Řešení úloh

(Teorie a algoritmy hledání řešení úloh)

22. 2. – 7. 3. 2012

Řešení úloh

První základní otázka: Co je to úloha ?
Jak lze úlohu definovat ?

Odpověď:

Mějme dány dvě množiny stavů:

$\mathfrak{X} = \{ x_1, x_2, \ldots x_K \}$ … množina výchozích stavů

$\mathfrak{Y} = \{ y_1, y_2, \ldots y_M \}$ … množina cílových stavů

Řešením úlohy pak rozumíme postup (posloupnost operací), kterým (kterými) převedeme úlohu z některého výchozího stavu x_i do definovaného cílového stavu y_j .

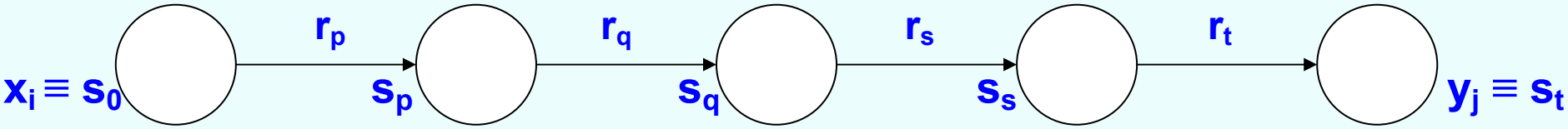
Obecně úlohu definujeme jako zobrazení

$$\mathfrak{X} \rightarrow \mathfrak{Y} \ .$$

Máme-li definován jen jeden konkrétní výchozí a jeden cílový stav úlohy, nabývá uvedené zobrazení tvar

$$x_i \rightarrow y_j \ .$$

Grafem reprezentujeme úlohu jako posloupnost stavů



s_p, s_q, s_s jsou vnitřní stavy (mezistavy) úlohy,

r_p, r_q, r_s,… jsou operátory popisující elementární operace.

Definujme množinu elementárních operátorů

$$RULES = \{ \mathbf{r_1}, \mathbf{r_2}, \ldots \mathbf{r_{L-1}}, \mathbf{r_L} \} .$$

Pak naši úlohu z předchozího obrázku lze vyjádřit

$$x_i \xrightarrow{R_{Kij}} y_j ,$$

kde $R_{Kij} = \mathbf{r_p}\ \mathbf{r_q}\ \mathbf{r_s}\ \mathbf{r_t}$ je kompoziční operátor,

$\mathbf{r_p}, \mathbf{r_q}, \mathbf{r_s}, \mathbf{r_t} \in RULES$, p,q,s,t ∈ { 1, …, L } .

Je-li více výchozích a cílových stavů, píšeme

$$\mathfrak{X} \xrightarrow{\mathfrak{R}} \mathfrak{Y} ,$$

$\mathfrak{R} = \{ R_{Kij} \}$ … množina všech kompozičních operátorů.

Definice: Úlohou potom nazveme trojici

$$(\mathfrak{X} , \mathfrak{Y} , \mathfrak{R}) ,$$

v níž vždy známe dvě složky a třetí určujeme.

Podle neznámé složky rozlišujeme úlohy:

- $(\mathfrak{X} , ? , \mathfrak{R})$ … deduktivní
- $(? , \mathfrak{Y} , \mathfrak{R})$ … abduktivní
- $(\mathfrak{X} , \mathfrak{Y} , ?)$ … induktivní

Pozn.: Abduktivní úloha poskytuje exaktní řešení pouze tehdy, je-li R_{Kij} bijektivní zobrazení (pro všechna $R_{Kij} \in \mathfrak{R}$).

Umělá inteligence vyšetřuje skupinu induktivních úloh, u nichž člověk musí formulovanou hypotézu řešení zpětně deduktivně prověřit, zda vyhovuje zadaným množinám stavů $\mathfrak{X}$ a $\mathfrak{Y}$.

Dva způsoby výběru vhodné hypotézy z $\mathfrak{R}$:

1. Apriorně zvolíme množinu operátorů, kterou parametrizujeme (nastavením vhodných parametrů vybereme takový kompoziční operátor, který vyhovuje zadaným množinám stavů $\mathfrak{X}$ a $\mathfrak{Y}$).
2. Apriorně zvolíme množinu elementárních operátorů, ze kterých se postupně snažíme „složit“ výsledný kompoziční operátor vyhovující zadaným množinám stavů $\mathfrak{X}$ a $\mathfrak{Y}$.

Poznámka: Induktivní úlohy jsou zpravidla nedeterministické .

Hledání řešení úlohy – hledání („sestavení“) takového kompozičního operátoru R_{Kij}, který vyhovuje zadaným množinám stavů $\mathfrak{X}$ a $\mathfrak{Y}$ a je v nějakém (obvykle daném) smyslu optimální.

Postup:

Metodou pokusů a omylů vytváříme strom řešení úlohy a současným jeho prohledáváním hledáme takový kompoziční operátor R_{Kij}, který vyhovuje množinám stavů $\mathfrak{X}$ a $\mathfrak{Y}$.

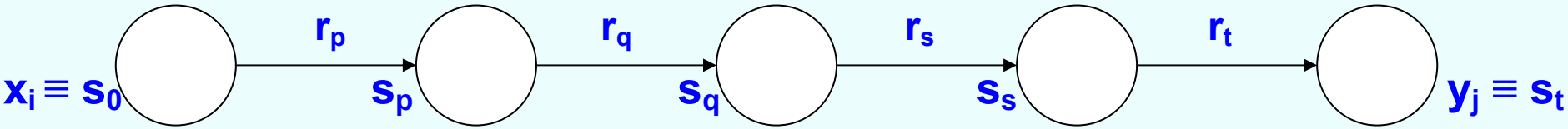
Procházení (prohledávání) stromu řešení:

- deterministické
- náhodné
- heuristické

Reprezentace úlohy

Úloha ($\mathfrak{X}$, $\mathfrak{Y}$, ?) má zpravidla definovánu množinu výchozích stavů, v nichž se nachází, a množinu cílových stavů, jichž má být řešením úlohy dosaženo – úlohu budeme reprezentovat ve stavovém prostoru .

Řekli jsme, že grafem reprezentujeme úlohu jako posloupnost stavů,



kde s_p, s_q, s_s jsou vnitřní stavy (mezistavy) úlohy,

$r_p, r_q, r_s, \ldots$ jsou operátory popisující elementární operace.

Zobecněním dostaneme metodu stavového prostoru .

Metoda reprezentace úlohy ve stavovém prostoru

Předpoklady:

1. Existuje konečná množina stavů, v nichž se úloha může nacházet:
   $S = \{ s_i \}$, $i = 0, 1, \ldots, N$.
2. Existuje alespoň jeden výchozí (počáteční) stav úlohy $s_0 \in S$.
3. Existuje konečná množina cílových (požadovaných) stavů úlohy
   $G = \{ g_j \}$, $j = 1, 2, \ldots, M$ taková, že $G \subseteq S$, $M \leq N$.
4. Existuje konečná množina elementárních operátorů
   $RULES = \{ r_l \}$, $l = 1, 2, \ldots, L$,
   které převádějí úlohu ze stavu s_p do stavu s_q, $p, q = 0, 1, \ldots, N$.

Potom:

Stavový prostor úlohy je definován dvojicí

$$(\mathbf{S} , \mathit{RULES}) .$$

Konkrétní řešení úlohy je definováno trojicí

$$(\mathbf{s_0} , \mathbf{s_j} , R_{Koj}) \text{ na } \mathbf{S} ,$$

kde R_{Koj} je kompoziční operátor, který převede úlohu z počátečního stavu $\mathbf{s_0}$ do stavu cílového $\mathbf{s_j} = \mathbf{g_k}$, čili

$$\mathbf{s_0} \xrightarrow{R_{Koj}} \mathbf{s_j} = \mathbf{g_k} \in \mathbf{G} , \; j = 0, 1, \ldots, N , \; k = 1, \ldots, M .$$

Poznámka: Nachází-li se úloha na počátku řešení ve stavu, který je žádaným cílovým stavem, lze řešení úlohy formálně zapsat

$$\mathbf{s_0} \xrightarrow{R_{Koo}} \mathbf{s_0} = \mathbf{g_1}$$

a R_{Koo} budiž prázdný kompoziční operátor, čili $R_{Koo} = \mathbf{r}_\varepsilon$, kde $\mathbf{r}_\varepsilon$ je prázdná elementární operace.

Řešením úlohy ve stavovém prostoru pak budiž kompoziční operátor

$$R_{Koj} = \mathbf{r}_1\, \mathbf{r}_2\, \mathbf{r}_3 \ldots \mathbf{r}_{r-1}\, \mathbf{r}_r$$

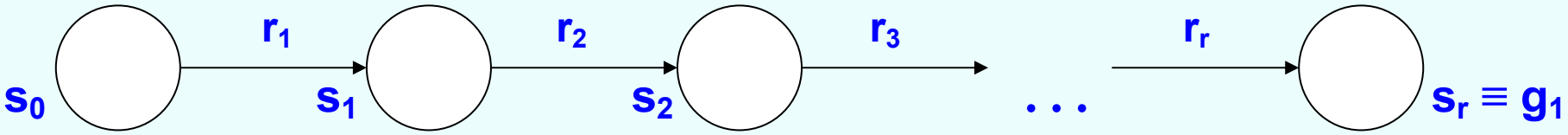
takový, že

$\mathbf{s}_1 \leftarrow \mathbf{r}_1(\mathbf{s}_0)$,

$\mathbf{s}_2 \leftarrow \mathbf{r}_2(\mathbf{s}_1) = \mathbf{r}_2(\mathbf{r}_1(\mathbf{s}_0))$, (zapíšeme $\mathbf{r}_2\, \mathbf{r}_1(\mathbf{s}_0)$)

. . . ,

$\mathbf{s}_r \leftarrow \mathbf{r}_r(\mathbf{s}_{r-1}) = \mathbf{r}_r(\mathbf{r}_{r-1}(\ldots \mathbf{r}_1(\mathbf{s}_0))$.



Hledání řešení úlohy

Př.:

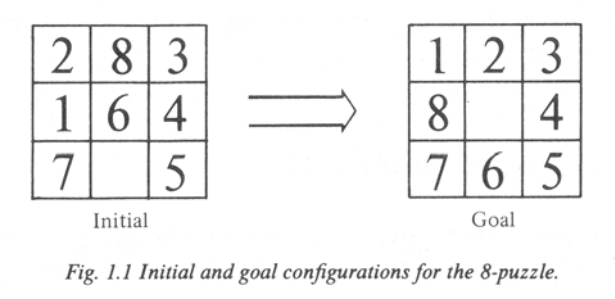
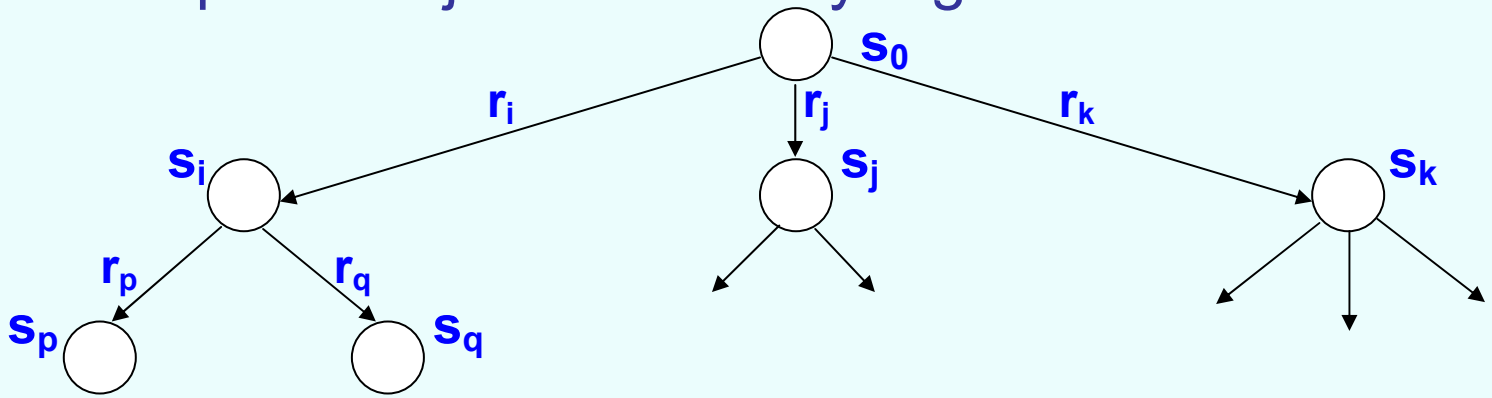


Fig. 1.1 Initial and goal configurations for the 8-puzzle.

Úlohu reprezentujeme stromovým grafem



kde uzly grafu reprezentují stavy úlohy,
hrany grafu reprezentují přechody mezi stavy (aplikace elementárních operátorů)

Definice: Uzel, do něhož vede bezprostřední hrana, je nazýván bezprostředním následovníkem – uzly s_i , s_j , s_k jsou bezprostředními následovníky uzlu s_0 , uzly s_p , s_q jsou bezprostředními následovníky uzlu s_i atd.

Definice: Nalezení všech bezprostředních následovníků uzlu s_k se nazývá expanzí uzlu s_k .

Definice: Počet hran vedoucích z uzlu s_0 do uzlu s_p definuje hloubku uzlu s_p .

STROM ŘEŠENÍ ÚLOHY

Definice:

Strom řešení úlohy je orientovaný graf definovaný takto:

1. Graf má jediný uzel bez bezprostředního předchůdce – **kořen**. Tento uzel reprezentuje **výchozí stav** úlohy.
2. U každého uzlu definujeme rekurzivně **hloubku** uzlu:
   - kořen má hloubku **0** ,
   - má-li uzel hloubku **d** , má každý jeho bezprostřední následovník hloubku **d+1** .
3. Uzly, které nemají žádného bezprostředního následovníka (listy), reprezentují
   - **cílový stav** úlohy,
   - stavy, na něž nelze aplikovat žádný z operátorů,
   - stavy, o nichž jsme usoudili, že jsou z nějakých důvodů neperspektivní (jejich další rozvíjení nemá smysl).

4. Orientovaná hrana reprezentuje přechod úlohy z daného (aktuálního) stavu do stavu nového – **aplikaci elementárního operátoru**.

Aplikaci všech možných elementárních operátorů na stav úlohy, jíž získáme všechny možné následující stavy úlohy (bezprostřední následovníky), nazveme **expanzí uzlu**.

Nalezení řešení úlohy = nalezení cesty spojující počáteční uzel grafu řešení úlohy (výchozí stav, kořen grafu) s listem reprezentujícím cílový (žádaný) stav úlohy.

PRODUKČNÍ SYSTÉM

Produkční systém se skládá z:

- **databáze** úlohy obsahující **fakta**
- **báze znalostí** obsahující **produkční pravidla** ve tvaru

  podmínka ⟶ akce

- **řídicího mechanismu** – jehož úkolem je:
  - provést volbu, které aplikovatelné pravidlo bude použito,
  - vybrat fakta z databáze, která budou dosazena do podmínky zvoleného produkčního pravidla,
  - ukončit řešení (výpočet), je-li splněna cílová podmínka
- **množiny cílů**, které mají být splněny

Cílová podmínka produkčního systému:

a) explicitní, odvozená z množiny cílů,

b) implicitní, nejde-li na daný obsah databáze aplikovat žádné další produkční pravidlo.

Pracovní režimy řídicího mechanismu:

- přímochody
- zpětnochody

ZÁKLADNÍ PROCEDURA PRODUKČNÍHO SYSTÉMU

$DATA \longleftarrow$ výchozí stav ; { poč. stav databáze }

repeat

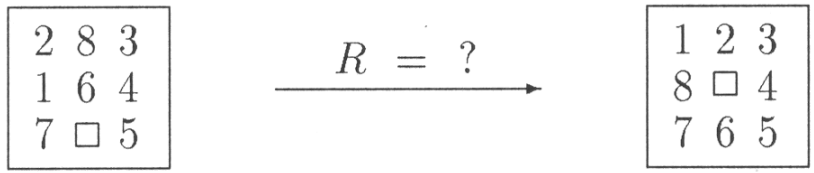
  select (nějaké) pravidlo $r_i \in RULES$, které je aplikovatelné na $DATA$;

  $DATA \longleftarrow r_i\,(DATA)$; { aplikace r_i na $DATA$ }

until $DATA$ splňují cílovou podmínku;

INDETERMINISMUS

Př.: Hlavolam „8“



Pravidla hry můžeme obecně zapsat jako

if podmínka **then** akce

nebo ve tvaru

podmínka → akce .

Pravidla pro přesouvání kamenů (prázdné políčko nazvěme „blank“):

"blank" není v horním řádku	⟶	posuň "blank" nahoru
"blank" není v dolním řádku	⟶	posuň "blank" dolů
"blank" není v levém sloupci	⟶	posuň "blank" doleva
"blank" není v pravém sloupci	⟶	posuň "blank" doprava

ŘÍDICÍ MECHANISMY (řídicí strategie)

- neodvolatelné (irrevocable)
- pokusné (tentative)

Př.: Výběr pravidel gradientní metodou (hill climbing algorithm):

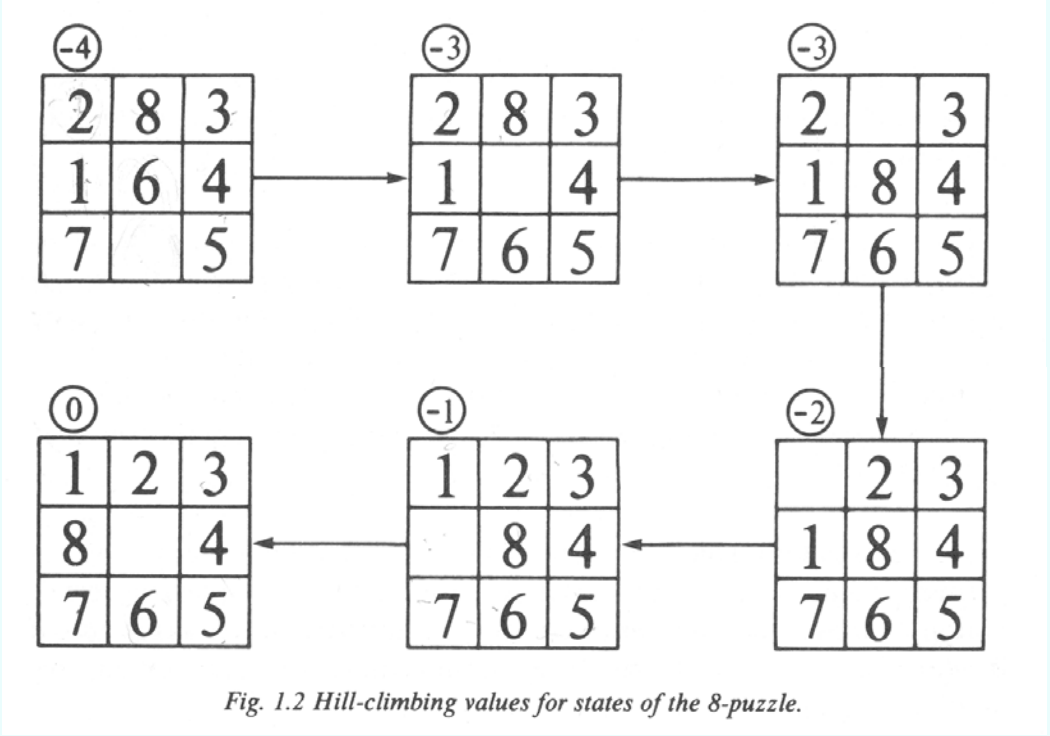


Fig. 1.2 Hill-climbing values for states of the 8-puzzle.

