Hraní her

Hry jako problém vyhledávání

Hry byly vždy zajímavou oblastí i z toho hlediska, že představují snadno representovatelný problém pro jednotlivé stavy a agenti jsou obvykle omezeni na malý počet dobře definovaných akcí. Z hlediska umělé inteligence hry představují idealisaci světů, v nichž nepřátelští agenti se snaží působit tak, aby snížili oponentovu prosperitu. Příkladem je hra v šachy (v umělé inteligenci, vzniklé brzy po vzniku programovatelných počítačů, se touto hrou zabýval již v roce 1950 Claude Shannon—objevitel informační teorie—a Alan Turing). Šachy mají jednoduchá pravidla, stav světa je zcela přístupný (veškerá informace popisující daný stav je k dispozici), representace je jednoduchá z hlediska prohledávání prostoru možných šachových pozic, popis stavů je korektní do posledního detailu (na rozdíl od např. řešení problémů skutečné války). Navíc se jedná o problém vyžadující k řešení inteligenci.

Přítomnost oponenta (protivníka) komplikuje rozhodovací proces oproti problému s vyhledáváním, protože oponent vnáší prvek *nejistoty* do situace. Všechny programy her musí uvažovat **problém nahodilosti** (zmíněný již dříve). Existuje určitý rozdíl: oponent představuje jiný typ nahodilosti než např. hod kostkou nebo vliv počasí, protože se co nejvíce snaží učinit co nejméně příznivý tah z hlediska druhého hráče—kostka a počasí jsou normálně považovány (možná chybně) za neutrální k cílům agenta z hlediska *úmyslného* bránění dosažení cíle.

Hlavním problémem při řešení her je však to, že jsou obvykle příliš obtížné k vyřešení. Např. šachy mají průměrný faktor větvení 35 a šachová partie často zabere 50 tahů pro každého z obou protivníků, takže prohledávací strom má cca 35^{100} uzlů—existuje však samozřejmě "pouze" cca 10^{40} *různých* korektních pozic. Složitost her tedy vnáší do řešení nový prvek nejistoty—nikoliv z důvodu nedostatku informace, ale kvůli tomu, že hráč nemá obecně dostatek času propočítat přesné důsledky libovolného tahu. Místo toho hráč musí důsledky odhadnout, např. na základě minulé zkušenosti, a provést akci před tím, než s jistotou zjistí skutečné důsledky. Z tohoto hlediska jsou to právě *hry*, které *jsou daleko více podobné reálnému světu než standardní vyhledávací problémy* diskutované doposud.

Jiným problémem u her je časové omezení—penalisace neefektivnosti bývá velmi vysoká (v šachu nedodržení přiděleného času vede k prohře partie, přičemž hledaným cílem je výhra). Zatímco implementace hledání typu A^*, která je o 10% méně efektivní, vede pouze k o něco delší době výpočtu pro nalezení cíle, šachový program s efektivností ve využití času sníženou o 10% bude prostě prohrávat, přestože ostatní parametry bude mít stejné jako časově efektivnější program-protivník.

Řešení problémů spojených s hraním her obyčejně vyžaduje analýzu nalezení teoreticky nejlepšího tahu, dále výběr techniky pro určení dobrého tahu vzhledem k časovému omezení, **prořezání** (stromu) kvůli odstranění částí, které neovlivňují finální výběr, a heuristickou **vyhodnocovací funkci** , která umožní přibližné posouzení užitečnosti stavu bez nutnosti kompletního prohledávání.

