Interreg

EVROPSKÁ UNIE

Rakousko-Česká republika

Evropský fond pro regionální rozvoj

INFORMATIKA

Algoritmy a datové struktury

UNIVERSITY OF APPLIED SCIENCES UPPER AUSTRIA

VŠTE

EVROPSKÁ UNIE

Obsah

1. ALGORITMUS

Pojmem algoritmus se myslí přesný návod nebo postup, kterým lze vyřešit daný typ úlohy. Popisuje teoretický princip řešení, na rozdíl od přesného (konkrétního) zápisu v daném programovacím jazyce.

Algoritmus by měl splňovat jisté vlastnosti:

Konečnost – Každý algoritmus musí končit v konečném počtu kroků. Tento počet kroků může být libovolně velký, ale pro každý jednotlivý vstup musí být konečný. Postup, který tuto podmínku nesplňuje, nelze nazývat algoritmem, ale jen výpočetní metodou.

Obecnost – Algoritmus neřeší jeden konkrétní problém, ale obecnou třídu obdobných problémů.

Determinovanost – Každý krok algoritmu musí být jednoznačně a přesně definován. V každé situaci musí být zřejmé, co a jak se má provést, jak má algoritmus pokračovat. Některé algoritmy nejsou determinované – obsahují prvek náhody (např.: genetické algoritmy)

Výstup (neboli resultativnost) – algoritmus má alespoň jeden výstup, veličinu, která je v požadovaném vztahu ke vstupům. Výstup tvoří odpověď na problém.

Elementárnost – algoritmus se skládá z konečného počtu jednoduchých (elementárních) kroků

1.1. Proces vývoje algoritmů

Existuje několik postupů, které slouží k navrhování algoritmů:

Metoda shora dolů – postup řešení rozkládáme na jednodušší operace, dokud nedospějeme k elementárním krokům.

Metoda zdola nahoru – z elementárních kroků vytváříme prostředky, které nakonec umožní zvládnout požadovaný problém.

Případně, **kombinace obou uvedených metod**, kdy postup shora dolů doplníme částečným krokem zdola nahoru. Například použijeme knihovnu funkcí, vyšší programovací jazyk nebo systém pro vytváření programů.

Obvyklými postupy algoritmů jsou metody: Rozděl a panuj, Hladový algoritmus, Dynamické programování a backtracking.

- **Metoda Rozděl a panuj**, dělí problém na dílčí úlohy, které musí být na sobě nezávislé. Tyto dílčí úlohy poté řeší. Často je implementován rekurzivně nebo iteračně.
- **Haldový algoritmus** se většinou používá pro řešení optimalizačních úloh. Vybírá vždy lokální minimum (maxima) ve snaze najít globální minimu (maximum). Díky tomuto postupu není vždy ideální a nemusí dosáhnout globálního minima (maxima)
- **Dynamické programování** dělí problém na dílčí úlohy, obdobně jako metoda Rozděl a panuj, ale v tomto případě mohou být tyto části závislé.
- **Backtracking**, neboli hledání s návratem je způsob řešení algoritmických problémů založený na prohledávání stavového stromu problému. Je to vylepšení řešení hrubou silou (brute-force)

1.2. Typy algoritmů

Algoritmy lze rozdělit na několik druhů:

- **Rekurzivní algoritmy**, které využívají (volají) samy sebe.
- **Pravděpodobnostní (probabilistické),** které provádějí některá rozhodnutí náhodné (pseudonáhodné) volby.
- **Paralelní algoritmy**, které dělí úlohy mezi více počítačů (procesorů, vláken).
- Dalším typem jsou **genetické algoritmy** pracující na základě napodobení biologických evolučních procesů.
- A **heuristické algoritmy**, které nehledají přesné konkrétní řešení, ale jen nějaké vhodné přiblížení. Používají se v situacích, kdy dostupné zdroje nepostačují na využití exaktních algoritmů, nebo nejsou žádné vhodné exaktní algoritmy vůbec známy.

2. ABSTRAKTNÍ DATOVÝ TYP – ADT

2.1. Popis

Abstraktní datový typ – ADT – je výraz používající se na pro typy dat, která jsou nezávislá na vlastní implementaci.

Používáním ADT se snažíme o zjednodušení a zpřehlednění programu, který provádí operace s daným datovým typem.

Každý ADT je realizovatelný pomocí základních algoritmických operací, jako je přiřazení, sčítání, násobení, podmíněný skok atd.

2.1. Vlastnosti ADT

ADT má mít následující vlastnosti:

- **Všeobecnost implementace** – jednou navržený ADT může být zabudován a bez problémů používán v jakémkoliv programu.
- **Přesný popis** – propojení mezi implementací a rozhraním musí být jednoznačné a úplné.
- **Jednoduchost** – uživatel se nemusí starat o vnitřní realizaci a správu ADT v paměti.
- **Zapouzdření** – rozhraní jako uzavřená část, uživatel ví, co ADT dělá, ale ne jak to dělá
- **Integrita** – uživatel nemůže zasahovat do vnitřní struktury dat
- **Modularita** – „stavebnicový“ princip programování je přehledný a umožňuje snadnou výměnu části kódu. Při hledání chyb mohou být jednotlivé moduly považovány za kompaktní celky. Při zlepšování ADT není nutné zasahovat do celého programu.

Pokud je ADT programován objektově, pak jsou většinou tyto vlastnosti implicitně splněny.

2.1. Operace v ADT

S ADT lze provádět základní typy operací. Mezi ně patří Konstruktor, Selektor a Modifikátor.

- **Konstruktor** vytváří novou hodnotu ADT, stará se o sestavení platné vnitřní reprezentace hodnoty na základě dodaných parametrů.
- **Selektor** má na starosti získání hodnot, které tvoří složky nebo vlastnosti konkrétní hodnoty ADT.
- A **modifikátor** provádí změny hodnot datového typu.

3. ANALÝZA ALGORITMŮ

3.1. Porovnání efektivity procesů

Jelikož stejný problém lze obvykle řešit několika různými postupy (algoritmy), je proto potřeba mít nějaký nástroj, který nám umožní porovnávat efektivnost daného postupu. Buď můžeme algoritmy porovnávat **experimentálně**, nebo **teoreticky**.

Experimentální analýze je časově náročná. Časová náročnost se samozřejmě zvyšuje s množstvím a velikostí vstupů. Tento typ analýzy vyžaduje konkrétní implementaci algoritmu, což samozřejmě sebou nese nároky na další znalosti. Těžko se hledá nějaký průměrný případ. Proto je lepší soustředit se na nejhorší možný případ, který se lehce analyzuje a je kritický pro většinu aplikací, ať už se jedná o hry, finance, robotiku, automatické operace.

