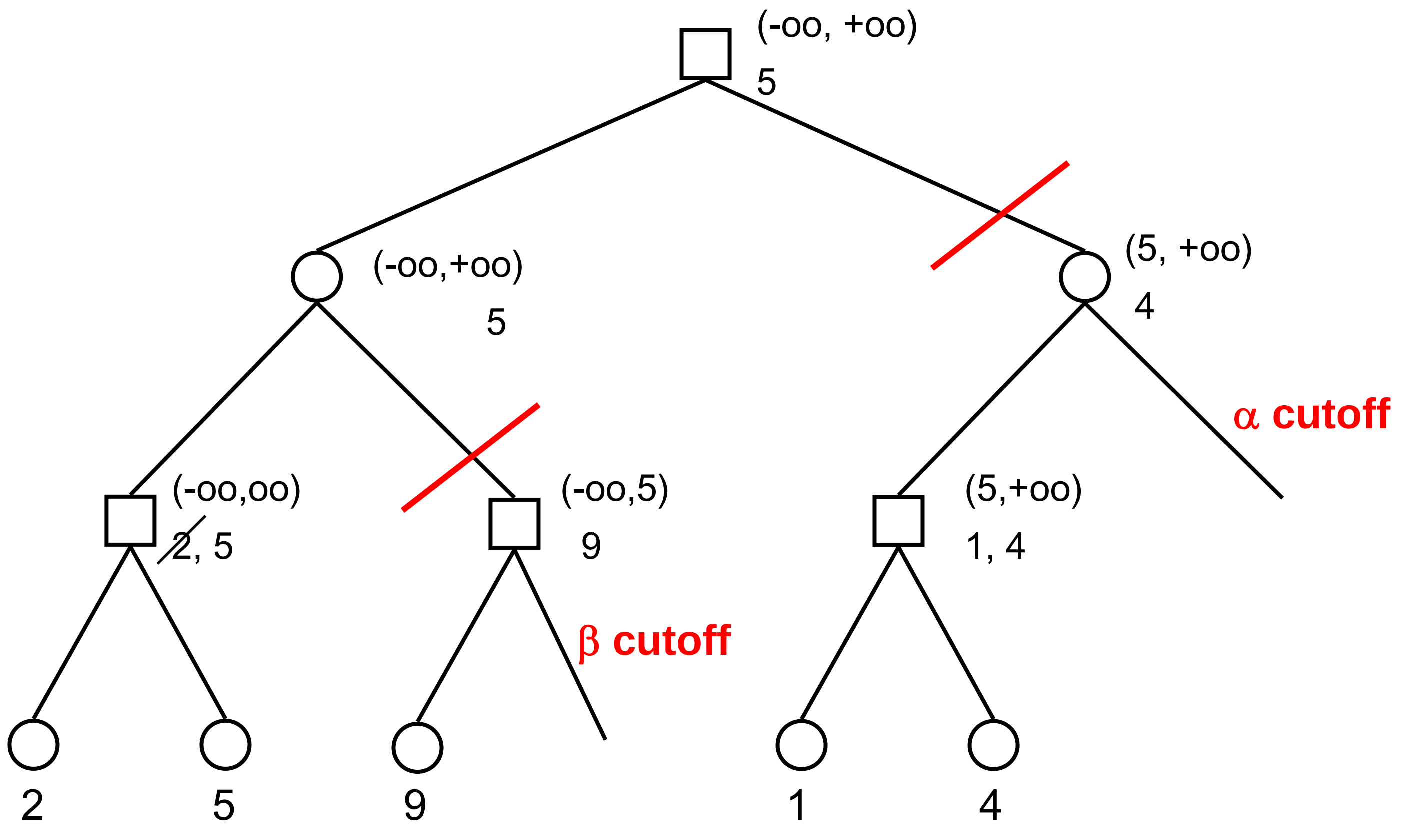
- computes the minimax value without examining all nodes
  - Branch-and-bound (Land & Doig 1960, Lawler & Wood 1966, ...)
  - Alphabeta (1963, 1975)

- Examine nodes with a **search window**
  - start with search window $(-\infty,\infty)$
  - α is minimal value (lower bound) known so far for MAX
  - β is maximal value (upper bound) known so far for MIN
  - disregard all nodes $\leq \alpha$ and $\geq \beta$

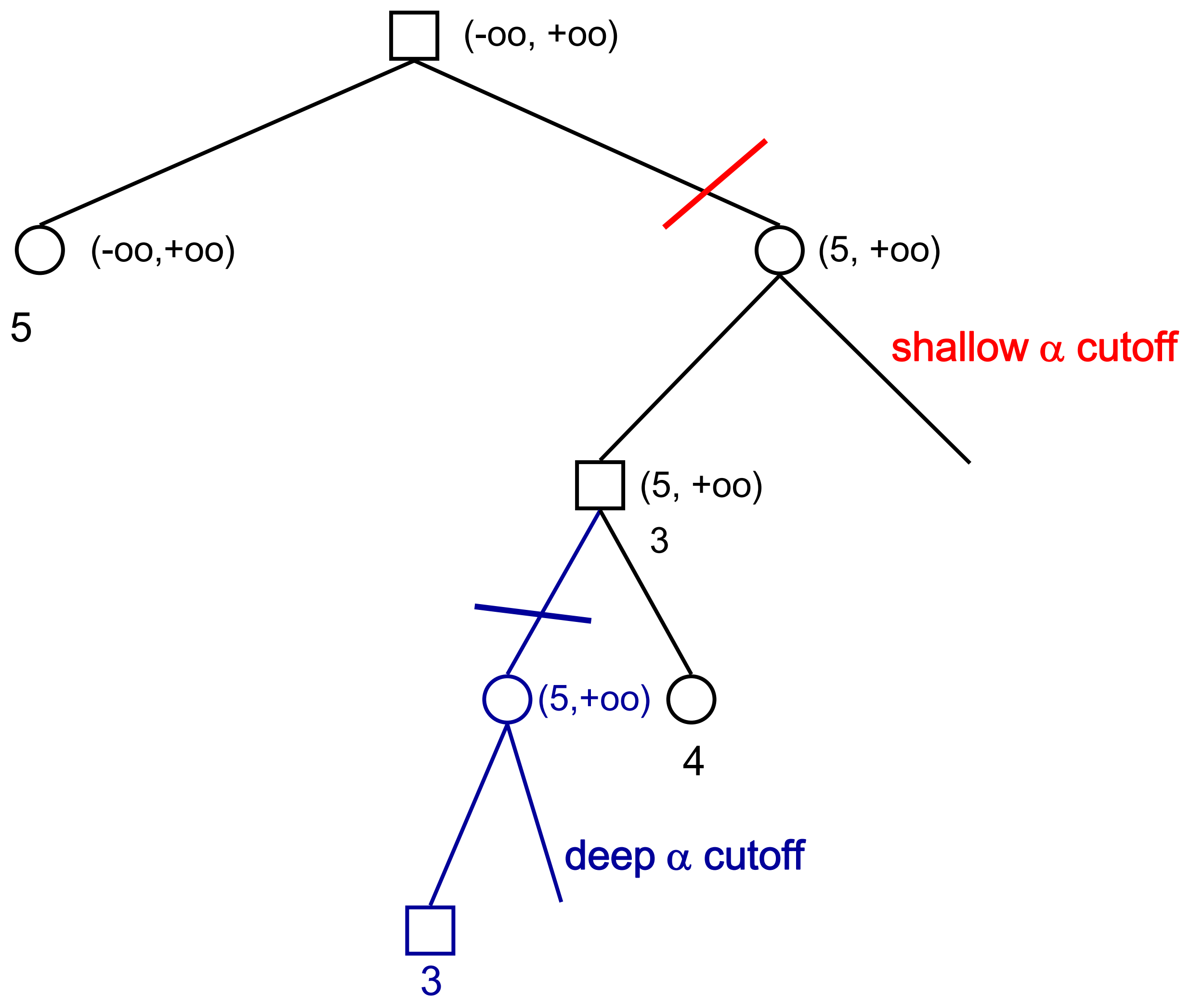
Minimax Version of Alphabeta

```
int AB_MAX(position p; int α, β);
{
        if (p == LEAF) return(Evaluate(p));
        val = α;
        for (i = 1; i ≤ w; i++) {
                val = max (val, AB_MIN (p_i, val, β));
                if (val ≥ β) return (val);     /* beta cut-off */
        }
        return (val);
}

int AB_MIN(position p; int α, β);
{
        if (p == LEAF) return(Evaluate(p));
        val = β;
        for (i = 1; i ≤ w; i++) {
                val = min (val, AB_MAX (p_i, α, val);
                if (val ≤  α) return (val);     /* alpha cut-off */
        }
        return (val);
}
```

Alphabeta



Shallow and Deep Cutoff



Brief History

From Minimax to Alphabeta

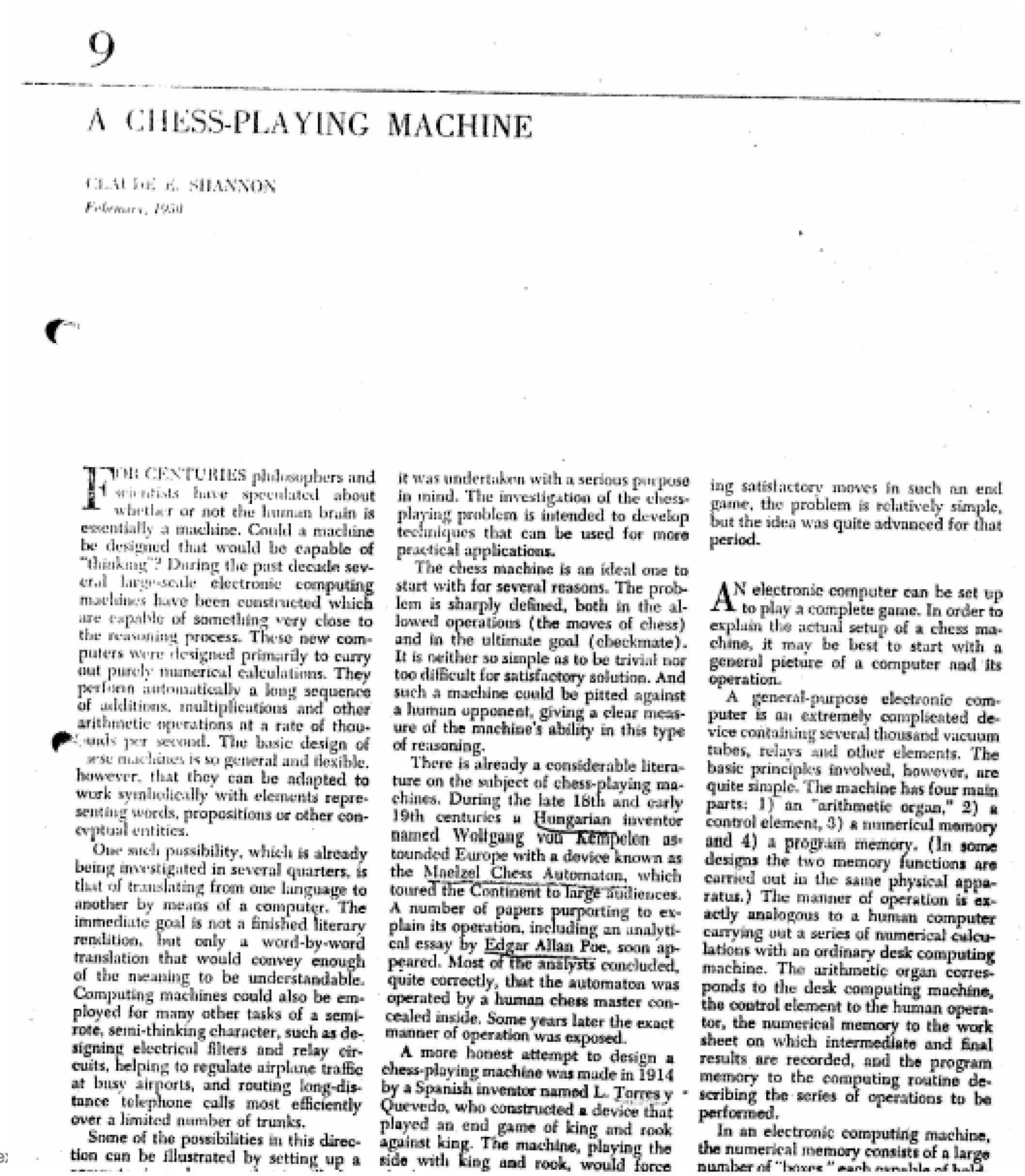
1956: McCarthy was the first who noticed that not all branches need to be examined (personal conversation, see Knuth & Moore, 1975)

1958: Newell, Shaw and Simon used branch-and-bound in their chess program

1963: Brudno published the first paper on alpha-beta (in Russian)

1975: Knuth & Moore published the full alpha-beta, including deep cut-offs

C.E. Shannon, Programming a Computer for Playing Chess, *Philosophical Magazine 41*, 7 (1950), 256-275.



9

A CHESS-PLAYING MACHINE

CLAUDE E. SHANNON

February, 1950

FOR CENTURIES philosophers and scientists have speculated about whether or not the human brain is essentially a machine. Could a machine be designed that would be capable of "thinking"? During the past decade several large-scale electronic computing machines have been constructed which are capable of something very close to the reasoning process. These new computers were designed primarily to carry out purely numerical calculations. They perform automatically a long sequence of additions, multiplications and other arithmetic operations at a rate of thousands per second. The basic design of these machines is so general and flexible, however, that they can be adapted to work symbolically with elements representing words, propositions or other conceptual entities.

One such possibility, which is already being investigated in several quarters, is that of translating from one language to another by means of a computer. The immediate goal is not a finished literary rendition, but only a word-by-word translation that would convey enough of the meaning to be understandable. Computing machines could also be employed for many other tasks of a semi-rote, semi-thinking character, such as designing electrical filters and relay circuits, helping to regulate airplane traffic at busy airports, and routing long-distance telephone calls most efficiently over a limited number of trunks.

Some of the possibilities in this direction can be illustrated by setting up a

it was undertaken with a serious purpose in mind. The investigation of the chess-playing problem is intended to develop techniques that can be used for more practical applications.

The chess machine is an ideal one to start with for several reasons. The problem is sharply defined, both in the allowed operations (the moves of chess) and in the ultimate goal (checkmate). It is neither so simple as to be trivial nor too difficult for satisfactory solution. And such a machine could be pitted against a human opponent, giving a clear measure of the machine's ability in this type of reasoning.