Pokusné řídicí strategie

- metoda (algoritmus) navracení (backtracking)
- metody hledání v grafu řešení úlohy
  - slepé strategie (neinformované)
  - informované strategie (zpravidla heuristické)

ALGORITMUS NAVRACENÍ (BACKTRACKING)

deterministický – prod. pravidla se aplikují ve stanoveném pořadí

- backtracking stavový
- backtracking operátorový (Floydův)

2. Řešení úloh

Recursive procedure **BACKTRACK** (*DATA*)
{*DATA* reprezentuje databázi příslušného stavu řešení}

1. **if TERM** (*DATA*) **then return** *NIL*;

   { **TERM** je logická funkce (predikát) nabývající pravdivostní hodnoty **true** v případě, že obsah databáze *DATA* splňuje cílovou podmínku implementovaného produkčního systému. Jako příznak úspěšného ukončení hledání řešení je procedurou vracen *NIL*, resp. prázdný seznam. }

2. **if DEADEND** (*DATA*) **then return** *FAIL*;

   { **DEADEND** je logická funkce nabývající hodnoty **true** v případě, že mechanismus se dostal do chybového (fail) stavu (podmínky viz výše) a nelze dále úspěšně pokračovat vpřed k hledanému cíli řešení. V takovém případě vrací procedura symbol *FAIL* jako příznak dosažení chybového stavu. }

3. *RULES* ← **APPRULES** (*DATA*);

   { **APPRULES** je funkce určující, která produkční pravidla a v jakém pořadí budou na daný obsah databáze *DATA* aplikována }

4. LOOP: **if NULL** (*RULES*) **then return** *FAIL*;

   { není-li žádné další aplikovatelné pravidlo, končí procedura chybovým (fail) stavem a vrací symbol *FAIL* }

2. Řešení úloh

5. $R \leftarrow$ **FIRST** (*RULES*);

   { aplikováno bude v pořadí první, resp. podle daného kritéria "nejlepší" pravidlo }

6. *RULES* $\leftarrow$ **TAIL** (*RULES*);

   { z posloupnosti aplikovatelných produkčních pravidel je vyjmuto zvolené pravidlo }

7. *NEW_DATA* $\leftarrow$ **R** (*DATA*);

   { na obsah databáze *DATA* je aplikováno produkční pravidlo **R** z posloupnosti *RULES* a je vygenerován nový obsah databáze *NEW_DATA* }

8. *PATH* $\leftarrow$ **BACKTRACK** (*NEW_DATA*);

   { procedura **BACKTRACK** je vyvolána rekursivně pro nový obsah databáze }

9. **if** *PATH* = *FAIL* **then** **goto** LOOP;

   { vrátilo-li vyvolání procedury **BACKTRACK** chybový příznak *FAIL*, přechází na použití dalšího produkčního pravidla (pokud takové existuje) }

10. **return** **CONS** (*R*, *PATH*);

    { v opačném (úspěšném) případě je přidáno nově aplikované produkční pravidlo na čelo seznamu (vrchol zásobníku) reprezentujícího vygenerovanou cestu ve stromu řešení z počátečního uzlu do uzlu aktuálního }

2. Řešení úloh

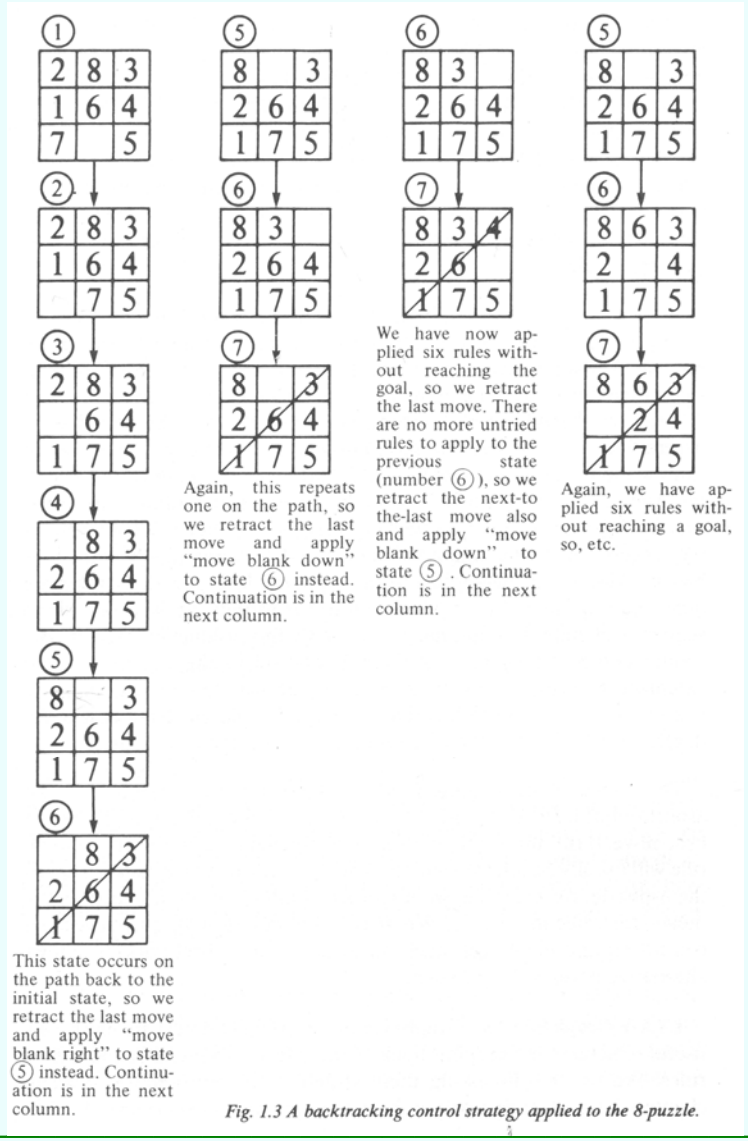


Fig. 1.3 A backtracking control strategy applied to the 8-puzzle.

2. Řešení úloh

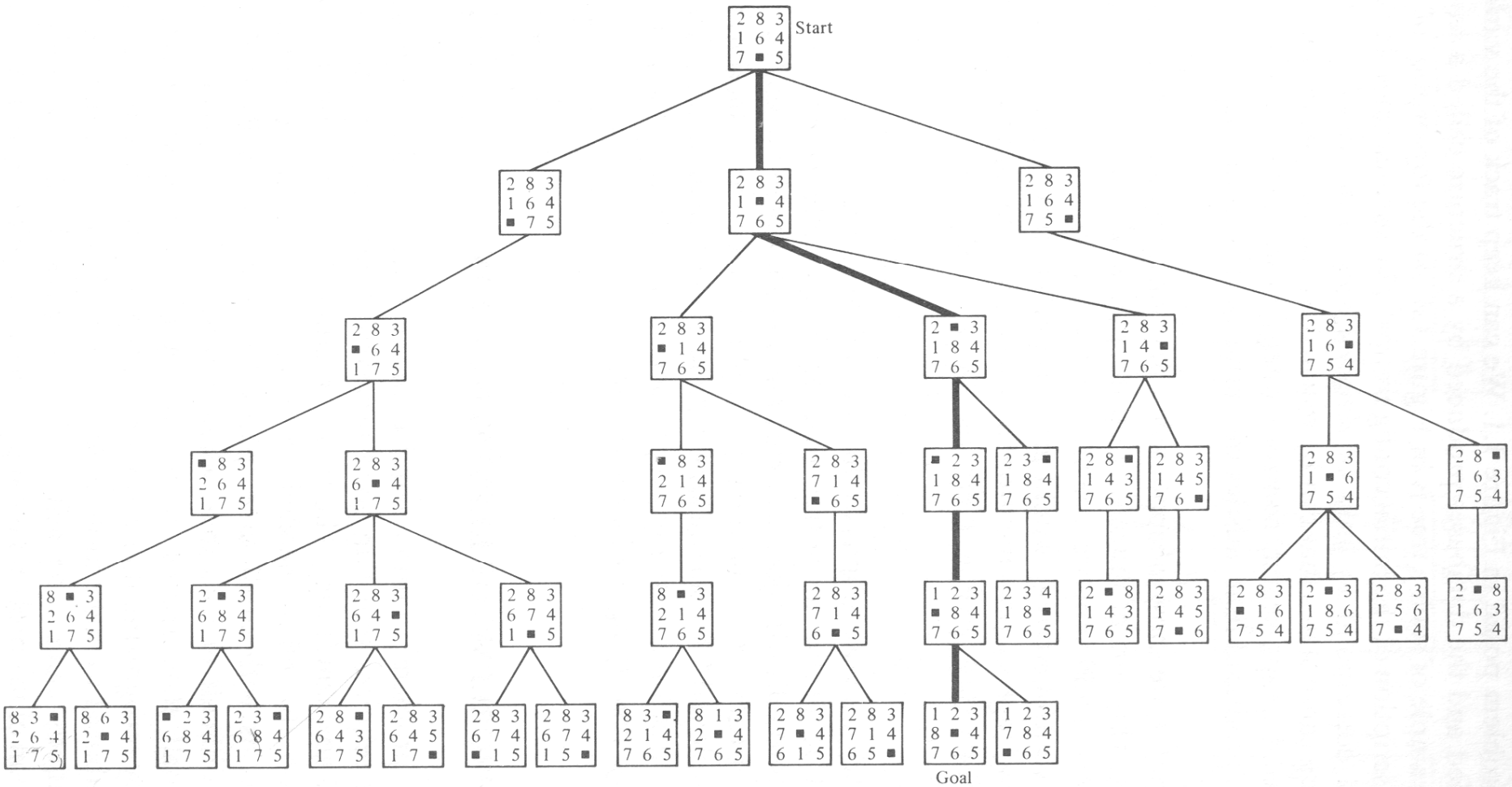


Fig. 1.4 A search tree for the 8-puzzle.

METODY HLEDÁNÍ V GRAFU (GRAPH SEARCH)

Zapamatovávají si celou dosud vygenerovanou část stromu řešení –

- jsou paměťově náročné,
- jsou použitelné účinné globální heuristiky.

Základem metod hledání v grafu jsou:

- expanze uzlu (aplikace všech možných produkčních pravidel na $DATA$ příslušející danému uzlu grafu),
- ohodnocující funkce přiřazující uzlu ohodnocení – zpravidla heuristická *(globální heuristika)*.

Př.: **Problém obchodního cestujícího** – jak objet zákazníky v N místech s minimálními náklady, např.:

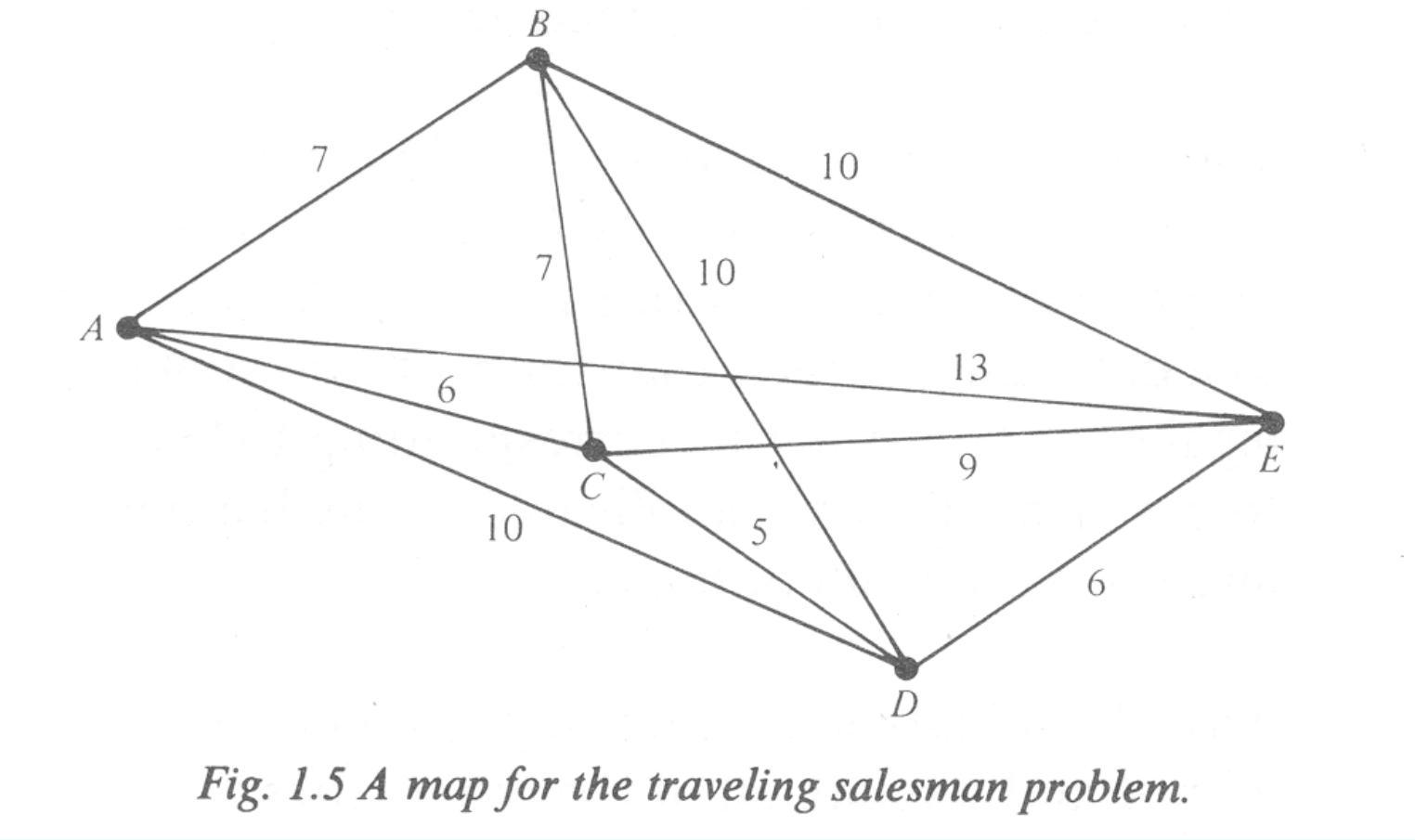


Fig. 1.5 A map for the traveling salesman problem.

Graf řešení úlohy obchodního cestujícího:

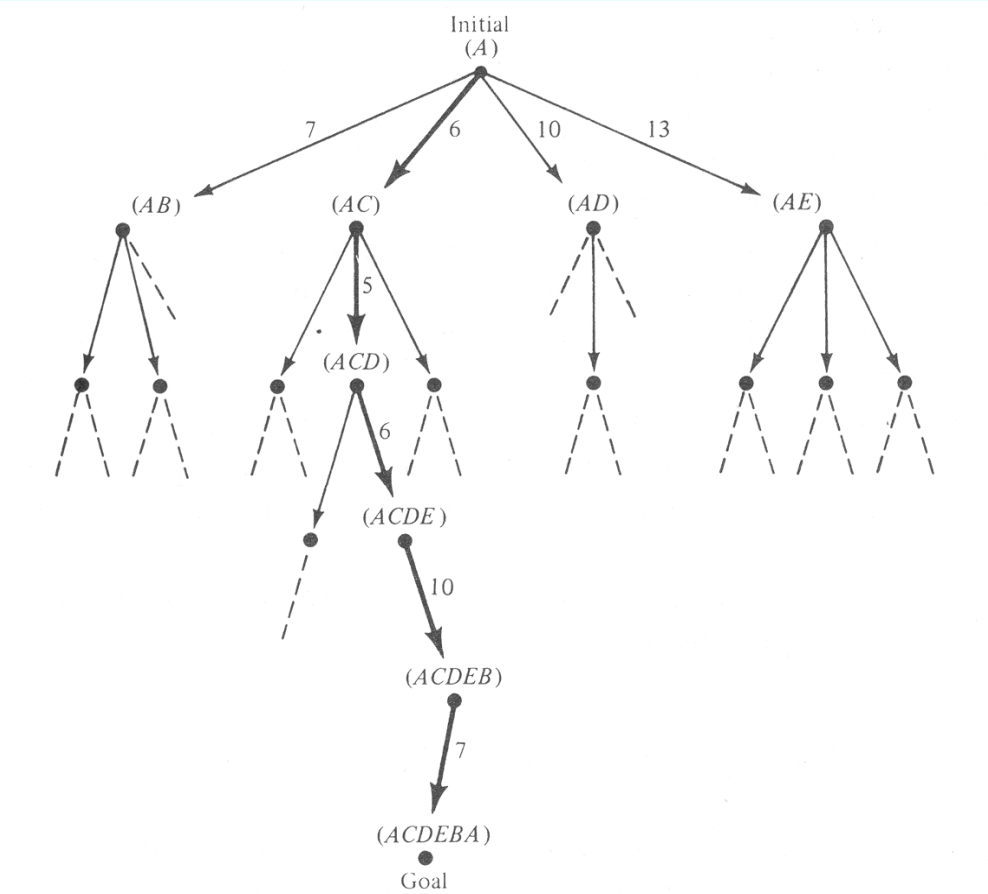


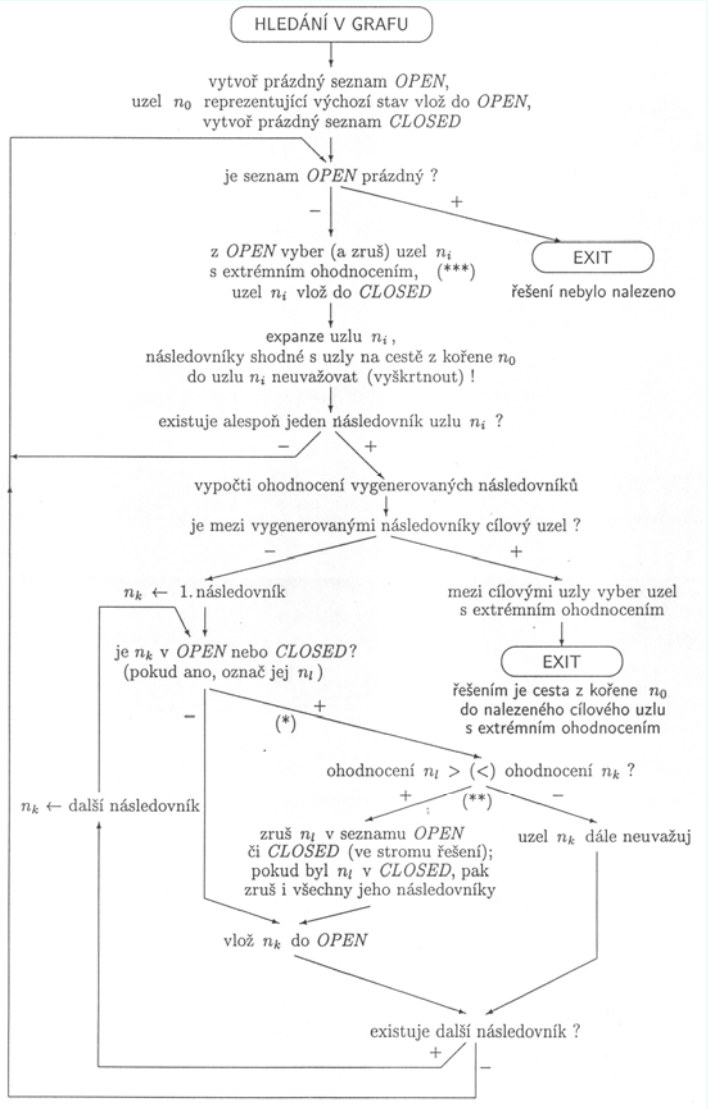
Fig. 1.6 A search tree for the traveling salesman problem.