Dokonalé rozhodování ve hrách pro dvě osoby

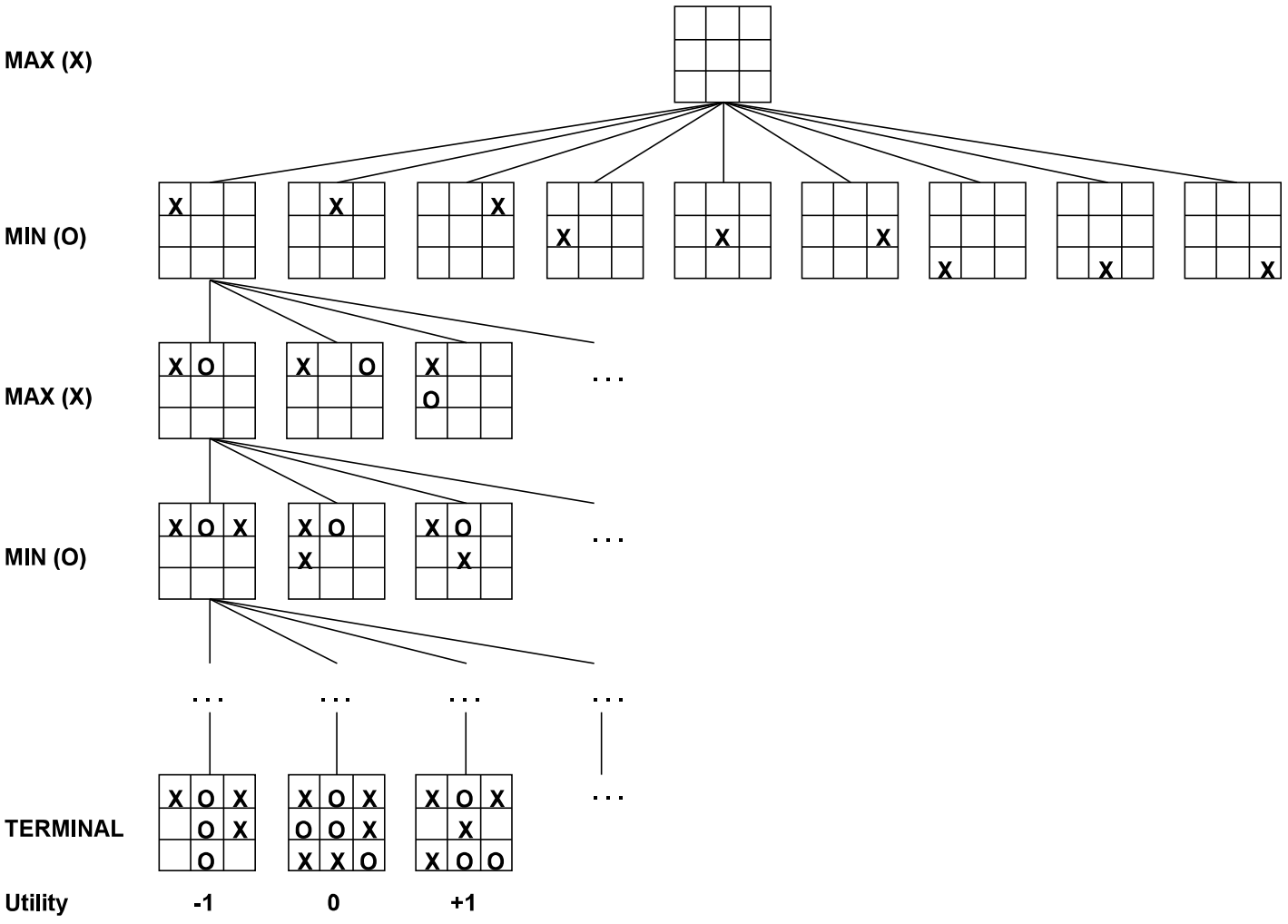
Zde se uvažuje obecný případ hry, kde jsou dva hráči-protivníci, nazývaní MAX a MIN. Hráč MAX hru začíná a pak se oba střídají v tazích až do konce hry. Vítěz na konci dostane odměnu v bodech (poražený někdy dostane pokutu v bodech). Hru lze formálně definovat jako druh hledacího problému s následujícími složkami:

- **Počáteční stav**, který zahrnuje pozici na hrací desce (např. šachovnici apod.) a údaj o tom, kdo je na tahu.
- Soubor **operátorů**, definujících korektní tahy, které hráč může učinit.
- **Cílový test**, který určí, zda hra je či není skončena (stavy, kde hra je ukončena, jsou tzv. **konečné stavy**).
- **Funkce přínosnosti** (utility function), někdy nazývaná jako **funkce zisku**, poskytující v numerické formě výsledek hry—v šachu je to vítězství, prohra nebo nerozhodný výsledek (remíza), což je representováno pomocí hodnot +1, −1, nebo 0. Některé hry mají širší interval, např. backgammon od +192 do −192.

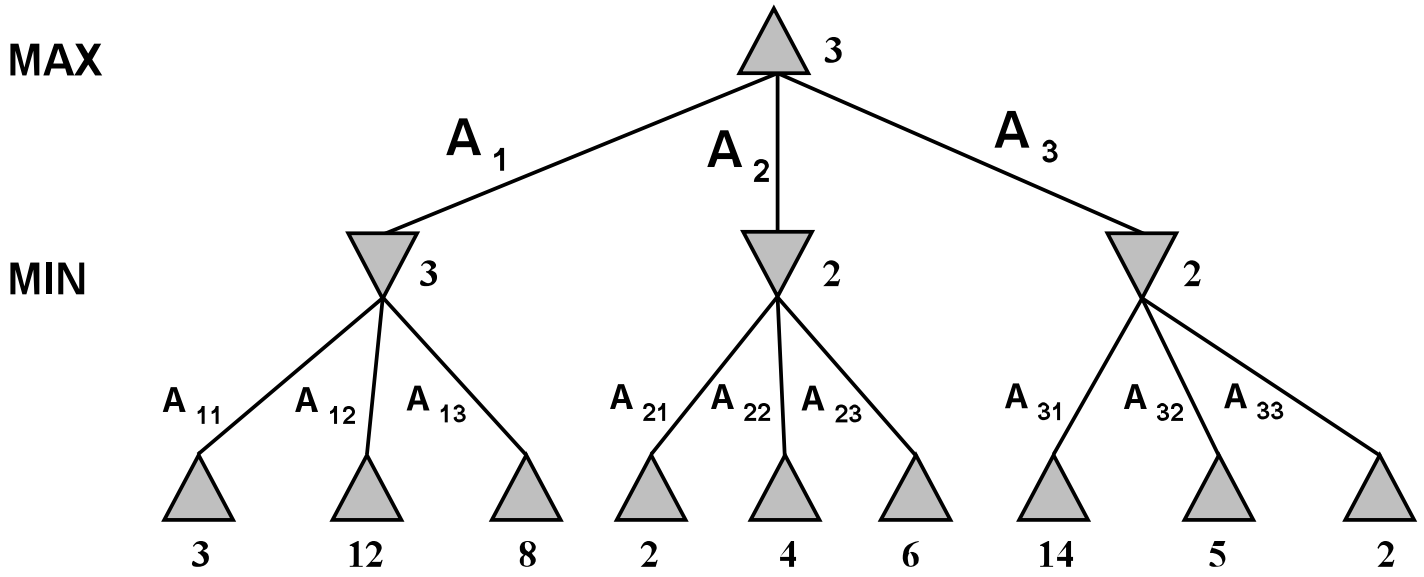
U normálního vyhledávacího problému by MAX prostě hledal sekvenci tahů vedoucích do vítězného konečného stavu (s použitím funkce přínosnosti), a po nalezení cesty by vybral první z tahů takové sekvence. Protihráč MIN ale hru ovlivňuje, proto MAX musí najít **strategii** vedoucí ke konečnému stavu bez ohledu na to, co MIN dělá.

