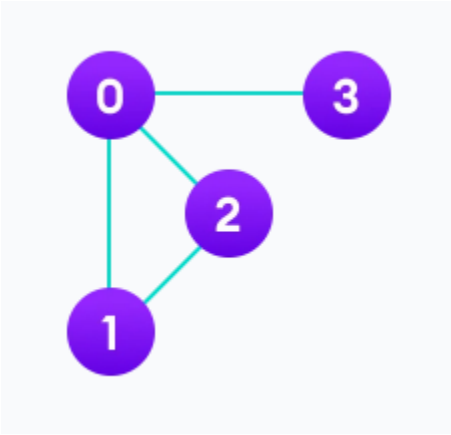
ZALG – 6. cvičení

Grafy

- Graf je nelineární datová struktura skládající se z konečného počtu vrcholů a hran. Vrcholy jsou někdy také označovány jako uzly a hrany jsou spojnice, které spojují libovolné dva uzly v grafu. Formálně je tedy graf složen z množiny vrcholů (V) a množiny hran (E).
- Graf se značí G(V, E)



Terminologie

- Sousední vrcholy – dva vrcholy jsou sousední, jestliže jsou spojeny hranou
- Sousední hrany – dvě hrany spolu sousedí, jestliže mají společný vrchol
- Stupeň vrcholu – počet vrcholů, se kterými daný vrchol sousedí
- Cesta - Je to posloupnost vrcholů, pro kterou platí, že v grafu existuje hrana z daného vrcholu do jeho následníka. Žádné dva vrcholy (a tedy ani hrany) se přitom neopakují.
- Cyklus – cesta, která začíná a končí ve stejném vrcholu.
- Ohodnocení hran - číslo přiřazené hraně, může např. představovat délku hrany, počet obsluh hrany apod.

- Prázdný graf - neobsahuje žádný vrchol, tedy ani žádnou hranu
- Nulový graf - obsahuje pouze vrcholy, neobsahuje žádnou hranu
- Triviální graf - případ nulového grafu, obsahuje pouze jeden vrchol
- K-regulární graf – všechny vrcholy mají stejný stupeň
- Úplný graf – všechny dvojice vrcholů v grafu jsou spojený, bude tam tedy $\binom{n}{2}$ hran

Orientované x neorientované grafy

- Pojmem orientovaný graf se označuje takový graf, jehož hrany jsou uspořádané dvojice. Hrany orientovaného grafu mají tedy pevně danou orientaci.
- Neorientovaný graf se v teorii grafů označuje takový graf, jehož hrany jsou dvouprvkové množiny. Hrany neorientovaného grafu nemají danou orientaci. Tudíž výrazy (x, y) a (y, x) označují stejnou hranu.

Souvislost grafu

- Graf nazveme souvislým, jestliže mezi libovolnými dvěma vrcholy *u* a *v* existuje cesta z *u* do *v* i z *v* do *u*
- Graf G[V,H] nazveme **podgrafem** grafu $G_0[V_0,H_0]$ tehdy, pokud platí $V \subseteq V_0$ a $H \subseteq H_0$.
- Komponentem grafu nazveme maximální souvislý podgraf; má-li graf jeden komponent, potom je souvislý.
- Strom je typem grafu, který je nejmenší souvislý

- Kostra souvislého grafu G je takový podgraf souvislého grafu G na množině všech jeho vrcholů, který je stromem.
- Minimální kostra grafu – kostra grafu s minimálním součtem ohodnocení hran do kostry zařazených.