There is already a considerable literature on the subject of chess-playing machines. During the late 18th and early 19th centuries a Hungarian inventor named Wolfgang von Kempelen astounded Europe with a device known as the Maelzel Chess Automaton, which toured the Continent to large audiences. A number of papers purporting to explain its operation, including an analytical essay by Edgar Allan Poe, soon appeared. Most of the analysts concluded, quite correctly, that the automaton was operated by a human chess master concealed inside. Some years later the exact manner of operation was exposed.

A more honest attempt to design a chess-playing machine was made in 1914 by a Spanish inventor named L. Torres y Quevedo, who constructed a device that played an end game of king and rook against king. The machine, playing the side with king and rook, would force ing satisfactory moves in such an end game, the problem is relatively simple, but the idea was quite advanced for that period.

AN electronic computer can be set up to play a complete game. In order to explain the actual setup of a chess machine, it may be best to start with a general picture of a computer and its operation.

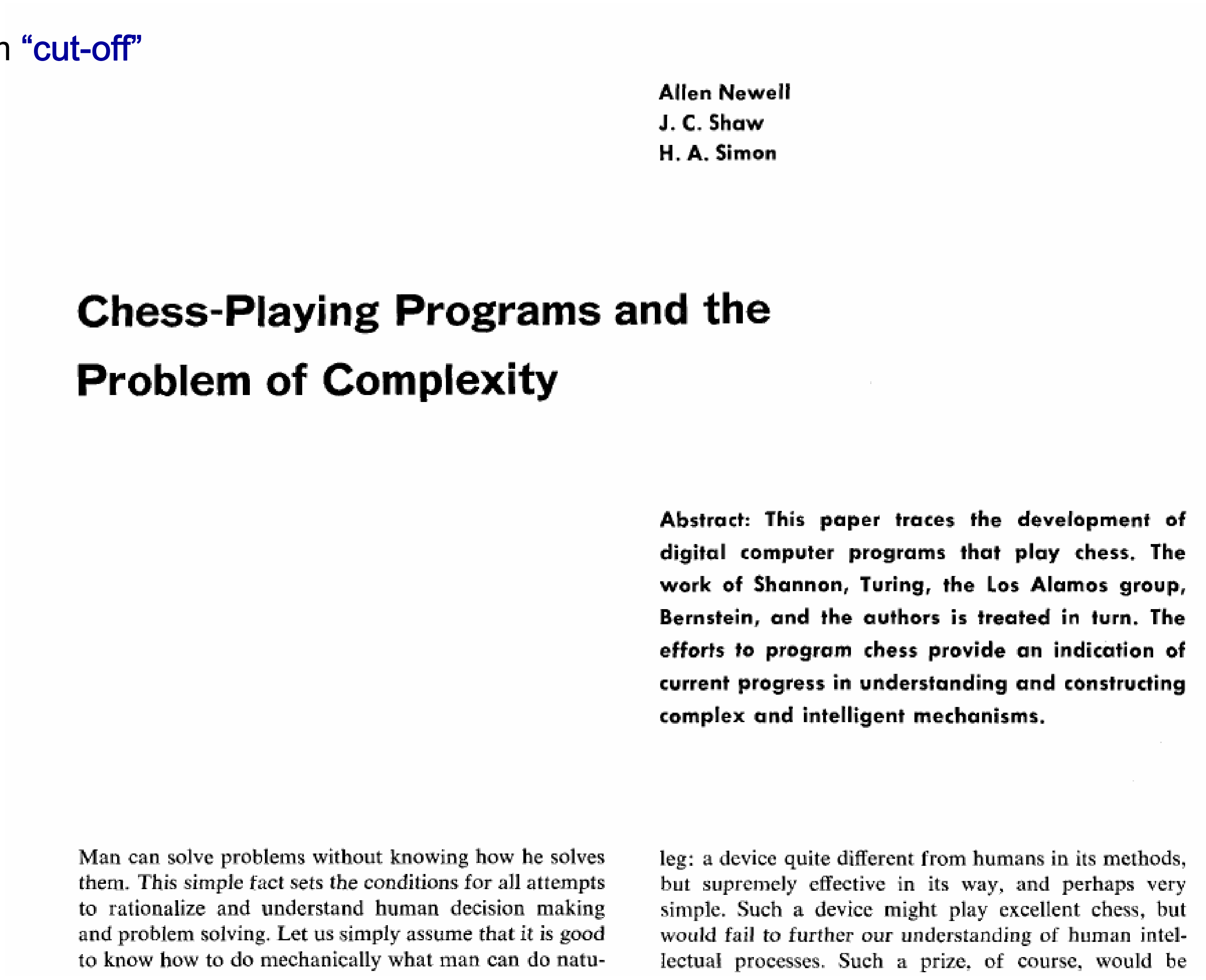
A general-purpose electronic computer is an extremely complicated device containing several thousand vacuum tubes, relays and other elements. The basic principles involved, however, are quite simple. The machine has four main parts: 1) an "arithmetic organ," 2) a control element, 3) a numerical memory and 4) a program memory. (In some designs the two memory functions are carried out in the same physical apparatus.) The manner of operation is exactly analogous to a human computer carrying out a series of numerical calculations with an ordinary desk computing machine. The arithmetic organ corresponds to the desk computing machine, the control element to the human operator, the numerical memory to the work sheet on which intermediate and final results are recorded, and the program memory to the computing routine describing the series of operations to be performed.

In an electronic computing machine, the numerical memory consists of a large

INEVITABLE ADVANTAGE of man over the machine is illustrated in this drawing. At top human player loses to machine. In center nettled human player revises machine's instructions. At bottom human player wins.

A. Newell, J.C. Shaw and H.A. Simon, Chess Playing Programs and the Problem of Complexity, *IBM J. of Research and Development 4*, 2 (1958), 320-335.

→ first used the term "cut-off"



Allen Newell
J. C. Shaw
H. A. Simon

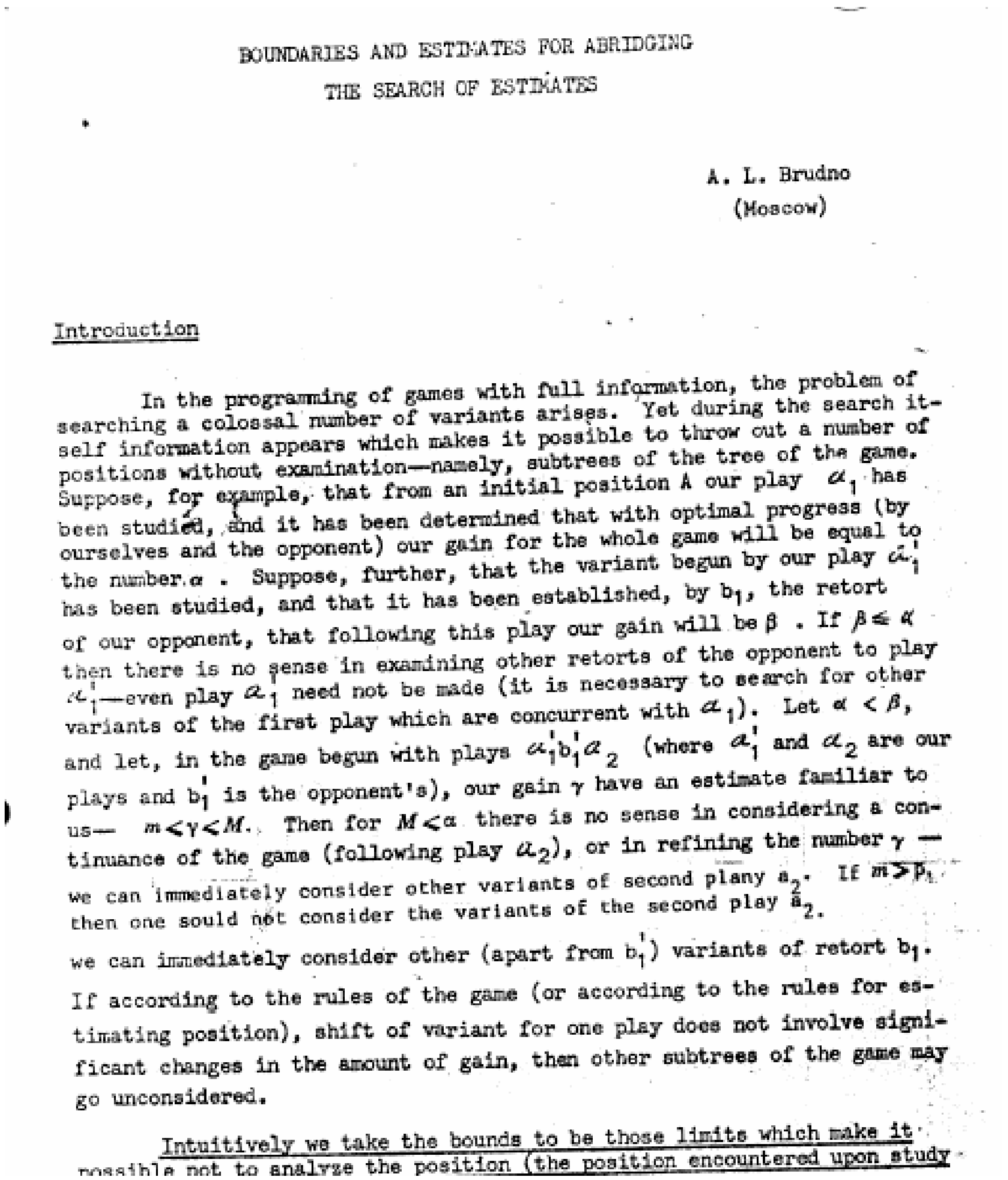
Chess-Playing Programs and the Problem of Complexity

Abstract: This paper traces the development of digital computer programs that play chess. The work of Shannon, Turing, the Los Alamos group, Bernstein, and the authors is treated in turn. The efforts to program chess provide an indication of current progress in understanding and constructing complex and intelligent mechanisms.

Man can solve problems without knowing how he solves them. This simple fact sets the conditions for all attempts to rationalize and understand human decision making and problem solving. Let us simply assume that it is good to know how to do mechanically what man can do natu-

leg: a device quite different from humans in its methods, but supremely effective in its way, and perhaps very simple. Such a device might play excellent chess, but would fail to further our understanding of human intellectual processes. Such a prize, of course, would be

A.L. Brudno, Bounds and Valuations for Abridging the Search of Estimates, *Problems of Cybernetics 10*, (1963), 225-241. Translation of Russian original in *Problemy Kibernetiki* 10, 141-150 (May 1963).



BOUNDARIES AND ESTIMATES FOR ABRIDGING
THE SEARCH OF ESTIMATES

A. L. Brudno
(Moscow)

Introduction

In the programming of games with full information, the problem of
searching a colossal number of variants arises. Yet during the search it-
self information appears which makes it possible to throw out a number of
positions without examination—namely, subtrees of the tree of the game.
Suppose, for example, that from an initial position A our play α_1 has
been studied, and it has been determined that with optimal progress (by
ourselves and the opponent) our gain for the whole game will be equal to
the number α. Suppose, further, that the variant begun by our play α_1'
has been studied, and that it has been established, by b_1, the retort
of our opponent, that following this play our gain will be β. If $\beta \leq \alpha$
then there is no sense in examining other retorts of the opponent to play
α_1'—even play α_1 need not be made (it is necessary to search for other
variants of the first play which are concurrent with α_1). Let $\alpha < \beta$,
and let, in the game begun with plays $\alpha_1' b_1' \alpha_2$ (where α_1' and α_2 are our
plays and b_1' is the opponent's), our gain γ have an estimate familiar to
us— $m < \gamma < M$. Then for $M < \alpha$ there is no sense in considering a con-
tinuance of the game (following play α_2), or in refining the number γ —
we can immediately consider other variants of second plany a_2. If $m > \beta_1$
then one sould not consider the variants of the second play a_2.
we can immediately consider other (apart from b_1') variants of retort b_1.
If according to the rules of the game (or according to the rules for es-
timating position), shift of variant for one play does not involve signi-
ficant changes in the amount of gain, then other subtrees of the game may
go unconsidered.

Intuitively we take the bounds to be those limits which make it
possible not to analyze the position (the position encountered upon study

submitted 31 May 1962

A.L. Samuel, Some Studies in Machine Learning Using the Game of Checkers, *IBM J. of Research and Development 3*, (1959), 210-229.

Paper on machine learning, coined the word “ply” – meaning a half move.

A.L. Samuel, Some Studies in Machine Learning Using the Game of Checkers II – Recent Progress, *IBM J. of Research and Development 11* (1967) 601-17.

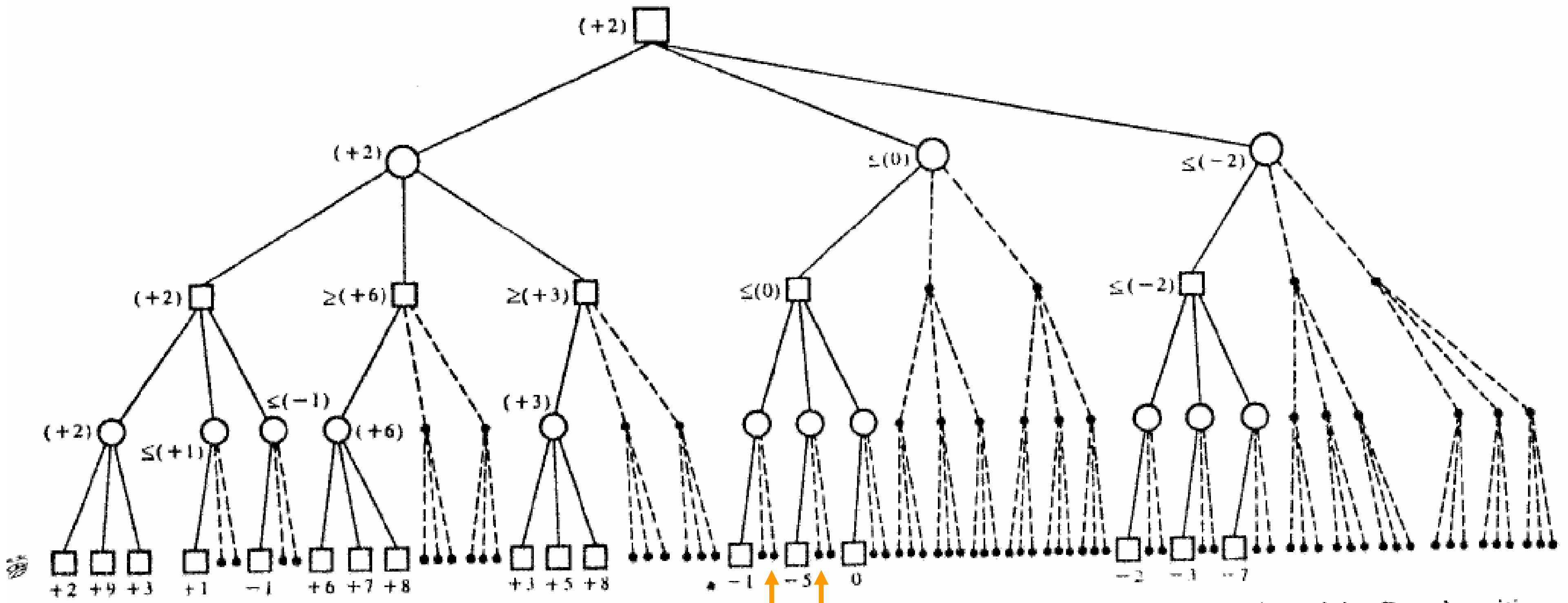


Figure 1 A (look-ahead) move tree in which alpha-beta pruning is fully effective if the tree is explored from left to right. Board positions for a look-ahead move by the first player are shown by squares, while board positions for the second player are shown by circles. The branches shown by dashed lines can be left unexplored without in any way influencing the final move choice.

