```
function MINIMAX-DECISION(game) returns an operator

  for each op in OPERATORS[game] do
     VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
  end
  return the op with the highest VALUE[op]
```

```
function MINIMAX-VALUE(state, game) returns a utility value

  if TERMINAL-TEST[game](state) then
     return UTILITY[game](state)
  else if  MAX is to move in state then
     return the highest MINIMAX-VALUE of SUCCESSORS(state)
  else
     return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Algoritmi 10: Minimax-algoritmi [RN95, kuva 5.3].

Minimax-periaate on muotoiltu luvun 3.1.3 kaltaisena syvyyssuuntaisena etsintäalgoritmina 10, joka arvottaa etsintäpuun solmut lähtien liikkeelle nyt juuresta lehtien sijaan.

Algoritmi olettaa, että kaikki pelit päättyvät: esimerkiksi toistuvaan pelitilanteeseen vievä siirto

Shakissa merkitsee pelin päättymistä tasapeliin.

GO-pelissä on kielletty.

Molemmissa tapauksissa on etsinnän kuluessa tutkittava pelin historiaa eli pelipuun polkua juuresta nykyiseen solmuun samoin kuin luvun 3.1.7 tavassa 1.

Algoritmi 10 tutkii koko pelipuun, vaikka juuri todettiin sen olevan käytännössä mahdotonta. Muutetaan sitä siis seuraavasti:

- Puuta kasvatetaan vain siihen saakka kunnnes valittu *katkaisuehto (cutoff-test)* laukeaa, ei enää lopputilanteisiin asti.

  Yksinkertaisia katkaisuehtoja saadaan esimerkiksi

- valitsemalla etukäteen vakiosyvyys d jota pidemmälle ei edetä (kuten luvussa 3.1.4) tai
- syventämällä etsintää iteratiivisesti (kuten luvussa 3.1.5) kunnes sille varattu aika täyttyy.

Nämä ehdot kärsivät kuitenkin *tapahtumahorisonttiongelmasta (horizon problem)*, jossa etsintä päättyy liian aikaisin havaitakseen pelitilanteen todellisen laidan:

Kuvassa 47 musta pääsee shakkaamaan viidesti peräkkäin, mutta lopulta valkea saa kuningattaren ja todennäköisesti voittaa koko pelin.

Tätä voi yrittää ehkäistä pysäyttämällä etsintä vasta *hiljaisissa (quiescent)* pelitilanteissa, joissa näyttää siltä, ettei pelitilanteessa tapahdu enää dramaattisia yllätyksiä.

Toisaalta kuvassa 47 pelitilanne näyttää pitkään perin yllätyksettömältä. . .

• Katkaisukohdassa arvioidaan solmun todellista arvoa valitulla *evaluointifunktiolla (evaluation function)*.

Esimerkiksi shakin alkeiskirjoissa annetaan yksinkertaistettuja pistelaskusääntöjä materiaaliedun arviointiin:

Lasketaan yhteen sotilas 1, ratsu/lähetti 3, torni 5, kuningatar 9, "hyvä" sotilas- tai kuningasasema $\frac{1}{2}$,. . . (Vrt. kuva 48.)

Tämä on esimerkki yleisestä *lineaarisesta* evaluointifunktiosta. Vastaavia nyrkkisääntöjä korttipelille Bridge esiteltiin luvussa 3.2.3 etsintäheuristiikkoja keksittäessä.

