Masarykova univerzita

Fakulta informatiky

PB016 Umělá inteligence

Sbírka příkladů

Semestr Podzim 2024

Matěj Pavlík, Terézia Mikulová, Ondřej Huvar, Roman Solař

Chyby hlašte na: 469088@mail.muni.cz

Jak se sbírkou pracovat

Sbírka je rozdělena do dvanácti kapitol, které odpovídají tematických celkům postupně probíraným na přednáškách a cvičení. Každá kapitola je **rozdělena do podkapitol**.

Před cvičením si ve sbírce prolistujte příslušnou kapitolu. Pro úspěšné složení odpovědníků potřebujete zhlédnout přednášku a porozumět všem částem kapitoly, které jsou označeny svislou čarou po levém boku (jako tento odstavec). Pokud pro vás budou surové definice těžko stravitelné, může být pro pochopení jednodušší si přečíst i „omáčku“ okolo, která není zvýrazněná, ale umožňuje pojmům pomocí příkladů lépe porozumět.

Na cvičení se budete ještě před implementačními úlohami zhruba 30 minut věnovat příkladům ve sbírce, které jsou označené hvězdičkou ★. V případě cvičení na logiku se budete příkladům ze sbírky věnovat po celou dobu cvičení.

Doporučovaný postup je po krátkém vysvětlení a rozmyšlení problému řešit zvolené příklady demonstračně, případně ve spolupráci se všemi studenty kladením vhodných otázek. Na samostatné řešení příkladů nebude na cvičení prostor. U příkladů s více pododrážkami je rozsah procvičení ponechán na zvážení cvičících.

Po cvičení můžete své znalosti látky ověřit řešením dalších příkladů ze sbírky, případně rozšířit a doplnit znalosti z přednášky pročítáním doprovodného textu. Podobný postup je doporučen i pro samotnou přípravu na zkoušku.

Příklad. Ve sbírce se nacházejí příklady dvou typů – číslované a nečíslované. Nečíslované příklady jsou uvedeny vždy v úvodu k dané podkapitole a slouží k demonstraci či ilustraci probíraného konceptu na konkrétní situaci. Číslované příklady jsou pak uvedeny za vodorovnou dělicí čarou a jsou určeny ke společnému řešení ve cvičení (jsou-li označeny hvězdičkou ★), případně k samostatnému vypracování.

V zelených rámečcích naleznete řešení příkladů. U nečíslovaných příkladů se zobrazují řešení vždy, u číslovaných příkladů hledejte řešení ve verzi sbírky s klíčem.

V modrých rámečcích jsou ve sbírce vyobrazeny definice pojmů.

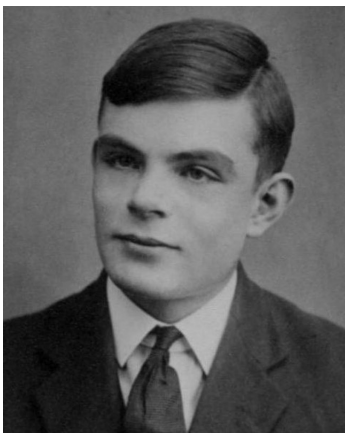
Obsah

1 Úvod do umělé inteligence, Řešení problémů

Úvodní kapitola nabízí čtenáři sbírky první setkání s umělou inteligencí. Představuje nejprve Turingův test k určení schopnosti stroje chovat se inteligentně a příklady k zamyšlení nad hranicemi a možnostmi oboru umělé inteligence jako takového (v této jediné části sbírky jsou příklady ponechány bez řešení). Další část pak prezentuje pojem *problému* a na řadě příkladů ilustruje, jak může volba vhodné datové struktury či strategie k řešení daného úkolu ovlivnit samotnou možnost úkol vyřešit.

1.1 Umělá inteligence a Turingův test

V roce 1950 navrhnul Alan Turing tzv. *Turingův test* (který původně sám nazval „imitační hrou“, viz původní Turingův článek), jež si klade za cíl ověřit schopnost stroje vykazovat inteligentní chování.



Alan Turing ve věku 16 let

Turing v článku nejprve uvádí, že navrhuje uvážit otázku, zda stroje umí myslet. Jelikož však není snadné definovat, co znamená „myslet“, navrhuje nahradit tuto poněkud vágní otázku jinou, a sice zda „Lze sestrojit počítač, který by byl schopen složit Turingův test?“

Definice 1: *Turingův test* je navržen jako hra tří hráčů, z nichž jeden je stroj podrobený zkoušce a ostatní dva jsou lidé. V základní variantě Turingova testu jsou do různých místností umístěni A) testovaný stroj, B) člověk, C) rozhodčí. Rozhodčí může komunikovat se zbylými dvěma formou textových zpráv a má za úkol zjistit, který z nich je stroj a který je člověk. Stroj uspěl v Turingově testu, nedokáže-li ho rozhodčí spolehlivě rozeznat od člověka.

Složení Turingova testu počítačem vyžaduje zvládnutí několika oblastí oboru. Pro porozumění zpráv a jejich generování je potřeba *zpracování přirozeného jazyka (NLP)*. Pro uchování údajů a vyvozování závěrů je nutné ovládnout *reprezentaci a vyvozování znalostí*. V neposlední řadě je potřeba metod *strojového učení*, které umožňují učit se a adaptovat se na změny vnějšího prostředí.

Na druhou stranu není pro složení testu nutné zvládnout obory jako *robotická manipulace* či *počítačové vidění*, jelikož veškerá komunikace probíhá výhradně textovými zprávami.

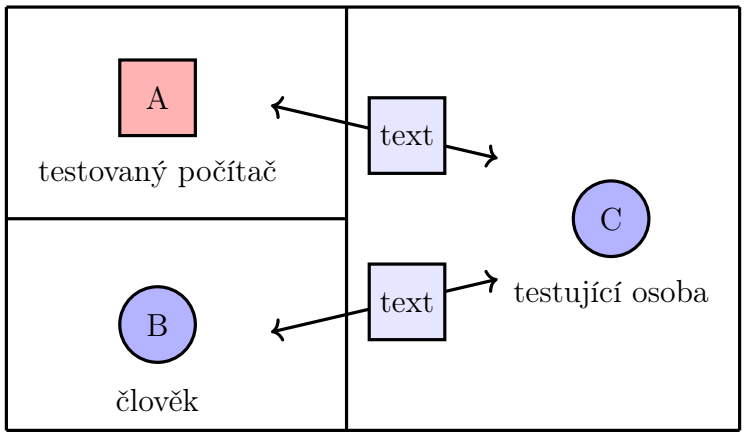


Schéma uspořádání Turingova testu

Příklad 1.1.1. Definujte vlastními slovy následující pojmy:

a) inteligence,

b) umělá inteligence,

c) agent,

d) racionálnost,

e) logické usuzování.

Nalezněte možné námitky proti uvedeným definicím a zpřesněte je.

Příklad 1.1.2. Jsou reflexy racionální? Jsou inteligentní? Zdůvodněte.

Příklad 1.1.3. Do jaké míry jsou následující počítačové systémy příklady umělé inteligence?

a) Čtečka čárových kódů v supermarketu.

b) Internetové vyhledávače.

c) Hlasové zadávání příkazů telefonu.

d) Síťové směrovací algoritmy, které dynamicky reagují na stav sítě.

Příklad 1.1.4. Přečtěte si původní článek Alana Turinga Computing Machinery and Intelligence. Turing v sekci 6 ve článku rozebírá různé námitky vznesené vůči Turingovu testu. Které z námitek jsou stále platné? Napadnou vás nové námitky, které vzešly z pozdějšího vývoje v oboru?

Příklad 1.1.5. Upravíme-li Evansův program pro řešení problémů s geometrickou analogií, aby získal v standardním IQ testu hodnotu 200, jednalo by se o program inteligentnější než člověk? Vysvětlete.

Příklad 1.1.6. Alan Turing ve své práci představil seznam věcí, které by stroje nemusely nikdy umět: být milé, být nápadité, být krásné, být přátelské, být iniciativní, mít smysl pro humor, rozeznat správné od špatného, dělat chyby, zamilovat se, vychutnat si jahody se

šlehačkou, okouzlit někoho, poučit se ze zkušenosti, používat správná slova, přemýšlet samy o sobě, mít chování rozmanité jako člověk, provést něco skutečně nového. Kterých se již podařilo dosáhnout? Které jsou v principu dosažitelné počítačem? Které z nich jsou stále problematické, protože vyžadují vědomé mentální stavy?

Příklad 1.1.7. Argument čínského pokoje je myšlenkový experiment, který se snaží prokázat, že počítače nemohou mít „mysl“, „chápání“ či „vědomí“, a to nezávisle na tom, jak inteligentní chování vykazují. Nastudujte si o argumentu čínského pokoje více. Znamená vyvrácení argumentu čínského pokoje, že vhodně naprogramované počítače mohou mít mentální stavy? Plyne z přijetí argumentu, že počítače mentální stavy mít nemohou?

1.2 Řešení problémů

V této sekci se poprvé setkáme s některými problémy a úvodem do strategií, které lze uplatnit při jejich řešení. K některým z nich se budeme v dalších kapitolách vracet při probírání konkrétních metod řešení problémů (jako například heuristické prohledávání).

Problémem rozumíme zadání množiny konfigurací (např. přiřazení čísel volným políčkům sudoku), mezi nimiž hledáme konfigurace, které splňují konkrétní vlastnosti (např. vlastnost, že je sudoku vyplněno správně). Alternativou k hledání konfigurací splňujících nějakou vlastnost je hledání konfigurací, které jsou v jistém smyslu „nejlepší“. V takovém případě hovoříme o *optimalizačním problému*. Co přesně rozumíme problémem, může být navíc v konkrétním kontextu blíže upřesněno.

Definice 2: Při řešení *optimalizačního problému* je cílem nalézt mezi jeho konfiguracemi takovou, která je mezi všemi ostatními nejlepší podle předem daného kritéria.

Plánujeme-li si semestr tak, abychom získali co nejvíce kreditů a zároveň nám zůstalo co nejvíce volného času na své koníčky a kamarády, řešíme optimalizační problém. Naopak snažíme-li se naskládat nákup do ledničky tak, aby se dala zavřít, problém optimalizační neřešíme, protože jsme spokojeni s jakýmkoliv řešením, které nám umožní dovřít dveře ledničky.

Při řešení problémů hraje důležitou roli počet prohledávaných konfigurací. Přirozenou snahou je tento počet co nejvíce zredukovat, čehož lze dosáhnout využitím znalostí o problému a vhodnou volbou datové struktury.

Příklad. Uvažte *problém n dam*. V problému n dam je rozmisťováno n dam na šachovnici o rozměru $n \times n$. Řešením jsou taková rozmístění (všech dam), v nichž se žádná dvojice dam neohrožuje na řádku, ve sloupci ani diagonálně.

a) Spočtěte počet různých konfigurací problému, tj. počet různých rozmístění dam na šachovnici.

b) Navrhněte datovou strukturu reprezentující konfigurace problému.

c) Navrženou datovou strukturu upravte tak, aby svým návrhem omezovala počet různých konfigurací, ale ne počet potenciálních řešení. Vyjděte z povahy problému a promítněte požadavky na řešení problému do návrhu datové struktury. Spočítejte nový počet konfigurací.

a) Každá dáma může být umístěna na jednu z $n \times n = n^2$ pozic. Pro n dam tedy celkově získáme $\underbrace{n^2 \cdot \ldots \cdot n^2}_{n} = n^{2n}$ rozmístění. Pro 8 dam máme $8^{16} \approx 2,8 \cdot 10^{14}$ konfigurací.

b) Na reprezentaci souřadnic jedné dámy lze použít dvojici (x, y). Konfiguraci pak reprezentuje pole n takových dvojic $[(x_1, y_1), \ldots, (x_n, y_n)]$.

c) Jistě nelze umístit více dam na jedno políčko šachovnice. Můžeme tedy předpokládat unikátnost součadnic a místo pole použít množinu souřadnic $\{(x_1, y_1), \ldots, (x_n, y_n)\}$. Počet konfigurací v takovém případě je $n^2 \cdot (n^2 - 1) \cdot \ldots \cdot (n^2 - n + 1) = \frac{(n^2)!}{(n^2-n)!}$. Konkrétně pro 8 dam je to $64 \cdot 63 \cdot \ldots 57 \approx 1,8 \cdot 10^{14}$ konfigurací.

V návrhu lze jít ještě dále. Z povahy problému víme, že v každém sloupci bude právě jedna dáma – jinak by v nějakém sloupci byly alespoň dvě a ty by se vzájemně ohrožovaly. Dámám v seznamu lze tedy postupně přiřadit x-ovou souřadnici 1 až n: $[(1, y_1), \ldots, (n, y_n)]$ a celý seznam zredukovat na seznam pouze y-ových souřadnic: $[y_1, \ldots, y_n]$. Pokud nepřipustíme opakování hodnot v seznamu (čili více dam na řádku), omezíme počet možných konfigurací na $n \cdot (n-1) \cdot \ldots \cdot 1 = n!$. Pro 8 dam je to $8! = 40320$ konfigurací.

Příklad demonstruje zásadní vliv promítnutí znalostí o problému do přístupu k jeho řešení. Zatímco naivní řešení problému n dam vyžaduje pro $n = 8$ prohledávání prostoru přibližně $2,8 \cdot 10^{14}$ konfigurací, vhodnou úpravou strategie lze číslo redukovat na 40320. Už pro relativně malá čísla n tedy volba strategie rozhoduje o tom, zda je problém prakticky řešitelný či ne.

Příklad 1.2.1. ★ Pro uvedné problémy spočítejte počet různých konfigurací, tj. potenciálních řešení. Pro každý problém navrhněte datovou strukturu pro reprezentaci konfigurací problému.

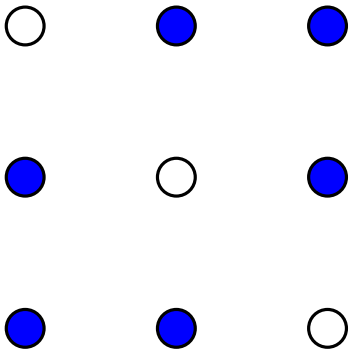
a) *Sudoku.* Uvažujte obecné sudoku $n \times n$ s k hodnotami zadanými.

b) *Problém přiřazení.* V problému přiřazení je dána množina n úkolů T, množina n pracovníků W a cenová funkce $c : T \times W \to \mathbb{R}$, která určuje náklady na vykonání úkolů jednotlivými pracovníky, tj. cena práce pracovníka w na úkolu t je dána jako $c(t, w)$. Cílem je rozdělit úkoly mezi pracovníky (jeden úkol na jednoho pracovníka) tak, aby celková cena práce byla co nejmenší.

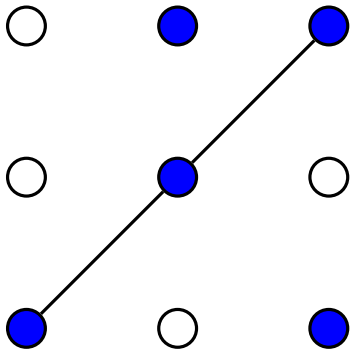
c) *Problém batohu.* Je zadána množina n položek $1, \ldots, n$ o hmotnostech $w_1, \ldots, w_n$ a hodnotách $v_1, \ldots v_n$ a maximální nosnost batohu W. Cílem je z položek vybrat takové,

aby součet jejich hmotností nepřekročil W a zároveň byl co nejvyšší.

Příklad 1.2.2. ★ Uvažte problém *žádné tři v řadě*. V tomto problému je zadána mřížka rozměrů $n \times n$ (kde $n \in \mathbb{N}$) a cílem je zjistit, kolik do ní lze umístit bodů tak, aby žádné tři body neležely v jedné přímce.



Na mřížce 3×3 lze rozmístit maximálně 6 bodů (zvolené body jsou vyplněny modře).



Uvedené rozmístění nesplňuje požadavek, aby tři body neležely v jedné přímce.

a) Spočtěte počet různých konfigurací problému, tj. různých rozmístění bodů (i těch nesplňujících požadavek zadání).

b) Navrhněte datovou strukturu reprezentující konfigurace problému.

c) Navrženou datovou strukturu upravte tak, aby svým návrhem omezovala počet různých konfigurací, ale ne počet potenciálních řešení. Vyjděte z povahy problému a promítněte požadavky na řešení problému do návrhu datové struktury. Spočítejte nový počet konfigurací.

d)* Proveďte asymptotické srovnání počtu konfigurací v naivním a vylepšeném řešení.

e)* Ještě dále vylepšete návrh datové struktury, aby byl počet konfigurací asymptoticky menší než vzorové řešení části c).

Příklad 1.2.3. ★ Které z následujících problémů jsou problémy optimalizační? Zdůvodněte.

a) Problém n dam.

b) Sudoku.

c) Problém přiřazení.

d) Žádné 3 v řadě.

Příklad 1.2.4. Implementujte řešení problému n dam využitím naivního přístupu a přístupu omezujícího počet prohledávaných konfigurací. Experimentálně zjistěte časy nalezení všech řešení pro různá malá n oběma způsoby. Porovnejte s dříve vypočítaným prohledávaným počtem konfigurací. Lze pozorovat úměru mezi časem výpočtu a počtem konfigurací?

Příklad 1.2.5. Implementujte řešení problému žádné 3 v řadě využitím naivního přístupu a přístupu omezujícího počet prohledávaných konfigurací. Experimentálně zjistěte

časy nalezení všech řešení pro různá malá n oběma způsoby. Porovnejte s dříve vypočítaným prohledávaným počtem konfigurací. Lze pozorovat úměru mezi časem výpočtu a počtem konfigurací?

2 Prohledávání stavového prostoru

V předchozí kapitole jsme si vyzkoušeli, jakými způsoby se dá řešit několik klasických problémů, které spadají do oblasti umělé inteligence. Obecně samozřejmě existuje velké množství různých druhů problémů, a tedy i strategií jak je řešit. Často se ale dají využít některé známé univerzální postupy.

V této kapitole si formálně představíme jednu z takových strategií, a to *prohledávání stavového prostoru*. Ukážeme si, jak správně problémy formulovat, představíme si několik různých prohledávacích algoritmů a zaměříme se i na to, v čem se jednotlivé algoritmy liší.