Reprezentace grafů

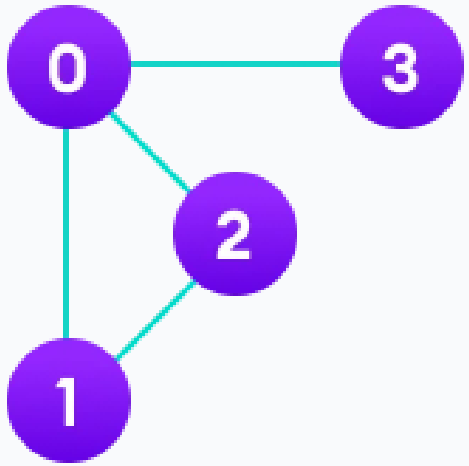
- Matice sousednosti (Adjacenční matice)
- Seznamy sousedů
- Matice incidence
- Seznamy hran

Matice sousednosti

- Matice typu nxn
- Jestliže máme neorientovaný neohodnocený graf, tak pokud mezi vrcholy i a j existuje hrana, tak v adjacenční matici bude a_{ij}=a_{ji}=1, jinak a_{ij}= 0. Matice sousednosti bude tedy symetrická. U ohodnoceného neorientovaného grafu je to stejné, jen místo 1 bude hodnota hrany

- Jestliže existuje v orientovaném grafu hrana mezi vrcholy i a j, jdoucí od i k j, tak bude a_{ij}=1, případně bude a_{ij} rovno váze hrany. V opačném případě bude a_{ij}=0. Zde už matice sousednosti není nutně symetrická

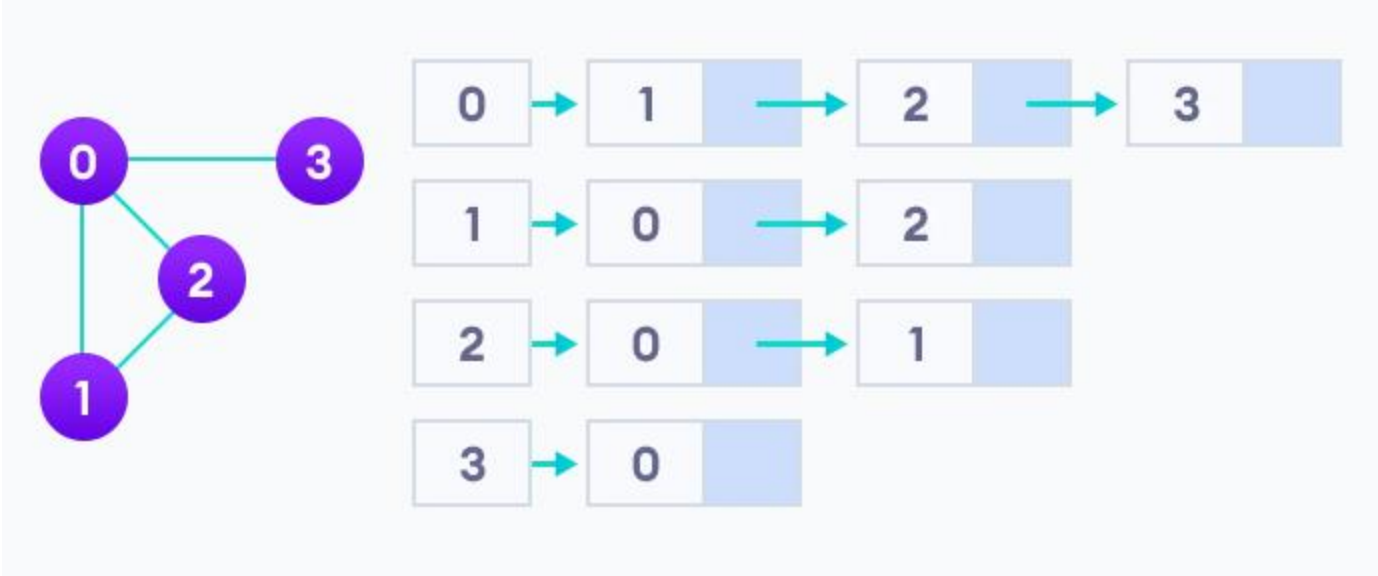
Matice sousednosti (Adjacency matrix)



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Seznamy sousedů (Adjacency list)

- Seznam sousedů reprezentuje graf jako pole spojových seznamů.
- Index pole představuje vrchol a každý prvek v jeho spojovém seznamu představuje ostatní vrcholy, které tvoří hranu s vrcholem.