• Evaluointifunktion laatu on keskeinen koko peliohjelman laadulle:

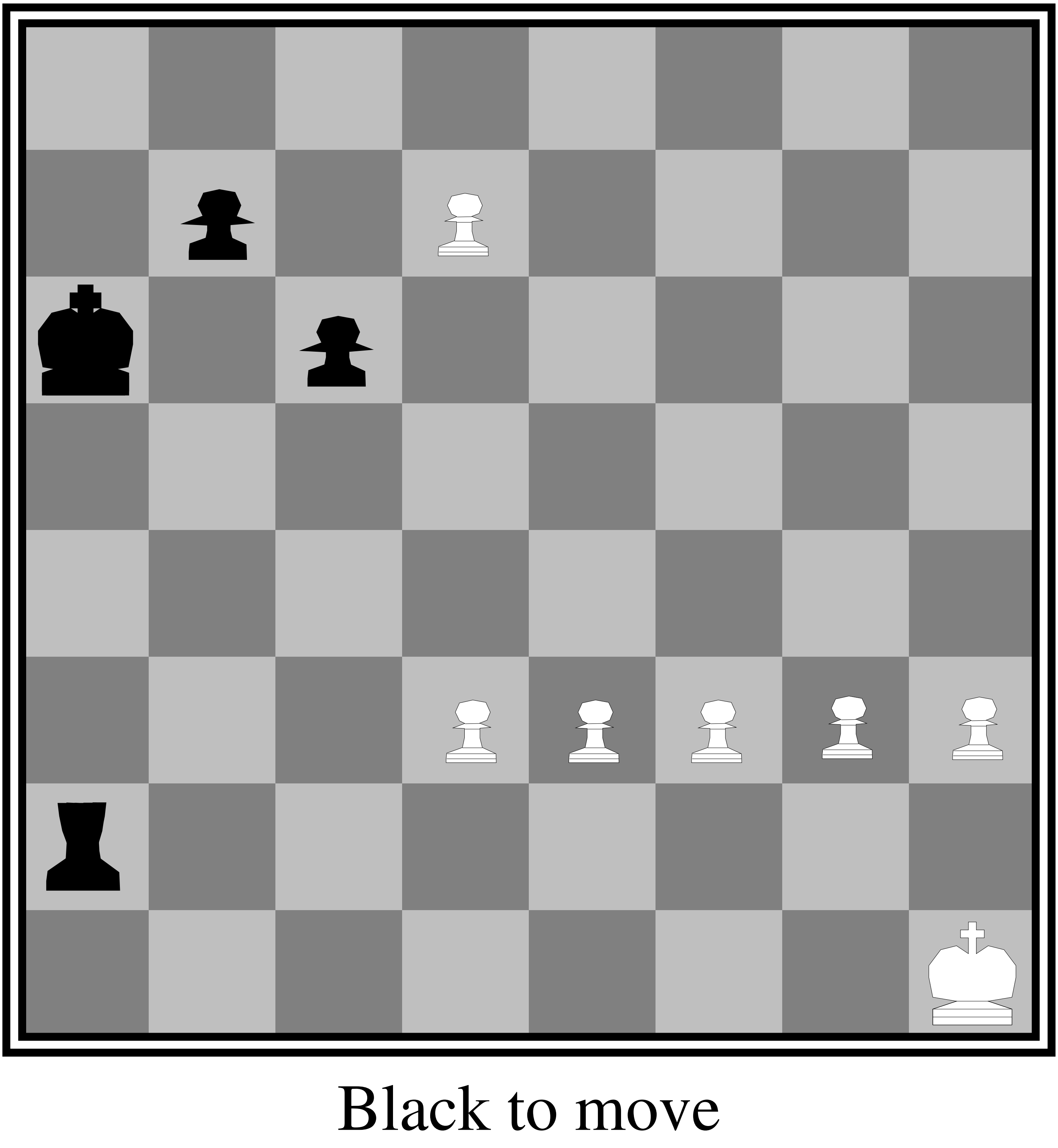
Solmulle H annetun arvon $\textsc{Eval}(H)$ tulisi heijastaa mahdollisimman luotettavasti sitä, kuinka pelitilanteesta H jatkuvassa pelissä *odotusarvoisesti* tulisi käymään.

Arvo $\textsc{Eval}(H)$ edustaa näet yksinään kokonaista solmusta H alkavaa pelipuun alipuuta, jonka lehtien L arvot $U(L)$ saattavat vaihdella suurestikin.