D.E. Knuth and R.W. Moore, An Analysis of Alpha-beta Pruning, *Artificial Intelligence 6*, 4 (1975), 293-326.

this is a consequence of the theory developed below. On levels 4, 5, . . ., however, procedure *F*2 is occasionally able to make "deep cutoffs" which *F*1 is incapable of finding. A comparison of Fig. 3 with Fig. 2 shows that there are five deep cutoffs in this example.

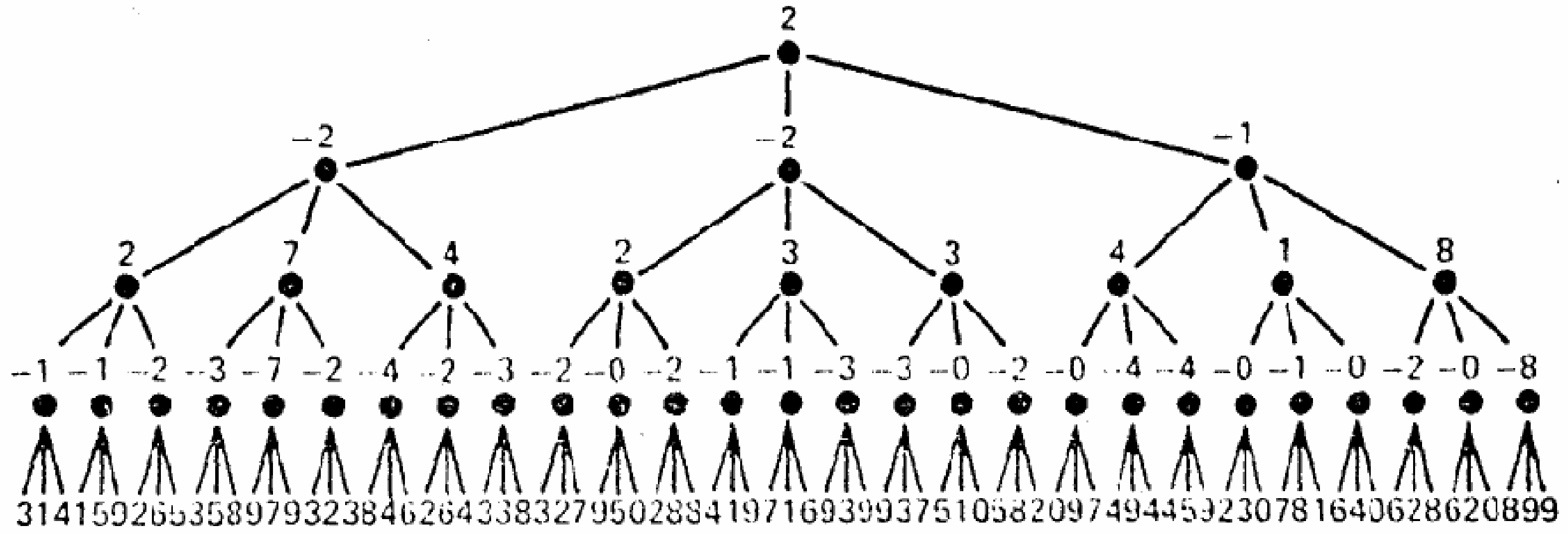


FIG. 1. Complete evaluation of a game tree.

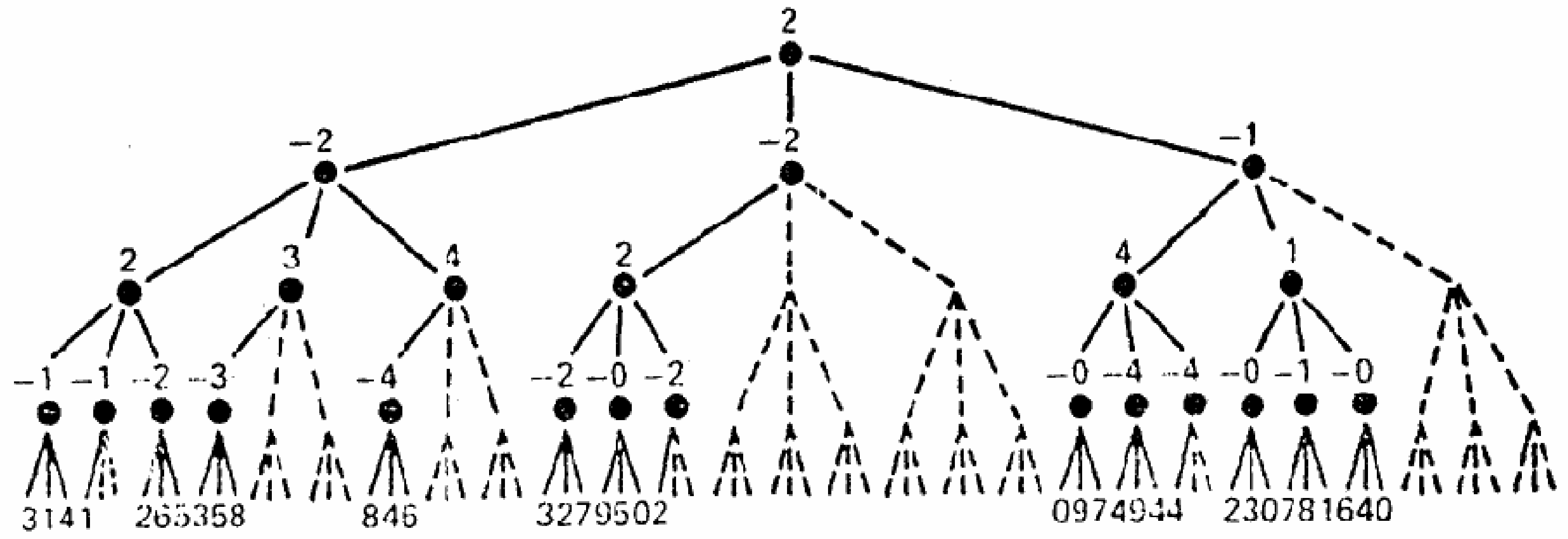


FIG. 2. The game tree of Fig. 1 evaluated with procedure *F*1 (branch-and-bound strategy).

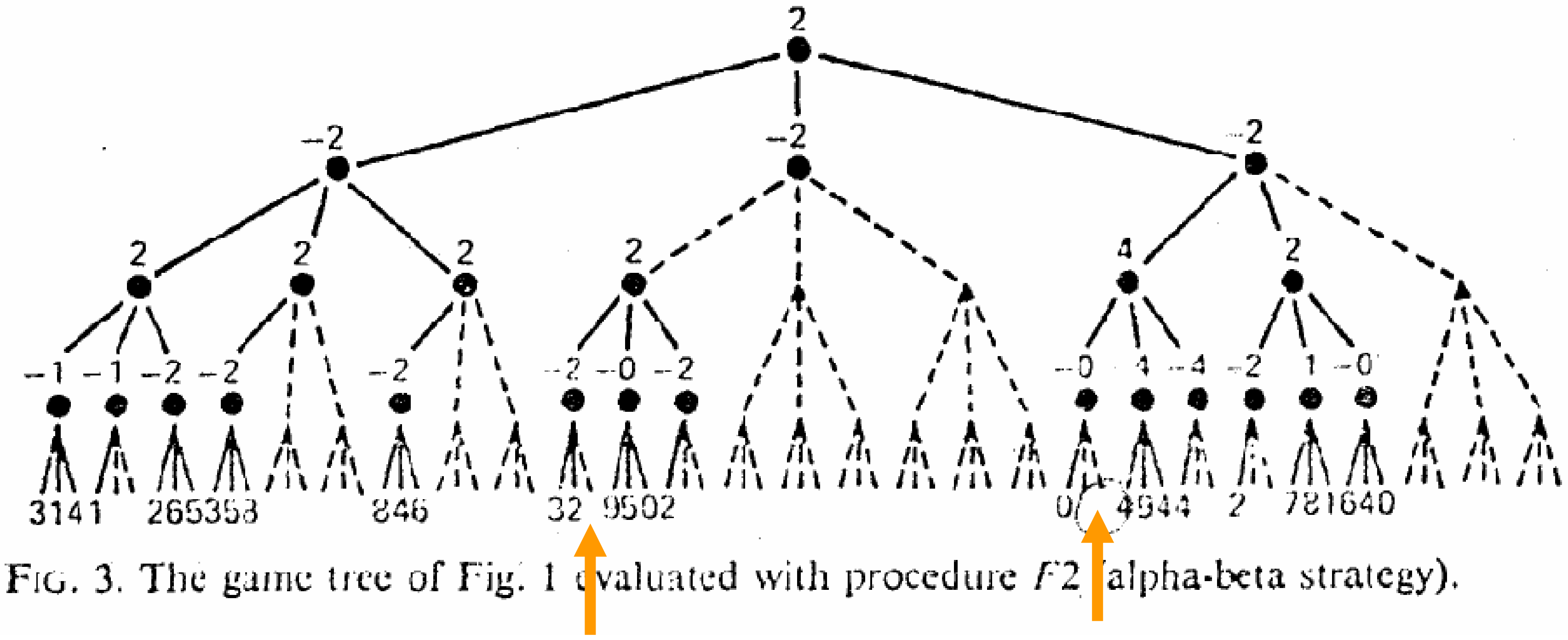
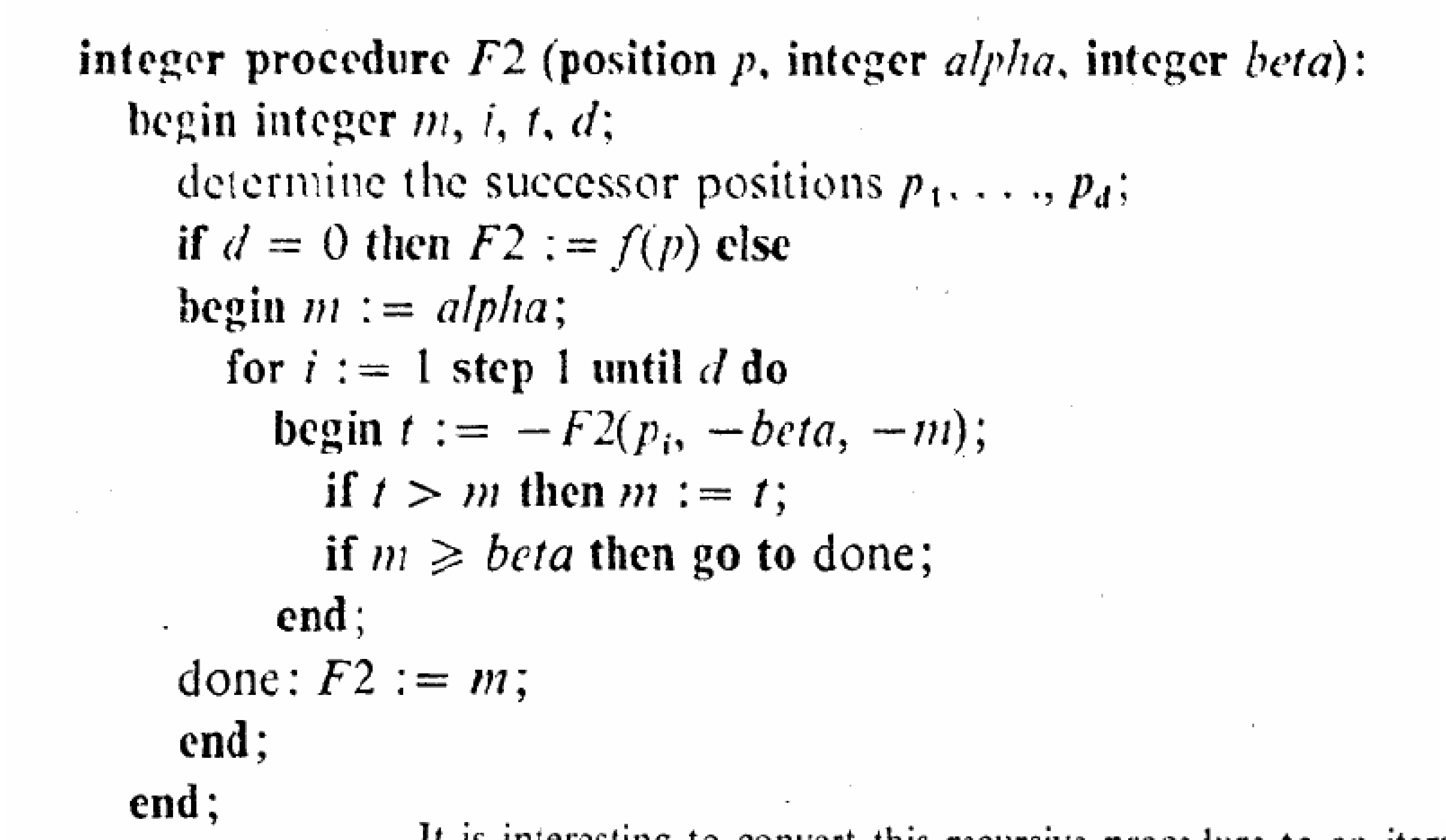
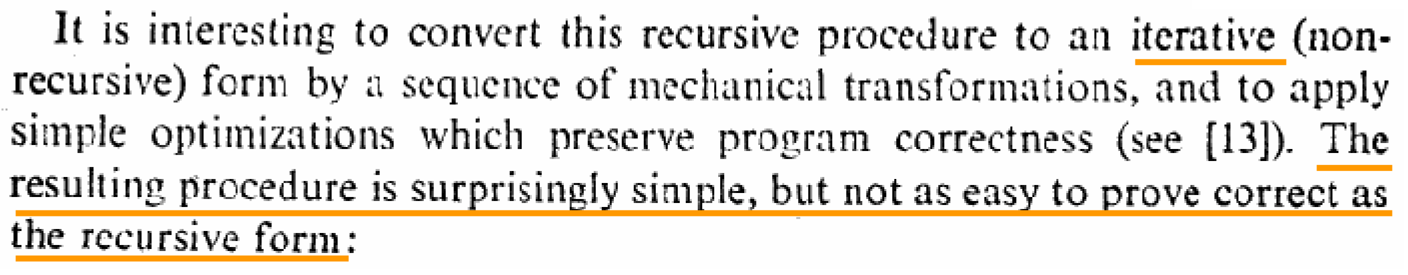


FIG. 3. The game tree of Fig. 1 evaluated with procedure *F*2 (alpha-beta strategy).