Strategie obsahuje korektní tah (tah podle pravidel) pro MAXe jako odpověď na každý možný tah MINa. Existují cesty k nalezení strategie, i když omezením pro výpočet může být (a bývá) časový limit.

Následující obrázek ukazuje část vyhledávacího stromu pro hru Tic-Tac-Toe (piškvorky). Z počátečního stavu má MAX na výběr 9 možností. Hra zahrnuje střídání MAXe vkládajícího ✘ a MINa vkládajícího ○, dokud není dosaženo listů (odpovídajících konečným stavům)—stavy, kdy jeden hráč má 3 v řadě nebo všechna políčka jsou obsazena. Číslo u každého listu znázorňuje výsledek z hlediska MAXe (vyšší hodnota je výhodnější pro MAXe a horší pro MINa). Zde je na MAXovi, aby použil vyhledávací strom (začíná hru):



Avšak i tak jednoduchá hra jako piškvorky je příliš složitá z hlediska presentace celého vyhledávacího stromu, proto je vhodné se podívat na zcela triviální hru znázorněnou na dalším obrázku:



Možné tahy pro MAXe jsou označeny A_1, A_2, a A_3; možné odpovědi MINa na A_1 jako A_{11}, A_{12}, A_{13}, atd. Hra končí poté, co MAX i MIN učiní jeden tah (jednotahová hra, skládající se ze dvou půltahů). Výnosy v koncových stavech této hry jsou v rozmezí 2–14.

Pro nalezení optimální strategie pro MAXe existuje algoritmus zvaný **minimax**, tj. hledání, který první tah je nejlepší. Minimax se skládá z pěti kroků:

- Generuj strom celé hry, směrem dolů ke konečným stavům.
- Aplikuj funkci zisku na každý terminální stav, aby byla určena jeho hodnota.
- Použij výnos koncových stavů k určení výnosnosti uzlů o jednu úroveň vyhledávacího stromu výše. Uvažme nejlevější tři listy na obrázku—v uzlu ▼ nad těmito listy má MIN možnost táhnout a jeho nejlepší možností je hrana A_{11} vedoucí k minimální hodnotě, 3. Proto lze tomu uzlu ▼ přiřadit hodnotu 3, i když na něj nelze přímo aplikovat funkci zisku. Hodnota 3 je za předpokladu, že MIN udělá správný tah. Obdobně jsou ostatní dva uzly ▼ oceněny hodnotou 2.
- Pokračuj ve zpětném přidělování hodnot z listů uzlům směrem ke kořeni stromu, v každém kroku v rámci jedné úrovně uzlů.
- Nakonec, po dosažení kořene, MAX vybírá tah, který vede k nejvyšší hodnotě, tedy v nejhornějším uzlu ▲ na obrázku má MAX výběr ze tří tahů vedoucích k ziskům 3, 2, a 2 (v tomto pořadí). Proto je MAXův nejlepší zahajovací tah A_1. Takový postup se nazývá **rozhodnutí minimax**, protože maximalizuje zisk za předpokladu dokonalé hry protivníka, který se snaží MAXův zisk minimalizovat tahem A_{11}.

Minimax algoritmus lze formálněji opět zapsat pomocí funkcí: `Minimax-Decision` (výběr z tahů, které jsou k dispozici) a `Minimax-Value` (ohodnocení vybíraných tahů). Algoritmus vrací operátor odpovídající nejlepšímu možnému tahu (tj. tahu s nejvyšším číselným ziskem) za předpokladu, že protivník hraje tak, aby zisk minimalizoval. Funkce `Minimax-Value` prochází celý strom hry až k listům, aby určila návratovou hodnotu stavu.

```
function Minimax-Decision(game) returns operátor

     for each op in Operators[game] do
          Value[op] ← Minimax-Value(Apply(op,game),game)
     end

     return op // s nejvyšší hodnotou Value[op]
```

```
function Minimax-Value(state,game) returns hodnotu zisku

     if Terminal-Test[game](state) then
          return Utility[game](state)
     else if MAX je na tahu ve stavu state then
          return nejvyšší Minimax-Value z Successors(state)
     else
          return nejnižší Minimax-Value z Successors(state)
```

Je-li maximální hloubka stromu *m* a existuje *b* korektních tahů v každé pozici, pak je časová složitost algoritmu minimax $O(b^m)$. Algoritmus v podstatě je hledání prvně do hloubky, takže paměťové požadavky jsou zde pouze lineární vzhledem k *m* a *b*.

Pozn.: Pro praktickou hru jsou časové požadavky zcela nepraktické, ale algoritmus se používá jako základ realističtějších metod a pro matematickou analýzu hry.

Nedokonalé rozhodování

Algoritmus minimax předpokládá, že program má dost času k prohledání všech cest k terminálním stavům (ukončení hry vzhledem k jejím pravidlům, např. výhra/prohra/remíza v šachu)—to však zcela obvykle je nepoužitelné.

Claude Shannon ve svém článku r. 1950 *(Programming a computer for playing chess. Philosophical Magazine, 41(4): 256-275)* navrhl, aby místo procházení všech cest do terminálních stavů a použití vyhodnocovací funkce program ukončil hledání dříve a aplikoval heuristickou **evaluační funkci** na listy zmenšeného stromu. Znamená to změnit minimax dvěma způsoby: funkce výpočtu zisku je nahrazena evaluační funkcí `Eval` a terminální test je nahrazen testem na přerušení vyhledávání ve stromu `Cutoff-Test`.

