Bakalářské zkoušky (příklady otázek)

2024-09-03

1 Doplněk regulárního výrazu (společné okruhy)

Mějme následující regulární výraz nad abecedou $\Sigma = \{a, b\}$:

$$R = ((a + b)(a + b))^* ab$$

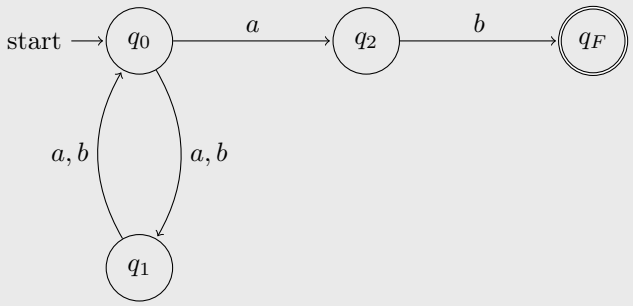
Označme jako L regulární jazyk popsaný výrazem R.

1. Sestrojte (co nejmenší) *nedeterministický* konečný automat A rozpoznávající jazyk L.
2. Pomocí podmnožinové konstrukce převeďte automat A na *deterministický* konečný automat B.
3. Z automatu B sestrojte *deterministický* konečný automat C rozpoznávající *doplněk* jazyka L.

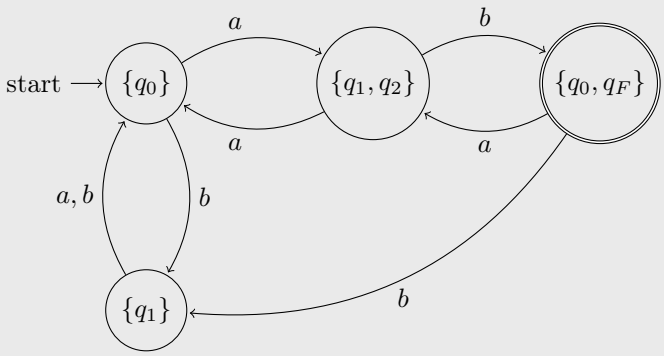
Sestrojené automaty A, B a C znázorněte pomocí stavových diagramů.

Nástin řešení

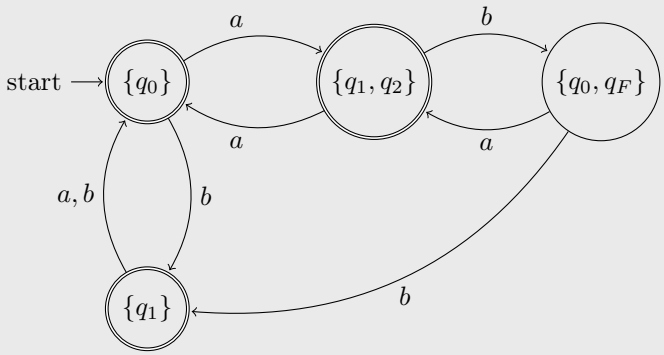
1. Nejmenší NFA má 4 stavy: $A = (\{q_0, q_1, q_2, q_F\}, \{a, b\}, \delta_A, q_0, \{q_F\})$, kde δ_A popíšeme stavovým diagramem:



2. Výsledkem podmnožinové konstrukce je následující DFA: $B = (\{\{q_0\}, \{q_1\}, \{q_1, q_2\}, \{q_0, q_F\}\}, \{a, b\}, \delta_B, q_0, \{\{q_0, q_F\}\})$, kde δ_B popíšeme stavovým diagramem:



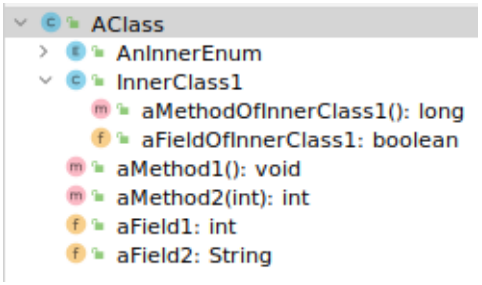
3. Stačí zaměnit přijímající a nepřijímající stavy:



(Všimněte si, že přechodová funkce automatu B je totální, proto není třeba přidávat FAIL stav.)

2 Reprezentace typů (společné okruhy)

1. Představte si, že vyvíjíte integrované vývojové prostředí (IDE). V okně projektu je potřeba zobrazit typy a elementy typů (tj. jejich metody, datové položky, vnitřní/vnořené typy – viz obrázek níže). Ve zvoleném jazyce (Java, C++, C#) navrhněte vhodnou sadu tříd/interfaců, které vám umožní uchovávat informace o typech tak, aby mohly být zobrazeny v okně v IDE. Informace zahrnují alespoň názvy typů, jejich druhy, metody typů a jejich signatury, datové položky a vnitřní/vnořené typy. Navrhněte vaše řešení tak, aby bylo v budoucnu snadno rozšiřitelné. Vytvořte pouze deklarace tříd/interfaců, tj. bez těl jejich metod.



2. Implementujte metodu, která bere jako parametry (i) typ (jak jste jej definovali pro část otázky výše) a (ii) „operaci“. Metoda aplikuje danou operaci na daný typ a všechny elementy typu. Pokud založíte procházení struktury typu na nějakém obecně známém řešení, explicitně toto řešení pojmenujte. Pro operaci použijte vhodný koncept, který nabízí vámi zvolený jazyk.
3. Napište kód, který zavolá vaší metodu z předchozího bodu. Operace předaná metodě je: „vytiskne název elementu, ale pouze v případě, že element reprezentuje datovou položku nebo typ (včetně vnitřních/vnořených typů)“.

Nástin řešení

1. Existuje více správných řešení. Typické řešení je obvyklá hierarchie tříd, např. abstraktní třída Element, a dále pak třídy, které od Element dědí, tj. třída Type, třída Method, třída Field, atd.
2. Existuje více správných řešení. Průchod strukturou může být založen např. na vzoru „Visitor“, nebo mohou být typy deklarovány jako „sealed“ (pokud to jazyk umožňuje) a průchod používá pattern matching, atd.

   Pro reprezentaci operace lze použít interface nebo funkcionální typ apod.
3. V závislosti na reprezentaci bude operace definována jako anonymní třída, lambda výraz atd.

3 Integrály a primitivní funkce (společné okruhy)

1. Definujte, co je „primitivní funkce“ k dané funkci f na daném otevřeném intervalu I.
2. Zformulujte pravidlo “per partes” (též známé jako integrování po částech) pro výpočet primitivní funkce. Můžete se pro jednoduchost omezit na situaci, kdy všechny uvažované funkce jsou spojité.

3. Nechť $P \subseteq \mathbb{R}^2$ je oblast roviny ohraničená zdola osou x a shora grafem funkce $f(x) = x\cos(x)$ pro $x \in [0, \pi/2]$. Formálněji řečeno,

$$P = \left\{(x, y) \in \mathbb{R}^2;\ 0 \leq x \leq \frac{\pi}{2} \ \wedge\ 0 \leq y \leq x\cos(x)\right\}.$$

Spočítejte plošný obsah oblasti P.

Nástin řešení

1. Funkce F je primitivní funkce k funkci f na intervalu I, pokud má F v každém bodě $x \in I$ derivaci rovnou $f(x)$.
2. Pravidlo pro integrování per partes říká, že pokud na nějakém intervalu I jsou f a g spojité funkce, F je primitivní funkce k f a G je primitivní funkce ke g, pak na tomto intervalu platí

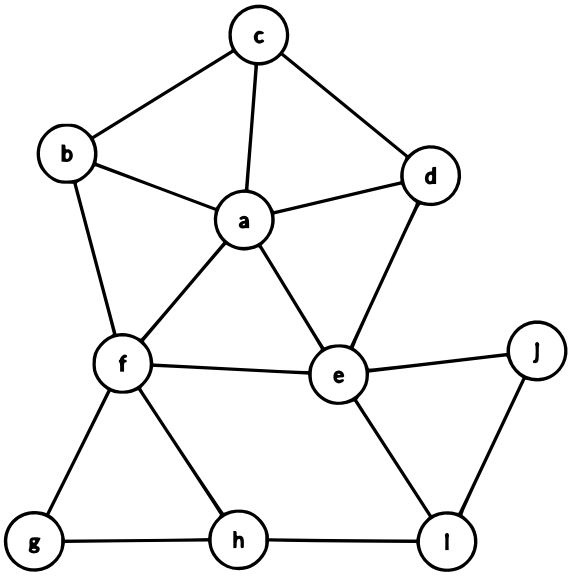
$$\int f(x)G(x)\mathrm{d}x = F(x)G(x) - \int F(x)g(x)\mathrm{d}x.$$

3. Hledaný plošný obsah lze vyjádřit pomocí integrálu jako $\int_0^{\pi/2} x\cos(x)\mathrm{d}x$. Hodnotu integrálu určíme použitím metody per partes pro výpočet určitého integrálu:

$$\begin{aligned}\int_0^{\pi/2} x\cos(x)\mathrm{d}x &= [x\sin(x)]_0^{\pi/2} - \int_0^{\pi/2} \sin(x)\mathrm{d}x \\ &= \left(\frac{\pi}{2} - 0\right) - [-\cos(x)]_0^{\pi/2} \\ &= \frac{\pi}{2} - 1.\end{aligned}$$

4 Barevnost grafů (společné okruhy)

1. Definujte, co je barevnost grafu.
2. Jaké platí omezení pro barevnosti rovinných grafů?
3. Určete barevnost grafu na obrázku níže.



Nástin řešení

– Barevnost grafu je minimální počet barev, kterými lze obarvit vrcholy grafu tak, aby žádné dva sousední vrcholy neměly stejnou barvu. Formálně, dobré obarvení grafu G k barvami je funkce $f : V \to \{1, \ldots, k\}$, kde V je množina vrcholů

grafu, splňující $f(u) \neq f(v)$ pro každé dva sousední vrcholy $u, v \in V$, a barevnost grafu $\chi(G)$ je nejmenší k, pro které existuje dobré obarvení G k barvami.

- Pro rovinné grafy platí, že jejich barevnost je nejvýše 4 a tento odhad je těsný.
- Barevnost grafu na obrázku je 4. Horní odhad lze nahlédnout buď z věty výše (graf je rovinný), nebo konstruktivně: pro vrcholy s písmeny $a, b, \ldots, j$ lze použít barvy v pořadí $4, 3, 2, 1, 2, 1, 4, 3, 4, 3$. Dolní odhad vyplývá z toho, že graf na obrázku má jako podgraf 5-cyklus s připojeným univerzálním vrcholem a protože 5-cyklus má barevnost 3, musí mít graf na obrázku barevnost alespoň 4.

5 Jazyk SQL (specializace WDOP)

Uvažujte doménu hudební scény.