Experimentální analýza časové náročnosti probíhá v prostředí, kde běží daná implementace algoritmu (program), většinou za pomoci nějaké interní funkce pro měření času. Běh programu závisí na vstupech a jejich složení, navíc ne všechny vstupy jsou zahrnuty v každém běhu programu. Porovnání dvou algoritmů vyžaduje stejný hardware i software a stejné obsazení paměti,

Místo experimentální analýzy lze použít jisté **teoretické postupy**. Teoretická analýza využívá popis algoritmu pomocí operací namísto konkrétní implementace. Bere do úvahy všechny vstupy a umožňuje ohodnotit rychlost algoritmu nezávisle na hardware nebo software.

Jedním z těchto nástrojů je tzv. **Pseudo-kód**. Pseudo-kód umožňuje a využívá vyšší úrovně popisu algoritmů. Popis je více strukturovaný než běžně psaný text, ale méně detailní než konkrétní implementace. Je to preferovaný zápis pro popis algoritmů. Jeho výhodou i nevýhodou je, že skrývá problémy konkrétní implementace.

Pseudo-kód využívá klíčová slova pro popis algoritmu:

Pro řízení běhu: -If...then...else (condition), while...do, repeat...until, for...do (cycles)

Hlavička: Algorithmus `Name(arg1, arg2...)`, input, output

Volání procedury (Methoden, Algorithmus): `var.Name(arg1,arg2,...)`

Návrat hodnoty: `return` *`Expression`*

Výrazy:	←	Zuschreibung
	=	Gleichheit
	+, -, n^2,...	Mathematische Operationen

3.1. Primitivní operace

Primitivní operace je základní operace provedená algoritmem, je identifikovatelná v pseudo-kódu, nezávislá na programovacím jazyku a měla by být přesně definovaná. Takovou primitivní operací může být třeba vyhodnocení výrazu, přiřazení hodnoty do proměnné, indexování v poli, volání nebo návrat z procedury a podobně.

Podobně můžeme využít **asymptotickou notaci (big O, Bachmann-Landau notace).** Určuje operační náročnost algoritmu tak, že zjišťuje, jakým způsobem se bude chování algoritmu měnit v závislosti na změně velikosti (počtu) vstupních dat. Obvykle se používá asymptotická časová a prostorová složitost. Používaný zápis znamená, že náročnost algoritmu je menší než $A+B \cdot f(N)$, kde A a B jsou vhodně zvolené konstanty a N je veličina popisující velikost vstupních dat. Zanedbáváme tedy multiplikativní i aditivní konstanty, tzn. $O(N+1000)=O(1000 \cdot N)=O(N)$. Zajímá nás jen chování funkce pro velké hodnoty N.

Pro určení časové náročnosti algoritmu pomocí big O notace musíme nalézt největší možný počet primitivních operací, který pak vyjádříme pomocí big O notace.

4. FRONTY A ZÁSOBNÍKY

4.1. Zásobník

Zásobník je datová struktura, která slouží pro ukládání dat. Je charakterizován způsobem manipulace s daty – k datům přistupuje pomocí principu LIFO (**Last In First Out**). Lze si ho představit jako zásobník na talíře.

ADT zásobník musí obsahovat minimálně operace pro:

- vložení objektu,
- vrácení a odebrání posledního objektu,
- dotaz na vrchol zásobníku,
- jeho velikost,
- jestli je zásobník prázdný.

Pokud se pokusíme provést na prázdném zásobníku operaci pop, nebo top, dostaneme výjimku *EmptyStackException*.

Aplikací zásobníku je například historie prohlížeče webových stránek, Undo sekvence v editorech nebo řetězec volání jednotlivých procedur. Zásobník lze využít jako pomocnou datovou strukturu pro jiné algoritmy, případně jako část jiných datových struktur.

Zásobník lze implementovat nejjednodušeji pomocí pole. Prvky přidáváme zleva doprava a pomocná proměnná drží index posledního prvku.

Díky vlastnostem pole získáme následující vlastnosti:

- ***n*** – počet prvků v zásobníku
- Paměťová náročnost - ***O(n)***
- Časová náročnost každé operace – ***O(1)***

Pole nám však přináší také jistá omezení:

- Na začátku musíme definovat velikost zásobníku
- Velikost zásobníku nelze jednoduše změnit
- Přidání prvku do plného zásobníku vyvolá výjimku specifickou pro implementaci

4.2. Fronta

Fronta je datová struktura typu **FIFO** (**First In First Out**).

Minimální implementace fronty musí obsahovat operace pro:

- vložení položky na konec fronty,
- výběr položky ze začátku fronty,
- dotaz na začátek fronty,
- její délky a obsazenost.

Stejně jako zásobník i fronta může vyhodit výjimku při operaci *dequeue* nebo front nad prázdnou frontou – *EmptyStackException*.

Aplikací fronty je například pořadník, fronta, přístup ke sdíleným zdrojům (tiskárny), multiprogramování. Frontu lze využít také jako pomocnou datovou strukturu pro jiné algoritmy, případně jako část jiných datových struktur.

Frontu lze implementovat pomocí pole. Pro zlepšení vlastností se využívá kruhového pole. Máme pak dvě proměnné, *f* – index prvního prvku a *r* index posledního prvku zvětšený o jedna (ukazuje na první volné místo).

5. VEKTORY, SEZNAMY, SEKVENCE

5.1. Vektory

Vektor rozšiřuje pojem pole ukládáním sekvence libovolných objektů. Prvek ve vektoru může být čten, vkládán a odebírán pomocí určení jeho pořadí.

Vektor umožňuje **základní operace**:

- prvek na určitém pořadí,
- záměnu prvku na konkrétním místě,
- vložení na konkrétní místo a odebrání prvku z konkrétního místa,
- zjistit velikost, a jestli je daný vektor prázdný.

Operace s vektory mohou vyhodit výjimku v případě nesprávného indexu (obvykle záporného). Aplikací vektoru je tříděná kolekce objektů (základní databáze).

Vektor lze implementovat pomocí pole. To nám pak přináší následující vlastnosti:

- Proměnná ***n*** určuje délku vektoru
- Operace *isEmpty(), elemAtRank(r), replaceAtRank(r, O)* - časová náročnost *O(1)*
- Operace *insertAtRank(r, O)* - časová náročnost *O(n)*
- Operace *removeAtRank(r)* - časová náročnost *O(n)*

5.2. Seznamy

Další datovou strukturou je **Seznam**. Seznam je posloupnost pozic ukládající libovolná data. Zavádí vztahy před/po mezi pozicemi.

Obecnými operacemi jsou dotaz na velikost, a dotaz na prázdný seznam. Další operace jsou dotaz, jestli je daný prvek první nebo poslední, získání prvního a posledního prvku, dále pak prvek předcházejí, či následující.

ADT seznam obsahuje operace:

- replaceElement(p, o),
- swapElements(p, q),
- insertBefore(p, o),
- insertAfter(p, o),
- insertFirst(o),

- insertLast(o),
- remove(p).

Seznamy lze rozdělit na single linked list – jednosměrný spojový seznam - a double linked list – obousměrný spojový seznam.