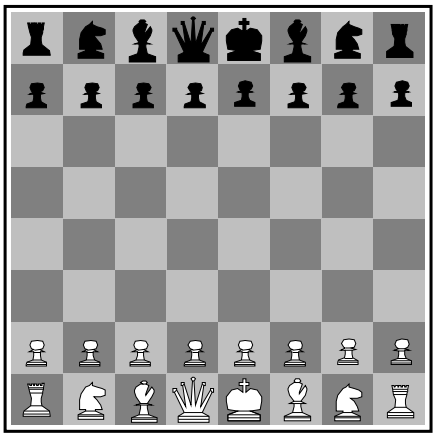
Evaluační funkce

Evaluační funkce vrací *odhad* očekávaného výnosu hry pro danou pozici. Tato myšlenka nebyla a není nic nového, odjakživa tak šachisté (a hráči jiných her) hráli, tj. vypracovali si způsoby odhadu vítězných šancí pro každou stranu na základě snadno spočítatelných charakteristik pozice.

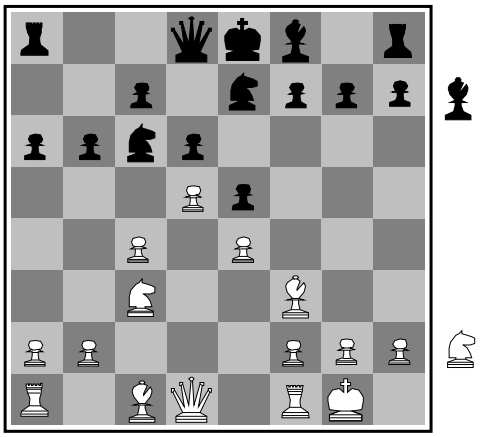
Např. učebnice pro začátečníky poskytuje v úvodu **přibližné materiální hodnoty** pro každou figuru (p=1, J=S=3, V=5, D=9), další záležitosti, jako “dobrá pěšcová struktura” a “bezpečnost krále” mohou být hodnoceny jako půl pěšce, apod. Z toho pak vyplývá, že např. výhoda 3 bodů navíc postačuje k vítězství, i když samozřejmě roli hraje mnoho dalších věcí, nikoliv jen materiální hodnota pozice (oběti figur, aj.).

Z toho všeho tedy plyne, že výkonnost programu hrajícího šachy je extrémně závislá na kvalitě vyhodnocovací (evaluační) funkce.

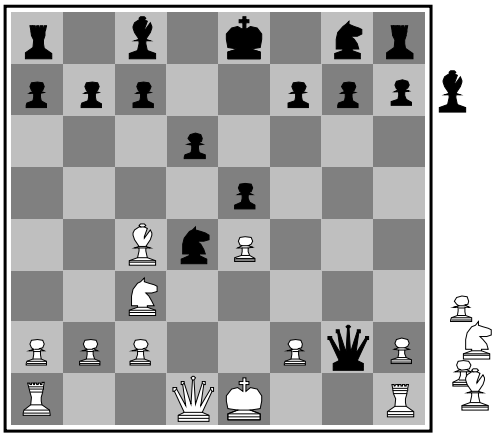
Obrázek ukazuje čtyři různé pozice s jejich ohodnocením (white/black to move = bílý/černý na tahu, fairly even=vyrovnaná pozice, white slightly better=bílý stojí o něco lépe, black winning=černý stojí na výhru, white about to lose=bílý stojí na prohru).



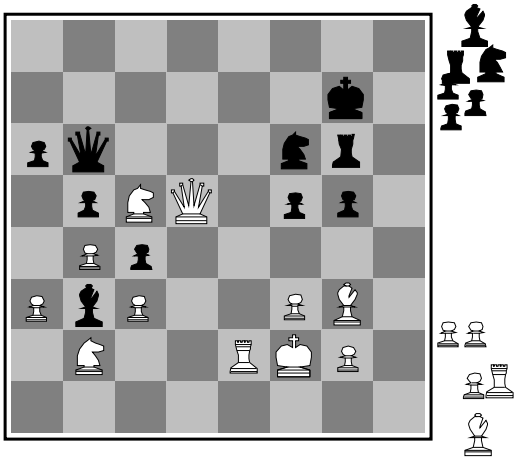
(a) White to move
Fairly even



(b) Black to move
White slightly better



(c) White to move
Black winning



(d) Black to move
White about to lose

Otázkou je, jak přesně měřit kvalitu:

- Evaluační funkce se musí shodovat s funkcí zisku v koncových stavech.
- Měření nesmí trvat dlouho (viz časová složitost); jediným řešením je kompromis mezi přesností evaluační funkce a časovými nároky.
- Evaluční funkce musí přesně zobrazovat skutečné šance na výhru.