Jos $\textsc{Eval}(H)$ mallintaa suureen $U(L)$ odotusarvoa kun L valitaan umpimähkään, niin se mallintaa samaa pelistrategiaa kuin katkaistu etsintäkin:

Edetään harkitusti tilanteeseen H saakka, sen jälkeen umpimähkään.

Tällöin korkea $\textsc{Eval}(H)$ vihjaa, että tilanteesta H alkavasta alipuusta T löytynee myös korkeaan arvoon $U(L)$ vievä strategia.

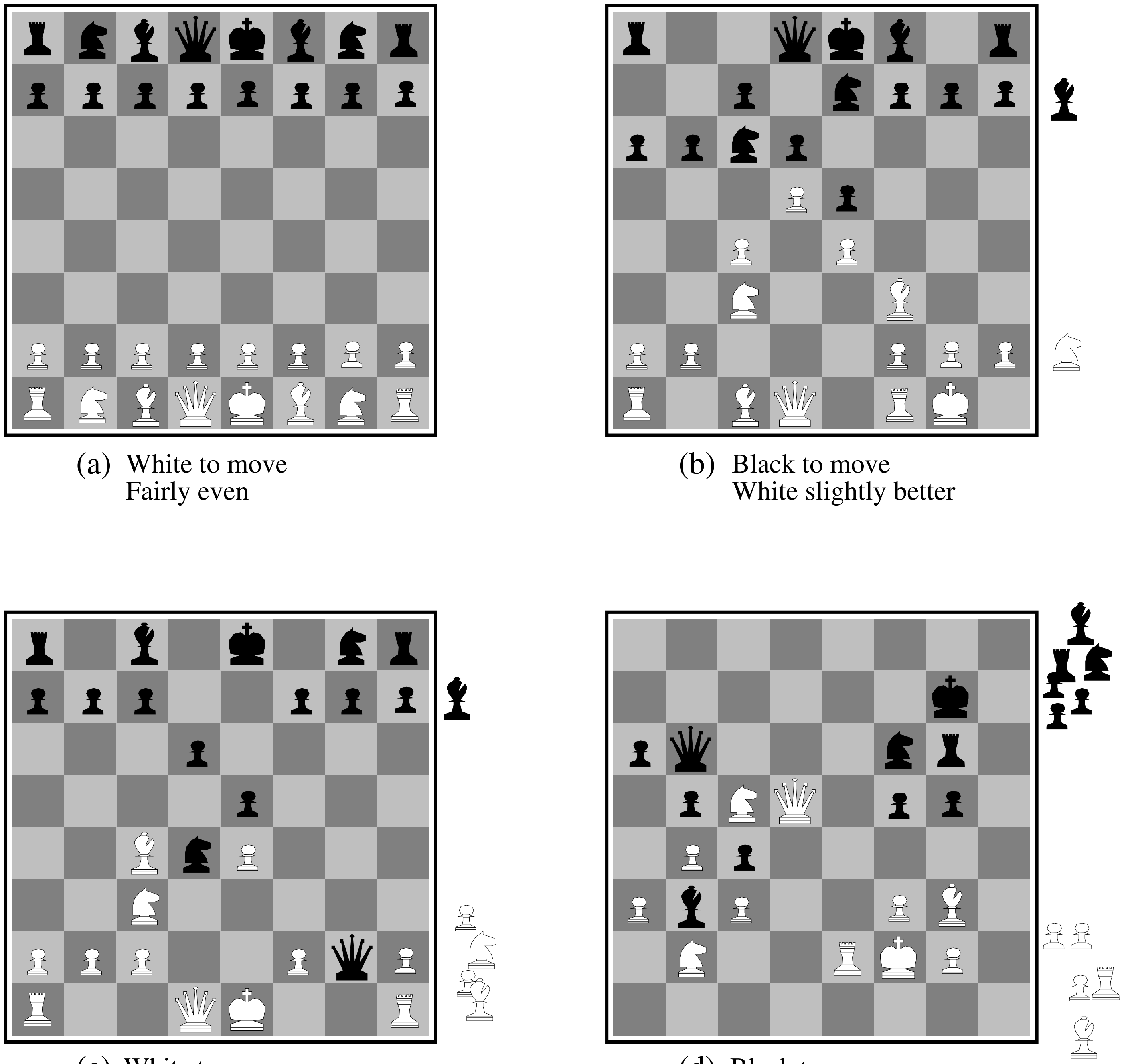


Kuva 47: Musta shakkaa, valkea voittaa [RN95, kuva 5.5].