ALGORITMUS HLEDÁNÍ V GRAFU (GRAPH SEARCH)

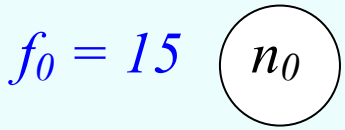
1. Vytvoř seznam $OPEN$, vlož do něj uzel n_0 a urči jeho ohodnocení f_0. Vytvoř prázdný seznam $CLOSED$.
2. Je-li seznam $OPEN$ prázdný, řešení neexistuje, tj. ukonči řešení.
3. Ze seznamu $OPEN$ vyber uzel n_i, který má *extrémní* (minimální nebo maximální) ohodnocení. Uzel n_i zruš v seznamu $OPEN$ a umísti jej do seznamu $CLOSED$.
4. Proveď expanzi uzlu n_i, přičemž ihned vyškrtni všechny bezprostřední následovníky, kteří už jsou předchůdci uzlu n_i. Zbylé bezprostřední následovníky n_j zařaď do seznamu $OPEN$ a urči (vypočti) jejich ohodnocení f_j. Neexistuje-li žádný bezprostřední následovník, přejdi na krok 2.
5. Jsou-li mezi vygenerovanými následovníky uzlu n_i uzly *cílové*, vyber cílový uzel s extrémním ohodnocením a ukonči prohledávání. Ve vygenerovaném stromu řešení najdi cestu od kořene stromu k cílovému uzlu a jako nalezené řešení poskytni posloupnost elementárních operátorů (produkčních pravidel) prováděných podél nalezené cesty. Ukonči řešení.
   V opačném případě pokračuj krokem 6.

6. Pro každého bezprostředního následovníka uzlu n_i, kterého označíme n_j, a jeho ohodnocení f_j, proveď:
   (a) Není-li uzel n_j s vypočteným ohodnocením f_j ani v seznamu $OPEN$ ani v seznamu $CLOSED$ (uzel se ve vygenerovaném stromu řešení dosud nevyskytuje), umísti jej do seznamu $OPEN$.
   (b) Jestliže se uzel n_j s ohodnocením f_j v seznamu $OPEN$ nebo $CLOSED$ již vyskytuje, avšak s ohodnocením $f'_j \neq f_j$ pak
   – je-li $f'_j > (<) f_j$, vypusť uzel n_j s ohodnocením f'_j ze seznamu $OPEN$ nebo $CLOSED$ (ze stromu řešení) a umísti nově vygenerovaný uzel n_j s ohodnocením f_j do seznamu $OPEN$;
   – není-li $f'_j > (<) f_j$, zruš v seznamu $OPEN$ nově vygenerovaný uzel n_j s ohodnocením f_j.
   (c) Dojde-li ke zrušení nějakého uzlu n_j ze seznamu $CLOSED$, musejí být zrušeni rovněž všichni jeho následovníci; přitom je lhostejno, zda se nalézají v seznamu $OPEN$ nebo $CLOSED$ (musejí být zrušeni ve stromu řešení, tj. zlikviduje se celý podstrom s kořenem n_j).
7. Pokračuj krokem 2.

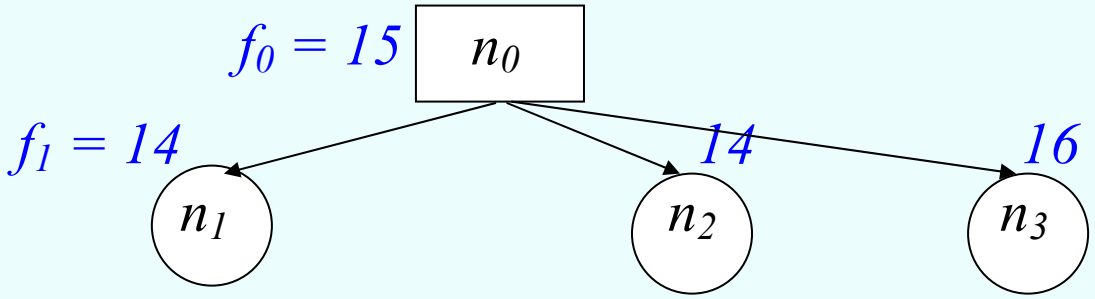
2. Řešení úloh



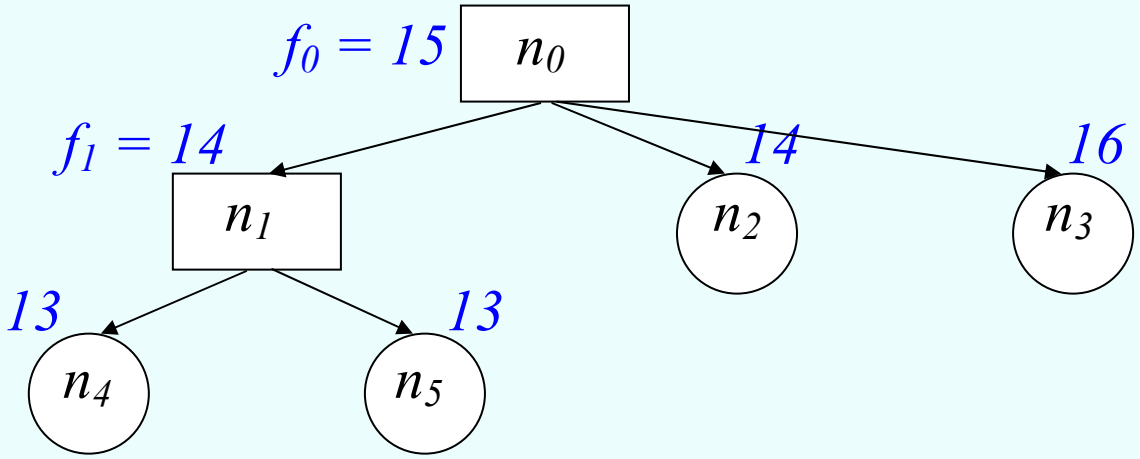
Princip vytváření stromu řešení úlohy:



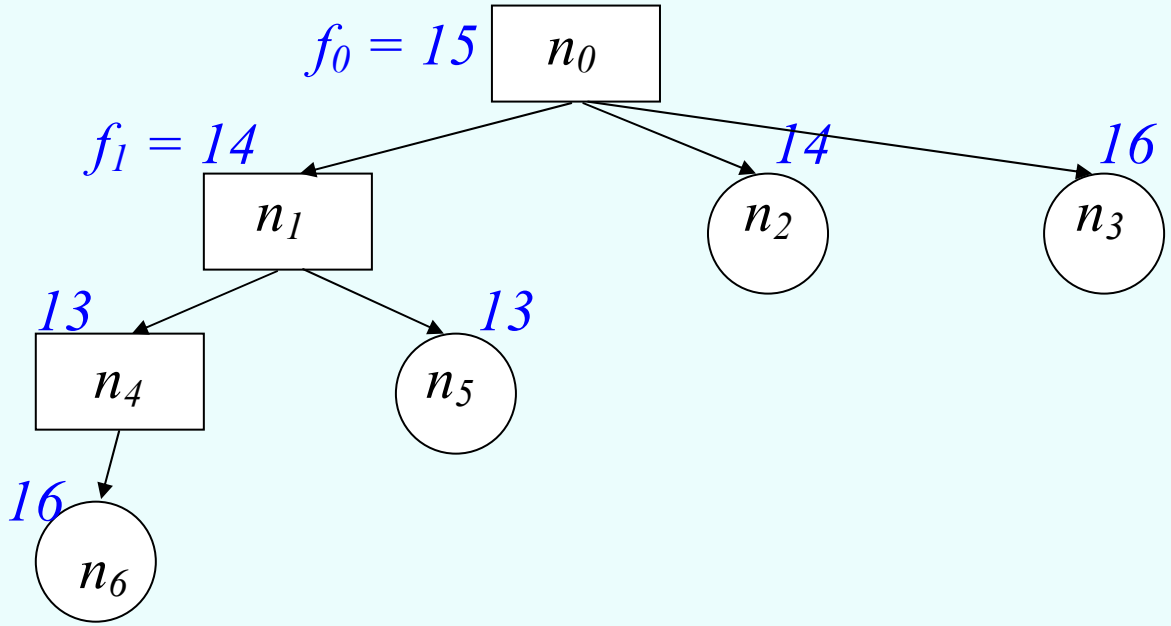
Princip vytváření stromu řešení úlohy:



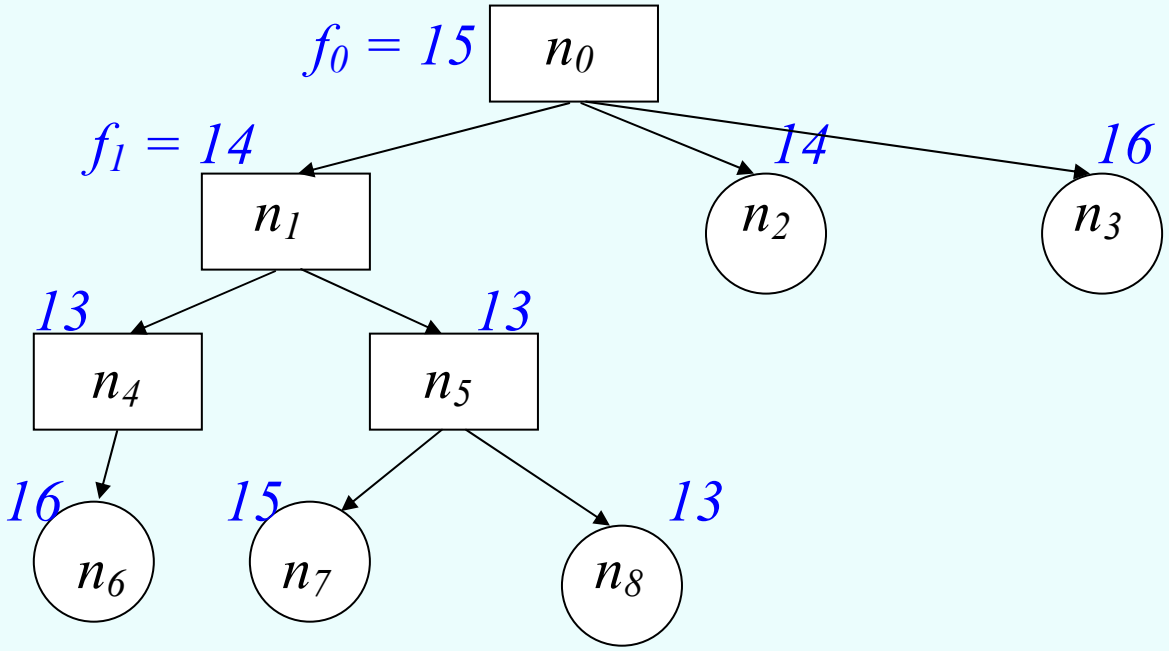
Princip vytváření stromu řešení úlohy:



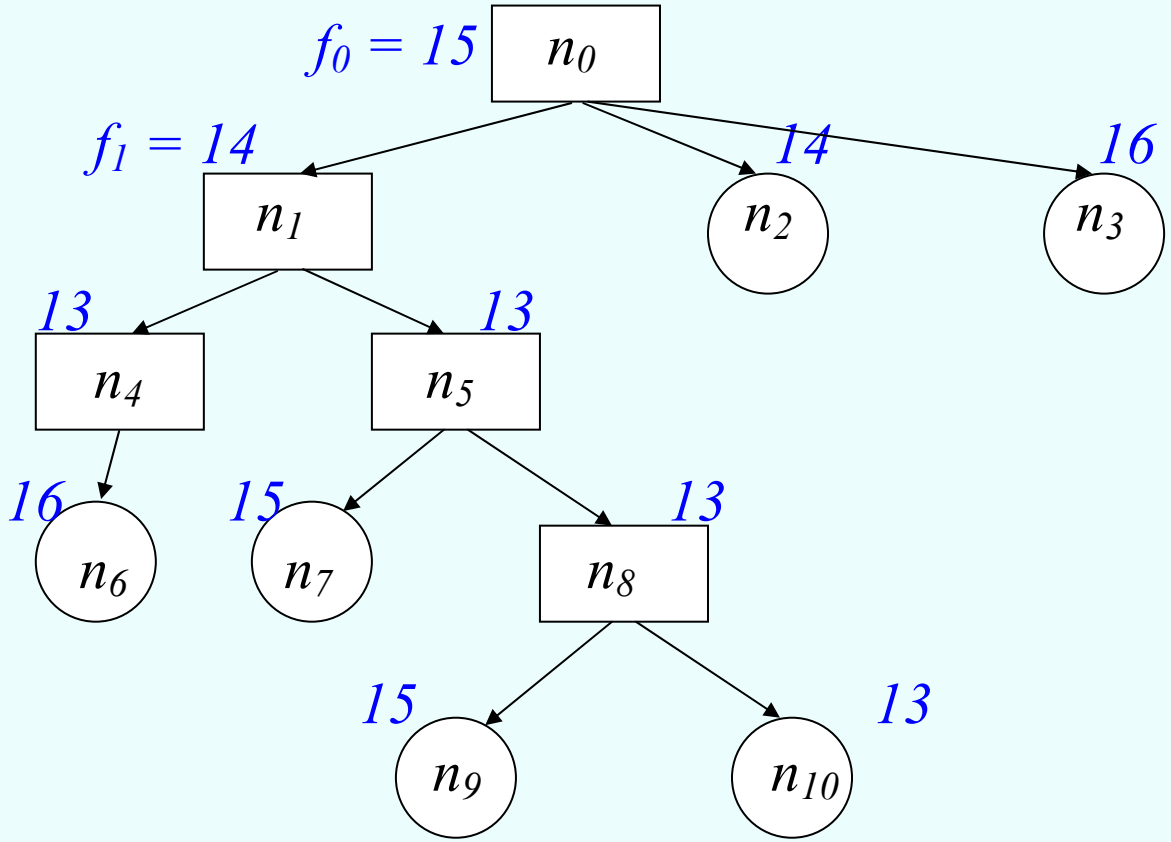
Princip vytváření stromu řešení úlohy:



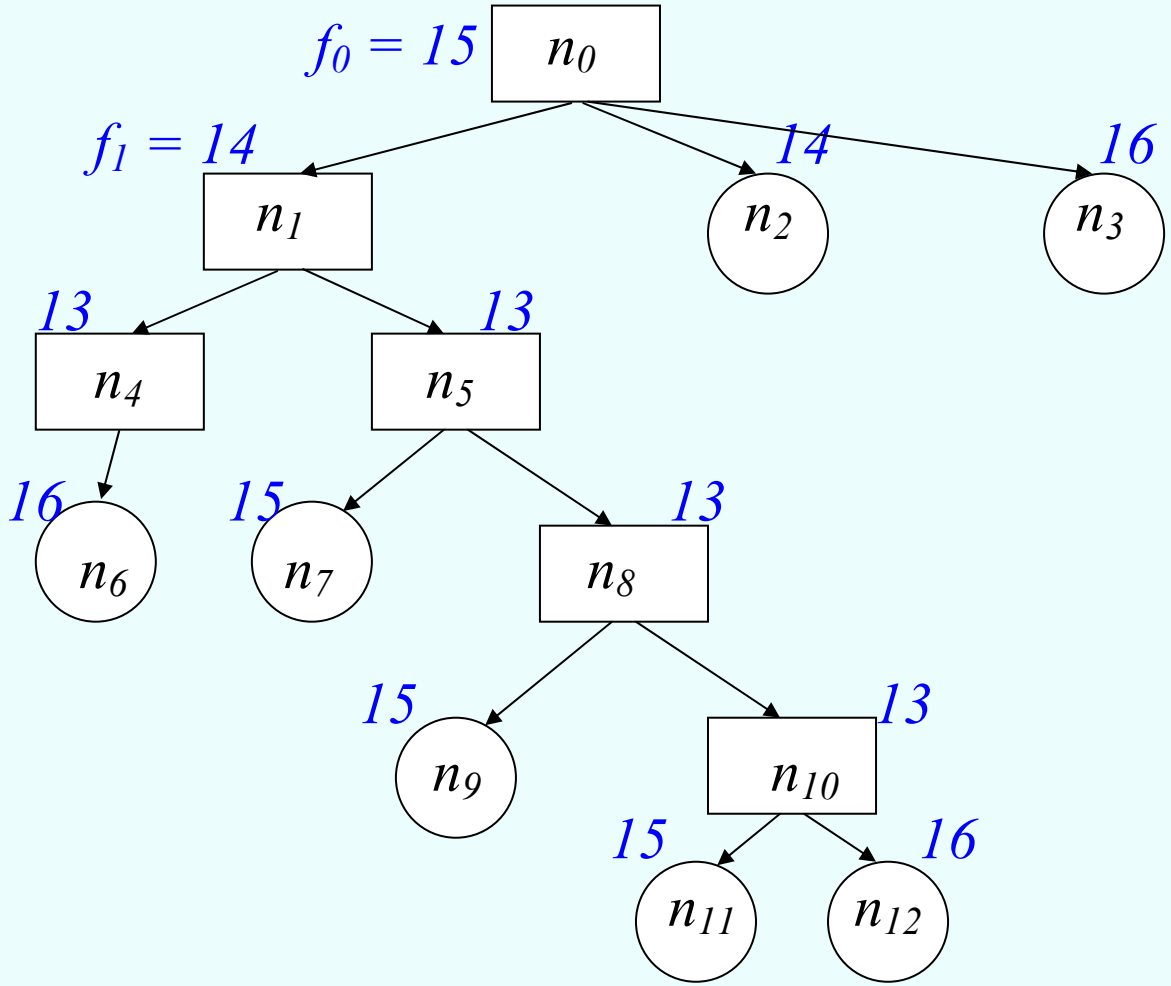
Princip vytváření stromu řešení úlohy:



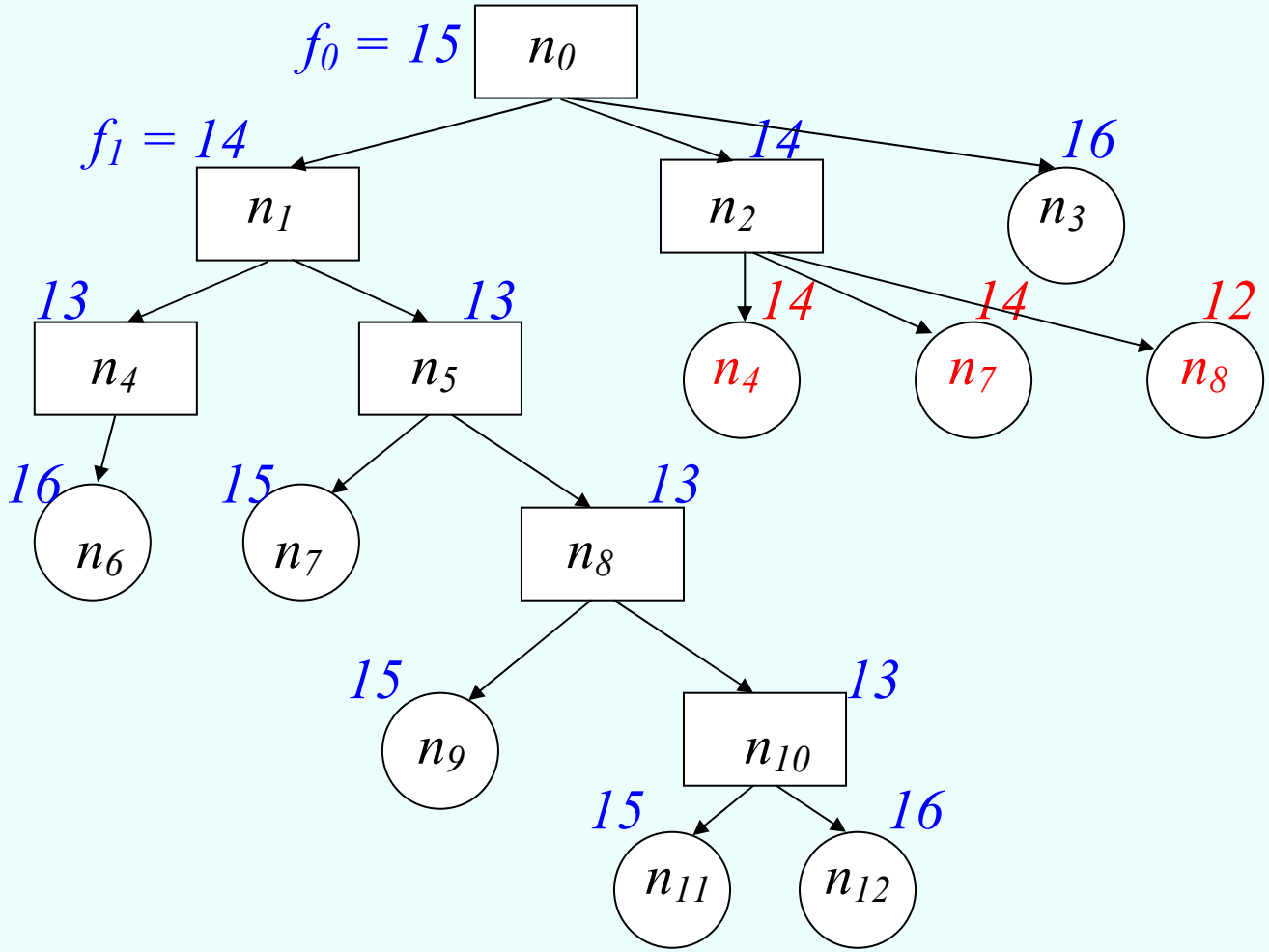
Princip vytváření stromu řešení úlohy:



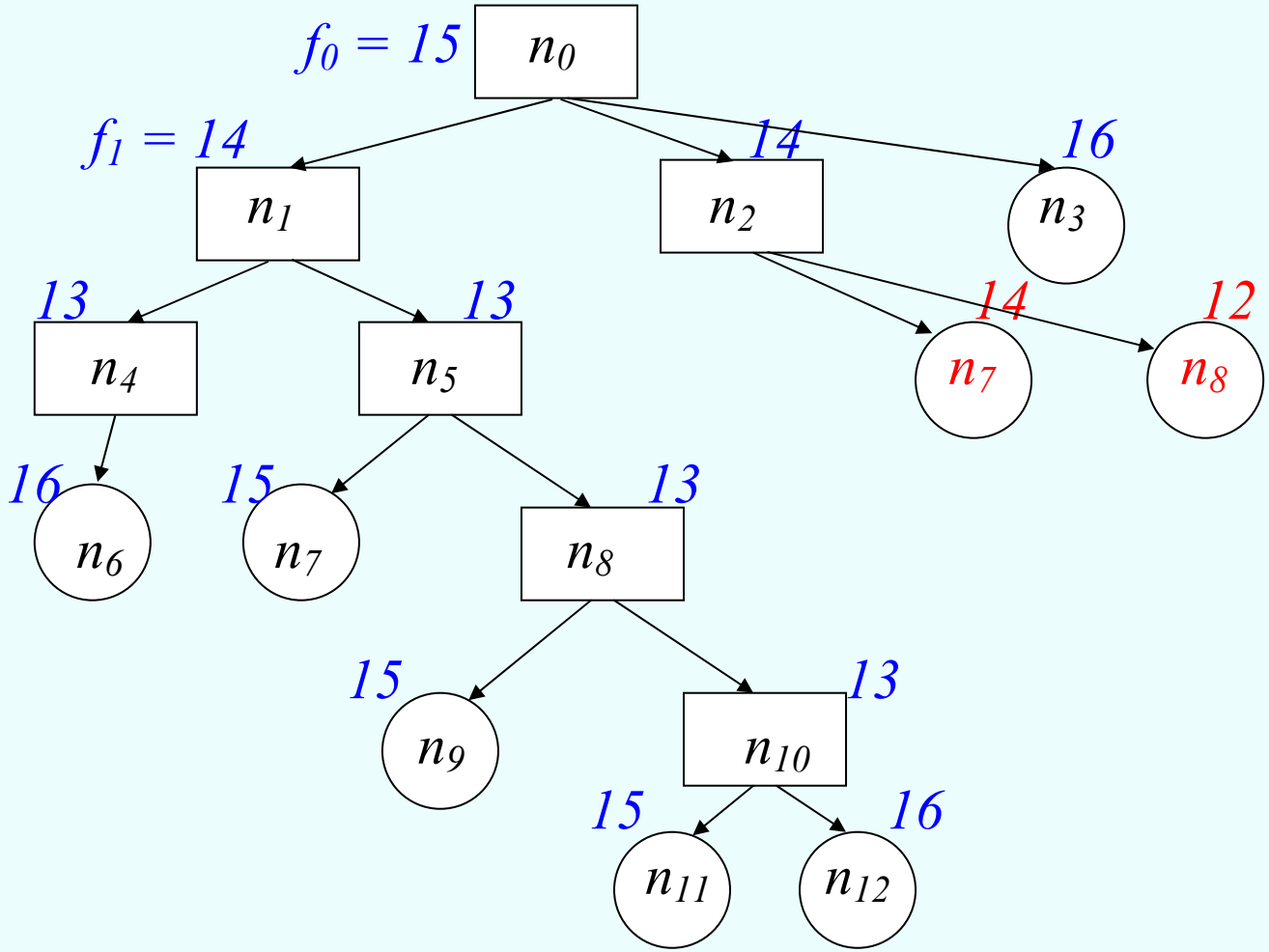
Princip vytváření stromu řešení úlohy:



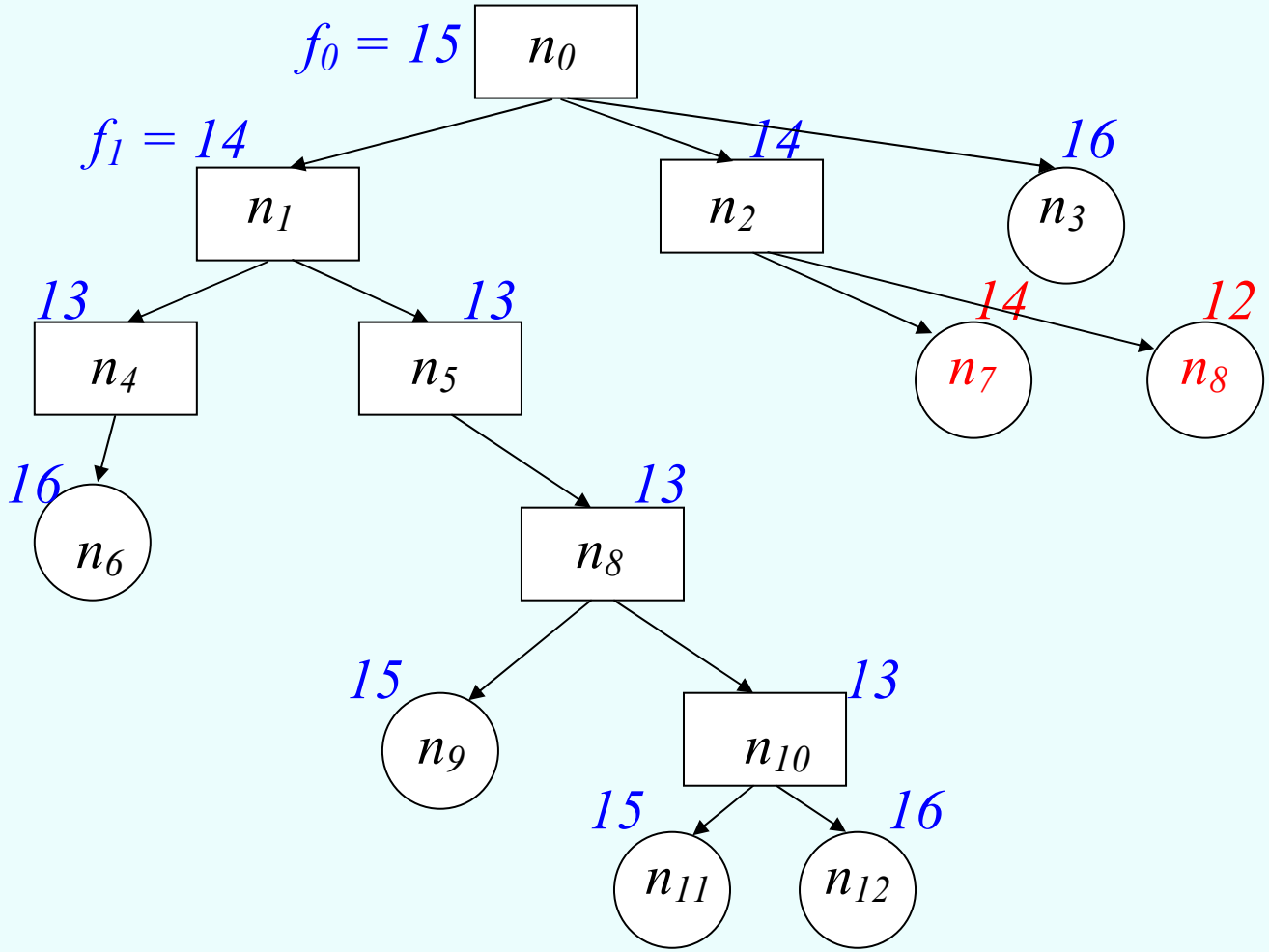
Princip vytváření stromu řešení úlohy:



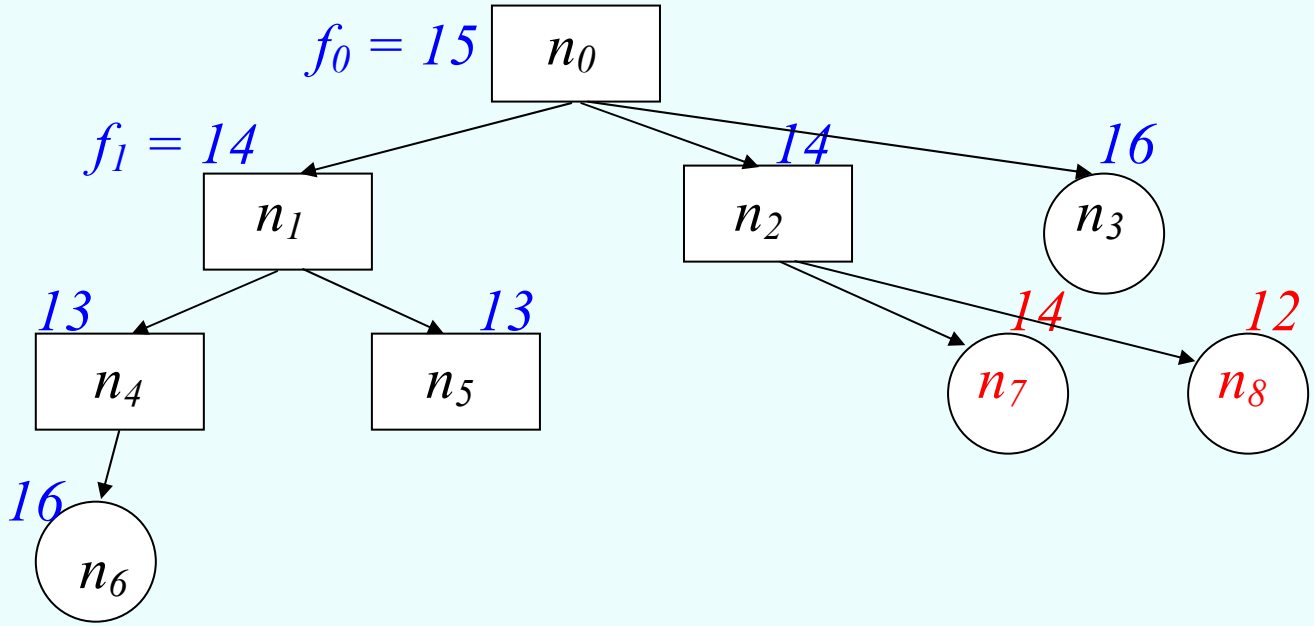
Princip vytváření stromu řešení úlohy:



Princip vytváření stromu řešení úlohy:



Princip vytváření stromu řešení úlohy:



SLEPÉ STRATEGIE HLEDÁNÍ ŘEŠENÍ ÚLOHY

a) do hloubky (depth first search)

Speciální případ základního algoritmu hledání v grafu, v němž platí:

- ohodnocení uzlu = hloubka uzlu,
- v kroku 3 ((***)) se bere maximum,
- v kroku 3 ((***)) je nutno přidat test, zda již bylo dosaženo zadané maximální hloubky; pokud ano, vracíme se ke kroku 2.

Př.: „8“

2. Řešení úloh

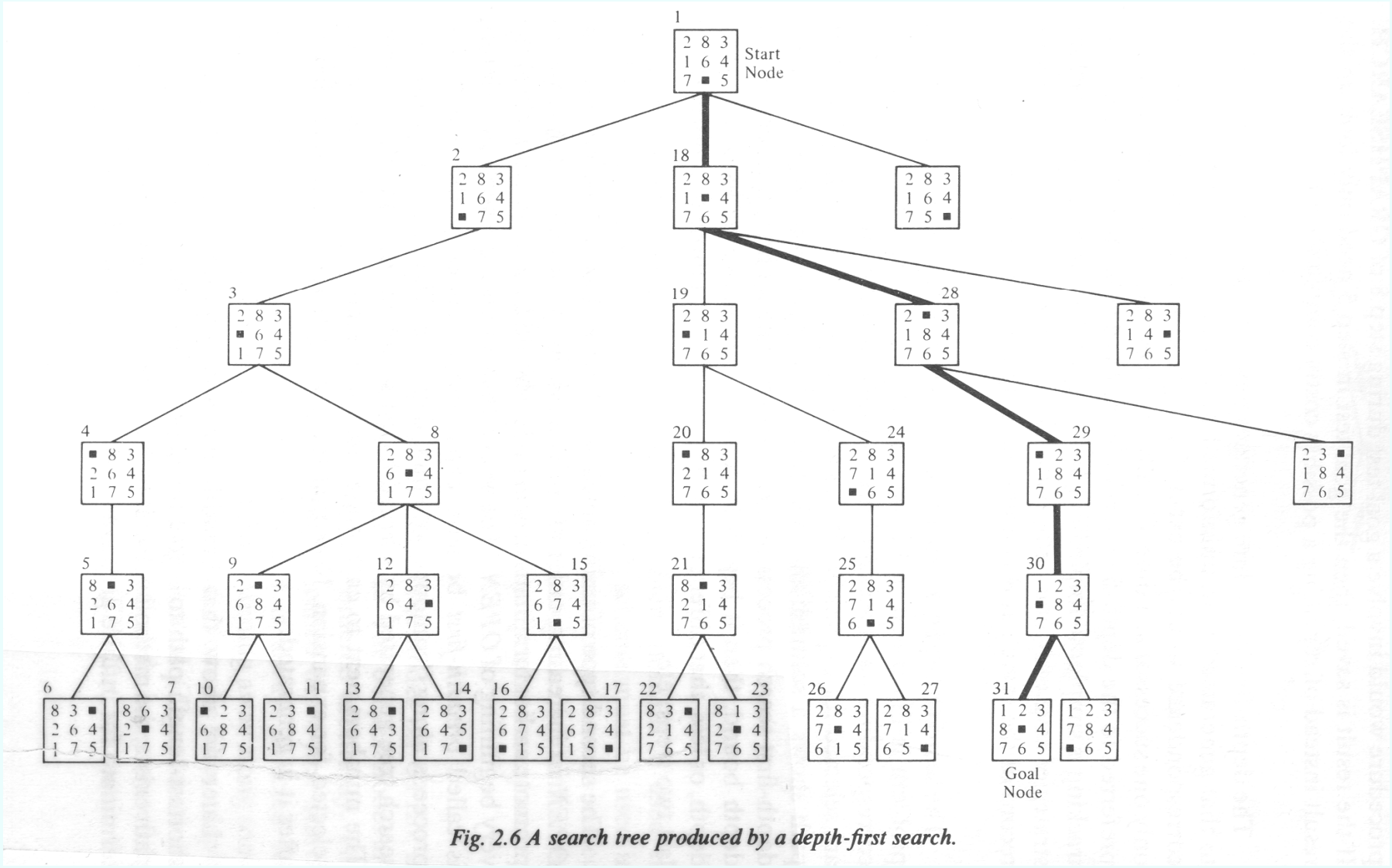


Fig. 2.6 A search tree produced by a depth-first search.

SLEPÉ STRATEGIE HLEDÁNÍ ŘEŠENÍ ÚLOHY

b) do šířky (breadth first search)

Speciální případ základního algoritmu hledání v grafu, v němž platí:

- ohodnocení uzlu = hloubka uzlu,
- v kroku 3 ((***)) se bere minimum.

Poznámka: Při prohledávání stromu řešení algoritmem prohledávání do šířky je vždy nalezena nejkratší cesta k cílovému uzlu, pokud tato cesta existuje.

