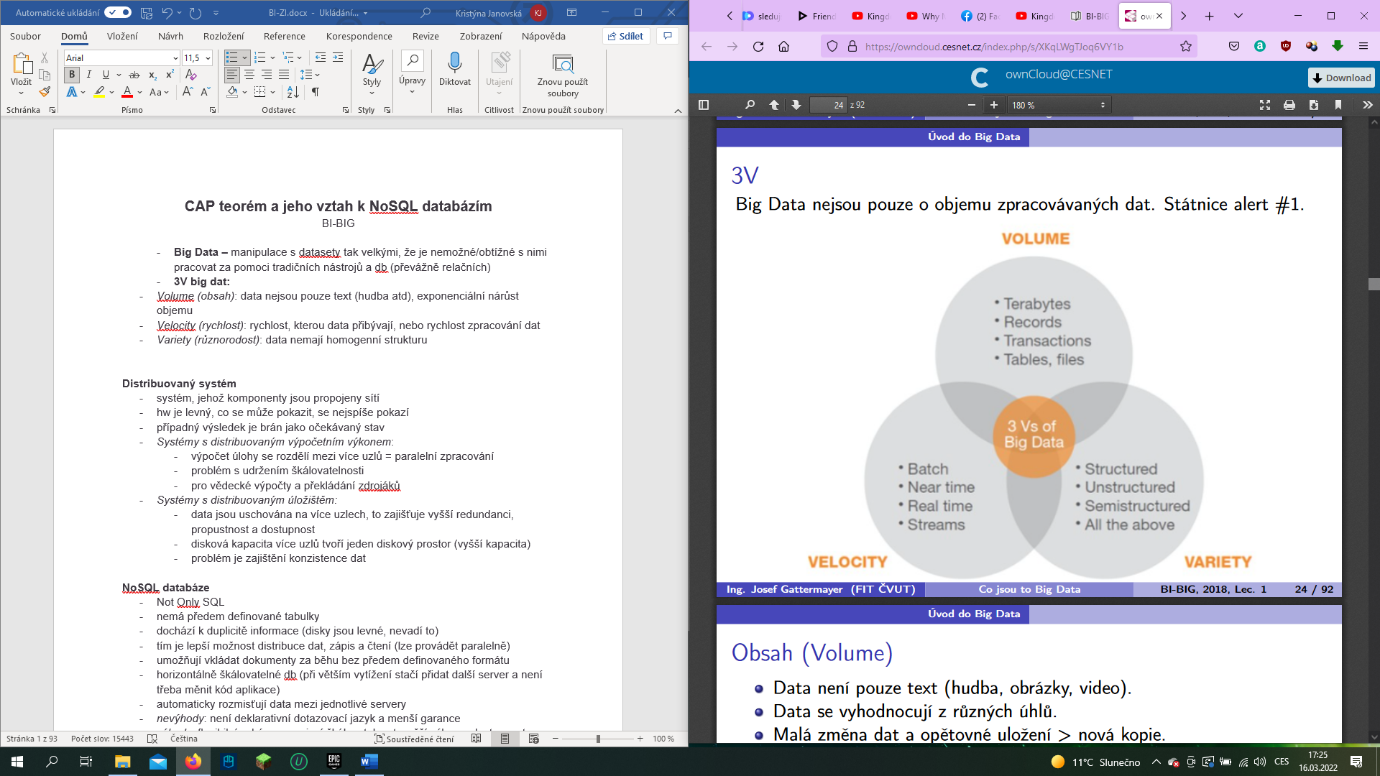
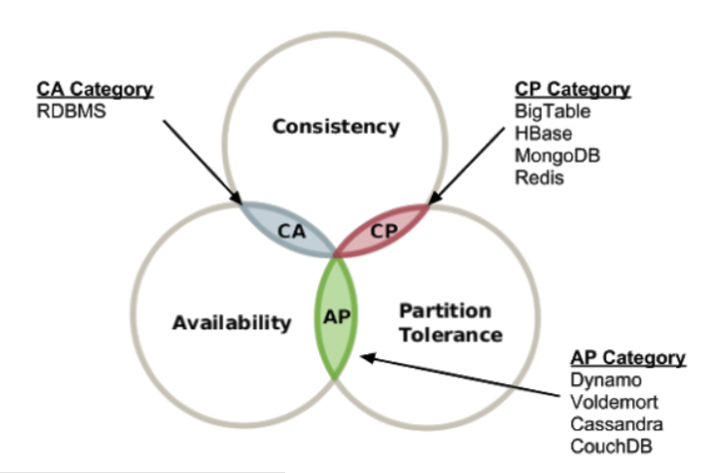
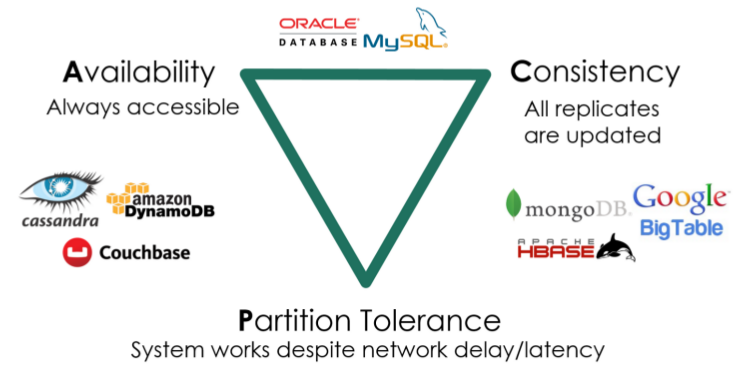
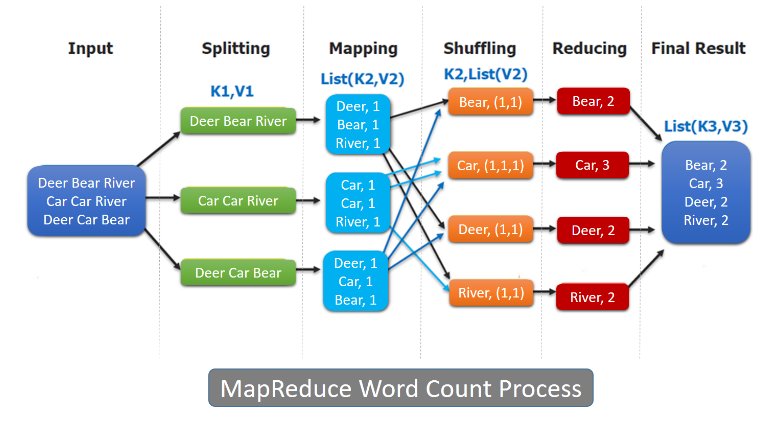
**CAP teorém a jeho vztah k NoSQL databázím**

BI-BIG

* **Big Data –** manipulace s datasety tak velkými, že je nemožné/obtížné s nimi pracovat za pomoci tradičních nástrojů a db (převážně relačních)
* **3V big dat:**
  + ***Volume*** *(obsah)*:
    - Data nejsou pouze text (hudba obrázky, video, ...)
    - Exponenciální nárůst objemu
    - Data se vyhodnocují z různých úhlů
    - Terabyty, petabyty
  + ***Velocity*** *(rychlost)*:
    - Rychlost, kterou data přibývají
      * Čím větší rychlost, tím menší aktuálnost informace
      * Pohyby kurzů na burze
    - Rychlost zpracování dat
      * U výstupu pro uživatele v reálném čase
      * U periodického preprocessingu klidně déle
  + ***Variety*** *(různorodost)*:
    - data nemají homogenní strukturu – adaptibilita na různé podoby dat (video, dokumenty)
* **Distribuovaný systém**
  + systém, jehož komponenty jsou propojeny sítí
  + hw je levný, co se může pokazit, se nejspíše pokazí
  + případný výsledek je brán jako očekávaný stav
  + ***Systémy s distribuovaným výpočetním výkonem***:
    - výpočet úlohy se rozdělí mezi více uzlů = paralelní zpracování – minimalizace času
    - problém s udržením škálovatelnosti (alespoň lineární)
    - pro vědecké výpočty a překládání zdrojáků, moderní architektura serverových aplikací
      * proof of work kryptoměny – Bitcoin, Ethereum
  + ***Systémy s distribuovaným úložištěm****:*
    - data jsou uschována na více uzlech, to zajišťuje vyšší redundanci, propustnost a dostupnost
    - disková kapacita více uzlů tvoří jeden diskový prostor (vyšší kapacita)
    - problém je zajištění konzistence dat
    - BitTorrent, proof of storage kryptoměny (Chia)
    - Datové a výpočetní operace mohou probíhat současně
* **NoSQL databáze =** Not Only SQL
  + Nemá předem definované tabulky
  + Dochází k **duplicitě informace** (disky jsou levné, nevadí to)
  + Tím je lepší možnost distribuce dat, zápis a čtení (lze provádět paralelně)
  + Umožňují vkládat dokumenty za běhu bez předem definovaného formátu
  + Horizontálně škálovatelné db (při větším vytížení stačí přidat další server a není třeba měnit kód aplikace)
  + Automaticky rozmisťují data mezi jednotlivé servery
  + *Nevýhody*: není deklarativní dotazovací jazyk a menší garance
  + *Výhody*: flexibilní schéma, masivní škálovatelnost, vyšší výkon a dostupnost
* **CAP teorém =** 3 základní vlastnosti distribuovaných systémů, přičemž každý může mít nejvíce 2
  + Konzistence, dostupnost a tolerance výpadku
  + Hodnoty nejsou striktně binární, mohou se měnit v čase
  + Slouží stále dobře návrhářům k ujasnění priorit
  + **C = Konzistence (consistency)**
    - Pro každý požadavek je vrácen správný výsledek
    - V případě distribuovaného úložiště vrací každý uzel aktuální a stejná data
    - V případě výpočtu se výsledky od jednotlivých uzlů neliší
  + **A = Dostupnost (Availability)**
    - Na každý požadavek přijde odpověď
    - V kontextu teorému čas nerozhoduje, v kontextu praxe systém musí být dostupný
  + **P = Tolerance výpadku (Partition tolerance)**
    - Určuje chování podpůrného systému, na kterém služba běží
    - Určuje, jestli je systém schopen fungovat i v případě výpadku části paměti
    - výpadek dílčí části systému je brán dnes už jako předpokládaný stav (od 90. let na ústupu)
  + **C + A** (relační db)
    - V případě výpadku části sítě je to problém
    - Než aby vrátil nekonzistentní výsledky, neodpoví radši vůbec
  + **C + P** (MongoDB)
    - Pokud systém nevrátí aktuální data, vrátí chybu
  + **A + P** (Cassandra)
    - Neexistuje záruka, že data jsou aktuální
    - tolerantní vůči výpadku části služby
* **Segmentovaná konzistence a dostupnost**
  + Existují systémy, které nemají požadavky na všechny typy dotazů stejné
* Systém rozdělen na komponenty, každá bude splňovat vždy danou část CAP
* Celý systém nezaručuje ani konzistenci, ani dostupnost (to poskytují pouze jeho části)
* může probíhat na několika úrovních
* *rozdělení podle dat*: jiná data mohou vyžadovat jinou úroveň dostupnosti, file systémy
* *rozdělení podle operací*: operace pro zápis mohou mít jiné požadavky na konzistenci a dostupnost než operace pro čtení, db systémy

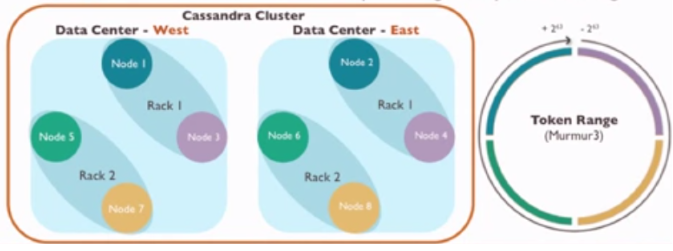
**MapReduce model: principy a jeho využití pro dotazování Big Data**

BI-BIG

* **MapReduce model**
* Programovací model pro distribuované paralelní výpočty
* původně od Google, Apache MapReduce = výpočetní framework nad Hadoopem
* Master-slave architektura (jedno zařízení/proces přebírá jednosměrné řízení nad 1 a více procesy/zařízení)
* Designovaný pro zpracování velkých dat
* *základní myšlenka*: výpočet k datům, ne data k výpočtu (přesouvání dat je drahé na zdroje a náchylné k chybám sítě)
* má *2 fáze*: **mapování** a **redukce**
* zpracování probíhá v místě uložení, je lepší škálovatelnost
* distribuce dat na navzájem nezávislé části ⇒ lze dosáhnou snadného zpracování v uzlech
* *principy*: paralelní zpracování, škálovatelnost, výpočet k datům, tolerantní vůči chybě
* **Mapovací funkce**
  + vstupem mapovací funkce je seznam prvků, na které je následně aplikována transformace
  + je bezstavová, převádí vstup na výstup typu *[klíč, hodnota]*
  + např. převod stringů na UpperCase nebo počet stejných slov ve větě
* **Redukční funkce**
  + agreguje seznam hodnot, který jí je poskytnut, do menšího množství
  + např. jednoduchý součet
* **Postup**
  + nejprve je vstup rozdělen na části (resp. referuje se na určitou část dat, počet částí = počet mapperů), na těch proběhne mapování, poté shuffle & sort a nakonec se data zredukují
  + *mapování*: vstup je seznam hodnot, výstup je pár *[klíč, hodnota]* (v příkladě níž je to [jméno, počet výskytů])
  + *shuffle & sort*: obě fáze probíhají paralelně, třídění a posílání dat po síti (z map do reduce), kritické místo, data jsou seřazena podle komparátoru klíče, do redukční funkce se posílají páry se stejným klíčem,
  + *redukce*: aplikace redukční funkce, výstup není seřazený, může se provést 2x (nejprve před přenosem dat přes síť, aby se zmenšil objem dat)
* **MapReduce InputFormat** – definuje strategii jak data přečíst a jak je rozdělit do mapperů
* **Partitioner** – vstup je výstupní klíč – hodnota z map tasku – rozhoduje o tom, do kterého reduceru půjdou která data
* **MapReduce Combiner –** stejný interface jako reducer, provádí operaci reduceru ještě předtím, než pošleme data přes síť – snížení velikosti dat k přesunu, vhodné pro komutativní funkce
* **Využití –** analytické úlohy, klasifikace, data mining, indexace a vyhledávání
* například Google toto využívá pro generování Google indexu a Apache Hadoop pro spojení HDFS

**Typické databázové stroje pro Big Data (Cassandra nebo HBase) - architektura, databázový model, distribuce dat**

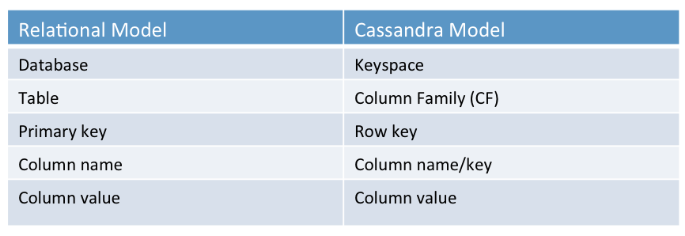
BI-BIG

* **Apache Cassandra** – NoSQL db, AP systém v CAP, používá CQL
  + open source distribuovaný databázový systém
  + ukládá data mezi mnoha servery rozmístěnými v mnoha datacentrech
  + **Data model** – Wide-Column Store
    - Modelování – vhodné navrhnout strukturu pro nejčastěji používané dotazy
      * de-normalizace a duplikace dat – zvyšuje rychlost čtení
  + Vysoká dostupnost
  + Nemá single point of failure
  + **CAP**
    - Podporuje Consistency a Availability
    - Toleruje Network partitioning
  + vyniká v rychlém zápisu (zapisuje se do logu a do memtable, která se periodicky flushuje do SSTables a log se vyprázdní)
  + postavená na denormalizaci (absence JOIN operací) a datové redundanci (zapisujeme tak jak chceme číst)
  + součást **Wide-Column family** (rozrůstá se do šířky, ne do délky) a implementace ukládání stylem hodnota-klíč ⇒ “fancy hash table”
* **Základní funkce Cassandry**
  + replikace: systém sám replikuje podle kritérií zadaných programátorem
  + transparentní škálování: systém sám škáluje (programátor píše stejný kód pro lokální instanci a pro 400 uzlový cluster
  + nastavitelná konzistence: pro jednotlivé dotazy, i za běhu aplikace
  + nastavitelná síťová strategie: správce nastavuje strategii, podle které se ukládají data do jednotlivých racků (množiny uzlů) a datacenter
  + Cassandra Query Language (CQL): podobné SQL
  + postavené na MapReduce: podpora Hadoop MapReduce
* **Architektura**
  + **Node –** jedna Cassandra instance
    - Identifikován nejvyšším tokenem v jednom segmentu Token range
    - Odpovědný za partitions a partition keys
    - Může vlastnit jeden velký segment nebo více menších – každý menší segment je virtuální node
    - Výchozí node je v Racku 1 a Datacentru 1
    - Obsahuje v paměti:
      * *Memtables –* CQL tabulky s indexy
      * *CommitLog*
    - Obsahuje na disku:
      * *SSTables* – statické soubory periodicky ukládané z Memtables
  + **Virtuální node**
    - chová se stejně jako normální node
    - rychlejší Bootstrap a Decommission
  + **Bootstrap** – schopnost rovnoměrného rozdělení dat když se přidá nový node do clusteru
  + **Decommission** – schopnost rovnoměrného rozšíření dat když se node odebere z clusteru
  + **Rack** – logický set nodů
    - umístění node v racku a datacentru umožňuje geograficky vědomé směrování žádostí o zápis/čten
  + **Data center** – logický set/skupina/seskupení setu racků
  + **Cluster** – celý set nodes který mapuje na jeden kompletní token ring
    - node ví o svém clusteru přes conf/cassandra.yaml
  + [**Keyspace**](#kix.tbs6r0k5ufsg) – determinuje replication factor, replication strategy
  + **Partition** – jedna uspořádaná a replikovatelná jednotka dat v node
    - identifikovaná tokenem = partition key – generován partitionerem
    - přidání partition do node – primární klíč tabulky nastaví hodnoty aby se zahashovali do tokenů pro užití jako Partition keys
  + **Partitioner** – určuje jakým způsobem budou partitions a jejich repliky rozděleny po clusteru
    - generuje Partition keys – hashuje hodnotu primary key z žádosti o zápis na token aby byl použit jako partition key
    - Je hashovací funkce
    - **Replikace** – každý uzel si drží část segmentu clusteru (data ring), při zápisu určí partitioner (rozdělovač) hodnotu tokenu hashovací funkcí a primární replika se uloží na místo spravované daným uzlem. Podle replikačního faktoru se přidají další repliky.
    - **Typy replikace:**
      * *MurMur3Partitioner*: MurMur3 hash – rozmisťuje data rovnoměrně po clusteru
      * *RandomPartitioner*: MD5 hash
      * *ByteOrderPartitioner*: - na základě lexikálního pořadí bitů, složité vyvažování, nerovnoměrná distribuce dat, dobré sekvenční vyhledávání
    - **Replication factor** – na kolik node by měl být zápis replikován
    - **Replication strategy** – na jaký node by se měla každý replika uložit
      * Simple – jednoduchá strategie – jeden factor pro celý cluster – pro data v 1 datovém centru, určí se uzel, kde je uložena replika a pak následující po směru
        + NetworkTopology – unikátní faktor pro každý data center

distribuování replik přes racky a data centra

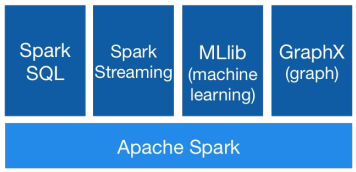
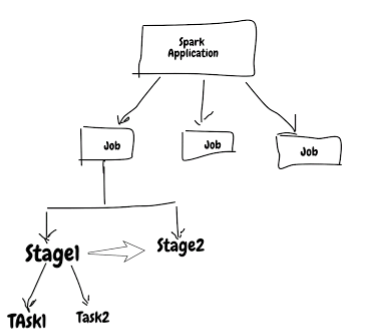
* + - * + OldNetworkTopology - RackAware

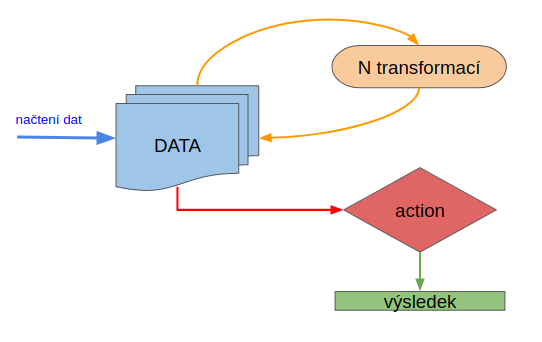
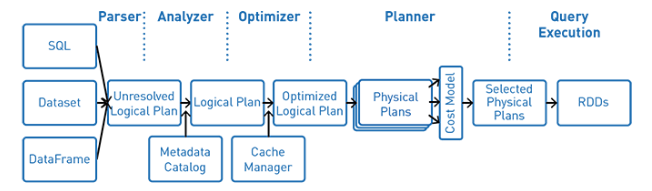
distribuování dat přes racky ve stejném data centru

* + - * Síťová strategie –určen první uzel, kde se uloží replika a pak nejbližší po směru v jiném racku
  + **Komunikace mezi uzly –** pomocí gossip protokolu – každý uzel posílá informace o sobě dalším 3 uzlům, opatřeno časovým razítkem, takže se uzly dozví rychle o problému (výpadek, přetížení...)
  + **Koordinátor –** uzel vybraný klientem, který obdrží request, vybírá Cassandra driver
    - Řídí replikační proces (kolikrát se mají data na uzlech replikovat, replikační strategie)
    - Řídí úroveň konzistence – kolik nodů musí potvrdit žádost o zápis/čtení před odesláním odpovědi klientovi
      * konzistence pro zápis/čtení: ANY, ONE, …, QUORUM, ALL
    - **Zachování konzistence**
      * **Read repair** – pokud má node nekonzistentní data tak se na pozadí pošle aktualizační požadavek a data se zaktualizují
      * **Anti entropy node repair** – opravný nástroj, který se sám dotáže na konzistenci dat a postupně je opraví
      * **Hinted handoff** – obnovovací mechanismus, když je node nedostupný
        + Coordinator uloží hinted handoff když cílový node pro zápis je offline/nepotvrdí žádost o zápis
        + Hint se uloží buď na ostatní nodes s replikami nebo na Coordinator
* **Databázový model**
  + **Keyspace**: udržuje pohromadě všechny CF a replikační faktor, každý keyspace může mít replikační faktor jiný.
  + **Column Family (CF)**: sdružuje řádky, které obsahují sloupce, mapa map (SortedMap<RowKey, SortedMap<ColumnKey, ColumnValue»)
  + **Row**: řádka je identifikována jednoznačným (primárním) klíčem a obsahuje jednotlivé sloupce s daty, sloupce můžou obsahovat odlišné data ⇒ řádka nemá pevnou struklturu
  + **Column**: má svůj název, hodnotu a časové razítko, které určuje čas vložení (koordinátoři se podle něj rozhodují, jestli jsou sloupce aktuální)
    - Typy sloupců
      * Standard: klasický sloupec s jednou hodnotou
      * Composite: spojený sloupec, pokud PK je složen z více sloupců
      * Expiring: sloupec s omezenou dobou platnosti
      * Counter: čítací sloupce pro zvyšování hodnoty ve sloupci
    - Sdílený sloupec – sdílený všemi řádky se stejným partition key
* **Apache Hbase**
  + CP systém (z CAP), nepodporuje SQL
  + Open-source nerelační distribuovaná db
  + Část Apache Hadoop
  + Hosting obrovských tables
  + Odolné vůči chybám při ukládání velkého množství řídkých dat
  + MapReduce model
  + Rychlé a konzistentní čtení (i zápis) na velkých souborech dat s vysokou propustností a nízkou latencí
  + Lineární a modulární škálovatelnost

**Principy výpočtů v distribuovaných systémech – Spark framework nebo jiná podobná technologie. Principy distribuované indexace – Elastic nebo jiná podobná technologie**

BI-BIG

* **Spark** – framework pro distribuované výpočty (další např. MapReduce)
  + umí batch, stream (nekončící kolekce dat) a machine learning
  + REPL – interactive processing
  + **Architektura**:
    - Core – RDD (resilient distributed dataset)
    - SQL – zpracování strukturovaných dat
    - Streaming – datasety a DataFrame API
    - MLib (machine learning) – ML algoritmy, featurization, lineární algebra, statistika
    - GraphX – grafové výpočty
  + **Principy**:
    - In-memory (narozdíl od MapReduce to nerozhází po síti)
    - ****Fault tolerant
    - Lazy evaluation (trigger je akce, vyhodnocení až v okamžiku, kdy je to opravdu potřeba)
    - Immutable data, funkcionální programování
    - Podporuje paralelní zpracování
    - Master/slave architektura (driver program, cluster manager a worker nodes, driver komunikuje s worker nody a posílá jim tasky)
  + **SparkContext** – instance Spark aplikace, zodpovídá za převod aplikace na orientovaný graf (DAG) jednotlivých úloh
  + **Job** – počítá výsledek akce, největší akce výpočtu
  + **Stage** – fyzická jednotka výpočtu, od shuffle do shuffle (např. joiny, reduce, agg. fce...)
  + **Task** – výpočet jedné stage nad jednou částí dat, v jednom vlákně na jednom exekutoru (např. 800 tasků na 800 načtení souborů z HDFS)
  + **Resilient DIstributed Dataset (RDD)**
    - Kolekce objektů rozdělených na části
    - Immutable (změna není možná), přežijí ztrátu worker nodu (ztracená partition se dopočítá) in-memory (rychlé a náročné na paměť)
    - další např DataSet a DataFrame
  + **Transformace**
    - Narrow – potřebují data jen z jedné partition (map, union, filter…)
    - Wide – potřebují data z více partition (groupByKey, join...), vyžadují shuffle
  + Akce – collect, take, top, count, reduce, fold, aggregate, foreach, saveAsTextFile
  + **Processing pipeline –** sestavuje se z transformací a akcí
    - lazy evaluated – evaluace (překlad do DAG – directed acyclic graph) je spuštěna až s akcí
      * DAG se před rozdvojením počítá vícekrát



* **ElasticSearch** – distribuovaný open source fulltext vyhledávač
  + Dostupný, rychlý a dobře škálovatelný
  + Komunikace probíhá přes HTTP
  + Hlavní komponenta **Elastic Stacku** (nástrojů pro ukládání, analýzu a vizualizaci dat)
  + Do ElasticSearch tečou nezpracovaná data z různých zdrojů
  + Nejprve jsou tato data analyzována, normalizována a poté jsou indexována
  + **Indexace**:
    - Po indexaci mohou uživatelé spouštět komplexní dotazy a získávat souhrny svých dat
    - Používá invertovaný index pro rychlé fulltextové vyhledávání (invertovaný index je tvořen jedinečnými slovy, které se objevují v celém setu dokumentů, a identifikuje všechny dokumenty, kde se toto slovo nachází, jako ve VWM)
    - Během indexovacího procesu Elasticsearch ukládá dokumenty a vytváří **invertovaný index**, aby bylo možné prohledat data dokumentů v reálném čase
    - Indexování je inicializováno index API, kde se dají přidávat/aktualizovat JSON dokumenty v konkrétním indexu
  + **Elasticsearch Index**
    - Kolekce navzájem souvisejících JSON dokumentů

každý dokument koreluje množinu klíčů (názvy polí) s odpovídajícími hodnotami (stringy, čísla, geolokace a jiné typy dat)

**Dědičnost atributů a metod v jazyku Java, souvislost s kovariantními typy, statičností a nestatičností, seznamem vyhazovaných výjimek a viditelností (private, protected, public, default).**