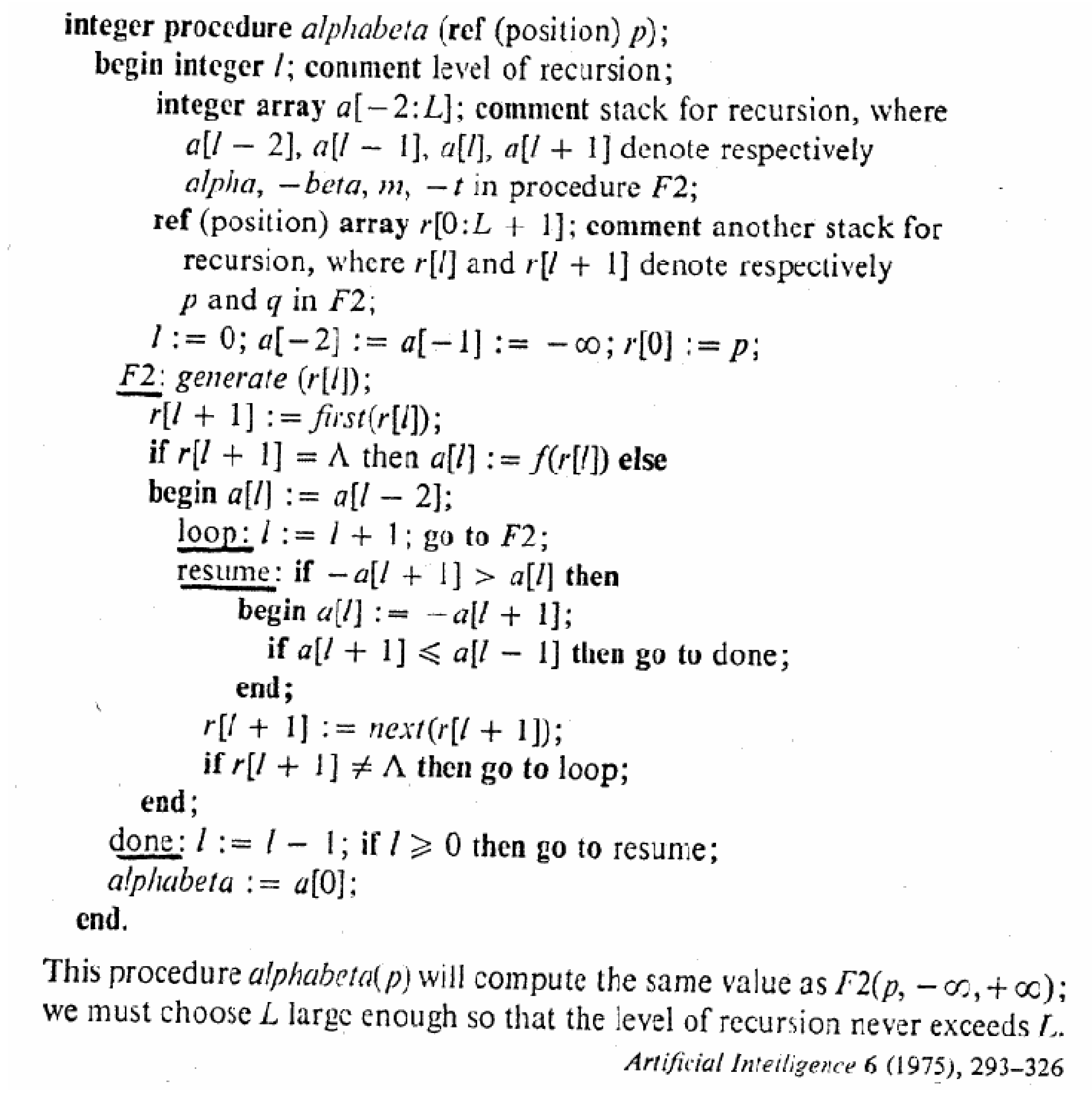
Knuth & Moore‘s Function F2 (aka AlphaBeta)



```
integer procedure F2 (position p, integer alpha, integer beta):
  begin integer m, i, t, d;
    determine the successor positions $p_1, \ldots, p_d$;
    if $d = 0$ then $F2 := f(p)$ else
    begin $m := alpha$;
      for $i := 1$ step 1 until $d$ do
        begin $t := -F2(p_i, -beta, -m)$;
          if $t > m$ then $m := t$;
          if $m \geqslant beta$ then go to done;
        end;
    done: $F2 := m$;
    end;
  end;
```



It is interesting to convert this recursive procedure to an iterative (non-recursive) form by a sequence of mechanical transformations, and to apply simple optimizations which preserve program correctness (see [13]). The resulting procedure is surprisingly simple, but not as easy to prove correct as the recursive form:

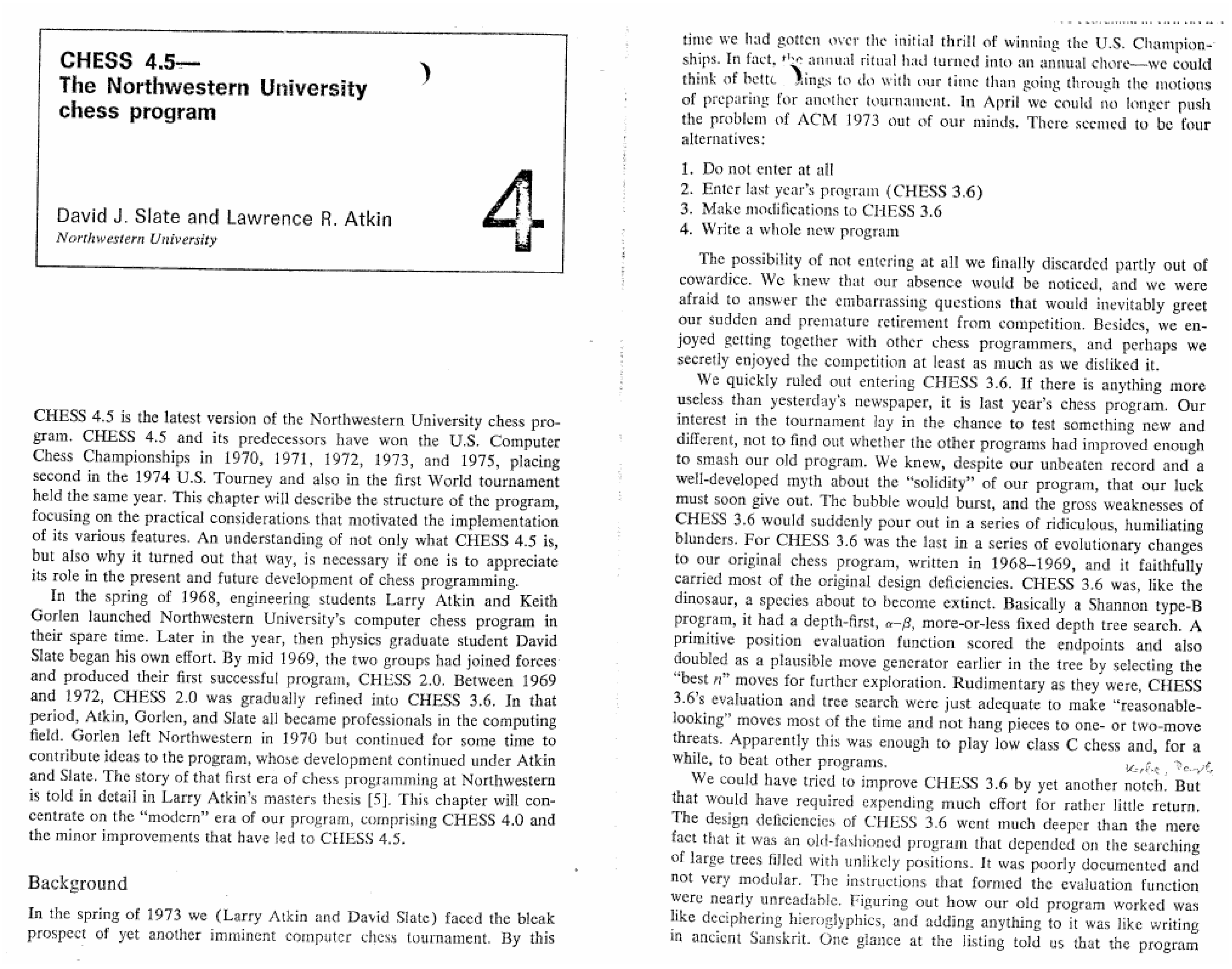


```
integer procedure alphabeta (ref (position) $p$);
  begin integer $l$; comment level of recursion;
      integer array $a[-2:L]$; comment stack for recursion, where
        $a[l-2]$, $a[l-1]$, $a[l]$, $a[l+1]$ denote respectively
        $alpha$, $-beta$, $m$, $-t$ in procedure $F2$;
      ref (position) array $r[0:L+1]$; comment another stack for
        recursion, where $r[l]$ and $r[l+1]$ denote respectively
        $p$ and $q$ in $F2$;
      $l := 0$; $a[-2] := a[-1] := -\infty$; $r[0] := p$;
    F2: generate $(r[l])$;
      $r[l+1] := first(r[l])$;
      if $r[l+1] = \Lambda$ then $a[l] := f(r[l])$ else
      begin $a[l] := a[l-2]$;
        loop: $l := l + 1$; go to $F2$;
        resume: if $-a[l+1] > a[l]$ then
            begin $a[l] := -a[l+1]$;
              if $a[l+1] \leqslant a[l-1]$ then go to done;
            end;
          $r[l+1] := next(r[l+1])$;
          if $r[l+1] \neq \Lambda$ then go to loop;
      end;
    done: $l := l - 1$; if $l \geqslant 0$ then go to resume;
    $alphabeta := a[0]$;
  end.
```

This procedure $alphabeta(p)$ will compute the same value as $F2(p, -\infty, +\infty)$; we must choose L large enough so that the level of recursion never exceeds L.

Artificial Intelligence 6 (1975), 293–326

D.J. Slate and L.R. Atkin, CHESS 4.5 - The Northwestern University Chess Program, in *Chess Skill in Man and Machine*, P. Frey (ed.), Springer-Verlag, 1977, 82-118.



CHESS 4.5— The Northwestern University chess program

4

David J. Slate and Lawrence R. Atkin
Northwestern University

CHESS 4.5 is the latest version of the Northwestern University chess program. CHESS 4.5 and its predecessors have won the U.S. Computer Chess Championships in 1970, 1971, 1972, 1973, and 1975, placing second in the 1974 U.S. Tourney and also in the first World tournament held the same year. This chapter will describe the structure of the program, focusing on the practical considerations that motivated the implementation of its various features. An understanding of not only what CHESS 4.5 is, but also why it turned out that way, is necessary if one is to appreciate its role in the present and future development of chess programming.

In the spring of 1968, engineering students Larry Atkin and Keith Gorlen launched Northwestern University's computer chess program in their spare time. Later in the year, then physics graduate student David Slate began his own effort. By mid 1969, the two groups had joined forces and produced their first successful program, CHESS 2.0. Between 1969 and 1972, CHESS 2.0 was gradually refined into CHESS 3.6. In that period, Atkin, Gorlen, and Slate all became professionals in the computing field. Gorlen left Northwestern in 1970 but continued for some time to contribute ideas to the program, whose development continued under Atkin and Slate. The story of that first era of chess programming at Northwestern is told in detail in Larry Atkin's masters thesis [5]. This chapter will concentrate on the "modern" era of our program, comprising CHESS 4.0 and the minor improvements that have led to CHESS 4.5.

Background

In the spring of 1973 we (Larry Atkin and David Slate) faced the bleak prospect of yet another imminent computer chess tournament. By this time we had gotten over the initial thrill of winning the U.S. Championships. In fact, the annual ritual had turned into an annual chore—we could think of bette[illegible]ings to do with our time than going through the motions of preparing for another tournament. In April we could no longer push the problem of ACM 1973 out of our minds. There seemed to be four alternatives:

1. Do not enter at all
2. Enter last year's program (CHESS 3.6)
3. Make modifications to CHESS 3.6
4. Write a whole new program

The possibility of not entering at all we finally discarded partly out of cowardice. We knew that our absence would be noticed, and we were afraid to answer the embarrassing questions that would inevitably greet our sudden and premature retirement from competition. Besides, we enjoyed getting together with other chess programmers, and perhaps we secretly enjoyed the competition at least as much as we disliked it.