V jednosměrném seznamu prvek obsahuje odkaz na následující uzel, v případě obousměrného seznamu prvek obsahuje odkaz i na předcházející uzel.

5.3. Sekvence

ADT **Sekvence** je spojením vektoru a seznamu, k prvkům tak lze přistupovat jak pomocí pozice, tak i pořadí. Kromě vektorových a seznamových operací obsahuje i propojující operace **atRank(r)** a **rankOf(p).**

Sekvence je obecný základní typ použitelný pro ukládání uspořádaného souboru prvků. Je obecnou náhradou za zásobník, frontu, vektor nebo seznam. Lze ji použít i jako malou databázi.

6. STROMY

Strom představuje model hierarchické struktury, která se skládá z uzlů, mezi nimiž je vztah rodič-dítě.

Strom lze použít jako organizační diagram, souborové systémy, či programovací prostředí.

Pro popis stromů a jejich částí se využívá následující terminologie:

6.1. Druhy uzlů

- kořen (root),
- vnitřní uzel (inner node) - uzel, který není kořenem, ani listem,
- list (leaf node, external node) - uzel, který nemá žádné potomky,
- rodič (parent node) - uzel, který přímo předchází daný uzel na cestě od listu ke kořeni,
- potomek (child node) - uzel, který přímo následuje za daným uzlem na cestě od kořene k listu,
- sourozenec (sibling) - jako sourozenci se označují uzly se stejným rodičem,
- předek (ancestor node, predecessor node) - uzel, který leží před daným uzlem na cestě ke kořeni (nejbližší předek je rodič),
- následník (successor node) - uzel, který leží za daným uzlem na cestě od kořene k libovolnému listu (nejbližší následník je potomek),
- hloubka (depth) - hloubka stromu je délka nejdelší cesty od kořene k listu, přičemž prázdný strom má definovánu hloubku jako -1,
- úroveň (level) - většinou se používá ve významu množiny uzlů, které se nachází ve stejné vzdálenosti od kořene, počítáno dle počtu uzlů.

6.2. Struktura

- podstrom (subtree) - podgraf stromu, který je také stromem (obecně se nejčastěji setkáváme s podstromy tvořenými tak, že se vezme nějaký uzel stromu jako nový kořen a zbytek struktury se zachová),
- větev (branch) - každá cesta od kořene k listu.

6.3. Operace pro manipulaci se stromy

Obecné operace:

- integer size(),
- boolean isEmpty(),
- objectIterator elements(),
- positionIterator position().

Přístupové operace

- position root(),
- position parent(),
- positionIterator children(p),
- Dotazovací operace,
- boolean isInternal(p),
- boolean isExternal(p),
- boolean isRoot(p).

Aktualizační operace

- swapElements(p, q),
- object replaceElement(p, o).

Jelikož je strom hierarchickou strukturou, lze jej procházet několika způsoby.

Pre-order průchod

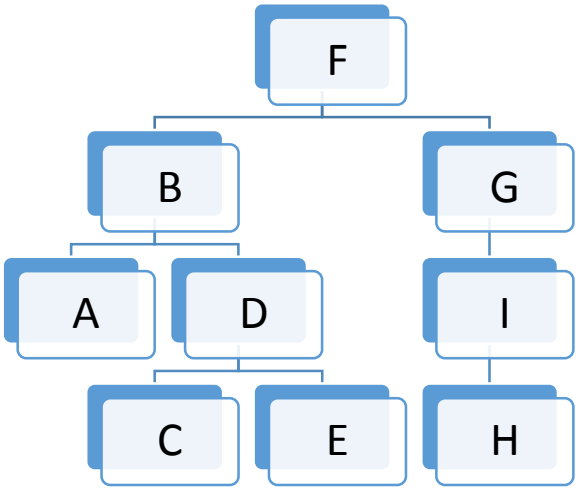
- Kontrola jestli je uzel prázdný nebo null,
- Zobrazení dat aktuálního uzlu,
- Průchod levým podstromem rekurzivním voláním pre-order funkce,
- Průchod pravým podstromem rekurzivním voláním pre-order funkce.

In-order průchod

- Kontrola jestli je uzel prázdný nebo null,
- Průchod levým podstromem rekurzivním voláním in-order funkce,
- Zobrazení dat aktuálního uzlu,
- Průchod pravým podstromem rekurzivním voláním in-order funkce.

Post-order průchod

- Kontrola jestli je uzel prázdný nebo null,
- Průchod levým podstromem rekurzivním voláním post-order funkce,
- Průchod pravým podstromem rekurzivním voláním post-order funkce,
- Zobrazení dat aktuálního uzlu.



Každý typ průchodu nám poskytuje jiné výsledky.

- Pre-order: F, B, A, D, C, E, G, I, H
- In-order: A, B, C, D, E, F, G, H, I
- Post-order: A, C, E, D, B, H, I, G, F

Pomocí ADT strom lze definovat i Binární strom, či jiné další typy.

Binární strom rozšiřuje definici stromu o to, že každý uzel má nejvýše dvě děti, které tvoří uspořádanou dvojici (levý potomek, pravý potomek).

Binární strom přidává další operace:

- **position leftChild(p)**
- **position rightChild(p)**
- **position sibling(p)**

7. PRIORITNÍ FRONTA A HALDA

7.1. Prioritní fronta

Prioritní fronta uchovává kolekci položek, kde položka je uspořádanou dvojicí klíč (priorita)-hodnota.

Základní implementace by měla obsahovat operace:

- **insertItem(k, o),**
- **removeMin(),**
- **minKey(k, o),**
- **minElement(),**
- **size(),**
- **isEmpty().**

Aplikací prioritní fronty jsou např.: aukce a burzy.

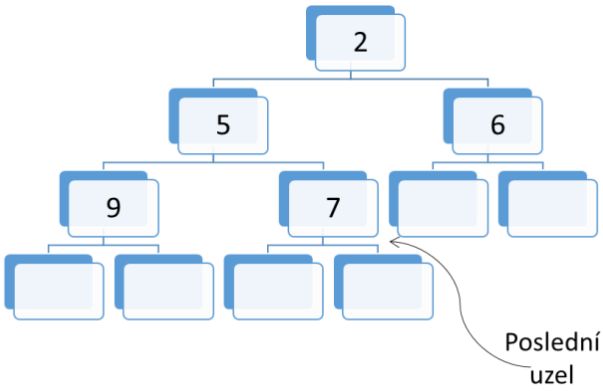
Klíčem prioritní fronty mohou být libovolné objekty, na nichž lze definovat pořadí a lze je uspořádat. Dva rozdílné prvky (hodnoty) mohou mít stejný klíč (prioritu).

Pro použití prioritní fronty je nutné zavést ADT Comparator, který nám umožní porovnávat dva objekty.