Př.: „8“

2. Řešení úloh

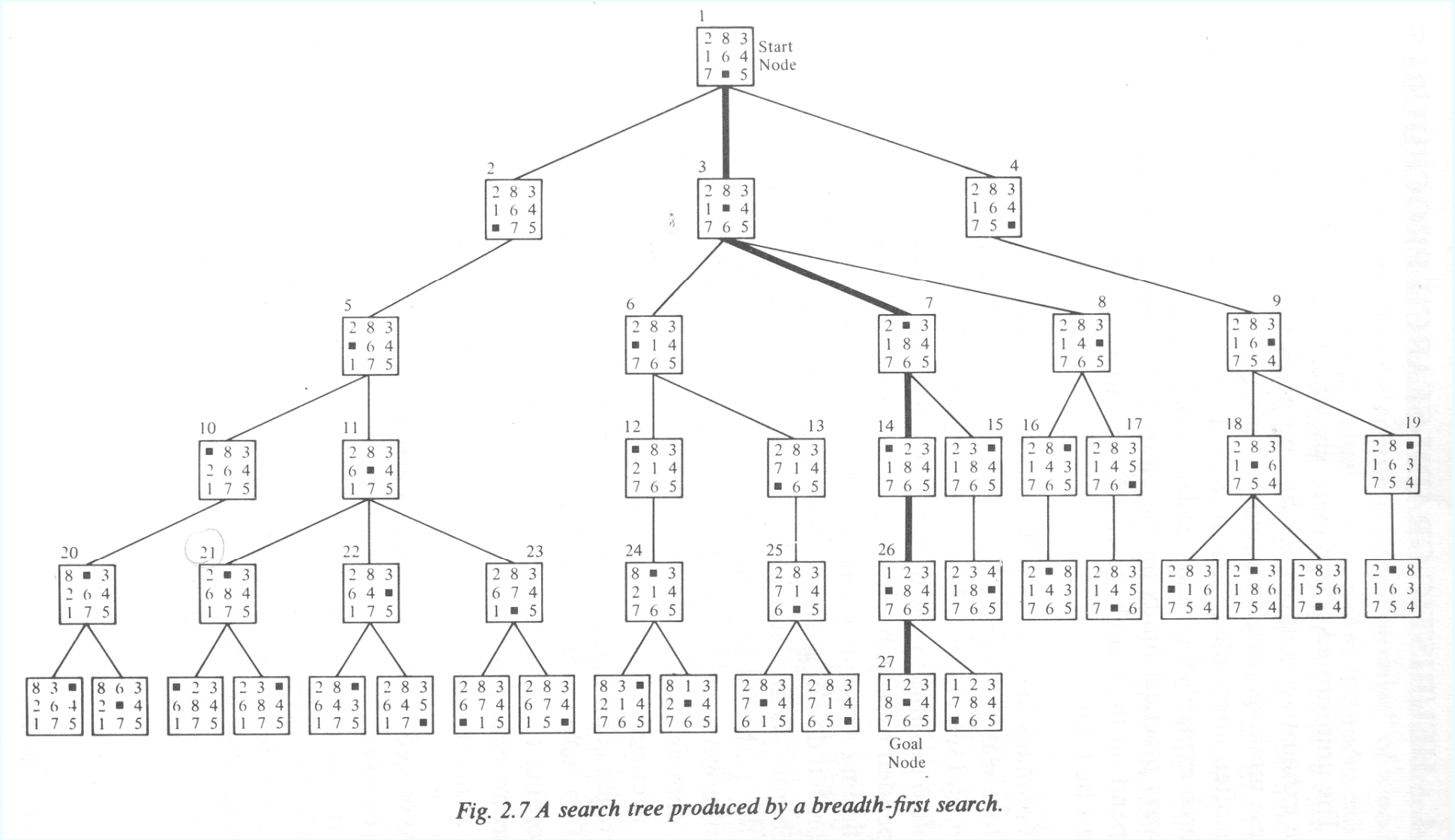


Fig. 2.7 A search tree produced by a breadth-first search.

INFORMOVANÉ (CÍLENÉ) STRATEGIE HLEDÁNÍ ŘEŠENÍ
(HEURISTICKÉ PROHLEDÁVÁNÍ)

Základní myšlenka: Nalézt takovou heuristiku, která pomůže najít řešení při relativně nízkém počtu vygenerovaných uzlů.
Ilustrace obrázkem:

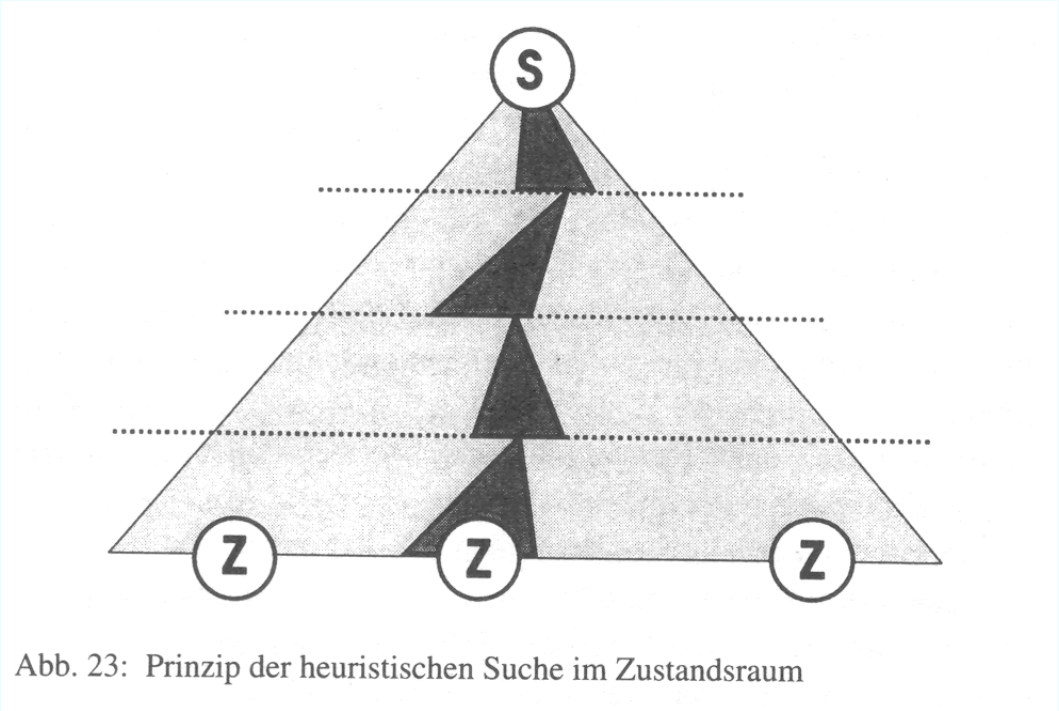


Abb. 23: Prinzip der heuristischen Suche im Zustandsraum

Princip: Hledáme algoritmus ohodnocující funkce pro nalezení optimální (např. nejkratší, „nejlevnější“…) cesty mezi daným a cílovým uzlem grafu řešení úlohy a její extrém.

Zpravidla narazíme na dva protichůdné požadavky:

- chceme nalézt optimální cestu (např. s minimálními náklady),
- požadujeme minimální náklady na prohledávání stromu řešení.

Př.: Heuristické prohledávání pro „8“:

Zvolíme ohodnocující funkci ve tvaru

$$f(n_i) = d(n_i) + w(n_i) ,$$

kde $d(n_i)$ buď délka cesty ve stromu řešení od výchozího uzlu k uzlu n_i,

$w(n_i)$ počet kamenů v „databázi“ uzlu n_i, které neleží na svých místech.

Pozn.: Položíme – li $f(n_i) = d(n_i)$, dostáváme algoritmus prohledávání do šířky.

2. Řešení úloh

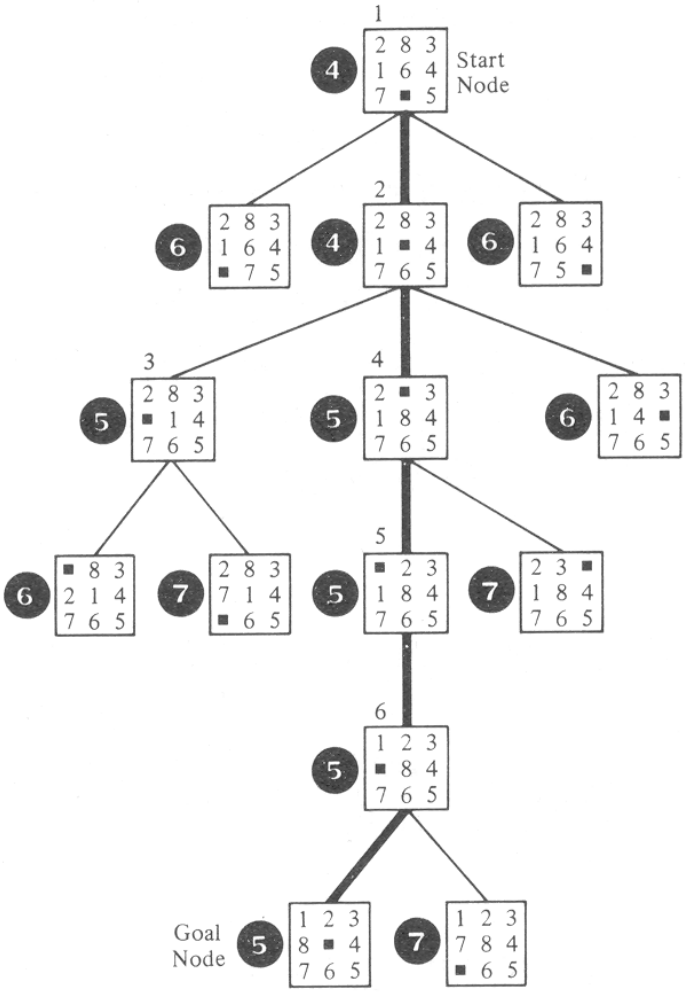


Fig. 2.8 A search tree using an evaluation function.

Algoritmus prohledávání A, A*

Modifikace základního algoritmu hledání v grafu

Za optimální cestu v grafu budeme považovat cestu s minimální „cenou“ – definujeme proto ohodnocující funkci uzlu $f(n_i)$, která poskytne skutečnou (minimální) „cenu“ cesty z výchozího uzlu do cílového uzlu vedoucí přes uzel n_i:

$$f^*(n_i) = g^*(n_i) + h^*(n_i) ,$$

kde $g^*(n_i)$ je skutečná (minimální) „cena“ cesty z výchozího uzlu n_0 do uzlu n_i,

$h^*(n_i)$ je skutečná (minimální) „cena“ cesty z uzlu n_i do cílového uzlu n_g.

Zřejmě $f^*(n_0) = h^*(n_0)$ a $f^*(n_g) = g^*(n_g)$.

Optimální cestu z n_0 do n_g apriorně neznáme, tedy nelze exaktně určit ani hodnotu $f^*(n_i)$ pro kterýkoli uzel n_i. Použijeme proto odhad „ceny“ optimální cesty ve tvaru

$$f'(n_i) = g'(n_i) + h'(n_i) ,$$

kde $g'(n_i)$ je odhad „ceny“ optimální cesty z výchozího uzlu n_0 do uzlu n_i (odhad hodnoty funkce $g^*(n_i)$),

$h'(n_i)$ je odhad „ceny“ optimální cesty z uzlu n_i do cílového uzlu n_g (odhad hodnoty funkce $h^*(n_i)$).

$g'(n_i)$ je zřejmě odhad „ceny“ cesty do uzlu n_i, což vyčíslíme jako součet ohodnocení všech hran na cestě z n_0 do n_i a zjevně platí

$$g'(n_i) \geq g^*(n_i) .$$

Stanovení $h'(n_i)$ je obtížné, nazýváme ji ryze heuristickou funkcí.

Definice: Základní algoritmus hledání v grafu s ohodnocující funkcí vyčíslenou podle vzorce

$$f'(n_i) = g'(n_i) + h'(n_i)$$

nazýváme algoritmem **A** ; je-li $h'(n_i)$ nezáporným (v optimálním případě „nejtěsnějším“) dolním odhadem funkce $h^*(n_i)$, hovoříme o „optimálním“ algoritmu **A*** (platí pro $0 \leq h'(n_i) \leq h^*(n_i)$).

Na ryze heuristické funkci $h'(n_i)$ zpravidla vyžadujeme její monotónnost.

Příklad: „8“ s ryze heuristickou funkcí definovanou jako

$$h'(n_i) = P(n_i) + 3 * S(n_i) , \qquad \text{kde}$$

$P(n_i)$ je součet vzdáleností každého kamene od svého cílového místa v možných posuvech, $S(n_i)$ je míra porušení pořadí kamenů.

2. Řešení úloh

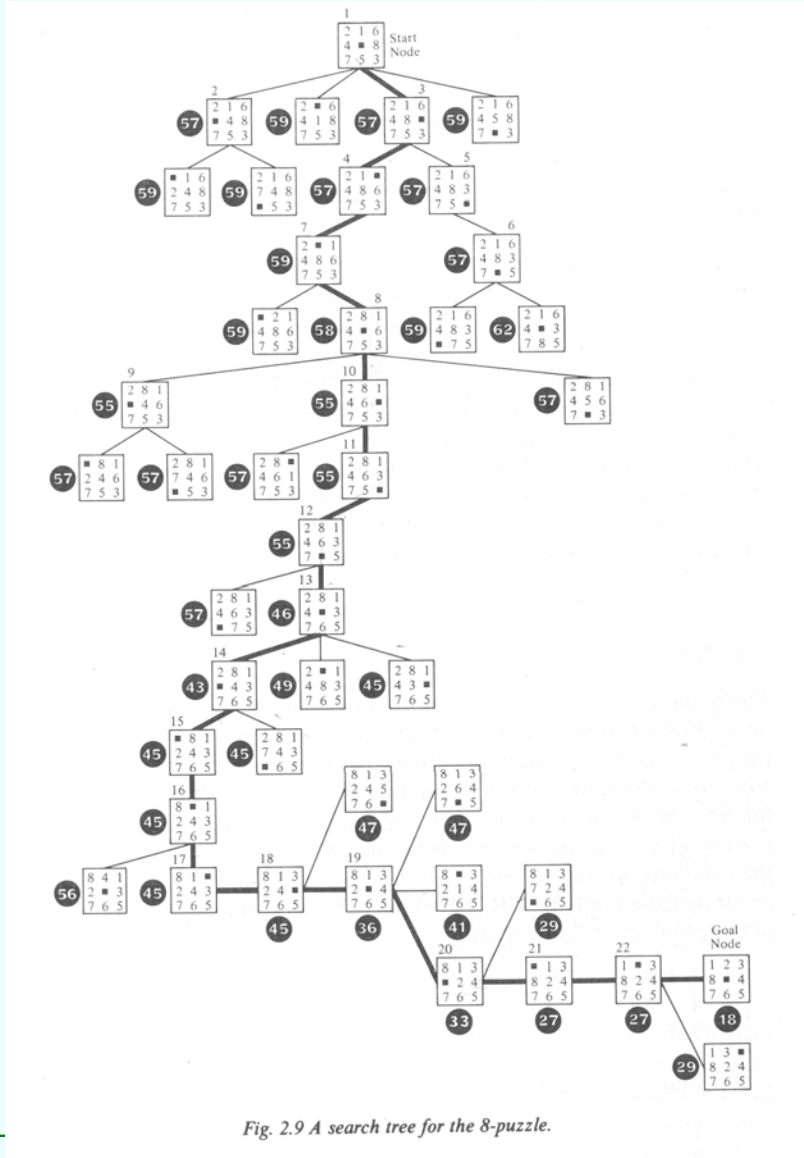


Fig. 2.9 A search tree for the 8-puzzle.

Efektivnost algoritmu hledání řešení úlohy

(algoritmu prohledávání grafu řešení úlohy)

Definice: Více informovaným algoritmem rozumíme algoritmus s ryze heuristickou funkcí $\boldsymbol{h'(n_i)}$, která je „těsnějším“ dolním odhadem funkce $h^*(n_i)$ – více informovaný algoritmus vyžaduje přesnější heuristickou informaci.

Efektivnost algoritmu prohledávání je dána:

- „cenou“ cesty z výchozího uzlu n_0 do cílového uzlu n_g ,
- počtem vygenerovaných uzlů v grafu řešení úlohy,
- algoritmickou složitostí výpočtu heuristické funkce.

Hraní her pro dva a více hráčů

Počítač je při hraní jakékoli hry:

- silný v komplikovaných situacích s množstvím kombinací,
- má obrovskou znalost zahájení a koncovek,
- nevnímá hluk, stress a psychický tlak,
- tvrdý hráč proti soupeři bez fantazie, je však "překvapen" originálními tahy,
- má malou nebo žádnou znalost strategie, díky efektu horizontu neodhalí plán na mnoho kol dopředu,
- je obvykle zaměřen na materiální výhodu – neobětuje figuru či pozici, pokud se to brzy nevyplatí.

Algoritmy hraní her –

– algoritmus minimaxu a alfa-beta prořezávání

Hraní her je zcela přirozeně spojováno s představou prohledávání stavového prostoru, tedy jakési zkoušení možných tahů. Téměř všichni lidé právě tímto způsobem hrají – přemýšlejí, co by mohl soupeř udělat, pokud oni táhnou takhle.

Základní rozdělení algoritmů:

1) algoritmy, které prohledávají celý stavový prostor do určité hloubky,
2) algoritmy, prohledávající jen "dobře vypadající" části stavového prostoru,
3) cílově orientované algoritmy

Nejčastěji se používají programy prvního typu, ale je i hodně programů třetího typu (ty jsou ale většinou dobré jen na jednu konkrétní věc - v případě šachu např. některé koncovky).

Klíčové otázky hraní her:

1. Je nalezené řešení dobré ?
   - libovolné řešení (lze použít heuristiku)
   - nejlepší (prohledat celý prostor)
2. Mohu "vrátit" tah ?
   - ano (osmička)
   - ne (šachy)
3. Je "jistý" vstup ?
   - ano (šachy)
   - ne (poker)

Klasické deskové logické hry

– Základem je PROBLEM SOLVING – snažíme se dostat z počátečního stavu do cílového (např. mat) za použití pravidel.

– Významný rozdíl je v tom, že hráči mají rozdílné úkoly. Cílem jednoho je maximalizovat ohodnocující funkci, zatímco cílem druhého je minimalizovat jí.