Onneksi funktion $\textsc{Eval}$ ei tarvitse saada eri solmujen H_1 ja H_2 numero*arvoja* oikein, kunhan niiden *järjestys* vain säilyy, eli

$$\textsc{Eval}(H_1) \leq \textsc{Eval}(H_2) \tag{12}$$

vain jos H_2 on "ainakin yhtä lupaava pelitilanne" kuin H_1.



(a) White to move
Fairly even

(b) Black to move
White slightly better

(c) White to move
Black winning

(d) Black to move
White about to lose

Kuva 48: Shakkitilanteita arvioineen [RN95, kuva 5.4].

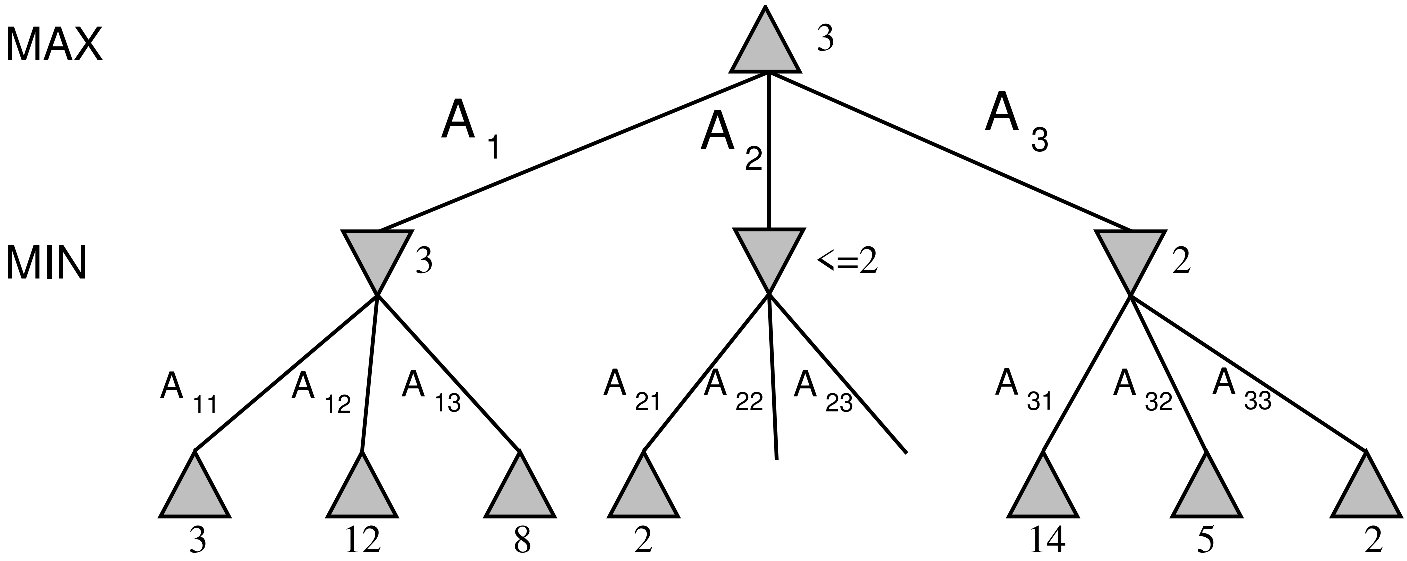
4.3.2. Alfa-beta -karsinta

Minimax-algoritmi 10 on käytännössä yhä liian hidas vaikkei se enää kävisikään läpi koko pelipuuta.