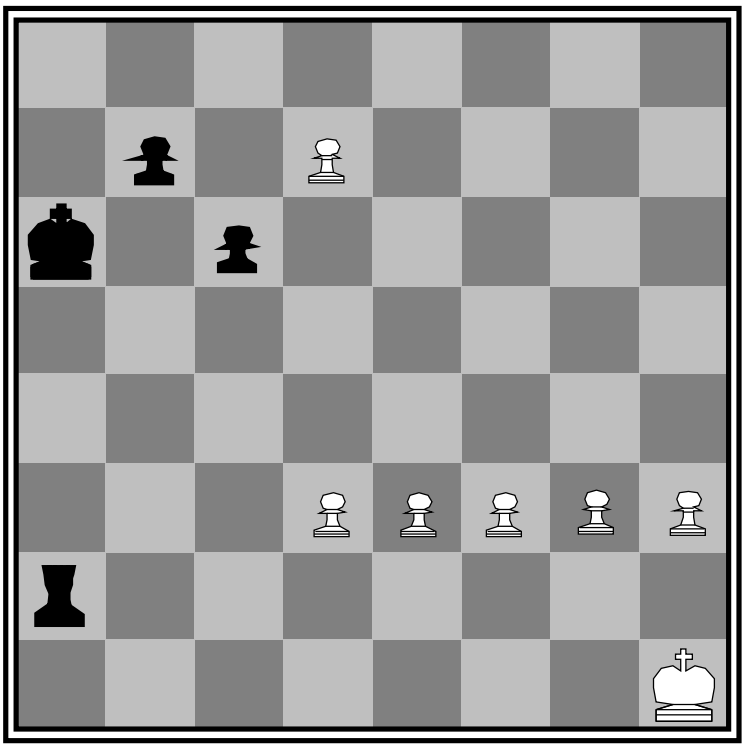
Termín “šance na výhru” souvisí s tím, že hledání je přerušeno před dosažením terminálních stavů, takže se neví, co je dále, proto ta “pravděpodobnost”.

Přerušení vyhledávání

Nejjednodušší je stanovit pevnou mez *d* pro hloubku, v závislosti na časové spotřebě. Robustnější přístupy využívají iterativní hledání do hloubky (metoda zmíněná dříve). Jakmile je spotřebován čas přidělený na výpočty hodnot vznikajících pozic po různých pokračováních, skončí se a vybrán je doposud nejlepší nalezený tah. Zde může dojít k řadě problémů, např. efekt horizontu apod.

Důmyslnější hledání než materiální propočty do předem stanoveného limitu používají např. vyhodnocování pozic do dosažení tzv. *klidné* pozice (ukončena všechna braní, šachování, apod.), kdy je nepravděpodobné, že dojde k velkým změnám v blízké budoucnosti.

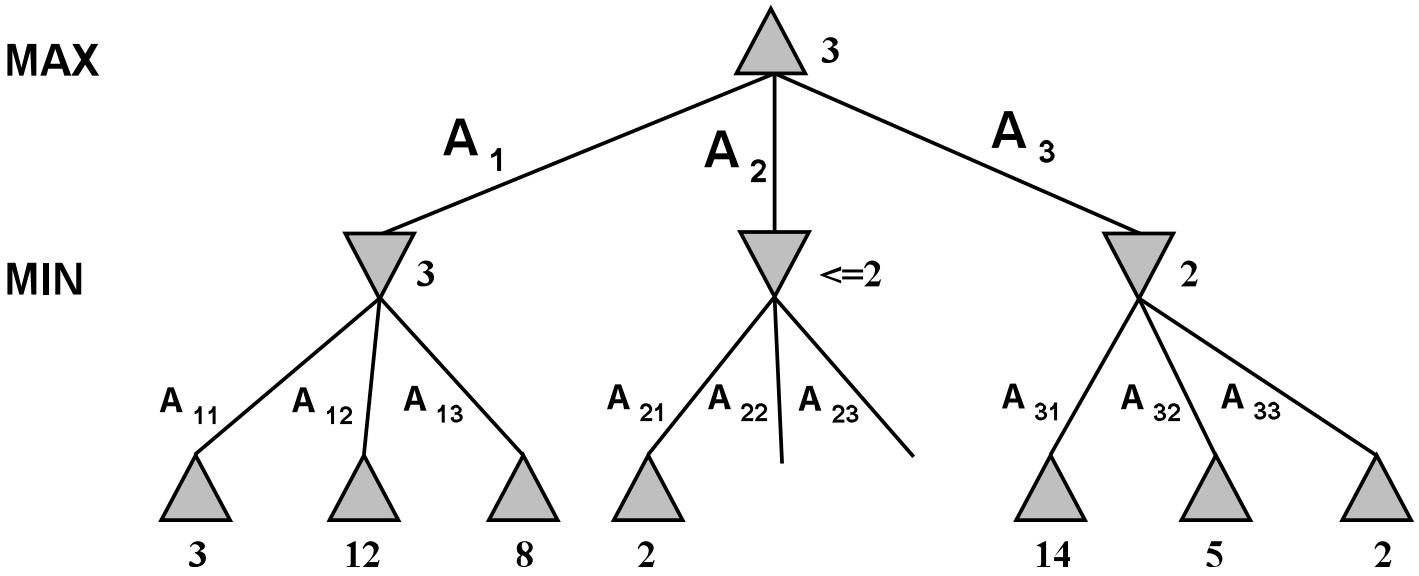
Problém horizontu je obtížnější: program se bude snažit šachováním věží odsunout přeměnu bílého pěšce na sedmé řadě za dámu až "za horizont", kam nevidí a tím mu zmizí z mysli prohra; viz obrázek (černý je na tahu). V současnosti stále neexistuje obecné řešení problému horizontu.