We quickly ruled out entering CHESS 3.6. If there is anything more useless than yesterday's newspaper, it is last year's chess program. Our interest in the tournament lay in the chance to test something new and different, not to find out whether the other programs had improved enough to smash our old program. We knew, despite our unbeaten record and a well-developed myth about the "solidity" of our program, that our luck must soon give out. The bubble would burst, and the gross weaknesses of CHESS 3.6 would suddenly pour out in a series of ridiculous, humiliating blunders. For CHESS 3.6 was the last in a series of evolutionary changes to our original chess program, written in 1968–1969, and it faithfully carried most of the original design deficiencies. CHESS 3.6 was, like the dinosaur, a species about to become extinct. Basically a Shannon type-B program, it had a depth-first, α–β, more-or-less fixed depth tree search. A primitive position evaluation function scored the endpoints and also doubled as a plausible move generator earlier in the tree by selecting the "best n" moves for further exploration. Rudimentary as they were, CHESS 3.6's evaluation and tree search were just adequate to make "reasonable-looking" moves most of the time and not hang pieces to one- or two-move threats. Apparently this was enough to play low class C chess and, for a while, to beat other programs. [illegible]

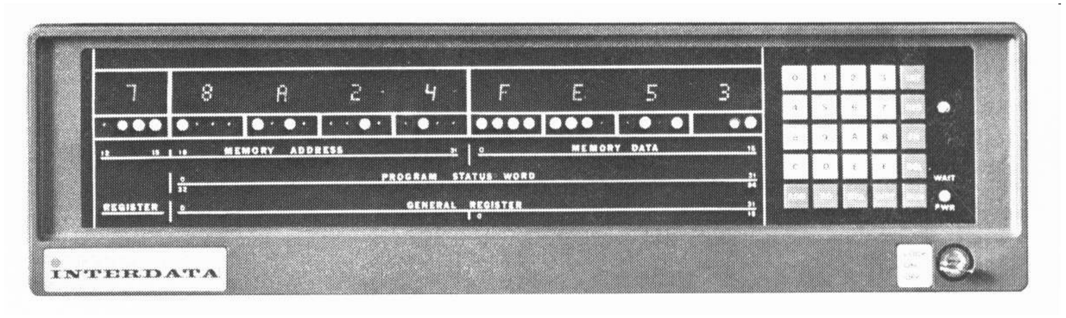
We could have tried to improve CHESS 3.6 by yet another notch. But that would have required expending much effort for rather little return. The design deficiencies of CHESS 3.6 went much deeper than the mere fact that it was an old-fashioned program that depended on the searching of large trees filled with unlikely positions. It was poorly documented and not very modular. The instructions that formed the evaluation function were nearly unreadable. Figuring out how our old program worked was like deciphering hieroglyphics, and adding anything to it was like writing in ancient Sanskrit. One glance at the listing told us that the program

P.W. Frey and L.R. Atkin, Creating a Chess Player, in *The BYTE Book of Pascal*, B.L. Liffick (ed.), BYTE/McGraw-Hill, Peterborough NH, 2nd Edition 1979, 107-155.

- published the complete code of a simple chess program “MicroChess”
  - move generator
  - evaluation function
  - search algorithm (alphabeta)
  - bit boards & hash tables (transposition & refutation, killer heuristic)
  - ...

Chess Program MURKS (1979)

- At the University of Hamburg, I got access to an **Interdata M85**

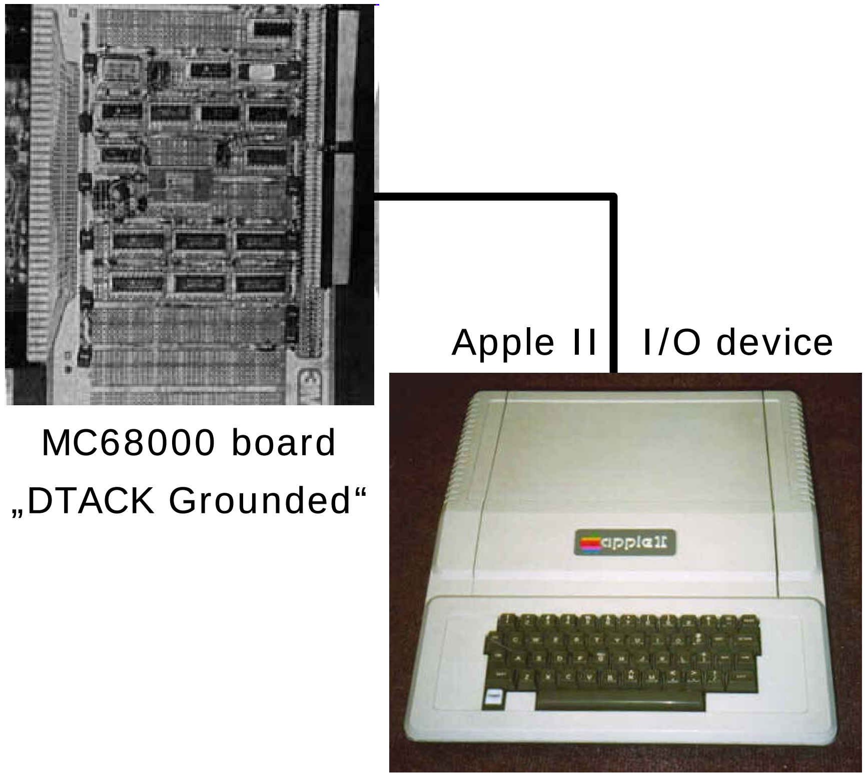


- designed the chess program “**MURKS”**, partly implemented in microcode

- world chess champion **Mikhail Botvinnik** visited us and played against MURKS

“MicroMurks” (1980)

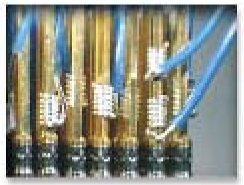
- ported chess program to Motorola MC68000 („**MicroMurks**“)
- *1st World Micro-Computer Chess Championship*, London
- terrible hardware failure in I/O, winner: Fidelity Chess Challenger X (Spracklen)



Building a Computer for MicroMurks (1981)

Built our own MC68000 microcomputer from scratch (with 4 students)

- own chips
- own system design
- own VLSI layout
- own wire wrapping $\longrightarrow$
- own operating system
- own (assembler) chess program



participated 2nd World Micro-Computer Chess Championship, Travemünde

World Computer Chess Champions (WCCC)

#	Year	Place	Winner	Developer(s)	Country
1	1974	Stockholm	Kaissa	Donskoy	USSR
2	1977	Toronto	Chess 4.6	Slate/Atkin	USA
3	1980	Linz	Belle	Thompson	USA
4	1983	New York	Cray Blitz	Hyatt	USA
5	1986	Cologne	Cray Blitz	Hyatt	USA
6	1989	Edmonton	Deep Thought	Hsu	USA
7	1992	Madrid	Chessmachine	Schroeder	Netherlands
8	1995	Hong Kong	Fritz	Morsch/de Gorter/Feist	Germany/Netherlands
9	1999	Paderborn	Shredder	Meyer-Kahlen	Germany
10	2002	Maastricht	Junior	Ban/Bushinsky	Israel
11	2003	Graz	Shredder	Meyer-Kahlen	Germany
12	2004	Ramat-Gan	Junior	Ban/Bushinsky	Israel

World Microcomputer Chess Champions (WMCCC)

1	1980	London	Chess Challenger	Spracklen USA
2	1981	Travemünde	Fidelity X	Spracklen USA
3	1983	Budapest	Elite A/S	Spracklen USA
4	1984	Glasgow	Elite X	Spracklen USA
5	1985	Amsterdam	Mephisto	Lang UK
6	1986	Dallas	Mephisto	Lang UK
7	1987	Rome	Mephisto	Lang UK
8	1988	Almeria	Mephisto	Lang UK
9	1989	Portoroz	Mephisto	Lang UK
10	1990	Lyon	Mephisto	Lang UK
11	1991	Vancouver	Gideon	Schroeder Netherlands
12	1993	Munich	Hiarcs	Uniacke UK
13	1995	Paderborn	MChess-Pro 5.0	Hirsch USA
14	1996	Jakarta	Shredder	Meyer-Kahlen Germany
15	1997	Paris	Junior	Ban/Bushinsky Israel
16	1999	(same as 9th WCCC, Paderborn.)		
17	2000	London	Shredder	Meyer-Kahlen Germany
18	2001	Maastricht	Deep Junior	Ban/Bushinsky Israel

Back to Search Algorithms:

Alphabeta is simple, elegant and efficient.
Can we do better?

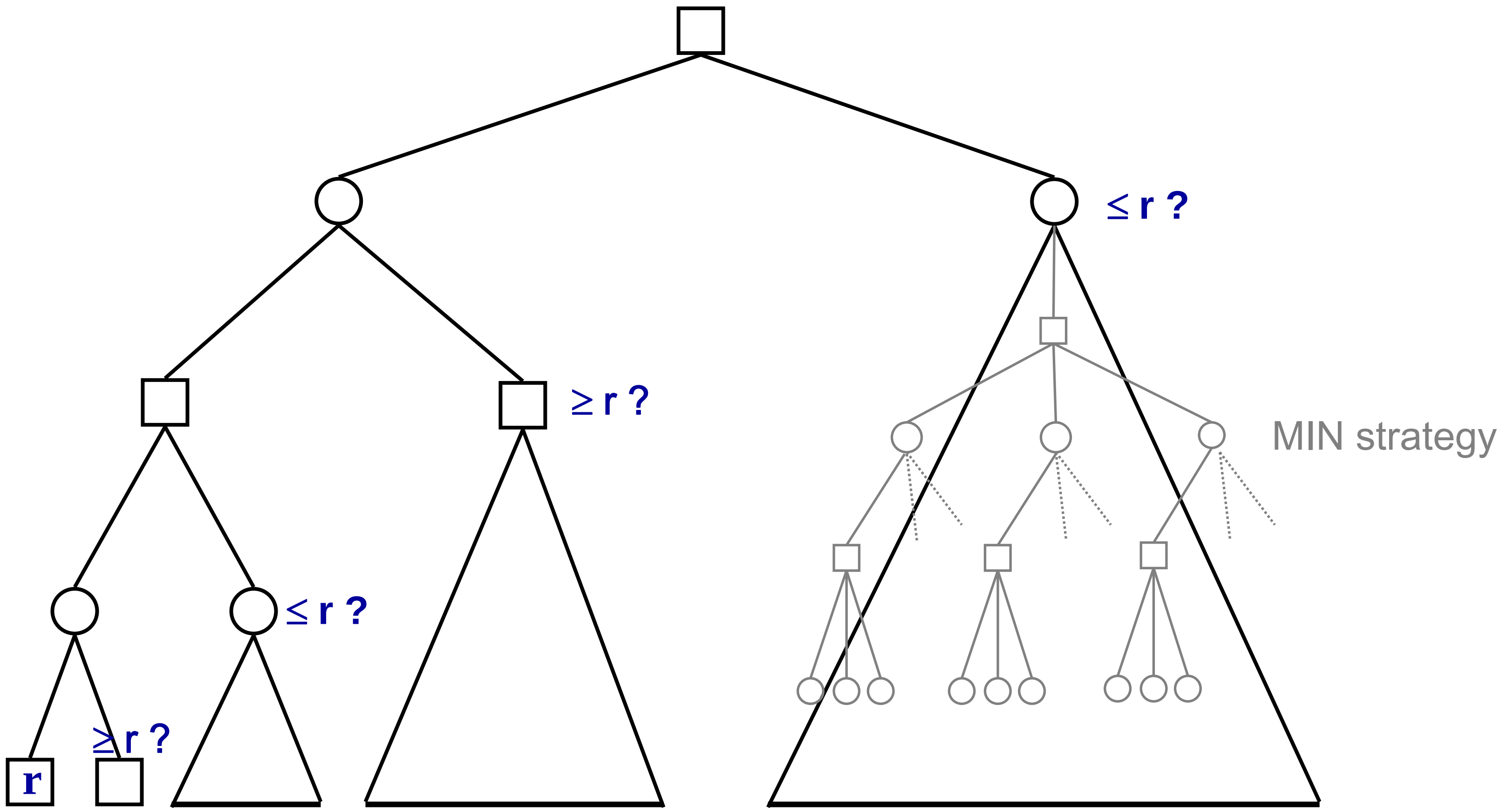
The Scout Algorithm: Test and, if necessary, evaluate.

J. Pearl, 1980

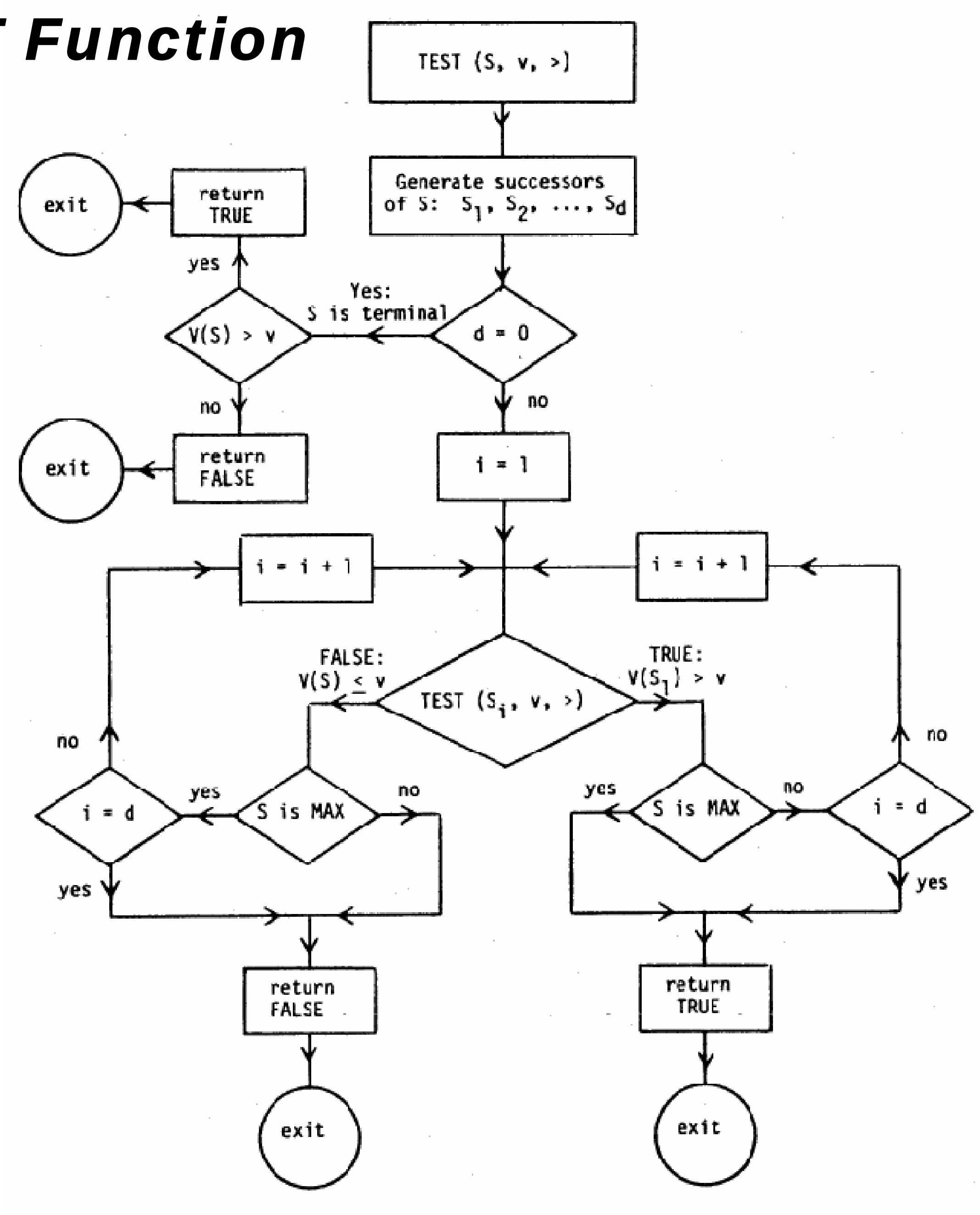
Perform a boolean test. If positive, we are done.