BI-PJV

* **OOP** – nahlíží na entity a vztahy jako modely věcí z reálného světa
  + Abstrakce – zobecnění pohledů na částečná řešení
  + Dědičnost – získávání vlastností a schopností děděním z vhodné třídy a případně i dat nebo částečných požadavků z interface
  + Polymorfismus – jednotné zacházení s různými objekty mající některé společné zděděné schopnosti
  + Zapouzdření – skrývání částí tříd a interface
* **Třída** – „hotový plán“ – charakterizována vlastnostmi které uchovává (data,, členské proměnné) a funkčními možnostmi (metody)
  + **Objekt** – instance třídy (vytvářet pouze dynamicky pomocí new)
  + Statické třídní proměnné
  + Nestatické instanční proměnné objektu
  + Statická metoda x instanční metoda (operace nad objektem)
  + Po vytvoření objektu se jeho atributy inicializují nulovými hodnotami
  + „extends“ pro dědění
* **Rozhraní** – **interface** – norma, požadavek, vyžaduje implementaci
  + Lze považovat za značně omezenou abstraktní třídu, která má všechny metody abstraktní
  + Od Java 8 ještě default a static metody
  + Umožňuje částečný pohled na třídu a slouží jako norma či požadavek
  + Není žádný základní interface – nemusí mít žádného předka
  + Může být přímým potomkem více interfaců uvedených v seznamu za klíčovým slovem extends
  + Nemůže být potomkem žádné třídy
  + Nemůže být finální
* **Abstraktní třída** – neúplný plán
  + Může být i přímý potomek konkrétní třídy
  + Může implementovat více rozhraní
  + Může, ale nemusí mít abstraktní metody
  + Modifikátor abstract
  + Alespoň jeden konstruktor, ale nelze podle ní objekty vytvářet
  + Nemůže být finální
  + Statický kontext je funkční = static atributy, metody, inicializátory a výčty
  + Abstraktní metoda v definici místo těla jen středník
* Třída extends třída, rozhraní extends rozhraní, třída implements rozhraní
* **Object** – pratřída všech tříd, není nutné uvádět za extends – každá třída je přímým potomkem
  + Nemá předka – nic nedědí
  + Nemá atributy
  + Definuje 11 generálních metod, všechny ostatní třídy je dědí
    - toString, equals, hashCode, clone, getClass, notifz, wait, ...
* **Dědičnost** (inheritance)
  + Vztah třída a nadtřída – superclass, přímou nadtřídu vyznačuje třída ve své hlavičce za klíčovým slovem extends, jinak je přímým předkem Object
  + Vztah třída a podtřída – subclass
  + Dědičnost usnadňuje:
    - Přístup k existujícím třídám a interfacům a jejich pochopení
    - Vytváření nových tříd a interfaců s využitím vlastností a schopností již existujících, ověřených a dokumentovaných předků
* **Dědičnost atributů**
  + Rozhraní (interface) může mít jen konstanty = **public final static** atributy
  + Při vícečetném dědění stejnojmenných atributů nastane kolize
  + U atributů je vazba známa již při překladu = nepočítá se za běhu
  + Atributy v potomkovi můžeme **překrýt** – variable hiding
  + Jedná se o jiný atribut téhož jména, překrývající atribut:
    - Nemusí zachovat statičnost, či nestatičnost
    - Nemusí zachovat viditelnost
    - Může mít jiný typ
* **Dědičnost metod**
  + Dědí se pouze viditelné členy – atributy, metody, vnitřní třídy, vnitřní interfacy, výčty
  + **Viditelnost** – nesmí mít příliš restriktivní přístupový modifikátor
    - **Public** = dědí a vidí všichni potomci, vidí všichni
    - **Protected** = dědí a vidí všichni potomci
    - **Default** = **friend** = dědí potomci v témže balíčku, vidí všichni ve stejném balíčku
    - **Přivate** = nevidí a nedědí nikdo
  + Potomek nemůže dědictví odmítnout, a tedy není nikdy chudší než předek
* **Statické metody a dědičnost**
  + Statické metody mohou být definovány v: Třídách, Abstraktních třídách, Rozhraních
  + Vlastnosti statických metod:
    - Nemohou být abstraktní – vždy mají tělo
    - Lze volat pomocí jména třídy
    - Lze volat pomocí referenční proměnné (nedoporučuje se)
    - Nelze „přepsat“ (Override)
    - Lze překrýt v potomkovi (podobně jako u atributů)
    - Překladač určí vazbu v době překladu
* **Přepsání metody – overriding**
  + Nevyhovuje-li potomkovi zděděná nefinální metoda, lze v potomkovi deklarovat metodu, která:
    - Má shodnou signaturu
    - Má kovariantní návratový typ či subtyp – avšak ne primitivní
      * **Kovariantní návratový typ** – návratový typ přepisované metody – povoluje „zúžit“ návratový typ přepsané metody bez potřeby castu nebo kontroly návr. Typu
        + Metoda podtřídy může vrátit podtyp návratového typu té metody z nadtřídy
        + Např. když je v nadtřídě metoda co vrací hodnotu typu List, tak podtřída může vrátit List, ale taky kterýkoliv z podtypů Listu – ArrayList, Stack, Vector, …
    - Nemění statičnost – nestatičnost
    - Nezužuje modifikátor přístupu
    - Nerozšiřuje množinu kontrolovaných výjimek udaných za throws
* **Zapouzdření** (encapsulation)
  + Ve třídách a objektech lze ukrýt atributy, metody, konstruktory, vnitřní třídy a vnitřní interfasy a tím podpořit spořádanost a bezpečnost
  + Inicializátory jsou skryté již tím, že nemají jméno
  + Míru zapouzdření určuje modifikátor viditelnosti členu či konstruktoru (public, …)
  + K nedostup. členům nějaké třídy lze přistupovat z jiné třídy jen pomocí neprivátních metod oné třídy
  + V interfacech lze zapouzdřovat jen atributy a jen úrovní „default“
* **Hierarchie tříd**
  + Vztah nadtřída-podtřída je tranzitivní – jestliže je x nadtřídou y a y nadtřídou z, je x nadtřídou z
  + Pro referenční proměnné platí: do proměnné typu Tnad může být přiřazena ref. na objekt typu Tpod
  + Na objekt referencovaný proměnnou typu Tnad lze vyvolat pouze metodu deklarovanou ve třídě Tnad
  + Jde-li však o objekt typu Tpod, metoda se provede tak, jak je dáno třídou Tpod
  + Hodnotu referenční proměnné typu Tnad lze přiřadit referenční proměnné typu Tpod pouze s použitím přetypování, které zkontroluje, zda referencovaný objekt je typu Tpod
* **Gettery** a **settery** – dobrovolná jmenná konvence pro přístupové metody k zejména privátním atributům, které jejich hodnotu vydávají (getter, accessor) anebo mění (setter, mutator) usnadňuje tvorbu nadstavbového softwaru
* **Použití dědičnosti – substituční princip** – dědičnost, pokud je podtřída skutečně specializací nadtřídy
  + Všude, kde lze použít nějaká třída, musí jít použít i její podtřída, aniž by uživatel poznal rozdíl
  + Netýká se jen syntaxe, ale i sémantiky (tu je nutné vhodně popsat)
  + Podtřída musí dodržovat kontrakt nadtřídy
  + Vztah (ISA) musí být trvalý
  + Odvozené třídy nesmí nikdy vyžadovat více (precondition)
  + Odvozené třídy nesmí poskytovat méně než bázová třída (postcondition)
* **Přetypování – casting**
  + Objekty, atributy, lokální proměnné, parametry a návratové hodnoty metod mají definovaný, neměnný typ – dva druhy typů:
    - **Primitivní** (boolean, char, byte, short, int, long, float, double)
    - **Referenční** – třídy, interfejsy a pole (primitivní i referenční)
  + **Přetypování** = změna pohledu na objekt prostřednictvím reference jiného typu, typ objektu je neměnný, daný třídou, dle níž byl zkonstruován
  + Přetypovat potomka na předka lze přiřazením
  + Přetypovat (casting) předka na potomka lze jen tehdy, je-li referovaný objekt typu potomka takto:

Potomek y = (Potomek) x;

* + Chybné přetypování → výjimka ClassCastException
* **Výjimky** – objekty, slouží k indikaci závad programu
  + Jejich třídy tvoří podstrom třídy Throwable, od které dědí vesměs všechny své metody
  + Kompilátor v čase kompilace(compile-time) rozeznává výjimky:
    - **Kontrolované**(checked) – vyžaduje ošetření
      * Klauzulí try-catch
      * Vyznačení v hlavičkách metod
    - **Nekontrolované**(unchecked) – kompilátor nevyžaduje vyznačení ani ošetření
      * Považovány za programátorské chyby, které budou odstraněny
  + Pokud výjimka patří do stromu java.lang.RuntimeException, je nekontrolovaná, jinak je kontrolovaná
  + **Odchytávání výjimek** – klauzule try může být vnořena do kteréhokoli bloku, a tedy i do bloku jiné kl. try
    - Typy bloků:
      * Hlídač — **try** (právě jeden) - dozírá, byla-li v bloku vyhozena
        + Výjimka nebyla vyhozena – dokončí se blok try a předá řízení uklízeči = finally
        + Výjimka byla vyhozena – další příkazy v try bloku se již neprovedou – hlídač zjistí typ výjimky

Procházením řešitelů shora dolů se snaží najít prvního kompetentního řešitele = jeho parametr je typem nebo nadtypem výjimky – nalezne-li ho, předá mu řízení

Nenalezne-li řešitele, předá řízení uklízeči a pak nevyřešenou výjimku vyhodí do nadbloku

* + - * Řešitel — **catch** (libovolný počet) – má prostřednictvím parametru referenci k aktuální výjimce i dalším proměnným
        + Může situaci trochu napravit anebo alespoň řádně ohlásit, nelze však nijak zařídit pokračování v nedokončeném hlídaném bloku
        + I když řešitel vůbec nic neudělá, je odchycená výjimka „vyřešena“
      * Uklízeč — **finally** (nanejvýš jeden)
    - Klauzule try – catch – finally může mít tři tvary:
      * try – catch – finally
      * try – catch
      * try – finally
    - I řešitelé a uklízeč mohou vyhodit nějakou výjimku – tu však zpracuje (dynamicky) obalová try klauzule. Též lze sestrojit novou výjimku a jako její příčinu vložit tu původní.
    - Šíření neodchycené výjimky po dokončení bloku finally
      * Je-li volajícím blokem obalová try klauzule postupuje se obdobně, jako v případě nalezení odpovídající catch části ve volaném bloku
      * Je-li tím blokem metoda, pak tato metoda vyhodí výjimku do příkazu metody odkud byla zavolána a tam se postupuje obdobně
      * Nenalezne-li se žádný kompetentní řešitel, vyhodí se výjimka do obalové kl. JVM – odtud byla zavolána metoda main(String[ ] args) - pak JVM vypíše hlášení a ukončí běh
  + **Vlastní výjimky** – lze vytvářet vlastní výjimky jako potomky existujících většina nepřidává žádné vlastní atributy ani metody, jen definuje dva konstruktory a vše ostatní dědí z třídy Throwable
  + Try with resources – zdroje typu AutoCloseable – existuje-li blok try, budou na jeho konci všechny zdroje automaticky uzavřeny – úhlednější
  + **Řetězení výjimek** – pokud při zpracování výjimky vznikne další, je původní potlačena (supressed), ale je stále dosažitelná

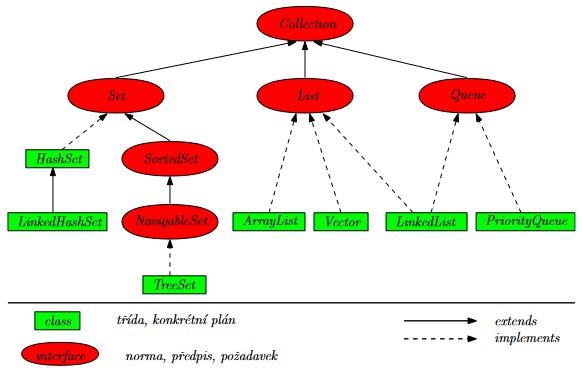
**Vstup a výstup v Javě, rozdělení proudů, jejich navazování, serializace a externalizace dat, výjimky, jejich rozdělení a zpracování**

BI-PJV

* **IO** vs. **NIO**
  + IO – proudově orientované, Stream
    - IO operace jsou blokující, vlákno stojí, dokud nejsou data načtena/zapsána
  + NIO – používá Buffer, lze využít pro vstup i výstup
    - Operace IO jsou neblokující, programátor kontroluje, zda jsou data připravena
    - Přináší Selector – umožňuje jednomu vláknu řídit více kanálů (Channel)
    - Path, Files, Stream API, funkcionální programování
* **Soubory**
  + Typické **operace**
    - Otevření souboru
    - Čtení údaje
    - Zápis údaje
    - Uzavření souboru
  + Přístup k údajům (čtení nebo zápis) může být:
    - Sekvenční nebo libovolný (adresovatelný)
  + Sekvenční přístup umožňuje pouze postupné (sekvenční) čtení nebo zápis údajů – použitelný pro soubory, roury mezi procesy, sockety přes internet...
  + Soubory s libovolným přístupem umožňují adresovatelné čtení nebo zápis údaje (podobně jako pole)
  + Způsob přístupu k údajům v souboru je dán programem
  + Zde se budeme zabývat hlavně sekvenčními soubory
* **Soubory a proudy** – Java rozlišuje soubory (file) a proudy (stream)
  + **Soubor** = množina údajů uložená ve vnější paměti počítače
  + **Proudy** = nástroje k přenosu informací např. z/do souboru, ale také do/ze sítě, paměti, jiného programu atd.
  + Informace může mít tvar znaků, bajtů, skupin bajtů (obrázky, ...), objektů, ...
  + Přenos informace se děje dvoj/třívrstvě v proudech (streams):
    - Otevření přenosového proudu pro bajty či znaky
    - Otevření přenosového proudu pro datové typy Javy
    - Filtrace dat podle požadavků – bufferování, řádkování, ...
* Rozdělení proudů:
  + **Textové proudy** – abstraktní třídy Reader a Writer musí zajistit konverzi znaků mezi OS a Javou
    - Public metody**,** vyhazují výjimku **IOException**
  + Abstraktní třídy InputStream, OutputStream
    - Public metody, vyhazují výjimku **IOException**
  + **Překlad bytů na znaky** – InputStreamReader a OutputStreamWriter
  + Interface DataInput a DataOutput – definují (abstraktní) **metody pro čtení/zápis** všech primitiv z/do proudu
  + FileOutputStream – **kopírování souboru** po bytech
* **Serializace objektů** do výstupu
  + Serializable objekty lze přenést objektovým proudem tzv. serializací
  + Rozloží hodnoty jeho atributů na byty:
    - Modifikátory přístupu nemají vliv, všechna pole jsou serializovatelná
    - Neserializují se atributy označené static či transient ani třídy ani metody
    - Rekurz. probírkou referencí se serializují i všechny refer. objekty tedy i velmi rozsáhlý graf
* **Obnovení objektů** – **deserializace** vstupu
  + K dispozici musí být odpovídající třídy – kompatibilní dle serialVersionUID
  + Nevolá konstruktory Serializable tříd – nechceme iniciální hodnoty
  + Volá konstruktory nadtříd, které nejsou Serializable
  + Nepřenesené atributy nastaví na jejich defaultní hodnoty
* **Identifikace serializovatelné třídy** – serialVersionUID
  + Stanovené výpočtem (hešová hodnota) či přiřazením
  + U přeložené třídy lze zjistit programem serialver
  + Při serializaci se přikládá do objektového proudu – viz třída java.io.ObjectStreamConstants
  + Při deserializaci se porovná s hodnotou třídy na vstupní straně
  + Při **neshodě se vyhodí výjimka**
  + Programátor může explicitně stanovit, že třídy jsou kompat. tím, že třídě vnutí určitou hod. Atributu
* **(De)serializace a dědičnost** – pokud serializujeme instanci potomka (musí být Serializable) pak
  + I rodič implementuje Serializable a pak je serializován v rámcipotomka
  + Nebo předek musí mít bezparam. konstr. (implicitní či explicitní). Tento je volán během deserializace
  + Naše vlastní ovládání serializace a deserializace - (De)serializace funguje automaticky, ovládáme výběr a pořadí
* **Externalizace** – vlastní „serializace“ – java.io.Externalizable – implements Externalizable + explicitní public konstruktor bez parametrů a dvě metody plně zodpovědné za ser-deser objektu i dat nadtříd
  + Externalizace je rychlejší než serializace, nebot’ JVM neanalyzuje strukturu
* **Komprese** – třída Deflater – můžeme vytvořit standardní soubor .zip, .gzip nebo .jar, který je čitelný standardními programy, např. gzip, unzip, WinZip
* **Dekomprese** – třída Inflater – Takto lze dekomprimovat standardní .zip, .gzip a .jar.
* Třída **RandomAccessFile** – adresovatelný přístup, čtení, zápis i připisování jakoby bytového pole realizovaného na periferním zařízení
* Collator – české řazení

**Kolekce a JCF, jejich rozdělení dle implementovaných rozhraní, kolekce a metody hashCode, equals a implementace Comparable, synchronizace kolekcí**

BI-PJV

* **Kolekce** (= kontejnery) – objekty, které slouží k ukládání, načítání, zpracování a přenášení  
  prvků stejného typu
  + **JCF = Java Collections Framework** – sada tříd a rozhraní, která implementují běžně opakovaně použitelné datové struktury kolekce
  + Algoritmy kolekcí – polymorfní realizace metod tříd
  + Pomalejší než pole
  + **Výhody** použití kolekcí
    - Ulehčení programování
    - Kratší a přehlednější kód
    - Rychlejší algoritmy
    - „neomezený rozsah“ počtu ukládaných objektů
    - Zvýšení čitelnosti programů
    - Lepší přenositelnost
* Genericita – do kolekcí je možné vkládat instance třídy Object a při výběru přetypovávat zpět
* Nadrozhraní **Collection** – potomci rozhraní Set, List, Queue
  + **Set** – ADT množina, prvky se neopakují, může být seřazená
    - Oproti Collection nepřidává žádné další metody
    - Implementace – HashSet, LinkedHashSet, TreeSet, ...
  + **List** – ADT množina, prvky mají indexy, prvky se mohou opakovat
    - Základní implementace – ArrayList, LinkedList – JCF, Vector
    - Indexovaný přístup prvkům
  + **Queue**, Dequeue – ADT fronta, obousměrně zřetězená fronta, LIFO, FIFO, prvky se mohou opakovat
    - Potomci BlockingQueue, Deque, LinkedList, PriorityQueue
* Další bois
  + Rozhraní Iterator, ListIterator
  + Třída **ArrayList** (extends AbstractList) – implementace List pomocí natahovacího pole, podobné Vector
  + Třída **LinkedList** (extends AbstractSequentialList) – patří do JCF, obousměrně zřetězený seznam jako implementace List
    - Implements List, Deque, Cloneable, Serializable
  + Vector, Stack extends Vector – nejsou součástí JCF, synchronizované
  + Třída **HashSet** – ADT množina, patří do JCF
    - Extends AbstractSet, implements Set, Cloneable, Serializable
    - Negarantuje pořadí procházení prvků
    - **Hešování** – technika, která v ideálním případě zaručí vložení, odebrání, zjištění přítomnosti prvku v konstantním čase
      * **Hešovací funkce** – zajišťuje mapování prvků kolekce na int, který slouží k výpočtu indexu do kolekce
        + Ideálně pro dva různé prvky vytvoří **hashCode** dvě různé hodnoty
        + Mnohdy to nejde, např String, počet různých řetězců výrazně převyžuje int
  + Hashtable – extends Dictionary, implements Map
    - Index počítán přes modulo velikosti – rychlý výpočet a při zvětšování rychlé rozdělení zřetězených kyblíků
    - Významné nižší bity
  + HashMap – extends AbstractMap implements Map
    - Index počítán bitovým násobením
    - Vylepšení výpočtu indexu kyblíku (bucket) – významné nižší bity – ztráta efektivity

1. Výpočet hashCode
2. Přepočet pomocnou metodou
3. Bitové maskování podle velikosti kolekce

* Metody **hashCode a equals** – úzce spolu souvisí
  + Pokud zastíníme equals, musíme zastínit hashCode
  + Správně vytvořený objekt musí splňovat:
    - Pokud jsou dva objekty stejné (podle equals) musí metody hashCode vracet stejnou hodnotu
  + **Equals** – implementace relace ekvivalence
    - **Reflexivní** – x.equals(x) musí vrátit true, pro každé x (**mimo null**)
    - **Symetrická** – pro jakékoliv x a y musí x.equals(y) == true právě tehdy, když y.equals(y) == true
    - **Tranzitivní** – pro jakékoliv x, y, z musí platit, že když x.equals(y) == true a y.equals(z) == true, pak x.equals(z) == true
    - **Konzistentní** – pro jakékoliv odkazové hodnoty x a y musí platit, že buď x.equals(y) vrací true nebo stále vrací false za předpokladu, že nedojde ke změně žádných informací použitých v porovnáních equals daného objektu
    - Pro všechny odkazové hodnoty x, které nejsou null, musí x.equals(null) vrátit false
  + **hashCode** – s ohledem na equals zamixuje důležité vlastnosti objektu

1. uložte magickou prvočíselnou konstantní nenulovou hodnotu (např. 17) v proměnné typu int nazvané result
2. pro každý významný atribut f ve svém objektu = každý atribut zvažovaný metodou equals, vypočtěte kód tmp typu int (podle datového typu)
3. result = result \* něco + tmp
4. pokračujeme s dalším atributem a bodem 2
5. vraťte výsledek, pokud již nejsou žádné další důležité atributy
   * + když konečně skončíte s vytvářením metody hashCode(), zeptejte se sami sebe, zda mají rovné instance rovné hešovací kódy.
     + Pokud ne, zjistěte proč a problém opravte.
     + Vždy nejprve vytvořte equals (případně spolu s využitím IDE)

* **Kolekce a řazení** – rozdělení
  + Způsoby řazení
    - Přirozené řazení, třída implementuje **Comparable**, pokud ne, nastane chyba za běhu
      * Metoda sort používá vylepšený quickSort
      * Metoda sort volá compareTo
    - Řazení s komparátorem
      * **Komparátor** = třída implementující rozhraní **Comparator** a tedy metodu compare (o1, o2), pokud není typována
      * Každá třída může mít 1 způsob přirozeného řazení a kolik chceme komparátorů
  + Aplikace řazení
    - Využití seřazené kolekce
    - Dodatečné seřazení neseřazené kolekce pomocí Collections
  + Vztah mezi equals, compareTo, compare a hashCode
    - List a Queue požadují equals
    - compareTo, compare musí být pro binarySearch, seřazené kolekce, řazení
    - hashCode je pro všechny kolekce s názvem HashNěco
    - o1.equals(o2) == true ⇐⇒ o1.compareTo(o2) == 0
    - musí platit i pro compare a to pro všechny komparátory
* Další supr rozhraní
  + Interface SortedSet – extends Set
    - Prvky se nesmí duplikovat – nesmí obsahovat prvky e1, e2, pro které platí e1.compareTo(e2)==0, srovnej s e1.equals(e2)==true
    - SortedSet uchovává prvky ve vzestupném přirozeném pořadí nebo v pořadí daném komparátorem
  + Interface Map – asociativní pole
    - Mapa – map =´zobrazení keys -> values, tedy klíčů na hodnoty
* **Synchronizace kolekcí**
  + U jednovláknových aplikací není třeba synchronizovat
  + U vícevláknových je nutné zajistit, aby jedno vlákno nepřistupovalo ke kolekci, kterou upravuje jiné vlákno = nutno synchronizovat
  + Kolekce jsou standardně nesynchronizované, a tak rychlejší
  + Synchronizace obalením JCF kolekce
  + Synchronizace použitím staré, ale synchronizované kolekce
    - Možno použít Vector, Stack, Hashtable

**Java 8 - Stream API a lambda výrazy, rozdělení operací nad streamy, funkční rozhraní Predicate, Function, Supplier, atd**

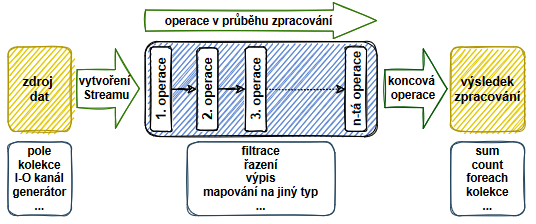
BI-PJV

* **Lambda výrazy** – výrazy, které lze předávat v programu podobně jako proměnné bez nutnosti je obalovat do rozhraní
  + Typ dynamicky určen podle kontextu
  + V podstatě funkce bez formální deklarace
  + Pomocí lambda výrazů lze procházet kolekce
* **Java 8 – funkční rozhraní**

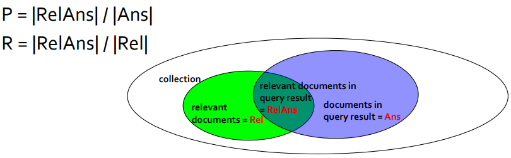
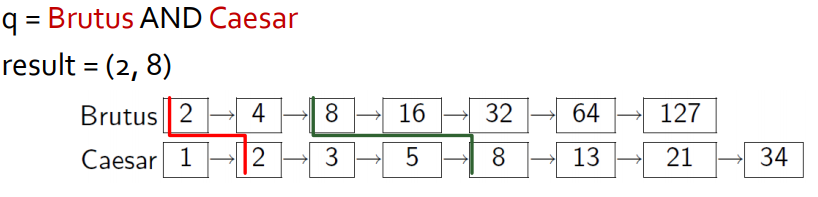
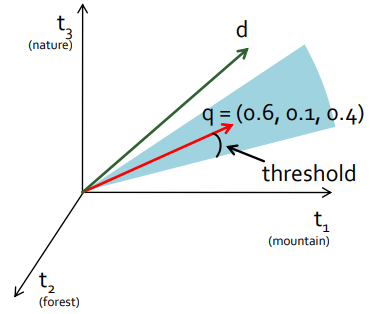
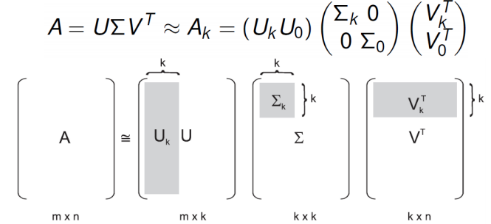
* + Informaci o typech můžeme vynechat (typová kontrola stále funguje), a také složené závorky a return
  + Pokud má metoda funkčního rozhraní jen jeden parametr, můžeme vynechat i kulaté závorky
  + Závorky nemůžeme vynechat, když není žádný parametr, nebo je jich víc
  + Funkce často nebudeme deklarovat jako proměnné
    - Funkci prostě vytvoříme a hned předáme jako argument metodě  
      nebo konstruktoru, které přijímají dané funkční rozhraní
  + Odkazy na metody – v Javě 8+ se pomocí nich můžeme odkazovat na metody i konstruktory a konvertovat je na funkce, resp. funkční rozhraní
  + Odkazy na konstruktory – metoda nebo konstruktor musí mít vhodný počet a typ parametrů, které odpovídají požadovanému funkčnímu rozhraní
  + V praxi část Supplier<Long> supplier často vynecháte, nebudete deklarovat proměnnou a jen předáte ”odkaz na metodu“ někam, kde je požadováno funkční rozhraní
  + **Vlastní funkční rozhraní** – java.util.function
    - Funkce obvykle nebudeme používat hned (to bychom mohli rovnou zavolat ten kód), ale pravděpodobně je předáme do jiné části programu, kde se budou volat
  + **Skládání funkcí** - u compose() se nejdřív zavolá vnitřní funkce (ta předaná jako parametr)
    - Místo compose() můžeme pro skládání použít andThen(), kde to funguje přesně naopak
    - Výsledkem skládání funkcí je opět funkce s patřičnými generickými
    - Typy – nemusíme hned apply(), ale můžeme si ji uložit do proměnné a používat opakovaně nebo předat někam dál
  + **Predikáty** – Predicate – obor hodnot je boolean
    - Definiční obor je daný generickým typem
    - Metoda Predicate.test(T t) vyhodnocuje pravdivost predikátu nadzadanou hodnotu
    - Skládání predikátů – stejně jako javovské výrazy pospojované AND a OR operátory se i predikáty vyhodnocují zleva doprava a zkráceně
      * Stejně jako u klasických výrazů s && a || je dobré i predikáty řadit podle výpočetní/paměťové složitosti a podle pravděpodobnosti získání výsledku
* **Stream API** – definice **roury**