1. Napište v jazyce SQL příkazy, které vytvoří tabulky pro reprezentaci zpěváků, kapel a skladeb. Vytvořte alespoň dva atributy pro každou z těchto entit (např. jméno a země původu zpěváka) a použijte vhodné datové typy.
2. Upravte vhodnými SQL příkazy tabulky tak, aby bylo možné ukládat informace o členství zpěváků v kapelách a o interpretech skladeb. Předpokládejte, že každý zpěvák je členem maximálně jedné kapely a každá skladba může mít maximálně jednoho interpreta. Kapela ale může mít více zpěváků a zpěvák může interpretovat více skladeb. Zajistěte referenční integritu.
3. Napište SQL příkaz, který vloží do tabulek zpěváka, který je členem nějaké kapely a interpretuje nějakou skladbu.
4. Napište v SQL dotaz, který vypíše jména všech zpěváků, kteří nejsou členy žádné kapely.
5. Vysvětlete, co dělá následující SQL dotaz (obecně i nad aktuálně vloženými daty):

```
SELECT nazev, COUNT(*)
FROM kapela
GROUP BY nazev
HAVING COUNT(*) > 2;
```

6. Napište SQL příkazy, které smažou vytvořené tabulky.

Nástin řešení 1.

```
CREATE TABLE zpevak (id_z INT PRIMARY KEY, jmeno VARCHAR(100), zeme VARCHAR(200));
CREATE TABLE kapela (id_k INT PRIMARY KEY, nazev VARCHAR(100), zalozena INT);
CREATE TABLE skladba (id_s INT PRIMARY KEY, titul VARCHAR(100), delka TIME);
```

2.

```
ALTER TABLE zpevak ADD COLUMN clen INT;
ALTER TABLE zpevak ADD FOREIGN KEY (clen) REFERENCES kapela (id_k);
ALTER TABLE skladba ADD COLUMN interpret INT;
ALTER TABLE skladba ADD FOREIGN KEY (interpret) REFERENCES zpevak (id_z);
```

3.

```
INSERT INTO kapela VALUES (1, 'Queen', 1970);
INSERT INTO zpevak VALUES (101, 'Freddie Mercury', 'Zanzibar', 1);
INSERT INTO skladba VALUES (201, 'Under Pressure', '00:04:08', 101);
```

(Pozor na pořadí nebo ADD FOREIGN KEY až po vložení všech dat.)

4.

```
SELECT jmeno FROM zpevak z
WHERE NOT EXISTS(SELECT * FROM skladba WHERE interpret = z.id_z);
```

5. Vrací název a počet členů kapel, které mají více než dva zpěváky. Pro daná data nevrátí nic.

6.

```
DROP TABLE skladba;
DROP TABLE zpevak;
DROP TABLE kapela;
```

(Pozor na pořadí nebo nejdřív DROP CONSTRAINT.)

6 Moderní databázové systémy (specializace WDOP)

Uvažujte problematiku multi-modelových dat a databází.

1. Vysvětlete pojem „multi-modelová data“ a rozdíl mezi multi-model a single-model databázemi. Uveďte příklad.
2. Uveďte alespoň dvě výhody a dvě nevýhody multi-modelových databází oproti tradičním single-model databázím.
3. Pro libovolnou multi-modelovou databázi vytvořte jednoduchý příklad databázového schématu, resp. dat a nad nimi multi-modelový dotaz. Demonstrujte na něm uvedené výhody multi-modelových databází.
4. Vysvětlete rozdíl mezi multi-modelovou databází a polystorem.

Nástin řešení 1. Multi-modelová data jsou data, která kombinují více než jeden model. Multi-modelové databáze jsou databáze, které takovou kombinaci nativně podporují. Např. PostgreSQL kombinuje relační a dokumentový model pomocí embeddingu, tj. sloupec relační tabulky může být typu JSON/JSONB a obsahovat dokumenty. Single-model databáze podporují jen jeden model.

2. Výhody: data jsou uložena nativně pomocí optimálního modelu, jsou uložena v jediném systému (ne v několika single-model systémech), existuje k nim jediný společný dotazovací jazyk, ...
Nevýhody: neexistuje žádný standard pro kombinaci modelů (jaké, jak), standardní dotazovací jazyk (kromě SQL/XML a SQL/JSON pro relační/dokumentová data) apod., implementace multi-model systému je složitá (modely mají z principu protichůdné vlastnosti), existující systémy mají stále mnoho omezení, ...

3. Např. v PostgreSQL vytvoříme klasickou relační tabulku se sloupcem pro dokumentová data a dotážeme se na ně:

```
CREATE TABLE zakaznik (
  id      INTEGER PRIMARY KEY,
  jmeno    VARCHAR(50),
  objednavky  JSONB
);
INSERT INTO zakaznik
VALUES (1, 'Karel Novák',
'{"Cislo_obj":"o43",
   "Polozky":[
    {"Id_produktu":"p34", "Jmeno_produktu":" Myš", "Cena":230, "Pocet":2},
    {"Id_produktu":"p56", "Jmeno_produktu":"Klávesnice", "Cena":788}]
  }');
INSERT INTO zakaznik
VALUES (2, 'Petr Novák',
'{"Cislo_obj":"o33",
   "Polozky":[
    { "Id_produktu":"p777", "Jmeno_produktu":"Počítač", "Cena":34000 }]
  }');

SELECT jmeno,
       objednavky->>'Cislo_obj' as cislo_obj,
       objednavky#>'{Polozky,0}'->>'Jmeno_produktu' as Jmeno_produktu
FROM zakaznik
WHERE objednavky->>'Cislo_obj' <> 'o33';
```

Jak je vidět, tak nám stačí jeden systém, ale rozšíření jazyka SQL o konstrukty pro JSON data je systémově specifické, další modely nemusíme mít podporované atd.

4. Polystore stejně jako multi-model databáze ukládá multi-modelová data, ale pro jednotlivé modely využívá samostatné single-model databáze. Centrální prvek (mediator) pak zajišťuje jejich management, dekompozici a vyhodnocení dotazu, spojení mezivýsledků atd. Hlavní výhodou je využití robustních prověřených single-model systémů, nevýhodou je komplexita mediatoru, malá podpora v komerčním světě (spíš akademické systémy).

7 Web (specializace WDOP)

1. Vysvětlete návrhový vzor Front Controller a jeho využití pro vývoj webových aplikací. S pomocí pseudokódu nebo PHP demonstrujte základní myšlenku implementace.

2. Uvažujte následující webové (REST) API:

- `GET /api/singer` – vrátí JSON, který je polem identifikátorů zpěváků, například:

  ```
  ["jaroslav-1980", "pavelka-1958"]
  ```

- `GET /api/singer/{identifikátor}` – vrátí JSON, který obsahuje jméno zpěváka, například:

  ```
  { "name": "Jaroslav" }
  ```

- `DELETE /api/singer/{identifikátor}` – smaže zpěváka

Pro popsané API napište v JavaScriptu, který poběží ve webovém prohlížeči, funkci, která dostane jako argument jméno zpěváka a provede následující kroky:

1. smaže všechny zpěváky daného jména,
2. pro každého smazaného zpěváka přidá HTML element `li` se jménem zpěváka do elementu s id `deleted-list`.

Pokud si nejste jisti názvem nějaké funkce/proměnné, popište její chování. Výsledný kód může být strukturován do více funkcí, na drobné syntaktické chyby nebude při hodnocení brán zřetel. V kódu nemusíte řešit obsluhu chybových stavů.

Nástin řešení 1. Základní myšlenkou je mít jedno vstupní místo do aplikace pro obsloužení požadavků od klienta. Návrhový vzor je použitý na straně serveru.

V zásadě stačí switch, nebo třída podobného stylu. Skládá ze dvou částí – routing a dispatching, tedy například:

```
// Routing.
$handler = routing($_GET, $_SERVER);
// Necháme metodu obsloužit - dispatching.
$handler->handle($_GET, $_POST, $_SERVER);
```

Dále je možné uvést například inicializaci a ošetření chybového stavu.

2. Implementace by měla obsahovat fetch na seznam a pak na jednotlivé zpěváky. Následně fetch, který provede delete. Poslední je vytvoření a vložení HTML elementu.

Například:

```
async function deleteSingersByName(name) {
  const identifiers = await fetchJson("./api/singer");;
  for (const identifier of identifiers) {
    const url = "./api/singer/" + encodeURI(identifier);
    const singer = await fetchJson(url);
    if (singer.name !== name) {
      continue;
    }
    await fetch(url, {method: "DELETE"});
    addToList(singer.name);
  }
}
```

```
async function fetchJson(url) {
  return await (await fetch(url)).json();
}

function addToList(name) {
  const li = document.createElement("li");
  li.innerText = name;
  document.getElementById("deleted-list").appendChild(li);
}
```

8 Základy indexování (specializace WDOP)

Pro potřeby této otázky uvažujme databázi vytvořenou v první otázce specializace („Jazyk SQL").

1. Vysvětlete přímé/nepřímé indexování a primární/sekundární index.

2. Uveďte příklady možného použití přímého primárního, přímého sekundárního a nepřímého sekundárního indexu pro tabulku reprezentující zpěváky (zpevak). Jaké jsou výhody a nevýhody použití indexů?

3. Předpokládejme přímé primární indexování sloupce zpevak.id (identifikátor zpěváka) pomocí redundantního B-stromu bez odloženého štěpení, stupně 3, tedy 2 hodnoty a 3 ukazatele. Demonstrujte postupné vkládání záznamů s hodnotou klíče: 1, 2, 3, 4, 5, 0.

Nástin řešení 1. Přímé indexování ukazuje přímo na data, nepřímé indexování ukazuje na jiný index. Primární index odpovídá fyzickému pořadí uložení záznamů, sekundární index mu neodpovídá.

2. Přímý primární – primární klíč, zpevak.id_z. Takový klíč může být jen jeden. Sekundární indexy je možné použít na všechny ostatní sloupce, kde očekáváme časté dotazování. Přímá a nepřímá varianta je zde jen implementačním detailem.

Výhodou indexu je rychlejší dotazování, nevýhodou pak pomalejší vkládání a mazání, kdy je třeba index upravit.

3. Postupně, první 4 jsou zapsány na jeden řádek, jedná se vždy o kořen a jeho potomky. Úrovně jsou oddělené čárkou.

1:

[1]

2:

[1,2]

3: Tady je možné [2], [1] [2,3] i [2], [1,2] [3] – záleží na tom, na jakou stranu uvažujeme ostrou nerovnost, jestli '<,>=' ,nebo '<=,>'. Důležité je, že je třeba být v tomto konzistentní i ve zbytku řešení.

4:

'<,>=': [2,3], [1] [2] [3,4]

'<=,>': [2], [1,2] [3,4]

5:

'<,>=': [3], [2] [4], [1] [2] [3] [4,5]

'<=,>': [2,4], [1,2] [3,4] [5]

0:

'<,>=': [3], [2] [4], [0,1] [2] [3] [4,5]

'<=,>': [2], [1] [4], [0,1] [2] [3,4] [5]

9 Homogenní podgrafy (specializace OI-G-PADS, OI-G-PDM, OI-O-PADS, OI-PADS-PDM)

1. Zformulujte Ramseyovu větu pro dvojice, s libovolným konečným počtem barev, ve verzi pro nekonečné grafy.
2. Platí následující tvrzení? Nechť $G = (A, B, E)$ je nekonečný bipartitní graf, tj. A a B jsou nekonečné množiny a E je podmnožina $A \times B$. Pak existují nekonečné podmnožiny $A_1 \subseteq A$ a $B_1 \subseteq B$ takové, že G buď obsahuje všechny hrany mezi A_1 a B_1, nebo neobsahuje žádnou takovou hranu.

Nástin řešení

1. Nechť p je přirozené číslo a nechť G je nekonečná klika s hranami obarvenými barvami $1, \ldots, p$. Pak existuje nekonečná podmnožina $K \subseteq V(G)$ taková, že všechny hrany G mezi vrcholy K mají stejnou barvu.
2. Toto tvrzení neplatí. Například nechť $A = \{a_1, a_2, \ldots\}$, $B = \{b_1, b_2, \ldots\}$ a $E = \{(a_i \in A, b_j \in B) : i < j\}$. Když $A_1 \subseteq A$ a $B_1 \subseteq B$ jsou nekonečné množiny, pak nutně existují indexy $i_1 < i_2 < i_3$ takové, že $a_{i_1}, a_{i_3} \in A_1$ a $b_{i_2} \in B_1$, a tedy $a_{i_1}b_{i_2}$ je hrana a $a_{i_3}b_{i_2}$ je nehrana.