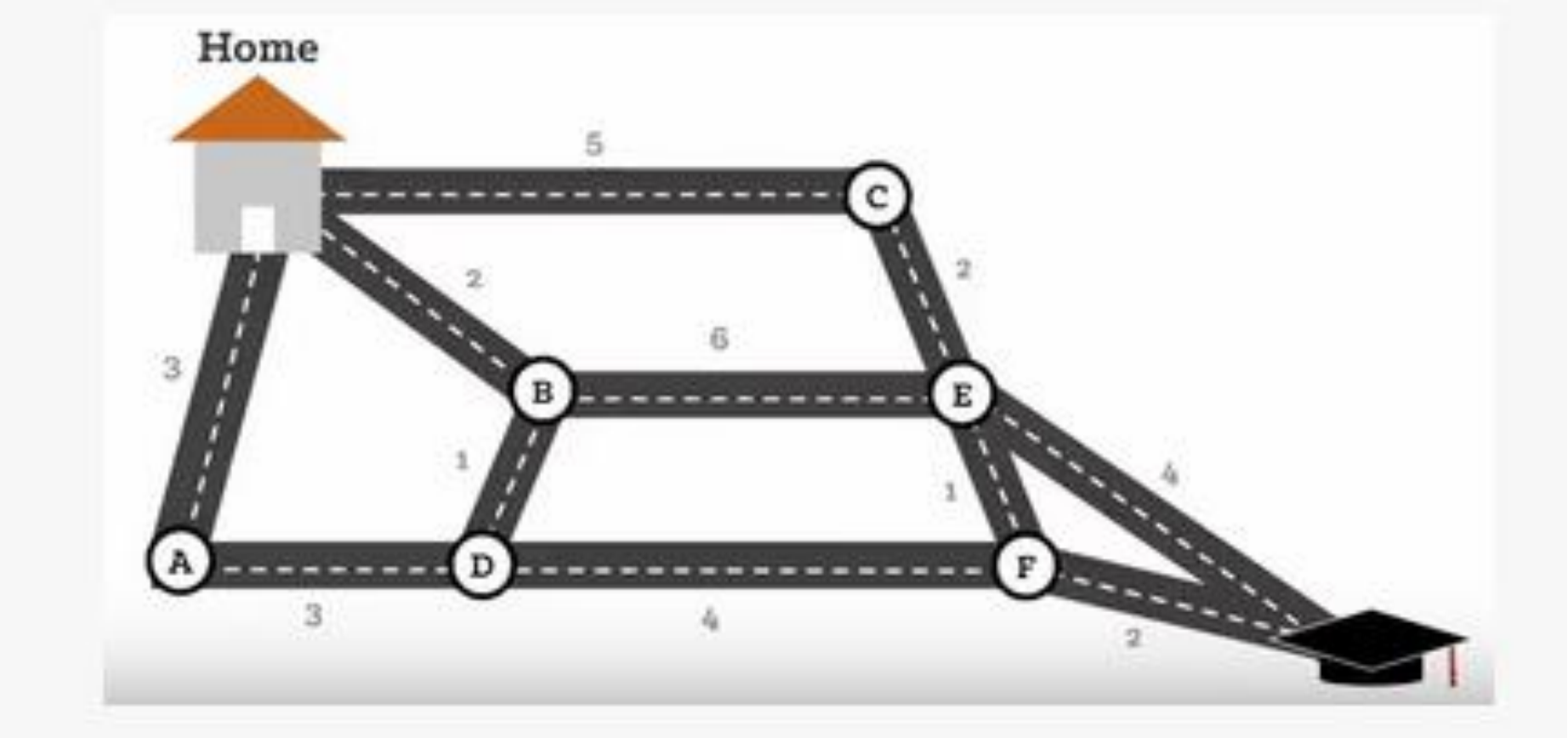
```cpp
class graph // unoriented
{
protected:
    std::vector<std::vector<double>> adjacency_graph;
public:
    graph(int n);
    graph(const std::vector<std::vector<double>>& v);
    bool is_out_of_bounds(unsigned int i, unsigned int j) const;
    void set_weight(unsigned int i, unsigned int j, double weight);
    double get_weight(int i, int j) const;
    unsigned int number_of_vertices() const;

    // Shortest path algorithms
    std::pair<std::vector<double>, std::vector<int>> dijkstra(unsigned int start_vertex) const;
    std::pair<std::vector<double>, std::vector<int>> bellman_ford(unsigned int start_vertex) const;
};
```