If negative, we found a better minimax value.

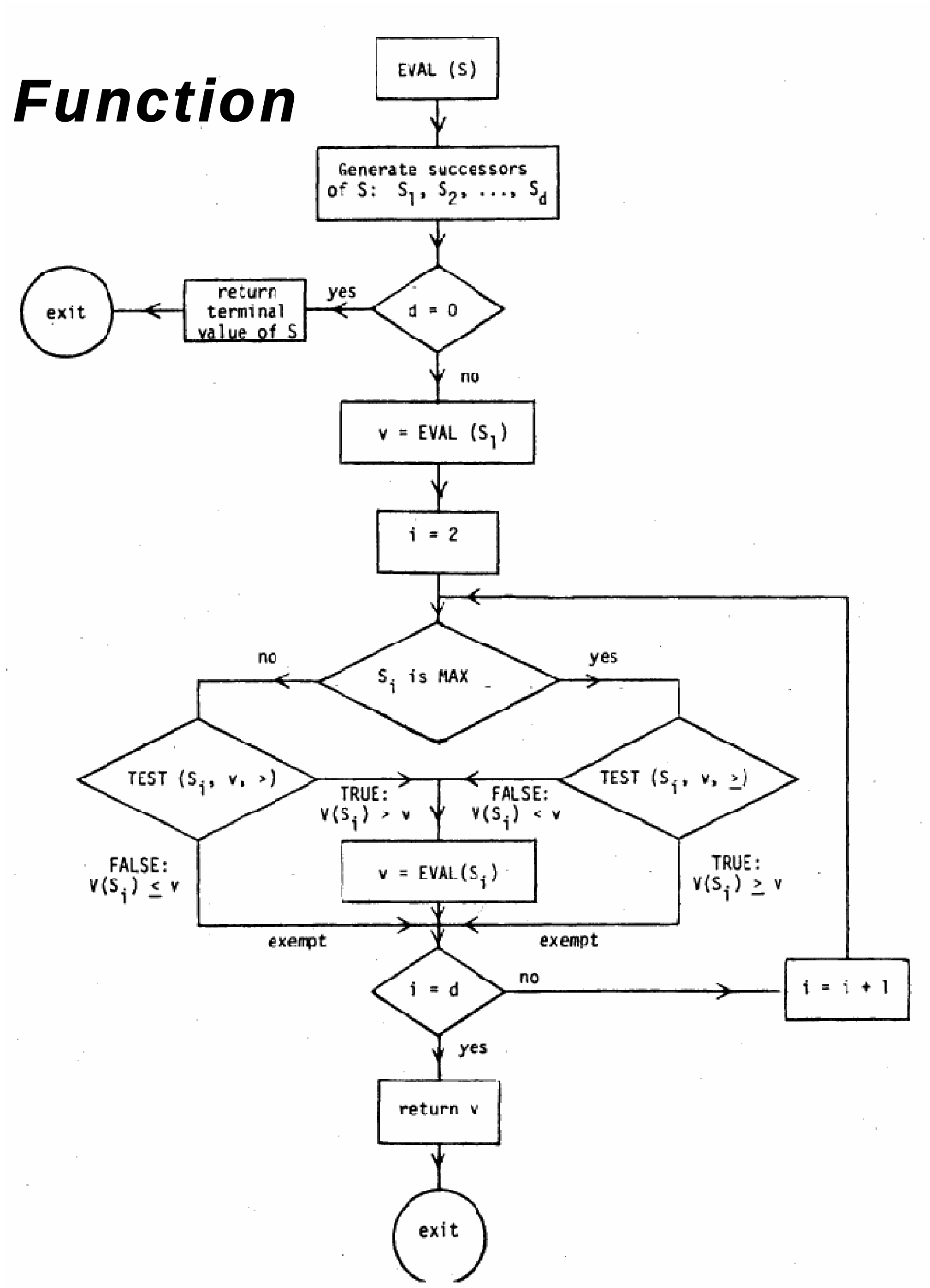
Need to evaluate, i.e re-visit the subtree.



Scout‘s TEST Function

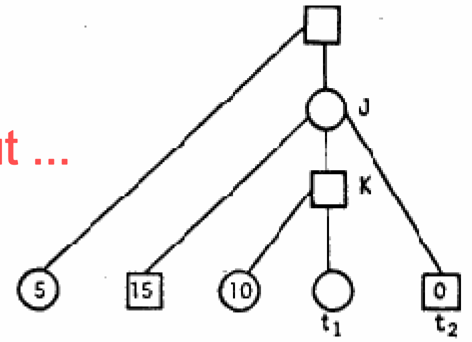


Scout‘s EVAL Function



extra caution in testing prior to evaluation may sometimes pay off, causing it t skip nodes which would be visited by α–β. In the diagram below, the nod marked t_1 would be examined by the α–β procedure but ignored by SCOUT

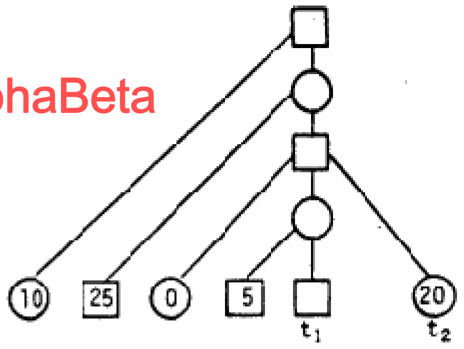
AlphaBeta does not dominate Scout ...



When J is submitted to the test TEST(J, 5, >), the zero value assigned to nod t_2 causes the test to fail, whereas during the TEST(K, 5, >) phase, t_1 is skippe by virtue of its elder sibling having the value 10. α–β, on the other hand, ha no way of finding out the low value of t_2 before t_1 is examined.

The converse situation can, of course, also be demonstrated. The diagra below shows how a node (t_1) which is visited by SCOUT is cut off by α–β However, the asymptotic performance of SCOUT is at least as good as that o

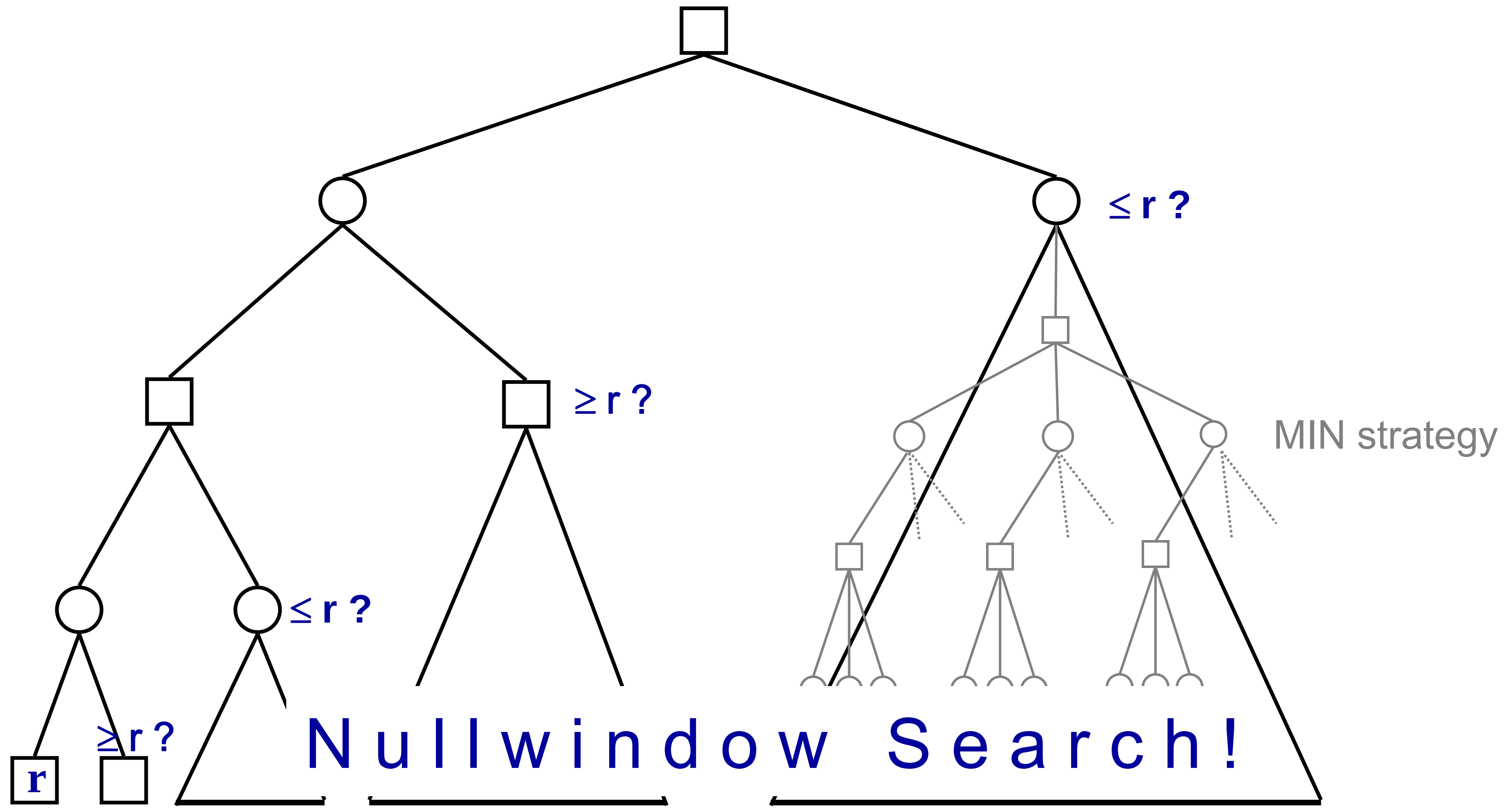
... and Scout does not dominate AlphaBeta



The NegaScout Algorithm

A. Reinefeld, 1982

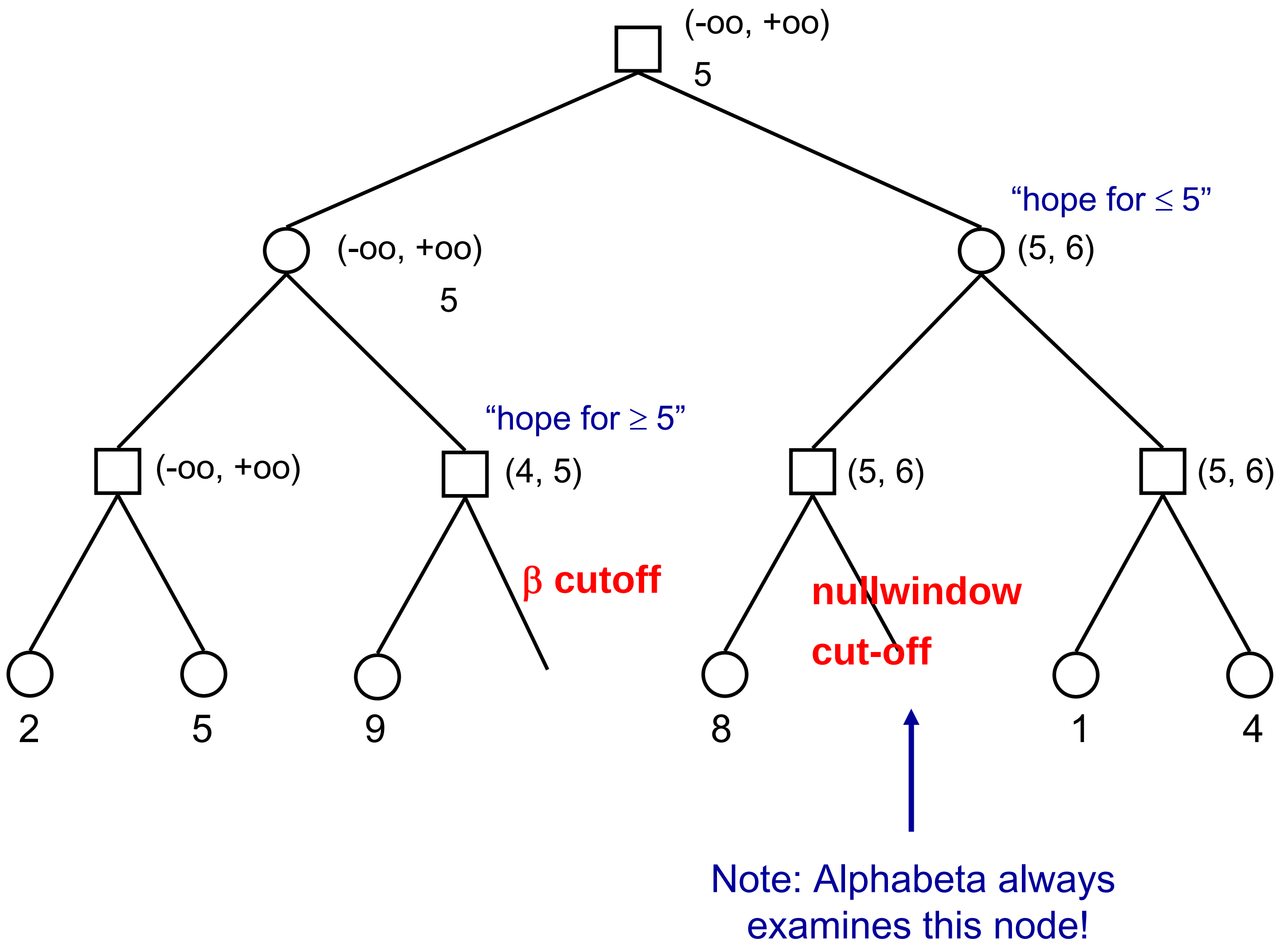
Perform a „test“ with a narrow search window.
If positive, we are done.
If negative, re-visit subtree with open window.