1. Vytvoření zdroje, proudu (kolekce, pole, generující funkce, I-O kanál, ...)
2. Operace zpracovávající Stream – 0 až N
3. Koncová operace
   * Data se nemění,
   * „mezivýsledky“ jsou Stream
   * Operace se řetězí
   * Použitelné jen jednou
   * Výpočet začne až po koncové operaci

* **Proud – stream** – vytvoření proudu:
  + Z kolekcí prostřednictvím metod stream() a parallelStream()
  + Z polí prostřednictvím metody Arrays.stream(Object[])
  + Z továrních metod třídy Stream, např.: Stream.of(Object[])
  + IntStream range(int, int)
  + Stream iterate(Object, UnaryOperator)
  + Řádky souboru lze získat metodou BufferedReader.lines()
  + Proud náhodných čísel lze získat metodou Random.ints()
  + Proud položek v ZIP-souboru lze získat metodou JarFile.stream()
  + **Práce s proudy** 
    - Obsah proudu můžeme během práce filtrovat
    - Proudy můžeme také řadit
    - Prvky proudu můžeme konvertovat na jiný typ
* **Funkční rozhraní** – funkční rozhraní je nový termín používaný pro rozhraní (interface), která mají právě jednu abstraktní metodu
  + Známe Comparable, Runnable
  + Nově java.util.function
  + **Consumer** = čistí konzumenti – nic nevrací
    - Consumer<T> void accpet(T value)
  + **Supplier** = producenti – nemají parametry a vracejí hodnotu zadaného typu
    - Supplier<T> T get()
  + **Function** = funkce – zpracují svůj parametr a vrátí funkční hodnotu
    - Function<T, R> R apply(T value)
  + **Operátory** – unární, binární, ... návrátová hodnota stejného typu jako parametr
    - UnaryOperator<T> T apply(T operand)
  + **StringJoiner** – operace merge – nevkládá prefix a sufix, různé oddělovače – druhý seznam vložen jako jedna položka
    - Public StringJoiner(CharSequence delimiter, CharSequence prefix, CharSequence suffix)

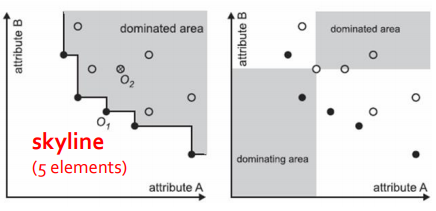
**Techniky pro vyhledávání textových, webových a multimediálních dokumentů: modely, algoritmy, aplikace. Optimalizace webových stránek pro vyhledávače**

BI-VWM

* **Dokument** – full-text objekt nebo anotace jiného objektu na webu
  + Kolekce = množina dokumentů
  + Term = slovo (fráze), které se objevuje v dokumentu
  + Slovník = množina všech různých termů v souboru dokumentů
* Prvky vyhledávání na webu
  + crawling – stahování obsahu
  + indexing – zpracování obsahu do formátu vhodného pro vyhledávání
  + searching – získávání relevantního obsahu pomocí query
* **Modely získávání informací z webu:** (často se kombinují)
  + *Query* – jednorázové hledání na základě přesného dotazu
    - algoritmus vrátí množinu vyhovujících dokumentů (seřazenou podle ranku nebo neseřazenou)
  + *Browsing* – iterativní navigace v databázi, neznáme přesný dotaz
    - manuálně prohledáváme explicitní (linky mezi entitami) nebo virtuální (série queries) graf
    - algoritmus vrátí podmnožinu db objektů, set clusterů nebo hierarchii
  + *Filtering* – formulace pevného pořadavku (explicitní – statický dotaz x implicitní – doporučování)
    - algoritmus vrací dynamicky měnící se odpověď pro explicitní dotaz (např. subscription kanálu na YouTube) nebo implicitní (např. doporučení na YT)
* Pro vyhledávání v souboru dokumentů je možné použít klasické string-matching algoritmy (např. Knuth-Morris-Pratt), ale ty jsou pomalé. Dokumenty se musí předzpracovat a indexovat.
* **Předzpracování kolekce**
  + Odstranění stop-slov, které nic neřeknou o obsahu (výplňková slova)
  + Lemmatizace – zjednodušení termů (např. uvedení slov do 1. pádu)
* **Měření kvality odpovědi**
  + Relevantní dokument – výsledkem dotazování na kolekci
  + Efektivita = míra uživatelské spokojenosti s výsledkem
  + **Precision** – pravděpodobnost, že dokument ve výsledku je relevantní
  + **Recall** – pravděpodobnost, že relevantní dokument je ve výsledku
  + **False alarm** – dokument není relevantní, ale byl tak vyhodnocen
  + **False dismissal** – dokument je relevantní, ale nebyl tak vyhodnocen
  + **P-R** křivka (precision-recall křivka) – ukazuje závislost precision a recallu – typicky kompromis
* **Booleovský model –** vytváří binární term-by-document matici
  + ****Dokument = množina termů x term = množina dokumentů, ve kterých se vyskytuje (lepší pro dotazování)
  + Řádek = vektor termů, sloupec = vektor dokumentů
  + 0 = term v daném dokumentu není, 1 = je
  + Hodně řídká matice, proto se implementuje pomocí invertovaných seznamů
  + **Invertovaný seznam** – každý term má vlastní list, ve kterém jsou uvedeny seřazené (podle ID) dokumenty, ve kterém se term nachází, výsledek je potom daný “merge sortem”
  + Operace AND, OR, NOT
  + Výsledky dokonale reflektují dotaz, člověk musí vědět co hledá
  + Pozitiva – jednoduchý na implementaci, efektivní pomocí invertovaného indexu
  + Negativa – všechny dokumenty mají stejnou váhu
* **Rozšířený booleovský model –** implementuje booleovský model + zavádí **váhy termů** podle toho, jak jsou v dokumentu důležité
  + kombinace booleovské algebry s vektorovým modelem
  + vyhazuje seřazené výsledky
  + umožňuje hledat i částečně relevantní dokumenty, ne jen plně relevantní
  + *pozitiva*: stále jednoduché pro uživatele, možnost specifikace velikosti odpovědi, efektivnější než standardní model
  + *negativa*: výpočetně náročné
* **Vektorový model =** bag of words model
  + Dokument = bag of terms (bag = multi-množina, query = bag of terms, jako dokument)
  + hledá podobné dvojice dokumentů
  + každý dokument je vektor dimenze *m* (*m* = # termů ve slovníku), který obsahuje váhu každého termu ze slovníku
  + každá dimenze prostoru patří nějakému termu ze slovníku
  + dotaz q se převede také na vektor a hledají se vektory jemu nejbližší na základě jeho směru (podle kosinové vzdálenosti), takže podobné vektory si budou blízké, odlišné naopak vzdálené – tím se dá odlišit, jaké dokumenty do odpovědi ještě zahrnout (nastavuje se nějaký threshold)
  + např. při hledání plagiátorství nebo vyhledávání v Googlu
  + opět velmi řídký, takže se znovu používá invertovaný index, kde každý term si v seznamu drží dokument a svou váhu v tomto dokumentu
  + *pozitiva*: lze vyhledávat i podle příkladu, efektivní a jednoduché, výsledky jsou seřazené
  + *negativa*: neporadí si s homonymy a synonymy, geometrizací se v modelu ztrácí sémantika informace, syntax není zahrnutá
* **LSI vektorový model** (latent semantic indexing) – úprava vektorového modelu
  + cíl je adresovat problém homonym a antonym a snížit dimenzi vektoru
  + využití metody SVD na předzpracování term-by-document matice (výsledná matice je **concept-by-document**, dokument tedy není modelován termy, ale koncepty)
  + tato matice se vynásobí s vektory v *U* reprezentující koncepty, které jsou seřazeny podle důležitosti (důležitost rychle klesá, takže se bere jen *k* konceptů)
  + dotaz se zobrazí do prostoru konceptů jako a opět proběhne porovnání podle kosinové vzdálenost
  + pozitiva: zahrnutá sémantika v dokumentu, částečně řeší homonym a synonyma, redukce dimenzionality (pomocí SVD)
  + negativa: drahé předzpracování, není vhodný invertovaný index, koncept nemá nic společného s lingvistikou, jen je to lineární kombinace termů
* **Word2Vec**: rychlejší, lepší
  + Machine learning přístup – nesupervizované učení na korpusu
  + Dva modely – continuous bag of words, skip-gramy
  + NLP
* **Optimalizace webových stránek pro vyhledávače** (SEO)
  + způsob úpravy webových stránek, aby byly dobře hodnocené vyhledávači
  + obsah stránky musí být přehledný pro člověka i stroj
  + hodnotí se každá stránka, nikoliv doména (nutnost optimalizace pro každou stránku)
  + hodnotí se především tyto prvky:
    - *název souboru* – krátké a relevantní (co nejkratší URL)
    - *<title> tag* – nejdůležitější informace o stránce, stručný a výstižný
    - *header tagy* - důležité informace pro vyhledávače i v tagách <h1> - <h6> (sestupně)
    - *meta tagy* – dnes už nejsou důležité, ale stále je to způsob, jak předat informace uživateli
    - *textové modifikátory* - <strong>, <b> ...
    - *optimalizace obrázků* – nevyužívat obrázek jako link, počítače jim zatím moc nerozumí, třeba vyplnit *alt* tag
    - *interní odkazy* – vytvořit interní strukturu linků, aby spideři mohli rychle navigovat
  + **Generování klíčových slov** je nejdůležitější pro SEO proces, nejlepší klíčová slova dávat na hlavní stránky, méně důležitá na podstránky, důležité je vložit na správná místa
    - Tři kroky:
      * Analýza potenciálního dopadu klíčových slov
      * Výběr vhodných klíčových slov
      * Umístění klíčových slov na správná místa
    - Nutný mix generických klíčových slov a specifických, dohromady tak 4
  + obsah stránky má být originální a kvalitní (duplikáty jsou penalizované), aktualizovaný a mít optimalizovaný ne-html dokumenty
* **Struktura webových stránek**
  + design: krátký název, tematická struktura, hierarchie linků
  + sitemap: pomáhá navigovat roboty na stránce
  + robots.txt: informace o tom, které stránky mohou být indexovány a které ne
  + kontaktní údaje a zásada ochrany osobních údajů: dělá to stránky důvěryhodnější
  + .htaccess: soubor s konfigurací pro Apache server
* **Analýza odkazů**
  + **Web graf** – uzly jsou webové stránky a orientované hrany URL adresy nalezené na stránce, které odkazují na jiné stránky (**inlink** – odchozí hrana, **outlink** – příchozí hrana)
  + **Hub** – stránka s hodně outlinky x **Authority** – stránka s hodně inlinky (lze být oboje)
  + **Podpora stránky** – web odkazuje na jiný web
  + **Relevantní stránky** – stránky odkazují jedna na druhou
  + **Ko-citace** – stránka odkazuje na několik stránek
  + **Sociální volba** – na stránku je odkazováno z několika stránek
  + **Tranzitivní podpora** – p1 odkazuje na p3 přes p2
  + **Hodnocení webových stránek** – hodnocení popularity na základě statistiky inlinků
    - **Page rank** = nezáporné reálné číslo vypočítané použe ze strukturálních informací (query independent, fulltext independent)
    - Hodnotící algoritmy: PageRank, HITS
    - **PageRank** – stránka je důležitá, pokud je na ni odkazováno jinými důležitými stránkami
    - **HITS** – stránka je dobrý hub, pokud odkazuje na dobré autority, a stránka je dobrá autorita, pokud na ni odkazují dobré huby
* **Backlinky** – dobré hodnocení u zajetých stránek činí stránky důvěryhodnější, dávat si pozor na nedůvěryhodné stránky (např pomocí PageRanku)
* **SMO (Social Media Optimization)** – dostat se do podvědomí přes sociální sítě
* **Google Analytics** – používané pro sledování celkového pohledu aktivit na stránce
* **Řazení stránek vyhledávačem** – hodnocení relevance stránky na základě uživatelských dat, strukturálního kontextu a behaviorálního kontextu
  + Funkce pro částečné hodnocení – založeny na relevanci obsahu, popularitě/reputaci stránky atd. (layout, click-data statistiky)
  + Koncové řazení
    - One-step ranking – globální hodnotící funkce agregující částečné hodnocení, top-k operátor
    - Multi-step ranking – postupný re-ranking používající částečná hodnocení
* **Neetické SEO –** způsob jak rychle a na krátkou dobu dosáhnout dobrého hodnocení od vyhledávače
  + Používá se pro spammování, často se odhalí a zabanuje stránku

**Vyhledávání v multimediálních databázích, podobnostní vyhledávání podle obsahu, podobnostní dotazování, agregační operátory, indexování metrické podobnosti, aproximativní vyhledávání**

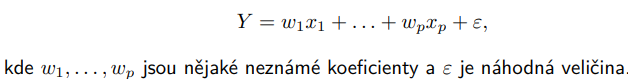
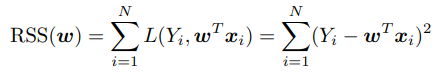
BI-VWM

* **Multimédia** – více než jeden typ digitálního média (obrazový, audio, video obsah)
  + Jakýkoliv typ nestrukturovaného média (fotky, data ze senzorů, hudba…) – hlavně digitalizované povahy, také lidmi zpracovaná data (text, XML, ilustrace, aplikace, ...)
  + Kontext – popis/anotace obsahu – sousední prvky, klíčová slova, GPS...
  + Obsah – pixely, rádiové vlny…
* **Textové vyhledávání v multimédiích**
  + Např. obrázek na FB – vyhledávání na základě popisu, tagů, komentářů, geolokaci uživateli (a jeho kontextu), albu (sousední prvky), převažující barvy
  + Výhody – stejná implementace jako u textového vyhledávání
  + Nevýhody – je subjektivní a nekompletní, může být nahrazeno machine learningem
* **Struktura databází**
  + Relační – pevná struktura i sémantika
  + Full-textové – pevná sémantika, volná struktura
  + Multimediální – volná struktura a sémantika (problém s dotazováním)
* **Podobnostní vyhledávání podle obsahu**
  + Pokud data nemají strukturu, ani sémantiku – musí se nějak interpretovat (raw obsah se musí dostat do vyšší úrovně informace)
  + Řeší extrakci příznaků a stanovení podobnostní funkce
  + **Extrakce příznaků –** transformuje multimediální objekt do strukturovaných deskriptorů
    - Získání deskriptorů:
      * Low-level features (computer vision, kognitivní psychologie)
      * Kombinace low-level features (kompetence prohledávání dat)
    - MPEG7 definuje vizuální, audio a pohybové deskriptory
    - Forma deskriptorů: vektor (histogramy), (un)ordered set (pro množiny a sekvence)
  + **Podobnostní funkce** – pro porovnávání deskriptorů
    - Má mimikovat sémantiku podobnosti originálních multimediálních objektů
  + **Modelování podobnostní funkce –** porovnává deskriptory, čím blíže, tím lepší vzdálenost
    - Typy vzdáleností:
      * Vektorové vzdáleností (Minkowského, kosinové, kvadratické vzdálenosti)
      * Adaptivní vzdálenosti (Hausdorffova vzdálenost)
      * Sekvenční vzdálenosti (editační vzdálenost)
  + Aplikace – image retrieval pomocí globálních i lokálních příznaků, shape retrieval pomocí časových řad (použití dynamic time warping vzdálenosti nebo longest common subsequence), audio retrieval pomocí waveform spectral information/melodie/score
* **Podobnostní dotazování –** na rozdíl od SQL se nelze dotazovat deskriptory
  + **Formalizovaný model –** procedura na extrakci příznaků + funkce vzdálenosti (nepodobnosti)
  + Dotazování příkladem – získá se multimediální objekt, extrahuje se deskriptor q, použije se podobnostní funkce a výsledky jsou buď z rozsahu, nebo kNN
  + Typy podobnostních dotazování
    - **Podobnostní řazení** – máme-li objekt q dotazování z universa deskriptorů U, získáváme databázi seřazených deskriptorů řazených podle vzdáleností jejich prvků od q
    - **Rozsahové dotazování** – máme poloměr vzdálenosti r (hranice nepodobnosti), rozsahové dotazování vrací všechny databázové deskriptory, jejichž vzdálenost od q je menší rovno r
    - **K nejbližších sousedů dotazování** – máme počet požadovaných deskriptorů k, kNN dotaz vrací databázi deskriptorů nejbližších ke q
* **Agregační operátory**
  + Databázové operátory = operace nad databází s výsledkem
  + Dotaz – příklad jako parametr, očekávaný výsledek je malá podmnožina databáze
  + Operátor – neparametrická operace – nad dynamickou databází, opakované zpracování neparametrického operátoru nad statickou databází vede ke stejnému výsledku, odpověď dlouhá
  + **Similarity join –** spojení deskriptorů databáze A s deskriptory db B
    - Spojení na základě vzdálenosti nebo kNN
    - Self-join: A=B (na detekci téměř duplicitních prvků)
    - Join založený na vzdálenostním dotazování (poloměr)
    - Join založený na kNN
  + **k nejbližších párů** – vzdálenostní funkce vybere k párů (x,y) z AxB, které mají nejmenší vzdálenost
    - Vhodné pro dynamické nebo streamované db
  + **Skyline operátor –** dotazování nad jedním příkladem nemusí být vypovídající, chce to více
    - K nalezení nejlepších prvků (ty, které dominují)
    - **Skyline** – podmnožina prvků, které nejsou dominovány jinými prvky
      * Prvek není **dominován**, jestliže neexistuje jiný prvek, který je lepší ve všech atributech
    - Problém – není limitován velikostí, neexistuje řazení
  + **Top-k operátor –** agregační procedura tvořící finální řazení na základě částečných
    - Máme vybrané atributy a agregační funkci a podle nich vybereme *k* nejlepších objektů
    - Top-k operátor vyhodnocuje agregační funkci na částečných „hodnostech“ (ranks) a vrací k objektů s nejvyšším agregovaným rankem
    - Dlouhé předzpracování, důležitá normalizace
    - **Faginův algoritmus:**
      * Seřaď data pro každý sloupec sestupně
      * Paralelně procházej jednotlivé sloupce a nalezené hodnoty zapiš do tabulky
      * Jakmile najdeš všechna data pro *k* objektů (napříč všemi sloupci), stop a najdi chybějící atributy pro ostatní objekty a doplň do tabulky
      * Použij agregační funkci a vyber *k* nejlepších
    - **Threshold algoritmus:**
      * U Fagina zbytečné procházení mnoha objektů
      * Zvol threshold
      * Procházej znovu sloupce a k tomu rovnou pro každý objekt doplň do tabulky i ostatní hodnoty a vyházej ty sloupce, které mají nejmenší číslo. Pokud je nejmenší číslo <= thresholdu, konči
* **Indexování metrické podobnosti**
  + Počítání podobnostní funkce je drahé a pro velké db nepoužitelné
  + Používá se indexování pomocí metrických vzdáleností mezi některými prvky, tím se prostor rozpadne na několik částí a my víme, ve které leží hledaný prvek. Tato část se projde lineárně.
  + Musí splňovat klasické metrické vlastnosti (symetrie, reflexivita, pozitivní definitnost a trojúhelníkovou nerovnost)
  + Rozdělením prostoru vznikne metický index
  + **Lower-bound distance** – pro filtrování prvků v db,
    - Využívá trojúhelníkové nerovnosti
    - P je pivot – statický objekt z databáze, známe jeho vzdálenosti od všech prvků
  + **Metric access methods (MAMs) –** pro efektivní vyhledávání v metrických prostorech
    - Používá jen vzdálenosti mezi prvky
    - Využívá lower-bounding filtrování při hledání
    - Architektury:
      * Flat pivot tables (LAESA – vektory vybraných pivotů, AESA – každý prvek je pivot)
        + Mapování dat na prostor pivotů, získáme matici vzdáleností
      * Hierarchické metody (ball-partitioning – M-Tree, hyperplane-partitioning – GNAT)
        + Dělení metrického prostoru a tvoření hierarchie regionů
      * Hashové indexy (D-index, prozradí, jestli je prvek vně/uvnitř kružnice)
        + Podobnostní hashovací funkce – každý objekt zahashován do bucketu, objekty sobě blízké by měly být v tom samém
      * Kombinace předchozích (PM-Tree, M-index) – pivot tabulky s hierarchickými indexy,...
      * Index-free metody
* **Aproximativní vyhledávání**
  + Přesné vyhledávání může být časově náročné, vystačíme si s aproximativním, které je mnohem rychlejší
  + Lidé taky neumí formulovat dotazy, takže je to leckdy v pohodě
  + **Indikátor indexovatelnosti** – na základě průměru a rozptylu řekne, jak dobře strukturovaná data jsou, pokud je vysoký MAMs jsou neúčinná
  + Aproximativita

1. garance, že je výsledek dotazování do nějaké míry relevantní
2. každý objekt ve výsledku je relevantní s nějakou pravděpodobností
3. bez garance – experimentální pozorování
   * **FastMap**
     + Používá kosovu vzdálenost a pythagorovu větu – projekce objektů na „pivot osu“
     + Algoritmus:
       - Od objektu vybere nejvzdálenější bod a od něj jeho nejvzdálenější
       - Udělá vektor p1, p2 a ořízne body za vektorem na straně, kde není původní objekt
       - Repeat
   * **M-strom s aproximativním kNN vyhledáváním**
     + Klasicky má rekurzivně zanořené kruhové regiony (menší a menší) a využívá prioritní frontu a pole kandidátů na nejbližší sousedy
     + 4 heuristiky:
       - Zpomalení vylepšení – pokud se pole mění málo, zastavíme
       - Přibližně správné vyhledávání – vrácení sousedé nejsou o tolik dál, než ti praví, nejbližší
       - Dobrá aproximace zlomků
       - PAC dotazy (přibližně aproximativní dotazy)
   * **Permutační indexy –** podobné pivot-table indexům
     + Místo pivot-table (matice vzdáleností) je pro každý databázový objekt permutace pivotů
     + V indexu nejsou uloženy vzdálenosti, ale pořadí pivotů

**Lineární regrese, regularizace pomocí hřebenové regrese**

BI-VZD

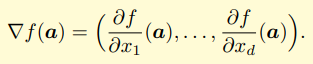
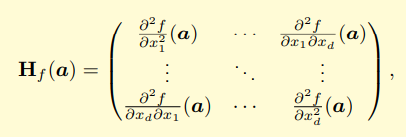
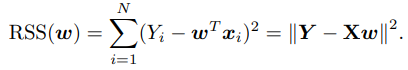
* **Lineární regrese** – na základě p příznaků X1...Xp predikujeme hodnotu **vysvětlované proměnné** Y
  + Předpokládáme **lineární závislost** vysvětlované proměnné na hodnotách příznaků
    - 
    - w1...wp – neznámé koeficienty
    - Ɛ – náhodná veličina – část Y, která je nevysvětlitelná pomocí hodnot příznaků – vlivy, které neznáme/nezahrnujeme
* **Model** lineární regrese – nepředpokládáme, že je závislost perfektní, že pro stejné hodnoty x1...xp příznaků X1...Xp dostaneme vždy stejnou hodnotu Y
  + 
  + Eε = 0
  + **Intercept** – koeficient w0 – očekávaná výchozí hodnota Y při nulových příznacích
  + Pokud zavedeme konstantní příznak X0 = x0 = 1 + vektorové značení x=(x0...xp)T, w=(w0...wp)T:
    - w = vektor vah
* **Predikce** v modelu lineární regrese
  + Odhad vektoru vah
  + Predikce Y v x:
  + Eε = 0 EY=wTx – za předpokladu Eε = 0 plyne, že odhad je bodovým odhadem střední hodnoty EY v bodě x, skutečná hodnota Y v bodě x má totiž navíc koeficient ε a je náhodná veličina
* Měření chybovosti pomocí **ztrátové funkce** – odhad vektoru parametrů
  + Chceme najít vektor vah w tak, aby byla chyba modelu co nejmenší – odhad
  + Měření chyby – **ztrátová funkce** – aplikace na skutečnou hodnotu Y a odpovídající predikci
    - Nezáporná funkce
    - Spojitá vysvětlovaná veličina kvadratická ztrátová funkce:
* **Metoda nejmenších čtverců**
  + Velikost chyby modelu v x je – v jakém x bychom měli vyhodnocovat a minimalizovat vzhledem k w?
  + **Minimalizujeme součet chyb** přes všechny body trénovací množiny (xi, Yi) pro i = 1...N
    - Součet chyb pro body pro kvadratickou ztrátovou funkci

= reziduální součet čtverců

* + Metoda nejmenších čtverců = minimalizace RSS(w) odhad
* **Extrémy funkce více proměnných**
  + w je vektor – minimalizace funkce více proměnných RSS(w)
  + **Parciální derivace** funkce f(x1...xd) podle xi v bodě a=(a1...ad)

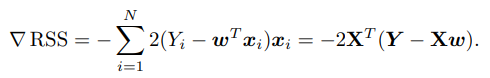
= derivace funkce g(xi) = f(a1...ai-1, xi, ax+1,...,ad): nebo

= funkce, která každému bodu, kde je konečná, přiřadí hod. parciální derivace podle xi v tomto bodě

* + **Gradient funkce** f v bodě a – vektor parciálních derivací pro všechny příznaky x1...xd v bodě a
    - Ukazuje směr maximálního růstu funkce v daném bodě, je kolmý na vrstevnici
    - Pokud má funkce lokální extrém a gradient existuje, musí být nulový
    - Je to zobrazení, které každému bodu, kde to lze, přiřadí gradient v tom bodě – vektor z , jehož složky jsou jednotlivé parciální derivace
    - f je funkce d proměnných, která má v bodě konečné parciální derivace
  + **Hessova matice** funkce f v bodě a – matice druhých parciálních derivací podle všech proměnných
    - Postačující podmínka pro existenci lokálního extrému:
* **Minimalizace RSS(w)** – metoda nejmenších čtverců = trénování
  + Problematika minimalizace funkce více proměnných

1. Vyjádření RSS:

2. Parciální derivace podle w0...wp: – dostaneme gradient

3. Gradient:

4. Položíme ∇RSS = 0 a získáme normální rovnici:



5. Hessova matice



* Je vždy pozitivně semi-definitní

 neostré lokální minimum v jakémkoliv bodě w, který řeší normální rovnici

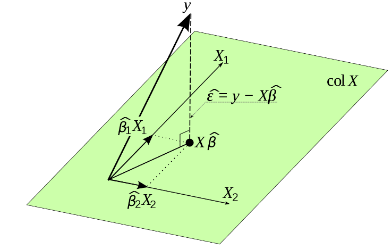
6. Předpokládejme, že XTX je **regulární matice** normální rovnice má **jednoznačné řešení**



* + - bod = bod ostrého lokálního minima, RSS v něm nabývá globálního minima
    - OLS = ordinary least squares solution
  + **Predikce v bodě x**:
* **Metoda gradientního sestupu** – k minimalizaci RSS(w), pokud jsou příznaky silně korelované (sloupce matice X skoro závislé)
  + Pomocí rekurentního
    - Postupně konstuujeme posloupnost vektorů, o kterém doufáme, že konverguje ke skutečnému řešení
    - = učící parametr, může záviset na i
    - Ukazuje směrem nejvyššího růstu díky zápornému znaménku děláme kroky ve směru nejvyššího poklesu
    - Pro vhodné metoda konverguje do globálního optima daného jinak se ale může i zaseknout v lokálním optimu
* **Vlastnosti lineární regrese**
  + Diskriminativní metoda – přímo odhadujeme
  + Rezistentní k problémům dimenzionality – je to parametrická metoda (ideálně pro p příznaků +1 intercept může k určení přesného modelu stačit p+1 bodů trénovací množiny)
  + **Geometrická interpretace**
    - Minimalizace je ekvivalentní minimalizaci pro optimální je Euklidovská vzdálenost a v nejmenší možná
    - i-tý sloupec matice