10 Optimalizace (specializace OI-O-PADS)

1. Definujte pojem „totálně unimodulární matice“.
2. Uvažte následující variantu řezového problému na stromech: Je dán strom $T = (V, E)$ zakořeněný ve vrcholu $r \in V$ spolu s k dvojicemi jeho vrcholů $(s_1, t_1), \ldots, (s_k, t_k)$ takovými, že pro každé $i \in \{1, \ldots, k\}$ leží vrchol s_i na cestě mezi r a t_i. Úkolem je najít podmnožinu hran $F \subseteq E$ minimální velikosti takovou, že v grafu $(V, E \setminus F)$ pro žádné $i \in \{1, \ldots, k\}$ neexistuje cesta mezi s_i a t_i. Formulujte tento problém jako úlohu celočíselného lineárního programování, ve kterém jediné proměnné jsou 0/1-proměnné pro hrany.
3. Popište, jak problém z předešlého bodu optimálně vyřešit v polynomiálním čase. Můžete využít následující lemma:

   Lemma 1. Pro instanci řezového problému z předešlého bodu (tj. strom $T = (V, E)$ a k dvojic $(s_1, t_1), \ldots, (s_k, t_k)$ výše popsaných vlasností), nechť $M \in \{0, 1\}^{k \times E}$ je matice daná následujícím předpisem: $M_{i,e} = 1$ právě když hrana e leží na cestě mezi s_i a t_i. Pak matice M je totálně unimodulární.

Nástin řešení

1. Definice: Matice M je totálně unimodulární, pokud pro každou její čtvercovou podmatici A platí $\det(A) \in \{1, -1, 0\}$.
2. Označme x_e proměnnou odpovídající hraně $e \in E$. Dále pro každé $i \in \{1, \ldots, k\}$ označme p_i množinu hran na cestě mezi vrcholy s_i a t_i.

$$\begin{aligned} \min & \sum_{e \in E} x_e & & \\ & \sum_{e \in p_i} x_e \geq 1 & & \forall i \in \{1, \ldots, k\} \\ & x_e \in \{0, 1\} & & \forall e \in E \end{aligned}$$

3. Všimneme si, že k podmínek $\sum_{e \in p_i} x_e \geq 1$ z předešlého celočíselného LP je možné s využitím matice M z Lemma 1 zapsat jako

$$M\mathbf{x} \geq \mathbf{1} \ .$$

   Protože matice M je podle Lemma 1 totálně unimodulární, platí, že každý vrchol polyhedru

$$P = \{\mathbf{x} \in \mathbf{R}^E \mid M\mathbf{x} \geq \mathbf{1},\ \mathbf{x} \geq \mathbf{0}\}$$

   je celočíselný. Řešení (celočíselné) dané řezové úlohy lze proto získat jako řešení úlohy lineárního programování $\min \mathbf{1}\mathbf{x}, \mathbf{x} \in P$, což je možné v polynomiálním čase.

11 Mnohostěny (specializace OI-G-PADS, OI-G-PDM)

1. Spočítejte počet vrcholů a hran mnohostěnu v $\mathbb{R}^d$ duálního k hyperkrychli $[-1,1]^d$.
2. Kolik nejvýše vrcholů může mít konvexní mnohostěn v $\mathbb{R}^6$, který je průnikem 11 uzavřených poloprostorů? Zdůvodněte. (Případně uveďte alespoň asymptotický odhad pro n podprostorů, který však nebude hodnocen plným počtem bodů.)

Nástin řešení

1. $2d$ vrcholů a $2d(d-1)$ hran. Při výpočtu lze např. využít znalosti toho, že duálním mnohostěněm k hyperkrychli je křížový mnohostěn, nebo vztahu mezi svazy stěn mnohostěnu a jeho duálu.
2. Z duality stačí spočítat počet faset cyklického mnohostěnu s 11 vrcholy, např. podle Galeova kritéria. Výsledek $\binom{n-3}{3} + \binom{n-4}{2} = \binom{8}{3} + \binom{7}{2} = 56 + 21 = 77$. Asymptoticky $\Theta(n^3)$.

12 Mohutnost množin (specializace OI-G-PDM, OI-PADS-PDM)

1. Napište definice pojmů „množina A má ostře menší mohutnost než množina B“ a „množina A je spočetná“. Ke každému z pojmů uveďte jeden příklad množiny (množin), které danou relaci splňují, a jeden, který ji nesplňuje (stačí bez zdůvodnění).
2. Uvažme tvrzení „neexistuje nespočetná množina ostře menší mohutnosti, než reálná čísla“, tj. takzvanou hypotézu kontinua. Co můžete říct o platnosti tohoto tvrzení? Svoji odpověď vysvětlete z hlediska existence bezesporných rozšíření a modelů Zermelovy–Fraenkelovy teorie množin.

Nástin řešení

1. Množina A má ostře menší mohutnost než množina B, jestliže existuje prosté zobrazení (injekce) z A do B, ale ne z B do A. Například množina přirozených čísel má ostře menší mohutnost než množina reálných čísel, ale pouze neostře menší než množina celých čísel.

   Množina A je spočetná, je-li konečná nebo má stejnou mohutnost, jako množina přirozených čísel. Například množina celých čísel je spočetná, ale množina reálných čísel není.
2. Hypotéza kontinua je nerozhodnutelná v Zermelovy–Fraenkelově teorii množin, tuto teorii tedy lze rozšířit o axiom postulující buď platnost nebo neplatnost hypotézy kontinua a výsledná teorie je bezesporná (samozřejmě za předpokladu, že samotná Zermelova–Fraenkelova teorie je bezesporná). Proto existují modely Zermelovy–Fraenkelovy teorie množin, v nichž je hypotéza kontinua pravdivá, a modely, ve kterých neplatí.

13 Dynamické programování (specializace OI-G-PADS, OI-O-PADS, OI-PADS-PDM)

Uvažujme následující problém: Je dána posloupnost čísel $a_1, \ldots, a_n \in \{1, \ldots, m\}$ a číslo $s \in \mathbb{N}$. Chceme najít vybranou podposloupnost $a_{i_1}, \ldots, a_{i_k}$ (kde $i_1 < \ldots < i_k$), jejíž součet je roven s. Budeme jí říkat *poklad.*

1. Navrhněte algoritmus, který rozhodne, zda poklad existuje.
2. Algoritmus upravte, aby jeden z pokladů vypsal.
3. Nechť ke každému číslu a_i přiřadíme cenu $c_i \in \mathbb{N}$. Algoritmus upravte, aby našel nejlevnější poklad, tedy takový, který má nejnižší součet cen prvků. Stačí vypsat optimální cenu.

Rozeberte časovou a prostorou složitost všech tří algoritmů. Složitosti by měly být polynomiální vzhledem k n a m.

Nástin řešení

1. Budeme počítat tabulku $X_{ij} \in \{0,1\}$ pro $0 \le i \le n$, $0 \le j \le s$. Hodnota $X_{ij} = 1$ říkat, že je z $a_1, \ldots, a_i$ možné vybrat podposloupnost se součtem j. Tabulku budeme vyplňovat v pořadí rostoucích i.

1. Nejprve položíme $X_{00} \leftarrow 1$ a $X_{0j} \leftarrow 0$ pro všechna $j > 0$.
2. Pro $i = 1, \ldots, n$:
3. Pro $j = 0, \ldots, s$:
4. Je-li $j \geq a_i$ a $X_{i-1,j-a_i} = 1$, položíme $X_{ij} \leftarrow 1$.
5. Jinak položíme $X_{ij} \leftarrow X_{i-1,j}$.
6. Otestujeme, zda $X_{ns} = 1$.

Případy v krocích 4 a 5 odpovídají tomu, že nový prvek a_i v podposloupnosti buď použijeme, anebo nepoužijeme.

2. Kdykoliv $X_{ij} = 1$, zapamatujeme si navíc P_{ij}, což bude číslo posledního prvku, který jsme do podposloupnosti přidali. Stačí tedy v kroku 4 nastavovat $P_{ij} \leftarrow i$ a v kroku 5 $P_{ij} \leftarrow P_{i-1,j}$. V posledním kroku pak rekonstruujeme nalezený poklad od posledního přidaného prvku k prvnímu.
3. Předefinujeme X_{ij} na minimální cenu podposloupnosti z $a_1, \ldots, a_i$, která má součet j; nebo ∞, pokud taková podposloupnost neexistuje. Budeme inicializovat $X_{00} \leftarrow 0$ a $X_{0j} \leftarrow \infty$ pro všechna $j > 0$. Kroky 4 a 5 nahradíme $X_{ij} \leftarrow \min(X_{i-1,j-a_i} + c_i, X_{i-1,j})$. Na konci vypíšeme X_{ns}.

Časová i prostorová složitost všech tří algoritmů činí $\Theta(ns)$. Pokud nepotřebujeme vypisovat nalezený poklad, stačí si z matice pamatovat aktuální a předchozí řádek, čímž prostor redukujeme na $\Theta(s)$ při zachování času.

14 Uzavřené množiny v metrických prostorech (specializace OI-G-PADS, OI-G-PDM, OI-O-PADS, OI-PADS-PDM)

1. Nechť (M, d) je metrický prostor, tj. M je množina a d je metrika na množině M. Napište, jak je definována otevřená množina a jak je definována uzavřená množina v prostoru (M, d).
2. Uvažujme množinu reálných čísel $\mathbb{R}$ spolu s její obvyklou metrikou d danou předpisem $d(x, y) = |x - y|$. U každé z následujících množin rozhodněte (a stručně zdůvodněte), zda je uzavřená v metrickém prostoru $(\mathbb{R}, d)$:
   (a) množina $X_a = [0, +\infty)$
   (b) množina $X_b = \{\frac{1}{n}; n \in \mathbb{N}\} = \{1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots\}$
3. Uvažujme opět reálná čísla $\mathbb{R}$ s obvyklou metrikou. Nechť f je funkce z $\mathbb{R}$ do $\mathbb{R}$. Řekneme, že číslo $x \in \mathbb{R}$ je *nulový bod* funkce f, pokud platí $f(x) = 0$. Dokažte, že pokud je funkce f spojitá na $\mathbb{R}$, tak je množina jejích nulových bodů uzavřená.