7.2. Halda (Heap)

Halda (Heap) je binární strom, který uchovává klíče jako interní uzly. Pro každý uzel stromu mimo kořen platí, že klíč uzlu je větší než klíč jeho rodiče. Pro haldu definujeme kompletní binární strom. Nechť *h* je výška stromu, pak pro *i* do 0 do *h-1* je 2^i uzlů hloubky *i*.

Poslední uzel haldy, je vnitřní uzel, který se nachází nejvíce vpravo na úrovni *h-1*.



Haldu lze použít pro implementaci prioritní fronty. Pak ukládáme položku (klíč, hodnota) v každém interním uzlu a uchováváme odkaz na pozici posledního prvku.

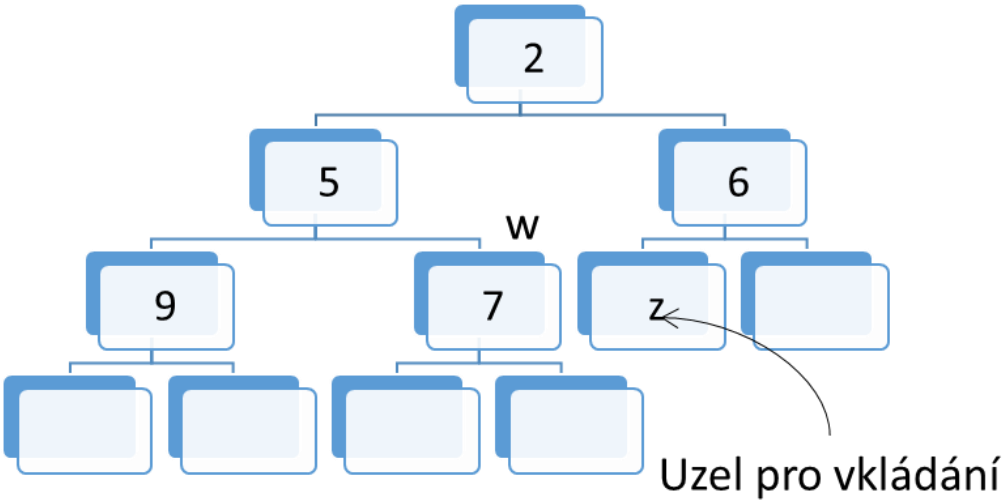
7.3. Manipulace s haldou:

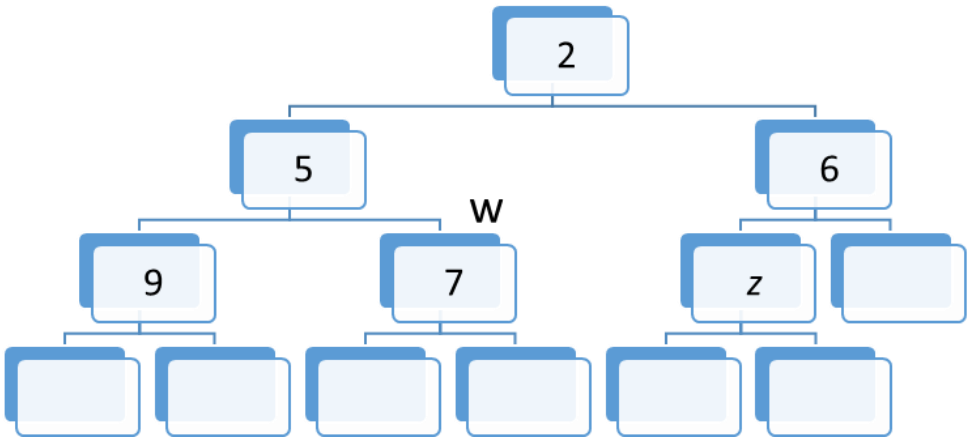
Operace insertItem(k, o)

- Nalezení uzlu, kam se bude vkládat
- Uložení klíče k do uzlu z, změna uzlu z na interní uzel
- Obnovení uspořádanosti haldy (kontrola vlastností) – operace upheap()

Operace removeMin()

- Odpovídá odebrání kořene z hromady (uzel 2)
- Výměna kořene za poslední uzel (2 – 7)
- Změna uzlu w a jeho dětí na list
- Obnovení uspořádanosti haldy – operace downheap()





upheap()

- Vložení uzlu může narušit uspořádání.
- Algoritmus *upheap* obnoví uspořádání prohazováním klíče *k* vzhůru od vloženého uzlu.
- Končí ve chvíli, kdy se vložený uzel stane kořenem, nebo když rodičovský klíč je roven nebo menší než *k.*

downheap()

- Odebrání kořene může narušit uspořádání.
- Algoritmus *downheap* obnoví uspořádání prohazováním klíče *k* dolů od kořene.
- Končí ve chvíli, kdy se vložený uzel stane listem, nebo když klíč potomka je roven nebo větší než *k.*

8. SLOVNÍKY A HASHOVACÍ TABULKY

8.1. Slovník

Jako **Slovník** označujeme ADT obsahující kolekci klíč-hodnota, ve které lze vyhledávat. Operace, které lze se slovníkem provádět:

- **findElement(k),**
- **insertItem(k, o),**
- **removeElement(k),**
- **size(),**
- **isEmpty(),**
- **keys(),**
- **elements()**

Aplikací slovníku je adresář, autorizace kreditních karet, slovník, překlad doménového jména na IP adresu.

Log File – slovník implementovaný jako neuspořádaná sekvence (double linked list). Objekty ukládáme v libovolném pořadí.

Náročnost operací:

- Vložení objektu O(1)
- Nalezení prvku, odebrání prvku O(n)

Log file je vhodný pro malé slovníky, nebo aplikace, kde je vkládání nejčastější operací, zatímco vyhledávání a odebírání se provádí zřídka.

Operace **findElement(k)** na slovníku implementovaném pomocí sekvence založené na poli uspořádaném podle klíčů, se provádí jako binárním vyhledávání. V každém kroku je číslo kandidáta dělené dvěma, končí po logaritmickém počtu kroků.

Vyhledávací tabulka je Slovník implementovaný pomocí uspořádané sekvence. Uchováváme položky slovníku v sekvenci založené na poli uspořádané podle klíčů, je nutný externí comparator pro klíče.

Náročnost operací:

- Nalezení prvku O(log(n))
- Vložení prvku, odebrání prvku O(n)

Efektivní pro malé slovníky, nebo aplikace, kde se nejčastěji provádí vyhledávání.

Binární vyhledávací strom je binární strom, pro který platí:

- u, v a w jsou tři uzly takové, že u je v levém podstromu v a w je v pravém podstromu v
- $key(u) \leq key(v) \leq key(w)$
- Externí uzly neuchovávají žádnou položku
- In-order průchod dává klíče v zvyšujícím se pořadí.

8.2. Hashovací tabulka

Hash funkce *h*, je taková funkce, která přiřazuje klíči daného typu celočíselnou hodnotu z intervalu od 0 do N-1. Cílem této funkce je uniformě rozdělit klíče v daném intervalu.

Hashovací tabulka pro daný typ klíče obsahuje hash funkci a pole (tabulku) o velikosti N.