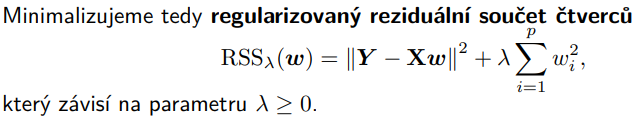
vektor je **lineární kombinací** sloupců matice X s koeficienty

leží v **lineárním podprostoru** prostoru , který je **obalem** sloupců

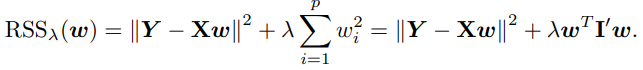
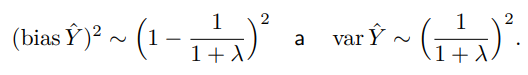
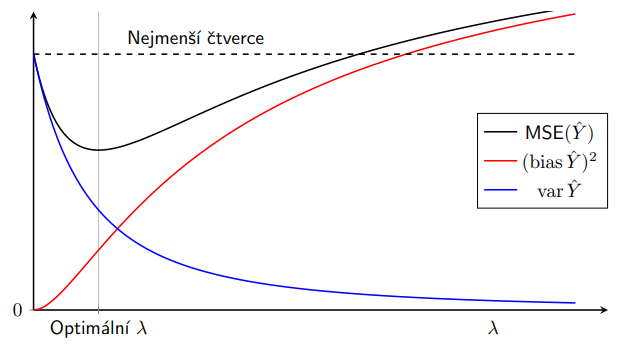
* + - Pro různé hodnoty pak celý prostor pokrývá:
    - Chceme-li **minimalizovat vzdálenost** od podprostoru sloupců matice (roviny), hledáme **bod v podprostoru sloupců matice X, který je k Y nejblíže**
      * je k  nejblíže, jestliže je vektor na ten podprostor kolmý

Je kolmý na všechny vektory x0...xp, které ho generují:

– **normální rovnice**

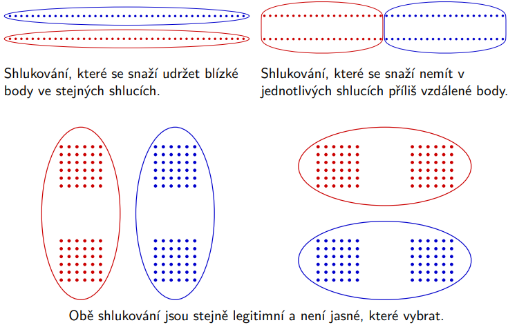
* + - * Pro jakékoliv řešení normální rovnice je a tedy i nejmenší možné **globální minimum**
    - Normální rovnice má **jednoznačné řešení**, pokud je regulární, to je právě tehdy když jsou sloupce matice **lineárně nezávislé**
    - **Problém kolinearity –** sloupce X jsou „skoro“ lineárně závislé – existuje lineární kombinace sloupců, které dávají téměř nulové vektory, zatímco jiné l.k. vrací mnohem větší vektory – inverze teoreticky existuje, ale problematický výpočet – odhad je citlivý na malé nevhodné změny Y velký rozptyl
      * Řešení **–** přigenerování dat, snížení počtu příznaků, změna minimalizované funkce, regularizace (k RSS přidáme regularizační člen – odstranění kolinearity)
* **Hřebenová regrese** = **L2 regularizace**
  + Řeší problém kolinearity zavedením **penalizačního členu** – úměrný kvadrátu normy vektoru koeficientů bez interceptu (w0 nepenalizujeme, protože se jedná pouze o posun)

minimalizujeme **regularizovaný reziduální součet čtverců** s parametrem

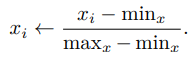
* + - normální RSS
    - v minimu budeme cílit na vektory w, které mají co nejmenší složky
  + po úpravě získáme reziduální součet čtverců
    - I´ = diagonální matice s x0,0 = 0
  + Gradient:
  + Hessova matice:
  + Normální rovnice:
  + Jednoznačným řešením je:
* **Bias-variance tradeoff**
  + Jelikož z trénovací množiny je v důsledku náhodnosti *ε* náhodný vektor, je i jakožto funkce náhodný vektor
  + **Očekávaná chyba** – měřená kvadratickou ztrátovou funkcí –
    - **Bayesovská chyba** – první člen – neodstranitelný, je daný náhodností modelu
    - **MSE** – druhý člen – střední kvadratická chyba odhadu parametru EY
  + Po úpravě MSE: = **rozklad očekávané chyby**
    - – vychýlení odhadu
  + U hřebenové regrese platí, že s rostoucím vychýlení roste a rozptyl klesá = **bias-variance tradeoff**
    - Hledáme optimální , kde je chyba nejmenší
    - Minimalizujeme MSE pomocí křížové validace
    - Obvykle nejdřív standardizace příznaků

**Metoda nejbližších sousedů, metriky, metody shlukové analýzy (k-nejbližších center, hierarchické shlukování)**

BI-VZD

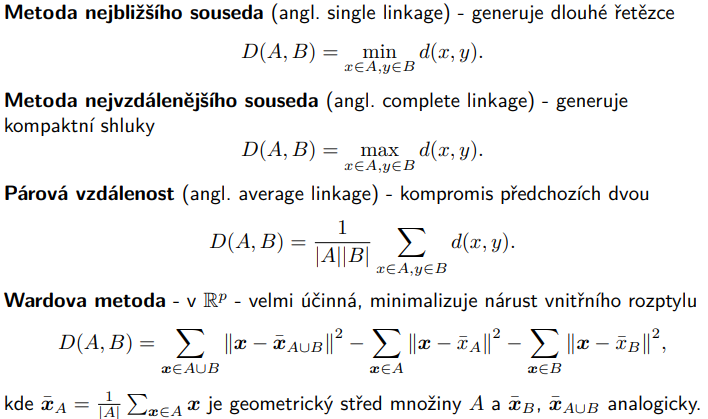
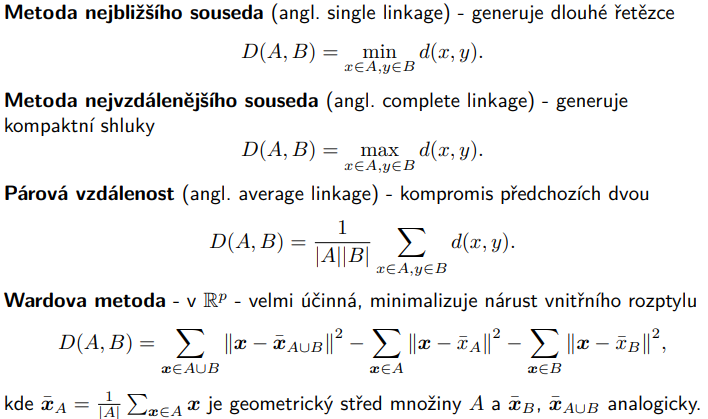
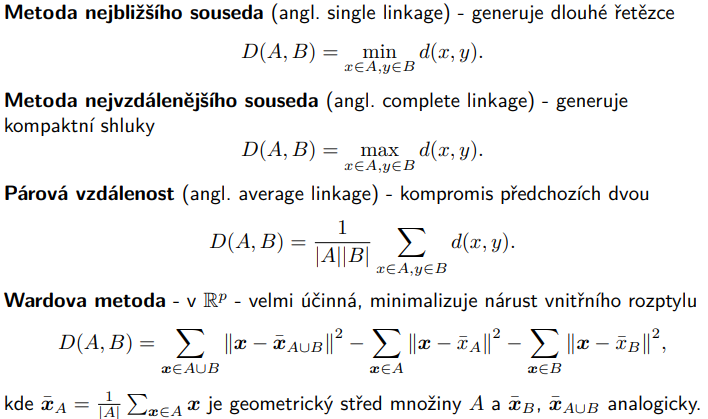
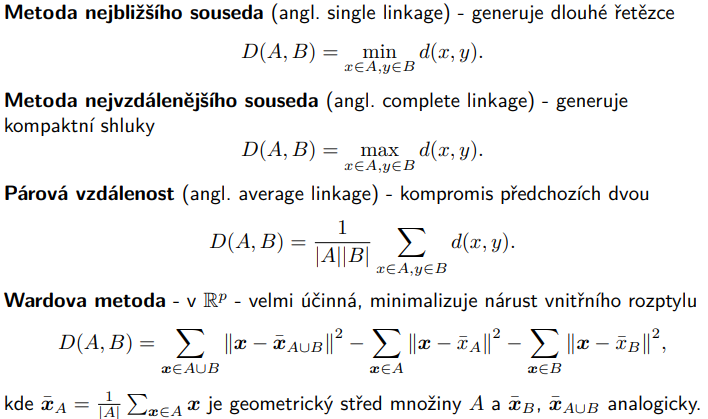
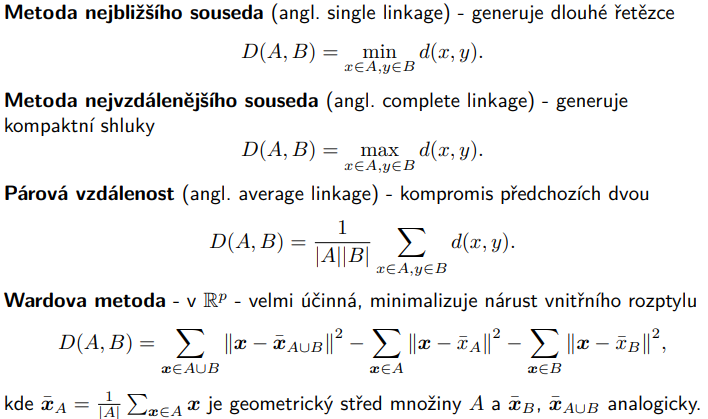
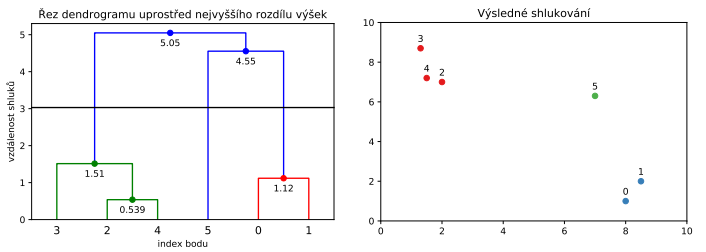
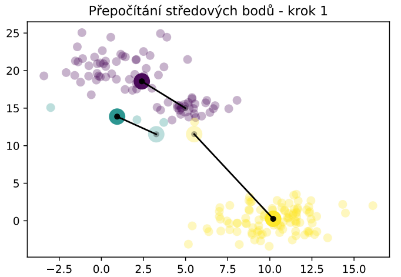
* **Nesupervizované učení** – nemáme veličinu, kterou bychom u trénovacích dat znali a snažili se ji predikovat
  + Cíl = porozumění struktuře dat na základě jich samotných
* **Supervizované učení** – trénovací data se známými hodnotami vysv. pr.
* **Shlukování** – clustering – třízení dat do skupin – shluků tak, že:
  + Blízké body budou ve stejném shluku
  + Vzdálené body budou v různých shlucích
* **Vzdálenost** – metrika na množině : funkce taková, že :

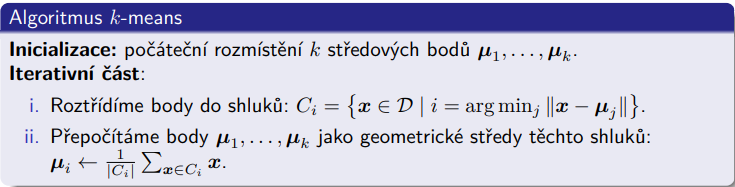
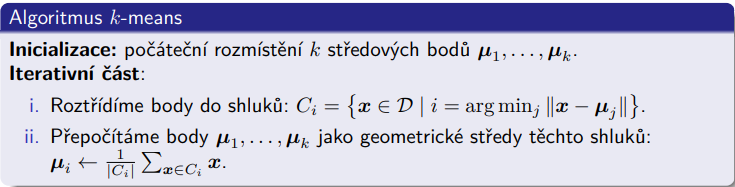
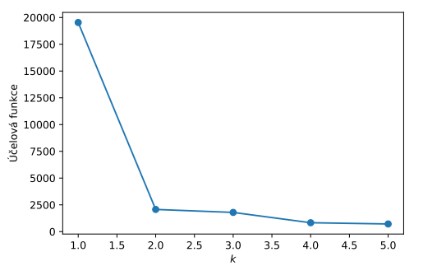
1. **Pozitivní definitnost**:
2. **Symetrie**:
3. **Trojúhelníková nerovnost**:
   * **Minkowského k-metriky**:
     + k=2: **Eukleidovská vzdálenost** ():
     + k=1: **Manhattanská vzdálenost** ():
   * **Čebyševova vzdálenost** ():
   * **Levenshteinova vzdálenost**: minimální počet změn z jednoho řetězce na druhý

* **Vstupy shlukován**í
  + Metrický prostor X se vzdáleností d
  + Množina dat
  + (obvykle) požadovaný počet shluků k
* **Výstupy shlukování**
  + Rozklad množiny D na jednotlivé shluky
  + Bod je tedy v i-tém shluku, jestliže
  + Grafický výstup pro hierarchické shlukování – dendrogram
* **kNN** – metoda nejbližších sousedů
  + chceme predikovat hodnotu vysvětlované proměnné pro datový bod
  + v trénovacích datech najdeme k bodů, které mají od x nejmenší vzdálenost
  + predikce založena na známých hodnotách vysvětlované pro těchto k bodů – supervizované učení
  + regrese průměr z hodnot x klasifikace nejčastější hodnota
  + trénovací data jsou sama o sobě naučeným modelem učení vlastně neprobíhá
  + výpočetně náročná predikce = hledání nejbližších sousedů (pomáhá indexace – vyhledávací strom)
  + **Hyperparametry**:
    - K – **počet nejbližších sousedů** – zvýšením k zabraňování přeučení
    - **Míra vzdálenosti** – nejobvyklejší Minkowského k-metriky, lze použít i vzdálenosti, které se v každé dimenzi chovají jinak
    - **Váhy nejbližších sousedů** – určují „sílu hlasu“ při predikci
      * Obvykle klesají se vzdáleností:
  + **Normalizace dat** – protože kNN je citlivá na typy jednotlivých příznaků
    - Normalizace každého příznaku do intervalu [0, 1] – zabraňování extrémním úletům způsobeným pestrou škálou příznaků a jejich významů
  + Nominální příznaky (např. číslo městské části) – řeší modifikace metriky nebo one-hot encoding
  + **Prokletí dimenzionality** – s přibývajícím počtem dimenzí (příznaků) exponenciálně roste vel. Prostoru
    - Řídnutí a vzájemné vzdalování se dat
    - Zmenšení rozdílů mezi vzdálenými a blízkými body
* **Hierarchické shlukování** – hladový aglomerativní algoritmus
  + N ... počet prvků množiny dat D

1. Začátek: každý bod = 1 shluk (N shluků)
2. Iterujeme:
   1. Najdeme 2 shluky, které jsou si nejblíž
   2. Spojíme je do nového shluku

po N-1 opakováních 1 shluk se všemi body

* + Třeba určit **zastavovací kritérium** – nejčastěji počet shluků k, příp. Lineární hodnota vzdálenosti shluků
  + Měření vzdálenosti shluků
    - Vstup = vzdálenost dvojice bodů d(x,y)
    - **Metoda nejbližšího souseda** (single linkage) – generuje dlouhé řetězce:
    - **Metoda nejvzdálenějšího souseda** (complete linkage) – generuje kompaktní shluky:
    - **Párová vzdálenost** (average linkage) – kompromis předchozích dvou
    - **Wardova metoda** – v – účinná, minimalizuje nárůst vnitřního rozptylu
  + **Dendrogram** – vizualizace hierarchického shluk. – strom, kde vrcholy = shluky, které při běhu vznikly
    - Listy = počáteční jednoprvkové shluky, kořen = finální shluk všech bodů
    - Na jeho základě můžeme zvolit finální počet shluků, můžeme měnit počet shluků
    - Časově náročné pro velké datové soubory
* **k-Means** – k nejbližších center
  + shlukování jako **optimalizační úloha** – definována **účelová funkce** – ohodnocuje daný rozklad množiny na jednotlivé shluky, cílem je nalézt rozklad, který **účelovou funkci minimalizuje**
  + Algoritmus:

1. počáteční rozmístění k středových bodů
2. Iterujeme:
   1. Roztřídíme body do shluků: Pro každý středový bod určíme jemu odpovídající shluk jako podmnožinu bodů, ke kterým je blíž, než všechny ostatní středové body
   2. Spočítáme nové středové body jako geometrické středy těchto shluků
   * V každé iteraci má účelová funkce stejnou nebo nižší hodnotu
   * Zastavení algoritmu = dostatečně malá změna hodnoty účelové funkce mezi iteracemi
   * Výsledek významně závisí na inicializační části – obvykle náhodné generování středových bodů
   * Algoritmus následně opakovaně spouštěn – finální výsledek = výsledek běhu s nejnižší ú. f.
   * Problematika volby k – nezbytné stanovit k dopředu
     + Optimální k = hodnota, pro kterou se mění pokles ú. f. Z hodně prudkého na méně prudký – hledání **lokte** – nehodí se vždy (další metoda např. silhouette)
   * Dokáže konvergovat k lokálnímu optimu, globální je NP-těžké
   * rychlejší než hierarchické shlukování, ale nutná volba k předem

**Naivní Bayesův klasifikátor, modely podmíněných pravděpodobností**

BI-VZD

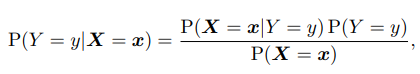
* **Klasifikace pomocí podmíněné pravděpodobnosti**
  + Klasifikační úloha, p diskrétních příznaků, chceme predikovat diskrétní vysvětlovanou proměnnou
  + Příznaky náhodný vektor s hodnotami X
  + Postup:

1. Odhad na základě trénovací množiny
2. **Využijeme pravděpodobnost k predikci Y:

= MAP odhad (maximum a posteriori)

= hodnota, která je při x nejpravděpodobnější

* **Využití Bayesovy věty** – jak odhadnout

1. Odhadneme
2. Podle Bayesovy věty:

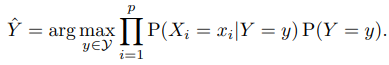
- potřebujeme odhad P(Y=y) – triviální

* + Pokud chceme argument maxima, můžeme zahodit P(X=x) – je pro všechny y stejný:

*∝ = „rovnost, až na násobek konstantní vzhledem k y“*

predikce:

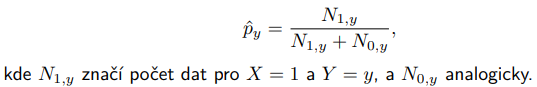
* + - Potřebujeme odhadnout – pomocí Naivního Bayesova klasifikátoru
* **Naivní Bayesův klasifikátor**
  + Předpoklad: za podmínky Y=y jsou všechny příznaky nezávislé

* + Naivita = pro fixní hodnotu vysvětlované proměnné předpokládáme, že jsou příznaky nezávislé
  + **MAP odhad**:
  + Díky rozkladu na součin marginálních podmíněných pstí jsou příznaky **separovány**

odhad podmíněné pravděpodobnosti každého příznaku probíhá nezávisle na ostatních

rezistence proti problémům s dimenzionalitou (k OK odhadu stačí málo dat i s nárůstkem příznaků)

* + Obvykle nepřesný predpoklad nezávislosti špatný odhad sdružené pod. psti
    - Nevadí, MAP bude OK, pokud má skutečná hodnota y vyšší odhadnutou pravděpodobnost, než ostatní hodnoty – často ano
* **Modely podmíněných pravděpodobností** – problematika odhadu , kde X je jeden z příznaků

1. **Bernoulliho rozdělení**
   * + Nejjednodušší situace -

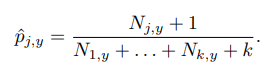
vhodné Ber. Rozdělení s parametrem

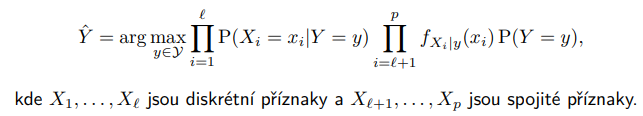
odhad

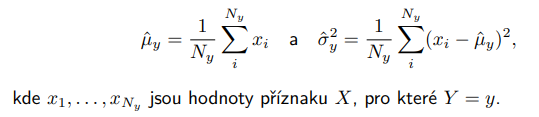
* + - * ...počet dat pro x = 1 a Y = y
    - Jedná se o maximálně věrohodný odhad – MLE param. Be. Rozdělení
    - Nevýhoda – v moc malé/velké nemusí být v trénovací množině pro Y=y zastoupeny obě hodnoty kolaps - (nebo 1)

nutné vhodné počáteční rozdělení

* + - * apriorní rozdělení odrážející naši expertní znalost, kterou hodláme na základě napozorovaných dat zpřesňovat
      * řešení = add-one smoothing (Laplace’s rule of succession) – netrpí kolapsem

1. **Kategorické rozdělení**
   * + X nabývá k různých hodnot c1...ck a podmíněnými pstmi p1,y, ..., pk,y - P(X = cj |Y = y) = pj,y
     + Opět je to MLE a trpí kolapsem, proto je získán robustnější MAP odhad k-rozměrného parametru
2. **Spojité rozdělení**

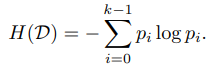
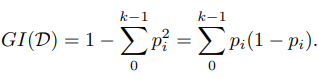
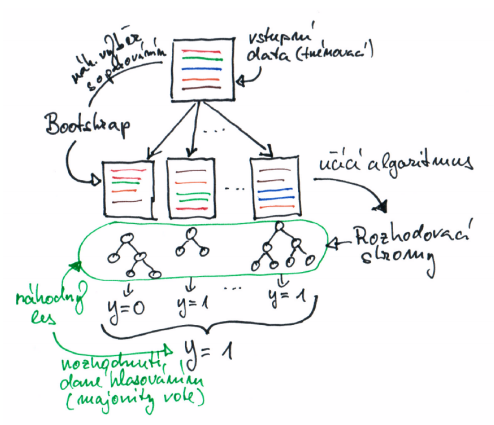
* X je spojitá náhodná veličina hustota místo podmíněné psti
* MAP odhad:
  + x1...xl ... diskrétní příznaky a xl+1...xp ... spojité příznaky

1. **Gaussovo rozdělení**
   * + Častým modelem podmíněného rozdělení je ve spojitém případě normální rozdělení se střední hodnotou a rozptylem
     + MLE odhady:
       - Kde x1...xNy jsou hodnoty příznaku X, pro které Y=y

* **Generativní přístup k predikci** – vytváříme model sdružené pravděpodobnosti
  + Např. Naivní Bayes
  + Model sdruž. Psti představuje úplnou informaci o rozdělení, ze kterého byla data „generována“
  + Může být použit pro generování nových pozorování
* **Diskriminativní přístup k predikci** – odhadujeme na základě trénovacího datasetu podmíněnou pravděpodobnost přímo, bez modelu pro sdruženou pravděpodobnost
  + Např. logistická regrese nebo neuronové sítě
* **Aplikace Bayesova klasifikátoru** – klasifikace textů – bag-of-words model

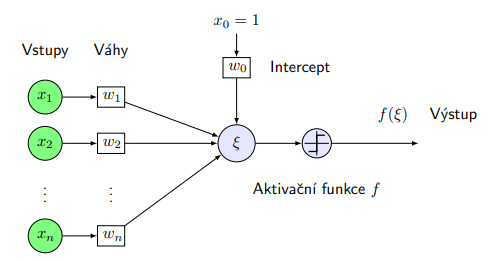
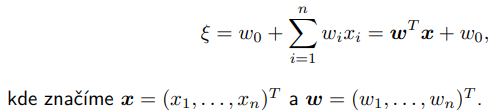
**Rozhodovací stromy: algoritmus konstrukce stromů, hyperparametry. Náhodné lesy**

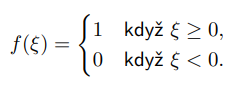
BI-VZD

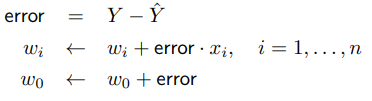
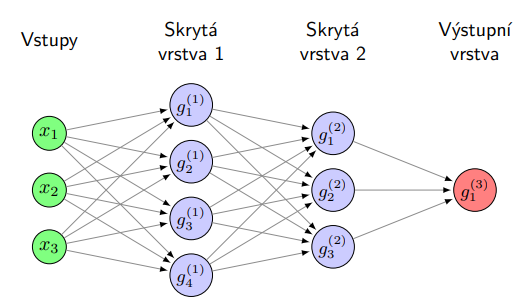
* **Supervizované učení –** zjišťujeme, jak vysvětlovanou proměnnou ovlivňují příznaky
  + Funkční vztah „aby co nejvíce platilo“
* **Rozhodovací strom** – strom, který má ve svých vnitřních uzlech pravidla a v listech předpovězenou hodnotu
  + klasifikační (výsledná předpovězená hodnota je ta s největším zastoupením v listu) x regresní (průměr hodnot)
  + velmi často binární
  + pokud listy stromu obsahují všechny kombinace hodnot příznaků, není to strom, ale index (strom by se totiž nemýlil, nereálné pro velká data)
  + kvalita se měří rozdělením dat na trénovací a testovací množinu a pak podle nějaké metriky se měří chyba (např. RMSE)
* **Konstrukce stromu**
  + Vstup – N-řádková tabulka s hodnotami pro binární vysvětlovanou proměnnou, p binárních příznaků
  + Cíl – strom hloubky k, který správně přiřadí hodnotu Y co nejvíce řádkům z tabulky
  + Konstrukce optimálního stromu je NP-úplný problém
* **ID3 algoritmus** – suboptimální hladový algoritmus
  + 1 (nepoužitý) příznak dělí data na 2 části tak, že vzniklé rozdělení maximalizuje vybrané kritérium
  + Množina dat rozdělena 2 části na každou zvlášť aplikován stejný postup výsledkem další 2 příznaky použité jako kritérium rozdělení na 4 podmnožiny dat
    - Dokud nenastane zastavovací kritérium (max. hloubka stromu, nedostatek dat, ...)
* **Stromy pro klasifikaci**
  + **Entropie** – funkce míry neuspořádanosti
    - Pro rozdělené množiny příznakem při konstrukci stromu
    - Maximální, pokud počet 1 a 0 je stejný (všechny hodnoty mají stejné zastoupení)
    - Nulová (minimální), pokud jsou v množině zastoupeny pouze stejné hodnoty
    - p ... poměr počtu hodnot v množině
  + **Gini index** – použití místo entropie pro množinu D s k různými hodnotami
    - Míra toho, že nově přidaný prvek bude špatně klasifikován
  + **Informační zisk** – pro výběr příznaku, který rozdělením dat nejvíce sníží neuspořádanost
    - Entropie mínus vážený součet entropií podstromů vzniklých rozdělení
    - t = počet prvků v podstromu / celkový počet prvků
    - lze použít Gini index i Entropie
    - lepší je vyšší
    - pokud je hodnota IZ pro různé příznaky vyjde stejná (a nejvyšší), můžeme si libovolně vybrat
    - v případě spojitých příznaků se používají pravidla porovnání X < xi (např. pro každé 10. x, pokud je hodnot až moc)
  + **Klasifikační přesnost** – počet správně klas. dat/počet všech dat
* **Stromy pro regresi** – spojitá vysvětlovaná proměnná – nelze využít entropie/gini index
  + **MSE – „mean squared error“** – odchylka od střední hodnoty (skoro výběrový rozptyl)
    - Chceme, aby hodnoty vysvětlované proměnné v rámci listu byly co nejblíže střední hodnoty
  + Algoritmus **CART** – hladový algoritmus, vrcholy stromu se volí tak, aby se **minimalizovalo MSE**
    - Místo MSE – MAE – místo čtverce vzdálenosti absolutní hodnota
* **Hyperparametry** rozhodovacího stromu – to, co určuje tvar nebo komplexnost modelu
  + Maximální hloubka stromu, minimální počet dat v množině, minimální nutná hodnota informačního zisku, gini/entropy kritérium, ...
  + Ladí se např. křížovou validací a měří se chyba (RMSE, MSE, ...), data se dělí na testovací a trénovací množiny (příp. Validační)
  + Počet náhodně vybraných příznaků, ze kterých si hladový algoritmus vybere, podle kterého se bude větvit
* Výhody rozhodovacích stromů – nenáročnost na přípravu dat, jednoduché, rychlé učení, dobře interpretovatelné, poradí si s kategoriálními i spojitými příznaky, s chybějícími hodnotami
* Nevýhody rozhodovacích stromů – nerobustní (malá změna dat – velká změna struktury stromu), většinou pouze binární stromy, optimalita NP-úplná, snadné přeučení
* **Náhodné lesy**
  + **Ensamble metoda** – místo 1 modelu více modelů a následní kombinace predikcí do fin. Rozhodnutí
    - Bagging (náhodné lesy) x boosting (AdaBoost)
      * Boosting – vážená data, strom se učí, aby správně predikoval hlavně datové body s vyšší vahou, Ada Boost = zvyšování váhy bodům, které v předchozím stromu byly klasifikovány špatně
  + Bagging = bootstrap aggregating
  + **Bootstrap** – výběr s opakováním – zajišťuje důležitou pestrost modelů
    - Chci dataset velikosti 5 – náhodně 5x vyberu řádek z tabulky – můžou se opakovat
  + **Náhodný les pro klasifikaci**
  1. Ze vstupního D n datasetů pomocí bootstrapu
  2. Na každém Di naučení rozhodovacího stromu T1...Tn
  3. Každý datový bod (řádek z D) proženeme stromy T1...Tn a uložíme jejich rozhodnutí Y1...Yn
  4. T1...Tn = náhodný les, finální rozhodnutí Y = většinové rozhodnutí stromů
  + Náhodný les pro regresi – analogicky, predikce = průměr predikcí stromů
  + **Hyperparametry**:
    - Počet stromů v náhodném lese
    - Maximální hloubka stromů v lese
    - Počet příznaků, ze kterých si hladový algoritmus vybírá ten, podle kterého se bude větvit (+ pestrost modelů)
  + Výhody – robustní, odolné vůči přeučení
  + Nevýhody – ztrácejí jednoduchost a snadnou interpretovatelnost