Nástin řešení

1. Množina X je otevřená v (M, d), pokud pro každý bod $x \in X$ existuje okolí x kladného poloměru, které je podmnožinou X, neboli existuje $\varepsilon > 0$ takové, že množina $\{y \in M;\ d(x, y) < \varepsilon\}$ je podmnožinou X. Množina X je uzavřená, pokud její doplněk $M \setminus X$ je otevřená množina. Jiná možná definice říká, že množina X je uzavřená, pokud každá konvergentní posloupnost, jejíž prvky patří do X, má limitu v X.
2. (a) Množina X_a je uzavřená: její doplněk je množina $(-\infty, 0)$, která je zjevně otevřená.
   (b) Množina X_b není uzavřená: její doplněk obsahuje nulu, ovšem každé okolí nuly kladného poloměru má neprázdný průnik s X_b, tedy doplněk X_b není otevřená množina a X_b není uzavřená.
3. Předpokládejme, že $f\colon \mathbb{R} \to \mathbb{R}$ je spojitá, označme N_f množinu jejích nulových bodů a $\overline{N}_f = \mathbb{R} \setminus N_f$ množinu jejích nenulových bodů. Chceme ukázat, že $\overline{N}_f$ je otevřená množina. Volme tedy libovolné $x \in \overline{N}_f$ a dokažme, že nějaké okolí x kladného poloměru je podmnožina $\overline{N}_f$. Z definice $\overline{N}_f$ platí $f(x) \neq 0$, označme tedy $\varepsilon = |f(x)|/2 > 0$. Z definice spojitosti plyne, že existuje $\delta > 0$ takové, že pro každé x' splňující $|x - x'| < \delta$ platí $|f(x) - f(x')| < \varepsilon$. Potom ovšem každé x' splňující $|x - x'| < \delta$ patří do $\overline{N}_f$, neboť platí $|f(x')| > |f(x)| - |f(x) - f(x')| > 2\varepsilon - \varepsilon > 0$. Tedy okolí x o poloměru δ je celé obsaženo v $\overline{N}_f$. Z toho plyne, že $\overline{N}_f$ je otevřená množina a N_f je tedy uzavřená.

   Alternativně je možné se při řešení této úlohy odvolat na větu, která říká, že u spojité funkce je vzorem uzavřené množiny vždy uzavřená množina. Protože $\{0\}$ je uzavřená množina, je její vzor, tedy množina N_f, také uzavřená množina.

15 Barvy a barevné systémy (specializace PGVVH-PG, PGVVH-VPH)

1. Jak barvy vnímá lidské oko? Uveďte alespoň přibližný koncept. Který barevný prostor na počítači tím byl inspirován?
2. Jak barvy zobrazuje počítačová technika? Popište několik systémů pro ukládání barev na počítači (barevné prostory). Všímejte si rozdílného přístupu a rozdílné technologie vzniku barev. Popište aspoň dva odlišné mechanismy vzniku barev.
3. Uveďte příklad barevného systému vhodného pro intuitivní zadávání barvy laickým uživatelem. Naznačte, jak by se tento barevný systém přepočítával do systému, který se používá v počítačových monitorech.

Nástin řešení 1. Na sítnici běžného lidského oka jsou čípky a tyčinky, barevnému vnímání přispívají hlavně čípky. Máme tři druhy čípků (podle třech druhů fotopigmentu): S, M, resp. L. Jsou nejvíce citlivé na krátké vlnové délky (S, modrá), střední vlnové délky (M, zelená), resp. dlouhé vlnové délky (L, červená). Z těchto tří kanálů se pak na sítnici a později v CNS skládá barevný vjem.

Skutečné skládání barev v CNS je komplikovanější, avšak zhruba řečeno, S, M a L čípky daly vzniknout technickému/počítačovému barevnému prostoru RGB.

2. Technicky se barvy zobrazují zejména dvěma mechanismy:

A. sčítání barev (aditivní princip) se uplatňuje všude, kde se jednotlivé barevné kanály sčítají a jejich prostá superpozice pak určuje výsledek. Minimální signál odpovídá černé barvě, maximum ve všech kanálech pak barvě bílé. Barevné systémy: RGB, sRGB. Technologie grafických výstupních zařízení: CRT monitory, LED a OLED displeje, projektory s třemi lampami.

B. subtraktivní skládání barev se vyznačuje tím, že se od bílého/multispektrálního světla "odečítají"/filtrují specifické složky tak, aby vzniklo požadované spektrum. Minimální signál odpovídá bílé barvě (původní zdroj světla), maximum pak černé (vše se odfiltrovalo). Barevné systémy: CMY, CMYK. Technologie zobrazení barev: tisk na papír, jakákoli grafika na papíře nebo zdi (olejové barvy, tempery, akrylové barvy, tuše a inkousty, pastely, pastelky...), LCD, TFT monitory a televize (zvenku se však tváří jako RGB zařízení)

3. Hodil by se k tomu např. HSV systém (barevný kruh), kde H (hue) je základní barva, S (saturation) je sytost a V (value) jas. H a S se dají snadno intuitivně zadávat v barevném kruhu nebo na obdélníkové ploše.

Převádění HSV na RGB (zhruba řečeno): podle V by se určovala hodnota maximální složky na výstupu a S definuje, jak silná bude minimální složka: je-li S nulové, platí $R = G = B$ (jedná se o monochromatickou barvu), pro $S = 1$ je naopak aspoň jedna z výstupních složek nulová. Pro všechny barvy kromě černé platí $S = (max - min)/max$. Jestliže již máme spočítané min a max, poslední vstupní parametr H určí, které RGB kanály budou nabývat minima (aspoň jeden), které maxima (také aspoň jeden) a jakou hodnotu bude mít ten poslední kanál. Celkem bychom dostali šest různých vzorců pro šest výsečí barevného kruhu.

16 Řádkové vyplňování mnohoúhelníka (specializace PGVVH-PG, PGVVH-VPH)

Budeme se zabývat vybarvením vnitřku rovinného mnohoúhelníka v rastrovém prostředí (čtvercová mřížka pixelů), obrys útvaru není potřeba obkreslovat.

1. Jak by měl být mnohoúhelník zadán a jaké možnosti definice jeho vnitřku mají smysl?
2. Popište základní princip algoritmu řádkového vyplňování 2D mnohoúhelníka. Uveďte i případné pomocné datové struktury používané v průběhu vyplňování.
3. Jak by se algoritmus zjednodušil, kdyby měl vyplňovat pouze konvexní útvary?

Nástin řešení 1. Mnohoúhelník je zadán uzavřenou posloupností vrcholů, nezáleží, ve kterém vrcholu se začne ani na orientaci (CW nebo CCW). Každý vrchol má dvojici souřadnic $[x, y]$. Dvě nejčastější definice vnitřku:

1. Podle parity (odd-even rule) - každá hrana mění příslušnost do vnitřku polygonu, přejdu přes jednu hranu – jsem uvnitř, přejdu přes další – jsem opět venku.
2. Podle stupně obtočení (winding rule) - jakoby byl obvod polygonu vyroben z provázku, pokud do daného bodu zapíchnu prst, jsem venku, pokud lze provázek odstranit, jinak jsem uvnitř.

2. Řádkové vyplňování: reprezentuji si všechny nevodorovné hrany a budu udržovat jejich průsečíky s vodorovnou vykreslovanou řádkou (scanline). Touto řádkou postupuji shora dolů po jednom pixelu a na ní vždy jednoduše vybarvím vnitřní pixely polygonu. K tomu mi poslouží setřídění průsečíků zleva doprava (podle souřadnice x - viz seznam C níže) a aplikace příslušného pravidla z 1. Po řádkách se postupuje indukcí: řádka se posune o jeden pixel dolů, seznam průsečíků C se musí upravit:

1. Pokud se na nové řádce nachází některý z vstupních vrcholů polygonu (viz seznam I), tak vzniká nový průsečík (průsečík zanikne automaticky, když se vynuluje jeho čítač)
2. Ostatní průběžné průsečíky se posunou v souřadnici x o diferenci (směrnici, tedy konstantu, kterou jsme si dopředu u každé hrany spočítali dělením dx/dy) a dekrementuje se jejich čítač
3. Protože se horizontální polohy průsečíků mění, musí se přetřídit seznam C

Algoritmus končí, když zanikne poslední průsečík (tj. jeho hrana skončí). Pomocná data: setříděný seznam průsečíků C ("current": x, dx/dy, čítač = za kolik řádků hrana skončí). Vstupní data se převedou na vstupní seznam I "input", kde jsou dosud nezpracované hrany ($[x, y]$, dx/dy, výška hrany v pixelech = iniciální hodnota pro čítač); tento seznam je nejlepší udržovat setříděný podle y (v každé řádce se v něm bude hledat, jestli nějaký nový průsečík nevzniknul).

3. Konvexní mnohoúhelník: seznam C se zredukuje na dvojici průsečíků (začátek a konec vyplňování), nemusí se třídit. Přechod na další řádku (scanline) bude jednodušší - pokud některá z těchto dvou hran končí, nahradí se sousední hranou (v souladu s pořadím hran v obrysu polygonu).

17 Model odrazu světla BRDF a stínování (specializace PGVVH-PG, PGVVH-VPH)

Tato otázka se týká lokálního modelu odrazu světla na povrchu tělesa při zobrazení 3D scény.

1. Popište jednoduchý lokální model odrazivosti světla (jak barva pozorovaná na povrchu tělesa závisí na poloze a intenzitě světelného zdroje a pozici pozorovatele).
2. Jak se započítávají příspěvky více světelných zdrojů, které může 3D scéna zobrazovat? Je vhodné zohlednit vzdálenost světelých zdrojů od tělesa?
3. Jak je možné dosáhnout efektu hladkého tělesa, i když je 3D model sestaven z trojúhelníků? Uvažujte postup: původně je povrch tělesa reprezentován exaktně matematicky jako hladká plocha (třeba v parametrickém tvaru). Pro uložení povrchu v paměti GPU se vytvoří síť trojúhelníků, která původní hladkou plochu aproximuje. Jak se takový povrch stínuje, aby nebyly vidět jednotlivé trojúhelníky?

Nástin řešení 1. Např. Phongův model: odražené světlo se skládá z difusní složky E_D (diffuse), lesklého odrazu E_S (specular) a případně se přidává nefyzikální okolní složka E_A (ambient). Základem difusní složky je $cos(\alpha)$, kde α je úhel dopadu světla od normály. Základem lesklého odrazu je $cos(\beta)^h$, kde β je odchylka pozorovacího směru od dokonale zrcadlového odrazu a h exponent odlesku (highlight). Difusní barva je barvou tělesa, barva odlesku se obvykle ztotožňuje s barvou zdroje světla. Okolní složka je jen konstantní příspěvek s barvou tělesa, který se přičítá bez ohledu na jakékoli směry a úhly pohledu.

Další detaily, vzorce a obrázky - viz prezentace z přednášky NPGR003 nebo jakákoli učebnice základů 3D grafiky.

2. Pokud na těleso svítí ve scéně více zdrojů světla, použije se ve vzorci jedenkrát okolní složka E_A, difusní $E_D(i)$ a lesklé složky $E_S(i)$ se pro jednotlivé zdroje sčítají.

Vzdálenosti zdrojů světla by se správně fyzikálně měly tlumit čtvercem vzdálenosti ($1/{d_i}^2$). V běžné LDR grafice to však vede k příliš velkým kontrastům, proto se v praxi zabydlel nesprávný vzorec $1/(a \cdot {d_i}^2 + b \cdot d + c)$, kde a, b a c jsou konstanty (aspoň jedna musí být nenulová a nastavuje se individuálně podle scény).

3. Použije se interpolační stínování: Gouraudovo nebo Phongovo. To znamená, že potřebujeme znát ve vrcholech mnohostěnu původní normálové vektory. Při Gouraudově metodě se ve vrcholech spočítá barva povrchu (za pomoci všech materiálových konstant, normálového vektoru a směrů ke světelným zdrojům), a tato RGB barva se dál interpoluje do všech kreslených pixelů/fragmentů. Phongova metoda je dokonalejší (lépe se v ní vykreslí odlesky E_S), spočívá v tom, že se do fragmentů interpoluje normálový vektor a celý výpočet barvy se přenese do fragmentů/pixelů (k tomu je potřeba implementace stínování ve fragment/pixel shaderu).