Seuraavaksi esitetään eräs tapa *karsia (prune)* pelipuuta, eli jättää joitakin sen alipuita tutkimatta ilman muutosta kokonaistulokseen.

Karsintaperiaatteemme on *alfa-beta* (α-β), joka on saanut nimensä karsintaa ohjaavista luvuistaan

$$\begin{matrix}\alpha\\ \beta\end{matrix} = \begin{matrix}\text{ala}\\ \text{ylä}\end{matrix}\text{raja pelaajan } \begin{matrix}\max\\ \min\end{matrix} \text{ saamalle arvolle.}$$



Kuva 49: Kuvan 46 alfa-beta -karsinta [RN95, kuva 5.6].

Esimerkiksi kuvan 46 pelipuusta voidaan jättää tutkimatta kuvan 49 osoittamat solmut:

Syvyyssuuntainen minimax-etsintä (algoritmi 10) vasemmalta oikealle antaa

1. ensin vaihtoehdolle A_1 minimax-arvon 3,

2. sitten vaihtoehdolle A_{21} minimax-arvon 2
3. jonka nojalla vaihtoehdon A_2 minimax-arvo on *enintään* tuo 2
4. eli pelaaja $\max$ ei koskaan pelaisi vaihtoehtoa A_2 vaan vaihtoehdon A_1 (tai vielä sitäkin paremmin).

Yleinen karsintaperiaate näkyy kuvasta 50:

Pelaaja (player) $\max$ ($\min$) ei koskaan valitse solmua n jos hänellä on jokin parempi vaihtoehto $m > n$ ($m < n$) pelipuun juuresta solmuun n vievällä polulla.

Koska minimax-algoritmi 10 eteni syvyyssuuntaisesti, se tarkastelee aina yhtä polkua kerrallaan.

Pidetään siis yllä vain tähän polkuun liittyviä lukuja α ja β etsinnän edetessä. Saadaan algoritmi 11.

- Algoritmin 11 karsintatehokkuus riippuu siitä järjestyksestä, missä solmun lapset käydään läpi:

  jos kuvan 49 alipuu A_3 olisi aloitettu lapsesta A_{33}, niin

  3. $\textsc{Min-Value}(A_3, 3, +\infty) =$
     (a) $\textsc{Max-Value}(A_{33}, 3, +\infty) = 2$

     $= 3.$

  Algoritmi 11 karsii sitä paremmin, mitä aikaisemmin kunkin solmun oikea minimax-arvo tavataan sen lapsista.

  Jos voisimme järjestää lapset aina etukäteen oikein ilman niiden läpikäyntiä, niin voisimme pelata peliä täydellisesti ilman etsintää. . .