Klíč se nahradí hash hodnotou. Může se však stát, že pro dva klíče se vygeneruje stejná hash hodnota – dojde ke kolizi. To lze řešit v zásadě dvěma způsoby:

- zřetězení (chaining) – kdy se kolidující položky ukládají jako sekvence,
- otevřené adresování – položka se uloží na jiné místo v tabulce.

9. ŘADÍCÍ ALGORITMY I.

9.1. Vysvětlení

Řadící algoritmy, někdy nesprávně označované jako třídící algoritmy, slouží k seřazení daného souboru dat do specifického pořadí, ať už abecedně nebo podle čísel. Dvojice klíč-hodnota se řadí podle klíče a na hodnotu není brán zřetel.

Řadící algoritmy lze rozdělit na stabilní a nestabilní, podle toho, zda zachovávají pořadí položek se stejným klíčem, přirozené a nepřirozené – přirozený pracuje rychleji s částečně seřazenou množinou.

Lze je také rozdělit podle typu řazení:

- Výběrem
- Vkládáním
- Záměnou
- Slučováním

Nejznámější algoritmy – typ algoritmu je obsažený v jeho pojmenování

- Bubble sort Bublinkové řazení
- Heap sort Řazení haldou
- Insertion sort Řazení vkládáním
- Merge sort Řazení slučováním
- Quicksort Rychlé řazení
- Selection sort Řazení výběrem

Další algoritmy založené na jiném principu

- Bucket sort Přihrádkové řazení
- Radix sort Třídění podle základu
- Counting sort Řazení počítáním četnosti

9.2. Bubble sort

- Implementačně jednoduchý algoritmus
- Opakovaně prochází seznam a porovnává dva sousední prvky
- Univerzální, pracuje lokálně (nepotřebuje dodatečnou paměť)
- Prvky s nejvyšší hodnotou probublávají na konec seznamu

Algoritmus:

```
procedure bubbleSort( A : list sortable items )
    n = length(A)
    repeat
        swapped = false
        for i from 1 to n-1 inclusive do
     if A[i-1] > A[i] then
                        swap( A[i-1], A[i] )
                swapped = true
            end if
        end for
    until not swapped
end procedure
```

9.3. Heap sort

- Jeden z nejlepších obecných algoritmů založených na porovnávání prvků
- Není stabilní
- Využívá datovou strukturu halda a její vlastnosti

Algoritmus:

```
procedure heapsort(a, count) is
    input: an unordered array a of length count
    heapify(a, count)
   end ← count - 1
    while end > 0 do
         swap(a[end], a[0])
        (the heap size is reduced by one)
        end ← end - 1
        (the swap ruined the heap property, so restore it)
        shiftDown(a, 0, end)
```

Nejmenší prvek je kořenem – umístíme na první místo v poli a kořen odebereme

downheap() – obnovení haldy podle pravidel. Opakujeme odebírání kořene a obnovení haldydokud halda není prázdná

9.4. Insertion sort

- Jednoduchá implementace
- Efektivní na malých množinách
- Efektivní na částečně seřazených množinách
- Stabilní
- Dokáže řadit data tak, jak přicházejí na vstup
- Postup:
    - Posloupnost rozdělíme na seřazenou a neseřazenou tak, že seřazená obsahuje první prvek posloupnosti.
    - Z neseřazené části vybereme první prvek a zařadíme jej na správné místo v seřazené posloupnosti.
    - Prvky v seřazené posloupnosti posuneme o jednu pozici doprava.

- Seřazenou část zvětšíme o jeden prvek. Naopak neseřazenou část o jeden prvek zleva zmenšíme.
- Kroky 2–5 aplikujeme až do úplného seřazení neseřazené části.

Algoritmus

```
for i = 1 to length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
    end while
end for
```

9.5. Merge sort

- metoda Rozděl a panuj
- Nejhorší i průměrná časová složitost O(NlogN)
- Potřebuje navíc pole o velikosti N
- Stabilní, paralelizovatelný
- Postup:
    - Rozdělí neseřazenou množinu dat na dvě podmnožiny o přibližně stejné velikosti.
    - Seřadí obě podmnožiny.
    - Spojí seřazené podmnožiny do jedné seřazené množiny.

Algoritmus

```
mergesort(m)
      var list left, right
      if length(m) ≤ 1
            return m
      else
            middle = length(m) / 2
            for each x in m up to middle
                  add x to left
            for each x in m after middle
                  add x to right
left = mergesort(left)
            right = mergesort(right)
            result = merge(left, right)
      return result
```

9.6. Quicksort

- Jeden z nejrychlejších běžných algoritmů řazení založený na porovnávání prvků
- Časová složitost O(NlogN) – O(N2)
- Metoda Rozděl a panuj
- Rekurzivní algoritmus

- Postup:
  - výběr pivot – rozdělení posloupnosti na dvě části – větší a menší než pivot
  - Seřaď obě části stejným způsobem
- Volba pivotu – ideální medián
  - První prvek (jakákoli fixní pozice) – nevýhodné na částečně seřazených množinách
  - Náhodný prvek – ve skutečnosti pseudonáhodný
  - Medián tří (pěti…) – či jiného počtu prvků z fixních nebo náhodných pozic

Při správné implementaci nepotřebuje paměť navíc. Quicksort je nestabilní algoritmus. Způsob volby pivotu má vliv na řazení, ale v průměru jde o nejrychlejší známý univerzální algoritmus pro řazení polí v operační paměti počítače.

9.7. Selection sort

Jednoduchý algoritmus, jehož časová složitost je $O(N^2)$, je vhodný pro malé množství dat. Je univerzální, lokální a nestabilní.

Postup:

- Rozdělíme si posloupnost na seřazenou a neseřazenou část
- Najdeme prvek s nejmenší hodnotou v neseřazené části posloupnosti
- Zaměníme ho s prvkem na první pozici neseřazené části
- První prvek neseřazené části zahrneme do seřazené části a zároveň neseřazenou část zmenšíme o 1 prvek zleva
- Zbytek posloupnosti se uspořádá opakováním kroků 2 až 5 pro zbylou neseřazenou část

Porovnání řadících algoritmů:

Název		Časová složitost			Dodatečná paměť	Stabilní	Přirozená	Metoda
Anglicky	Česky	Minimum	Průměrně	Maximum				
Bubble sort	Bublinkové řazení	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano	záměna
Heapsort	Řazení haldou	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	ne	ne	halda, záměna
Insertion sort	Řazení vkládáním	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	ano	ano	vkládání
Merge sort	Řazení slučováním	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	ano	ano	slučování
Quicksort	Rychlé řazení	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	ne	ne	záměna
Selection sort	Řazení výběrem	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	zprav. ne	ne	výběr

9.8. Bucket sort

Rozděluje data do několika přihrádek, jeho časová náročnost je O(n*k), kde k=n/m, vstupní data n, počet přihrádek m.