2.1 Formulace problému

Abychom problémy mohli řešit (algoritmicky) pomocí prohledávání, je potřeba je nejdříve formálně zadefinovat. Vhodná formální definice problému se pak skládá z několika hlavních komponent, kterými jsou obvykle

- *iniciální stav*,
- *přechodové akce* (a případně jejich nezáporná *cena*),
- *cílová podmínka*.

Zmíněné komponenty nám dohromady *implicitně* zadávají takzvaný *stavový prostor* problému, tedy množinu všech stavů dosažitelných z iniciálního stavu. Tyto stavy můžeme interpretovat jako vrcholy (uzly) grafu, přechodové akce pak určují hrany daného (přechodového) grafu. Díky této interpretaci můžeme řešit různorodé problémy pomocí obecných algoritmů grafového prohledávání. *Řešení problému* je obvykle posloupnost přechodových akcí, která nás dostane z iniciálního stavu do cílového – hledáme tedy cestu v přechodovém grafu. Zajímá-li nás *optimální řešení*, hledáme obvykle takovou cestu, která je nejkratší (nebo nejlevnější) ze všech řešení.

Schopnost implicitně zadat stavový prostor je v oblasti umělé inteligence velmi důležitá. Stavové prostory uvažovaných problémů mohou být enormní (např. pro hru šachy), nebo dokonce nekonečné. Takové stavové prostory obecně nemůžeme reprezentovat explicitně pomocí množiny všech uzlů a hran, ale musíme si vystačit s implicitním zadáním.

Příklad. Uvažte *problém n dam* z minulé kapitoly. Nejdříve zadefinujte problém formálně. Určete

a) iniciální stav,

b) přechodové akce (a případně jejich cenu),

c) cílovou podmínku.

Až budete mít tuto implicitní definici přechodového grafu, zamyslete jak v takovém případě vypadá graf explicitně (jaká je množina vrcholů a množina hran).

Problém můžeme zadefinovat například následovně.

a) Iniciální stav je prázdná hrací plocha.

b) Přechodová akce je přidání dámy na hrací plochu, pokud ta již neobsahuje n dam. Každá akce má jednotkovou cenu.

c) Cílová podmínka vyžaduje, aby bylo na ploše všech n dam tak, aby se neohrožovaly.

Stavový prostor v takovém případě tvoří množina všech rozestavení *0 až n* dam na hrací ploše. Přechodová hrana je mezi stavy s, t právě tehdy, když s obsahuje o jednu dámu méně než t a zároveň t má všechny kromě jedné dámy na stejných pozicích jako s.

Příklad 2.1.1. ★ Zadefinujte formálně problém *8-posunovačky* z přednášky. Zamyslete se i nad tím, jak bude vypadat přechodový graf.

Příklad 2.1.2. Aplikujeme-li na číslo 4 sekvenci operací *faktoriál*, *odmocnina* a *dolní celá část*, můžeme získat libovolné přirozené číslo[1]. Například, číslo 5 lze získat jako

$$\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \rfloor = 5$$

Formulujte tento problém formálně a opět se zamyslete i nad tím, jak bude vypadat přechodový graf. Liší se nějak stavový prostor od předchozích příkladů?

Příklad 2.1.3. Uvažte zjednodušený problém hledání letecké trasy mezi světovými hlavními městy, který musí řešit webové stránky aerolinek. Předpokládejte, že zákazníci hledají takovou (rozumnou) trasu, která je nejlevnější. Zamyslete se, jak byste řádně formulovali tento problém.

Příklad 2.1.4. Uvažujte situaci, kdy se snažíte naplánovat dovolenou po hlavních městech Evropy. Cesta bude začínat i končit v Praze, nejste nijak časově omezeni a máte v úmyslu procestovat právě všechna evropská hlavní města (některá klidně vícekrát). Jak byste řádně formulovali tento problém?

2.2 Algoritmy prohledávání

Řešením problému prohledávání je posloupnost akcí. Prohledávací algoritmy proto postupně zvažují různé takové možné posloupnosti. Možné posloupnosti akcí začínající v počátečním stavu tvoří takzvaný *prohledávací strom* s počátečním stavem v kořeni. Intuitivně, prohledávací algoritmy postupně expandují dosud neprozkoumané uzly tohoto prohledávacího stromu, který se pak "rozrůstá", až dokud nenarazí na uzel odpovídající cílovému stavu.

Existuje mnoho různých strategií jak prohledávat stavový prostory. Liší se hlavně v tom, jakým způsobem vybrat uzel k expanzi. Výběr vhodného algoritmu závisí na typu problému,

[1] Donald Knuth, 1964

našich požadavcích na průběh výpočtu a našich případných dodatečných znalostech o problému. Pokud o problému nemáme žádnou doplňující znalost, která by nám ulehčila práci, obvykle volíme některý z algoritmů *neinformovaného prohledávání*. Naopak, máme-li nějakou dodatečnou informaci, můžeme ji využít například jako základ heuristiky pro některý z algoritmů *informovaného prohledávání*. Obě tyto skupiny si představíme v následujících sekcích.

Mezi nejdůležitější vlastnosti, podle kterých můžeme nejrůznější algoritmy porovnávat a hodnotit, patří následující čtyři.

Definice 3:

- Algoritmus je *úplný*, jestliže nalezne řešení vždy, když existuje.
- Algoritmus je *optimální*, pokud platí, že nalezne-li nějaké řešení, je toto řešení nejlepší ze všech (například z pohledu ceny nebo délky cesty).
- *Časová složitost* algoritmu udává maximální čas potřebný k vyřešení problému.
- *Prostorová složitost* algoritmu udává maximální množství paměti potřebné k vyřešení problému.

Časovou i prostorovou složitost algoritmu vyjadřujeme v závislosti na nějaké vlastnosti vstupu (typicky jeho velikosti). Obecně uvažujeme složitost *asymptotickou*, a to v *nejhorším případě*. V některých případech však může být výhodné uvažovat i např. složitost očekávanou. Posuzování, zda je algoritmus optimální, je vždy vztaženo ke konkrétní uvažované *metrice* (ta je často jasná z kontextu). Některé algoritmy mohou být optimální z pohledu hloubky či délky řešení (tedy uvažujeme délku cesty v grafu), jiné zase z pohledu obecné ceny řešení (důležité pro ohodnocené grafy). Má-li algoritmus postupně nalézt více řešení, můžeme optimalitu intuitivně chápat tak, že pro každá dvě nalezená řešení algoritmus dříve nalezne to lepší.

Jak bylo uvedeno v předchozí sekci, v oblasti AI prohledávaný stavový prostor typicky reprezentujeme implicitně (pomocí iniciálního stavu a přechodových akcí). K vyjádření složitosti problémů pak používáme následující metriky, které vychází z implicitní reprezentace.

Definice 4:

- *Faktor větvení* (branching factor) b je maximální počet následníků kteréhokoli uzlu.
- *Hloubka cíle* (goal depth) d je délka nejkratší cesty z iniciálního stavu do některého cílového uzlu.
- *Maximální hloubka* (maximum depth) m je délka nejdelší cesty v grafu.

Příklad. Uvažte přechodový graf pro *problém n dam*, který jsme si definovali v ilustrativním

příkladu v Podsekci 2.1. Budeme pracovat s konkrétním grafem pro $n = 8$. Určete pro něj

a) faktor větvení b,

b) hloubku cíle d,

c) maximální hloubku m.

a) Na prázdnou plochu můžeme umístit dámu kdekoli, tedy iniciální stav má 64 následníků. Žádný jiný stav tolik následníků mít nemůže. Proto $b = 64$.

b) K řešení se nemůžeme dostat před přidáním 8 dam. Zároveň existuje rozmístění 8 dam takové, které splňuje cílovou podmínku. Proto $d = 8$.

c) Maximum dam, které můžeme postupně položit je 8, a tedy $m = 8$.

Příklad 2.2.1.
Zamyslete se a pokuste se určit faktor větvení b, hloubku cíle d a maximální hloubku m i pro přechodový graf problému *8-posunovačky* z příkladu v Podsekci 2.1.

Příklad 2.2.2. Uvažme nyní jednoduchou hru piškvorek na hrací ploše velikosti 3×3. Opět určete faktor větvení b, hloubku cíle d a maximální hloubku m i pro stavový prostor tohoto problému.
Zkuste se také zamyslet, kolik existuje možných různých her (posloupností tahů).

2.3 Neinformované prohledávání

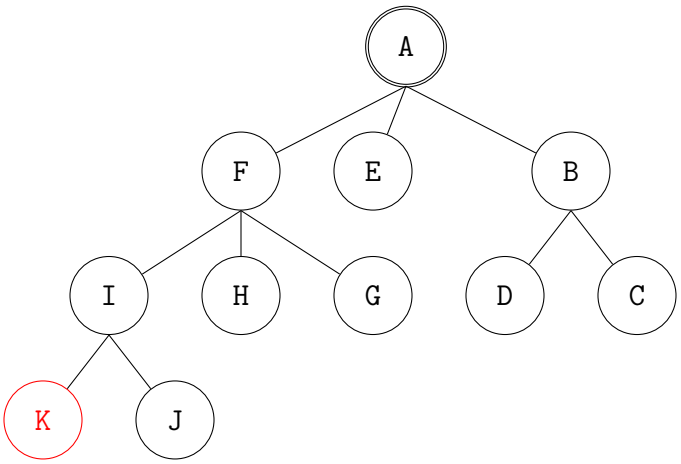
V případě, že o problému nemáme žádnou další informaci (kromě těch z formální definice), jsme obecně nuceni graf prohledávat takzvaně „slepě“. Tyto strategie prohledávají stavový prostor prostým expandováním uzlů a testováním cílové podmínky. Různé algoritmy se liší například v tom, v jakém pořadí uzly expandují. Nejznámějšími strategiemi jsou *prohledávání do šířky* (BFS) a *prohledávání do hloubky* (DFS). S těmito algoritmy jste se jistě hlouběji setkali již v rámci některého předchozího předmětu.

Prohledávání do šířky je jednoduchá strategie, kde nejprve expandujeme iniciální uzel grafu, poté všechny jeho následníky, poté všechny jejich následníky, a tak dále. Obecně platí, že veškeré uzly v určité vzdálenosti („úrovni“) od iniciálního uzlu jsou prozkoumány předtím, než expandujeme uzly v úrovni o jedna vyšší.

Prohledávání do hloubky naopak nejdříve vždy expanduje jen prvního následníka každého uzlu. Tímto způsobem intuitivně prozkoumá vždy „aktuálně nejhlubší“ neprozkoumaný uzel. Jakmile expanduje listový uzel, vrací se zpět pomocí *backtrackingu* a prozkoumává stejným způsobem další dosud nenavštívené následníky.

Mezi další algoritmy, se kterými budeme pracovat, patří *prohledávání do hloubky s limitem*, *prohledávání podle ceny* (uniform-cost search) a *prohledávání s postupným prohlubováním* (IDS). Jistě je znáte z přednášky.

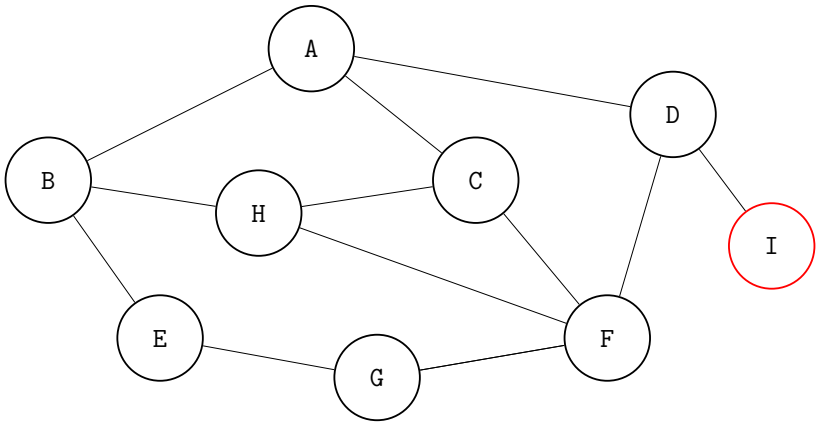
Příklad. Mějme následující graf, který zachycuje stavový prostor problému. Uvažujte situaci, kdy postupně prohledáváme prostor algoritmy BFS a DFS z iniciálního stavu `A`, cílovým stavem je `K`. V jakém pořadí dané algoritmy navštíví jednotlivé stavy? Máte-li na výběr více následníků, postupujte abecedně.



a) BFS: A - B - E - F - C - D - G - H - I - J - K
b) DFS: A - B - C - D - E - F - G - H - I - J - K

Příklad 2.3.1. Uvažujte stejný strom jako v předchozím příkladě. Jak se změní pořadí uzlů, uvažujeme-li tentokrát, že je iniciálním stavem `G`? Cílový stav zůstává `K`.

Příklad 2.3.2. ★ V předchozích příkladech jste si vyzkoušeli simulaci algoritmů na stromovém stavovém prostoru. Uvažte nyní následující graf. Určete v jakém pořadí algoritmy BFS a DFS navštíví jednotlivé stavy, uvažujeme-li nejdříve stav `A` jako iniciální, a poté také stav `B` jako iniciální. Uvažujte, že výpočet skončí nalezením prvního řešení.



Opět, máte-li na výběr více následníků, postupujte abecedně.

Příklad 2.3.3. ★ Uvažte opět graf z předchozícho příkladu. Nechť tentokrát spustíme prohledávání do hloubky z iniciálního uzlu `I` a nechť cílový uzel je nyní `F`. Uvažujte, že výpočet skončí nalezením prvního řešení.

a) Je algoritmem nalezené řešení optimální?

b) Co kdybychom namísto DFS použili DFS s limitem $l = 4$?

c) A jak by tomu bylo při použití prohledávání s postupným prohlubováním (IDS)?

Příklad 2.3.4. Mějme opět situaci, kdy prohledáváme výše uvedený graf. Tentokrát nechť je iniciální stav `A` a cílový stav `F`. Jako algoritmus zvolíme DFS s limitem. Není-li řečeno jinak, uvažujte, že výpočet skončí nalezením prvního řešení.

a) Jaký je nejvyšší limit, pro který algoritmus nalezne optimální řešení?

b) Od jaké hodnoty limitu algoritmus nalezne vždy stejné řešení?

c) Uvažujte scénář, kdy prohledáváme celý graf (výpočet neskončí prozkoumáním cílového stavu). Od jaké hodnoty limitu bude výpočet pro DFS s limitem probíhat zcela stejně jako pro klasické DFS?

Příklad 2.3.5. Vymyslete příklady přechodových grafů (případně přímo problémů) takových, že:

a) Prohledávání do hloubky nikdy nenalezne řešení.

b) Prohledávání do hloubky nalezne řešení dříve než prohledávání do šířky.

c) Prohledávání do hloubky bude procházet uzly ve stejném pořadí jako prohledávání do šířky.

Každou odrážku řešte zvlášť.

Příklad 2.3.6. Rozhodněte o pravdivosti následujících tvrzení, své odpovědi odůvodněte. Můžete předpokládat že prohledáváme strom s konečným faktorem větvení, kladnou cenou přechodů a alespoň jedním dosažitelným cílovým uzlem.

a) Prohledávání do hloubky s limitem je úplné.

b) Prohledávání s postupným prohlubováním má stejnou prostorovou složitost jako prohledávání do hloubky.

c) Prohledávání s postupným prohlubováním má horší časovou složitost než prohledávání do šířky.

d) Prohledávání do šířky je optimální.

Příklad 2.3.7. Popište stavový prostor takový, že prohledávání s postupným prohlubováním má daleko větší složitost než prohledávání do hloubky (jako například $\mathcal{O}(n^2)$ vs. $\mathcal{O}(n)$).

Příklad 2.3.8. Uvažte stavový prostor, kde iniciální stav je číslo 1 a každý stav k má dva následníky – čísla $2k$ a $2k + 1$.

a) Načrtněte část stavového prostoru pro stavy 1 až 15.

b) Předpokládejte, že cílový stav je 11. Určete v jakém pořadí algoritmy BFS, DFS s limitem $l = 3$ a prohledávání s postupným prohlubováním navštíví jednotlivé stavy.

c) Pokuste se nalézt algoritmus, který vypíše řešení bez jakéhokoli prohledávání. Využijte znalostí o doméně a formulaci problému.

2.4 Heuristické prohledávání

Máme-li o problému nějakou dodatečnou informaci, často ji můžeme využít k efektivnějšímu prohledávání stavového prostoru. Konkrétně se budeme zabývat strategiemi, které při výběru uzlů k expandování využívají informaci o (odhadu) blízkosti stavů k cíli. Tyto algoritmy obvykle expandují nejdříve ty uzly, které se jeví (v určitém smyslu) jako *nejslibnější*. Souhrnně se tyto prohledávácí strategie označují jako *best-first search*. Informaci o tom, jak přínosný se jeví daný uzel (jaký je odhad jeho ceny), nám dává takzvaná *ohodnocovací funkce* $f(n)$. V průběhu výpočtu si držíme prioritní frontu uzlů uspořádaných dle této ceny a expandujeme stav s aktuálně nejmenší cenou.