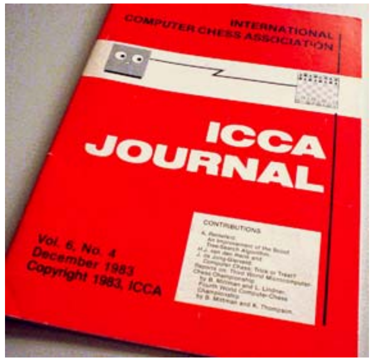
```
int NegaScout ( position p; int α, β );
{                                   /* compute minimax value of position p */
        int a, b, t, i;
        determine successors p_1,...,p_w of p;
        if ( w == 0 )
                return ( Evaluate(p) );                  /* leaf node */
        a = α;
        b = β;
        for ( i = 1; i <= w; i++ ) {
                t = -NegaScout ( p_i, -b, -a );
                if ( t > a && t < β && i > 1 && d < maxdepth-1 )
                        a = -NegaScout ( p_i, -β, -t );    /* re-search */
                a = max( a, t );
                if ( a >= β )
                        return ( a );                       /* cut-off */
                b = a + 1;                       /* set new null window */
        }
        return ( a );
}
```

```
int AlphaBeta ( position p; int α, β );
{                              /* compute minimax value of position p */
        int a, t, i;
        determine successors p_1,...,p_w of p;
        if ( w == 0 )
                return ( Evaluate(p) );                  /* leaf node */
        a = α;

        for ( i = 1; i <= w; i++ ) {
                t = -AlphaBeta ( p_i, - β, -a );



                a = max( a, t );
                if ( a >= β )
                        return ( a );                    /* cut-off */


        }
        return ( a );
}
```

NegaScout Example



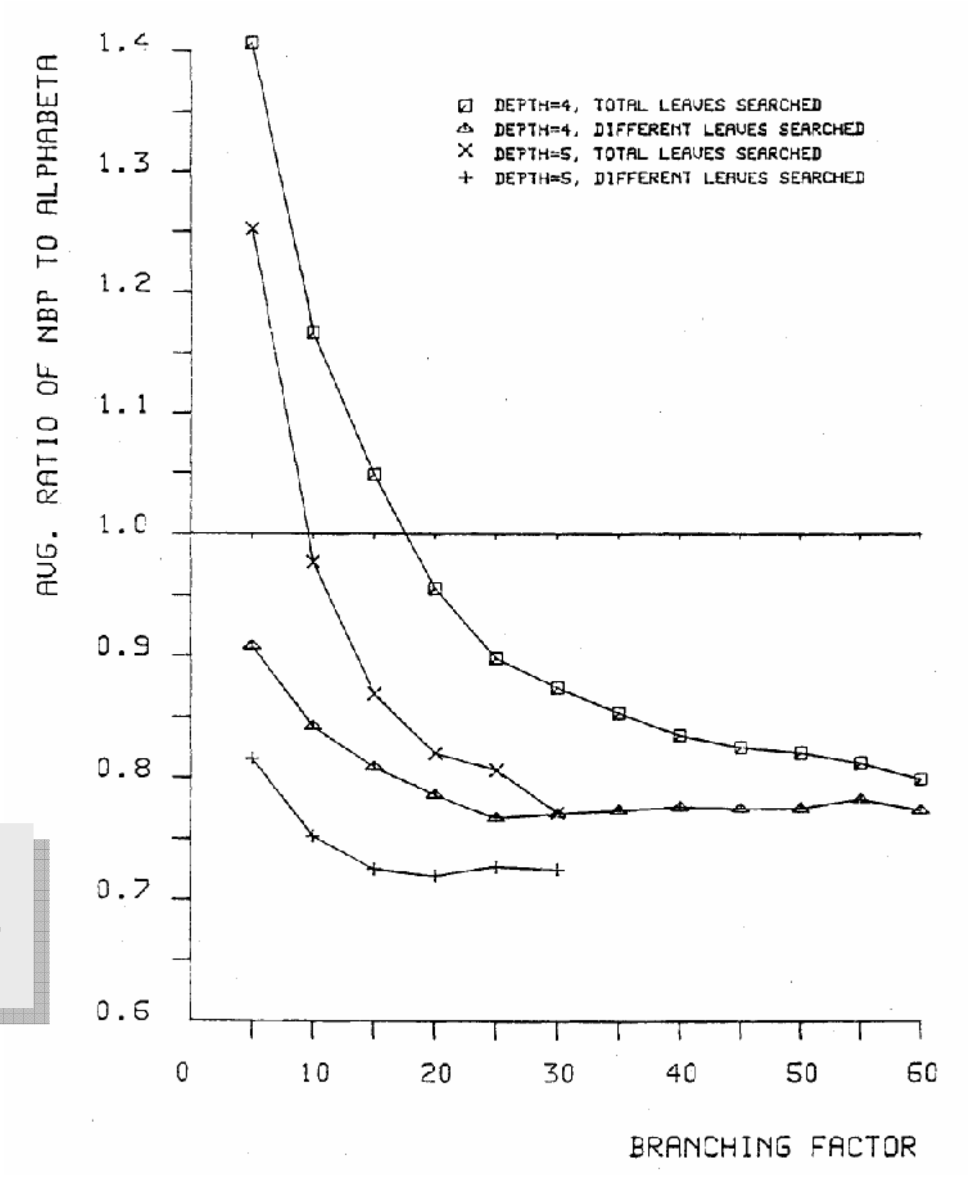
NegaScout vs. Alphabeta



A. Reinefeld,

An Improvement of the Scout Tree-Search Algorithm,

Int. Computer Chess Assoc. J. 6, 4 (1983), 4-14.



Is Best First Search better?

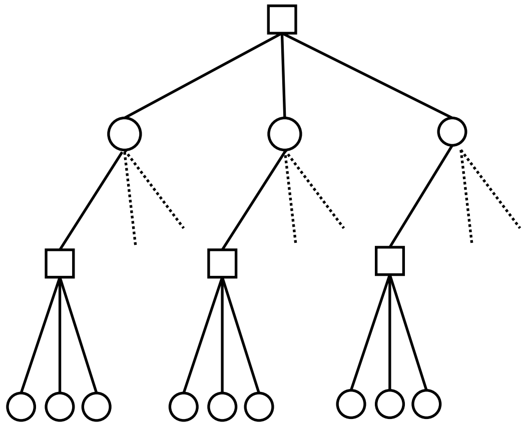
SSS* = State Space Search

- SSS* is a best first search.
- SSS* maintains an **OPEN list** with descriptors of the active nodes.
- Descriptors are sorted in decreasing order of their *h* values.
- A **descriptor *(n, s, h)*** consists of
  - a node identifier *n*
  - a status $s \in$ { LIVE, SOLVED }
    - LIVE: *n* is still unexpanded and *h* is an upper bound on the true value
    - SOLVED: *h* is the true value
  - a merit *h*

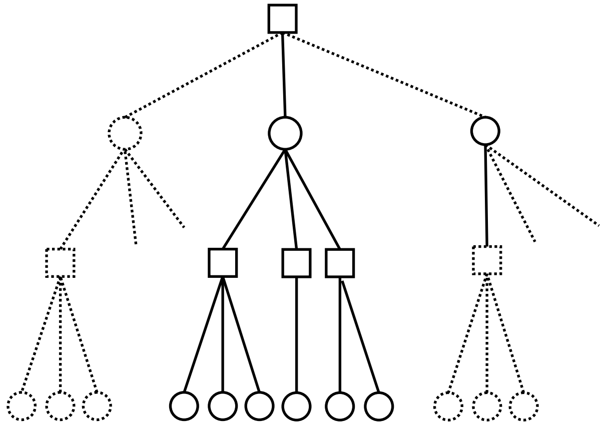
G.C. Stockman:
A minimax algorithm better than alpha-beta?
AIJ 12,2 (1979).

SSS's Two Search Phases*

1. Node Expansion Phase: Top down expansion of a **MIN strategy**.



2. Solution Phase: Bottom up search for the **best MAX strategy**.



SSS switches back and forth between the phases!*

```
int SSS* (node n; int bound);
{
   push (n, LIVE, bound);
   while ( 1 ) {
        pop (node);
        case node.status of
        LIVE:
               if (node == LEAF)
                      insert (node, SOLVED, min(eval(node),h));
               if (node == MIN-NODE)
                      push (node.1, LIVE, h);
               if (node == MAX-NODE)
                      for (i=w; 1; j--)
                             push (node.j, LIVE, h);
        SOLVED:
               if (node == ROOT-NODE)
                      return (h);
               if (node == MIN-NODE) {
                      purge (parent(node));
                      push (parent(node), SOLVED, h);
               }
               if (node == MAX-NODE)
                      if (node has an unexamined brother)
                             push (brother(node), LIVE, h);
                      else push (parent(node); SOLVED, h);
   }
}
```

SSS is too complex and too slow!*

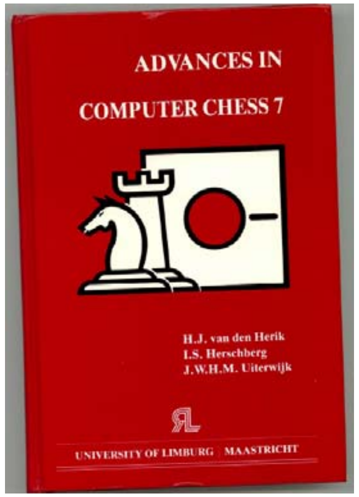
1. In each step, the node with the maximum *h*-value is removed from OPEN.
2. Whenever an interior MAX-node gets *SOLVED*, all direct and indirect descendents must be **purged** from OPEN

   These two steps alone take 90% of the CPU time!

“The meager improvement in the pruning power of SSS is more than offset by the increased storage space and bookkeeping (e.g. sorting OPEN) that it requires. One can safely speculate therefore that alphabeta will continue to monopolize the practice of computerized game playing.”*
[J. Pearl 1984]

I. Roizen, J. Pearl:
A minimax algorithm better than alpha-beta? Yes and No.
AIJ 21,1 (1983).

*A Minimax Algorithm **Faster** than Alphabeta*



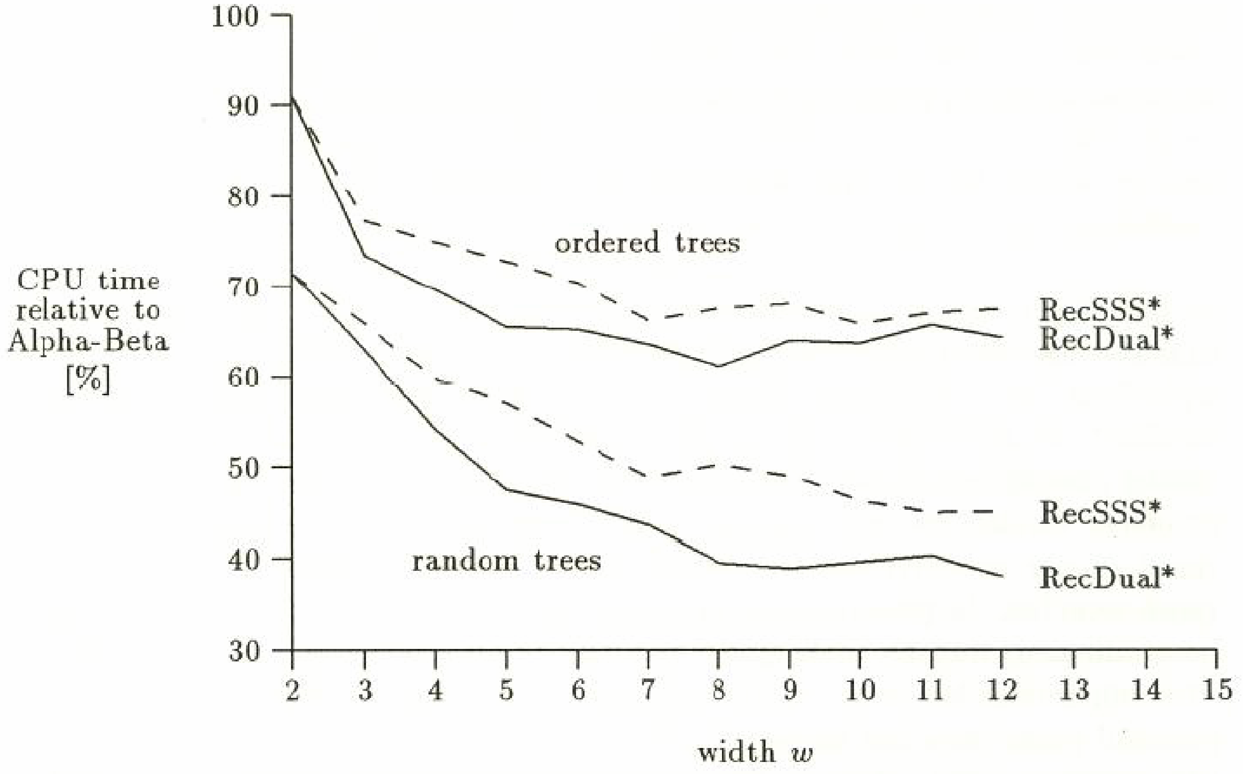


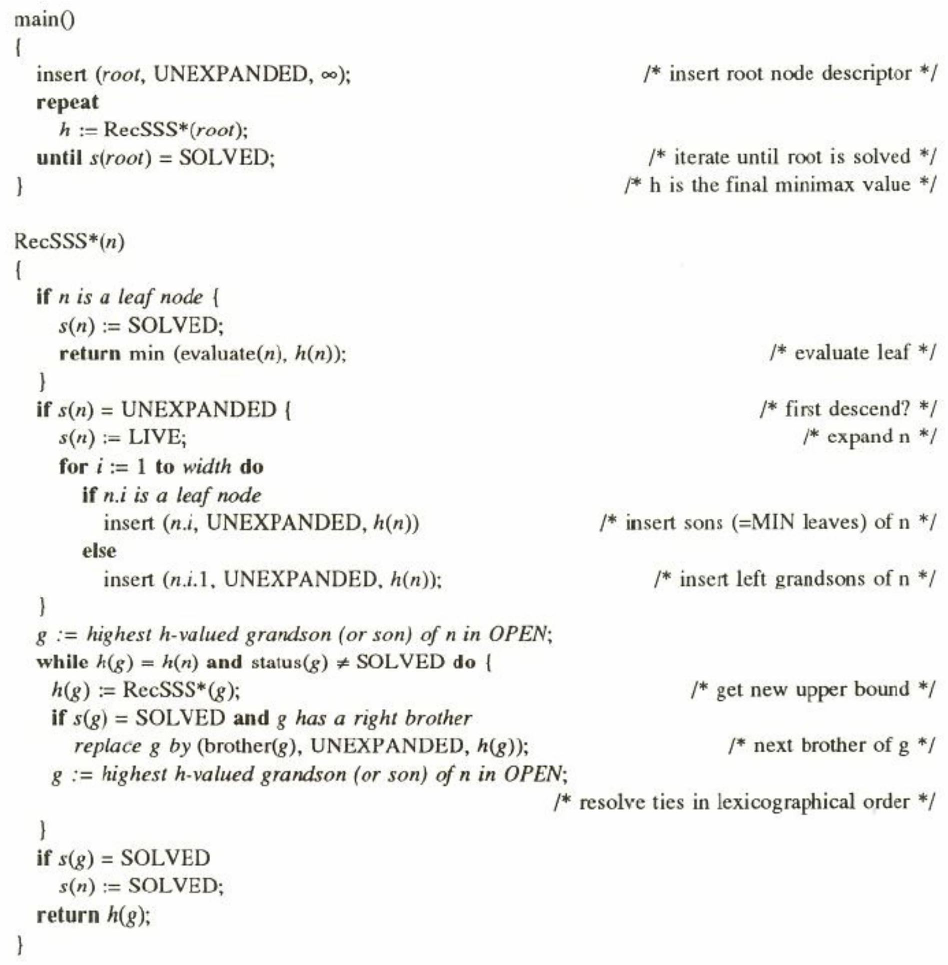
Figure 6: Performance in some trees of depth 8.