Pro použití Bucket sortu jsou nutné předpoklady:

- Vhodný pro rovnoměrně rozložené hodnoty vstupních dat.
- Algoritmus pro seřazení přihrádek musí být stabilní

Postup:

- Vstupní data jsou rozdělena do předem definovaného počtu přihrádek.
- Na každou přihrádku volán stabilní řadicí algoritmus.
- Jednotlivé přihrádky postupně kopírovány do výstupního pole.

Výhody bucket sortu jsou, že je dobře paralelizovatelný a nemusí mít všechna data v paměti najednou.

9.9. Radix sort

Řadí celá čísla procházením všech číslic. Existují dva přístupy:

- LSD (Least Significant Digit) – řazení podle nejméně významných číslic (odzadu)
- MSD (Most Significant Digit) – nejvíce významné číslice (odpředu)

Časová složitost: O((z+n)*$\log_z u$), kde z je základ zvolené číselné soustavy, *n* počet čísel na vstupu a *u* je maximální rozmezí čísel na vstupu

Nehodí se pro neomezeně velké vstupy.

Příklad LSD radix: 170, 45, 75, 90, 802, 2, 24, 66 ⇒ 17<u>0</u>, 9<u>0</u>, 80<u>2</u>, <u>2</u>, 2<u>4</u>, 4<u>5</u>, 7<u>5</u>, 6<u>6</u> ⇒ 8<u>0</u>2, 2, 2<u>4</u>, <u>4</u>5, <u>6</u>6, 1<u>7</u>0, <u>7</u>5, <u>9</u>0 ⇒2, 24, 45, 66, 75, 90, <u>1</u>70, <u>8</u>02

9.10. Counting sort

Vhodný pro velké soubory s malým množstvím diskrétních hodnot, je stabilní. Jeho časová náročnost je *O(N+M)* a paměťová náročnost *O(M.)*

Předpoklady:

- Počet různých hodnot (M) významně menší než celkový počet prvků (N).
- Pomocné pole – zápis a čtení v konstantním čase (pole indexované hodnotou nebo hashem hodnoty)

Algoritmus:

- zleva (či zprava) projde vstupní pole
- pro každý prvek zvýší v pomocném poli četnost výskytu tohoto prvku
- ke každé položce přičte počet výskytů všech předchozích položek (získá přesnou pozici hranice)
- začne zprava procházet neseřazené pole
- pro každý prvek se podívá do pomocného pole na horní hranici pro umístění
- na tuto hranici ho umístí a zároveň ji sníží o jedna
- takto postupuje, dokud neprojde celé pole

10. PATTERN MATCHING

Pattern matching, neboli porovnávání vzorů, je vyhledávání jistého vzoru v dané sekvenci. Obvykle hledání podřetězce v řetězci.

Nejprve je nutné si říct, co je to řetězec. Řetězec (string) je sekvence znaků z dané abecedy. Abeceda je množina všech možných znaků – Ascii, Unicode, {0,1}, {A, C, G, T}

Pokud je *P* řetězec délky *m*, pak podřetězec *P[i..j]* řetězce *P* se skládá ze znaků mezi *i* a *j*. Řetězec nacházející se před indexem *i* je předpona – prefix. Řetězec nacházející se za indexem *j* je přípona – suffix.

Aplikace – textové editory, vyhledávací nástroje, biologický výzkum.

Pro pattern matching existuje několik algoritmů.

Základním z nich je **Brute-Force** (hrubá síla).

Prochází text zleva doprava

Porovnává vzor *P* s textem *T*, pro všechny možné pozice dokud:

- Není nalezena shoda
- Nebyly vyzkoušeny všechny možné pozice

Časová náročnost toho algoritmu je *O(nm).*

Boyer-Moorův algoritmus prochází text od konce (zprava doleva).

Definujeme: Index *i* ukazuje do textu *T*, index *j* ukazuje do *P*

V průběhu vyhledávání mohou nastat 4 případy:

- T(i) není v P vůbec, posuneme i dále o délku P (zarovnáme P na další písmeno v T, tedy T(i+1))
- T(i) odpovídá P(j) - posuneme se v obou do leva a opakujeme (jako v brutální síle)
- T(i) není P(j), ale T(i) je v P před indexem j -> zarovnáme P doprava tak, aby T(i) odpovídal jeho výskytu v P a opakujeme
- T(i) není P(j), ale T(i) je v P za indexem j -> posuneme se doprava o 1 a opakujeme (nemůžeme se vracet, ale ani posunout o více dále)

Vrátíme *i* - pokud nalezneme celý pattern

Algoritmus je rychlejší než Brute-Force, přesto jeho složitost může být *O(mn + A),* kde A je velikost abecedy.

Pro aplikaci tohoto algoritmu je nutný preprocessing:

- zjistí, na které pozici má které písmena (směrem zleva)
- Je-li patern např.: „ABRAKADABRA„
- A dostane index 10, B dostane index 8, K = 4, D = 6 a R = 9. Pro ostatní písmena přiřadíme -1.
- Naimplementujeme tedy funkci Last(char input), která podle písmene vrátí tento index, a pak pro případy 3 a 4 porovnáváme Last(*T(i)*) a *j*, a tím pak víme, jestli i posunout na Last(*T(i)*) (pokud je Last(*T(i)*) < *j*) nebo pouze *i*++

Knuth-Morris-Pratt (KMP) algoritmus

Prohledává text zleva doprava, na rozdíl od brute-force nedělá všechna porovnávání. Pokud narazí na neshodu, posune se o více než jedno písmeno. Najdeme-li kus P (od začátku, tedy prefix), znaky tohoto prefixu odpovídají textu, není třeba je kontrolovat znovu. Konec nalezeného podřetězce může být také obsažen v začátku tohoto podřetězce. Takovou shodou je samozřejmě celá nalezená část P, proto hledáme od P+1. Takže jdeme od konce nalezeného kusu P zleva a zprava, a ve chvíli, kdy nenalezneme shodu, víme, o kolik se můžeme posunout. Toto lze předpočítat do tabulky – pak je vše O(1).