18 Textury ve 3D grafice (specializace PGVVH-PG)

1. Jaký je rozdíl mezi 2D a 3D texturou? Uveďte typické aplikace těchto dvou přístupů.
2. Jaké jsou hlavní výhody a nevýhody 3D textur při použití v ray-tracingu?
3. Jak je možné napodobit vnitřní strukturu materiálu (dřevo, mramor) pomocí textury nebo procedurální definice v paprskovém zobrazovači (ray-tracing, path-tracing)? Pište o simulaci struktury materiálu, ne o zobrazovači.

Nástin řešení 1. 2D textura je plošný rastrový obrázek nebo předpis, který se na povrch 3D objektu musí mapovat (mapování textury, texturové souřadnice $[u, v]$ nebo $[s, t]$). Mapování může dělat problémy v případě analyticky reprezentovaných tvarů (např. v ray-tracingu), v realtime grafice se 2D texturové souřadnice uvádějí explicitně v atributech vrcholů modelu.

3D textura reprezentuje přímo vnitřní strukturu materiálu, textura je zobrazením/předpisem s 3D definičním oborem.

Příklady 2D textur: tapeta, nálepka, vyfotografovaný povrch reálného předmětu.

Příklady 3D textur: dřevo, mramor, porézní materiály.

2. Výhody 3D textury: nemusí se mapovat, 3D souřadnice průsečíku (ray-based rendering) nebo fragmentu (GPU) se přímo použijí jako argumenty textury.

Nevýhody: při reprezentaci tabulkou (rastrem) je velká spotřeba paměti (a tím nepřímo i menší rychlost). Při použití šumových funkcí se musí počítat ve 3D, což může zpomalit výpočet. Menší nevýhoda: při animaci si musíme dát pozor na souřadnou soustavu, ve které texturu používáme, ta se musí pohybovat spolu s animovaným objektem, v případě deformací objektu musíme implementovat příslušnou deformaci definičního oboru textury (to nemusí být vůbec jednoduché).

3. Příklad simulace dřeva nebo mramoru:

Namodelujeme si nejdříve vnitřní strukturu daného materiálu v ideálním případě – u dřeva by to byly ideálně soustředné válcové plochy letokruhů, u mramoru dokonale rovnoběžné vrstvy jednotlivých minerálů. V obou případech to znamená mít prostorově definovanou funkci $C_{ideal} : [x, y, z] \mapsto [R, G, B]$.

Pak použijeme nějakou prostor deformující funkci (funkce), kterými vstupní prostor nahodile křivíme tak, aby to co nejlépe odpovídalo struktuře přírodního materiálu. Používají se generátory spojitého šumu (např. slavný Perlinův šum) nebo se syntetizuje umělá turbulence pomocí několika složek běžného šumu (zvyšující se frekvence a snižující se amplituda).

Nechť máme tři různé deformující (šumové) funkce N_x, N_y a N_z, kde např.

$$N_x : [x, y, z] \mapsto x'$$

pak výslednou šumovou texturu postavíme jako

$$C_{wood}(x, y, z) = C_{ideal}(N_x(x, y, z), N_y(x, y, z), N_z(x, y, z))$$

19 Kvaterniony a jejich využití v animaci (specializace PGVVH-VPH)

Jde nám o reprezentace orientace pevného tělesa v 3D prostoru.

Pevné těleso se obvykle v čase animuje tak, že se jako celek posunuje (translace) a otáčí (orientace, rotace). V této otázce se zaměříme pouze na tu druhou - rotační - složku pohybu.

1. Definujte kvaternion. Popište, jaký datový typ se při jeho implementaci používá (kolik zabírá paměti)? Napište některé základní algebraické vlastnosti kvaternionů. Jak byste odvodili vzorec pro násobení kvaternionů (abyste si ho nemuseli pamatovat)?
2. Která speciální třída kvaternionů se používá k reprezentaci orientace ve 3D? Jak pomocí kvaternionu otočit bod/vektor kolem dané osy o daný úhel?
3. Jaký je postup při animaci (přechodu) mezi dvěma orientacemi? Počáteční orientaci nechť vyjadřuje kvaternion q a koncovou orientaci kvaternion r. Čas t budeme pro jednoduchost předpokládat v intervalu $[0, 1]$. Napište vzorec pro orientaci v libovolném čase t. Naznačte postup, který by se použil pro složitější animaci mezi více klíčovými orientacemi.

Nástin řešení 1. Kvaternion je objekt z tělesa Q (co do dimenze algebraicky ekvivalentní s R^4), je vlastně 4D rozšířením tělesa komplexních čísel.

$$q = ix + jy + kz + w = (\mathbf{v}, w)$$

i, j, k jsou imaginární jednotky s vlastnostmi

$$i^2 = j^2 = k^2 = ijk = -1$$

z toho již lze odvodit další rovnosti

$$jk = -kj = i$$

$$ki = -ik = j$$

$$ij = -ji = k$$

sčítání je po složkách, násobení lze odvodit roznásobením podle původní definice a použitím předchozích tří rovnic. Konjugovaný kvaternion

$$q^* = (\mathbf{v}, w)^* = (-\mathbf{v}, w)$$

Norma (absolutní hodnota na druhou)

$$||q||^2 = n(q) = qq^* = x^2 + y^2 + z^2 + w^2$$

Převrácená hodnota

$$q^{-1} = q^*/n(q)$$

2. K reprezentaci orientace se používají jednotkové kvaterniony - každý jednotkový kvaternion $n(q) = 1$ se dá vyjádřit jako

$$q = (u_q sin\Phi, cos\Phi)$$

pro vhodný jednotkový 3D vektor u_q a úhel Φ. Tento kvaternion reprezentuje otočení kolem osy u_q o úhel Φ.

Protože platí $q = exp(\Phi u_q)$ a $logq = \Phi u_q$, lze jednoduše vyjádřit obecnou mocninu jednotkového kvaternionu:

$$q^t = exp(t\Phi u_q) = u_q sint\Phi + cost\Phi$$

(to budeme potřebovat později, v 3.)

Otáčený vektor/bod se rozšíří do 4D $p = [p_x, p_y, p_z, 0]$. Rotace tohoto vektoru/bodu kolem osy procházející počátkem u_q o úhel 2Φ je

$$p' = qpq^{-1} = qpq^*$$

3. Orientaci/rotaci v libovolném čase t vyjadřuje tzv SLERP (spherical linear interpolation) s definicí

$$SLERP(q, r, t) = q(q^*r)^t$$

(úpravami z předchozích částí odpovědi lze dospět k různým vyjádřením, např. lineární kombinaci q a r s pomocí goniometrických funkcí)

Pro složitější hladké animace založené na více klíčových snímkách (keyframes) bychom v každém klíčovém snímku definovali kvaternion q_i a celkovou „dráhu" orientace bychom definovali pomocí vhodného spline, např. navazujícími Bezierovými křivkami. De Casteljau algoritmus konstruuje libovolný bodu na i-tém úseku křivky pouze pomocí lineárních interpolací s parametrem t. V doméně kvaternionů bychom tyto interpolace nahradili operacemi $SLERP$ (např. se tam bude v první úrovni vyskytovat $SLERP(q_i, q_{i+1}, t)$).

20 Cache, výkonnost, asociativita, reprezentace datových struktur (specializace PVS)

Uvažujte následující (částečnou) implementaci binárního vyhledávacího stromu:

```
struct Node {
    int key;
```

```
    struct Node *left;
    struct Node *right;
};

bool contains (struct Node *root, int value) {
    while (root != NULL) {
        if (root->key == value) return true;
        if (root->key < value) root = root->left;
                          else root = root->right;
    }
    return false;
}
```

Předpokládejte dále, že funkce `contains` je opakovaně volaná s náhodnými hodnotami `value` na stromu alokovaném na heapu. Zvolte si (rozumně) parametry první úrovně datové cache procesoru a dalších relevantních komponent prostředí, napište je a zodpovězte (včetně zdůvodnění) následující otázky:

1. Pokud je L1 plně asociativní, jaká je největší velikost stromu, u které je zaručeno, že při volání `contains` bude v L1 docházet pouze k povinným (compulsory, cold) výpadkům?
2. Pokud je L1 omezeně asociativní, jaká je největší velikost stromu, u které je zaručeno, že při volání `contains` bude v L1 docházet pouze k povinným (compulsory, cold) výpadkům?
3. Dalo by se řešení prvních dvou bodů zlepšit (větší velikost), pokud bychom měli kontrolu nad strategií volby oběti? Pokud ano, jak?
4. Dalo by se řešení prvních dvou bodů zlepšit (větší velikost), pokud bychom měli kontrolu nad zarovnáním datových struktur? Pokud ano, jak?
5. Jak by se řešení prvních dvou bodů zlepšilo, pokud bychom zajistili z pohledu cache ideální přiřazení adres datových struktur? Jak by takové přiřazení vypadalo?

V řešení (zjednodušeně) předpokládejte, že mimo funkci `contains` se obsah L1 nemění.

Pro připomenutí, jako povinné (compulsory, cold) výpadky se označují takové, při kterých daná adresa dosud nebyla v cache hledána.

Nástin řešení Volené parametry například velikost L1 cache 32kB, velikost cache line 64B, strategie LRU, `int` 4B, pointer 8B.

1. Záruky na zarovnání struktury `Node` nemohou vyloučit umístění na hranici cache line, každý uzel stromu tedy může v nejhorším případě zabrat dvě cache lines. Pro plně asociativní cache máme k dispozici 32kB/64B = 512 cache lines, tedy nejvýše 256 uzlů stromu. Funkce `contains` není rekurzivní a má tak málo lokálních proměnných, že se vejdou do registrů, další cache lines tedy nebudou použity.
2. V nejhorším případě mohou adresy všech uzlů stromu v cache kolidovat, pak můžeme bez konfliktních (conflict) výpadků přistupovat nejvýše k tolika cache lines, kolik je stupeň asociativity cache. Počet uzlů však nemusíme dělit dvěma i pokud struktura `Node` bude ležet na hranici cache line, protože sousední cache lines nebudou patřit do stejného setu, pro stupeň asociativity n tedy máme k dispozici $n - 1$ uzlů.
3. Protože řešíme pouze povinné (compulsory) výpadky a neuvažujeme změnu obsahu L1 mimo volanou funkci, strategie volby oběti nemá na řešení vliv (s výjimkou hypotetické situace, kdy by cache byla před prvním voláním `contains` naplněna tak, že by strategie upřednostnila dřívější obsah před daty stromu, ale k tomu nemůže docházet opakovaně, jinak by to znamenalo, že strategie může nechat nějakou část cache navždy obsazenou nepoužívanými daty).
4. Zarovnáním můžeme předejít umístění struktury `Node` na hranici cache line, u prvního bodu se tedy počet uzlů zdvojnásobí, u druhého zvětší o 1.
5. V ideálním případě bychom alokovali struktury na po sobě jdoucích adresách (předpokládá se, že heap alokátor by měl metadata umístěna jinde), pak pro velikost struktury 24B můžeme přistupovat k 32kB/24B = 1365 uzlům. V případě částečně asociativní cache budou po sobě jdoucí adresy také vyhovovat (typická cache první úrovně bude indexovaná virtuální adresou a set budou volit nejnižší možné bity adresy).

21 Polymorfní kontejner (specializace PVS)

Napište v C++ deklarace a potřebné definice datové struktury (kontejneru, konkrétní operace jsou popsány níže), která umožní uložení hodnot různých typů, např. int, double, string, complex, zlomky, apod. Předpokládejte, že všechny uložitelné hodnoty konkrétního typu jsou zabaleny do třídy odvozené od společného předka AbstractVal s následující deklarací:

```
class AbstractVal {
public:
  virtual void print() = 0;
};
```

Implementace virtuální metody `print` v odvozených třídách zajistí výstup uložené hodnoty daného typu na standardní výstup.