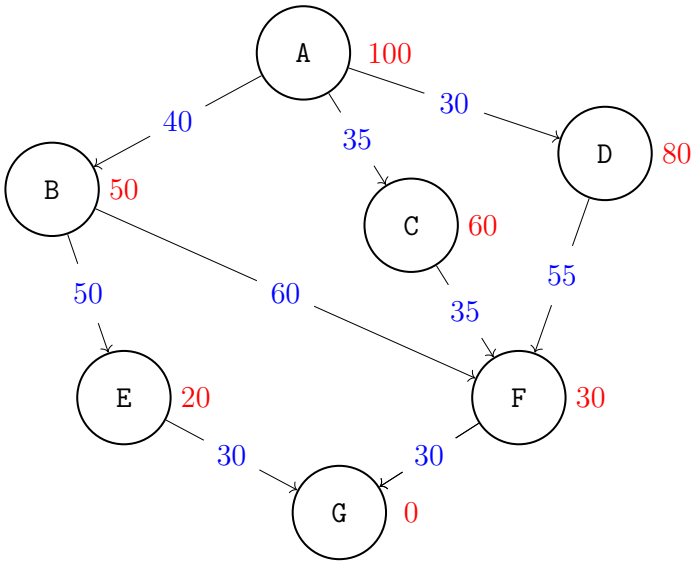
Jako komponentu při odhadu ceny většina algoritmů používá *heuristickou funkci* $h(n)$. Tato funkce reprezentuje určitou dodatečnou znalost o doméně problému, a dává jakýsi odhad na cenu zbývající cesty ze stavu k cíli. Aby byl tento odhad užitečný, některé algoritmy kladou na použité heuristiky podmínky. Jednou z nich je takzvaná *přípustnost*. Ta intuitivně říká, že heuristický odhad ceny cesty z uzlu do cíle nesmí být větší než cena opravdová. V jistém ohledu silnější podmínkou je pak *konzistence* heuristiky.

Definice 5:
Ohodnocovací funkce $f(n)$ pro každý uzel n dává celkový odhad jeho ceny. Uzly s nejnižším odhadem jsou expandovány nejdříve.

Heuristická funkce $h(n)$ pro každý uzel n dává odhadovanou cenu nejlevnější cesty z n do cílového uzlu. Na heuristiky klademe omezení, že musí být nezáporné, a pokud je n cílový uzel, pak $h(n) = 0$. Pro heuristiky dále platí:

1. Heuristická funkce h je *přípustná* pokud pro všechny stavy n platí $0 \leq h(n) \leq h'(n)$, kde $h'(n)$ je skutečná cena nejkratší cesty ze stavu n do cíle.
2. Heuristická funkce h je *konzistentní* (neboli *monotonní*) pokud pro každý stav n a každého jeho následníka m platí, že $h(n) \leq c(n, m)+h(m)$, kde $c(n, m)$ je cena přechodu z n do m.

V několika dalších demonstrativních příkladech budeme pracovat s následujícím ohodnoceným grafem reprezentujícím stavový prostor.



Pro daný graf platí, že ohodnocení přechodů jsou zobrazena modře, například `A-B` má cenu 40. Červenou barvou pak jsou pak zaznačeny hodnoty heuristiky pro každý stav, například heuristická hodnota stavu `A` je 100. Počáteční stav je `A`, cílový stav je `G`.

Stručně si představme dvě konkrétní strategie informovaného prohledávání. První je známa jako *greedy best-first search*, neboli *hladové heuristické hledání*. Druhý algoritmus se pak nazývá $A*$. Obě strategie mají společné to, že si v průběhu výpočtu udržují prioritní frontu s uzly k expandování, uspořádanými podle hodnoty $f(n)$. V čem se algoritmy liší, je vzorec, pomocí kterého počítají $f(n)$. Tento rozdíl se zdá být malý, ale celkově jsou jak úvahy za

oběma algoritmy, tak i jejich vlastnosti, velmi odlišné.

Greedy best-first search jednoduše ohodnocuje uzly jen podle hodnoty heuristiky, platí pro něj $f(n) = h(n)$. Algoritmu se říká *hladový*, protože se v každém kroku snaží dostat tak „blízko“ k cíli, jak jen to jde. To vede například k tomu, že algoritmus není obecně úplný ani optimální.

A* je sofistikovanější algoritmus. Kombinuje užitečné vlastnosti *Dijkstrova algoritmu* (doporučujeme si zopakovat) a zároveň rychlost heuristického prohledávání. Při ohodnocování uzlu bere kromě heuristiky v potaz i dosavadní cenu cesty, kterou jsme se do uzlu dostali, označme ji $g(n)$. Ohodnocovací funkce má tedy tvar $f(n) = g(n) + h(n)$. Použijeme-li heuristiku s vhodnými vlastnostmi, algoritmus je pak garantovaně optimální.

Příklad. Uvažujte situaci, kdy prohledáváme výše uvedený přechodový graf pomocí hladového heuristického prohledávání.

a) V jakém pořadí navštíví algoritmus jednotlivé stavy? Je-li v některém kroku na výběr více možností, postupujte abecedně.

b) Jak bude vypadat výsledná nalezená cesta? Je toto řešení optimální?

Pořadí stavů je `A-B-E-G`, což je v tomto případě zároveň i výsledná cesta. Toto řešení ovšem optimální není. Zkuste se zamyslet nad lepší cestou.

Příklad. Nyní uvažujte prohledávání daného grafu pomocí algoritmu A*.

a) Je heuristika zvolená v daném příkladu přípustná? A je konzistentní?

b) V jakém pořadí navštíví A* jednotlivé stavy? Je-li v některém kroku na výběr více možností, postupujte abecedně.

c) Jak bude vypadat výsledná nalezená cesta? Liší se od cesty nalezené hladovým algoritmem? Je řešení tentokrát optimální?

a) Heuristika je přípustná, jelikož pro každý stav je hodnota heuristiky menší nebo rovna ceně cesty do cílového stavu. Není ale konzistentní, jelikož $h(A) > c(A, C) + h(C)$.

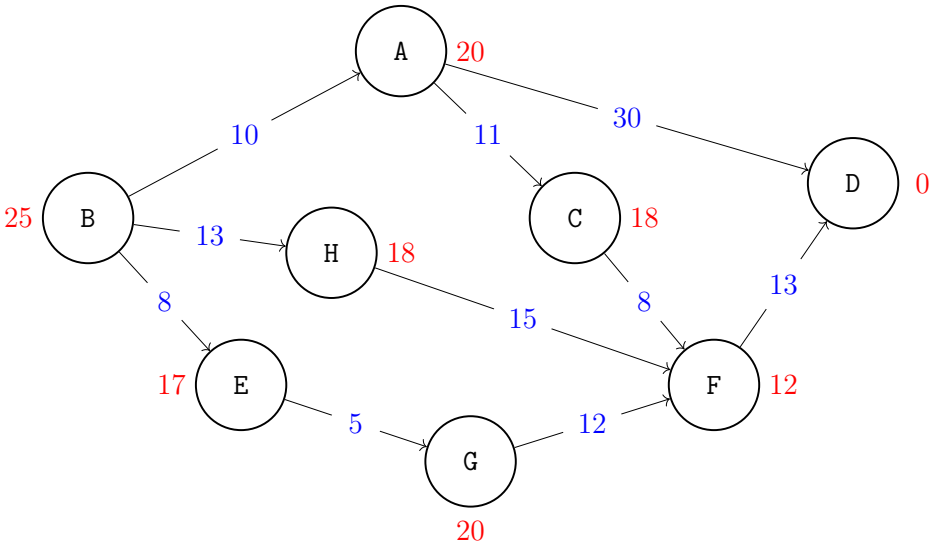
b) Pořadí stavů při výpočtu je `A-B-C-F-G`.

c) Výsledná cesta má tvar `A-C-F-G`, a tedy se liší od cesty nalezené hladovým algoritmem. Můžete si ověřit, že toto řešení je opravdu optimální.

Příklad 2.4.1. ★ Uvažme následující stavový prostor. Iniciální stav je `B`, cílový stav je `D`. Ohodnocení přechodů jsou opět zobrazena modře, hodnoty heuristiky červeně. Simulujte výpočet hladového heuristického algoritmu. Je-li více stavů ohodnoceno shodně, postupujte abecedně.

a) V jakém pořadí navštíví algoritmus jednotlivé stavy?

b) Jak bude vypadat výsledná nalezená cesta? Je optimální?



Příklad 2.4.2. ★ Opět simulujte prohledávání grafu z předchozího příkladu, tentokrát ale pomocí algoritmu A*. Je-li více stavů ohodnoceno shodně, postupujte abecedně.

a) Je heuristika zvolená v daném příkladu přípustná? A je konzistentní?

b) V jakém pořadí navštíví A* jednotlivé stavy?

c) Jak bude vypadat výsledná nalezená cesta?

Příklad 2.4.3. Uvažujte, že věž se může na šachovnici pohybovat o jakýkoli počet políček po přímce, vertikálně nebo horizontálně, ale nemůže přeskakovat ostatní figurky. Je Manhattanská vzdálenost přípustná heuristika pro problém posunutí věže z políčka A na políčko B v co nejmenším počtu tahů? Svou odpověď odůvodněte.

Příklad 2.4.4. ★ Rozhodněte o pravdivosti následujících tvrzení. Odpovědi zdůvodněte. Pokud není řečeno jinak, uvažujte konečný faktor větvení, cenu přechodů vyšší než nějaké kladné ϵ a alespoň jeden dosažitelný cílový uzel.

a) Hodnota přípustné heuristiky nikdy nepřevyšuje zbylou opravdovou cenu (vzdálenost) do cíle.

b) Algoritmus heuristického prohledávání, jehož fronta je uspořádána dle hodnoty $f(n) = g(n) + h(n)$ je úplný i optimální, pokud používá přípustnou heuristiku a zároveň ohodnocení uzlů $f(n)$ monotónně stoupá po jakékoliv cestě do cíle.

c) Prohledávání podle ceny je jak úplné, tak i optimální, pokud cena cesty nikdy neklesá.

d) Hladové heuristické prohledávání je jak úplné, tak i optimální, pokud je použitá heuristika přípustná a cena cesty nikdy neklesá.

Příklad 2.4.5. Která z následujících tvrzení jsou pravdivá? Odpovědi zdůvodněte.

a) Prohledávání do hloubky vždy expanduje alespoň tolik uzlů jako A* s přípustnou heuristikou.

b) $h(n) = 0$ je přípustná heuristika pro 8-posunovačku.

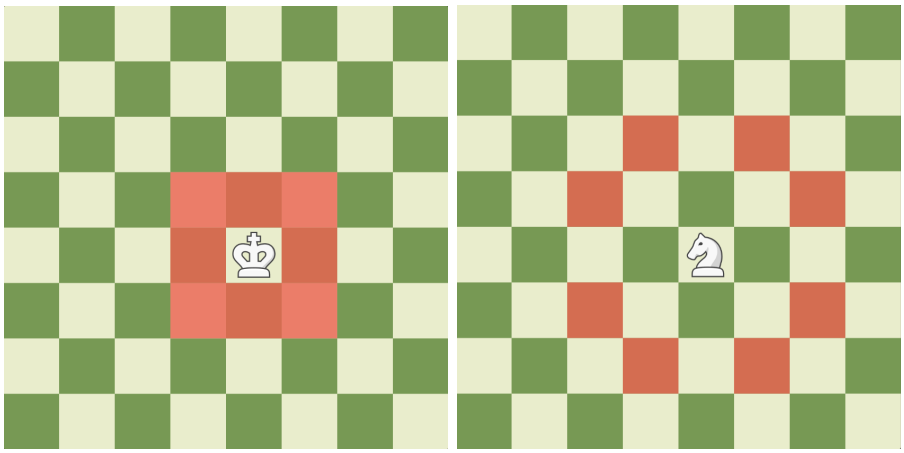
c) Prohledávání do šířky je úplné i pokud jsou povoleny přechody s nulovou cenou. Uvažujeme konečný faktor větvení.

d) Součet několika přípustných heuristik je opět přípustná heuristika.

Příklad 2.4.6. Ke každému z následujících prohledávacích problémů navrhněte alespoň jednu netriviální heuristiku pro A* algoritmus, která bude zároveň přípustná i konzistentní.

a) Na šachovnici 8×8 potřebujeme posunout figurku krále z horního levého rohu do dolního pravého rohu. Na šachovnici nejsou žádné jiné figurky. Král se může pohybovat na kterékoli z okolních osmi políček. Při návrhu heuristiky můžete použít x, y k označení vzdálenosti do cílového pole na osách x a y.

b) Na šachovnici 8×8 potřebujeme posunout figurku jezdce (koně) z horního levého rohu do dolního pravého rohu. Na šachovnici nejsou žádné jiné figurky. Jezdec se pohybuje klasicky podle pravidel šachů (viz obrázek). Při návrhu heuristiky opět doporučujeme použít x, y k označení vzdálenosti do cílového pole na osách x a y.



Možné tahy králem a jezdcem na šachovnici.

Příklad 2.4.7. Mějme problém pohybu figurky jezdce na níže zobrazené hrací ploše o rozměrech 3×4 z iniciálního pole S do cílového pole G. Čísla označují hodnotu heuristiky v daném stavu. Jezdec se pohybuje klasicky podle pravidel šachů (viz minulý příklad). Všechny přechody (pohyby jezdce) mají jednotkovou cenu.
Nejdříve načrtněte stavový prostor. Poté pro algoritmy DFS, BFS a A* určete v jakém pořadí navštíví jednotlivé stavy. Předpokládejte, že algoritmy negenerují cesty s cykly a je-li na výběr více následníků, postupují abecedně.

S (3)	H (1)	D (1)	K (1)
I (1)	J (2)	A (2)	E (2)
C (1)	B (2)	**G** (0)	F (3)

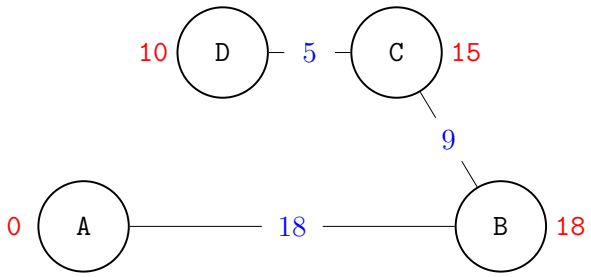
Příklad 2.4.8. Uvažujte prohledávání (konečného) stavového prostoru algoritmem A* s konzistentní heuristikou. Nechť existuje jeden dosažitelný cílový uzel n_g a nechť $C*$ je cena optimálního řešení.

a) Může se stát, že algoritmus neexpanduje některý uzel n_1, pro který platí $f(n_1) < C*$?

b) Může algoritmus prohledat jen a pouze uzly n takové, že $f(n) < C*$?

c) Můžeme před spuštěním výpočtu určit, kolik uzlů m takových, že $f(m) > C*$, bude expandováno?

Příklad 2.4.9. Mějme následující (neorientovaný) graf zadávající stavový prostor. Ohodnocení přechodů jsou jako obvykle zobrazena modře, hodnoty heuristiky červeně, cílový stav je `A`. Vašim úkolem bude simulovat *tree-search* verzi algoritmu hladového heuristického prohledávání. Pro *tree-search* verzi prohledávacích algoritmů platí, že se neudržuje seznam již navštívených uzlů (neřeší tedy cykly). Simulujte prohledávání ze stavu `C`. Jaké bude pořadí prohledávaných uzlů?



Příklad 2.4.10. Jsou následující tvrzení ohledně heuristik pravdivá? Své odpovědi dokažte.

a) Každá přípustná heuristika je i konzistentní.

b) Každá konzistentní heuristika je i přípustná.

c) Pro libovolný stavový prostor vždy existuje heuristika, která je přípustná i konzistentní zároveň.

Příklad 2.4.11. Rozhodněte a dokažte, zda jsou následující tvrzení o prohledávacích algoritmech pravdivá.

a) BFS je speciální případ prohledávání podle ceny (uniform-cost search).

b) Algoritmy DFS, BFS i uniform-cost search jsou speciálními případy best-first search.

c) Prohledávání podle ceny je speciální případ A* prohledávání.

3 Dekompozice problému, Problémy s omezujícími podmínkami

Tato kapitola se zaměřuje na některé aplikace prohledávání grafů, kterému se věnovala kapitola předchozí. Konkrétně se bude zabývat AND/OR grafy a spektrem problémů, na které je lze aplikovat – od problémové dekompozice, přes hry až po modelování logických výrazů.

Ve druhé části se budeme zabývat deklarativním přístupem k programování, jenž umí být značně elegantní a stručný. Na jeho pozadí se opět vykonává prohledávání stavového prostoru, které v mnoha případech funguje dobře, jindy však naráží na hranice efektivity.

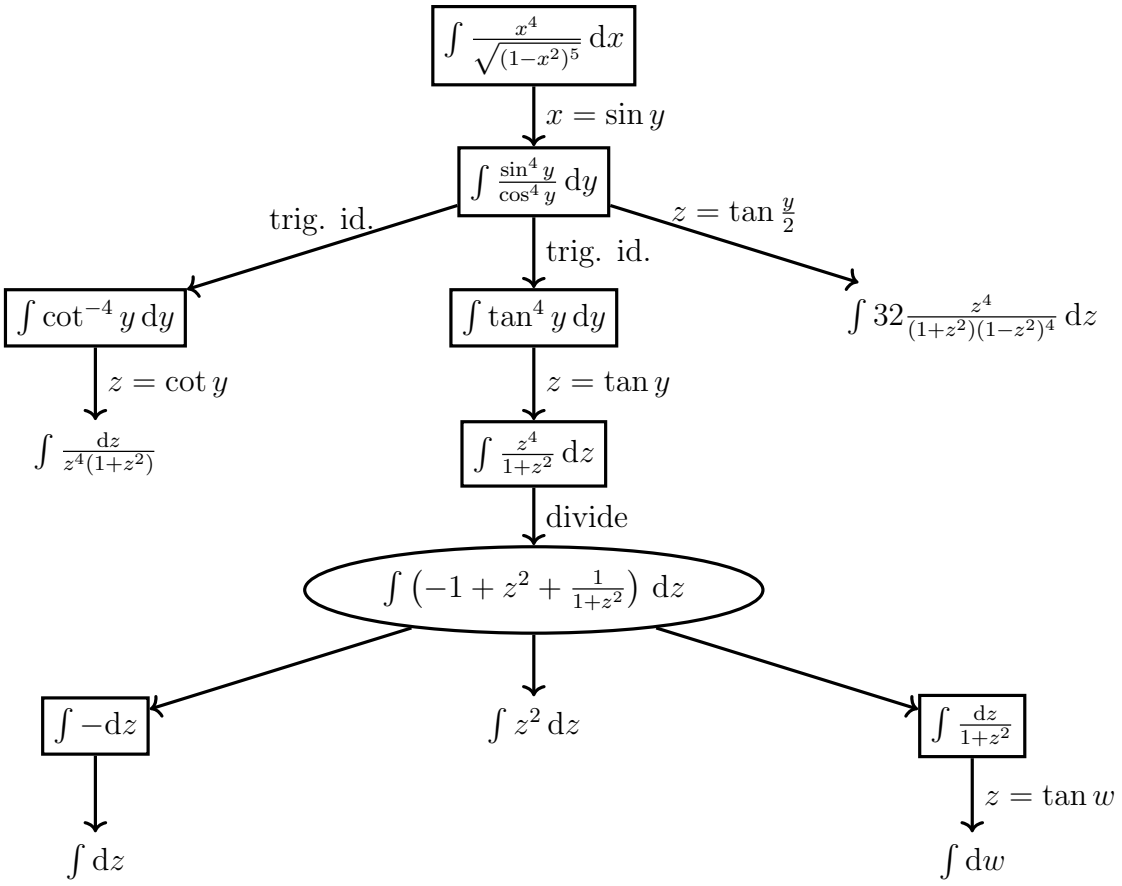
3.1 Dekompozice problému a AND/OR grafy

Definice 6: *AND/OR graf* je orientovaný graf s vrcholy typu AND nebo OR (souhrnně zvané vnitřní) a sérií koncových vrcholů $t_1, t_2, \ldots, t_n$.