Trie

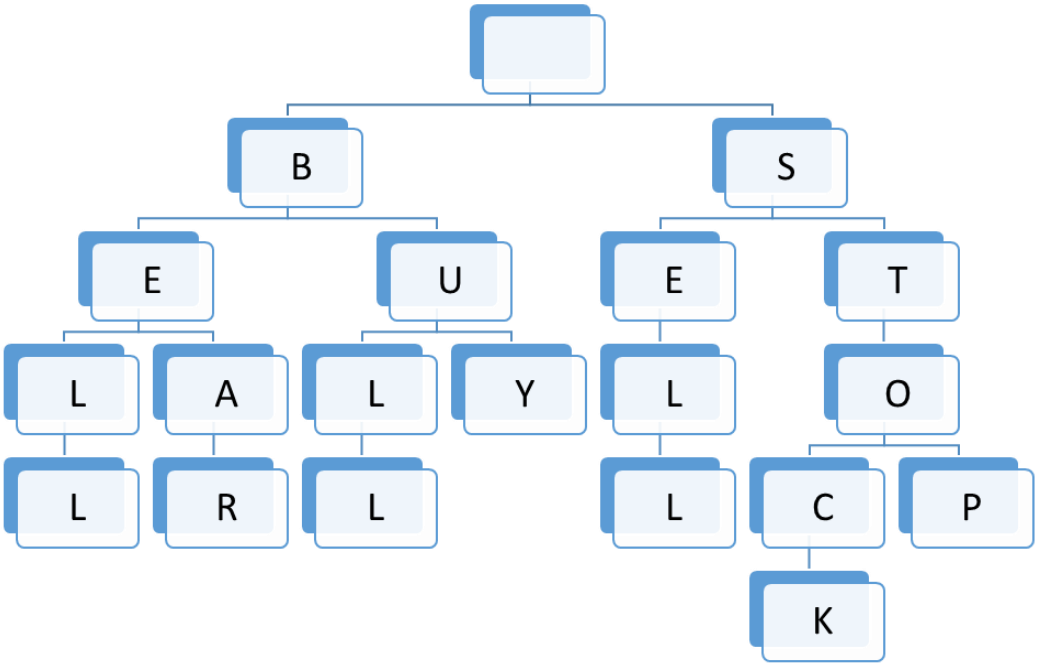
Trie je stromová struktura pro předzpracování textu. V každém uzlu je jedno písmeno. Délka cesty z uzlu k vrcholu určuje pořadí písmene ve slovu.

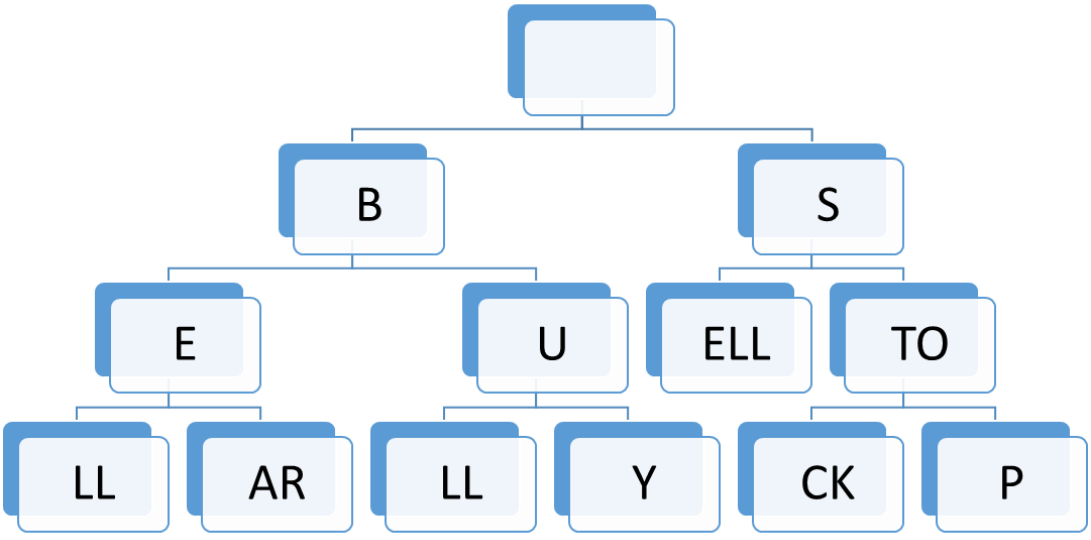
Vyhledávání má časovou náročnost O(dm), kde d je velikost abecedy, m délka slova.

Do struktury Trie je možné ukládat i celý text, pak je v každém uzlu jedno slovo.

Pro úsporu lze definovat tzv. komprimovaný Trie. Strom pak obsahuje uzly alespoň stupně dva (dvě písmena v jednom uzlu).

Příklad: S={BELL, BEAR, BULL, BUY, SELL, STOCK, STOP}


B
S
E
U
E
T
L
A
L
Y
L
O
L
R
L
L
C
P
K


B
S
E
U
ELL
TO
LL
AR
LL
Y
CK
P

11. TEORIE GRAFŮ

Graf tvoří uspořádaná dvojice *(V, E)*, kde *V* je množina vrcholů a *E* je množina hran. Každá hrana je určena právě dvěma vrcholy, volitelně pak směrem nebo váhou.

11.1. Typy hran

Existuje několik typů hran:

- Orientovaná – uspořádaná dvojice vrcholů *(u,v),* kde *u* je počátek, *v* je cíl
- Neorientovaná - uspořádaná dvojice vrcholů *(u,v)*
- Smyčky – hrana začíná a končí ve stejném vrcholu
- Multihrana (násobná, paralelní, rovnoběžná) – mezi vrcholy *(u,v)* vede více hran

11.2. Typy grafů

Stejně jako máme několik typů hran, existuje i několik typů samotných grafů.

- Orientovaný – všechny hrany jsou orientované
- Neorientovaný – všechny hrany jsou neorientované
- Multigraf – obsahuje multihrany

11.3. Terminologie:

- End vertices (or endpoints) of an edge
- Edges incident on a vertex
- Adjacent vertices
- Degree of a vertex
- Parallel edges
- Self-loop
- Path
    - sequence of alternating vertices and edges
    - begins with a vertex
    - ends with a vertex
    - each edge is preceded and followed by its endpoints
- Simple path
- path such that all its vertices and edges are distinct
- Cycle
    - circular sequence of alternating vertices and edges

    - each edge is preceded and followed by its endpoints
- Simple cycle
    - cycle such that all its vertices and edges are distinct

11.4. ADT Graf podporuje operace:

Přístupové operace

- aVertex()
- incidentEdges(v)
- endVertices(e)
- isDirected(e)
- origin(e)
- destination(e)
- opposite(v,e)
- areAdjecent(v,w)

Aktualizační operace

- insertVertex(o)
- insertEdge(v, w, o)
- insertDirectedEdge(v, w, o)
- removeVertex(v)
- removeEdge(e)

Obecné operace

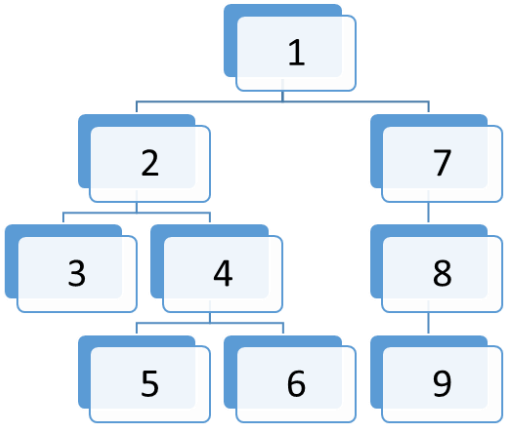
- numVertices()
- numEdges()
- vertices()
- edges()

Strukturu graf je potřeba nějakým způsobem prohledávat. Na výběr máme ze dvou možností – **DFS** (Depth-First Search – prohledávání do hloubky) a **BFS** (Breadth-First Search – prohledávání do šířky)

11.5. Depth-First Search

Prohledávání hloubky je úplný algoritmus (projde každý uzel). Jeho princip spočívá v tom, že expanduje prvního následníka každého vrcholu, pokud jej ještě nenavštívil. Pokud narazí na vrchol, z nějž už nelze dále pokračovat (nemá žádné následníky nebo byli všichni navštíveni), vrací se zpět backtrackingem.