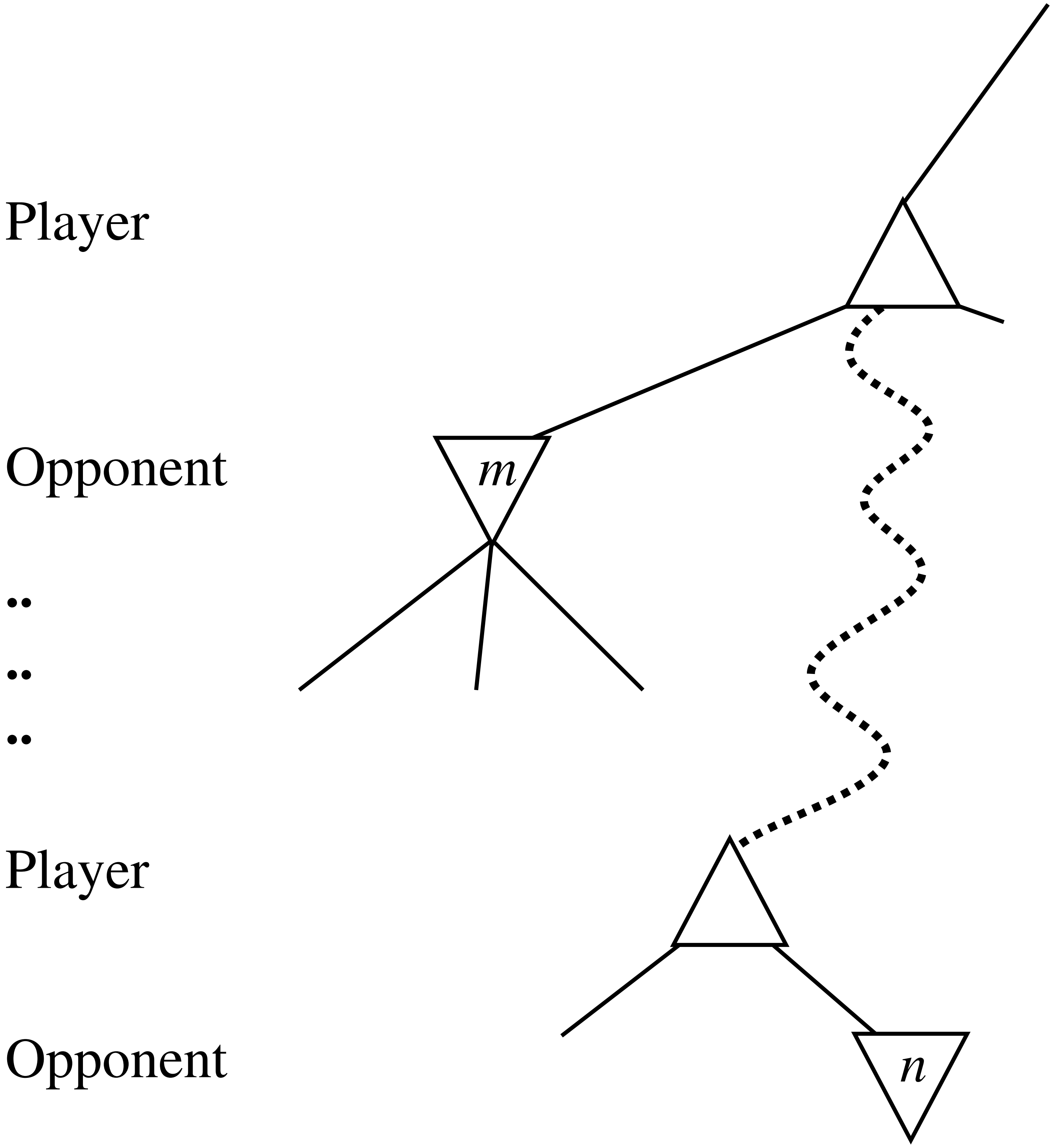
  Tai saavuttaa algoritmin 11 optimitapaus: tutkia vain $O(b^{d/2})$ solmua, missä

  – b on pelipuun haarautumisaste eli mahdollisten siirtojen määrä kussakin pelitilanteessa

  – d puun syvyys eli siirtojen kokonaismäärä kussakin pelissä.

  (Algoritmi 10 tutkii puolestaan kaikki pelipuun $O(b^d)$ solmua.)
- Silloin luvun 3.2.3 todellinen haarautumisaste olisikin enää $\sqrt{b}$ eikä b.

- Toisin sanoen samalla etsintätyöllä päästäisiin kaksi kertaa niin syvälle. Ja etsintäsyvyys eli "siirtoparien ajattelu eteenpäin" on keskeistä hyvässä strategiapelissä.