Typ vnitřního vrcholu nejčastěji interpretujeme tak, že k vyřešení OR uzlu je potřeba vyřešit kterýkoliv z následníků (potomků). V případě AND uzlu musíme vyřešit všechny následníky (potomky). Koncové vrcholy pak reprezentují dále nedělitelné podproblémy, ať už neřešitelné či se známým řešením. V jiném kontextu lze koncové vrcholy chápat jako splněné či nesplněné.

Jako příklad aplikace takového přístupu na problémovou dekompozici (a tedy i příklad AND/OR grafu) si uvedeme strategii řešení neurčitého integrálu. Postup, který si zde ilustrujeme, se v nějaké míře nachází u většiny matematických symbolických řešičů – jako je například WolframAlpha.

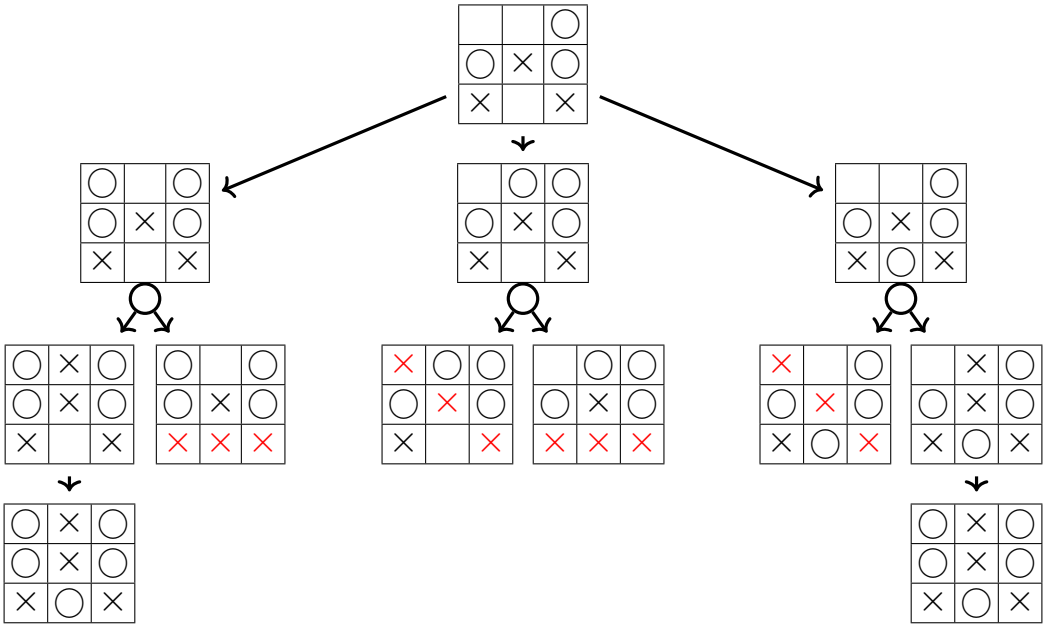
Na následujícím obrázku značíme *AND vrcholy elipsou*, zatímco *OR vrcholy obdélníkem*. Koncové uzly necháváme bez označení. Vrcholy s jediným následníkem mohou být AND i OR bez vlivu na sémantiku (význam) grafu – my v takovém případě volíme konzistentně obdélníkové ohraničení. OR vrcholy zde tedy reprezentují možné přístupy k řešení, kdežto AND uzel rozděluje problém na sadu podproblémů, které je třeba vyřešit všechny. V některých případech, kde je orientace hran grafu zřejmá (například z rozmístění vrcholů), se šipky v grafu vynechávají.



Častěji budeme vídat AND/OR grafy v kontextu tahových her dvou hráčů – například piškvorek. Omezíme se na *hry s perfektní informací*, což stručně znamená, že vždy známe úplný aktuální stav hry. Mimo piškvorek toto splňují například šachy či go, tímto způsobem by však nebylo možné modelovat třeba poker či kostky.

V některých ilustracích grafů her budou v dalším textu pro lepší přehlednost využívána kolečka pro vyznačení uzlů typu AND. Uzly typu OR a koncové uzly nebudou speciálně vyznačeny a mezi sebou je lze jednoduše rozlišit.

Následující AND/OR graf znázorňuje rozehranou hru piškvorek na hracím poli 3×3. Na tahu je hráč s kolečky, zvažující všechny své možné tahy. Stačí mu, aby pouze *jeden* z jeho tahů byl výherní či alespoň končící remízou, proto je vrchol grafu reprezentující aktuální stav hry *typu OR*. Uzly o úroveň níže představují možnou odpověď soupeře, jež může být libovolná. Hráč s kolečky se musí umět vyhnout prohře u *všech* možných protihráčových reakcí, tudíž jsou uzly ve druhé úrovni *typu AND*. V dalších úrovních se typy uzlů dále střídají.

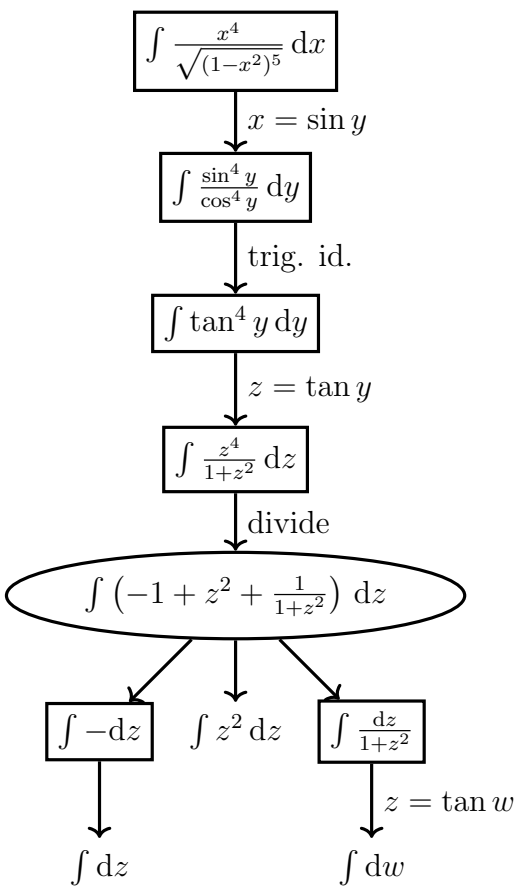


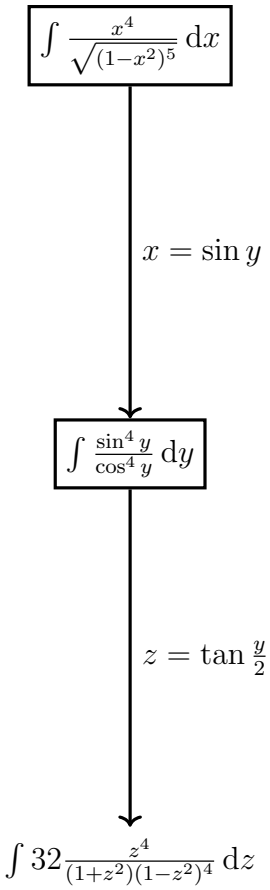
Analýzou grafu lze dojít k závěru, že za předpokladu, že soupeř neudělá chybu, hráč s kolečky prohrál. Každý z AND uzlů na druhé úrovni je nesplněný, neboť alespoň jeden z jeho následníků značí prohru hráče s kolečky (tři křížky v řadě). Počáteční OR vrchol reprezentující aktuální stav hracího pole je tedy také nesplněný, ačkoliv by bylo možné dosáhnout remízy s nedokonalým soupeřem.

Definice 7: *Stromem řešení T problému P s AND/OR grafem G* je podgraf grafu G, který je stromem a

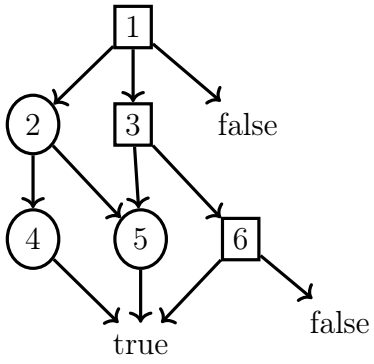
- jeho kořen je vrchol reprezentující problém P,
- je-li N vnitřní uzel T typu AND, pak každý jeho následník v G je i v T,
- je-li N vnitřní uzel T typu OR, pak právě jeden z jeho následníků v G je i v T.

Pro náš příklad s výpočtem integrálu může strom řešení vypadat například ilustrovanými dvěma způsoby. Praktický význam má však spíše prvně vykreslený strom, kde koncové vrcholy umíme snadno vyřešit.

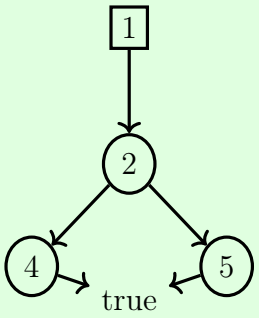


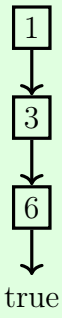


Příklad. Rozhodněte, zda je počáteční uzel, značený 1, splněný v následujícím AND/OR grafu. Vrchol chápeme jako splněný tehdy, když pro něj existuje v daném grafu strom řešení se všemi koncovými vrcholy nastavenými na *true*. Spočtěte také, kolik takových stromů řešení existuje.



Ano, svoji odpověď můžeme potvrdit ukázkou příslušného stromu řešení. My ukazujeme všechny 3 možné stromy řešení.





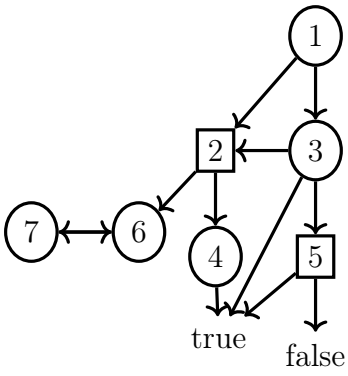
Příklad 3.1.1. Rozdělení na vnitřní a koncové uzly v definici AND/OR grafu není potřeba. Zamyslete se, jak ji upravit tak, abychom s pomocí vnitřních vrcholů mohli modelovat i uzly koncové.

Příklad 3.1.2. ★ Ukažte, že každý AND/OR graf lze převést na ekvivalentní bipartitní AND/OR graf, ve kterém jsou následníky vrcholů typu AND vždy vrcholy typu OR a opačně. Jaké to má praktické důsledky pro implementaci AND/OR grafů?

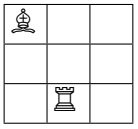
Příklad 3.1.3. Seznamte se s dekompozicí problému Hanojských věží z přednášky, kde problém redukujeme na jednoduchý AND/OR graf, kde jsou všechny vnitřní uzly typu AND. Ke kolika fyzickým přesunům disku dojde při počtu kotoučů n rovno

a) 1, b) 3, c) 4?

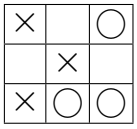
Příklad 3.1.4. ★ Rozhodněte, zda je počáteční uzel, značené 1, splněn v následujícím AND/OR grafu. Vrchol chápeme jako splněný tehdy, když pro něj existuje v daném grafu strom řešení se všemi koncovými vrcholy nastavenými na *true*. Spočtěte také, kolik takových stromů řešení existuje.



Příklad 3.1.5. Uvažme hru odehrávající se na hracím poli velikosti 3×3, kde se hráči střídají a každý posouvá svou figuru podle pravidel šachů, nesmí však vstoupit na pole, které již bylo dříve obsazeno. Podaří-li se hráči sebrat soupeři figuru, vyhrává, naopak je-li hráč na tahu a nemůže se nikam pohnout (neboť dosažitelná pole již byla v minulosti obsazena), prohrává. Úkolem je zanalyzovat situaci, kde proti sobě hraje věž a střelec, první zmíněný je na tahu a počáteční rozmístění je dáno obrázkem níže. Zkonstruujte příslušný AND/OR graf a určete, kdo v takovém případě vyhraje.



Příklad 3.1.6. ★ Uvažte rozehranou partii piškvorek zobrazenou na obrázku, v níž je právě na tahu hráč kreslící kolečka. Sestrojte AND/OR graf pro tuto partii. Nalezněte stromy řešení tohoto AND/OR grafu a interpretujte jejich vztah k výsledku partie.



3.2 Problémy s omezujícími podmínkami

V *imperativním programování* je dosahováno řešení problémů zadáváním přesných postupů jejich řešení (algoritmů). Problémy s omezujícími podmínkami (constraint satisfaction problem, CSP) jsou příkladem *programování deklarativního*, které naopak spočívá v zadání (neboli deklaraci) očekávaných vlastností řešení. Mezi příklady problémů s omezujícími podmínkami se řadí problém obarvení grafu, algebrogram či problém N dam.

Příklad. U následujícího programu poveďte typovou inferenci (tj. odvození typů) funkcí funkcí `f` a `g`. Neuvažujte přetížení, funkce budou mít vždy jeden typ.

```
x = f(g(0))
x = f(x)
x = g(x)
```

a) Jaká omezení (angl. constraints) na typy funkcí `f` a `g` lze z programu vyčíst?

b) Nalezněte řešení soustavy nalezených omezení, tj. určete typy funkcí `f` a `g`.

a) Z programu lze vyčíst následující sadu omezení. Jejich konkrétnější podoba by závisela na uvažovaném typovém systému.
   - `f` bere argument stejného typu, který vrací `g`
   - `g` bere argument typu `int`
   - `f` vrací stejný typ, který bere
   - `g` bere argument stejného typu, který vrací `f`

b) Řešením soustavy lze dojít k závěru, že obě uvedené funkce jsou typu `int` → `int`.

Uvažujme následující sudoku. Cílem je napsat sadu omezení popisující jeho řešení.

	2						9	
3		1	9		6	5		2
			8		4			
	9						5	
5			2		3			6
	7						2	
			4		7			
8		2	5		1	7		3
	5						8	

Prvním krokem je vždy nalézt proměnné daného problému, tedy hodnoty, které popisují řešení daného problému a jejichž hodnoty chceme nalézt. V případě sudoku se nabízí zavést si sérii proměnných $x_1, \ldots, x_n$, kde každá reprezentuje hodnotu prázdného políčka, přičemž n je počet prázdných políček. Při deklaraci proměnné je třeba specifikovat její doménu, tedy množinu hodnot, jakých může nabývat. Z pravidel sudoku plyne, že každá proměnná x_i může nabývat celočíselných hodnot mezi 1 a 9.

V platném řešení sudoku musí mít všechna pole ve stejném řádku, sloupci a čtverci jinou hodnotu, což je nutné v programu zohlednit. Při pojmenování prázdných políček popořadě po řádcích by omezení pro první řádek bylo $x_1 \neq 2, x_1 \neq x_2, \ldots, x_2 \neq 2, x_2 \neq x_3, \ldots$. Mnohé programovací jazyky umožňují místo podobné série podmínek napsat jediný výraz ve smyslu `ALL_DISTINCT`$(x_1, x_2, x_3, \ldots, x_7)$, při omezení domén těchto 7 proměnných na $\{1, 3, 4, 5, 6, 7, 8\}$. Tento zápis je jednak úspornější, a jednak je možnost řešič optimalizovat pro toto často se vyskytující omezení.

Na začátku této sekce jsme zmínili, že takovýto styl programování měl být zlatým grálem informatiky. Není těžké si představit proč – místo psaní dlouhého kódu aplikace bych jen zavedl sérii podmínek jako „když kliknu sem, stane se...“. Takové deklarativní programování by samozřejmě bylo poněkud složitější, než tu ukazujeme, ale to je nakonec imperativní kód ještě mnohem víc. Kde je tedy problém? Pozorný čtenář již jistě uhodl, že v samotném řešiči. Skutečně efektivní umíme sestrojit pro určité typy omezení, všude jinde spoléháme na chytré heuristiky při prohledávání všech možných ohodnocení proměnných.

Praktické využití má však dnes CSP při rozvrhování (používá ho i naše fakulta) či konstrukci mikročipů. S jeho pomocí lze modelovat kromě zmíněné typové inference a logických hádanek také maximální tok grafem či lexikální analýzu.

Po nepříliš stručném úvodu jsme připraveni si tyto nové pojmy definovat formálněji.

Definice 8: *Problém s omezujícími podmínkami (CSP)* je

- soubor proměnných $X_1, \ldots, X_n$, každá s neprázdnou doménou $D_1, \ldots, D_n$;
- soubor omezení $C_1, \ldots, C_m$; každé omezení je podmnožinou $D_1 \times \ldots \times D_n$;
- (někdy) účelová funkce $f : D_1 \times \ldots \times D_n \to \mathbb{R}$.

Řešením nazveme takovou n-tici $(x_1, \ldots, x_n) \in D_1 \times \ldots \times D_n$, která splňuje všechna omezení C_i, tj. $\forall i.(x_1, \ldots, x_n) \in C_i$.
Má-li CSP více než jedno řešení, může nás zajímat některé konkrétní, potom využíváme účelové funkce f a hledáme takové řešení, které funkci maximalizuje (či minimalizuje). V některých zdrojích pak takový problém značíme COP (*constraint optimization problem*).