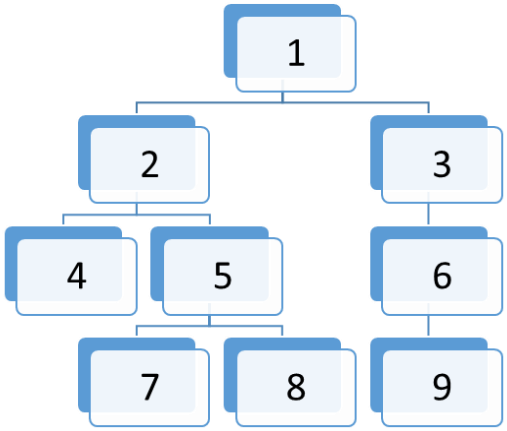
Prochází uzly v pořadí:



11.6. Breadth-First Search

Prohledávání do šířky je podobný algoritmus jako DFS, projde všechny sousedy startovního vrcholu, poté sousedy sousedů atd. až projde celou komponentu souvislosti

Prochází uzly v pořadí:



Základním otázkou z teorie grafů je nalezení nejkratší cesty mezi dvěma vrcholy. K tomu existuje několik algoritmů.

Dijkstrův algoritmus

- Konečný, funguje pouze na kladně ohodnoceném grafu
- $O(|V|^2+|E|)$ – V je počet vrcholů, E počet hran

Bellmanův-Fordův algoritmus

- Graf může obsahovat i záporné hrany
- $O(V \cdot E)$ – pomalejší než Dijsktrův alg.

Floydův-Warshallův algoritmus

- Orientovaný graf s kladnými hranami
- Nalezne nejkratší cestu mezi všemi vrcholy
- Časová náročnost – $O(V^3)$, paměťová náročnost $O(V^2)$

Johnsonův algoritmus

- Orientovaný graf, může mít i záporné hrany
- V řídkých grafech je rychlejší, než Floydův-Warshallův algoritmus
- Dokáže rozpoznat záporný cyklus v grafu a výpočet ukončit
- využívá Bellmanův-Fordův algoritmus, s jehož pomocí přehodnotí hrany tak, aby žádná neobsahovala zápornou hodnotu
- Po přehodnocení hran používá Dijkstrův algoritmus k nalezení nejkratších cest mezi všemi uzly.
- $O(V^2 \log_2(V)+ VE)$

12. GENETICKÉ ALGORITMY

12.1. Výklad

Genetické algoritmy patří mezi evoluční algoritmy a řadí se do oblasti umělé inteligence. Jsou třídou heuristických algoritmů. Využívají znalostí z evoluční biologie k řešení složitých problémů, pro které neexistuje exaktní algoritmus. Napodobuje techniky evoluční biologie:

- Dědičnost
- Mutace
- Přirozený výběr
- Křížení

Princip genetických algoritmů pracuje podle schématu:

- **Inicializace**: Vytvoř nultou populaci (obvykle složenou z náhodně vygenerovaných jedinců)
- **Začátek cyklu:** Vyber (zpravidla zčásti náhodné) z populace několik jedinců s vysokou zdatností
- Z vybraných jedinců vygeneruj novou generaci použitím následujících metod (operátorů):
    - **Křížení:** „prohoď" části několika jedinců mezi sebou
    - **Mutace:**- náhodně změň část jedince
    - **Reprodukce:** kopíruj jedince beze změny
- Vypočti zdatnost těchto nových jedinců
- **Konec cyklu**: Pokud není splněna zastavovací podmínka, tak pokračuj od bodu 2
- **Konec algoritmu**: Jedinec s nejvyšší zdatností je hlavním výstupem algoritmu a reprezentuje nejlepší nalezené řešení.

12.2. Terminologie

- **Fenotyp** – označení jedince
- **Genotyp, genom, chromozom** – reprezentace fenotypu
- **Chromozom** – dělí se na jednotlivé lineárně uspořádané geny (*i*-tý gen chromozomů stejného typu reprezentuje stejnou charakteristiku)
- **Alely** – různé hodnoty genu
- **Fitness hodnota** – z rozmezí 0-1, vyjadřuje kvalitu každého jedince

Každý jedinec může být zakódován (geneticky popsán) různým způsobem. Způsob popsání může být důležitý pro úspěch, či neúspěch řešení konkrétní úlohy.

12.3. Příklad:

Nultá generace (fitness hodnota = počet „1“):

- 0100011011 f=0,5
- 0101000100 f=0,3
- 1010110000 f=0,4
- 1110111000 f=0,6

Selekce

- *Vážená ruleta*: $p_i = \frac{f_i}{\sum_1^N f_i}$
  - Pravděpodobnost s jakou bude daný jedinec rodičem

- *Turnajová metoda*
  - Náhodný výběr skupin, z každé skupiny se rodičem stane jedinec s nejvyšší hodnotou fitness.

- *Ořezávání*
  - Seřadíme všechny jedince podle f hodnoty, ořízneme část s nízkou hodnotou, ze zbytku vybereme rodiče

- *Náhodný výběr*
  - Nejjednodušší metoda, f hodnota nehraje roli při výběru jedince pro rodičovství

- *Křížení*
  - Rodiče si vymění část genetického kódu
  - Nejjednodušší metoda – jednobodové křížení
  - Náhodně vybere místo pro řez
    - X: 010001|1011
    - Y: 111011|1000
  - P: 0100011000 f=0,3
  - Q: 111011 1011 f=0,8
  - Vícebodové křížení, možnost více než dvou rodičů

- *Mutace*
  - Náhodná změna náhodného genu v jedinci
  - Velmi malá pravděpodobnost

1. 0100011011 ⇒ 0101011011
2. 0101000100 ⇒ 0101100100
3. 1010110000 ⇒ 1010110100
4. 1110111000 ⇒ 1010111000

  - Lze dosáhnout vlastností, které se v původní generaci nevyskytují

Ukončení

- Dosažení maximálního počtu generací (časové omezení)
- Dosažení minimálního potřebného fitness skóre
- Alespoň jeden jedinec dosáhl dostatečně uspokojivého výsledku
- Dosažený přidělený rozpočet (počítačový čas/peníze)
- Po sobě jdoucí iterace nedosahují žádného zlepšení
- Ruční kontrola
- Kombinace výše uvedených kritérií