Black to move

Alfa-beta prořezávání

Aby se zmenšil vyhledávací strom, aniž by musel být příliš limitován malou hloubkou, je zapotřebí vyřadit některé jeho větve (co nejvíc větví). Tato metoda se nazývá **prořezávání** vyhledávacího stromu. Nejčastěji se využívá technika zvaná **alfa-beta prořezávání**, aplikovaná na minimax. Předpokládejme opět "dvoupůltahovou" hru na obrázku:

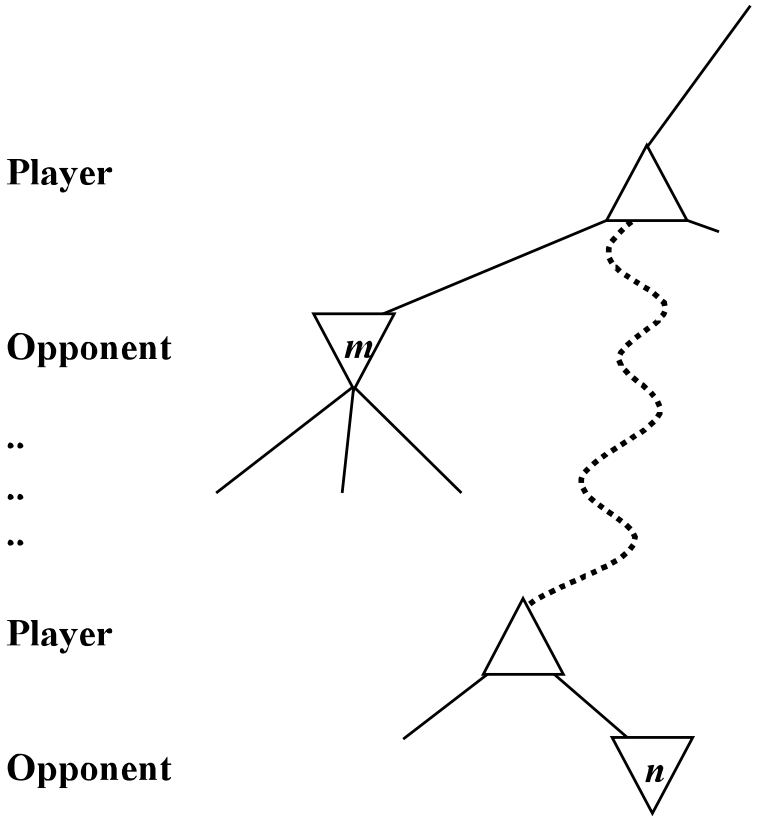


Hledání proběhne jako před tím, tj. A_1, A_{11}, A_{12}, A_{13} a uzel A_1 dostane minimaxovou hodnotu 3. Nyní následuje A_2 a A_{21}, což dá hodnotu 2: MAX hraje A_2 a MIN má možnost dosáhnout pozice s hodnotou 2.

Z toho v principu plyne, že tah A_2 má cenu *nanejvýš* 2 pro MAXe. Protože už je známo, že tah A_1 má cenu pro MAXe 3, není už tedy důvod hledat dále pod A_2. Zde tedy lze prořezat strom, protože další hledání v této větvi neovlivní výsledek.

V obecném principu lze uvažovat nějaký uzel *n* někde ve stromu (viz další obrázek). Pokud má hráč lepší možnost *m* buď v rodičovském nebo ještě vyšším uzlu, nebude *n* ***nikdy*** dosažen ve skutečné hře. K tomu závěru se dospěje prozkoumáním některých potomků uzlu *n* a větev není třeba dále uvažovat.

Minimax hledá prvně do hloubky, takže v každém okamžiku je nutno uvažovat uzly podél jediné větve.



Nechť α je doposud nejlepší nalezená hodnota v kterémkoliv bodě cesty pro MAXe a nechť β je doposud nejlepší (tj. nejnižší) nalezená hodnota pro MINa. Hledání alfa-beta počítá hodnoty α a β hlouběji podél cesty a odsekne podstrom, jakmile dosáhne hodnoty horší než současné α nebo β.

Algoritmus, popsaný formálněji, je rozdělen na funkce `Max-Value` a `Min-Value`. Funkce jsou aplikovány na MAXe a MINa, ale v principu dělají totéž: vracejí minimaxovou hodnotu uzlu kromě uzlů, které budou odříznuty. Alfa-beta vyhledávání je v podstatě kopie funkce `Max-Value` navíc s kódem, který uchovává a vrací nejlepší nalezený tah.

Efektivita alfa-beta algoritmu závisí na pořadí, v němž se vyhodnocují uzly. Nejlepší by bylo napřed vyhodnocovat *nejpravděpodobněji* nejlepší, ale to nelze udělat dokonale, jinak by funkce poskytovala dokonalou hru. Existuje několik analýz složitosti—pro náhodný výběr uzlů a $b > 1000$ je to $O((b/\log b)^d)$, což není o moc lepší než pouze pro b. Pro realističtější b bylo ukázáno, že lze počítat se složitostí přibližně $O(b^{3d/4})$ co do počtu uzlů. V praxi se používá napřed testování možných braní, pak hrozby, tahy vpřed, tahy ústupové apod., takže je možné se dost vzdálit náhodnému výběru uzlů a snížit náročnost. Uzly také nemají stejné b, což komplikuje matematický model využívající idealizovaného modelu stromu.