Příklad. Uvažme následující program. (Syntax neodpovídá přesně syntaxi z přednášky, ale je dostatečně intuitivní na to, aby nepotřebovala vysvětlení.)

```
X in 1..7
Y in 1..7
X + Y = 7
```

a) Kolik má takový problém řešení?

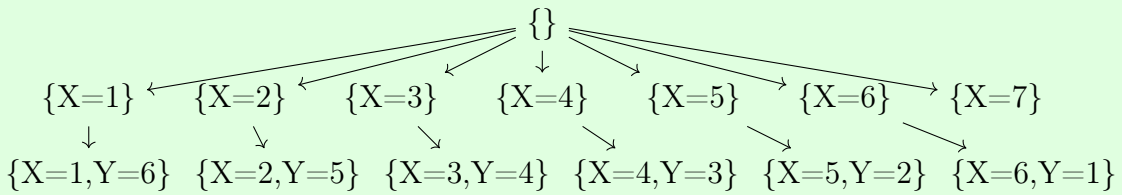
b) Kolik má řešení, přidáme-li omezení `X * Y is even`?

c) Jak vypadá příslušný graf stavů, který prohledáváme?

a) 6 – konkrétně `{(X=1,Y=6),(X=2,Y=5),(X=3,Y=4),(X=4,Y=3),(X=5,Y=2),(X=6,Y=1)}`.

b) 6, jelikož vždy právě jedna proměnná `X`, `Y` musí být sudá, aby v součtu daly 7. Přidané omezení tedy nemění zadání.

c) Začínáme v prázdném stavu, kde neznáme hodnotu ani jedné proměnné. Z něj se můžeme dostat do stavu, kde hodnota proměnné `X` je celé číslo mezi 1 a 7, jelikož jedině tyto stavy jsou v souladu se zadáním. Z každého z těchto stavů, kde známe pouze hodnotu proměnné `X` se však již dostaneme v souladu s omezeními v zadání do nanejvýš jediného stavu, kde je dána i hodnota proměnné `Y`. Všimněte si, že v jednom uzlu (`{X=7}`) naše cesta končí, z něj se nedokážeme dostat při splnění podmínek programu. Přísně vzato v našem grafu existuje mnoho jiných vrcholů – například `{X=3,Y=3}` – ty nekreslíme, neboť do nich nevede žádná cesta z počátečního stavu, a tak jsou pro naše prohledávání irelevantní.



Příklad 3.2.1. ★ Sestavte graf stavů pro následující CSP. Proměnné přiřazujte v sekvenčním pořadí podle jejich deklarace. Popište řešení.

```
A in 2..4
B in 2..3
C in 0..6

A - B >= C
A * (B-1) != B + C
A != B
```

Příklad 3.2.2. Převeďte následující algebrogram na CSP. Každé písmenko P v rozepsaném součinu zastupuje prvočíselnou číslici – ne však nutně *stejnou*. Řešení hledat nemusíte. Nezapomeňte specifikovat zavedené proměnné a jejich domény.

```
   PPP
    PP
 -------
  PPPP
 PPPP
 -------
 PPPPP
```

Příklad 3.2.3. ★ Jako kouzelný čtverec označíme čtvercovou matici, kde součet čísel na každém řádku a v každém sloupci je stejný. Jednotlivé buňky mohou nabývat celočíselných hodnot mezi 1 a n^2, kde n je dimenze uvažované matice.

Jedním z netriviálních řešení pro $n = 3$ je tento kouzelný čtverec.

2	9	4
7	5	3
6	1	8

Formulujte tento problém jako CSP pro uvedené $n = 3$. Kromě omezení nezapomeňte uvést význam zavedených proměnných a jejich domény.

Příklad 3.2.4. Latinský čtverec o velikosti n je čtvercová matice $n \times n$, obsahující v každém řádku (a stejně i sloupci) neopakující se čísla $1, \ldots, n$. Každý řádek i sloupec je tedy permutace na množině $\{1, \ldots, n\}$.

Uvádíme konkrétní příklad pro $n = 3$.

1	2	3
2	3	1
3	1	2

Formulujte tento problém jako CSP pro uvedené $n = 3$. Kromě omezení nezapomeňte uvést význam zavedených proměnných a jejich domény.

Příklad 3.2.5. Zajímavou aplikací CSP je známá Einsteinova hádanka, někdy nazývána zebra. Jejího autora neznáme, často se za něj uvádí Albert Einstein či Lewis Carroll. Týká se pěti sousedů, kteří žijí na ulici v domech v řadě vedle sebe. Každý z nich má jiné povolání, jiný dům, chová jiné zvíře, je jiné národností a preferuje jiné pití. Máme k dispozici několik výroků.

- Angličan žije v červeném domě.
- Španěl chová psa.

- Japonec je malíř.
- Ital pije čaj.
- Nor žije v prvním domě nalevo.
- Majitel zeleného domu pije kávu.
- Zelený dům je bezprostředně napravo od bílého.
- Sochař chová šneky.
- Diplomat žije ve žlutém domě.
- V prostředním domě pijí mléko.
- Nor žije vedle modrého domu.
- Houslista pije ovocný džus.
- Liška žije vedle doktora.
- Kůň je ustájen v domě vedle diplomatova.

K úloze patří i otázka. Kdo má zebru a kdo pije vodu?

Vašim úkolem není tuto úlohu vyřešit (ačkoliv to není těžké), ale převést ji na CSP, z jehož řešení bychom uměli odpovědět na zadanou otázku.

Příklad 3.2.6. Zamyslete se, jak formulovat jako CSP problém rozmístění jedné sady šachových figurek po klasické šachovnici velikosti 8×8 tak, aby se žádné dvě figurky neohrožovaly. Pěšce nemusíte uvažovat, stačí tedy umístit krále, dámu, 2 střelce, 2 koně a 2 věže.

4 Hry a herní strategie

Tato kapitola se zabývá hraním deterministických her dvou hráčů s úplnou informací. Bude zde představen jednoduchý algoritmus MINIMAX, který je však ve své základní podobě nedostačující pro řešení složitějších her jako piškvorky na velké hrací ploše či šachy kvůli přílišné velikosti prohledávaného stavového grafu. Řešením tohoto problému je zefektivnění algoritmu v podobě alfa-beta prořezávání.

V další části kapitoly budou představeny aplikace těchto postupů i na nedeterministické hry dvou hráčů.

4.1 MINIMAX

Zabývejme se nyní spolu *tahovou hrou pro dva hráče s perfektní informací*. Dobrým příkladem takové hry, která nás již nějakou dobu touto sbírkou provází, jsou piškvorky. Nejprve se spolu podíváme na hru, kde nám přestává stačit AND/OR graf, a poté přejdeme k MINIMAX grafu.

Pokud chceme modelovat hru s komplikovanějším vyhodnocením než jen výhra či prohra, tedy takovou, kde končíme s nějakým skóre, AND/OR grafy nám to neumožní. Koncové vrcholy totiž mohou být jen splněny či nesplněny, nejsme v nich schopni ukládat číselnou hodnotu skóre a ani s ní následně vyhodnocovat splněnost vnitřních uzlů. Budeme tedy generalizovat myšlenku AND/OR grafů, ovšem omezíme se teď již pouze na stromy.

Definice 9: *MINIMAX strom* je strom, kde každý vrchol je buď typu MIN, nebo MAX. Každý list je ohodnocen rozšířeným reálným číslem – tj. prvkem množiny $\mathbb{R} \cup \{-\infty, \infty\}$.

MINIMAX strom se vyhodnocuje směrem od listů, dokud nespočteme hodnotu kořene. Hodnota vrcholu typu MAX je rovna maximální hodnotě jeho následníků, hodnota uzlu typu MIN pak odpovídá hodnotě nejmenší napříč všemi následníky.

Definice 10: *Algoritmus MINIMAX* je rekurzivní procedura, která ohodnotí každý vrchol MINIMAX stromu.

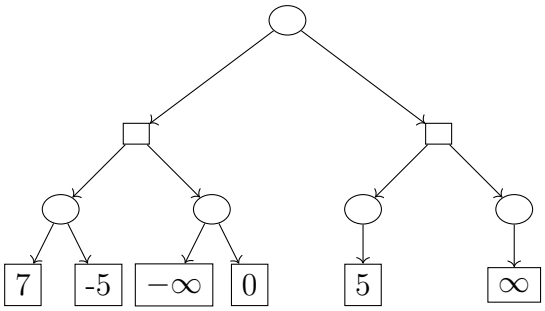
- Hodnota listů je dána jako součást definice MINIMAX stromu.
- Hodnota vrcholu typu MIN je nejmenší hodnota mezi jeho následníky.
- Hodnota vrcholu typu MAX je největší hodnota mezi jeho následníky.

Vypůjčíme si notaci AND/OR grafů a uzly typu MAX budeme značit obdélníkem $\square$ zatímco MIN vrcholy elipsou $\bigcirc$. Jiný častý způsob značení používá pro vrcholy MAX symbol $\bigtriangleup$, pro MIN uzly $\bigtriangledown$. Jelikož MINIMAX stromy použijeme pouze pro modelování tahové hry

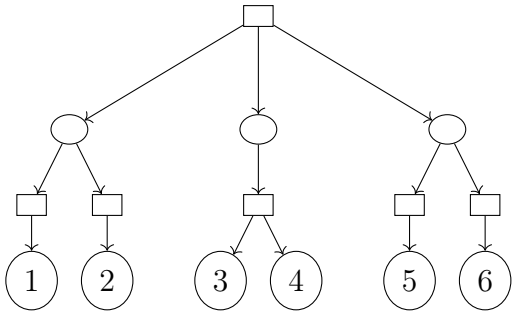
dvou hráčů, budeme se zabývat především těmi stromy, kde se uzly MIN a MAX střídají podle své vzdálenosti od kořene. Dále budeme uvažovat pouze hry s nulovým součtem, což znamená, že obdržím-li v nějaké hře skóre s, můj soupeř musí získat $-s$ bodů – každý list nám tak bude stačit ohodnotit jen podle našeho skóre a soupeřovo si snadno domyslíme. Jak se později ukáže, tato myšlenka je velmi důležitá: MINIMAX stromy budou fungovat jen pro hry s nulovým součtem či obecně hry, kde platí, že když maximalizuji svoje skóre, zároveň minimalizuji to soupeřovo – a naopak.

Příklad. Určete hodnotu kořene zadaných MINIMAX stromů.

a)

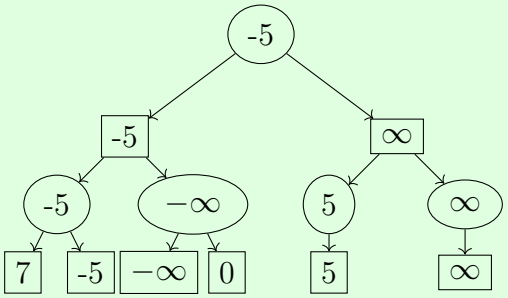


b)

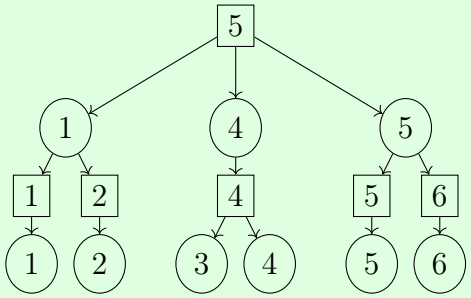


Vyhodnocujeme směrem od listů, uzel po uzlu.

a)

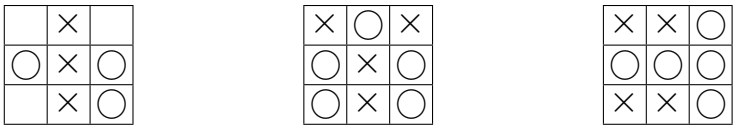


b)



Nyní nám zbývá se podívat, jakým způsobem lze využít tohoto modelu k popisu her. A jako obvykle si při ilustraci pomůžeme piškvorkami na hracím poli 3×3.

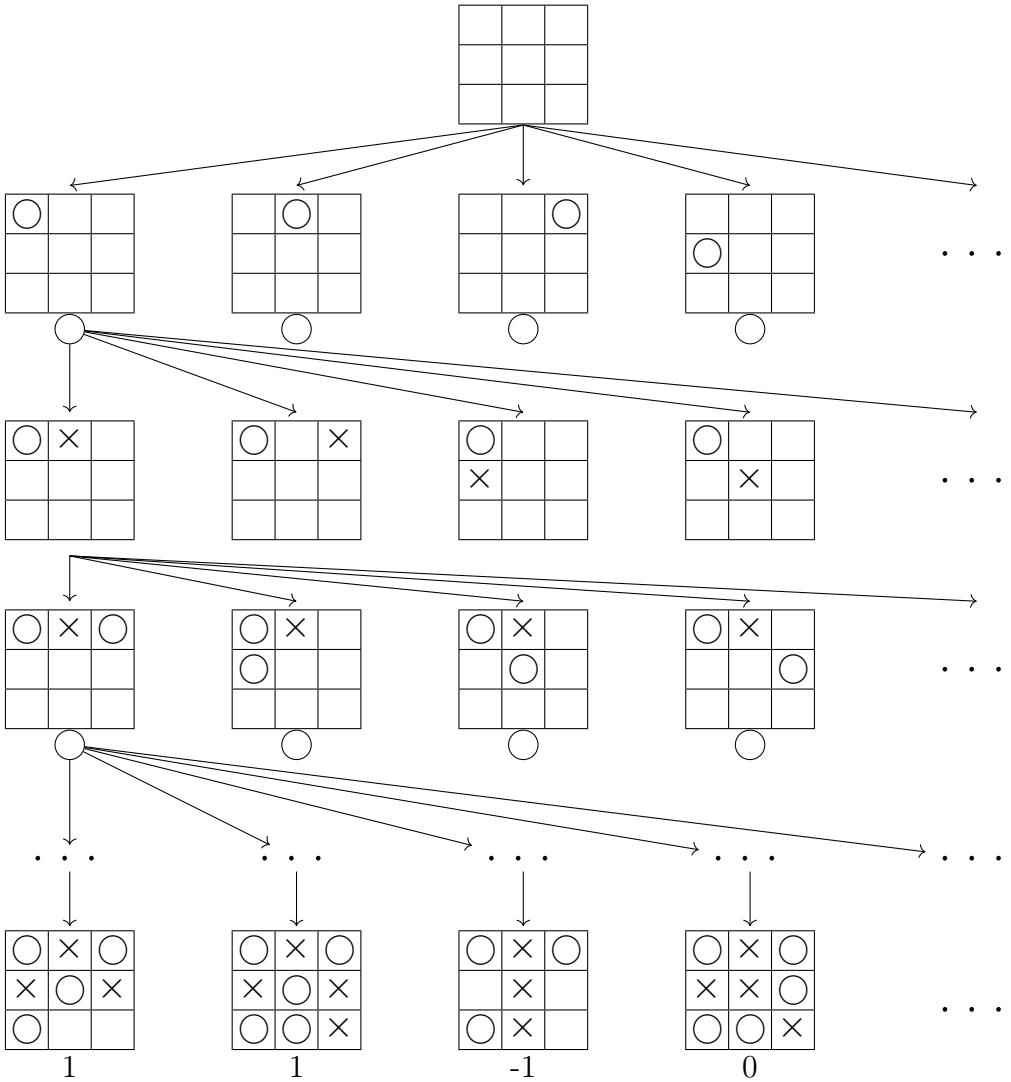
Každý list v našem stromu stavů (tedy herních konfigurací) ohodnotíme 0, 1 či -1 podle toho, zda se jedná o remízu, naši výhru či prohru. V této hře budeme první na tahu a kreslíme kolečka. Prohlédněme si tři vybrané listy a jejich přiřazenou hodnotu.



Podle našeho ohodnocení je jasné, že kdykoliv budu na tahu já, budu se snažit svoje skóre *maximalizovat*, kdežto protihráč vždy *minimalizovat*. Uzly grafu, kde jsem na řadě já tedy budou typu MAX a ostatní typu MIN.

Uvědomme si, že ne všechny listy jsou ve stejné hloubce (hra může skončit po různém počtu kol). Některé vrcholy se mohou vyskytovat duplicitně, neboť do nich lze „dojít“ různými cestami.

Připomínáme, že vrcholy typu MIN budeme značit malým kolečkem dole, ostatní vrcholy jsou pak typu MAX. Následuje ukázka malé části celého grafu.



Příklad 4.1.1. ★ Rozhodněte, zda je následující hry možné modelovat MINIMAX stromy. Pokud ano, zamyslete se nad tím, jak by takové stromy vypadaly.

a) šachy

b) sudoku

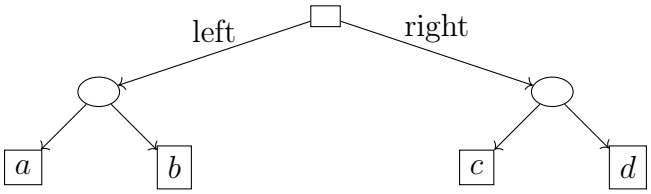
c) piškvorky na neomezené hrací ploše

d) kámen, nůžky, papír

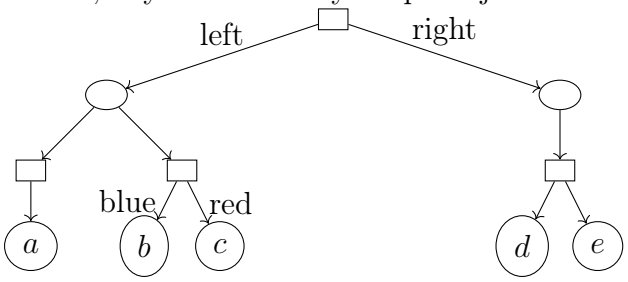
e) tenis

Příklad 4.1.2. ★ Doplňte nějaké konkrétní ohodnocení koncových stavů her s naznačeným MINIMAX stromem tak, aby začínající hráč dosáhl nejlepšího možného výsledku hraním zadaných strategií, předpokládáme-li, že jeho soupeř nedělá chyby. Jaké vztahy obecně musí v takovém případě platit pro hodnoty listů?

a) Výherní strategie nechť je *left*.