**Neuronové sítě, struktura perceptronové sítě, výpočet výstupu neuronu, učení perceptronové sítě**

BI-VZD + BI-ZNS

* **Neuronové sítě** – inspirovány neurony živých organismů
  + Schopnost extrakce a reprezentace závislostí na nezřejmých datech
  + Neuron = stavební kámen, jednoduchá výpočetní jednotka
  + Přijímá několik vstupů, produkuje jeden výstup
  + Schopnost řešit nelineární úlohy, učit se, zevšeobecňovat
* ****Jednovrstvý perceptron** – nejjednodušší model NS určený ke klasifikaci – 1 umělý neuron
  + Výstup neuronu aplikace nelineární aktivační funkce f na hodnotu vnitřního potenciálu *ξ* daného součtem vstupů x1...xn pronásobených vahami w1...wn a interceptu w0 (bias)
  + **Vnitřní potenciál**:
  + Výstup perceptronu:



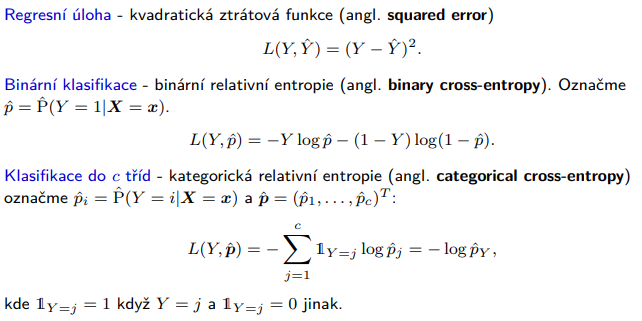
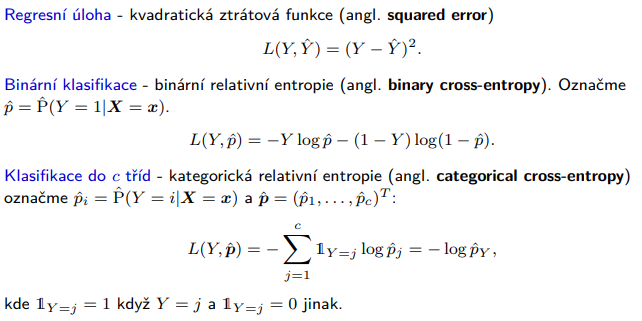
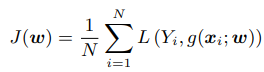
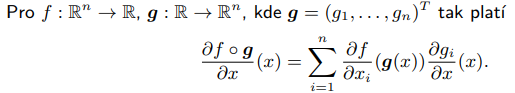
* + - f ... aktivační funkce – u perceptronu skoková funkce
    - tento výpočet = dopředný chod
    - neuron je aktivován, f(*ξ*) = 1, pokud
      * = prahová hodnota
  + Prostor vstupů rozdělen na 2 lineárně separované poloprostory, kdy v jednom z nich je neuron aktivní a v druhém ne. Váhy určují nadrovinu, která tyto 2 poloprostory separuje
  + Dopředný a zpětný chod
    - **Dopředný chod** – pro zadané váhy můžeme na základě hodnot vstupu x spočítat výstup
    - **Zpětný chod** – na základě hodnot získaných v dopředném chodu určíme chybu predikce a na jejím základě provedeme inkrementální update vah
      * ... predikce v bodě x trénovací množiny
      * ... skutečná hodnota
* **Vícevrstvá neuronová síť**
  + **Vícevrstvý perceptron** – síť se skládá z vrstev, které jsou propojené tak, že výstupy neuronů z jedné vstvy tvoří vstupy neuronů do další vrstvy
  + **Skryté vrstvy** – všechny vrstvy kromě té výstupní, jsou příznaky pro další vrstvy
  + Preference hlubších sítí – sofistikovanější a zajímavější příznaky
  + Hluboké učení – sítě s více než 3-5 skrytými vrstvami
  + Zvýšení počtu vrstev zvyšuje flexibilitu sítě, přidání neuronů snižuje výkonnost
  + Síť s jednou skrytou vrstvou dokáže na výstupu s libovolnou přesností aproximovat jakoukoliv spojitou funkci v
* **Učení vícevrstývch sítí**
  + **Dopředný chod** – uvažujme vrstvou neuronovou síť a označme *n1, ..., nl* počty neuronů v jednotlivých vrstvách a počet vstupních proměnných jako *n0.* Celá neuronová síť při dopředném chodu je reprezentována funkcí *,* která vznikne složením jedn. vrstev
  + **Zpětný chod** – algoritmus zpětného šíření chyby – back-propagation
    - NS jako funkce parametrů musí být skoro všude diferencovatelná
    - Použití gradientního sestupu
  + Docílení diferencovatelnosti – volba vhodné aktivační funkce pro skryté vrstvy
    - **RELU** – oříznutá lineární funkce
    - **Hyperbolický tangens**
  + **Aktivační funkce pro výstupní vrstvy** – převod napočítaných hodnot na hodnoty použitelné k predikci
    - Regresní úloha – predikce spojité proměnné
      * Ve výstupní vrstvě 1 neuron bez aktivační funkce -
    - Binární klasifikace
      * Ve výstupní vrstvě 1 neuron
      * Aktivační funkce = **sigmoida** – logistická funkce
      * Hodnota = pravděpodobnost příslušnosti ke třídě 1:
    - Klasifikace do c tříd
      * Ve výstupní vrstvě c neuronů
      * Aktivační funkce = **softmax**



* + - * + ... vektor vnitřních potenciálů c neuronů
        + ... aktivační funkce i-tého neuronu

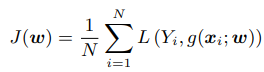
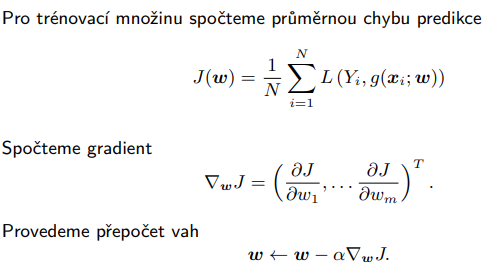
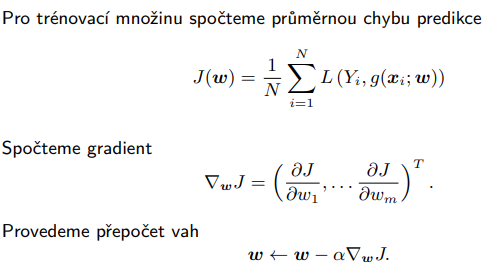
Výstup = pravděpodobnost příslušnosti ke třídě i

predikce v x:

* **Učení neuronových sítí** – minimalizujeme chybu predikce – měření pomocí průměrné hodnoty ztrátové funkce na trénovací množině
  + **Ztrátová funkce** – měří, jak dobře model predikuje konkrétní hodnotu z trénovací množiny
    - Regresní úloha – kvadratická ztrátová funkce
    - Binární klasifikace – binární relativní entropie
    - Klasifikace pro c tříd – kategorická relativní entropie
* **Učení gradientním sestupem**
  + Minimalizace chyby predikce měřené pomocí průměrné hodnoty ztrátové funkce L vzhledem k parametrům sítě w, které se vyskytují ve funkci g
  + **Minimalizace gradientním sestupem**
    - Výpočet gradientu:

pronásobování a sčítání parciálních derivací vrstev ve směru od výstupní vrstvy směrem ke vstupní = **zpětné šíření**

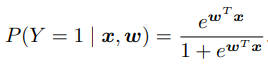
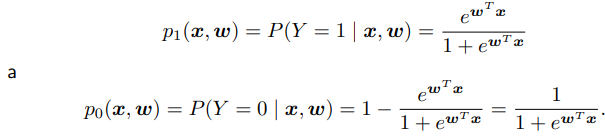
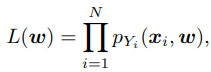
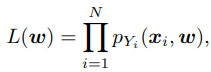
* + - Při standardním gradientním sestupu se počítá chyba přes celou trénovací množinu, což může představovat problém z pohledu paměti u velkých datasetů a mnoha parametrů, proto existují různá trénovací schémata
* **Dávkové učení = batch training** – počítá chybu přes celou trén. množinu, pak až krok proti směru gradientu
  + Neuronová síť s parametry a trénovací data
  + Postup učení:

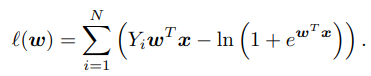
1. Inicializace w jako náhodná malá čísla
2. Opakujeme, dokud nesplní kritéria k zastavení:
   * 1. Pro trénovací množinu spočteme průměrnou chybu predikce
     2. Spočteme gradient
     3. Přepočet vah
   * Trénink v **minibatches** – výběr z několika bodů (přes ně výpočet chyby) a poté update vah
   * **Online training** – update pro každý bod zvlášť

* **Konvergence** – gradientní sestup míří v prostoru parametrů do míst, kde je gradient nulový
  + Jednovrstvá síť globální minimum
  + Vícevrstvá síť můžeme uváznout v lokálním minimu/sedlovém bodě (řídce rozmístěny, jejich hodnota je blízko globálnímu minimu
* **Regularizace** – kvůli problému s přeučováním u hlubokých NS
  + Přidání členu penalizujícího velikosti vah k účelové funkci
  + Dropout – náhodné vynulování některých neuronů
  + Nedostatek dat Data augmentation – tvorba více trénovacích dat s poruchami (otočení, škálování)
  + Využití předtrénované sítě
* Příklady architektur:
  + **Autoenkodér** – enkodér + dekodér (pro kompresi dat redukce dimenzionality)
    - Enkodér – tvorba abstraktních příznaků
    - Dekodér – návrat do pův. prostoru příznaků a replikace vstupních dat
    - Detekce odlehlých hodnot (velká rekonstrukční chyba)
  + **Konvoluční NS** – neurony spolupracují lokálně z toho skládání celku – sady vrstev, které počítají vhodné diskrétní konvoluce vstupů, využití okolních bodů pro rozpoznávání (např. klasifikace obrazu)
  + **Rekurentní NS** – zpracování posloupnosti dat (analýza textu, ...)
    - při výpočtu výstupu využití výstupů předchozího časového kroku – paměťové buňky
    - skryté stavy podchycení dlouhých časových závislostí

**Logistická regrese**

BI-VZD

* Určena pro klasifikaci, zde omezení na **binární klasifikaci**
* Rozhodnutí se jako u lineární regrese konstruuje pomocí lineární kombinace příznaků *w0 + w1x1 + ...+ wpxp*
* Z diskrétního problému dělá problém **spojitý** – namísto hodnoty vysvětlované proměnné predikujeme pravděpodobnost, že Y má hodnotu 1 (tzn. P(Y = 1) z intervalu [0, 1]):
  + je závislá na hodnotách příznaků x=(x1...xp) a koeficientů w=(w1...wp)
* Platí součet pravděpodobností – Stačí určit jedno
* **Logistická funkce**
  + Jak donutit *w0 + w1x1 + ...+ wpxp*, aby neutekl z intervalu [0, 1]?
  + Dosadíme ho do vhodně zvolené funkce, jejíž obor hodnot je podmnožina intervalu [0, 1]
    - Vždy dostaneme číslo smysluplně vyjadřující pravděpodobnost
* Obvyklá volba log. funkce – **Sigmoida**:
  + , za x dosazujeme *w0 + w1x1 + ...+ wpxp*
  + – nemůže být P(Y=1) = 1 nebo P(Y=0) = 1
  + Ostře rostoucí na , prostá, lichá, f(0)=1/2, limity pro x → −∞ a x → +∞ jsou 0 resp. 1
* **Model logistické regrese**
  + Binární vysvětlovaná proměnná Y s hodnotami 0 a 1, p příznaků X1, ..., Xp s konstantním X0 = 1
  + hledáme model pro odhad pravděpodobnosti, který pro x a koeficienty w má tvar:
  + Predikce – pro x se spočítá odhad pravděpodobnosti, je-li > ½, Y=1 – hranice rozhodnutí je 1/2
* **MLE odhad** parametrů w – metoda maximální věrohodnosti
  + Pravděpodobnost i-tého datového bodu s hodnotou vysvětlované proměnné Yi a s hodnotami příznaků xi pro zadané hodnoty parametrů w:
  + ****Předpokládáme vzájemnou nezávislost datových bodů
  + **Pravděpodobnost** konkrétních trénovacích dat lze zapsat jako součin pstí jednotlivých datových bodů při daném w – tuto funkci **maximalizujeme**
  + Odhad MLE odpovídá hodnotě p, pro která jsou trénovací data nejpravděpodobnější možná
* **Postup MLE**:

1. Hledáme **maximum fce**
2. **Logaritmujeme** – protože maximalizujeme pravděpodobnost, ale nezajímají nás konkrétní hodnoty – hledáme maximum funkce
3. Najdeme **gradient**

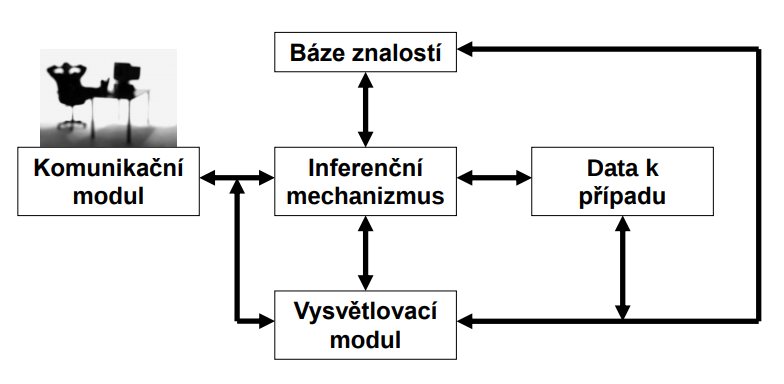
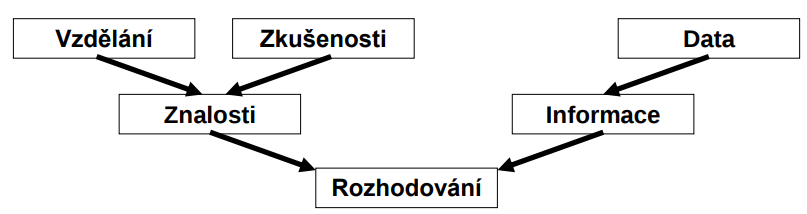
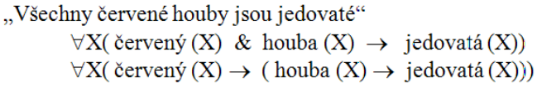
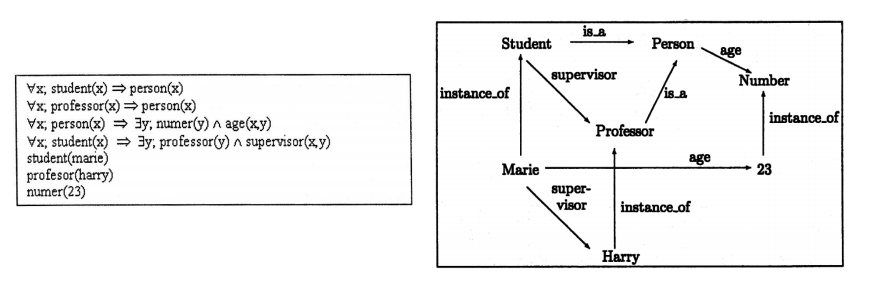
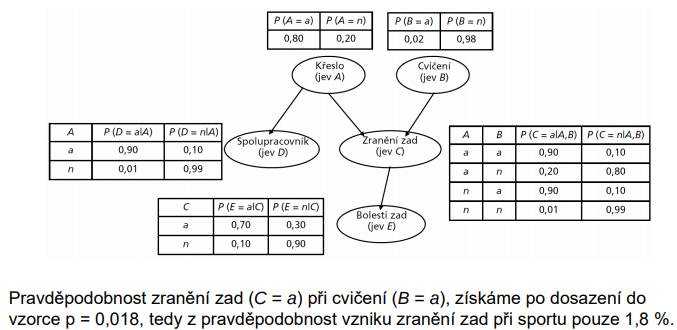
P ... vektor plný sigmoid se všemi exponenciálami

1. Položíme gradient nule, abychom našli **maximum**

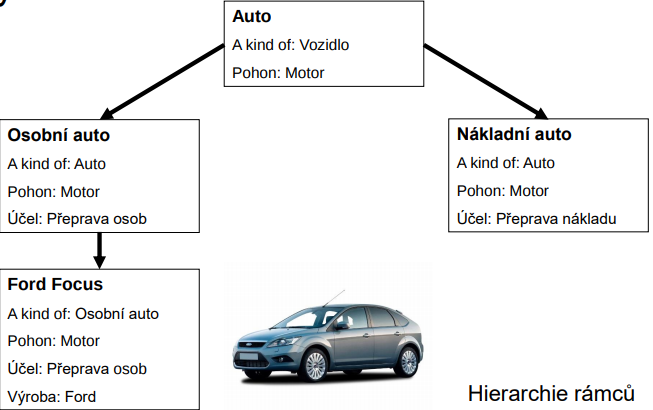
* Neumíme najít explicitní řešení jako u lineární regrese → je nutné použít numerické aproximativní metody (vícerozměrná verze Newtonovy metody nebo gradientní vzestup, které konvergují k lokálnímu maximu)
* stojí na předpokladu, že se chování dat dá zachytit ve tvaru daném sigmoidou jakožto fci *w0 + w1x1 + ...+ wpxp*
* výpočetně náročné a odhad není dán explicitně (počítač vrátí aproximaci nebo nic)

**Architektura znalostního systému, možnosti reprezentace znalostí**

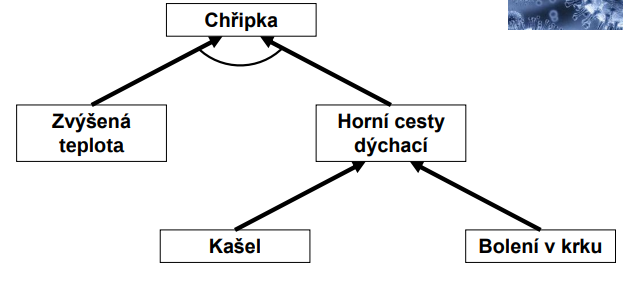
BI-ZNS

* **Znalostní systém** = inteligentní počítačový program, který využívá znalosti a inferenční procedury k řešení problémů, které jsou natolik obtížné, že pro své řešení vyžadují významnou lidskou expertízu
  + Simuluje rozhodovací činnost experta (podtyp – expertní systém)
  + V současnosti pohled na expertní systémy jako na podpůrný systém (asistent) ZNS – „nadstavba“ ES, automatizované získávání dat a znalostí
* **Charakteristické rysy ZS**:
  + Oddělení báze znalostí a mechanismu jejich využívání – databáze pravidel + „prázdný“ engine
  + Rozhodování za neurčitosti – podle pravděpodobnosti – důvěryhodnosti pravidel (použití heuristik)
  + Schopnost vysvětlování – zobrazení cesty, dialogový režim (možnosti)
  + Modularita a transparentnost báze znalostí
    - Kvalita přímo úměrná kvalitě báze znalostí
* Typy ZS:
  + **Prázdný ZS** (shell) – báze znalostí je prázdná (prolog)
  + **Problémově orientovaný ZS** – báze znalostí obsahuje znalosti z určité domény
  + **Aplikačně orientovaný** – obsahuje i konkrétní znalosti k řešení
  + **Diagnostický** – určuje, která z hypotéz koresponduje s daty z konkrétního systému
    - Diagnóza, interpretace, monitorování dat
  + **Generativní** – hypotézy generovány dynamicky, splnění úlohy s omezeními na řešení
    - Plánování (nalezení posloupnosti k dosažení cíle), návrh (tvorba konfigurací), predikce
* Architektura ZS
  + **Báze znalostí** – zakódované znalosti experta (pravidla)
  + **Inferenční mechanismus** – odvozovací mechanismus (odvozování závěrů)
  + **Báze dat** – ke konzultovanému případu
  + **Vysvětlovací modul** – vysvětluje a zdůvodňuje své rozhodnutí uživateli
  + Rozhraní pro komunikaci s uživatelem (**GUI**)
* **Báze znalostí**
  + **Data** – získána automatickým procesem, lze objektivně verifikovat
  + **Znalosti** – získány od experta, nelze objektivně verifikovat (zkušenosti, vzdělání, ...)
* Metody reprezentace znalostí
  + **Predikátová logika** – pro automatické dotazování – Robinsonův rezoluční princip
    - Jednoduchá reprezentace a aktualizace znalostí
    - Znalosti zapsány v jazyce predikátové logiky
  + **Sémantické sítě** – popis reality pomocí objektů a jejich vztahů (relací)
    - Relace vyjadřování znalostí
    - Přirozená reprezentace pomocí grafu (uzly – objekty, hrany – relace)
    - Kompaktní, obtížná změna struktury
  + **Bayesovské sítě** – popis pravděpodobnostní sítě vzájemných vazeb
    - Odhad pravděpodobnosti nastání jevu
    - Postaveny na podmíněné nezávislosti náhodných jevů





* + **Rámce** – reprezentují stereotypní situace – statické znalosti
    - Vyplňování stránek, do kterých se zapisují hodnoty položek
    - Hierarchie pojmů (a kind of) nebo dekompozice (part of)
    - Dají se zobrazit grafem
    - Objekty (jako v OOP)
  + **Pravidla** – nejběžnější způsob reprezentace
    - Pomocí IF-THEN pravidel (žádné else není)
    - **Tvrzení** (v předpokladu nebo závěru) může mít podobu:
      * Výrok – auto je červené
      * Dvojice (atribut, hodnota) – barva\_auta = červená
      * Trojice (objekt, atribut, hodnota) – auto\_32: barva = červená
    - Atributy mohou být numerické nebo kategoriální (binární, nominální, ordinální)
    - Pravidla v bázi snalostí se mohou znázornit **AND/OR grafem**
      * Uzly – výroky, orientace hran – pravidla
      * Konjuktivní vazba se znázorňuje obloučkem mezi hranami
    - 3 typy tvrzení – uzlů:
      * Dotazy – pouze v předpokladech
      * Cíle – pouze v závěrech
      * Mezilehlá tvrzení – v předpokladech i závěrech

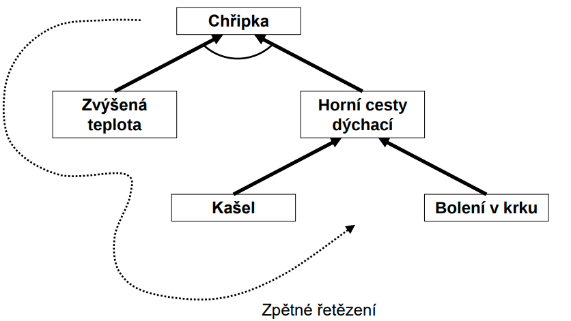
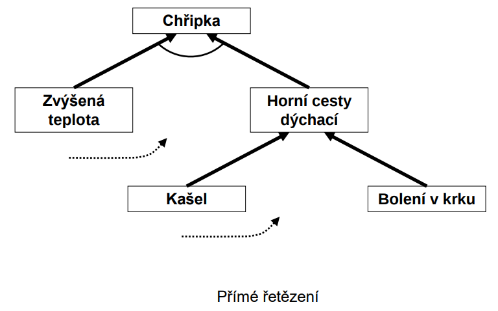


**Inferenční mechanismus, způsoby realizace inferenčního mechanismu**

BI-ZNS

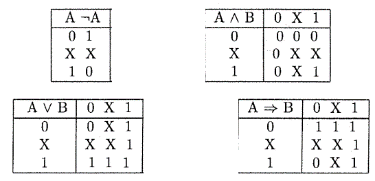
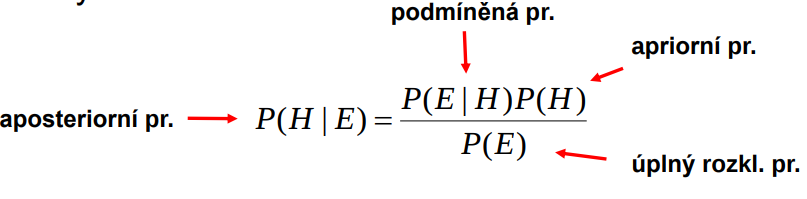
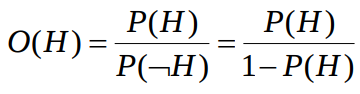
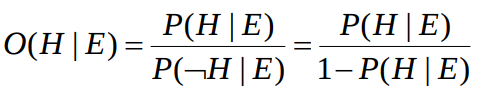
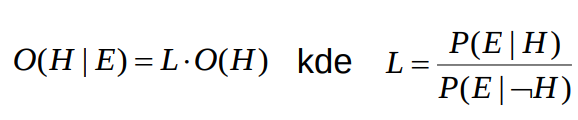
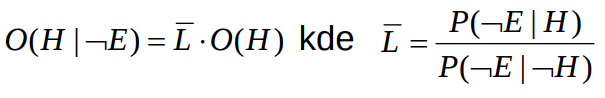
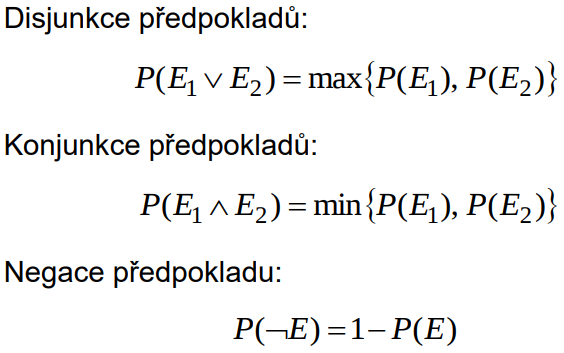
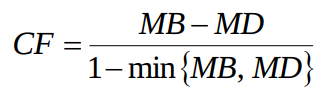
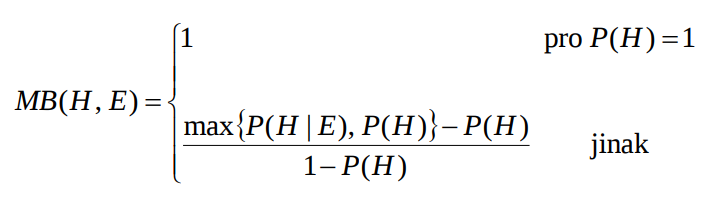
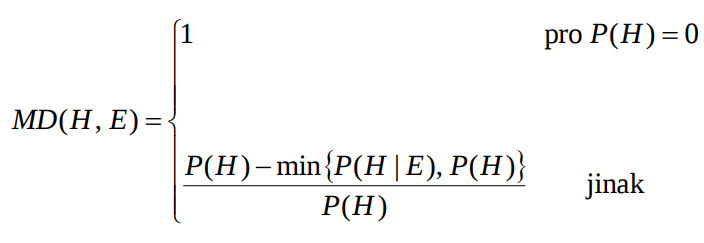
* **Inferenční mechanismus** – tvoří jádro ZS, „vyvozovací mechanismus“
* **Metody inferenčního mechanismu**
  + **Logické metody** – dedukce, abdukce a indukce vychází z implikace z výrokové logiky
    - Převádíme znalosti do matematické podoby výrokové logiky (pravidla)
    - Vyvozujeme závěry ze souboru pravidel
    - **Dedukce** 
      * platí pravidlo (implikace) a platí předpoklad odvozujeme platnost závěru (modus ponens) – A, (A ⇒ B | B)
      * platí pravidlo a neplatí závěr odvozujeme neplatnost závěru (modus tollens)