Kromě klasického hledání řešení se používají i jiné techniky:

– knihovna zahájení a koncovek
– rozpoznávání vzorů (šablon)

Algoritmus existence vítěze a minimaxu

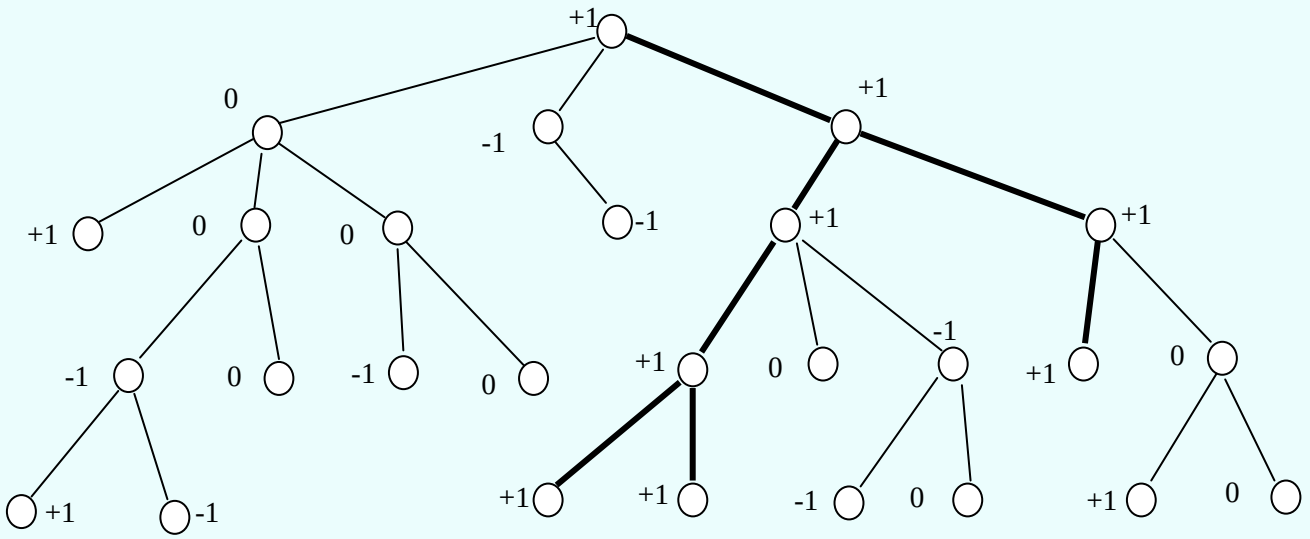
- Algoritmy na prohledávání stavového prostoru se používají pro hry dvou hráčů s úplnou informací, tzn. když má každý hráč všechny informace o hře a jejím současném stavu.
- Tato informace je konečná (nazývá se pozice) a obsahuje též údaj, který hráč je na tahu.
- Dále existují pravidla hry, která určují ke každé pozici konečný počet přípustných tahů, pro hráče, který je právě na tahu.
- Krokem hry (tahem, popř. půltahem) je, když si hráč, který je na tahu, vybere jeden z přípustných tahů a provede jej. Tím vznikne nová pozice a na tahu je soupeř.
- Hra pokračuje, dokud se nedostane do závěrečné pozice, u které musí být též definováno, kdo vyhrál či prohrál, příp. že jde o remízu.
- Na závěr je ještě nutné, aby pravidla vylučovala nekonečnou hru.

Existence vítěze

Hra je znázorněna stromem, v němž

- uzly reprezentují jednotlivé pozice (stavy) hry,
- hrany reprezentují přípustné tahy,
- listy stromu odpovídají koncovým pozicím (stavům) hry.

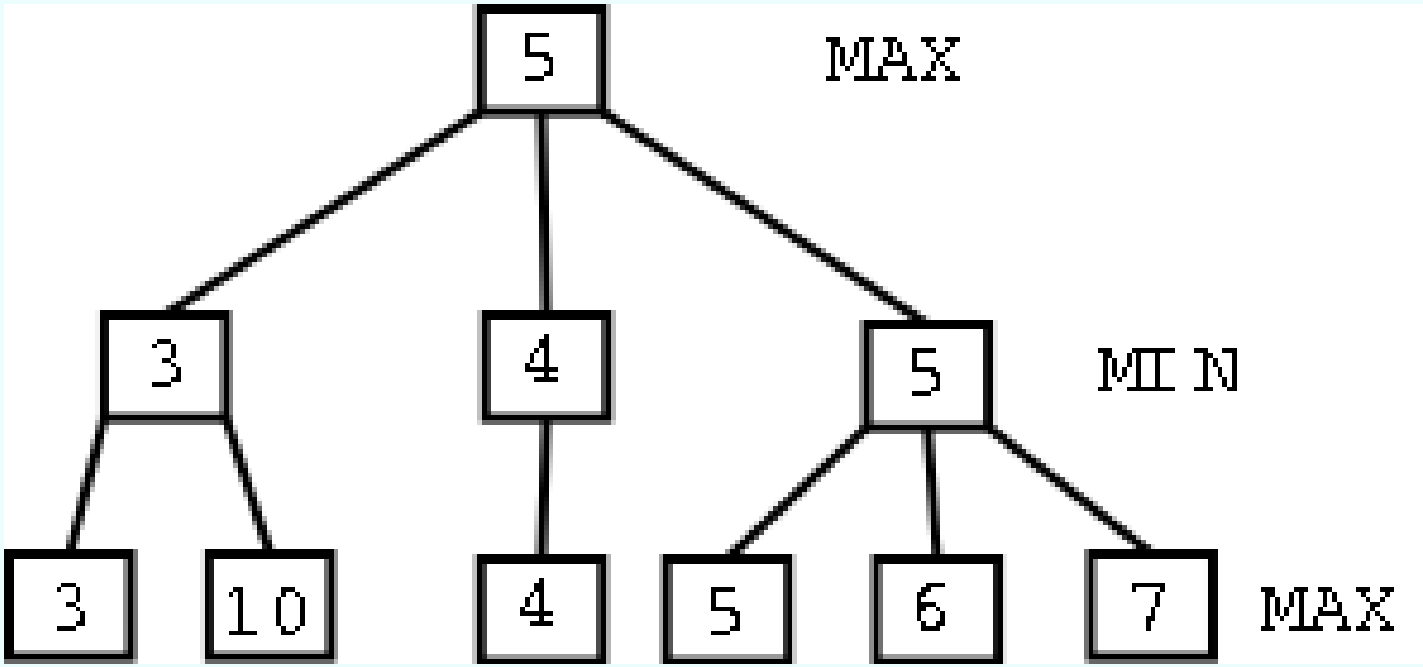
Př.: Pozice ohodnocená
„+1“ je vítězná,
„–1“ je prohraná,
„ 0 “ je remízou.
Zvýrazněné tahy jednoznačně vedou k výhře.



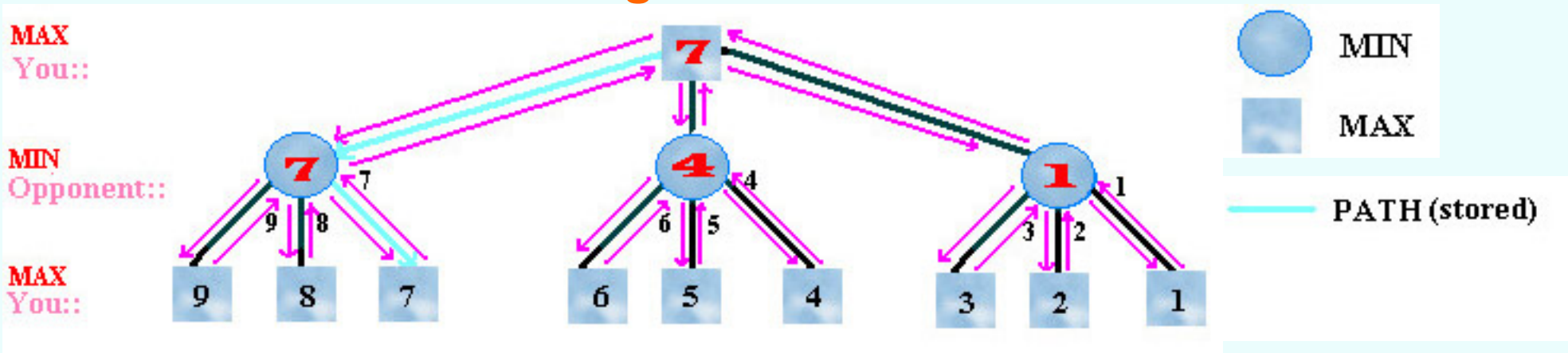
Algoritmus minimaxu

Používá ohodnocení uzlů, které zpravidla vyjadřuje pravděpodobnost, s jakou lze ve hře dosáhnout vítězství.

Proto – na úrovni hráče vybíráme maximum,
– na úrovni protihráče vybíráme minimum.

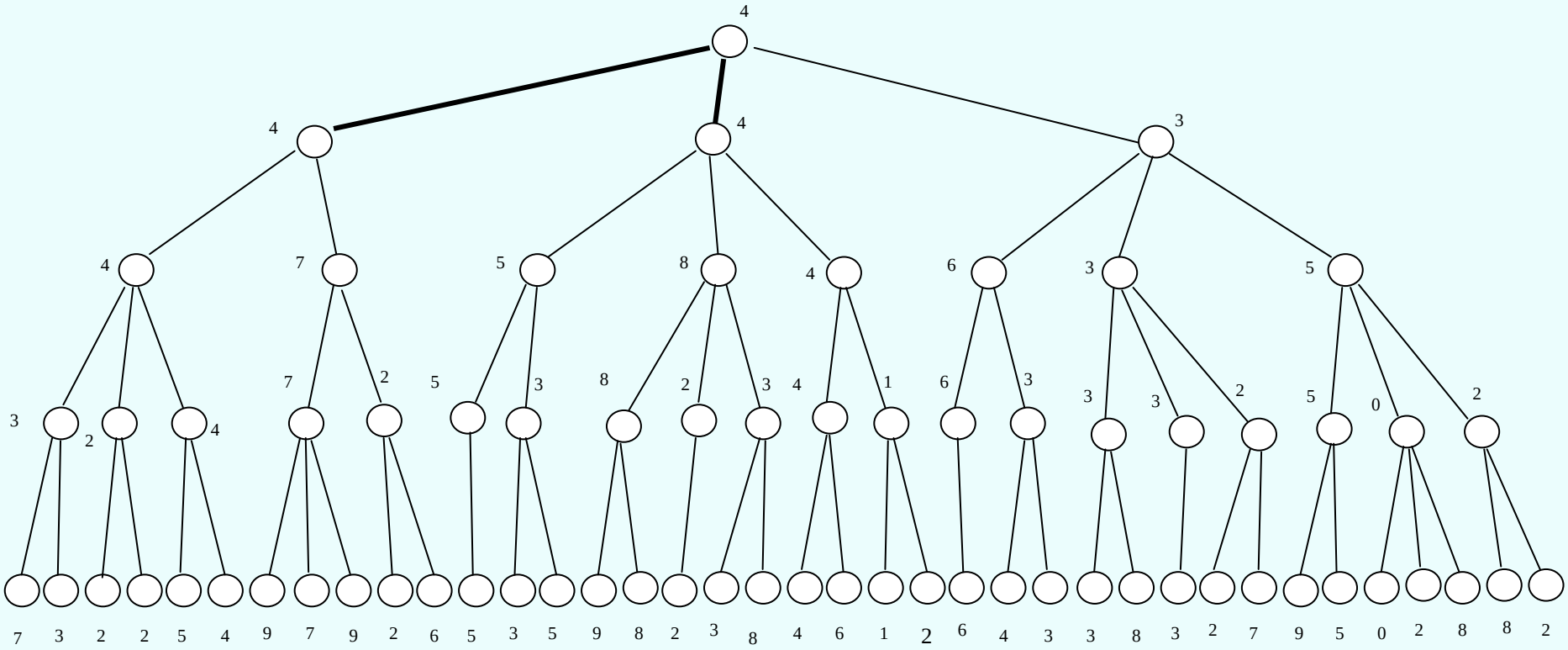


Hledání vítězného tahu algoritmem minimaxu:




```
minmax(u) {              //   u is the node you want to score
  if  u is a leaf  return ( score of u );
  else  if u in a min node
           for all childern of u: v1,..., vn
            return min({minmax(v1),...,minmax(vn)});
        else
           for all childern of u: v1, ..., vn;
            return max({minmax(v1),...,minmax(vn)});
 }
```

Příklad: Ohodnocení uzlů stromu řešení při hledání řešení metodou minimaxu:



Příklad: Hledání řešení hry „piškvorky“ (ve variantě omezené na hrací pole 3 x 3 čtverečky) metodou minimaxu

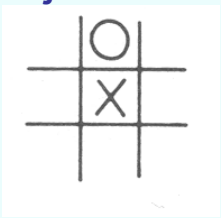
Popis úlohy a algoritmus výpočtu ohodnocující funkce:

Vložení křížku (x) do hracího pole značí tah 1. hráče a bude reprezentováno uzlem stromu, v němž budeme volit maximum (MAX).

Vložení kroužku (o) do hracího pole značí tah protihráče a bude reprezentováno uzlem stromu, v němž budeme volit minimum (MIN).

Výpočet ohodnocující funkce definujeme takto:

- nachází-li se 1. hráč (MAX-uzel) v pozici výherce, pak $f(n_i) = \infty$ (maxint);
- nachází-li se protihráč (MIN-uzel) v pozici výherce, pak $f(n_i) = -\infty$;
- není-li ani jeden z hráčů v pozici výherce, pak $f(n_i)$ vyčíslíme jako počet kompletních řádků, sloupců a diagonál (souvislých trojic křížků) hracího pole, které 1. hráč ještě může vyplnit – (minus) počet kompletních řádků, sloupců a diagonál (souvislých trojic kroužků) hracího pole, které ještě může vyplnit protihráč. Např. pro situaci:



$$f(n_i) = 6 - 4 = 2 \quad .$$

2. Řešení úloh

1. tah 1. hráče + 1. tah protihráče (část stromu řešení):

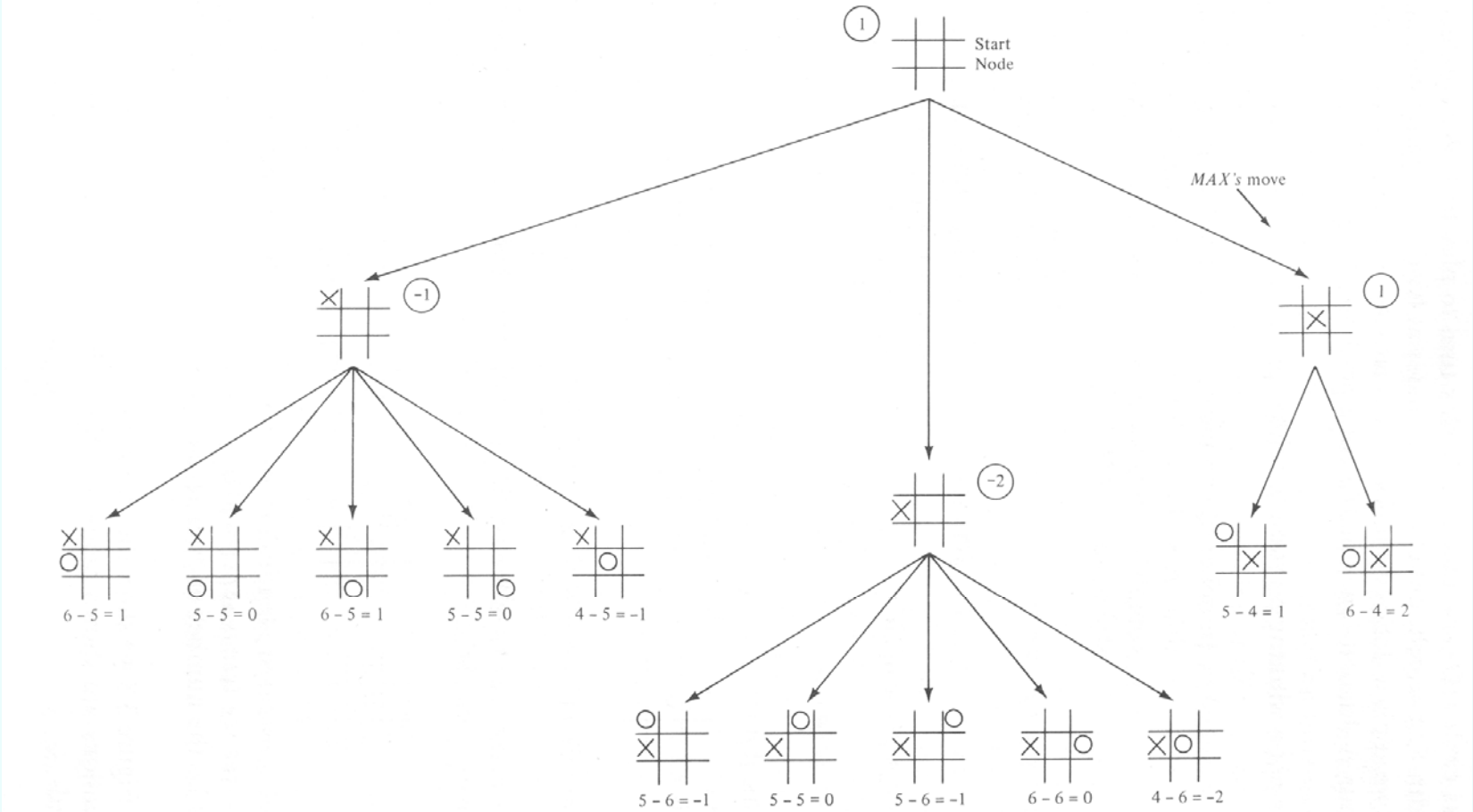


Fig. 3.8 Minimax applied to tic-tac-toe (stage 1).

2. tah obou hráčů (pouze část stromu řešení):

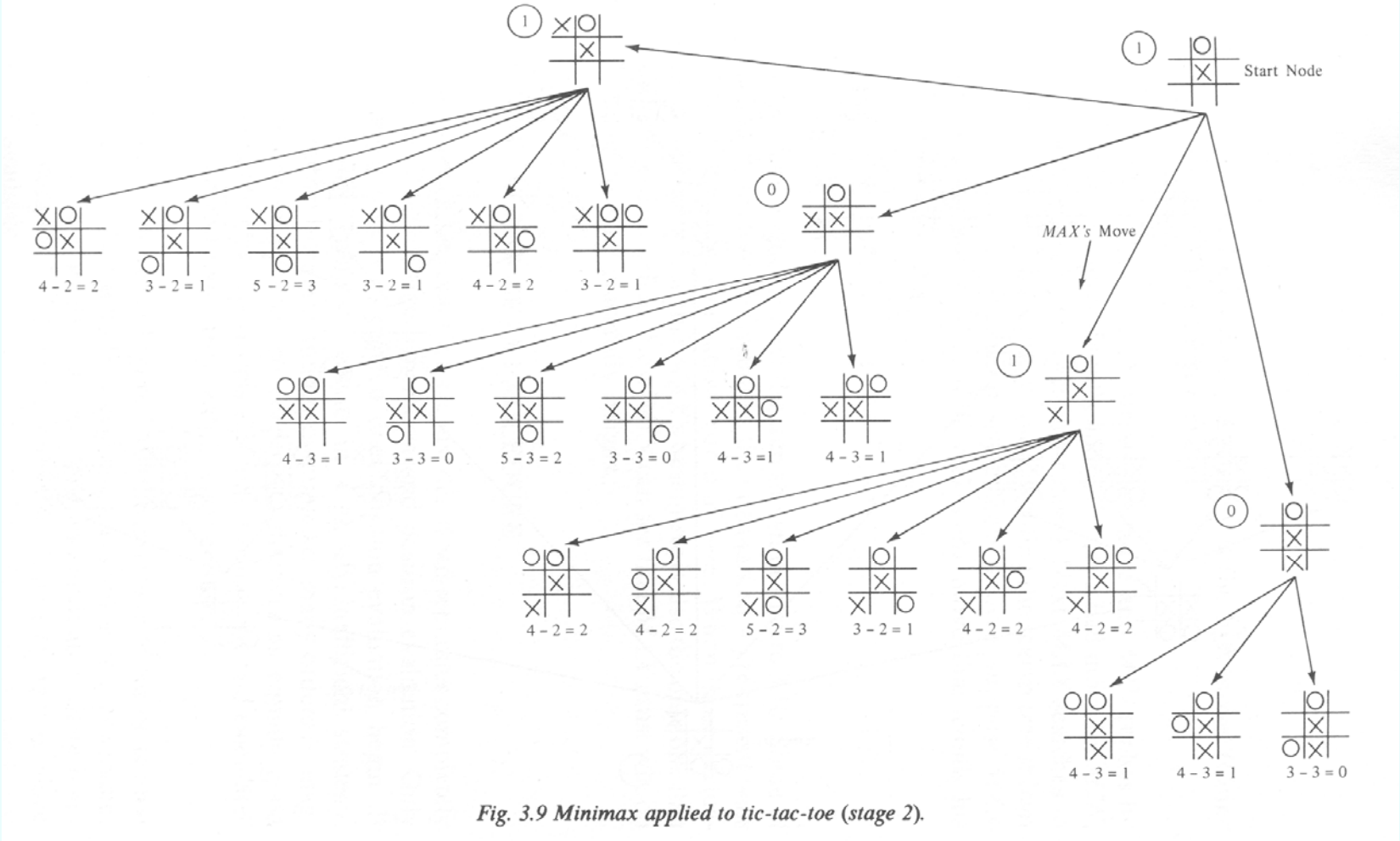


Fig. 3.9 Minimax applied to tic-tac-toe (stage 2).