- Jos lapset ovat aivan satunnaisessa järjestyksessä, niin algoritmi 11 tutkii $O((b/\log b)^d)$ solmua.
  Silloin todellinen haarautumisaste $b/\log b$ on lähes b.



Kuva 50: Alfa-beta-karsinnan periaate [RN95, kuva 5.7].

```
function MAX-VALUE(state, game, α, β) returns the minimax value of state
   inputs: state, current state in game
           game, game description
           α, the best score for MAX along the path to state
           β, the best score for MIN along the path to state

   if CUTOFF-TEST(state) then return EVAL(state)
   for each s in SUCCESSORS(state) do
       α ← MAX(α, MIN-VALUE(s, game, α, β))
       if α ≥ β then return β
   end
   return α
```

```
function MIN-VALUE(state, game, α, β) returns the minimax value of state

   if CUTOFF-TEST(state) then return EVAL(state)
   for each s in SUCCESSORS(state) do
       β ← MIN(β, MAX-VALUE(s, game, α, β))
       if β ≤ α then return α
   end
   return β
```

Algoritmi 11: Alfa-beta-algoritmi [RN95, kuva 5.3].

Alkukutsu $\textsc{Max-Value}($kuva 49$, -\infty, +\infty) =$

1. $\textsc{Min-Value}(A_1, -\infty, +\infty) =$
   (a) $\textsc{Max-Value}(A_{11}, -\infty, +\infty) = 3$
   (b) $\textsc{Max-Value}(A_{12}, -\infty, 3) = 12$
   (c) $\textsc{Max-Value}(A_{13}, -\infty, 3) = 8$
   $= 3$
2. $\textsc{Min-Value}(A_2, 3, +\infty) =$
   (a) $\textsc{Max-Value}(A_{21}, 3, +\infty) = 2$
   $= 3$
3. $\textsc{Min-Value}(A_3, 3, +\infty) =$
   (a) $\textsc{Max-Value}(A_{31}, 3, +\infty) = 14$
   (b) $\textsc{Max-Value}(A_{32}, 3, 14) = 5$
   (c) $\textsc{Max-Value}(A_{33}, 3, 5) = 2$
   $= 2$

$= 3.$

(Vakioparametri $game$ jätettiin selvyyden vuoksi pois.)

- Onneksi tämä asymptoottinen kaava pätee vasta kun $b > 1\ 000$, jolloin etsintäpuumenetelmät olisivat muutenkin käytännössä mahdottomia.

  "Luonnollisilla" haarautumisasteen b arvoilla algoritmi 11 tutkii $O(b^{3d/4})$ solmua.

  Jo melko yksinkertaiset lapsisolmujen käsittelyjärjestykset pääsevät usein melko lähelle tätä arvoa.

Edellä kuitenkin oletettiin, että funktion $\textsc{Eval}$ (tai U) arvot ovat jakautuneet satunnaisesti etsintäpuun lehdille.

Tämä tuskin pätee tavallisissa peleissä:

Jokin aikaisempi hyvä/huono siirto (eli sisäsolmu) korottaa/vähentää suurinta osaa lehtien arvoista kyseisessä alipuussa.

Näin saman alipuun eri solmujen arvot tapaavat riippua vahvasti toisistaan kullekin peli(tilantee)lle ominaisella tavalla.

Näiden piiloriippuvuuksien vuoksi hyvän strategiapelialgoritmin kehittämisessä

- kokeilulla
- koneoppimisella

on tärkeä rooli näitä yleisiä algoritmeja räätälöitäessä.

Alfa-beta-karsinta on yleisen *haaraudu ja rajoita (branch and bound)* -etsintäperiaatteen sovellus pelipuihin:

Kun etsintä haarautuu, yritä laskea rajoja sille, kuinka hyviä tuloksia voi vielä ilmaantua, ja karsi etsintähaaroja näiden rajojen salliessa.