Pro vlastní uložení potřebných dat použijte vhodný standardní kontejner. Řešení by mělo

- být maximálně efektivní
- využívat vhodné syntaktické konstrukce C++
- odpovídat zásadám kvalitního objektového návrhu a principům sw inženýrství

Drobné syntaktické chyby nejsou významné.

1.

Napište deklaraci polymorfního kontejneru `PolyCt` a definici metod `add` (pro přidání nového prvku) a `print` (pro výstup všech hodnot kontejneru na standardní výstup). Jiné operace na kontejneru (např. přístup k prvkům, vlastní iterace apod.) nejsou vyžadovány. Pokud je to pro vaše řešení potřebné, napište definici dalších metod (např. konstruktor/y apod.).

Příklad použití polymorfního kontejneru: (**** značí vhodný C++ kód, lze i prázdný)

```
class IntVal : public AbstractVal { **** };
class StringVal : public AbstractVal { **** };

PolyCt p;
p.add( **** 123 ****);
p.add( **** "abc" ****);
p.print();
```

Vytiskne: `123 abc`

2.

Upravte (pokud je to potřebné) třídu `PolyCt`, případně doplňte rozhraní `AbstractVal` a jeho implementaci v konkrétních typech tak, aby umožňovala přiřazení a copy-konstruktor celého kontejneru s hodnotovou sémantikou (tj. deep copy, tak, jak je to v C++ běžné). Nezapomeňte na možnost přiřazení do stejné proměnné (self-assignment).

Příklad:

```
PolyCt p1, p2;
p1.add( **** 123 ****);
p1.add( **** "abc" ****);
p2 = p1;
p1.add( **** 456 ****);
p2.print();
```

Vytiskne: `123 abc`

Nástin řešení 1.

Standardní kontejnery umožňují pouze uložení hodnot stejného typu, proto je vhodný abstraktní předek. V kontejneru bude uložen syntakticky ukazatel na společného předka, v runtime ukazatel na konkrétní zabalenou hodnotu. Nejvhodnější a nejjednodušší je použít `unique_ptr`, který je předán metodou add. Řešení pomocí `shared_ptr` není vhodné (zbytečně neefektivní). Řešení pomocí syrových ukazatelů je možné, ale vyžaduje zvýšenou opatrnost (a příslušný kód) pro memory management a zabránění memory leaks. Řešení pomocí pokročilejších technik (variant, type erasure apod.) je možné.

2.

Přímé kopírování hodnot referencovaných ukazatelem na `AbstractVal` není možné (slicing), stejně tak default copy konstruktor a přiřazení. Nejvhodnější řešení je pomocí metody clone deklarované v `AbstractVal` a implementované v konkrétních typech. Self-assignment je nutný ošetřit testem na rovnost zdroje a cíle, jinak by nastalo smazání dat. Řešení např. pomocí výčtového typu a switche přes možné typy není vhodné (rozšiřitelnost, open-closed principle). Použití variant a visit je možné, není však vyžadováno.

22 Datový model a SQL (specializace PVS)

V informačním systému nemocnice byla vytvořena relace Data
se schématem Data(**Čas, Pacient**, Nemoc, Lékař).

1. Analýzou funkčních závislostí se zjistilo, že:
   - Čas, Pacient → Nemoc, Lékař
   - Čas, Lékař → Pacient
   - Pacient → Nemoc

   Byly proto navrženy dva způsoby s cílem pomocí dekompozice získat normalizované relační schéma alespoň ve 3NF:

   (a) Pomocí závislosti $Cas, Pacient \rightarrow Nemoc$ dekomponovat relaci na relace
   Data1(**Čas, Pacient**, Nemoc) s klíčem {Čas, Pacient} a
   Data2(**Čas, Pacient**, Lékař) s klíči {Čas, Pacient} a {Čas, Lékař}

   (b) Pomocí závislosti Pacient → Lékař dekomponovat relaci na relace
   Data1(**Pacient**, Lékař) s klíčem {Pacient} a
   Data2(**Čas, Pacient**, Nemoc) s klíčem {Čas, Pacient}

   Zhodnoťte tyto návrhy z hlediska dosažení cíle a zvolte řešení, které by mělo být použité. Pokud se domníváte, že žádné z nich, navrhněte vlastní řešení. Svá rozhodnutí o použití či nepoužití všech řešení zdůvodněte.

2. Nad Vámi vybraným/navrženým schématem napište následjící dotazy v jazyce SQL:
   (a) Se kterými nemocemi se léčí více než 1000 pacientů?
   (b) Vypište bez použití ORDER BY poslední návštěvu (nikoli jen její čas) pacienta 'Martin Nemocný' u lékaře.
   (c) Kolik lékařů neléčí pacienty s nemocí "Zápal plic"?

Nástin řešení

1. – Při dekompozici je potřeba vybrat takovou funkční závislost relace, která brání v dosažení potřebné NF. Zde alespoň 3NF. Jediná taková je závislost Pacient → Nemoc, která porušuje 2NF. Jejím použitím vzniknou relace Data1(**Pacient**, Nemoc) a Data2(**Čas, Lékař**, Pacient), obě v BCNF. Žádná závislost se neztratí. Toto řešení tedy odpovídá zadání.
   – V případě prvního navrženého řešení se jedná o závislost na klíči, která žádnou NF neporušuje. Dostaneme tak dvě relace se stejným klíčem. Závislost Pacient → Nemoc stále porušuje 2NF v relaci Data1.
   – V případě druhého řešení použitá závislost Pacient → Lékař není v relaci Data platná (odvoditelná). Její použití by vedlo k dekompozici, která nesplňuje základní požadavek na bezeztrátovost spojení. Závislost Pacient → Nemoc stále porušuje 2NF, tentokrát v relaci Data2. Navíc se ztratila závislost Čas, Lékař → Pacient.
2. (a) Např. SELECT Nemoc FROM Data1 GROUP BY Nemoc HAVING COUNT(*)>1000
   (b) Např. SELECT * FROM Data2 WHERE Pacient='Martin Nemocný' AND Čas>=ALL
   (SELECT Čas FROM Data2 WHERE Pacient='Martin Nemocný')
   (c) Např. SELECT COUNT(DISTINCT N.Lékař)
   FROM Data2 N LEFT OUTER JOIN Data1 P ON (P.Pacient=N.Pacient AND P.Nemoc='Zápal plic')
   WHERE P.Pacient IS NULL

23 Správa verzí kódu (specializace PVS)

Na kódu bude pracovat více vývojářů. Kód bude hostován ve veřejném repozitáři a používán veřejností již v době vývoje. Máte navrhnout pravidla pro správu verzí vyvíjeného kódu. Pravidla zahrnují jednak názvy a účely větví (branches), které budou vývojáři používat, ale i informace kdy a odkud větve vznikají, a kdy, kam a za jakých podmínek ze větve spojují (mergují). Pravidla by neměla být komplikovanější, než vyžaduje daná situace. Popište pravidla potřebná pro následující situace, které na sebe navazují:

1. Na kódu pracuje několik vývojářů. Nestává se, že by se pracovalo na dvou nezávislých úpravách naráz. Je však potřeba zajistit, že veřejnosti je vždy k dispozici funkční verze.
2. Nyní se navíc stává, že různí vývojáři pracují na různých úpravách naráz. Nechtějí, aby si své rozpracované úpravy navzájem rozbíjeli.
3. Nyní se navíc stává, že se blíží datum vydání již vyvinutých úprav, které ještě nemusí být zcela otestovány. Přitom je třeba dále pracovat na dalších úpravách, které mohou být dokončeny i dříve, než testování předchozích úprav.
4. Nyní se navíc stává, že v již vydané verzi jsou nalezeny chyby, které je třeba prioritně opravit, nezávisle na stavu aktuálně rozpracovaných úprav.

Nástin řešení Popsat lze například Gitflow se vzrůstající složitostí.

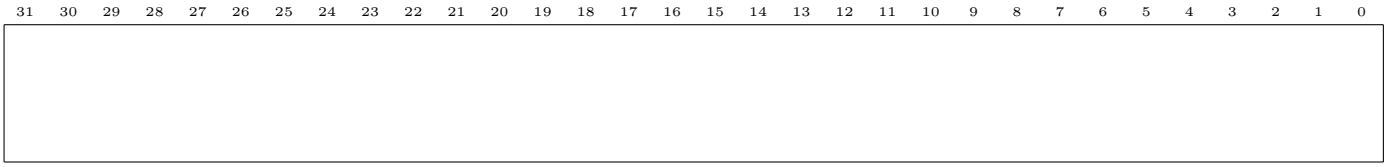
1. Zde stačí základní 2 branche:
   - `main` - existuje po celou dobu existence kódu, vždy obsahuje poslední funkční verzi pro veřejnost.
   - `develop` - sem přispívají vývojáři. Existuje po celou dobu projektu, může obsahovat i chybový stav. Když je dosaženo funkčního stavu vyvíjené úpravy, je mergnuto do `main`.
2. Oproti minulé situaci navíc/jinak:
   - `develop` - existuje po celou dobu existence kódu, slouží pro integraci různých úprav. Může obsahovat i chybový stav způsobený nedokončenou integrací různých úprav. Když je dosaženo funkčního stavu obsahujícího nové úpravy, je mergnuto do `main`.
   - `feature-xyz` - sem přispívají vývojáři. Pro každou úpravu je jedna větev. Začíná z `develop` spolu s prací na úpravě. Když je dosaženo funkčního stavu vyvíjené úpravy, je mergnuto do `develop` a branch je smazána.
3. Oproti minulé situaci navíc/jinak:
   - `develop` - odsud se do `main` již nemerguje.
   - `release` - při přiblížení termínu vydání je vytvořena z `develop`, přispívají se do ní pouze bugfixy. Po dokončení vydání je mergnuto do `main` i `develop`.
4. Oproti minulé situaci navíc/jinak:
   - `hotfix` - vzniká z `main` branche, obsahuje pouze opravu nalezené chyby. Je mergnuta zpět do `main`, ale i do aktuální `release` branche nebo `develop`, pokud aktuálně release neexistuje.

24 Stránkování (specializace SP)

Předpokládejme procesor s virtuálními adresami velikosti 32 bitů a fyzickými adresami velikosti 36 bitů, který pro překlad virtuálních adres používá softwarově plněný TLB s pevnou velikostí stránky 16 MB.

Položka v TLB má velikost 4 B a obsahuje (po řadě od nejvyšších bitů) (nejmenší) potřebnou část virtuální a fyzické adresy, operačním systémem přiřazený ASID (identifikátor adresového prostoru), global bit (při přístupu je ignorován ASID) a bity executable, writable, readable (s obvyklým významem) a valid bit. Zbylé tři nejnižší bity jsou rezervované jako 0.

1. Načrtněte formát položky TLB včetně počtu bitů jednotlivých polí.



2. Předpokládejte následující obsah TLB (první řádek je volný pro vaše poznámky):

0x26A01218
0x27A08278
0x27A07378
0x32A023D8
0x28A09378
0x25A04258
0x29A05350
0x25A06210

Dále předpokládejte, že aktuální ASID je nastaven na hodnotu `0x02` a dále se nemění.