3. tah obou hráčů vedoucí k vítězství:

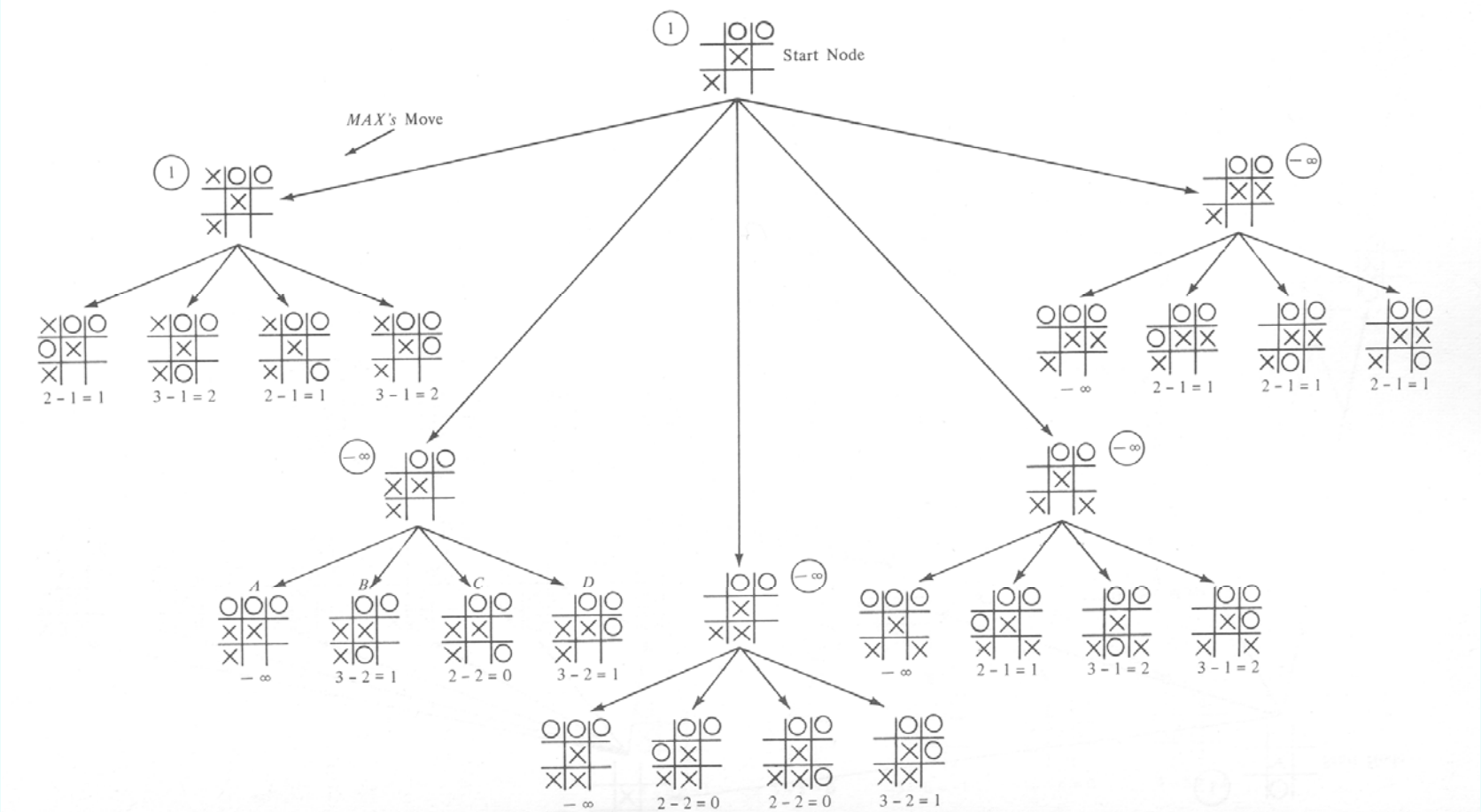
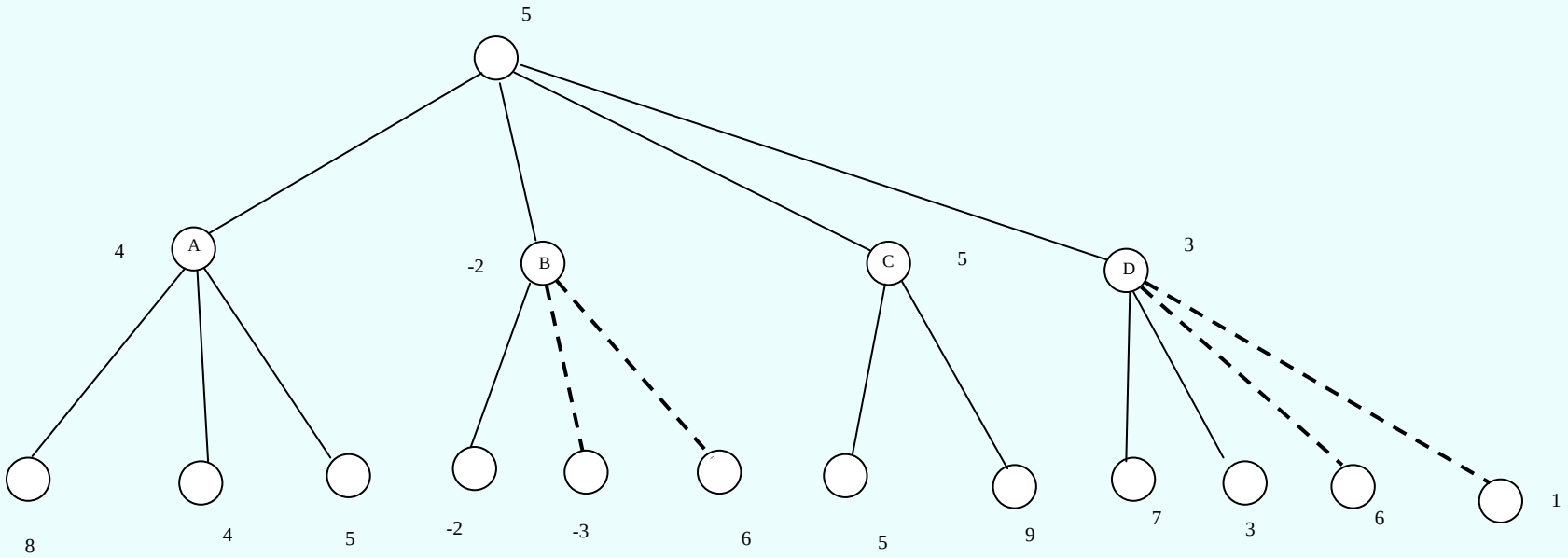


Fig. 3.10 Minimax applied to tic-tac-toe (stage 3).

Algoritmus alfa–beta prořezávání

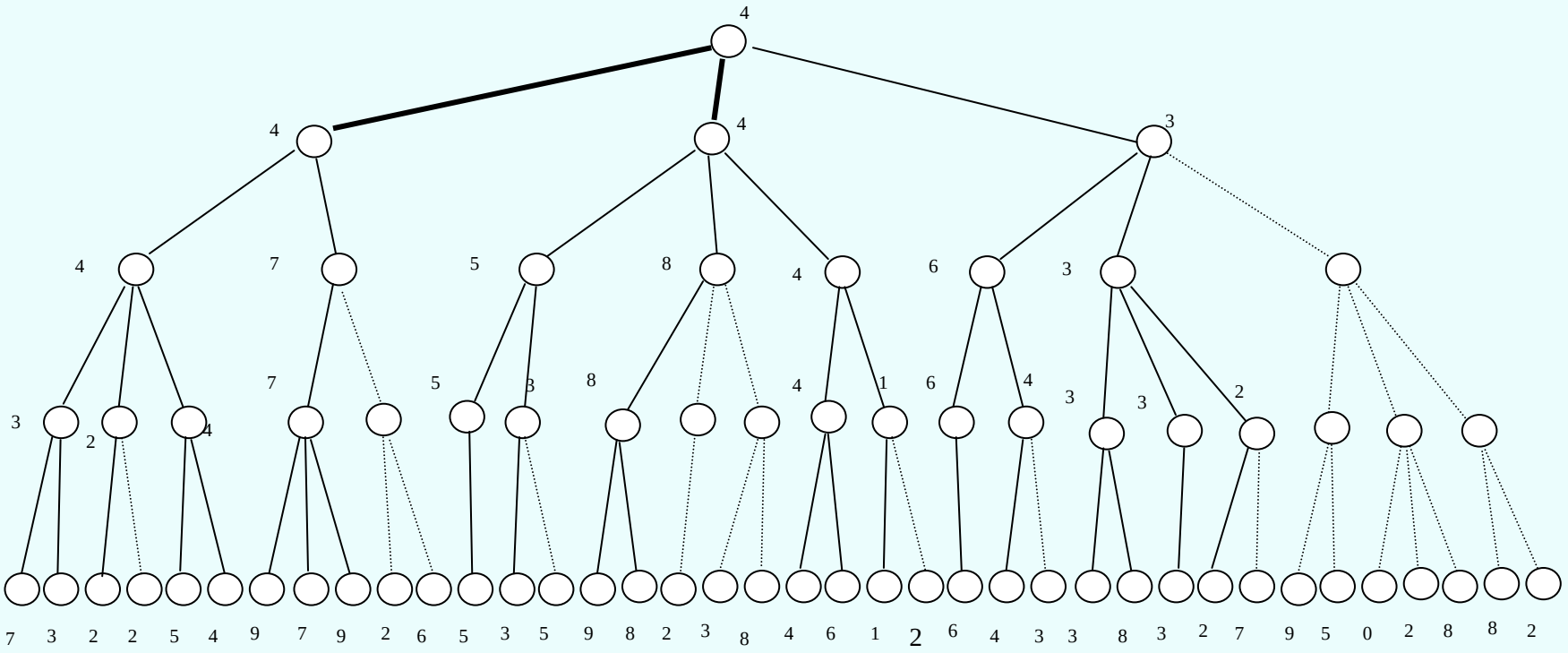
Podstata algoritmu: Když víme, že nějaká větev je špatná (tj. horší než doposud nalezená nejlepší), nepotřebujeme už vědět, jak moc je špatná. Navíc nás ani nezajímá, zdali tam nebude lepší podvětev, protože soupeř by se jí jistě uměl vyhnout.

Př.:

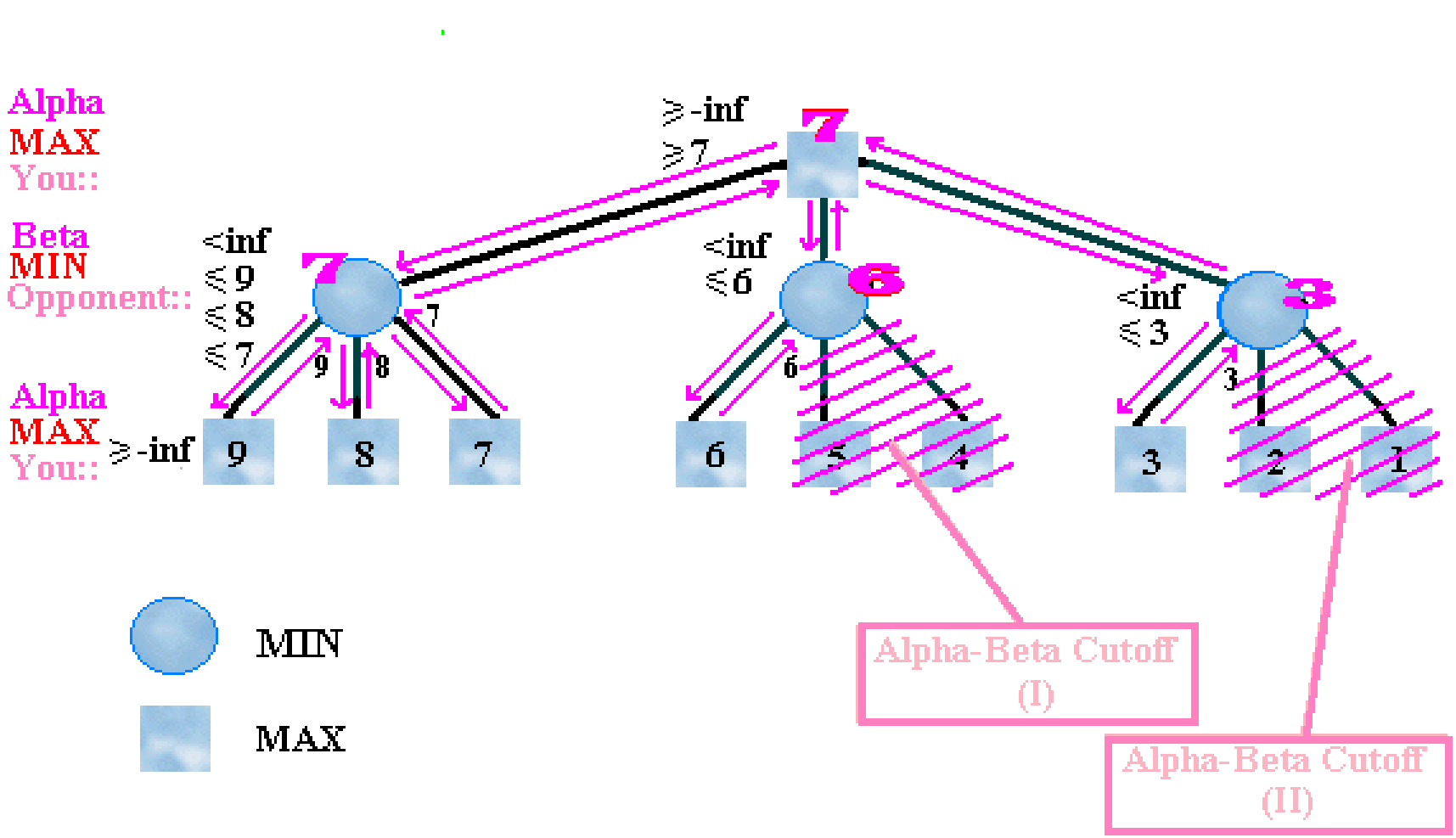


2. Řešení úloh

Postup: Pro množinu uzlů na jedné úrovni si pamatujeme nějaké maximum (nebo minimum, pokud jde o minimalizující úroveň) a pokud zjistíme, že ohodnocení synů dalšího uzlu v řadě je menší (větší) než toto maximum (minimum), nepotřebujeme tento uzel dál rozvíjet, protože víme, že už minimaxovou hodnotu svého otce neovlivní (tečkované větve).



Hledání vítězného tahu algoritmem alfa–beta prořezávání:



Algoritmus alfa–beta prořezávání v pseudokódu:

```
function MINIMAX-AB(N, A, B) is ;;                    //  Here A is always less than B
 begin
   if N is a leaf then
     return the estimated score of this leaf
   else
     Set Alpha value of N to -infinity
           and Beta value of N to +infinity;
     if N is a Min node then
      For each successor Ni of N loop
       Let Val be MINIMAX-AB(Ni,A,Min{B,Beta of N});
       Set Beta value of N to Min{Beta value of N,Val};
     When A >= Beta value of N then
     Return Beta value of N
       endloop;
       Return Beta value of N;
   else
    For each successor Ni of N loop
     Let Val be MINIMAX-AB(Ni,Max{A,Alpha value of N},B);
     Set Alpha value of N to Max{Alpha value of N, Val};
     When Alpha value of N >= B then
     Return Alpha value of N
       endloop;
    Return Alpha value of N;
 end MINIMAX-AB;
```

Jiná varianta implementace algoritmu:

```
minimax(in game board, in best_opposing_score, out score chosen_score,
                                                    out move chosen_move)
begin
        best_score = -infinity;
        Gt_generate_moves(current_mimx_data,moves_list);
        for (i = 0 to moves_list.num_moves-1) do
            new_board = board;
            Gt_move(board, moves_list[i],the_unmove);
            minimax(board, the_score,the_move);
            Gt_unmove(board,the_unmove);
            if (the_score > best_score) then
               best_score = the_score;
               best_move = moves_list[i];
               if (best_score > best_opposing_score) then
              cut;/* the opponent will not allow you to reach this node
                          in real life so there is no since in continuing*/
          endif
            endif
         enddo
         chosen_move = best_move;
      Gt_evaluate(current_mimx_data,chosen_score,best_score);
end.
```