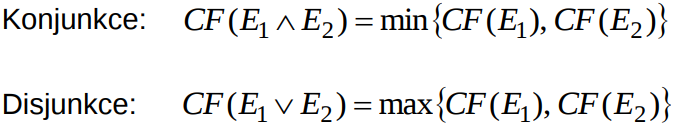
– (¬B, A ⇒ B | ¬A)

* + - * jestliže nemůže současně platit A a B a platí A, nemůže platit B (modus ponendo tollens) – B (¬(A∧B)∧A⇒¬B)
    - **Abdukce** – platí pravidlo a platí závěr
      * předpoklad může a nemusí platit domníváme se, že může platit (B, A ⇒ B | A)
    - **Indukce** (generalizace z příkladů) – opakované pozorování, že A a B se vyskytuje současně
      * odvozujeme, že mezi A a B je implikace – A ⇒ B nebo B ⇒ A
  + **Rezoluční metoda** – vyvozování závěrů ze souboru pravidel
  + **Zpětné řetězení** (backward chaining), též usuzování řízení cíli – diagnostické ZS
    - Vycházíme z cílů, chceme je odvodit a pokoušíme se nalézt pravidla umožňující tyto cíle potvrdit nebo vyvrátit
    - Procházíme zpětně od cílů (závěrů pravidel) k dotazům (předpokladům)
    - Využití dedukce
    - Pravidla se vyhodnocují v pořadí tak, jak jsou uložena v bázi znalostí nebo v pořadí stanovených priorit (míra neurčitosti)
    - Výroky se v předpokladech vyhodnocují tak jsou zapsány nebo podle cen
    - Výhody – účinné při malém množství hypotéz, diagnostika, hledá v logickém sledu
    - Nevýhody – postupuje se slepě od cíle dolů, nevýhodné při velkém množství hypotéz a málo vstupních datech
    - Vhodné pro hlubokou bázi znalostí s méně cíli, nemáme fakta a chceme vědět, co potřebujeme pro odvození cíle
    - *“Mám chuť na palačinky. ZS na základě stavu zásob řekne, zda je to možné.”*
  + **Přímé** (dopředné) **řetězení** (forward chaining), též usuzování řízené daty – generativní ZS, hry
    - vycházíme z předpokladů a odvozujeme závěry, které slouží jako předpoklady pro další pravidla (od předpokladů s cílům – cíle nejsou předem známy)
    - inference probíhá přímočaře – pokud neznáme platnost podmínky, snažíme se ji získat z jiného pravidla a rekurzivně pokračujeme dál
    - hodně záleží na pořadí vyhodnocování pravidel
    - výhody – z malého množství mnoho nových faktů, výhodné pro sběr informací a plánování
    - nevýhody – nepozná důležitost vstupních informací, může se ptát v nelogickém sledu, nevýhodné při malém množství hypotéz a velkém množství vstupních dat
    - *“Co si dáme k večeři? ZS se zeptá na co máme chuť a podle zásob doporučí recept.”*
    - Vhodné pro plochou bázi znalostí s více cíli, kdy máme fakta a chceme naše možnosti
* **Fungování ZS** – běh = smyčka 3 fází
  + Porovnání – tvorba rozhodovací množiny s aplikovatelnými pravidly
  + Rozhodnutí sporu – výběr 1 pravidla (instance) z množiny (podle priority, heuristiky, ...)
  + Úkon – provedení akcí pravé strany vybrané instance (vytváření, modifikace, rušení objektů, ...)
    - Důsledek může být přidání nebo odstranění předpokladu z množiny, přidání pravidla do báze znalostí, ...
* **Strategie řešení konfliktu**
  + DFS (prohledávání do hloubky) – preferují se pravidla používající aktuálnější data
  + BFS (prohledávání do šířky) – preferují se pravidla používající starší data
  + Strategie složitosti – preferují se speciálnější/složitější pravidla (s více podmínkami)
  + Strategie jednoduchosti – preferují se jednodušší pravidla
* **Analogie** – při případovém usuzování – vyhledávání podobných případů
  + Znalosti = soubor dříve vyřešených případů
  + Snazší vývoj ZS – netřeba zodpovídat experta, ale třeba definovat vhodnou metriku
  + Odvozování založeno na srovnávání vzdáleností mezi případy
* **Generování a testování** – pro generativní ZS
  + Znalosti jsou reprezentovány pravidly (IF situace THEN akce)
  + Opakovaně generujeme možná řešení a testujeme, jestli vyhovují všem požadavkům
  + Mohou být splněny podmínky více pravidel
  + Nasycení předpokladů – existuje objekt vyhovující podmínkám pravidla
  + Instance – dvojice tvořená pravidlem a jeho nasycením
* Inference v diagnostických ZS – prohledávání báze pravidel, aplikace pravidel
  + Příp. práce s neurčitostí
* Vyhledávání – exhaustivní (užití všeho) x neexhaustivní (spokojení se s prvním)
  + To samé aplikace pravidel
* **Vyvozování závěrů ZS**
  + Vysvětlování – třeba zdůvodnit rozhodnutí
    - Vysvětlovací schopnost ZS umožňuje lépe ladit BZ
    - Možnosti vysvětlovacího modulu:
      * Why – cesta v síti pravidel
      * How – aktivovaná pravidla
      * What-if
* Implementační jazyky – Prolog, LISP

**Neurčitost ve znalostních systémech, její vyjadřování a zpracování**

BI-ZNS

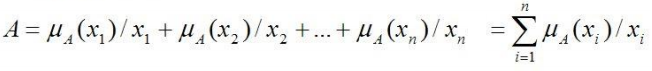
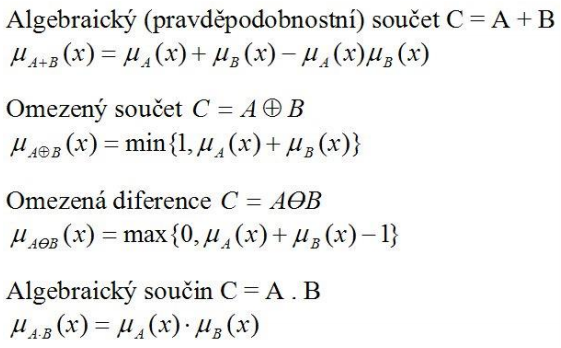
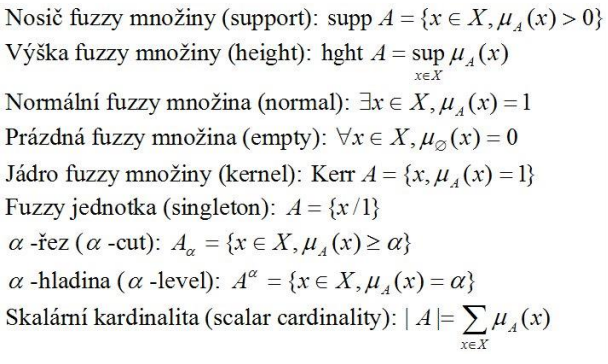
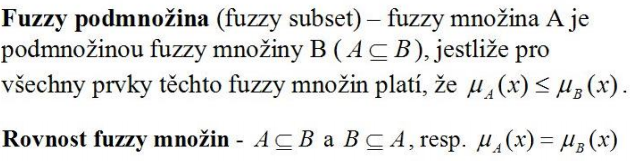
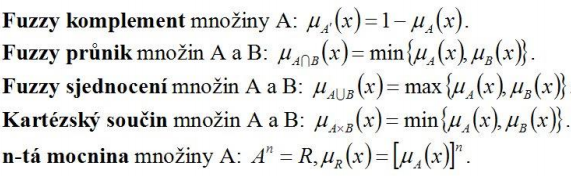
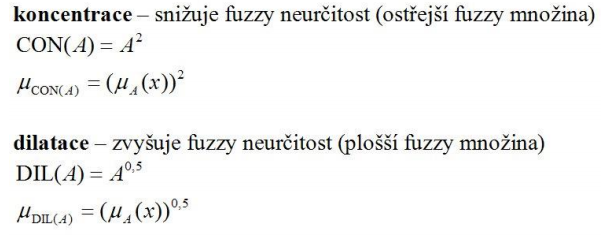
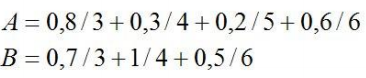
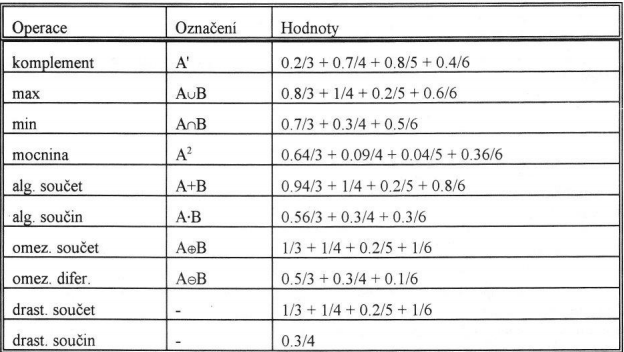
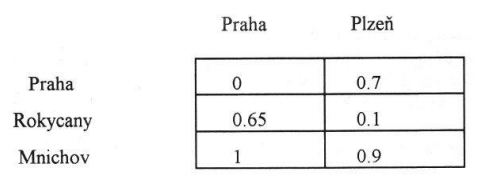
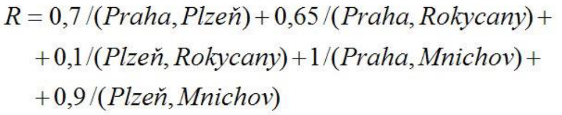
* **Neurčitost v ZS** – poznatky, které získáváme ze složitých systémů jsou neurčité a vágní nebo jsou nepřesně vyjádřeny
  + Z neurčitých znalostí a odpovědí v průběhu konzultace se odvozují neurčité závěry
* Příčiny neurčitosti
  + Problémy s daty – chybějící, nedůvěryhodné, nepřesné, nejisté
  + Nejisté znalosti – neplatí univerzálně, můžou obsahovat vágní pojmy
* **Vyjádření neurčitosti** – většinou numerická hodnota – parametr (jedno číslo, dvojice), liší se podle ZS
  + Váhy, pravděpodobnosti, stupně důvěry, faktory jistoty, ... – většinou intervaly (0, 1) nebo (-1,1)
  + Numerické hodnoty neurčitosti se přiřazují jednotlivým tvrzením nebo pravidlům
* Přístupy ke zpracování neurčitosti
  + **Ad hoc** – faktory jistoty, pseudobayesovské přístupy
  + **Teoretické principy** – teorie pravděpodobnosti, fuzzy množiny, fuzzy míra
* Problémy při zpracování neurčitosti – jak kombinovat neurčitá data, neurčitost předpokladu s neurčitostí celkového pravidla, stanovení neurčitosti závěru
* Vyjádření a zpracování neurčitosti
  + **Trojhodnotová logika** – true (1) – false (0) – unknown (X)
    - Dodefinování X do pravdivostní hodnoty logických spojek
  + **Váhy** – algebraická teorie – předpokládáme, že báze pravidel je tvořena pravidly ve tvaru A B(w)
    - A – předpoklad pravidla – kombinace (konjunkce) výroků nebo jejich negací
    - B – závěr pravidla – tvořen jedním výrokem
    - W – váha pravidla – číslo z intervalu (-1, 1) - -1 = určitě ne, 0 = nevím, 1 = určitě ano
  + **Bayesovský přístup** – pravidla ve tvaru EH, které říká, že předpoklad E podporuje závěr H
    - Neurčitost závěru H v závislosti na předpokladu E může být vyjádřena pomocí podmíněné pravděpodobnosti P(H|E)
    - Apriorní pravděpodobnostní šance:
    - Aposteriorní pravděpodobnostní šance:
    - Pravděpodobnost lze ze šance vypočítat podle vztahu
    - **Míra postačitelnosti** L – z Bayesova vzorce pro aposteriorní pravděpodobnost:
      * Velká hodnota L >> 1 = předpoklad E je postačující pro dokázání závěru H
      * Míru postačitelnosti L zadává expert
    - **Míra nezbytnosti** :
      * Malá hodnota << 1 = předpoklad E je nezbytný pro dokázání závěru H
      * Opět zadává expert
    - Váhy pravidel – pravidlo EH se chápe jako *“if E then H with weight L else H with weight ”,* tedy dvojice pravidel
      * Místo uvedených měr mohou být expertem zadány pravděpodobnosti , z nichž se tyto míry vypočtou
    - Kombinace více pravidel
      * Aposteriorní šance při nezávislosti předpokladů:
    - Kombinace více předpokladů – pomocí vztahů používaných ve fuzzy logice:
    - Výhody a nevýhody bayesovských přístupů
      * Výhody – dobře podložené teoretické základy, dobře definovaná sémantika rozhodování
      * Nevýhody – potřeba stanovení velkého množství různých pravděpodobností, riziko neúplnosti dat, předpoklady by měly b
      * ýt nezávislé
  + **Přístup založený na faktorech jistoty** – pravidla ve tvaru s faktorem jistoty CF
    - faktor jistoty z intervalu (-1,1) a je určen spojením míry důvěry (MB) a nedůvěry (MD)
    - Vyjadřuje stupeň důvěry v hypotézu H, jestliže je předpoklad E pravdivý
      * CF > 0: E zvyšuje důvěru H (+1 = maximální důvěra)
      * CF < 0: E snižuje důvěru H (-1 = maximální nedůvěra)
    - **Míra důvěry** (Measure of Belief) – interval (0, 1):
      * Přírůstek pravděpodobnosti (důvěry) hypotézy H získané (podporované) evidencí E
    - **Míra nedůvěry** (MD) – interval (0, 1):
      * Pokles pravděpodobnosti (důvěry) hypotézy H získané (podporované) evidencí E
    - předpoklad E nemusí být znám s absolutní jistotou (např. je odvozením z jiného pravidla s vlastním faktorem jistoty), výsledný faktor jistoty se pak vypočte
    - Kombinace více pravidel:



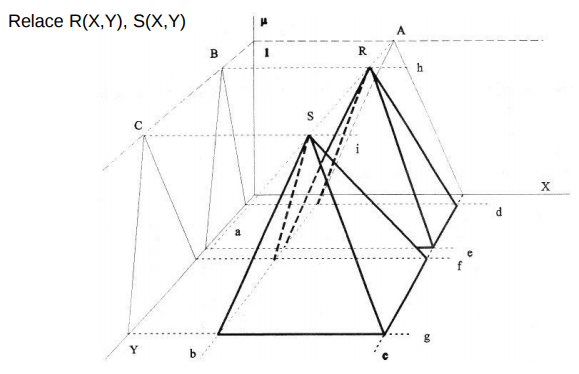
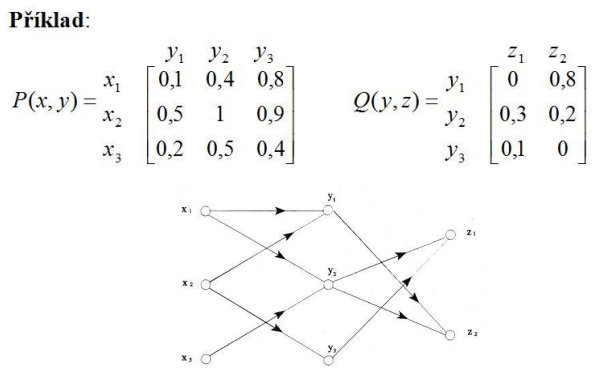
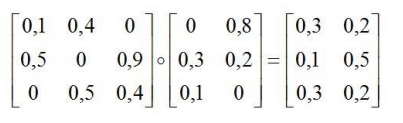
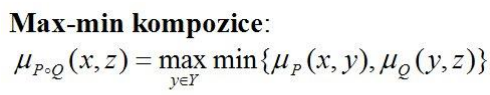
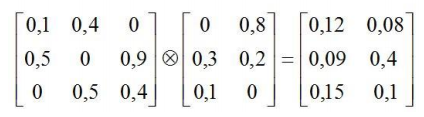
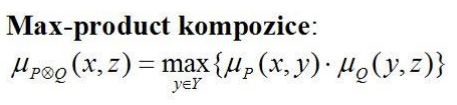
* + - *výhody*: jednoduchý a účinný výpočetní model, sběr potřebných dat je snažší než u ostatních metod
    - *nevýhody*: chybí teoretické základy, předpoklad nezávislostí Ei
  + **Nemonotónní usuzování** – neopírá se vyjádření neurčitosti jako číselné hodnoty
    - způsob inference, kdy je dříve učiněný závěr zpochybněn ve světle nové informace (předcházející znalost může přestat platit, když se dozvíme nové informace)
    - např. *„Každý pták létá“* → zpochybnění *„tučňák nelétá“* - přidání dodatečné formule (znalosti)

**Základy fuzzy logiky, neurčitost, relace**

BI-ZNS

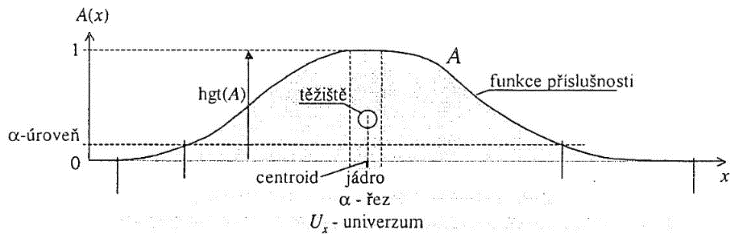
* **Fuzzy logika** – podmnožina klasické logiky (stejné značení) odvození od teorie fuzzy množin 
  + Věci v reálném světě nemají pouze binární jako u normální logiky (nepatří/patří)
  + Zavádí se **funkce příslušnosti** – každému prvku přidělí číslo z intervalu (0,1):
  + **Fuzzy množina** – psána ve tvaru:
  + **spočetná fuzzy množina** – výčet prvků s přísl. hod. funkce příslušnosti (nejde o součet, ale sjednocení)
  + můžeme znázornit pomocí zobecněných Vennových diagramů.
  + *příklad*: rozdělení lidí na malé a velké. Kde je ta hranice? Výčet malých můžeme zapsat jako A = 1/150 + 0.9/160 + 0.5/170 + 0.1/180 (fce příslušnosti/výška)
* **Základní pojmy fuzzy množin**
* **Operace modifikující funkce příslušnosti**
* **Multikriteriální rozhodování** používá operace, které jsou kombinacemi různých operací (např. vztahem pro funkci příslušnosti
  + parametr *y* z intervalu (0,1) řídí, jakou měrou se výsledná operace bude průnikem (minimum, *y* = 1) nebo sjednocením (maximem, *y* = 0)
* **Příklad:**
* **Fuzzy relace –** funkce příslušnosti pro dvojice prvků
  + relace vznikne jakou kartézský součin jednotlivých množin (tzn. minimum z hodnot funkcí příslušnosti)
  + dá se zobrazit *sagitálním grafem*: prvky (vrcholy) jsou spojeny ohodnocenými hranami (relace)
  + *Příklad*: dvojice měst, které jsou “velmi daleko”
    - Binární relace “velmi daleko” můžeme zapsat jako fuzzy množinu:

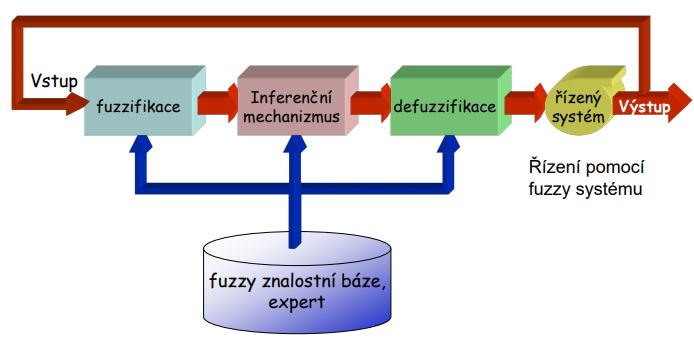
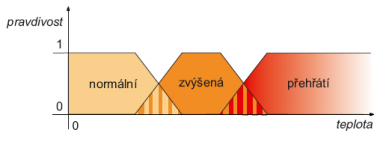
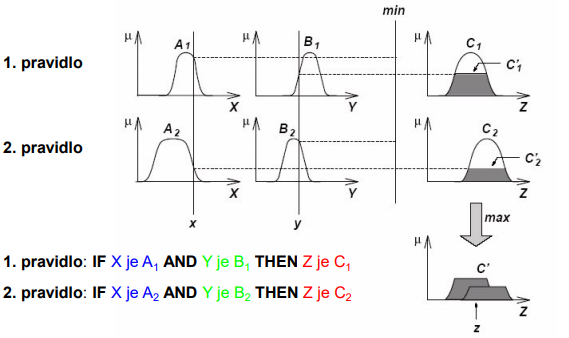
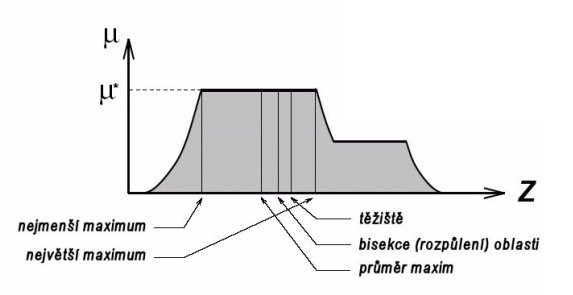


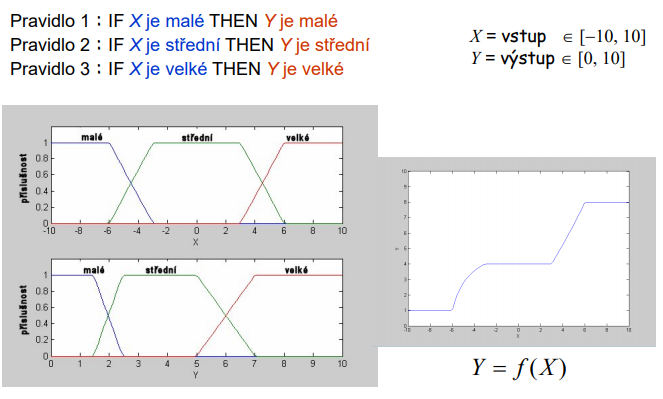
* **Výška fuzzy relace** – max. hodnota funkce příslušností dané relace
* **Projekce relace** – projekcí relace je fuzzy množina, značení:
* **Cylindrické rozšíření** – množina všech relací, které mají stejnou projekci, značení:
  + Ce(B) je hranol o podstavě tvořené funkcí příslušnosti množiny B s povrchovými přímkami d, e, h (tzn. udělá se kvádr místo čtyřstěnu)
    - Projekcí relace R(X,Y) do Y je množina B
    - Projekcí relace S(X,Y) do Y je množina C
    - Projekcí množin A a B do X je množina A
* **Kompozice relací** je podmnožina kartézského součinu taková, že , tehdy a jen tehdy, jestliže existuje alespoň jedno takové, že a
  + Základní typy kompozic:
    - **Max-min kompozice** – vezmou se složky z matic jako u násobení a z každé dvojice se vybere to nižší číslo, poté z těchto vybraných vezmu to největší a umístím na odpovídající místo ve výsledné matice
    - **Max-product kompozice** – jako násobení matic, ale místo násobení a sčítání se prvky nejdříve vynásobí a potom se vybere ta maximum ze součinů

**Reprezentace znalostí a inference pomocí fuzzy systémů**

BI-ZNS



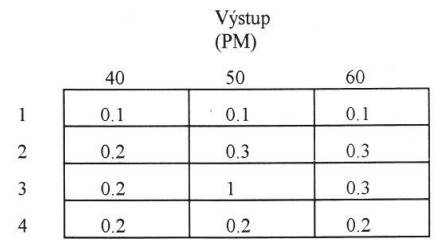
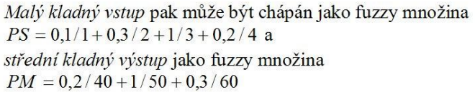
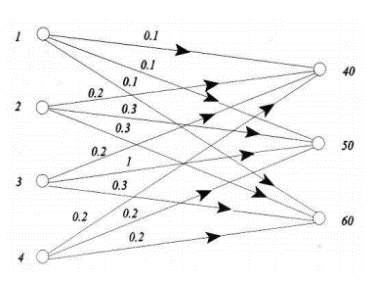
* **Funkce příslušnosti** – určuje, jak moc hodnota *x* patří do fuzzy množiny A
  + může mít různé tvary (gaussova křivka, trojúhelník, lichoběžník…)
* Základní pojmy:
  + vstupní/výstupní jazyková proměnná (nemusí být pouze číselné hodnoty)
  + vstupní logické fuzzy proměnné (termy)
  + ostrý (crisp) vstup/výstup
  + fuzzy hodnota – číslo, pravdivostní hodnota proměnné
* **Operace ve fuzzy logice**
  + negace: NOT (A) = 1 – A
  + fuzzy-logický součin: různé implementace, např. A AND B = min(A,B)
  + fuzzy-logický součet: různé implementace, např. A OR B = max(A,B)
* **Struktura fuzzy systému**
* **Fuzzifikace –** proces, který převádí číselné vstupní hodnoty jazykové proměnné na pravdivostní hodnoty souboru vstupních termů
  + např. teplota 37 °C ⇒ normální 0.0, zvýšená 0.3, přehřátí 0.7
* **Inferenční mechanismus** (vyvozovací mechanismus)
  + Soubor pravidel ve tvaru **„IF podmínka THEN důsledek“**
  + transformuje pravdivostní hodnoty vstupních proměnných (termů) na pravdivostní hodnoty výstupních proměnných
  + složení výsledku z dílčích závěrů – několik možných způsobů:
    - **Mamdaniho model** – pravidla tvaru „**IF** x **je** malé **AND** y **je** střední **THEN** z **je** velké“
      * **na levé straně podmínka vyjádřená jako fuzzy logická relace mezi stupni příslušnosti proměnných k fuzzy množinám
      * na pravé straně je podmínka vyjádřená jako stupeň příslušnosti proměnné k fuzzy množině
    - **Takagiho-Sugenův** model – pravidla tvaru „**IF** x **je** malé **AND** y **je** střední **THEN** z=f(x,y)“
      * Levá strana stejná jako u Mamdaniho
      * na pravé straně je podmínka vyjádřená jako **funkce proměnných**
      * Nejčastěji se volí funkce *f* jako polynom (lineární funkce)
        + *z = f(x,y) = a.x+b.y* (*a, b* koeficienty, které známe předem)
* **Defuzzifikace –** proces, kterým se převádí fuzzy výstup inferenčního mechanismu (pravdivostní hodnoty výstupních proměnných) na ostrou hodnotu s pomocí funkcí příslušnosti
  + Metody:
    - nejmenší maximum
    - největší maximum
    - těžiště
    - bisekce oblasti (stejně velké plochy nalevo i napravo)
    - průměr maxim
  + může tam být obrovský rozdíl, třeba zkoušet co funguje (lepší je těžiště/bisekce – nejsou to samé!)
* Pro nastavení modelů fuzzy systému se často používá neuronová síť.
* **Použití fuzzy systému** – termostaty, pračky, pohyb robota…
* Nevýhoda – potřeba odborníka na seřízení systému, ladění řady parametrů
* **Příklad**