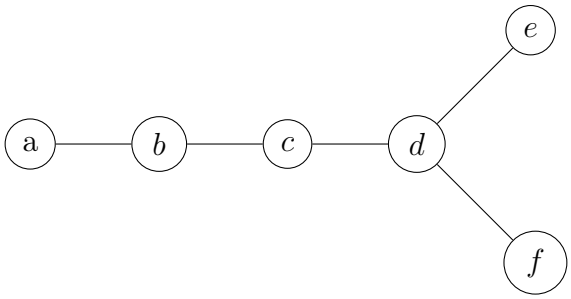
b) Výherní strategie nechť je volit *left* v prvním kole a *blue* ve druhém. Nezapomeňte zařídit, aby nás dokonalý soupeř s jistotou dovedl k volbě potřebné strategie.



Příklad 4.1.3. ★ Dokažte, že MINIMAX stromy rozšiřují AND/OR stromy. Jinými slovy ukažte, že každý AND/OR strom lze chápat jako MINIMAX strom.

Příklad 4.1.4. Ukažte, že obecný MINIMAX strom lze efektivně transformovat na ekvivalentní MINIMAX strom, kde se střídají typy vrcholů – tzn. následník má vždy jiný typ než jeho rodič. Efektivní transformací rozumíme takový algoritmus, která je polynomiální vzhledem k počtu vrcholů vstupního stromu. Ekvivalentní MINIMAX strom musí mít stejné listy se stejnými hodnotami jako strom původní a algoritmus MINIMAX musí napočítat stejnou hodnotu v kořeni.

Příklad 4.1.5. Uvažme hru dvou hráčů – pronásledovatele P a zloděje Z – na hracím poli zadaném náledujícím grafem.



Pronásledovatel je první na tahu a začíná na poli b, zloděj začíná na poli d. Každý se během svého tahu posouvá na vedlejší pole, přičemž se po jednom kroku střídají. Hra končí, jestliže se zároveň ocitnou na stejném poli. Zloděj se snaží přežít co nejdéle a hodnota koncové konfigurace je pro něj úměrná tomu, kolik kol odehrál.

- Lze tuto hru modelovat MINIMAX stromem? Lze ji vyřešit?
- Vykreslete a vyřešte strom pro hru omezenou 3 koly – tedy hráč P i Z se každý posunou nejvýše třikrát.

4.2 Alfa-beta prořezávání

Řešení deterministické hry s perfektní znalostí můžeme dostat i efektivnější procedurou, než je MINIMAX algoritmus. Zkusme ji spolu objevit.

Až do této chvíle jsme pro zjednodušení tvrdili, že algoritmus MINIMAX vyhodnocuje vrcholy směrem od listů bez toho, abychom podrobněji vysvětlili, proč tomu tak je a jak to detailně funguje. Podívejme se detailně na pseudokód této procedury.

```
def minimax(node):
    if node.neighbors is empty:  # nachazime se v listu
        return node.value
    best_so_far = minimax(node.neighbors[0])
    # vynech nulteho naslednika (souseda), protoze jsme ho jiz zpracovali
    for neighbor in node.neighbors[1:]:
        value = minimax(neighbor)
        if node.type == "MIN" and value < best_so_far:
            best_so_far = value
        if node.type == "MAX" and value > best_so_far:
            best_so_far = value
    return best_so_far
```

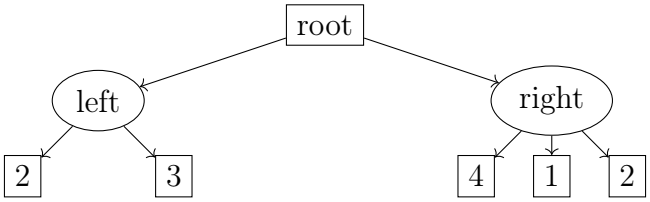
Příklad. Jaká je asymptotická časová složitost algoritmu MINIMAX?

Algoritmus probíhá zcela stejně jako prohledávání do hloubky (DFS), v každém uzlu se provedou navíc jen nějaká porovnání s konstantní časovou složitostí. Výsledná asymptotická časová složitost tedy je $\mathcal{O}(V+E)$, kde V je počet vrcholů prohledávaného grafu a E je počet

hran. Toto lze vidět i z toho, že každý uzel a každá hrana se v průběhu výpočtu navštíví právě jedenkrát.
V našem případě uvažujeme pouze souvislé grafy, můžeme tedy výsledek napsat stručněji jako $\mathcal{O}(E)$.

Spustíme-li výpočet nad kořenem MINIMAX stromu (tj. zavoláme `minimax(root)`), dostaneme se do for cyklu, kde proceduru rekurzivně zavoláme nad prvním následníkem kořene atp. až se dostaneme k listu. Jedná se tedy o prohledávání do hloubky s několika přidanými výpočty.

Podívejme se teď na jednoduchý příklad.



Budeme předpokládat, že uspořádání seznamu následníků odpovídá tomu na nákresu při čtení zleva doprava. Při vykonávání volání `minimax(root)` se tedy nejprve dostaneme do uzlu *left* a skrze něj okamžitě do listu s hodnotou 2. Dosavadní nejlepší (a tedy v tomto případě nejmenší) hodnota pro vrchol *left* je tedy 2, ale může se pochopitelně změnit při procházení dalšího následníku, ke kterému dochází hned vzápětí. V tomto případě se však nic nezmění a konečně zjišťujeme, že hodnota uzlu *left* je 2.

Kořen tedy skončil zpracování prvního následníka a nejlepší hodnota, kterou zatím viděl, je 2. Protože je typu MAX, víme, že nižší hodnotu už mít nemůže, ale jeho další následník (*right*) by ji mohl ještě zvýšit.

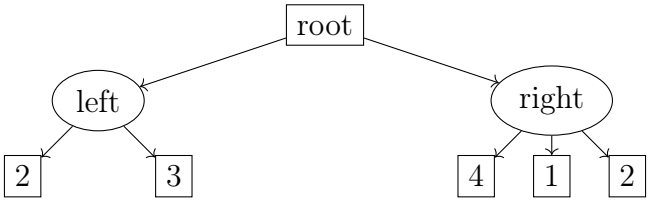
Při vyhodnocování uzlu *right* nejprve vidíme hodnotu 4, poté 1. Zde se na chvíli zastavíme (těsně předtím než bychom zpracovali poslední list). Víme, že konečná hodnota vrcholu *right* nebude vyšší než 1, protože minimalizuje. Na druhou stranu jsme si již před začátkem zpracování tohoto uzlu všimli, že maximalizující kořen *root* nebude mít nižší hodnotu než 2, kterou mu nabízí jeho levý následník. Prohledávání tedy můžeme okamžitě ukončit, jakýkoliv další následník uzlu *right* je nezajímavý a nemůže změnit hodnotu kořene, ačkoliv by pochopitelně ještě mohl snížit hodnotu svého předchůdce.

Ačkoliv jsme si v tuto chvíli ušetřili průchod jediným uzlem, představíme-li si místo listů našeho vzorového příkladu dostatečně velké podstromy, můžeme si hodně pomoci. Tato úspora se neprojeví na nejhorší časové složitosti algoritmu ALFA-BETA, zlepší se však v průměrném případě nad omezenou vstupní množinou MINIMAX stromů. Hodnotu kořenového uzlu však napočítáme stejnou (což obecně neplatí o ostatních vnitřních uzlech), tedy ALFA-BETA je korektní.

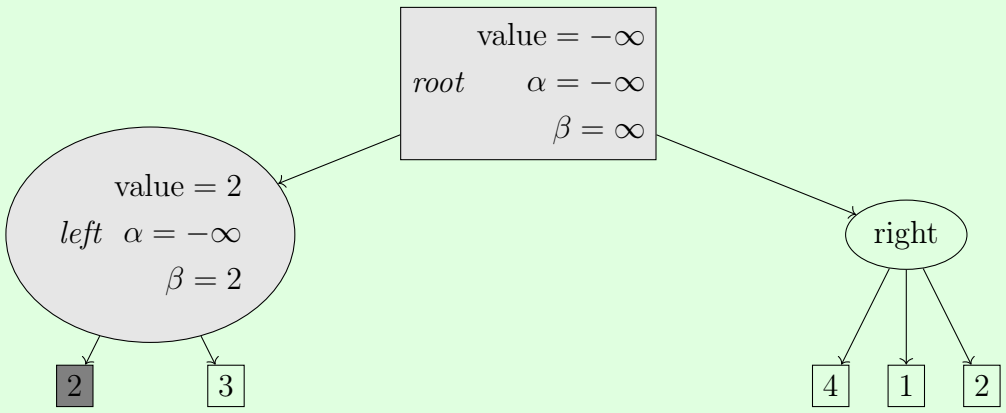
Nyní se již spolu podíváme na implementaci algoritmu ALFA-BETA. Pro přehlednost zcela oddělujeme rutinu pro uzly typu MIN a MAX, z důvodu stručnosti používáme v našem pseudokódu nekonečno `INFINITY` v inicializaci. Iniciální volání spustíme nad kořenem stromu obdobně jako předtím, tedy příkazem `alphabeta(root, -INFINITY, INFINITY)`.

```
def alphabeta(node, alpha, beta):
    if node.neighbors is empty:  # nachazime se v listu
        return node.value
    if node.type == "MAX":
        value = -INFINITY  # aby se prepsala pri objevu prvni hodnoty
        for neighbor in node.neighbors:
            value = max(value, alphabeta(neighbor, alpha, beta))
            if value >= beta:
                break
            alpha = max(alpha, value)
        return value
    else:
        value = INFINITY
        for neighbor in node.neighbors:
            value = min(value, alphabeta(neighbor, alpha, beta))
            if value <= alpha:
                break
            beta = min(beta, value)
        return value
```

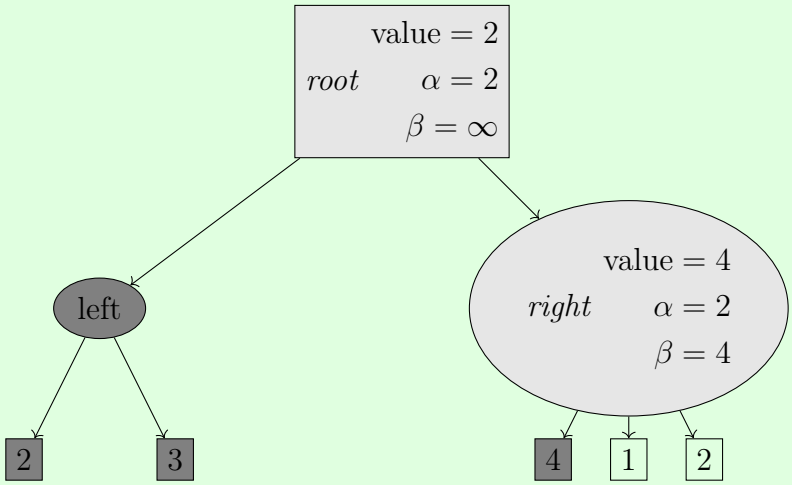
Příklad. Odkrokujte si algoritmus nad motivačním příkladem z úvodu této podsekce a přesvědčte se, že skutečně dojde k ořezání výpočtu tak, jak jsme předpověděli.



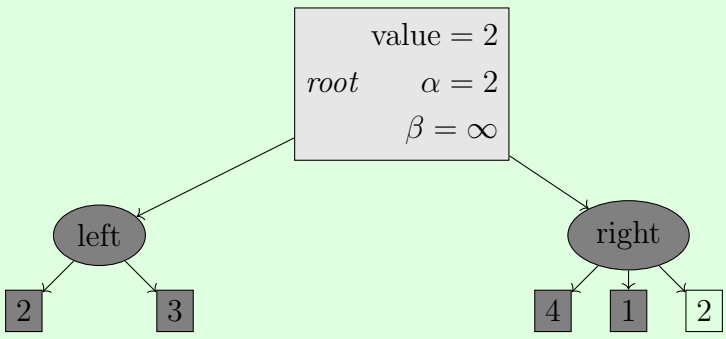
Výpočet začíná voláním `alphabeta(root, -∞, ∞)`, počáteční hodnoty jsou tedy $\alpha = -\infty, \beta = \infty$ a začínáme zpracovat kořenový vrchol. Při jeho výpočtu je potřeba najít hodnotu uzlu *left*, který se tedy bude muset vyčíslit (se stejnými hodnotami alfy a bety). Nejprve se objeví list s hodnotou 2, po jeho zpracování ukazuje hodnoty proměnných následující ilustrace. Světle šedý uzel je rozpracovaný, tmavě šedý je již úplně zpracovaný.



Následně se obdobně zpracuje list s hodnotou 3, čímž dokončíme výpočet hodnoty vrcholu *left* a vrátíme se s výpočtem zpět do kořene *root*. Ten aktualizuje svoje hodnoty a spustí výpočet nad svým druhým následníkem *right*, přičemž jsme se u levého následního dostali až k uzlu s hodnotou 3.



Jako další se zpracovává list s hodnotou 1, kdy se konečně projeví smysl zavedených proměnných α a β. Jelikož se nám hodnota *value* nastaví na 1, což je méně než aktuální hodnota proměnné α ve vrcholu *right*, výpočet v tomto uzlu se ukončí (a vrátí hodnotu 1 do kořene *root*).



Výpočet v tuto chvíli končí s výslednou hodnotou 2, přičemž poslední list jsme během výpočtu vůbec neviděli.

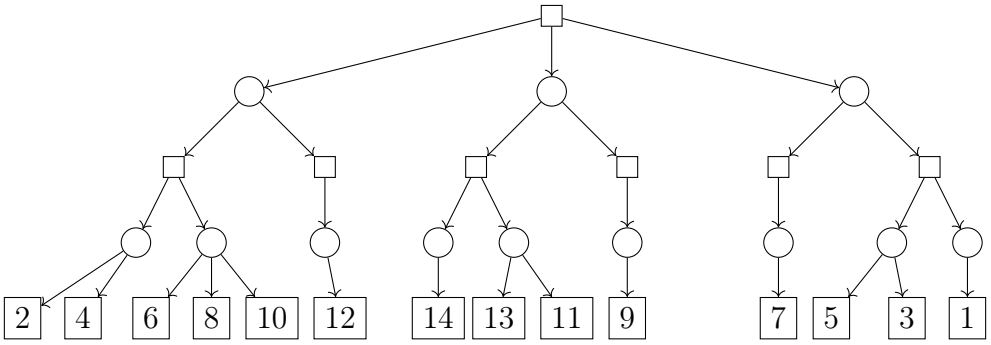
Příklad 4.2.1. Ukažte, že asymptotická časová složitost v nejhorším případě algoritmu ALFA-BETA nad MINIMAX stromy je stejná jako pro algoritmus MINIMAX.

Příklad 4.2.2. ★ Zadaný strom MINIMAX střídá po úrovních typy svých uzlů a každý jeho list je ve stejné hloubce. Takové v literatuře (a hlavně v praxi) budete vídat nejčastěji, neboť přirozeně vznikají při modelování mnoha zajímavých her. Jako v celém zbytku sbírky předpokládejte i zde, že algoritmus prochází vrcholy v pořadí zleva doprava.

a) Vyřešte strom s pomocí algoritmu ALFA-BETA. Především tedy zjistěte výslednou hodnotu kořene a které podstromy budou uřezány, tj. nebudou vůbec navštíveny. Při výpočtu věnujte zvláštní pozornost tomu, abyste porozuměli, jakým způsobem α odpovídá dolní hranici a β naopak horní.

b) Při zachování struktury grafu navrhněte vhodné hodnoty listů tak, aby nedošlo k žádnému prořezávání (a algoritmus tedy navštívil všechny listy).

c) Při zachování struktury grafu navrhněte vhodné hodnoty listů tak, aby došlo k největšímu možnému prořezávání (a algoritmus tedy navštívil co nejméně uzlů).



Příklad 4.2.3. ★ Při kterých z následujících transformací MINIMAX stromu může dojít ke změně nalezené optimální strategie?

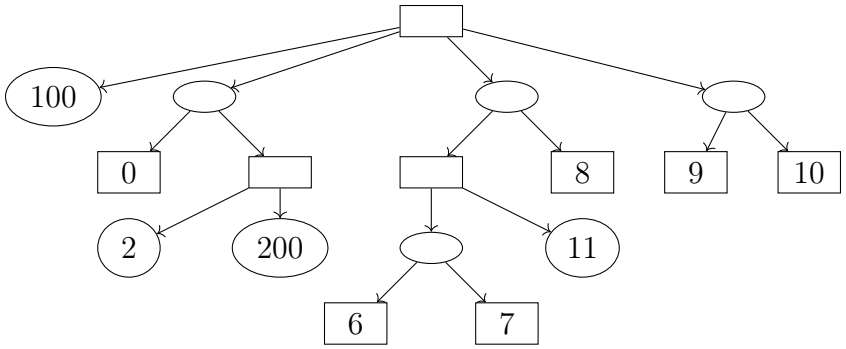
a) Ke všem hodnotám listů přičteme stejnou reálnou konstantu c.

b) Všechny hodnoty v listech vynásobíme stejnou konstantou c.

c) Hodnoty ve všech listech se libovolně změní tak, aby mezi nimi zůstalo zachováno jejich původní uspořádání.

d) Všechny uzly typu MIN změníme na MAX a naopak. Hodnoty v listech pronásobíme -1.

Příklad 4.2.4. Bez výpočtu určete a zdůvodněte, které podstromy by algoritmus ALFA-BETA v zadaném stromu ořezal. Následníci se prochází v pořadí zadaném obrázkem.

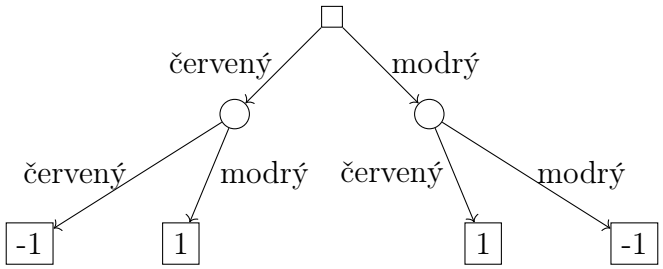


Příklad 4.2.5. Představme si MINIMAX strom, který se skládá pouze z uzlů typu MAX. Může při jeho vyhodnocování ALFA-BETA algoritmem dojít k nějakému prořezání?