A. Reinefeld:
A minimax algorithm faster than alpha-beta.
Advances in Computer Chess 7, (1994).

A Minimax Algorithm Faster than Alphabeta

We devised a best first search that is

- recursive
- has no pointers
- needs no explicit state removal from the OPEN list
- **is always faster than Alphabeta!**



```
main()
{
  insert (root, UNEXPANDED, ∞);                         /* insert root node descriptor */
  repeat
    h := RecSSS*(root);
  until s(root) = SOLVED;                               /* iterate until root is solved */
}                                                       /* h is the final minimax value */

RecSSS*(n)
{
  if n is a leaf node {
    s(n) := SOLVED;
    return min (evaluate(n), h(n));                     /* evaluate leaf */
  }
  if s(n) = UNEXPANDED {                                /* first descend? */
    s(n) := LIVE;                                       /* expand n */
    for i := 1 to width do
      if n.i is a leaf node
        insert (n.i, UNEXPANDED, h(n))                  /* insert sons (=MIN leaves) of n */
      else
        insert (n.i.1, UNEXPANDED, h(n));               /* insert left grandsons of n */
  }
  g := highest h-valued grandson (or son) of n in OPEN;
  while h(g) = h(n) and status(g) ≠ SOLVED do {
   h(g) := RecSSS*(g);                                  /* get new upper bound */
   if s(g) = SOLVED and g has a right brother
     replace g by (brother(g), UNEXPANDED, h(g));       /* next brother of g */
   g := highest h-valued grandson (or son) of n in OPEN;
                                                        /* resolve ties in lexicographical order */
  }
  if s(g) = SOLVED
    s(n) := SOLVED;
  return h(g);
}
```

Figure 2: RecSSS*.

Analysis

Best & Worst Case

Let I_A be the number of leaf node evaluations of search algorithm A. Then, for all search algorithms A:

$$w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} \leq I_A \leq w^d$$

We want the „best“ algorithm with minimum $I_A = f(w, d, p)$ for given w, d, p

Example w=30, d=12

best case: 1.458.000.000

worst case: 531.441.000.000.000.000 leaves

Assessing Tree Search Algorithms

1. Use empirical results from typical game trees, e.g. „**p-ordered trees**“.
   - not general

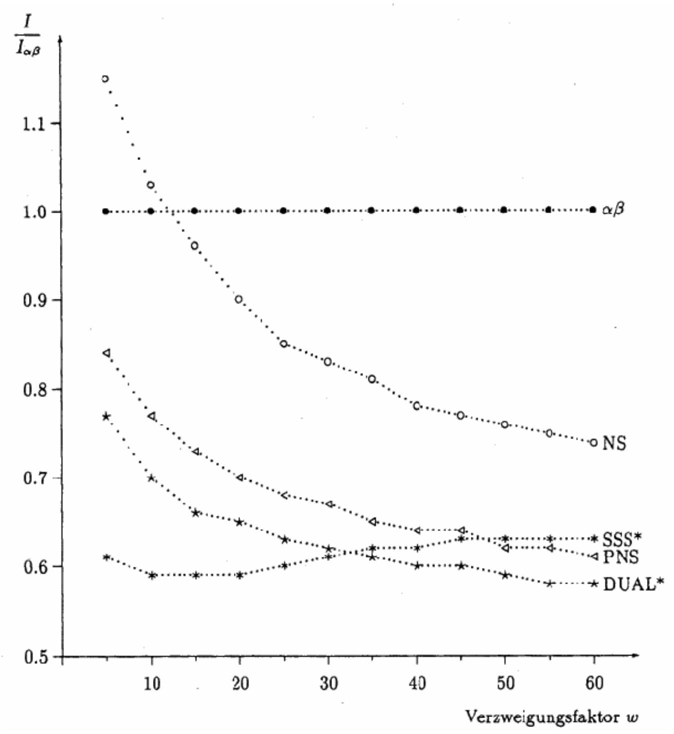
2. Analyze all possible **leaf node permutations**.
   - not feasible for large trees

3. Derive **dominance relations.**
   - helpful for simple cases

(1) Empirical Assessment

PARAMETER SPACE

- tree depth
- tree width
- uniform tree
- Leaf values
    - uniform distribution?
    - position dependant?



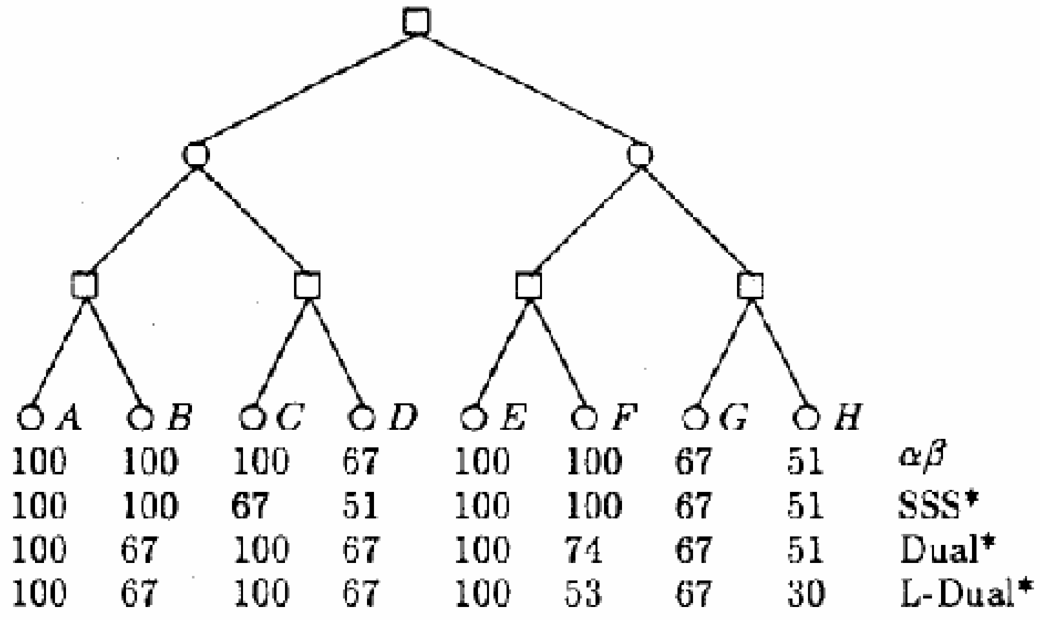
Methods for generating leaf values

- table with random values (access mod tab_size)
- store all values
- p-ordered values: determine w weight factors, apply function in each node

(2) All Leaf Value Permutations

Probability of leaf evaluation in trees of w=2, d=3 with **integer** leaf values.

All leaf values are different, resulting in 8! = 40320 trees.



A	B	C	D	E	F	G	H	
100	100	100	67	100	100	67	51	$\alpha\beta$
100	100	67	51	100	100	67	51	SSS*
100	67	100	67	100	74	67	51	Dual*
100	67	100	67	100	53	67	30	L-Dual*

Open questions:

- How about leaves with the same value?
- How about larger trees?

(3) Dominance Relations

Establish conditions, under which a node must be expanded.

Some results:

- SSS* dominates Alphabeta. [Stockman 1979]
- NegaScout dominates AlphaBeta. [Reinefeld 1983]
- DUAL* dominates Alphabeta. [Reinefeld 1986]
- In inferior root subtrees, DUAL* and NegaScout expand the minimal number of nodes. [Reinefeld 1986]
    - Corollary: DUAL* and NegaScout dominate SSS* in inferior root subtrees.
- Neither does NegaScout dominate SSS* nor vice versa: There exist trees in which NegaScout searches strictly fewer nodes than SSS* and vice versa.

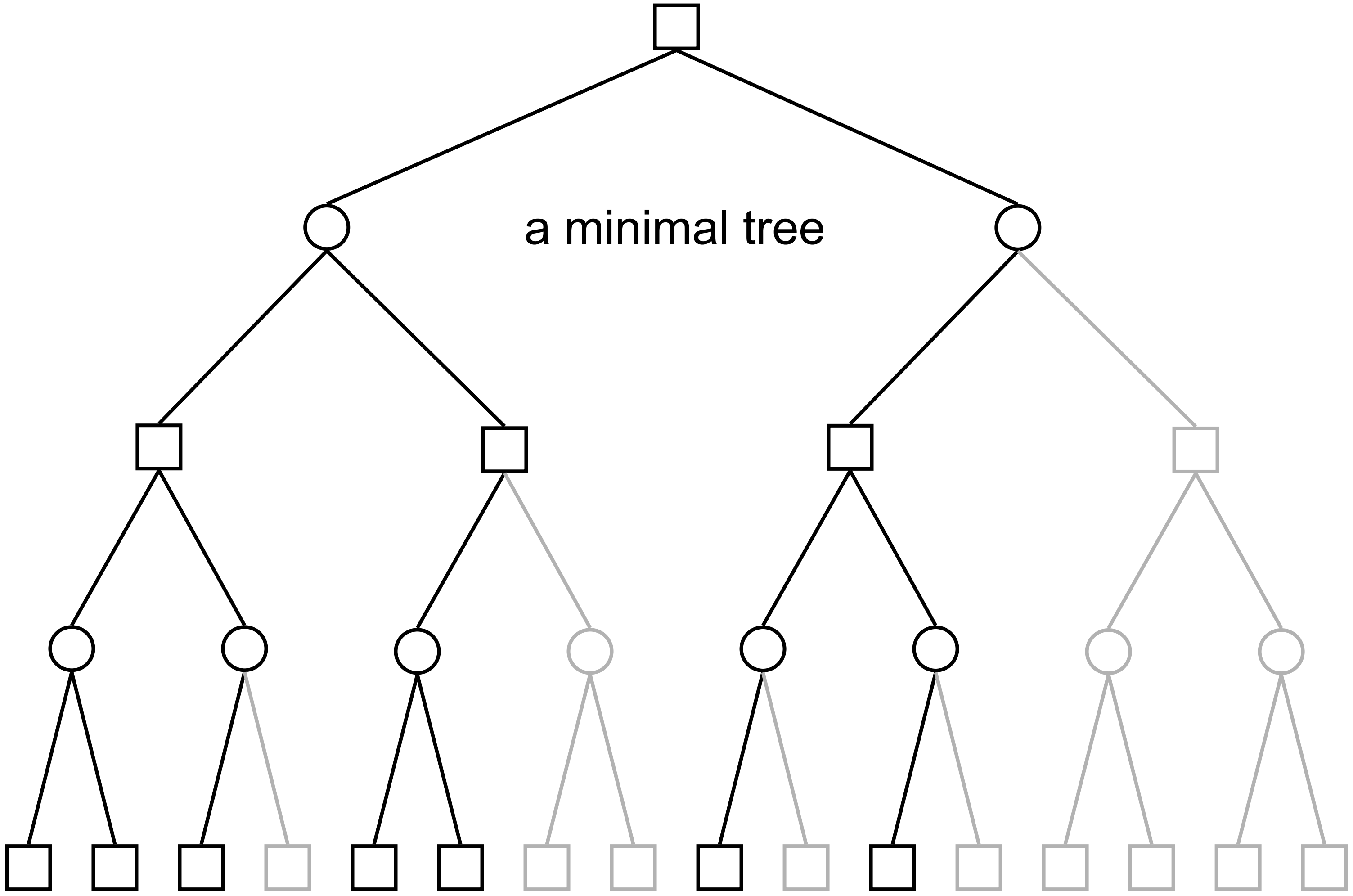
Summary?

There is no summary!

But many open questions.

Q1: Is the “Minimal Tree” minimal?

No! Need to deal with graphs (move transpositions!)



Q2: Does look-ahead improve the decision quality?

Computing a function of estimates is one of the deadly sins of statistics!

So, why should look-ahead improve the decision?

- **Improved visibility?**
  - Quality evaluation must increase by >50% each ply.
- **Filtering?**
  - Dependency among neighboring nodes only dampens the error slightly.
- **Avoiding traps?**
  - A small amount (~5%) of true leaves helps.

Q3: Why using Minimax?

Minimax assumes that the opponent acts according to the same rules and has the same information. Does this hold true?

Alternatives

- Product rule
- B* [Berliner 1979]
- B* with value ranges [Palay 1983]
- Conspiracy numbers [McAllester 1990?]

Q4: How to parallelize tree search algorithms?

- How to map the **process tree** onto the **processor topology**?
- How to exploit parallel FPGA hardware? (e.g. world champion `99, `03 Shredder)
- How to balance **search overhead** with **communication overhead**.
  - Chance of super linear speedup
- Early approaches
  - Tree-Split (Finkel/Fishburn, 1982): top-down strategy
  - PV-Split: bottom up strategy
  - Helpful Workers (Feldman et al.)
  - ... many more ...

Research consists of

three quarters of search!