Alfa-beta vyhledávací + prořezávací algoritmus:

Funkce `Max-Value`:

```
function Max-Value(state,game,α,β) returns minimax hodnotu stavu

     inputs:   state // okamžitý stav hry
               game  // popis hry
               α     // nejlepší skóre MAXe podél cesty do state
               β     // nejlepší skóre MINa podél cesty do 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
```

Funkce `Min-Value`:

```
function Min-Value(state,game,α,β) returns minimax hodnotu stavu

     inputs:   state // okamžitý stav hry
               game  // popis hry
               α     // nejlepší skóre MAXe podél cesty do state
               β     // nejlepší skóre MINa podél cesty do 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
```

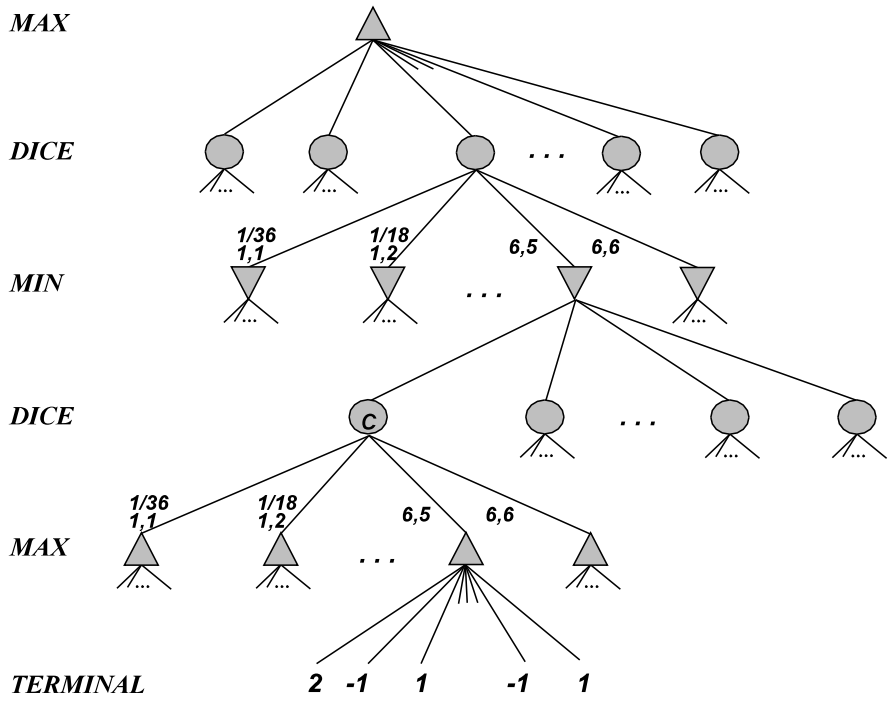
Pozn. 1: `Max-Value` volá `Min-Value`, `Min-Value` volá `Max-Value`.

Pozn. 2: Hraní her strojem je značně ovlivňováno *empirickými vlivy*, např. evaluace listů nejsou ve stromu rozmístěny náhodně. Velká chyba v nadřazeném uzlu má za následek, že sourozenci aktuálního uzlu (potomek chybového) mají hodnoty ve značné korelaci. Závisí to také na kořenové pozici a konkrétní hře. To má nepříznivý vliv na exaktní matematickou analýzu hry.

Hry obsahující prvek náhody

Na rozdíl od šachu existují v mnoha situacích reálného života události, které mohou dát vznik nepředvídatelnému stavu (ve smyslu, že nelze přesně říci, co bude následovat). U her lze napodobit vliv náhodných elementů např. hody kostkou. To má za následek, že hráč nemůže vytvořit kompletní strom vývoje hry z hlediska toho, co se stane po provedení určitého tahu (např. piškvorky vytvoření kompletního stromu hry umožňují).

Hry s náhodným prvkem musí kromě uzlů MAX a MIN obsahovat **uzly náhodné**, které jsou na obrázku znázorněny kolečky:



Pozn.: *dice* je hod kostkou.

Obrázek se vztahuje ke hře *backgammon* (česky asi *vrhcáby*), kde se používá hodu dvou kostek, tj. 36 náhodných možností (se stejnou pravděpodobností) ovlivňuje hru; zde ovšem je např. hod 5-6 totéž jako 6-5, takže počet *různých* hodů je omezen na 21 (šest dvojic 1-1, ..., 6-6 má pravděpodobnost 1/36, zbylých 15 dvojic má 1/18).

Cílem je opět vybrat tah z možností A_1, ..., A_n, které vedou k nejlepší pozici. Každá možná pozice ovšem nemá určitou hodnotu minimax—místo toho lze spočítat jen **očekávanou hodnotu**, kde průměr se bere přes všechny možné hodnoty hodu kostkami.

Pro koncové uzly se použije funkce zisku stejně jako u deterministických her. O úroveň výš jsou náhodné uzly. Na předchozím obrázku je jako příklad uzel označený ***C***, pro nějž lze stanovit **očekávanou max-hodnotu** označenou jako *expectimax(C):*

Nechť d_i je nějaká možná hodnota hodu oběma kostkami, $P(d_i)$ pravděpodobnost obdržení této hodnoty. Pro každý hod se spočítá zisk nejlepšího tahu pro MIN, a pak se sečtou zisky váhované pravděpodobností, že dojde k určité hodnotě hodu kostkami.

Soubor pozic generovaných korektními tahy nechť je $S(C, d_i)$ pro $P(d_i)$ vzhledem k aktuální pozici *C*. Potom pro očekávanou max-hodnotu platí:

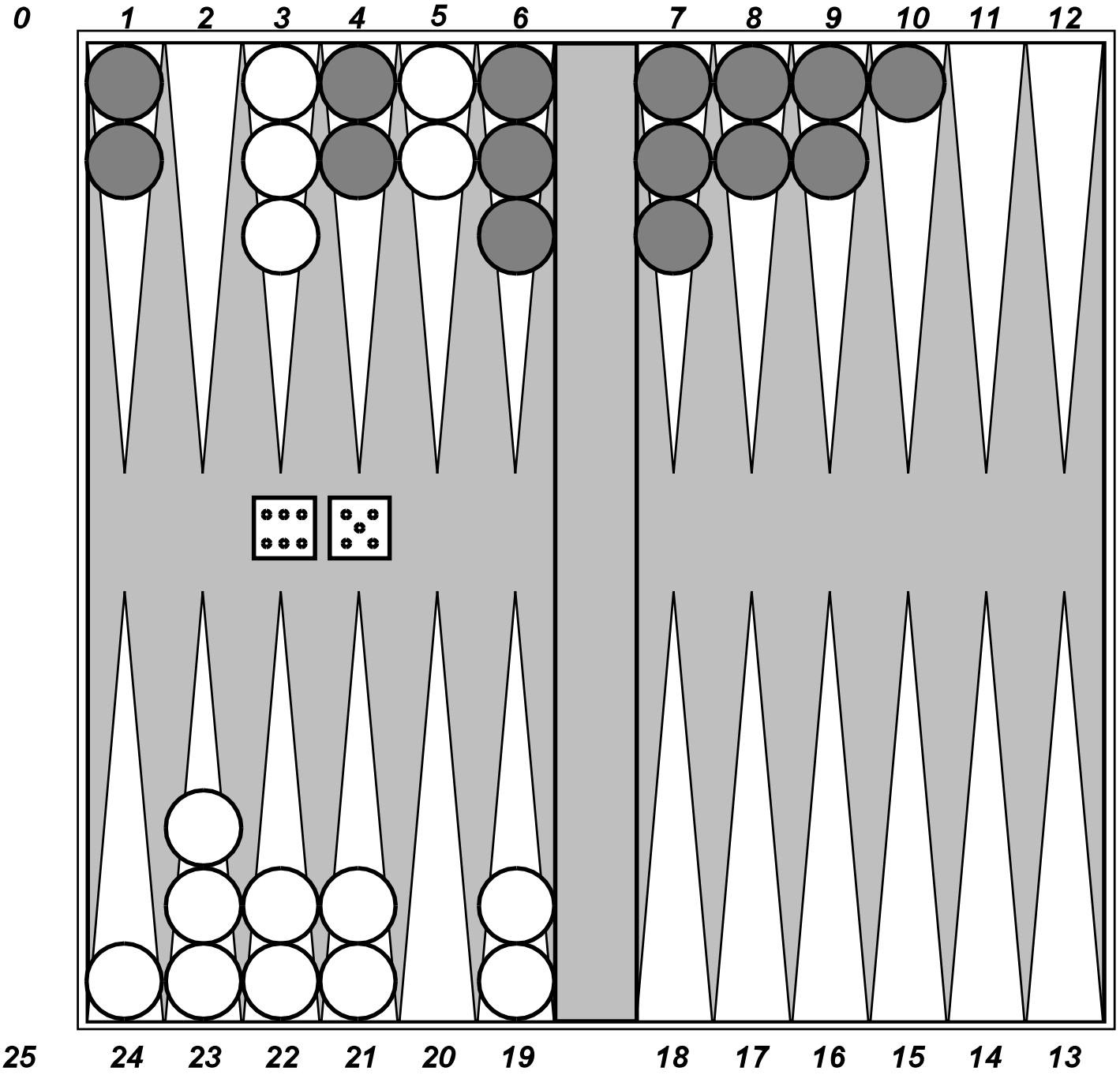
$$expectimax(C) = \sum_i P(d_i) \max_{s \in S(C,d_i)} (zisk(s))$$

Vztah dává očekávaný zisk pozice *C* za předpokladu nejlepší hry z obou stran.

O další úroveň výš (uzly označené symbolem ▼) lze aplikovat normální minimax, protože náhodným (pravděpodobnostním) uzlům byly již přiřazeny hodnoty zisku. Další úroveň výš vyžaduje výpočet **očekávané min-hodnoty** *expectimin(B)* pro nějaký uzel *B* obdobně výpočtu *expectimax*. Proces lze aplikovat rekursívně směrem k vrcholu stromu, vyjma vrcholu, kde jsou hodnoty hodu kostkou již známy.

Pozn.: Šachové programy hrající s velkou silou, na úrovni mezinárodních velmistrů, jsou v současnosti běžným komerčním produktem, viz např. informace na URL `http://www.chessbase.com`, a to i pro víceprocesorové počítače umožňující paralelní vyhledávání ve více větvích současně (např. program s komerčním názvem `Deep Fritz 7`). IBM experimentálně použila speciálně vyvinuté programy `Deep Thought` (sponsoring vývoje na Carnegie Mellon University v USA) a `Deep Blue`, kdy došlo k porážce mistra světa Gary Kasparova. Síle programů také značně pomáhá vysoká výkonnost současných počítačů, i když není rozhodující. V r. 1992 Gerry Thesauro pomocí strojového učení a umělých neuronových sítí vytvořil pro `backgammon` evaluační funkci, takže program má stabilní výkonnost, která ho řadí mezi první tři světové hráče. Japonská hra `go`, kde *b* se blíží 360, zcela likviduje běžné vyhledávací metody; není dosud vyřešena (a na program, který porazí nejlepšího hráče-člověka, je vypsána odměna 2 miliony US dolarů).

Ilustrace typické pozice ve hře backgammon:



Cílem hry je pro každého hráče přesunout všechny hrací kameny mimo hrací desku. Bílý hraje ve směru hodinových ručiček směrem k číslu 25, černý opačným směrem k 0. Kámen se může přesunout na libovolnou pozici vyjma té, kde jsou dva nebo více kameny protivníka. Přesun na pozici s jedním kamenem protivníka znamená, že protivníkův kámen musí začít znovu od počátku. V ukázané pozici bílý právě hodil 6-5 a má 4 korektní tahy: (5-10, 5-11), (5-11, 19-24), (5-10, 10-16) a (5-11, 11-16).