4.3 Nedeterministické hry

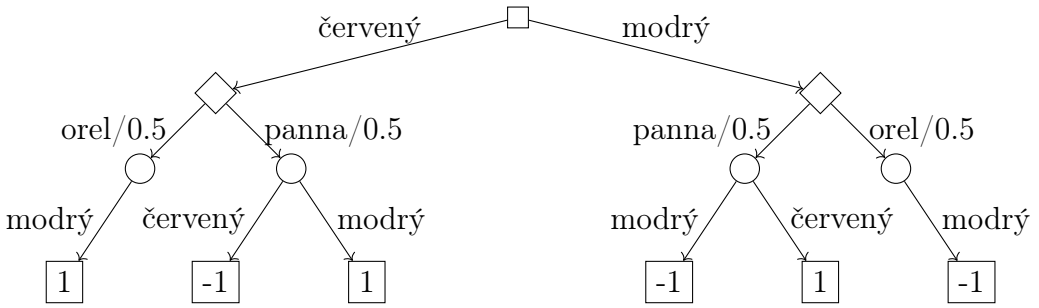
Až do této chvíle jsme tvrdě trvali na tom, aby uvažované hry neobsahovaly náhodu, neboť bychom ji neuměli modelovat. Nyní si ukážeme, že lze vhodně rozšířit MINIMAX stromy tak, aby náhodu v jisté míře postihly a my nad nimi pak mohli provádět smysluplné výpočty a předpovědi. Zjistíme však také, že některé vlastnosti a zákony, které dosud platily, se s tímto rozšířením zcela rozbíjí a mění.

Na začátek uvažme velmi triviální ilustrativní hru. V prvním kole si vezmu modrý, či červený žeton. V dalším kole provede to samé provede soupeř (může si zvolit i žeton stejné barvy). První hráč vítězí, drží-li jiný žeton než jeho soupeř, v opačném případě prohrává. Pro přehlednost přikládáme ještě příslušný MINIMAX strom této jednoduché hry.



Nyní naši hru obohatíme o náhodu. Před tím, než si druhý hráč zvolí svůj žeton, hodíme mincí. Padne-li orel, soupeř přijde o možnost volby – musí si vzít modrý žeton. V opačném případě se nic nemění a hra pokračuje jako předtím.

Pro modelování nedeterministické hry použijeme speciální vrchol (značený kosočtvercem $\diamond$) tam, kde dochází k rozhodnutí na základě náhody. Hrany, které z něj vychází, budou vždy označeny pravděpodobností, se kterou k výběru té konkrétní cesty dojde.



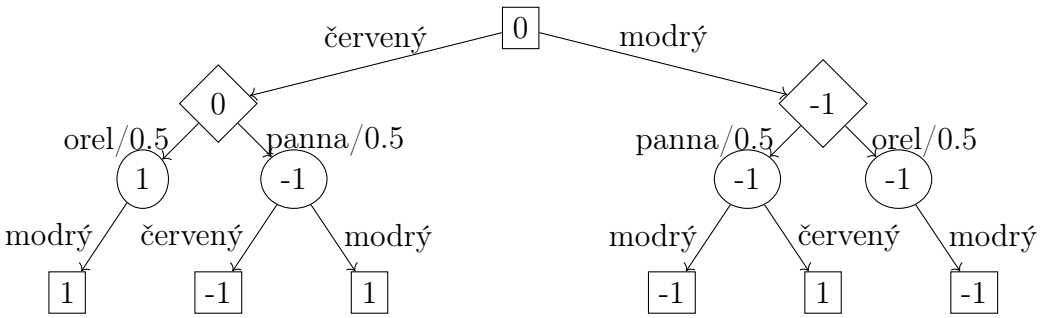
Už tedy chápeme, jak nedeterministické hry modelovat, zajímá nás však, jak se dopočítáme ke zjevnému závěru, že má začínající hráč volit červený žeton. K tomu využijeme upravený algoritmus MINIMAX, který rozšíříme o schopnost počítat hodnotu nového typu uzlu reprezentujícího náhodnou volbu $\diamond$.

Pro většinu aplikací je přirozené volit jako hodnotu vrcholu náhody jeho očekávanou hodnotu – tj. hodnotu každého potomka vynásobíme pravděpodobností, že bude vybrán, a výsledky pak sečteme napříč všemi potomky. Poněkud přehlednější zápis říká

$$\text{minimax(X)} = \sum_{\text{n...child of X}} \text{minimax(n)} \cdot \text{P(n)},$$

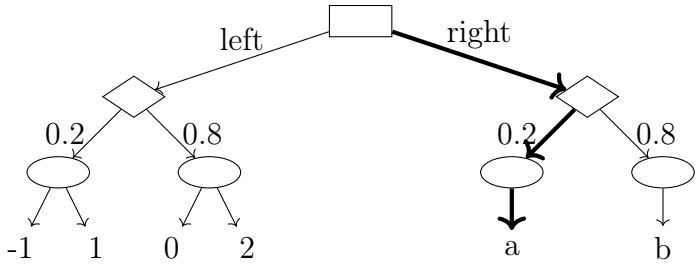
kde X je uzel reprezentující náhodu, $minimax(n)$ je spočtená hodnota potomka n a $P(n)$ je pravděpodobnost, že náhodně zvolíme právě potomka n – tj. číslo na odpovídající hraně grafu.

S touto znalostí již celý graf snadno doplníme. Je vidět, že očekávaný výnos začínajícího hráče je 0 a získá ho za předpokladu, že volí strategii *červený*.



Příklad 4.3.1. Sestavte vhodný strom a následně jej vyřešte pro následovně zadanou nedeterministickou hru dvou hráčů.
Začínající hráč si zvolí číslo 1, 2 nebo 3. Následně se hodí 3 mincemi. Padne-li na všech z nich stejná strana, soupeř si smí zvolit číslo -1 nebo 0. V opačném případě vybere číslo 1 či 2. Začínající hráč obdrží od soupeře částku rovnou součinu jejich čísel – v případě záporného součinu naopak musí příslušnou sumu protihráči zaplatit.

Příklad 4.3.2. Uvažme obrázkem zadaný MINIMAX strom rozšířený o vrcholy reprezentující náhodu. Hodnoty dvou listů jsou neznámé, označeny jako a a b.

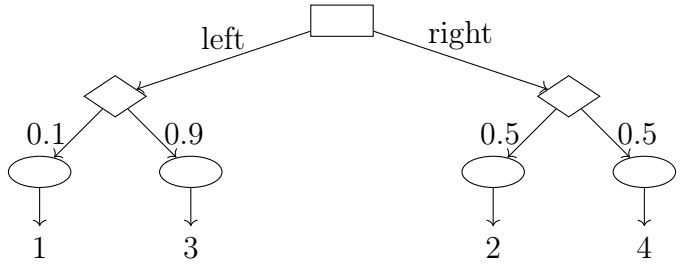


a) Co musí platit pro hodnoty a, b, aby optimální strategie odpovídala tučně vyznačené?

b) Nechť $b = 1$. Co musí platit pro hodnotu a, aby maximalizující hráč volil v prvním kole strategii *left*?

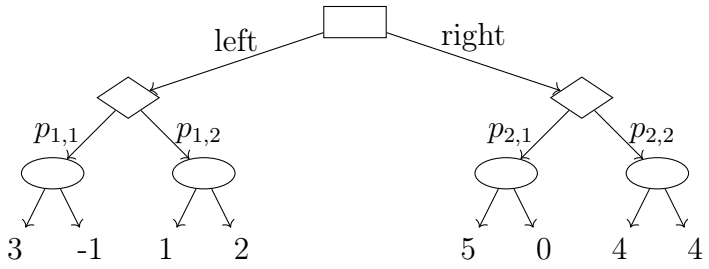
c) Nechť $b = -1$. Co musí platit pro hodnotu a, aby maximalizující hráč volil v prvním kole strategii *right*?

Příklad 4.3.3.



Nejprve pro obrázkem zadaný rozšířený MINIMAX strom ověřte, že optimální strategie začínajícího hráče je *right*.
Poté ukažte, že lze vhodně transformovat hodnoty listů tak, aby se optimální strategie změnila na *left*. Transformace musí zachovat původní uspořádání mezi listy; tedy jestliže měl list A před transformací menší hodnota než list B, bude tomu tak i po ní.

Příklad 4.3.4.
Uvažujeme rozšířený MINIMAX strom s neznámými pravděpodobnostmi větvení.



a) Nechť $p_{1,1} = \frac{2}{7}$. Co platí pro $p_{2,1}, p_{2,2}$, aby optimální strategie začínajícího hráče byla *right*?

b) Co pro neznámé pravděpodobnosti platí obecně, má-li být optimální strategie *right*?

c) Jak by musely vypadat hodnoty v listech, aby bez ohledu na hodnoty neznámých pravděpodobností byla optimální strategií *right*?

5 Výroková logika

Výroková logika je základní formalismus, kterým můžeme modelovat jednoduchá tvrzení, jejich pravdivost, vyplývání a jiné vztahy mezi nimi. Staví na pojmu *výrok*, což je elementární tvrzení, o jehož pravdivosti má smysl uvažovat. Výroky reprezentujeme výrokovými proměnnými, které s využitím logických spojek (a, nebo, pokud-pak atp.) můžeme skládat do složitějších formulí a tím modelovat složitější tvrzení (jako např. souvětí).

5.1 Syntax

Syntax vyjadřuje, jakým způbem ve výrokové logice zapisujeme platné formule – zatím však bez toho, abychom jim přiřazovali nějaký význam. První definice popisuje symboly, které používáme...

Definice 11: *Abeceda výrokové logiky* zahrnuje

- spočetně mnoho symbolů výrokových proměnných $p, q, r, \ldots,$
- symboly pro logické spojky $\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \ldots,$
- pomocné symboly závorek (a).

Ozn. $\mathcal{P} = \{p, q, r, \ldots\}$ množinu symbolů výrokových proměnných.

... další už nám přesně říká, jak vypadá správně utvořená formule ve výrokové logice.

Definice 12: *Formule výrokové logiky.*

- Každý symbol $p \in \mathcal{P}$ je formule.
- Jsou-li φ, ψ formule, pak rovněž $\neg(\varphi)$, $(\varphi) \vee (\psi)$, $(\varphi) \wedge (\psi)$, $(\varphi) \Rightarrow (\psi)$, $(\varphi) \Leftrightarrow (\psi)$, ... jsou formule.

Závorková konvence umožňuje vynechávat přebytečné závorky, nedojde-li k porušení sémantické jednoznačnosti formule.
Ozn. $\mathcal{F}$ množinu formulí výrokové logiky.

Poznáte nyní správně utvořenou formuli výrokové logiky?

Příklad 5.1.1. Uvažujte formule dle Definice 12. Rozhodněte, která z následujících slov jsou formule (neuvažujte závorkovou konvenci, tj. závorky musí přesně odpovídat definici):

a) $w_1 = \neg(\neg p)$

b) $w_2 = (p) \wedge (q) \wedge (r)$

c) $w_3 = (p) \Rightarrow ((\neg(q)) \vee (\neg(r)))$

d) $w_4 = ((p)\neg(q)) \vee (r)$

e) $w_5 = (p) = (r)$

5.2 Sémantika

Sémantika popisuje, jak máme logické formule chápat, přiřazuje celému formalismu konkrétní význam. První důležitý pojem, interpretace, vyjadřuje, zda konkrétní elementární výroky považujeme za pravdivé či nepravdivé.

Definice 13: *Interpretace* I je zobrazení $I : \mathcal{P} \to \{0, 1\}$ přiřazující pravdivostní hodnoty 0 (nepravda), 1 (pravda) jednotlivým výrokovým proměnným množiny $\mathcal{P}$.

V konkrétní interpretaci elementárních výroků pak můžeme vyhodnotit pravdivostní hodnotu celé formule.

Definice 14: *Valuace* (též *vyhodnocení*) příslušící interpretaci I je zobrazení $I : \mathcal{F} \to \{0, 1\}$ přiřazující pravdivostní hodnoty 0 (nepravda), 1 (pravda) jednotlivým formulím z $\mathcal{F}$.
(Jedná se o rozšíření interpretace z atomických formulí na všechny formule podle sémantiky logických spojek. Přesná definice je induktivní a zde ji neuvádíme.)

To s sebou přináší celou řadu pojmů, které blíže popisují vlastnosti dané formule, ať už obecně, či ve vztahu k dalším formulím.

Definice 15:

- Formule φ je *pravdivá v interpretaci* I, jestliže $I(\varphi) = 1$ (tj. vyhodnotí se v příslušné valuaci na hodnotu 1). V opačném případě je formule *nepravdivá v interpretaci* I.
- Formule φ je *logicky pravdivá* či *tautologie*, jestliže je pravdivá v libovolné interpretaci.
- Formule φ je *kontradikce*, jestliže je nepravdivá v libovolné interpretaci.
- Formule φ je *splnitelná*, jestliže je pravdivá v nějaké interpretaci. Tato splňující interpretace se nazývá *modelem* formule φ.
- Formule φ, ψ jsou *sémanticky ekvivalentní*, značeno $\varphi \approx \psi$, jestliže $I(\varphi) = I(\psi)$ pro libovolnou interpretaci I.

Podívejme se na vše zblízka na vzorovém příkladě.

Příklad. Mějme formuli výrokové logiky $\varphi \equiv p \wedge (q \Rightarrow \neg p)$. Sestavte pravdivostní tabulku formule φ a určete

a) zda je φ pravdivá v interpretaci $I(p) = 0$, $I(q) = 1$,

b) zda je φ kontradikcí či tautologií,

c) zda je φ splnitelná; případně nalezněte nějaký její model.

Pravdivostní tabulka formule vypadá následujícím způsobem.

ř.	p	q	p	$\wedge$	$(q$	$\Rightarrow$	$\neg p)$
1	0	0	0	**0**	0	1	1
2	0	1	0	**0**	1	1	1
3	1	0	1	**1**	0	1	0
4	1	1	1	**0**	1	0	0

Jednotlivé řádky odpovídají různým interpretacím (sloupečky p, q) a jim příslušícím valuacím jednotlivých částí (podformulí) formule φ (následující sloupečky). Sloupeček s tučně vyznačenými hodnotami odpovídá valuacím formule φ.

a) Formule φ je nepravdivá v interpretaci I, viz ř. 2.

b) Formule φ není ani kontradikcí, ani tautologií. (Všechny zvýrazněné hodnoty by musely být 0, resp. 1.)

c) Formule φ je splnitelná, neboť existuje interpretace, v níž je formule pravdivá. Viz ř. 3 tabulky. Modelem formule φ (jediným) je tedy interpretace $I(p) = 1$, $I(q) = 0$.

Příklad 5.2.1. Pro formuli výrokové logiky $\psi \equiv (r \Rightarrow p) \vee \neg(q \wedge r)$ sestavte pravdivostní tabulku a rozhodněte, zda formule

a) je logicky pravdivá,

b) je splnitelná,

c) je pravdivá v interpretaci I, kde $I(p) = I(q) = 0$, $I(r) = 1$,

d) je kontradikce či tautologie.

Nalezněte interpretaci I, kde $I(q) = 0$, takovou, že

e) ψ je pravdivá v I,

f) ψ je nepravdivá v I,

případně ukažte, že taková interpretace neexistuje.

Příklad 5.2.2. Bez použití pravdivostních tabulek rozhodněte, zda jsou následující formule tautologie.

a) $\varphi \equiv (\neg p \Rightarrow (q \wedge \neg q)) \Rightarrow p$

b) $\psi \equiv (p \Rightarrow p) \Rightarrow (p \wedge \neg(q \Rightarrow p))$

Příklad 5.2.3. Mějme formuli výrokové logiky $\varphi \equiv p \Rightarrow \neg(\neg q \vee r)$. Sestavte pravdivostní tabulku formule φ a určete

a) zda je φ pravdivá v interpretaci $I(p) = 0$, $I(q) = I(r) = 1$,

b) zda je φ kontradikcí či tautologií,

c) zda je φ splnitelná; případně nalezněte nějaký její model.

Příklad 5.2.4. ★ Mějme formuli výrokové logiky $\varphi \equiv (p \wedge q) \Leftrightarrow (\neg q \wedge r)$. *Bez použití pravdivostní tabulky* určete

a) zda je φ pravdivá v interpretaci $I(p) = 0$, $I(q) = I(r) = 1$,

b) zda je φ kontradikcí,

c) zda je φ tautologií,

d) zda je φ splnitelná; případně nalezněte nějaký její model.

Příklad 5.2.5. ★ Udejte příklad formule φ takové, že:

a) φ obsahuje právě 3 různé výrokové proměnné a je pravdivá právě ve 3 interpretacích,

b) φ obsahuje právě 3 různé výrokové proměnné, je pravdivá právě ve 3 interpretacích a obsahuje pouze logickou spojku $\Rightarrow$.

c) φ je kontradikce, obsahuje právě 2 různé výrokové proměnné, každou dvakrát.

(Uvažujte interpretaci jako zobrazení přiřazující hodnoty právě výrokovým proměnným vyskytujícím se v φ.)

Příklad 5.2.6. Zjistěte, kolik existuje vzájmeně neekvivalentních formulí výrokové logiky

a) obsahujících pouze 3 výrokové proměnné p_1, p_2, p_3 (i vícekrát),

b) obsahujících pouze n výrokových proměnných $p_1, \ldots, p_n$ (i vícekrát),

c) obsahujících pouze n výrokových proměnných $p_1, \ldots, p_n$ (i vícekrát) pravdivých právě v polovině interpretací.

5.3 Normální formy

Normální formy představují speciální způsob zápisu, z kterého lze přímo vyčíst některé sémantické vlastnosti formule. Základní stavební kameny formulí v normální formě jsou literály, znichž skládáme klauzule, resp. duální klauzule.