Co se stane při přístupech na následující virtuální adresy? V případě, že dojde k úspěšnému překladu, napište konkrétní fyzickou adresu, ke které procesor přistoupí. V případě, že překlad nebude moci procesor provést, ve stručnosti popište činnost procesoru a případnou (typickou) reakci operačního systému pro napravení situace.

Jednotlivé přístupy uvažujte izolovaně, tedy vycházejte vždy z obsahu TLB uvedeného výše, i kdyby v některém kroku případná reakce operačního systému mohla vést ke změně TLB.

(a) zápis na `0x29C0A154`

(b) čtení z `0x27B38247`

(c) čtení z `0x324F8235`

(d) zápis na `0x258A3CF2`

Odpovědi přiměřeně komentujte tak, aby bylo možné vidět, jak jste dospěli k výsledku.

Nástin řešení

1. Z velikosti stránky je na virtuální adresu potřeba 8 bitů, na fyzickou 12 bitů, k tomu 5 jednotlivých bitů (GXWRV), 3 bity reserved, do 32 zbývají 4 bity na ASID.
2. Je potřeba uvažovat pouze platné položky se správným ASID, popř. s global bitem.
   (a) `0x29C0A154` není v TLB, TLB miss
   (b) `0x27B38247`, položka `0x27A08278`, čtení z `0xA08B38247`
   (c) `0x324F8235`, položka `0x32A023D8`, čtení z `0xA024F8235`
   (d) `0x258A3CF2`, položka `0x25A04258` nemá writable bit, TLB write fault

   V případě platných položek dojde přímo ke čtení, v ostatních případech dochází k výjimce, procesor deleguje vyřešení situace na OS (ten typicky buď naplní TLB a restartuje instrukci, nebo vlákno ukončí).

25 Rozhraní pro synchronizaci (specializace SP)

Předpokládejme, že v následujícím programu volání `putchar` okamžitě zobrazí znak předaný jako parametr. S výjimkou volání `trylock` také předpokládejme, že volání žádné funkce nekončí s chybou.

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  static pthread_mutex_t mutex_one = PTHREAD_MUTEX_INITIALIZER;
5  static pthread_mutex_t mutex_two = PTHREAD_MUTEX_INITIALIZER;
6  static pthread_t thread_one, thread_two;
7
8  void *thread_one_function(void *unused) {
9      pthread_mutex_lock(&mutex_one);
```

```
10      pthread_mutex_lock(&mutex_two);
11      putchar('A');
12      putchar('B');
13      pthread_mutex_unlock(&mutex_one);
14      pthread_mutex_unlock(&mutex_two);
15      putchar('C');
16      putchar('D');
17
18      return NULL;
19  }
20
21  void *thread_two_function(void *unused) {
22
23      while (1) {
24          pthread_mutex_lock(&mutex_two);
25          int res = pthread_mutex_trylock(&mutex_one);
26          if (res == EOK) {
27              break;
28          } // else assert(res == EBUSY);
29          pthread_mutex_unlock(&mutex_two);
30      }
31
32      putchar('a');
33      putchar('b');
34      pthread_mutex_unlock(&mutex_two);
35      pthread_mutex_unlock(&mutex_one);
36      putchar('c');
37      putchar('d');
38
39      return NULL;
40  }
41
42  void main(void) {
43      pthread_create(&thread_one, NULL, thread_one_function, NULL);
44      pthread_create(&thread_two, NULL, thread_two_function, NULL);
45      pthread_join(thread_one, NULL);
46      pthread_join(thread_two, NULL);
47  }
```

1. Jaké výstupy může program zobrazit (vyznačte také případné uváznutí)?
2. Závisí pravděpodobnost pozorování možných výstupů z prvního bodu na tom, zda má použitý systém pouze jedno jádro (single processor) nebo více jader (multiprocessor)? Vysvětlete a zdůvodněte.
3. Za jakých okolností (konkrétně pro jaký běh výše uvedeného programu) může dojít k tomu, že volání `pthread_mutex_trylock` vrátí hodnotu `EBUSY` (tj. již zamčený mutex)?

Nástin řešení

1. Všechny permutace `ABCDabcd`, kde písmena stejné velikosti neporušují pořadí abecedy a výpisy `AB` a `ab` nejsou proložené mezi sebou. Pokud zámky nezaručují fairness, program může také stále běžet bez výpisu (první vlákno čeká na `two`, druhé vlákno neustále předbíhá, současně ale nemůže získat `one`).
2. Závislost by mohla existovat z různých důvodů, zejména při typickém kvantu (milisekundy) není na jednom jádře pravděpodobné prolnutí běhu obou vláken.
3. První vlákno zamkne `one`, přeplánuje se, druhé vlákno zamkne `two` a tou dobou je již `one` zamčený.

26 Sockety (specializace SP)

Uvažujme následující pseudokód jednoduchého socketového serveru.

```
1  socket = create_socket(AF_INET, SOCK_STREAM)
2  socket.bind('127.0.0.1', 8080)
```

```
3  socket.listen()
4  while True:
5      (client, remote_addr) = socket.accept()
6      data = client.recv(8)
7      client.send(remote_addr.addr + ':' + remote_addr.port)
8      client.close()
```

Tento server spustíme na stroji `test.vpn.example.com` (s IP adresou `192.168.11.4`).

1. Nepředpokládáme-li další zvláštní nastavení, z jakých strojů bude tento server dostupný? Proč? Jakou cílovou adresu a port musí klient použít pro připojení k tomuto serveru? Co by bylo nutné změnit pro provoz serveru na jiném portu?
2. Předpokládejme, že se k tomuto serveru připojí najednou více uživatelů. Jakým způsobem operační systém a samotný server rozliší jednotlivá spojení?
3. K serveru se připojíme dvakrát krátce po sobě ze stejného počítače a pošleme i stejná data. Můžeme očekávat identickou odpověď? Proč?
4. Jaké nejjednoduší chování uživatelů způsobí nedostupnost služby pro ostatní uživatele? Jakým způsobem můžeme udělat server odolnější (uvažujte možné změny napříč všemi softwarovými vrstvami)?

Nástin řešení

1. Server bude dostupný pouze z počítače, na kterém běží, `bind` uvádí jen lokální adresu `127.0.0.1:8080`. Provoz na jiném portu vyžaduje změnu čísla portu ve volání `bind`.
2. Spojení se rozliší podle zdrojové adresy a zdrojového čísla portu (liší se pro klienty ze stejného stroje) (cílová adresa je stejná).
3. Je to málo pravděpodobné, zdrojové číslo portu (strana klienta) je nejspíše náhodně přiřazené operačním systémem.
4. Stačí se připojit a neposlat žádná data, tj. `recv` se zablokuje na dlouhou dobu. Možná obrana: vícevláknový server, neblokující zpracování (např. `select`), snížení timeoutů, omezení maximálního času pro jedno připojení, omezení počtu připojení z jedné adresy atd.

27 Testování funkčnosti (specializace SP)

Zadání používá pro ilustraci jazyk Java a framework JUnit4, nicméně otázky se týkají principů testování a v odpovědích můžete použít i jiná podobná prostředí.

Uvažujme následující fragment kódu:

```
1  import org.junit.*;
2  import static org.junit.Assert.*;
3
4  public class NumberParserTest {
5      private NumberParser parser;
6      @Before public void setUp () { parser = new NumberParser(10); }
7      @Test public void testZero () { assertEquals(0, parser.parse("0")); }
8      @Test public void testOne { assertEquals(1, parser.parse("1")); }
9      @Test public void testMinusOne { assertEquals(-1, parser.parse("-1")); }
10 }
```

Pro úplnost je třída `NumberParser` implementována následujícím způsobem.