****

* **Postup:**
  + **Fuzzifikace**– Uděláme si pro každý objekt grafy (viz graf na straně 1) a přijdou nám nějaké číselné vstupy (37 °C). Mrknem se do těch grafů a řekneme, že s hodnotou, která přišla, to patří z 0.3 do funkce příslušnosti zvýšená a z 0.7 do přehřátí
  + **Inferenční mechanismus** – aplikujeme všechna pravidla ve tvaru IF podmínka THEN důsledek (např. *IF teplota je přehřátí AND sucho je velké THEN šance požáru je vysoká)* a vyhodí nám to pro každou podmínku nějaké číslo. Pokud 0, výsledek zahodíme, jinak vezmeme fuzzy graf pro objekt, kterému chceme predikovat stav (tady *šance požáru*) a podle toho, jaké číslo nám vyšlo, tak s takovým množstvím zahrneme závěr pravidla (zde *vysoká*) do celkového závěru (takže např. ve finále bude šance požáru vysoká 0.7, střední 0.3)
  + **Defuzzifikace** – vezmeme tenhle závěrečný graf a spočítáme (třeba pomocí těžiště všech funkcí příslušnosti) výslednou hodnotu (např. vyjde 70, takže šance požáru je ve výsledku 70 %)

**Reprezentace znalostí a inference pomocí neuronových sítí**

BI-ZNS + BI-VZD

* **Neuronové sítě** – inspirovány neurony živých organismů
  + Schopnost extrakce a reprezentace závislostí na nezřejmých datech
  + Schopnost řešit nelineární úlohy, učení se a zevšeobecňování
  + Využití – klasifikace, regrese, predikce časových řad
  + Distribuovaný výpočetní systém sestávající se z dílčích podsystémů – neuronů – inspirovaných nervovým systémem organismů, který jejich činnosti více či méně realizuje
  + Více v otázce 15
* **Fuzzy-logická NS** – chování systému lze napsat jako fuzzy relaci vyjádřenou tabulkou (také pomocí grafů – sagitální graf)
  + do vstupních proměnných je zakódován vstup i se všemi fuzzy hodnotami
  + neuronová síť se ze své podstaty po naučení chová přesně jako znalostní systém (snaží se co nejvíc přiblížit systému, které známe z dřívějška)
  1. Vrstva – převod hodnoty na míry příslušející do dané fuzzy množiny
  2. Vrstva – stupeň aktivace pravidla
  3. Vrstva – výpočet míry důležitosti pravidla k podílu na vstupu
  4. Vrstva – výpočet dílčího výstupu daného pravidla
  5. Vrstva – defuzzyfikace (výpočet konkrétní hodnoty výstupu)
* Navržený neuro-fuzzy systém obsahuje řadu parametrů (odpovídají vahám na spojích), které lze modifikovat – učit
  + Učení – nejčastěji backpropagation – algoritmus zpětného šíření chyby
* komplexní architektura NS umožňuje namapovat celou řadu algoritmů a následně optimalizovat váhy pomocí optimalizační metody (deep learning, fuzzy systémy…)
* Typy NS – vícevrstvá perceptronová síť (MLP), síť RBF, Kohonenovy samoorganizující se mapy
* Základní úlohy NS – klasifikace a refrese (aproximace) s učitelem/bez učitele
* **Extrakce pravidel z NS**
  + **TREPAN algoritmus** – „pedagogický algoritmus“ – back box, známe jen vstupy a odp. Výstupy
    - na bázi konstrukce rozhodovacího stromu s pomocí dotazů a již získaných odpovědí z naučené NS – jen generátor odpovědí na náhodné vstupy
    - vlastní konstrukce stromu – modifikovaný algoritmus ID2-of-3 – expanduje uzly s největším potenciálem zvýšit důvěryhodnost RS ve srovnání s původní NS
    - pro štěpení uzlu pravidla ve tvaru m-of-n, m je prahová hodnota kolik z n podmínek musí být splněno, kritéria zastavení = dostatečná čistota uzlu, počet vnitřních uzlů
    - nahrazuje NS, tedy méně přesný, s NS pracuje jen jako s black boxem
  + **CRED algoritmus** – „dekompoziční algoritmus“ – známe strukturu sítě, kterou musíme rozložit na části
    - Generuje po vrstvách – my generujeme ravidla mezi vstupem a 1. Skrytou vrstvou a druhou sadu mezi 1. S.v. a výstupem = dvouvrstvá síť

1. Převod spojitých výstup. prom. na diskrétní třídy – využití rozhodovacích stromů

2./3. Tvorba pravidel mezi skrytou vrstvou a výstupem/vstupem – tvorba dvou stromů

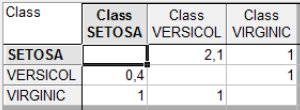
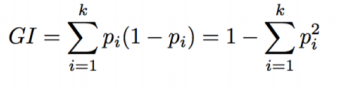
4. sestavení celkových pravidel z dílčích pravidel – substituce vstupních p. do mezilehlých p.

5. zjednodušení celkových pravidel

* + - Naučená NS se umí vypořádat se šumem – extrahovaná pravidla jím nebudou zatížena
    - Výběr vhodných atributů – zpětná vazba – zlepšení modelu
* **Báze znalostí** – trénovací data a NS na nich zkonstruované (včetně vah)
* **Inference** – rozhodování NS na základě vstupu nových dat

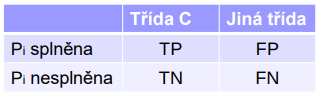
**Reprezentace znalostí a inference pomocí rozhodovacích stromů**

BI-ZNS + BI-VZD

* Postupné hierarchické rozdělování prostoru dat na podskupiny tak, aby v listech vytvářeného stromu byly (homogenní) skupiny dat náležící (v případě klasifikace) jedné třídě nebo reprezentující skupinu podobných hodnot (v případě regrese)
* **Klasifikační stromy = rozhodovací stromy**
  + Založeny na postupném rozdělování prostoru příznaků
  + Zařazení objektu do odpovídající třídy na základě diskrétních příznaků
  + Jednoduché, rychlé, snadná vizualizace
  + Nekladou podmínky na typ pravděpodobnostního rozdělení sledovaných veličin
  + Odolné vůči odlehlým a chybějícím hodnotám
* **Regresní stromy**
  + Neodhaduje se diskrétní veličina (kategorický výstup, jejíž rozdělení se uvnitř identifikovaných skupin – uzlů znázorňuje pomocí histogramu), ale spojitá veličina
    - Prokládá se hodnotami uvnitř uzlu normálním rozdělením – průměr a rozptyl
    - Průměr hodnot závislé veličiny
    - Výsledek je schodovitá predikce
* **C&RT stromy** – binární klasifikační a regresní stromy
  + Diskrétní veličiny přirozeně určené dělící body
  + Spojité veličiny diskretizace
  + **CART stromy** – tvorba hladovým způsobem
    - Tvorba: v každém kroku se úplným prohledáváním najde ta proměnná a takový dělící bod, které přinesou okamžité (aktuálně) nejvyšší zlepšení klasifikace nebo regrese
    - Tvořeny tak, aby co nejlépe odpovídaly trénovacím datům
  + Regresní stromy – minimalizuje se reziduální součet čtverců
  + Klasifikační stromy – minimalizuje se % chybné klasifikace, Gini index nebo vzájemná entropie
  + Učíme strom na TROCHU jiných datech úplně jiná struktura
  + **GC&RT** – zobecněné klasifikační a regresní stromy – klasické C&RT obohacené o podpůrné metody jako prořezávání, křížovou validaci atd.
* **Klasifikace –** přesnost ovlivňuje:
  + **Cenová/ztrátová matice** – sloupce skutečné hodnoty, řádky predikované třídy tabulka s penalizačními hodnotami, když vzor zařadím do špatné třídy
  + **Apriorní pravděpodobnosti** zastoupení jednotlivých tříd – vyjadřují pravděpodobnost s jakou vzor patří do dané třídy
    - Odhadnuté – na základě poměrného zastoupení tříd
    - Stejné – stejná pst zařazení vzoru do libovolné třídy
    - Definované uživatelem – vlastní pravděpodobnosti
    - Přímá vazba s cenovou maticí
  + **Poměrné zastoupení vzorů** jednotlivých tříd v datech
    - Gini index „nečistoty“ uzlu
    - pi = pst zastoupení kategorie uzlu
    - 0 když jsou v uzlu vzory jedné kategorie, maximální, když jsou rovnoměrně zastoupeny všechny
* **Zastavovací pravidla růstu stromu**
  + **Prořezávání na základě klasifikační chyby** – strom je prořezán tak, aby byla zachována stejná chyba klasifikace do všech tříd, používá nákladovou matici
  + **Prořezávání na základě odchylky:** používá pro prořezávání stromu rozdíl (odchylku) mezi pravděpodobností správné klasifikace nejlepšího modelu a aktuálně prořezávaného modelu
  + **Přímé zastavení růstu stromu –** zastavení růst stromu na základě dosažení maximálního předepsaného zastoupení chybných vzorů v uzlu, nepoužívá se zpětné prořezávání.
  + **Min. počet případů –** pokud počet příp. (vzorů) v uzlu klesne pod tuto hod., nebude se uzel dále dělit
  + **Maximální počet uzlů –** dělení stromu se zastaví při dosažení této hodnoty.
  + **Maximální výška stromu –** maximální počet úrovní stromu
* **Konstrukce stromu** – náhradní proměnné
  + umožňuje vytvořit strom, který dokáže klasifikovat vzory, kterým chybí nějaké hodnoty atributů
  + pokud chybí hodnota atributu, algoritmus využije náhradní atribut, který dělí vzory podobně jako původní (pokud ani ten nenajde, hledá dokud není vyčerpán seznam náhradníků)
* **Random forest** –kolekce libovolného počtu jednoduchých stromů, kde každý je modelem klasifikační/regresní úlohy a jedním výstupem, na kterém se podílí hlasováním (klasifikace), nebo průměrováním (regrese)
* **Boosted trees**
  + založené na boostingu (generování více modelů a jejich složení na základě tak, aby vznikl z vnějšího pohledu jeden výsledný model)
  + generujeme jednoduché stromy a každý následující strom je motivován, aby byl trénován na neúspěšných vzorech stromu předcházejícího
  + *inicializace*: každému vzoru (bodu) přiřadíme stejnou váhu a ta určuje šanci s jakou bude vzor vybrán (většinou rovnoměrně)
  + vybereme množinu (podle vah) a na ní se natrénuje jednoduchý strom, ten vyzkoušíme na trénovacích/testovacích datech
  + podle úspěšnosti upravíme váhy těm bodům, které tento strom klasifikoval špatně (ta určuje pravděpodobnost vybrání bodu v další iteraci)
  + v každém kroku dostáváme strom, který má velkou šanci být lepší než ten předchozí, výsledný model je složen ze stromů každé iterace (podle jejich vah)
* Přeučení – overfitting – přehnaný důraz na přesnost naučení chyb a šumu
* ***Báze znalostí***: trénovací data, zkonstruované stromy a hodnoty v jejich listech
* ***Inference***: rozhodování a odpověď jednotlivých stromů na základě vstupu
* **Extrakce pravidel z rozhodovacích stromů –** už na první pohled představuje RS soubor viditelných pravidel, ale existují sofistikované postupy na lepší extrakci sady pravidel
  + *důvody*: vyhovění struktuře báze pravidel, inferenčnímu mechanismu a vysvětlovacímu modulu, zjednodušení a zlepšení přesnosti (potlačení přeučení)
  + *počáteční stav*: existující rozhodovací strom a trénovací množina
  + *koncový stav*: sada optim. pravidel ve formátu *(IF složená podmínka THEN třída – neurčitost)*

1. *vytvoření primitivních pravidel* (výchozích stavů pro optimalizaci) – procházení stromu od kořene k listům

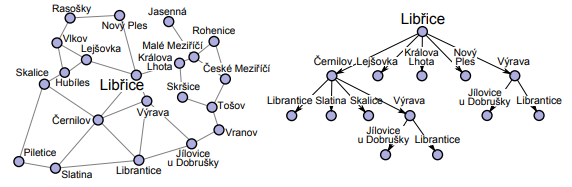
Pi … jednotlivé podmínky mezilehlých uzlů, C… třída představovaná listem

1. *optimalizace primitivního pravidla* – zjednodušení podmínek, sloučení podmínek a mezilehlých uzlů, prořezání stromu (ještě před vygenerováním pravidel)
2. *analýza důležitosti* – tabulka podmínky Pi a třídy C, vytvoření kontingenční tabulky a vytvoření faktoru jistoty
3. *odstranění podmínky Pi* – podmínka Pi je z pravidla vypuštěna, pokud její odstranění nesníží faktor jistoty, nebo hypotéza, že podmínka Pi je zbytečná, nemůže být zamítnuta na dané hladině významnosti (např. 1 %)
4. *přepočítání kontingenčních tabulek*

* **Heuristický prosévací algoritmus –** odstraňuje pravidla, které přispívají klasifikaci
  + počítá se výhodnost pravidla *r*: výhodnost = #vzorů(R) - #vzorů(R - {r}), odstraní se ty se zápornou a nulovou výhodností výstup: lokálně optimalizovaná redukovaná množina pravidel
* **Kombinování více souborů pravidel –** nejjednodušší způsob, jak sloučit více stromů, které řeší stejnou úlohu je převést na soubory pravidel a postupně slučovat tyto soubory

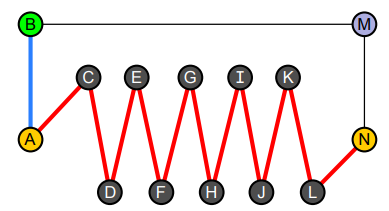
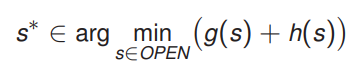
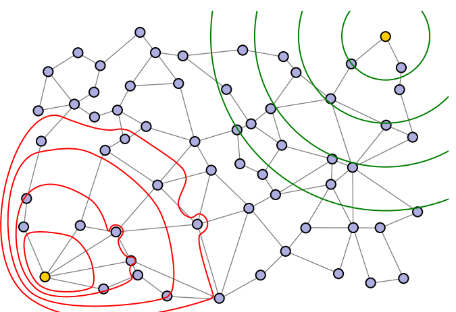
**Stavový prostor a neinformované prohledávání, stromová expanze, náhodné prohledávání, prohledávání do hloubky a do šířky**

BI-ZUM

* **Stavový prostor** = orientovaný graf, který lze prohledávat
  + Uzly = stavy S (popis stavu řešeného problému)
  + Hrany = akce A (umožňují přechod mezi stavy)
* **Prohledávání stavového prostoru** (S, A) – úloha je zadána počátečním stavem a množinou koncových stavů
  + Následníci stavu s – takové stavy s´, pro které platí (lze aplikovat akci ze stavu s vedoucí do stavu s´) množina následníků –
  + Cesta ve stavovém prostoru – orientovaná cesta z s1 do sn je posloupnost akcí (s1, a1, ...an-1, sn) taková, že platí ∀i ∈ {1, 2, ..., n−1}: ai = (si, si+1) ∧ si+1 ∈ Γ(si)
  + Problém popsaný stavovým prostorem se převádí na
    - Hledání cesty z počátečního stavu do koncového stavu – chci celou cestu
    - Hledání cílového stavu – cesta je mi jedno
* **Prohledávací strom** – Uvažujme stavový prostor X = (S, A), počáteční stav stavem a množinu koncových stavů . Prohledávací strom X je orientovaný kořenový strom s kořenem takový, že každá cesta v tomto stromě se vyskytuje i v grafu (S, A).
* **Prohledávání stromovou expanzí**
  + Vytvoř kořen stromu – počáteční uzel
  + Dokud žádný list není koncový stav:
    - Vyber nějaký list a expanduj
      * Připoj k listu všechny jeho následníky z (S, A)
  + Většina algoritmů zakazuje, aby se jeden uzel vyskytl ve stromě vícekrát, proto se rozlišují **3 stavy uzlů**
    - FRESH – uzel nebyl během hledání nalezen
    - OPEN – uzel byl nalezen, ale ne expandován
    - CLOSED – uzel byl nalezen a expandován
* Základní algoritmy pro neinformované prohledávání grafu
  + **Random search** – triviální algoritmus, který v každém kroku volí náhodný list pro expanzi stromu
    - v praxi nepoužitelný a může nalézt neoptimální řešení
  + **BFS** – prohledávání do šířky – expanze listů systematicky po patrech, zaručuje nejkratší cestu
    - Nároky na paměť a čas exponenciálně rostou s délkou nejkr. cesty k cíli – v praxi nepoužitelný
  + **DFS** – prohledávání do hloubky – expanduje strom systematicky, snaha o co největší zanoření
    - Typická implementace je expanze prvního nejhlubšího listu
    - Vhodný pro problémy, kde se nedá příliš bloudit, a přitom je vhodné se co nejrychleji vzdálit od počátečního stavu (např. pro N dam)
  + **Algoritmus rekonstrukce trasy** – společný pro všechny algoritmy
    - Pracuje s vytvořenou tabulkou předchůdců, kde si pro každý stav uložíme jeho předchůdce
    - Začíná rekonstruovat od koncového stavu/uzlu
  + Všechny tyto algoritmy jsou neinformované (zbytečně expandují irelevantní uzly) a neberou v potaz cenu prováděných akcí

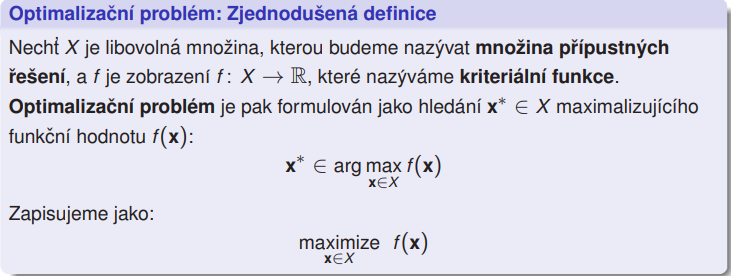
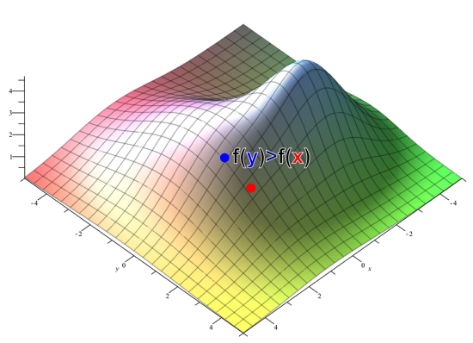
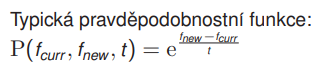
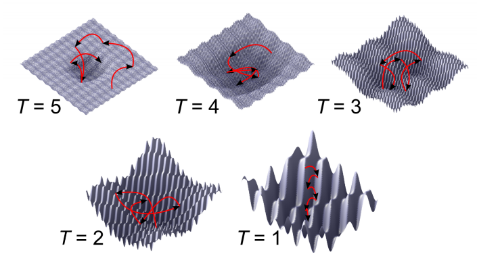
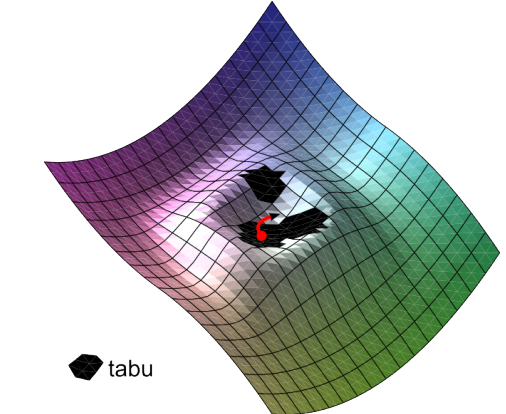
**Heuristické prohledávání stavového prostoru, heuristiky pro odhad ceny cesty, hladové prohledávání, algoritmus A\*.**

BI-ZUM

* **Stavový prostor s ohodnocenými akcemi** – Nechť (S, A) je stavový prostor, kde S je množina stavů a A je množina akcí. Trojici (S, A, c), kde c je funkce , nazýváme stavový prostor s ohodnocenými akcemi. Hodnota c(a) přiřazuje **cenu** každé akci
* **Cena cesty** = součet cen akcí nacházejících se na této cestě
* Algoritmy pro informované prohledávání pracují s expanzí stavů – přidávají do open množiny ještě neobjevené sousedy.
* **Dijkstrův algoritmus** – hledá nejlevější cestu z počátečního uzlu
  + Každý uzel si pamatuje cenu cesty k němu
  + Bez záporných cen, optimální řešení v čase O(|A| + |S|\*log(|S|))
  + při expanzi se vybírá ten list, který má nejlevnější délku cesty a pokud se při ní stane, že se expanduje uzel ze stavu OPEN, pro kterou je nalezená kratší cesta, je tento uzel přemístěn do odpovídající větve a cena je upravena **(relaxace)**
  + Jakmile se v listu objeví koncový uzel, máme zajištěno, že cesta je nejkratší možná
* **Informované prohledávání SP** – kdybychom měli odhad přibližné ceny cesty z jednotlivých open uzlů do cíle, mohli bychom uzly expandovat inteligentněji
* **Heuristika** – Nechť (S, A, c) je stavový prostor s ohodnocenými akcemi a je množina stavů pro úlohu prohledávání tohoto prostoru. Heuristika (heuristická funkce) je libovolná funkce taková, že h(s) udává odhad ceny cesty stavu k nejbližšímu a pro všechna platí .
  + **Optimální heuristika** h\* - vrací neomylnou, přímou cestu – vrací skutečnou cenu (často nedosažitelné)
  + **Přípustná heuristika** h –
    - Optimistická – nevrátí větší odhad, než je skutečná cena (jinak by např. A\* nefungoval)
  + **Monotónní heuristika** (konzistentní) -
    - Přechodem do dalšího stavu by se heuristický odhad délky zbývající cesty neměl snížit o více než o reálnou cenu této akce
  + **Dominující heuristika** – h1 dominuje h2, když
    - h\* dominuje všechny ostatní přípustné h
* **Best-first search algoritmy** – podle heuristického kritéria volí k expanzi nejslibnější uzel z OPEN
  + **Hladové prohledávání** (greedy search) – v každém kroku vybírá k expanzi stav s minimální hodnotou heuristické funkce
    - Nalezená cesta nemusí být nejkratší, pochybná rychlost (závisí na heuristice)
    - Minimalizace délky cesty zbývající k cíli
  + **A\*** – optimální Dijkstry a Greedy a. – součet funkcí
    - Expanduje uzly, které mají nejmenší součet funkcí (nejkratší cesta + nejkratší teoretická cesta – přípustná heuristika)
    - „paralelní větvě“ směřující k cíli
    - pokud je heuristika monotónní, nebo přípustná, najde optimální řešení (když není monotónní, tak pouze pokud nepoužívá množinu CLOSED)
* Minimum: min f(x) = x\_ takové, že x´ X: f(x\_) f(x´)
* Argument minima: arg min f(x) = {x\_  X | f(x\_) = min f(x)}
* Porovnávání heuristik – dominance, výpočetní náročnost

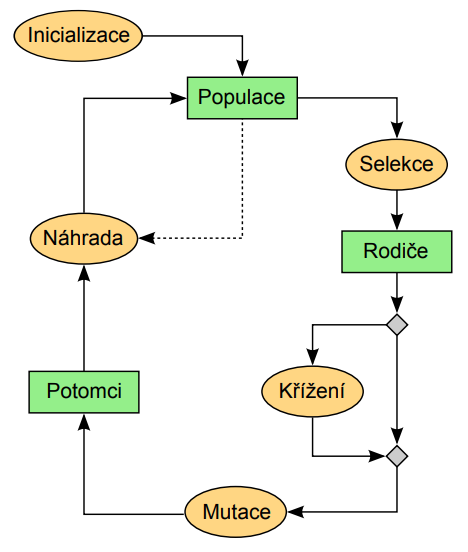
**Princip metod hill-climbing a tabu search. Simulované žíhání**

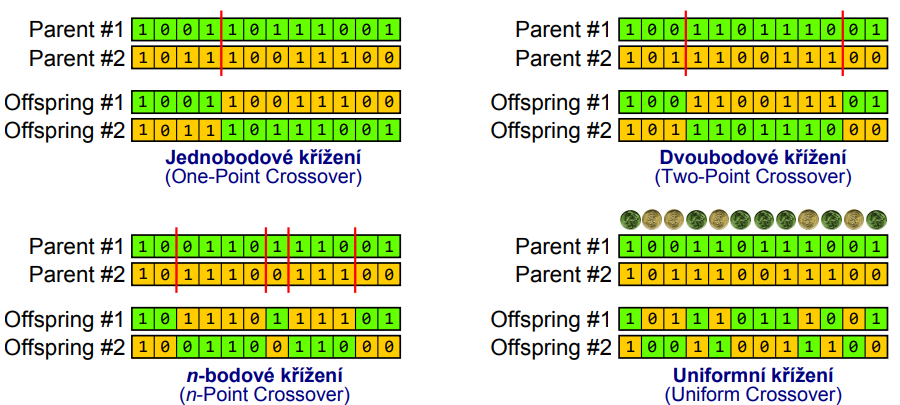
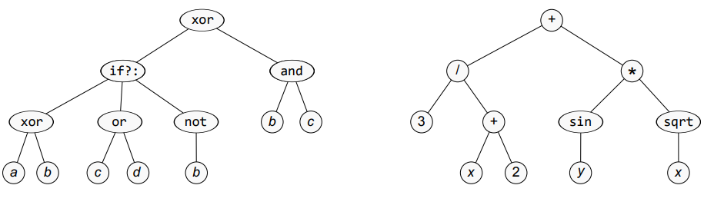
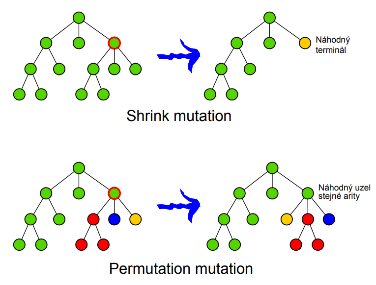
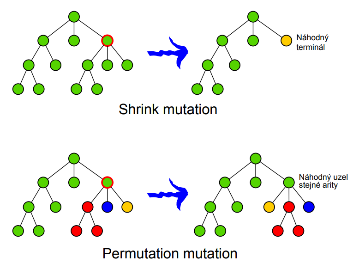
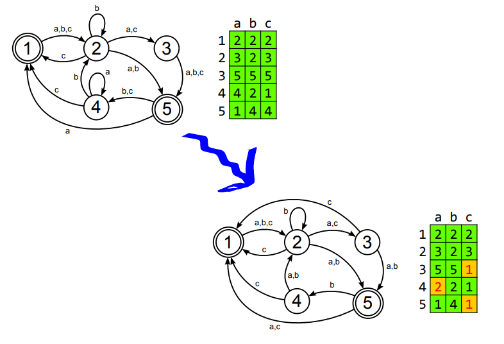
BI-ZUM