Definice 16:

- *Literál* je výroková proměnná nebo její negace.
- *Klauzule* je disjunkce literálů.
- *Duální klauzule* je konjunkce literálů.

Spojením klauzulí konjunkcí nebo duálních klauzulí disjunkcí získáme formuli v konjukntivní, resp. disjunktivní normální formě. Když navíc každá klauzule obsahuje všechny uvažované výrokové proměnné, bavíme se o úplné normální formě.

Definice 17: Uvažujme výrokové proměnné $p_1, \ldots, p_n$.

- Formule je v *konjunktivní normální formě (KNF)*, jedná-li se o konjunkci klauzulí (s navzájem různými množinami literálů).
- Formule je v *disjunktivní normální formě (DNF)*, jedná-li se o disjunkci duálních klauzulí (s navzájem různými množinami literálů).
- Obsahuje-li navíc každá klauzule (resp. duální klauzule) formule v KNF (resp. DNF) každou z výrokových proměnných $p_1, \ldots, p_n$ právě jednou, hovoříme o *úplné* konjunktivní (resp. disjunktivní) normální formě (ÚKNF, resp. ÚDNF)

Formule $(p \vee \neg q) \wedge (\neg p \vee \neg q \vee r)$ je v KNF, ale není v ÚKNF, protože první klauzule neobsahuje všechny implicitně uvažované výrokové proměnné p, q, r. Ekvivalentní formulí v ÚKNF je formule $(p \vee \neg q \vee r) \wedge (p \vee \neg q \vee \neg r) \wedge (\neg p \vee \neg q \vee r)$.

Pro převod do úplných normálních forem existuje poměrně přímočarý algoritmus, který je ilustrován na následujícím příkladu.

Příklad. Pomocí pravdivostní tabulky převeďte formuli $p \Leftrightarrow q$ do ÚDNF a ÚKNF.

Vytvoříme pravdivostní tabulku zadané formule a pro interpretace, v nichž je formule pravdivá, přidáme duální klauzuli do ÚDNF; pro interpretace, v nichž je formule nepravdivá, přidáme klauzuli do ÚKNF.
Klauzuli pro danou interpretaci I tvoříme tak, že pro každou výrokovou proměnnou p:

- je-li $I(p) = 0$, přidáme do klauzule literál p,
- je-li $I(p) = 1$, přidáme do klauzule literál $\neg p$.

V případě duální klauzule pro každou výrokovou proměnnou p, je-li $I(p) = 0$, přidáme literál $\neg p$, je-li $I(p) = 1$, přidáme literál p.

p	q	$p \Leftrightarrow q$	duální klauzule	klauzule
0	0	1	$(\neg p \wedge \neg q)$	
0	1	0		$(p \vee \neg q)$
1	0	0		$(\neg p \vee q)$
1	1	1	$(p \wedge q)$	

ÚDNF: $(\neg p \wedge \neg q) \vee (p \wedge q)$
ÚKNF: $(p \vee \neg q) \wedge (\neg p \vee q)$

Příklad 5.3.1. Pomocí pravdivostních tabulek nalezněte ÚDNF (úplnou disjunktivní normální formu) a ÚKNF (úplnou konjunktivní normální formu) formule $\psi \equiv (r \Rightarrow p) \vee \neg(q \wedge r)$ z Příkladu 5.2.1.

Příklad 5.3.2. Použitím ekvivalentních úprav nalezněte KNF následujících formulí.

a) ★ $\varphi \equiv (p \Rightarrow q) \Rightarrow r$

b) $\eta \equiv (p \Rightarrow q) \Leftrightarrow (p \Rightarrow r)$

c) $\psi \equiv \neg(p \Leftrightarrow \neg q) \vee \neg r$

Příklad 5.3.3. Použitím ekvivalentních úprav nalezněte DNF následujících formulí.

a) ★ $\psi \equiv (p \Leftrightarrow q) \Rightarrow (r \vee s)$

b) $\varphi \equiv (p \Rightarrow q) \Rightarrow r$

c) $\eta \equiv (p \vee \neg(\neg r \Rightarrow q)) \wedge (r \Rightarrow p)$

Příklad 5.3.4. Pomocí pravdivostní tabulky převeďte formuli $r \Rightarrow \neg s$ do ÚDNF.

Příklad 5.3.5. Pomocí pravdivostní tabulky převeďte formuli $\neg r \Rightarrow (r \wedge \neg s)$ do ÚKNF.

Příklad 5.3.6. Ekvivalentními úpravami převeďte formuli $(\neg q \wedge r) \vee (r \wedge \neg p)$ do ÚDNF.

5.4 Množiny formulí, splnitelnost, vyplývání

Občas je potřeba pracovat s několika tvrzeními zároveň a zkoumat vztahy mezi nimi. Proto zavádíme následující pojmy.

Definice 18: Množina formulí $\mathcal{T}$ je *splnitelná*, jestliže existuje interpretace I, v níž jsou pravdivé všechny formule $\varphi \in \mathcal{T}$. Potom říkáme, že množina $\mathcal{T}$ je *pravdivá v interpretaci I* a tato interpretace se nazývá *modelem* množiny $\mathcal{T}$. Není-li množina splnitelná, mluvíme o množině *nesplnitelné*.

Všimněte si, že prázdná množina $\varnothing$ je splnitelná a je pravdivá v libovolné interpretaci. Dále zavedeme pojem logického vyplývání.

Definice 19: Formule φ *logicky vyplývá* z množiny formulí $\mathcal{T}$, zapisováno $\mathcal{T} \vDash \varphi$, právě když je formule φ pravdivá v každém modelu množiny $\mathcal{T}$.

Jinak řečeno, formule (závěr) logicky vyplývá z množiny předpokladů, právě když je pravdivá v každé interpretaci, v níž je pravdivá i množina předpokladů.

Vyplývá-li tedy formule φ z prázdné množiny, $\varnothing \vDash \varphi$, jedná se o tautologii (rozmyslete si). Píšeme jenom $\vDash \varphi$. V podobném duchu vynecháváme množinové závorky, je-li množina předpokladů jednoprvková, tedy $\psi \vDash \varphi$ namísto $\{\psi\} \vDash \varphi$.

Následující věta dále propojuje logickou spojku implikace s logickým vyplýváním.

Věta 20 (o dedukci): Logické vyplývání $\{\psi_1, \ldots, \psi_n\} \vDash \varphi$ platí, právě když platí vyplývání $\{\psi_i, \ldots, \psi_{n-1}\} \vDash \psi_n \Rightarrow \varphi$.

Následující vzorový příklad ilustruje právě zavedené koncepty.

Příklad. Uvažujte množinu formulí $\mathcal{T} = \{\neg p \Rightarrow r, \neg(q \wedge r), p \vee \neg q\}$. Sestavte pravdivostní tabulku množiny $\mathcal{T}$ a určete

a) zda je $\mathcal{T}$ pravdivá v interpretaci $I(p) = I(q) = I(r) = 0$,

b) zda je $\mathcal{T}$ splnitelná; případně nalezněte nějaký její model,

c) zda platí logické vyplývání $\mathcal{T} \vDash p \wedge \neg r$,

d) zda platí logické vyplývání $\mathcal{T} \vDash r \Rightarrow \neg q$.

Pravdivostní tabulka množiny $\mathcal{T}$ vypadá následujícím způsobem.

ř.	p	q	r	$\neg p \Rightarrow r$	$\neg(q \wedge r)$	$p \vee \neg q$	$\mathcal{T}$	$p \wedge \neg r$	$r \Rightarrow \neg q$
1	0	0	0	0	1	1	0	0	1
2	0	0	1	1	1	1	1	0	1
3	0	1	0	0	1	0	0	0	1
4	0	1	1	1	0	0	0	0	0
5	1	0	0	1	1	1	1	1	1
6	1	0	1	1	1	1	1	0	1
7	1	1	0	1	1	1	1	1	1
8	1	1	1	1	0	1	0	0	0

a) Množina $\mathcal{T}$ není pravdivá v interpretaci I, viz ř. 1.

b) Množina $\mathcal{T}$ je splnitelná, neboť existuje interpretace, v níž je množina pravdivá. Viz např. ř. 2 tabulky. Modelem množiny $\mathcal{T}$ je tedy např. interpretace $I(p) = I(q) = 0$, $I(r) = 1$.

c) Vyplývání neplatí. Existuje interpretace, v níž je pravdivá množina $\mathcal{T}$ (předpoklad), ale neplatí závěr. Viz např. ř. 2.

d) Vyplývání platí. Ve všech interpretacích, v nichž je pravdivá množina $\mathcal{T}$ (předpoklad), je platný i závěr. Viz ř. 2, 5, 6, 7.

Příklad 5.4.1. Určete splnitelnost následujících množin formulí. Je-li množina splnitelná, nalezněte nějaký její model. Vyhněte se použití pravdivostních tabulek.

a) ★ $\mathcal{T}_1 = \{(p \Rightarrow q) \wedge r, q \wedge r, r \Rightarrow s, p \wedge \neg s\}$

b) $\mathcal{T}_2 = \{(p \vee q) \Leftrightarrow r, r, \neg p, q\}$

Příklad 5.4.2. ★ Je množina formulí $\mathcal{T} = \varnothing$ splnitelná? Dokažte.

Příklad 5.4.3. Ukažte, že pro každou množinu formulí $\mathcal{T}$ existuje formule φ, která je pravdivá v těch interpretacích I, které jsou modelem $\mathcal{T}$.

Příklad 5.4.4. Uvažujte množinu formulí $\mathcal{T} = \{r \wedge \neg q, \neg(\neg r \wedge p), p \Rightarrow r\}$. Sestavte pravdivostní tabulku množiny $\mathcal{T}$ a určete

a) zda je $\mathcal{T}$ pravdivá v interpretaci $I(p) = I(q) = I(r) = 1$,

b) zda je $\mathcal{T}$ splnitelná; případně nalezněte nějaký její model,

c) zda platí logické vyplývání $\mathcal{T} \vDash r$,

d) zda platí logické vyplývání $\mathcal{T} \vDash r \wedge p$.

Příklad 5.4.5. Uvažujte množinu formulí $\mathcal{T} = \{(p \Rightarrow q) \vee \neg r,\ \neg p \Rightarrow s,\ \neg t \vee p \vee \neg q\}$. Zadejte formuli φ tak, aby množina $\mathcal{T} \cup \{\varphi\}$ byla nesplnitelná.

Příklad 5.4.6. Rozhodněte, zda pro libovolnou neprázdnou množinu formulí $\mathcal{T}$ existuje formule φ taková, že φ není kontradikce a $\mathcal{T} \cup \{\varphi\}$ je nesplnitelná. Své tvrzení dokažte. (Argumentujte formálně s využitím pojmů interpretace, valuace atp.)

Příklad 5.4.7. ★ Uvažujte množinu formulí výrokové logiky $\mathcal{T}$ a formule φ, ψ. Rozhodněte o platnosti následujících tvzení:

a) Pokud $\mathcal{T} \vDash \varphi$ a $\mathcal{T} \vDash \psi$, pak $\mathcal{T} \vDash \varphi \vee \psi$.

b) Pokud $\mathcal{T} \vDash \varphi$ a $\mathcal{T} \vDash \psi$, pak $\mathcal{T} \vDash \varphi \wedge \psi$.

c) Pokud $\mathcal{T} \vDash \varphi \vee \psi$, pak $\mathcal{T} \vDash \varphi$ nebo $\mathcal{T} \vDash \psi$.

d) Pokud $\mathcal{T} \vDash \varphi \wedge \psi$, pak $\mathcal{T} \vDash \varphi$ a $\mathcal{T} \vDash \psi$.

e) Pokud $\mathcal{T} \vDash \varphi$, pak $\mathcal{T} \vDash \varphi \vee \psi$.

f) Pokud $\mathcal{T} \vDash \varphi$, pak $\mathcal{T} \vDash \varphi \wedge \psi$.

g) Pokud $\mathcal{T} \vDash \varphi$, pak $\mathcal{T} \cup \{\psi\} \vDash \varphi$.

h) Pokud $\mathcal{T} \cup \{\psi\} \vDash \varphi$, pak $\mathcal{T} \vDash \varphi$.

Příklad 5.4.8. ★ Ukažte, že logické vyplývání $\mathcal{T} \vDash \varphi$ platí, právě když množina $\mathcal{T} \cup \{\neg\varphi\}$ je nesplnitelná.
Poznámka: Tato vlastnost odpovídá principu důkazu sporem. Znova ji uvidíme v části zabývající se rezolucí, kde ukazujeme vyplývání pomocí nesplnitelnosti množiny formulí.

Příklad 5.4.9. S využitím věty o dedukci (a bez použití pravdivostních tabulek) rozhodněte, zda je formule $\xi \equiv p \Rightarrow (q \Rightarrow (\neg p \Rightarrow r))$ tautologie, kontratikce, splnitelná formule.

5.5 Logické spojky

Se základními logickými spojkami ($\vee$, $\wedge$, $\Rightarrow$, $\neg$, ...) jsme se již na intuitivní úrovni setkali. Podívejme se na ně ještě jednou zblízka.

Definice 21: Sémantika *n-ární logické spojky* $\square$ je dána funkcí $f_\square : \{0,1\}^n \to \{0,1\}$ následujícím způsobem: Valuace formule $\psi \equiv \square(\varphi_1, \ldots, \varphi_n)$ v interpretaci I je dána jako $I(\psi) = f_\square(I(\varphi_1), \ldots, I(\varphi_n))$, kde $I(\varphi_1), \ldots, I(\varphi_n)$ jsou valuace podformulí $\varphi_1, \ldots, \varphi_n$ příslušné interpretaci I.

Intuitivně, pro novou spojku můžeme zadat její sémantiku tabulkou (předepisující funkci $f_\square$) a formule s touto spojkou pak odpovídající způsobem vyhodnocovat. Všimněte si rovněž, že logické spojky nemusí být nutně unární (jako např. $\neg$) nebo binární (jako např. $\vee$), ale mohou mít libovolnou aritu.

Význam nové spojky ale můžeme definovat i ekvivalencí s jinou formulí. Lze tedy definovat např. nulární falsum (čili konstantní nepravdu) jako $\bot :\approx p \wedge \neg p$, či implikaci jako $\varphi \Rightarrow \psi :\approx \neg\varphi \vee \psi$.

Příklad. Definujte nulární verum $\top$ (čili konstantní pravdu) ekvivalencí s jinou formulí. Definujte disjunkci tabulkou.

Nulární verum: $\top :\approx p \vee \neg p$.
Disjunkce:

ř.	φ	ψ	$\varphi \vee \psi$
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	1

Ještě zavedeme pojem úplného systému logických spojek.

Definice 22: Množina logických spojek tvoří *úplný systém logických spojek*, pokud lze formulemi obsahujícími pouze spojky z dané množiny vyjádřit libovolnou logickou funkci (vnímáme-li formule jako funkce přiřazující zadané interpretaci svou valuaci).

Množina $\{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$ tvoří úplný systém logických spojek (důkaz vynecháváme). Ve skutečnosti, jak dále uvidíme, nám stačí pouze spojky $\{\neg, \Rightarrow\}$. Zavést ostatní spojky pomocí těchto dvou může v mnoha případech usnadnit argumentaci o formulích (stačí uvažovat o vlastnostech dvou spojek). V praxi (např. při tvorbě elektronických obvodů) se ještě používají úplné systémy logických spojek {NAND} a {NOR}[2], které si vystačí s jedinou spojkou.

Příklad 5.5.1. ★ Kolik existuje různých vzájemně neekvivalentních n-árních spojek?

Příklad 5.5.2. Uvažujte formule výrokové logiky nad abecedou obsahující pouze logické spojky $\Rightarrow, \neg$. Zaveďte spojky $\vee, \wedge, \Leftrightarrow$ jako syntaktické zkratky v tomto systému tak, aby měly obvyklý význam. (Tj. pro slova $\varphi \vee \psi, \varphi \wedge \psi, \varphi \Leftrightarrow \psi$, kde φ, ψ jsou správně utvořené formule, uveďte formule v uvažovaném systému, které jsou jimi reprezentovány.)

[2] $\varphi \mathrm{NAND} \psi :\approx \neg(\varphi \wedge \psi)$, $\varphi \mathrm{NOR} \psi :\approx \neg(\varphi \vee \psi)$

Příklad 5.5.3. Mějme formuli výrokové logiky $\varphi \equiv \neg(p \wedge (q \Box r)) \vee (q \Rightarrow r)$. Určete (např. tabulkou) sémantiku binární logické spojky $\Box$ tak, aby formule φ byla:

a) tautologie,

b) kontradikce,

c) ekvivalentní formuli $p \Rightarrow q$,

nebo konstatujte, že formule nemůže danou podmínku splnit. Odpovědi zdůvodněte.

Příklad 5.5.4. Kolika způsoby lze definovat sémantiku spojky $\Box$ z bodu a) předchozího příkladu?

Příklad 5.5.5. Ukažte, že následující množiny tvoří úplné systémy logických spojek. Vyjděte z předpokladu, že $\{\Rightarrow, \neg\}$ je úplný systém logických spojek. (Tj. vyjádřete formule $\varphi \Rightarrow \psi$ a $\neg\varphi$ pomocí formulí obsahující pouze spojky z příslušných množin.)

a) ★ {NOR}

b) {NAND}

Příklad 5.5.6. Uvažujte ternární logickou spojku Δ definovanou takto: $\Delta(\varphi, \psi, \theta) :\approx (\varphi \Rightarrow \psi) \wedge \neg\theta$. Rozhodněte, zda množina $\{\Delta\}$ tvoří úplný systém logických spojek, a svou odpověď dokažte. Můžete vyjít z předpokladu, že $\{\neg, \Rightarrow\}$ je úplný systém logických spojek.

Příklad 5.5.7. Uvažujte spojku $\not\Rightarrow$ definovanou takto: $\varphi \not\Rightarrow \psi :\approx \neg(\varphi \Rightarrow \psi)$. Rozhodněte, zda množina $\{\not\Rightarrow\}$ tvoří úplný systém logických spojek, a svou odpověď dokažte. Můžete vyjít z předpokladu, že $\{\neg, \Rightarrow\}$ je úplný systém logických spojek.