```
public class NumberParser {
    private int radix;
    public NumberParser(int radix) { this.radix = radix; }
    public int parse(String number) { return Integer.parseInt(number, radix); }
}
```

Nicméně, pro podstatu otázky není konkrétní implementace třídy `NumberParser` důležitá, v odpovědích se zaměřte i na obecné principy spojené s testováním funkčnosti.

1. Vysvětlete účel třídy `NumberParserTest` a jejích jednotlivých atributů a metod.
2. Do třídy `NumberParserTest` by šlo evidentně analogicky doplňovat další metody například takto:

   ```
   @Test public void testTwo { assertEquals(2, parser.parse("2")); }
   @Test public void testThree { assertEquals(3, parser.parse("3")); }
   ```

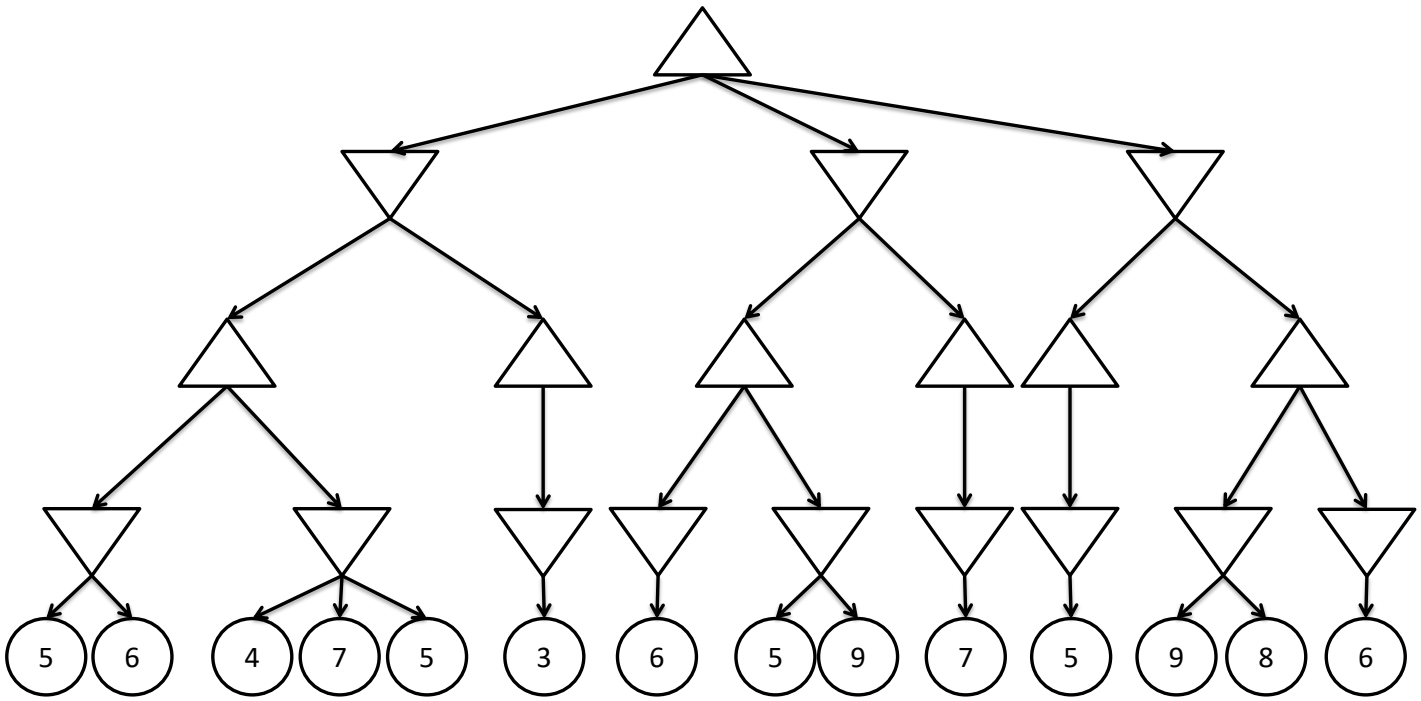
   Za jakých okolností by takové doplnění dávalo smysl, pokud vůbec? Zobecněte vaše vysvětlení.
3. Co se nejspíš stane, pokud v metodě `testMinusOne` vymažeme symbol mínus ve stringu (tj. tělo bude `assertEquals(-1, parser.parse("1"));`? Ve vaší odpovědi uvažujte nejen chování volaných metod, ale také širší kontext procesu vývoje software.

Nástin řešení

1. Testovací třída pro `NumberParser`, `@Before` obsahuje kód sdílený více testy, `@Test` metody jsou jednotlivé testovací případy; použitý framework pak řeší spuštění a autodiscovery všech testů.
2. Zde příliš smyslu nedává; obecně se snažíme testovat různé průchody programem (zde např. kladná a záporná čísla) a též okrajové případy (zde `0`, `MAX_INT`, `MAX_INT + 1` apod.).
3. Test skončí s chybou; pokud si jí nevšimne vývojář (ihned lokálně), očekáváme, že bude včasně odhalena v rámci (pre-integračního) testování na CI infrastruktuře.

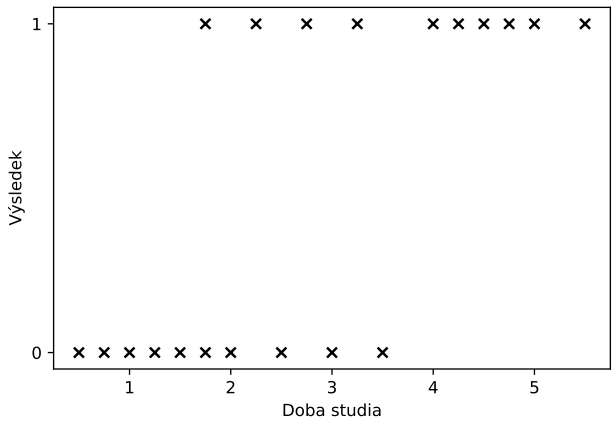
28 Minimax (specializace UI-SU, UI-ZPJ)

1. Popište algoritmus Minimax pro hru dvou hráčů. Vysvětlete, jaké má v praxi tento algoritmus nedostatky a jak je možné je řešit.
2. Zaveďte (popište) techniku alfa-beta prořezávání.
3. Na následujícím příkladu stromu hry s nulovým součtem symbol $\triangle$ označuje tah maximalizujícího hráče (tedy hrany vedoucí směrem dolů z tohoto symbolu odpovídají možným tahům maximalizujícího hráče). Obdobně symbol $\triangledown$ označuje tah minimalizujícího hráče. Symbol $\circ$ označuje koncový stav s uvedeným ohodnocením tohoto stavu. Doplňte do nekoncových stavů, jaké ohodnocení jim přiřadí algoritmus minimax za použití alfa-beta prořezávání. Vyznačte, které části stromu budou prořezány. Předpokládejte, že možné tahy se procházejí zleva doprava.



29 Logistická regrese (specializace UI-SU, UI-ZPJ)

Chceme použít logistickou regresi k vytvoření modelu, který bude předpovídat pravděpodobnost úspěchu studentů u zkoušky v závislosti na době učení. Pro získání dat jsme se po zkoušce ptali studentů, jak dlouho se učili. Jejich odpovědi společně s informací, jestli zkouška byla úspěšná, jsou v obrázku níže, který zobrazuje úspěch (1) a neúspěch (0) v závislosti na době učení.



1. Popište model logistické regrese. Jak se spočítá jeho výstup? Jakým způsobem se trénují jeho parametry? Jaká se používá ztrátová funkce?
2. Do obrázku výše zakreslete přibližně výstup modelu natrénovaného na zadaných datech. Nemusíte nic počítat.
3. Jakým způsobem by se model lišil, pokud bychom klasifikovali do více než dvou tříd (např. pokud bychom chtěli předpovídat i známku)?

30 Evaluace binárního klasifikátoru (specializace UI-SU, UI-ZPJ)

Testovací množina obsahuje 1000 příkladů. Binární klasifikátor z nich 400 klasifikoval do pozitivní třídy a 600 do negativní třídy s úspěšností (accuracy) 70 % a specificitou (specificity) 80 %.

1. Napište kompletní matici konfuze (confusion matrix).
2. Vypočtěte přesnost (precision) klasifikátoru.
3. Vypočtěte úplnost (recall) klasifikátoru.
4. Je možné na základě uvedených dat vypočítat F-score? Pokud ano, jak?

Nástin řešení

1. TN = 400, FN = 200, TP = 300, FP = 100
2. P = TP/(TP+FP) = 0.75, R = TP/(TP+FN) = 0.6
3. Ano. F1 = 2*P*R/(P + R)

31 Rozhodovací strom (specializace UI-SU)

Máme data se třemi atributy A, B a C a cílovou třídou Y uvedená v tabulkách níže.

1. Na trénovacích datech uvedených v tabulce popište algoritmus tvoření rozhodovacího stromu a strom postavte (plný, bez omezení na počet dat v listech a při dělení).
2. Pro atribut vybraný do kořene popište výpočet kritéria, na základě kterého jste daný atribut vybrali. (Potřebujete-li logaritmy, stačí přibližně, některé aproximace jsou v tabulce.)

x	1/5	1/3	2/5	3/5	2/3	4/5
$\approx log_2(x)$	-2.32	-1.58	-1.32	-0.74	-0.58	-0.32

3. Dále popište některou standardní metodu prořezávání. Pokud metoda využívá validační data, použijte validační data v tabulce.

Trénovací data:

	A	B	C	Y
0	1	2	6	n
1	1	2	6	n
2	3	2	8	y
3	3	4	6	y
4	3	4	8	y
5	3	4	8	n
6	3	4	8	n
7	3	4	6	y

Validační data:

	A	B	C	Y
8	1	2	8	n
9	3	4	6	y
10	3	4	6	y
11	3	4	8	y

Nástin řešení Pro výběr atributu se užívá entropie nebo gini index. Entropie pro dvouhodnotovou pravděpodobnost je definovaná

$$H(a,b) = -\frac{a}{a+b}\log_2(\frac{a}{a+b}) - \frac{b}{a+b}\log_2(\frac{b}{a+b})$$

např.

$$H(2/6, 4/6) = -\frac{1}{3}\log_2(\frac{1}{3}) - \frac{2}{3}\log_2(\frac{2}{3}) \approx -\frac{-1.58}{3} - \frac{2\cdot(-0.58)}{3} = \frac{2.74}{3} \approx 0.91$$

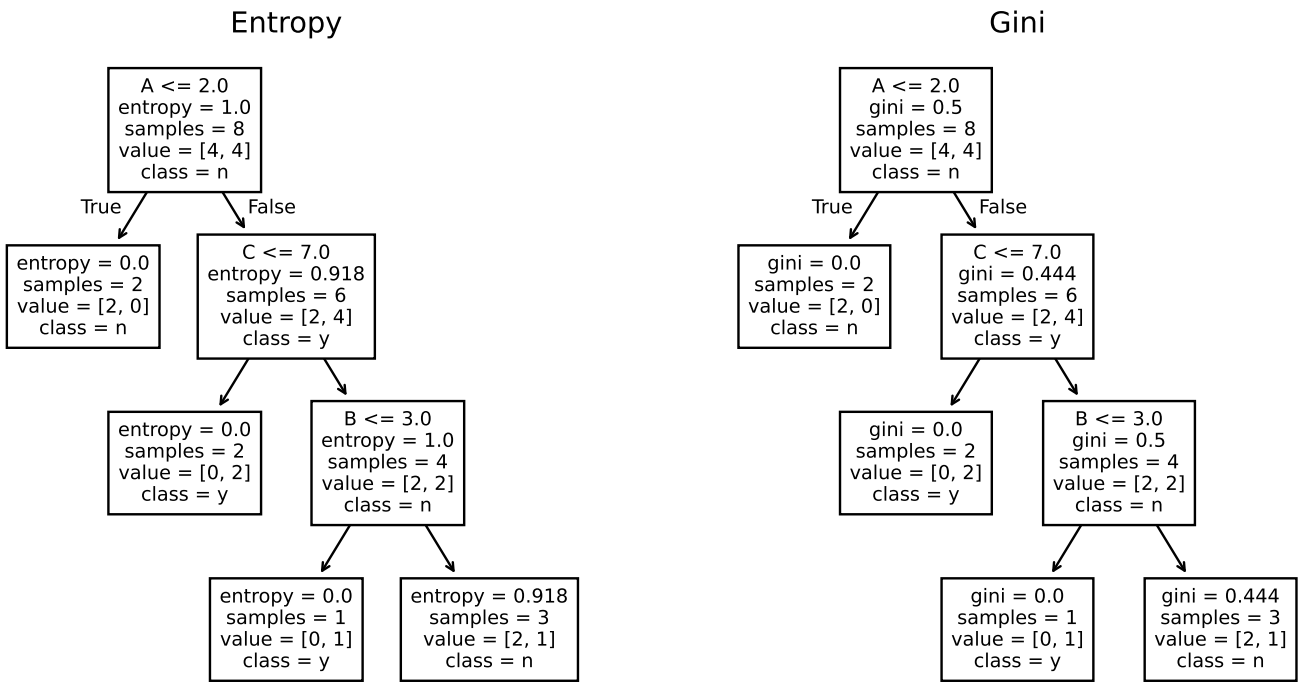
Entropický zisk je rozdíl entropie před dělením a vážené entropie po dělení; entropický zisk dělení dle A v kořeni je:

$$H(4,4) - \frac{2}{8}\cdot 0 - \frac{6}{8}H(2,4) \approx 1 - \frac{6}{8}\cdot 0.91 \approx 0.32.$$

Postavený strom vyjde stejný na základě entropie i gini, viz obrázek. Většinou je atribut k dělení zřejmý, např. dělení podle B v kořeni by vedlo k $[2, 1], [2, 3]$, což má nižší entropický zisk kvůli nenulové entropii v obou listech a více špatně klasifikovaných příkladů.

Prořezávání na základě validačních dat vyhodnotí, že je vhodné nahradit pravý uzel C listem, protože sníží validační chybu.

Alternativou může být použití cost complexity pruning.



32 Metoda TF-IDF (specializace UI-ZPJ)

Vysvětlete, k čemu se v NLP používá metoda TF-IDF. Uveďte vzorec pro výpočet obou komponent (u IDF postačí přibližný) a vysvětlete, proč jsou ve výsledném skóre obě komponenty kombinovány součinem.

Nástin řešení

- TF-IDF je statistická metoda používaná v oblasti vyhledávání informací k hodnocení důležitosti termu v rámci konkrétního dokumentu uvnitř kolekce dokumentů.
- TF (term frequency) je relativní četnost termu T v dokumentu D (tj. počet výskytů termu T v dokumentu D / celkový počet slov v dokumentu D). IDF (inverse document frequency) vyjadřuje „vzácnost" slova v kolekci dokumentů, obvykle vzorcem log(počet dokumentů/počet dokumentů obsahujících term T). IDF se v praxi vyskytuje v různých variantách s vyhlazováním nebo normalizací, podstatný je klesající průběh funkce vůči DF.
- Důležitost termu T v konkrétním dokumentu v rámci kolekce je vyjádřena součinem TF-IDF(T,D) = TF(T,D) · IDF(T), protože chceme dát nejvyšší váhu takovým termům, které jsou četné v daném dokumentu a současně nejsou běžné v kolekci dokumentů jako celku (tj. jsou informativní/specifické).