* **Algoritmy iterativní optimalizace**
  + Známá vnitřní struktura problému
  + Postupné zlepšování kandidujícího koncového stavu
  + řeší obecný optimalizační problém, znalosti o struktuře problému zjišťují až během běhu
  + metodou pokus-omyl
* **Optimalizační problém**
  + Lineární regrese (kurzy sázek), maximalizace x minimalizace (hill climbing x gradient descent)
  + Nezáleží, jestli maximalizujeme nebo minimalizujeme, výsledek je určen stejně
  + Obecný optimalizační problém obtížně řešitelný, ale existují speciální třídy, pro které jsou známy rychlé algoritmy (např. nejmenší čtverce, lineární programování, konvexní optimalizační problém)
* **Hill climbing**
  + Vrací lokální maximum, postup směrem největšího růstu (*steepest ascent*)
  + Končí, pokud dosáhl vrcholu
  + Výstup na Mt. Everest v mlze s amnézií (nezajímá ho, co leží za následujícím krokem)
  + Hladové lokální prohledávání – může vést k uváznutí v lokálním maximu, v hřebenu či plošině
  + Prokletí dimenzionality – s rostoucí dim. Roste objem prostoru exponenciálně, není možné navzorkovat okolí bodu s dostatečnou hustotou
  + **Varianty HC**:
    - Stochastic HC – výběr náhodného vyššího souseda
    - First-choice HC – náhodně generuje následníky, dokud nevygeneruje lepšího
    - Random-restart HC – náhodně generované počáteční stavy, dokud nedosáhneme cíle jako počátečního stavu
* **Simulované žíhání**
  + Rozšiřuje klasický HC o parametr *t* (teplotu) – pst, že následník bude vybrán, i když je horší (čím horší, tím menší pst)
  + Pokud je následník lepší, je vždy vybrán, jinak vybrán na základě *míry zhoršení* a *teploty*
* **Tabu search**
  + Snaha zabránit oscilaci a přinutit optimalizační algoritmus vymanit se z lokálního optima
  + **Tabu list** – popisuje části prostoru, kam se kandidující řešení nesmí vrátit
    - Většinou se vychází z podobnosti – Euklidovská, kosinová pro R, Hammingova pro binární vektory…
  + Pokud se algoritmus vyšplhal na vrchol (lokální optimum), je nucen zahájit sestup a najít další

**Evoluční výpočetní techniky**

BI-ZUM



* **Evoluční výpočetní techniky**
  + Rodina metod stochastické populační iterativní optimalizace
  + Inspirováno evoluční biologií
  + Fungují na principu šlechtění populace kandidujících řešení
  + Kvalitní řešení jsou vybrána selekcí k reprodukci, kde může probíhat křížení a mutace
  + Kompromisem mezi **exploitací** (zkoumání bezprostředního okolí) a **explorací** (náhodné procházky prostorem bránící v uváznutí v lokálním optimu)
* **Genotyp** – reprezentace řešení
* **Fitness funkce** – míra adaptace kandidujícího řešení na dané prostředí, je maximalizována
* **Jedinec** – označení pro kandidující řešení, dvojice (genotyp, fitness)
* **Populace** – množina šlechtěných jedinců
* **Generace** – čítač hlavních cyklů evolučních algoritmů (0. - inicializační)
* **Genetické operátory**:
  + **Inicializace** – vytvoření počáteční populace
  + **Selekce** – výběr z jedinců na základě populace, výsledná množina = rodiče
  + **Reprodukce** – proces tvorby potomků, probíhá křížením (vzájemná výměna informací mezi jedinci) a mutací (drobná změna genotypu jedince, důležité udržovat ji dynamickou, nejdříve velkou, pak malou, jinak předčasně konverguje), výsledná množina je označována jako potomstvo
  + **Náhrada** – náhrada všech nebo části jedinců v původní populaci potomky
* **Genetický algoritmus** – každý jedinec je řetězec nad abecedou (binární, ACGT, ...)
  + navržen jako univerzální black-box solver optimalizující binární řetězce
  + **Inicializace**
    - **Informovaná**
      * Vychýlení počáteční populace směrem ke „slibným“ oblastem stavového prostoru
        + Pomocí jednoduché heuristiky
        + Recyklace jedinců z předchozích běhů algoritmu
      * Hrozí nebezpečí, že nenávratně umístí celou populaci do lokálně optimální oblasti, z níž nepůjde vyváznout mutací ani křížením
    - **Neinformovaná** – nejčastěji používaná, univerzální
      * Vygeneruje populaci zcela náhodných binárních vektorů – pro každý bit hod mincí
  + **Selekce**
    - **Ruletová** – pravděpodobnost výběru jedince je přímo úměrná fitness
      * = pravděpodobnost výběru
      * = počet jedinců v populaci
      * = daný chromozom
      * = fitness hodnota chromozomu
      * Pravděpodobnostní rozdělení je rozdělení, ze kterého vybíráme rodiče pro křížení
    - **Turnajová** – náhodné vylosování k jedinců a výběr toho s nejlepším fitness
      * Není závislá na konkrétních hodnotách fitness
  + **Mutace**
    - Nejčastěji bit-flip mutace, kde se s pravděpodobností mutace invertují jednotlivé bity
    - Mutation rate – určuje, jak často budou mít potomci náhodné mutace
  + **Křížení**
    - Crossover point – bod, ve kterém se rodičovské řetězce rozdělí a části se zkříží, aby zformovaly potomka
    - Někdy se zcela vynechává, šlechtění pak probíhá pouze mutací
    - Připadají v úvahu i křížící operátory specializované pro daný typ problému
    - Nejčastější operátory:
  + **Elitismus**
    - V nové generaci zahrnuti někteří nejlepší rodiče z předchozí generace – garance růstu fitness funkce
    - Culling – všichni jedinci pod limitem jsou odhozeni
* **Genetické programování –** Jedinec je program
  + Genotyp – orientované kořenové stromy
  + Flexibilnější (struktura a velikost řešení jsou rovněž předmětem evoluce)
  + Listy – terminály, nezávislé proměnné
  + Vnitřní uzly – funkce (aritmetické operace, logické funkce atd.)
  + **Inicializace**
    - vygenerování náhodného stromu
    - *GROW*: větve stromu mají náhodnou hloubku, v nejnižší vrstvě jsou uzly vybrány z terminálů, v ostatních náhodně z funkcí i terminálů
    - FULL: větve mají stejnou hloubku, poslední vrstva je z terminálů, ostatní z funkcí
    - PCT1: náhodně generovaný strom se střední hodnotou počtu uzlů
  + **Mutace**
  + **Křížení –** nejčastěji se vymění 2 podstromy rodičů
* **Evoluční programování** 
  + Chování agentů bylo vyjádřeno **stavovými automaty**
    - Stavový automat popsán počátečním a cílovým stavem, množinou stavů a tabulkou přechodů
  + Používá operátory mutace, chybí operátor křížení
  + Mutaci může podléhat počáteční a cílový stav, množina stavů a tabulka přechodů
* **Evoluční strategie**
  + Genotyp: vektory reálných čísel (např. float váhy se v ANN blbě kódují jako binární řetězce)
  + **Selekce**
    - *(1 + λ)-ES*: 1 rodič vyprodukuje λ potomků, nejlepší je rodičem celé následující generace, odpovídá hill-climbingu
    - *(µ + λ)-ES*: z µ + λ potomků je vybráno µ nejlepších
    - *(µ, λ)-ES*: do další generace vybráno µ nejlepších potomků
  + **Mutace**
    - Malá změna jednotlivých složek *n*-rozměrného vektoru
    - **Gaussovská** – nejčastější, přičtení náhodného čísla z normálního rozdělení
    - **Endogenní strategické parametry** – jedinec ukládá i vektor rozptylů gaussovských rozdělení pro jednotlivé dimenze, které se mění v závislosti na úspěšnosti mutace (20 %) = forma metaevoluce
    - Míra mutace
      * Příliš velká – poškozuje užitečnou informaci v genotypech
      * Příliš malá – brání náhodnému zlepšování řešení
      * Dynamická – řízená úspěšností (ES) nebo řízená časem, např. v kombinaci s principem simulovaného žíhání (v počátečních generacích velké mutace, v pozdějších malé)
  + **Křížení** 
    - Zásadně uniformní (po složkách)
    - **Diskrétní** – daná složka vektoru se překopíruje vždy od jednoho z rodičů
    - **Aritmetické** – průměr z hodnot rodičů
    - Další parametr ρ, který určuje počet rodičů, kteří se podílí na tvorbě potomka
      * 
  + Problém **předčasné konvergence** – celá populace začne být stejná, ovládaná několika typy jedinců
    - Mutace není schopná to zlepšit, musela by být příliš velká
    - Křížení nefunguje, jedinci jsou příliš rozdílní
    - Odpovídá uváznutí v lokálním optimu
* **Niching** – metody, které berou v potaz podobnost genotypů – nepodobní jedinci se nekříží nebo nesoupeří
  + **Ostrovní model** – oddělené populace na vlastních ostrovech
    - Malá výměna informací mezi ostrovy
    - Každý ostrov sleduje vlastní evoluční trajektorii ve stavovém prostoru
  + **Fitness sharing** – podobní jedinci sdílejí fitness hodnotu
  + **Novelty search** – evoluční výhodu mají jedinci přicházející s novými využitími pro existující fenotyp
    - Inspirace biologickou fitness
    - Niching mezi fenotypy
  + **Deterministic crowding** – potomci soutěží o přežití s podobnějším z rodičů, výhra podle fitness
    - Mechanismus brání přemnožení příliš dobrého typu jedince
    - Potomek s vyšší fitness, než rodič jej nahradí, potomek s nižší fitness, než rodič přežije

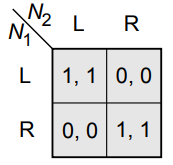
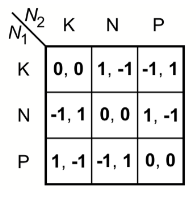
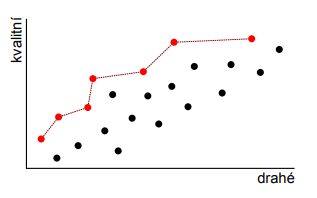
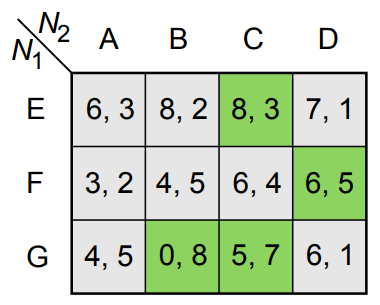
**Návrh inteligentního agenta, typy agentů, vlastnosti a požadavky na agenty. Multiagentní systémy, organizace multiagentního systému (kooperace, kompetice, koordinace, vyjednávání, komunikace).**

BI-ZUM

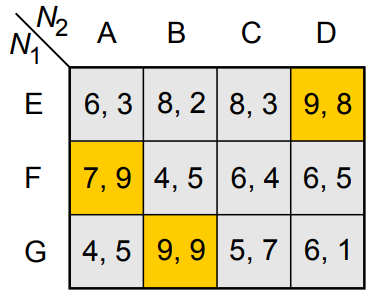
* **Agent** – cokoliv, co je schopno vnímat prostředí pomocí senzorů a jedná v něm pomocí efektorů
  + Agentní funkce (popisuje chování agenta) agentní program (fyzická implementace agentní funkce)
  + Agent = architektura + program
    - Program implementuje agentní funkci, která mapuje vjemy na akce
    - Architektura – přijímá vjemy od senzorů, posílá je programu a na základě jeho odpovědi jedná pomocí efektorů
  + Počítačový systém umístěn v prostředí, kde je schopen samostatné činnosti, aby dosáhl cíle
    - Klíčová vlastnost – autonomie, účelné činnosti
* **PEAS** – prostředí úlohy – musí být při návrhu agenta definováno
  + P – performance measure – metrika úspěšnosti
  + E – environment – prostředí
  + A – actuators – efektory
  + S – sensors – senzory
* **Vlastnosti prostředí**:
  + Plně pozorovatelné x částečně p. – jestli pořád zná celkový stav prostředí
  + Deterministické x stochastické – v det. není neurčitost
  + Epizodální x sekvenční – v ep. Je agentovo chování rozděleno na episody, které nezávisí na akcích v předešlé
  + Statické x dynamické – v dyn. Se prostředí může v průběhu měnit
  + Diskrétní x spojité – diskrétní má konečný jasně daný počet stavů
  + Single-agent x multi-agent
* **Multiagentní systém** – prostředí, ve kterém se pohybuje několik nezávislých agentů najednou
  + Distribuované řešení úloh
  + Každý z agentů:
    - Vnímá okolní prostředí
    - Jedná (provádí akce) tak, aby dosáhl svých cílů
    - Interaguje s ostatními agenty (kooperace a koordinace akcí)
  + aplikační oblasti – distribuované systémy, vysoce dyn. prostředí, nekooperativní rozvržení
* **Typy agentů v multiagentním systému**
  + ZNS
    - Reaktivní – reaguje na podněty z okolí a má předem danou množinu akcí, které může provádět, vybere tu nejvhodnější, nepoužívá paměť, nenabírá zkušenost
    - Intencionální – zvažuje své možnosti, jak dosáhnout cíle, je motivován k co nejlepším reakcím, má paměť a agenti tohoto typu se navzájem informují
    - Sociální – má k dispozici modely chování ostatních agentů a ty může využívat ke svému rozhodování
    - Hybridní – může vykazovat vlastnosti ostatních typů dle situace, univerzální
  + ZUM
    - Izolovaní – ekvivalent single-agentního systému
    - Kooperativní – jednají společně za účelem dosažení týmového cíle
    - Self-interested – maximalizuje vlastní dobro, zvažuje ostatní agenty
    - Hybridní – kombinace výše uvedených
* **Typy agentů z hlediska agentních funkcí**
  + Jednoduchý reflexní agent – aktuální vjemy, if-then
  + Reflexní agent se stavem – paměť, historie vjemů
  + Goal-based agent – akce voleny s ohledem na cíle pomocí plánování
  + Utility-based agent – zavádí utilitní funkci mapující stav prostředí na reálné číslo
* Vlastnosti a požadavky na agenty
  + **Autonomní** – pracuje bez zásahu z vnějšku, do jisté míry řídí své akce
  + **Sociální** – interaguje s ostatními agenty (kooperace, koordinace, vyjednávání)
  + **Reaktivní** – vnímá okolí a reaguje na změny
  + **Proaktivní** – sám se iniciativně pouští do řešení problémů
  + **Mobilní** – působit aspoň virtuálně na různých místech. Umístění v konkrétním místě prostředí je považováno za důležitou vlastnost.
  + **Korektní** – neposkytovat vědomě nepravdivé informace
  + **Benevolentní** – snažit se udělat to, o co je žádán
  + **Racionální** – jednat tak, aby dosáhl svých cílů
  + další důležité vlastnosti – *vysílání a přijímání informací ostatních agentů, provádění akcí, generování cílů a plánů…*
* Pro realizaci se využívá AI (machine learning, fuzzy, teorie her, genetické programování…)
* **Architektura multiagentního systému** 
  + Množina agentů + komunikační síť – fyzická struktura spolu s vlastnostmi a požadavky kladenými na jednotlivé části
  + Požadavky:
    - **Pokrytí** – je třeba zajistit, aby každé podúlože byla přidělena skupina řešících agentů
    - **Spojení –** aby agenti byli schopni si mezi sebou předávat informace (kooperace + předávání si dílčích výsledků)
    - **Výpočetní potenciál** – dostatečné množství agentů daný problém zvládnout
* Organizace multiagentního systému
  + **Kooperace** – schopnosti jednotlivých agentů vzájemně spolupracovat (sdělování dílčích výsledků, plánů, získávání informací)
  + **Kompetice** (soutěživost mezi agenty) – ostatní agenti jsou bráni za konkurenty. Snaha maximalizovat svůj vlastní zisk.
  + **Koordinace** (součinnost kooperujících agentů) – zvýšit pravděpodobnost a rychlosti dosažení stanovených cílů. Koordinace realizuje kooperaci. Prostředky: závazek, smlouva, kostra plánů a řešení úlohy
  + **Vyjednávání** *–* cílem je odstranit rozpory mezi agenty, vyvarovat se konfliktům, maximálně využít dostupné zdroje apod. Probíhá na základě vyjednávacího protokolu. Kooperující i kompetiční agenti. Lingvistické, rozhodovací a procesní vyjednávání
    - **Protokol spolupráce –** rekurzivní rozdělování práce mezi agenty
  + **Komunikace –** způsob předávání informací mezi agenty. Je prostředkem zejména pro kooperaci agentů

**Hry v normální formě. Analýza akčních profilů: Paretovo optimum, Nashovo equilibrium**

BI-ZUM

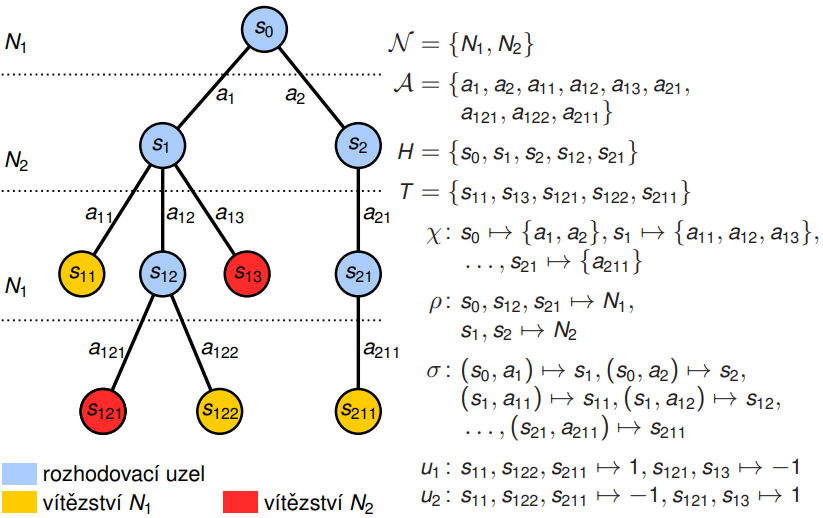
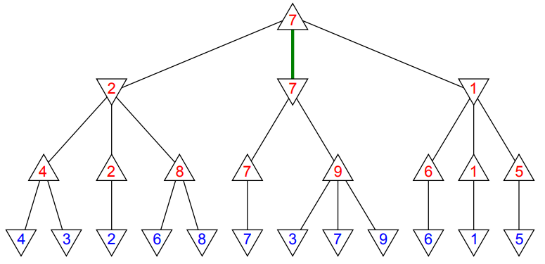
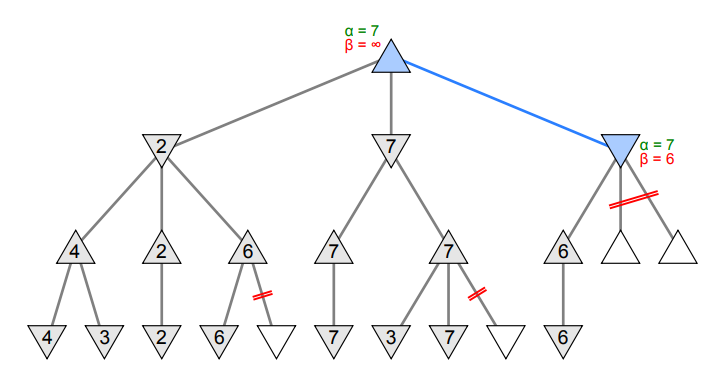
* **Teorie her –** matematická disciplína studující komplikované situace, kdy jsou utilitní funkce agentů ve vzájemné interakci
  + Kategorie
    - **Kooperativní** – modelovací jednotkou je tým hráčů
    - **Nekooperativní** – modelovací jednotkou je jeden hráč
  + Aparáty:
    - **Normální forma** – hra vyjádřená maticí
    - **Extenzivní forma** – hra vyjádřená stromem
* **Hry v normální formě**
  + **Konečná hra v NF pro n hráčů** je trojice G = (N,A,u), kde:
    - N = konečná množina hráčů
    - A = A1 x … x An, kde Ai je množina akcí, které má k dispozici hráč Ni, a z A je akční profil vyjadřující jednu konkrétní volbu akcí provedenou nezávisle **všemi hráči** ve hře
    - u = (u1...un) je posloupnost utilitních funkcí pro jednotlivé hráče
      * utilitní funkce udává výhodnost/nevýhodnost dané situace pro konkrétního hráče
  + **Common-payoff hra** – hra G, pro kterou platí, že ∀a ∈ A: u1(a) = u2(a) = ... = un(a)
    - tzn. utilitní funkce pro jeden akční profil je pro všechny hráče stejný
    - např. po které straně silnice jezdit (viz obrázek)
  + **Constant-sum hra –** G, pro kterou platí ∃c ∈ R: ∀a ∈ A: sum(ui(a)) = c
    - tzn. součet utilitních funkcí pro všechny akční profily jsou rovny konstantě *c*
    - pokud *c = 0*, pak je to *zero-sum* hra
    - např. kámen-nůžky-papír (viz obrázek)
* **Akční profily**
  + Hry popsané v normální formě poskytují pohled shora:
    - Hráči jsou si rovnocenní
    - Nenahlížíme na ně pohledem “hráč 1 má vyhrát, hráč 2 prohrát”
    - Jde o hledání akčních profilů optimálních z jistých globálních hledisek
  + **Paretovo optimum –** hledání kompromisu mezi vzájemně si odporujícími kritérii (hledání nejvíc dominujících prvků v nějakém kritériu, jako ve skyline)
    - Změnou akčního profilu již žádný hráč nemůže zvýšit svou utilitu, aniž by se snížila utilita jiných hráčů
    - Např. nákup levných, kvalitních výrobků
    - Graf: Červené jsou paretovsky optimální, protože neexistuje kvalitnější výrobek o stejné nebo nižší ceně a neexistuje levnější výrobek o stejné nebo vyšší kvalitě. Černé nejsou paretovsky optimální.
    - **Paretovská dominance** – Uvažujme hru v normální formě (N, A, u). Řekneme, že akční profil paretovsky dominuje akční profil , jestliže jsou splněny podmínky:
    - Akční profil profil je **paretovsky optimální**, jestliže neexistuje žádný akční profil profil , který jej paretovsky dominuje
  + **Nashovo equilibrium** 
    - Hry v normální formě předpokládají omezenou pozorovatelnost
    - Kdyby jeden hráč věděl, jaké akce zvolili ostatní hráči, byla by volba optimální akce triviální
    - **Best response** – Uvažujme hru v normální formě (N, A, u), akční profil , jednoho konkrétního hráče a jeho utilitní funkci ui. Označme

jako redukovaný akční profil definující akce všech hráčů kromě hráče Nj. Potom best response na a-i je množina

* + - Uvažujme hru v normální formě (N, A, u) a akční profil . Řekneme, že a je Nashovo equilibrium, jestliže
    - Nashovo equilibrium je takový akční profil, kde akce každého hráče představuje best response na akce ostatních hráčů
    - Za znalosti akcí zvolených jinými hráči jsou všichni hráči spokojeni s akcí, kterou zvolili
    - **Equilibrium** = rovnováha, stabilita, žádný hráč nechce svou strategii měnit
    - Oba hráči vybírají svoji akci tak, aby byla utilitní funkce pro jejich konkrétní akční profil co největší. Třetí sloupec nemůže mít NE, protože kdyby N2 vybral C, N1 by pro sebe vybral jako nejlepší E, to by se ale nelíbilo N2, protože pro něj by byla nejvýhodnější akce N1 G, tudíž by svůj tah chtěl změnit, což nesplňuje podmínky Nashova equilibria.

**Prohledávání herního stromu: Algoritmus Minimax, alfa-beta prořezávání a heuristiky**

BI-ZUM

* **Konečná hra v extenzivní formě pro n hráčů** – osmice
  + N = konečná množina hráčů
  + A = množina akcí
  + H = množina rozhodovacích uzlů
  + popisuje, které akce jsou v daném uzlu k dispozici
  + říká, který hráč je v daném uzlu na tahu
  + T = množina terminálních uzlů (listů stromu),
  + = prostá funkce následnictví
    - je prostá graf má tvar stromu herní strom
  + u = (u1, …, un), ui: vyjadřuje utilitu hráče Ni v jednotlivých terminálních uzlech
* **Dvouhráčová zero-sum hra** v extenzivní formě – hra v extenzivní formě , kde:
  + - V terminálu t nastává:
      * u1(t) = 1, u2(t) = −1 výhra N1
      * u1(t) = −1, u2(t) = 1 výhra N2
      * u1(t) = 0, u2(t) = 0 remíza
* **Hráči MIN a MAX –** uvažujeme zero-sum hru dvou hráčů, pak lze utilitní funkce obou hráčů nahradit jedinou funkcí *u: T R*
  + *MAX*: snaží se maximalizovat *u,* na tahu v kořeni stromu
  + *MIN*: snaží se minimalizovat *u* – „soupeř“
* **Perfektní hra –** hráč v každém tahu volí bezchybnou akci
  + **Bezchybné akce** *–* MAX zahrál akci *a\** maximalizující zaručenou minimální hodnotu *u* v terminálu podstromu při pesimistickém očekávání reakce hráče MIN (např. hráč může hru vyhrát a zvolil akci, která k výhře směřuje)
  + **Chybné akce** *–* hráč zahrál suboptimální akci vzhledem k pesimistickému odhadu reakce soupeře a jeho pozice se zhoršila (např. mohl zaručeně vyhrát, ale zvolil akci, kdy to dovolil soupeři ještě zvrátit)
  + Nezbytné využití aproximací a heuristik
  + Prohledávání stromu s omezenou hloubkou – v hloubce d ohodnotíme uzly heuristickou evaluační funkcí
* **Algoritmus Minimax** – průchodem do hloubky generuje kompletní herní strom (do hloubky d) z počátečního uzlu x0 (MAX)
  + Umožňuje zvolit optimální akci v kořenu stromu vzhledem k ohodnoceným listům
  + Předpokládá perfektní hru ze strany soupeře
  + Časová složitost O(bd), b = průměrný větvící faktor, d = hloubka stromu
  + **Algoritmus:**
    - Vygeneruj strom z kořene do hloubky *d*
    - Pokud narazíš na terminál, ohodnoť ho reálnou utilitní funkcí
    - Pro neterminály v hloubce *d* spočítej heuristickou evaluaci uzlu
    - pokud je rozhodovací uzel MAX, vyber maximální hodnotu utilitních funkcí synů
    - Pokud je rozhodovací uzel MIN, vyber minimální hodnotu utilitních funkcí synů
    - Po návratu do kořenu vidíš z jakého syna přišla největší hodnota a tím směrem se vydej (protože kořen je MAX uzel)
  + Problém – omezení hloubkou *d.* Ta je většinou malá, protože průměrný větvící faktor např. v šachách je 35, množství možných stavů roste exponenciálně. Proto je nutné Minimax optimalizovat. K tomu se nejčastěji používá *alfa-beta prořezávání*.
* **Alfa-beta prořezávání** – optimalizace Minimaxu
  + Algoritmus v každém expandovaném uzlu uchovává dvě hodnoty – α a β
  + **α** – největší utilita, kterou hráč MIN zaručeně nesníží při perfektní hře hráče MAX
  + **β** – nejmenší utilita, kterou hráč MAX zaručeně nezvýší při perfektní hře hráče MIN
  + Tyto hodnoty se aktualizují v průchodu stromem, opět nejprve od listů, poté rodičům
  + Pokud dojde k β ≤ α, tak se ostatní větve stromu proříznou a neprohledávají se další potomci, rodičům se updatujíα a β hodnoty (počátečně nastavené na -/+inf)
  + Časová složitost O(b¾\*d), optimálně až O(b½\*d) – záleží na pořadí uzlů
  + Zlepšení lze dosáhnout kvalitními heuristikami
    - **Killer heuristics** – jestliže daný tah způsobil prořez někde jinde ve stromu, prioritizuj tento tah
  + **Evaluační funkce** – minimax s a-b volí optimální akci v kořenu vzhledem k ohodnocení listů -> nutná rychlost, spolehlivost
    - Omezená hloubka – problém horizontu – co když v hloubce n + 1 dojde ke zvratu v můj prospěch, ale já přestanu v hloubce n?
    - Quiesence search – expanzi stromu končíme pouze ve stavech, po nichž bezprostředně nehrozí „divoké výměny“ figur
* Alternativy – neuronové sítě -> deep learning