Př.: **Strom řešení „piškvorek“ „prořezaný“ algoritmem alfa–beta prořezávání**

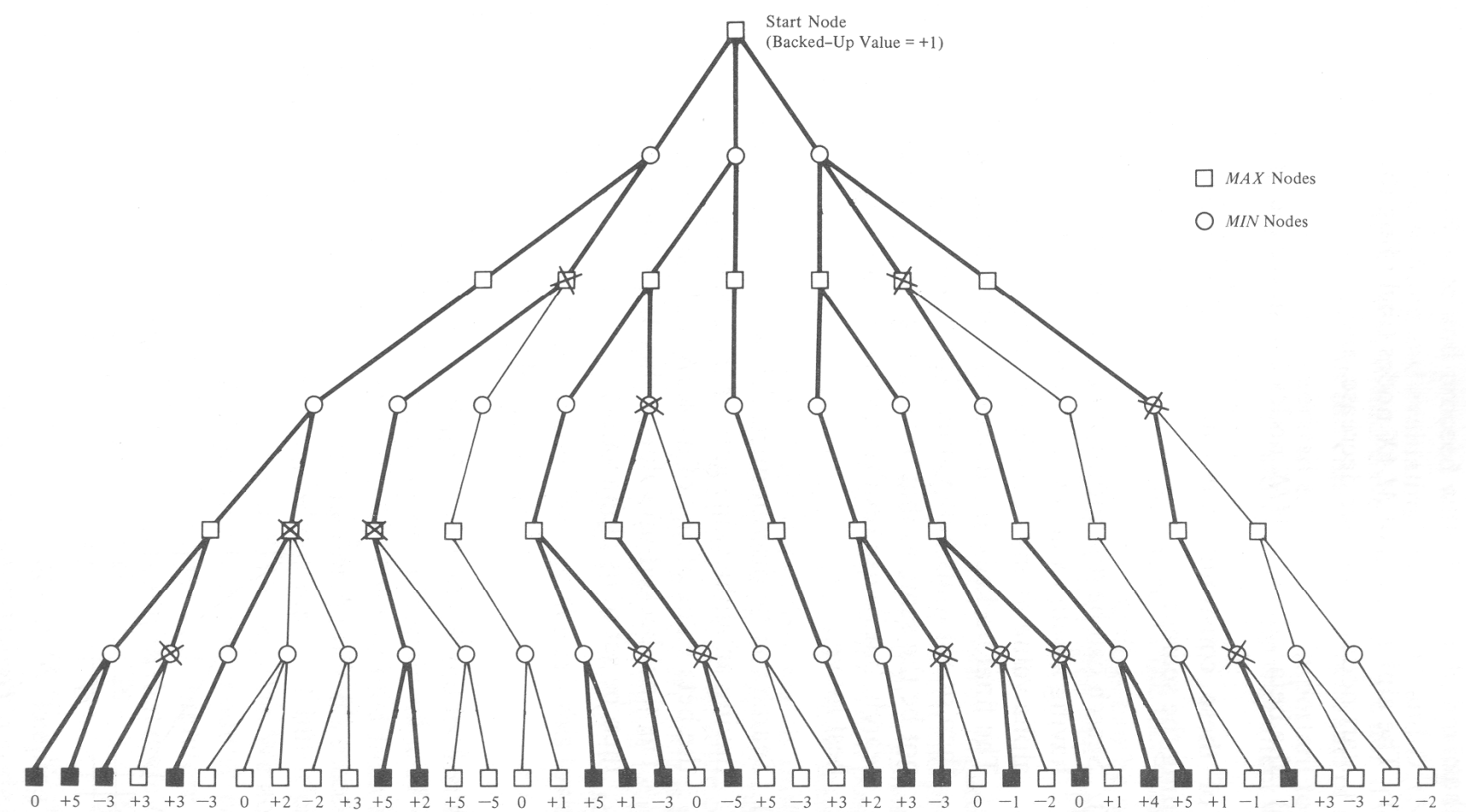
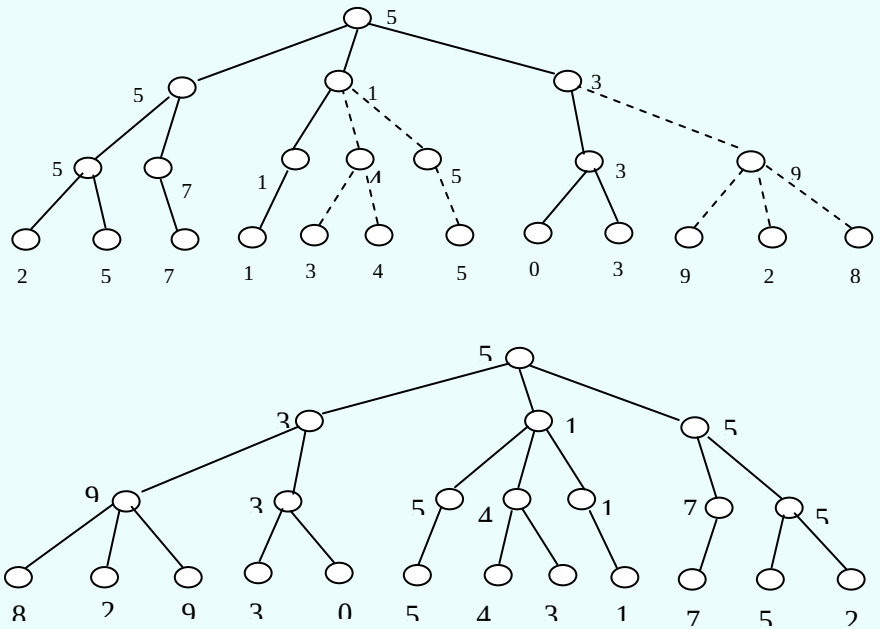


Fig. 3.12 An example illustrating the alpha-beta search procedure.

Efektivnost algoritmu alfa–beta prořezávání

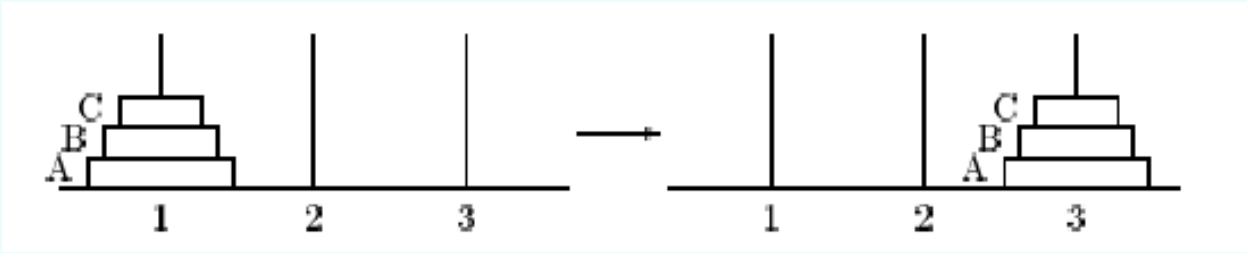
Z porovnání algoritmů minimaxu a alfa–beta prořezávání vyplývá vlastnost, že algoritmus alfa–beta prořezávání vrací hodnotu, kterou by vrátil původní algoritmus, aniž by musel projít celým stromem, což je vlastně to, co jsme potřebovali získat. Z předchozích obrázků je patrna jistá efektivita, ale je toto efektivní vždy? Může se stát, že je ořezávání lepší nebo naopak horší? Na následujícím obrázku jsou ukázány dva příklady:



Oba stromy jsou ohodnoceny algoritmem alfa–beta ořezávání. Alfa–beta metoda se neuplatní, protože strom nemá dost zde pater. Aby se uplatnila, jsou potřeba alespoň čtyři úrovně (viz výše). Na prvním obrázku byl strom algoritmem ořezán poměrně hodně, na druhém vůbec. Přitom jediná odlišnost, která je mezi oběma obrázky, je pořadí, v jakém jsou v obou stromech znázorněny tahy. Důkaz, proč tomu tak je, si proveďte jako cvičení.

Rozklad úlohy na podúlohy

Mějme dánu úlohu přemísťování hanojských věží podle obrázku. V počátečním stavu úlohy je hanojská věž skládající se z N kotoučů o různých průměrech situována na levém kolíku, který označme 1. Úkolem úlohy je přemístit jednotlivé kotouče na pravý kolík (označený 3) pomocí středního kolíku 2 tak, že se smí vždy přemísťovat jen vrchní kotouč a žádný kotouč nesmí nikdy ležet na kotouči menšího průměru. Kotouče označme podle velikosti (průměru) od největšího po nejmenší písmeny A, B, C, …, N (pod symbolem N si představme nejvyšší písmeno označující nejmenčí kotouč, např. pro věž o třech kotoučích N = C) :



Libovolný stav úlohy reprezentujeme seznamem ["1", "2", "3"] , kde symboly "1", "2", "3" představují seznamovou reprezentaci uložení kotoučů na kolíku "1", "2" či "3" v pořadí zdola nahoru. Nenachází-li se na kolíku žádný kotouč, bude seznam prázdný (reprezentován *nil*).

2. Řešení úloh

Vyobrazená úloha se pak dá symbolicky zapsat jako

$$s_0 \equiv [\,[\,A\,,B\,,C\,]\,,\mathit{nil}\,,\mathit{nil}\,] \rightarrow [\,\mathit{nil}\,,\mathit{nil}\,,[\,A\,,B\,,C\,]\,] \equiv s_g\,.$$

Úlohu symbolicky rozložíme na tři dílčí úlohy:

1. $[\,[\,A\,,B\,,C\,]\,,\mathit{nil}\,,\mathit{nil}\,] \rightarrow [\,X\,,Y\,,[\,A\,]\,]$
2. $[\,X\,,Y\,,[\,A\,]\,] \rightarrow [\,X'\,,Y'\,,[\,A\,,B\,]\,]$
3. $[\,X'\,,Y'\,,[\,A\,,B\,]\,] \rightarrow [\,\mathit{nil}\,,\mathit{nil}\,,[\,A\,,B\,,C\,]\,]$

Jiný způsob rozkladu úlohy na dílčí úlohy můžeme zapsat např.:

1. $[\,[\,A\,,B\,,C\,]\,,\mathit{nil}\,,\mathit{nil}\,] \rightarrow [\,[\,A\,]\,,[\,B\,,C\,]\,,\mathit{nil}\,]$
2. $[\,[\,A\,]\,,[\,B\,,C\,]\,,\mathit{nil}\,] \rightarrow [\,\mathit{nil}\,,[\,B\,,C\,]\,,[\,A\,]\,]$
3. $[\,\mathit{nil}\,,[\,B\,,C\,]\,,[\,A\,]\,] \rightarrow [\,\mathit{nil}\,,\mathit{nil}\,,[\,A\,,B\,,C\,]\,]$

Konjunktivně − disjunktivní (AND / OR) graf

Účel: Znázorňuje symbolicky rozklad úlohy na podúlohy.

Definice: Konjunktivně – disjunktivní graf (AND/OR graf nebo také transformační graf) je orientovaný acyklický graf s uzly, které nejsou listy, dvojího typu:

- AND-uzel představuje konjunkci podúloh, tj. k vyřešení (pod)úlohy reprezentované AND-uzlem je zapotřebí, aby každá z podúloh byla řešitelná (a vyřešena). Hrany vycházející z AND-uzlu jsou pro rozlišení typu uzlu spojeny obloučkem.
- OR-uzel představuje disjunkci podúloh, tj. k vyřešení (pod)úlohy reprezentované OR-uzlem stačí, aby alespoň jedna z podúloh byla řešitelná (a vyřešena).

2. Řešení úloh

Příklad: Reprezentujte následující soustavu logických formulí (v nichž A, B, …, Z označují výroky) konjunktivně – disjunktivním grafem:

(1) $A \wedge B \rightarrow M$

(2) $C \rightarrow M$

(3) $D \wedge B \rightarrow N$

(4) $E \wedge B \rightarrow P$

(5) $C \wedge G \rightarrow R$

(6) $H \rightarrow R$

(7) $E \wedge F \wedge G \rightarrow Q$

(8) $M \rightarrow X$

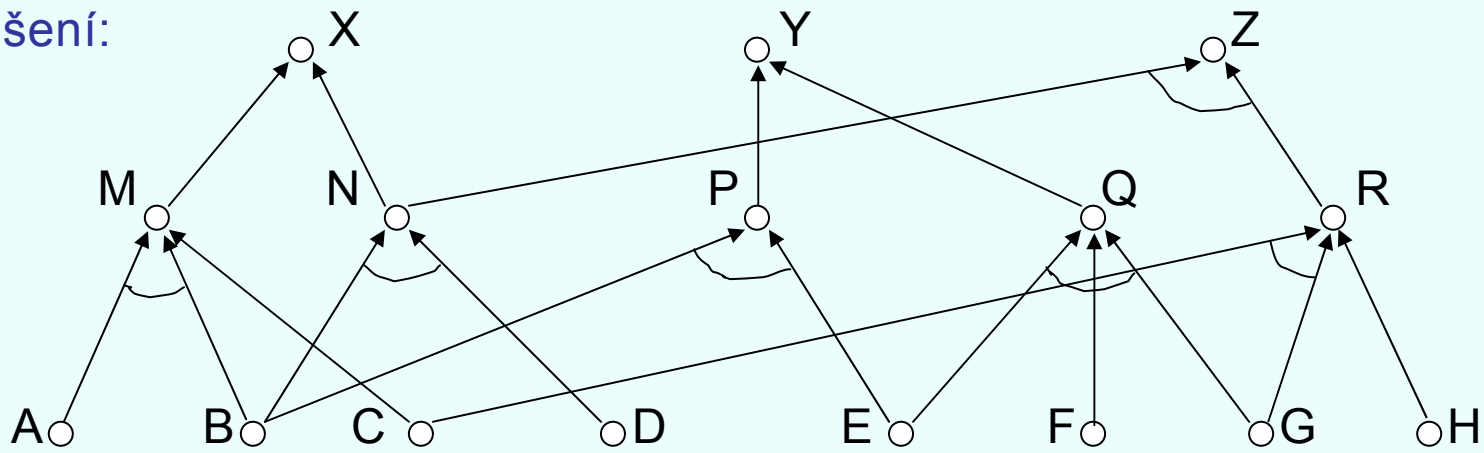
(9) $N \rightarrow X$

(10) $P \rightarrow Y$

(11) $Q \rightarrow Y$

(12) $N \wedge R \rightarrow Z$

Řešení:



Algoritmus rozkladu úloh na podúlohy

Předpokládejme libovolný stav s_i úlohy $\boldsymbol{G}$ popsaný

$$s_i = stav\,(s_i^{\,1},\ s_i^{\,2},\ \ldots,\ s_i^{\,M})\ ,$$

kde „stav“ je funktor reprezentující strukturu stavu s_0 , která má M složek (argumentů). Úlohu $\boldsymbol{G}$ můžeme zjednodušeně zapsat

$$\boldsymbol{G}\colon\quad s_0 \rightarrow s_g\ ,$$

kde s_0 je stav výchozí a s_g stav koncový (cílový). Pak lze úlohu přepsat do tvaru

$$\boldsymbol{G}\colon\ stav\,(s_0^{\,1},\ s_0^{\,2},\ \ldots,\ s_0^{\,M}) \rightarrow stav\,(s_g^{\,1},\ s_g^{\,2},\ \ldots,\ s_g^{\,M})\ .$$

Poznámka: Pro $s_0^{\,m} = s_g^{\,m}$ (pro každé m) platí, že úloha je vyřešena. To, co činí úlohu úlohou určenou k řešení, je nesouhlas odpovídajících si složek struktury stavu (argumentů funkce $stav$).

Nesouhlas mezi složkami s_0^m a s_g^m (pro každé m) se nazývá m-tá diference stavu. Pro některé druhy výrazů (příp. funktorů) lze diferenci vyjádřit číselně, jindy strukturně nebo logicky (musíme se spokojit s konstatováním, že „diference je či není “).

Úloha bude vyřešena, nalezneme-li takový kompoziční operátor R_{Kog}, který odstraní všechny diference, tj. postupně odstraní první a druhou a třetí a … a M-tou diferenci. Tím dojde k rozkladu úlohy $\boldsymbol{G}$ na M podúloh:

$$\boldsymbol{G_1}: \; stav\,(s_0^1, s_0^2, \ldots, s_0^M) \rightarrow stav\,(s_g^1, s_0^2, \ldots, s_0^M)\,,$$

$$\boldsymbol{G_2}: \; stav\,(s_g^1, s_0^2, \ldots, s_0^M) \rightarrow stav\,(s_g^1, s_g^2, \ldots, s_0^M)\,,$$

$$\ldots$$

$$\boldsymbol{G_M}: \; stav\,(s_g^1, s_g^2, \ldots, s_0^M) \rightarrow stav\,(s_g^1, s_g^2, \ldots, s_g^M)\,.$$

Pozor: Pravidla pro odstraňování diferencí mívají vedlejší efekty !

Př.: Rozklad řešení integrálu:

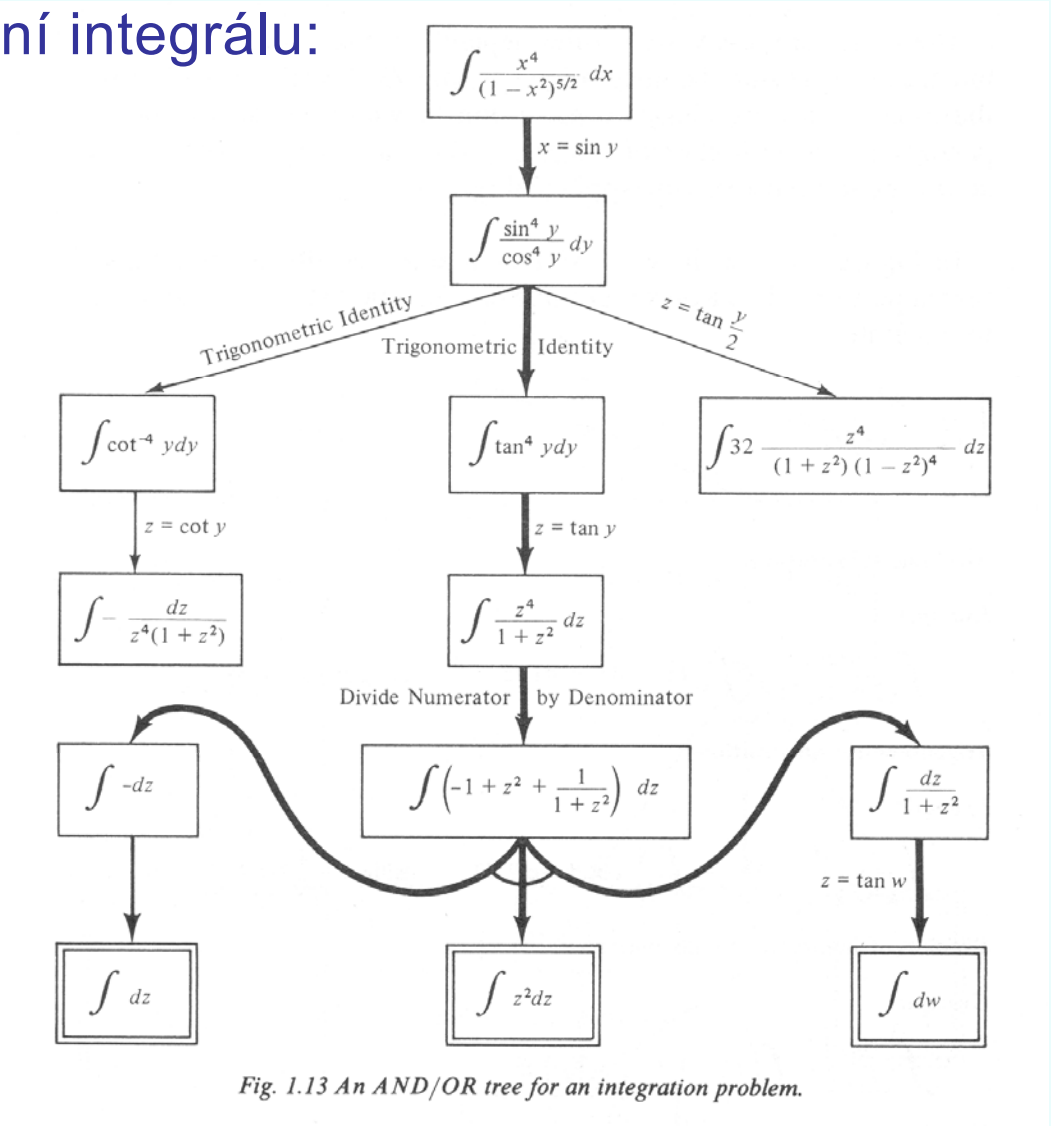


Fig. 1.13 An AND/OR tree for an integration problem.