Hladový algoritmus vs dynamické programování

- U hladového algoritmu je rozhodování učiněno pouze na základě aktuálních informací bez ohledu na budoucí dopad. Splnění stanovených kritérií je podmnožinou, která obsahuje pouze ta nejlepší a nejpřínosnější řešení. V případě proveditelného řešení, pokud podmínku splňuje více než jedno řešení, budou tato řešení přijata jako proveditelná, přičemž nejlepší řešení bude jediné ze všech.
- Dynamické programování je programovací technika, která umožňuje programátorům řešit problémy způsobem, který bere v úvahu vliv proměnných a dalších faktorů na řešení. Základní myšlenkou je rozdělit problém na menší, lépe zvládnutelné části a poté vyřešit každý z těchto částí samostatně.

Dynamické programování	Hladový algoritmus
Používán ke hledání optimálního řešení	Používán ke hledání optimálního řešení
vybíráme v každém kroku, ale výběr může záviset na řešení dílčího problému	děláme jakoukoli volbu, která se v danou chvíli jeví jako nejlepší, a pak řešíme dílčí problémy, které se objeví po provedení volby
Méně efektivní ve srovnání s hladovým přístupem	Efektivnější
Je zaručeno, že dynamické programování vygeneruje optimální řešení pomocí Principu optimality.	Zde žádná taková záruka získání optimálního řešení neexistuje

Nejkratší cesta v grafu



Dijkstrův algoritmus

- Dijkstrův algoritmus si uchovává všechny uzly v prioritní frontě řazené dle vzdálenosti od zdroje - v první iteraci má pouze zdroj vzdálenost 0, všechny ostatní uzly nekonečno. Algoritmus v každém svém kroku vybere z fronty uzel s nejvyšší prioritou (nejnižší vzdáleností od již zpracované části) a zařadí jej mezi zpracované uzly. Poté projde všechny jeho dosud nezpracované potomky, přidá je do fronty, nejsou-li tam již obsaženi, a ověří, zda-li nejsou blíže zdroji, než byli před zařazením právě vybraného uzlu mezi zpracované. To znamená, že pro všechny potomky ověřuje:

$$vzdálenost_{zpracovávaný} + délkaHrany_{zpracovávaný,potomek} < vzdálenost_{potomek}$$

- Pokud nerovnost platí, tak danému potomkovi nastaví novou vzdálenost a označí za jeho předka zpracovávaný uzel

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

Bellman Fordův algoritmus

- Lze využívat, i když jsou v grafu záporně ohodnocené hrany
- Operace relaxace - Do této operace vstupují dva uzly a hrana, která mezi nimi vede. Pokud je vzdálenost zdrojového uzlu sečtená s délkou hrany menší než aktuální vzdálenost cílového uzlu, tak se za předchůdce cílového uzlu na nejkratší cestě označí zdrojový uzel (a vzdálenost cílového uzlu se přepočítá)
- Délka cesty ze zdrojového do každého z cílových uzlů může být dlouhá maximálně |U|-1 hran
- Proto pokud pustíme operaci relaxace na všechny hrany grafu |U|-1 krát, tak již musí být nalezeny všechny nejkratší cesty

```
bellman-ford(vrcholy, hrany, zdroj)

// krok 1: inicializace grafu
for each v in vrcholy
  if v=zdroj then v.vzdálenost := 0
             else v.vzdálenost := nekonečno
  v.předchůdce := null

// krok 2: opakovaně relaxovat hrany
for i from 1 to size(vrcholy)-1
  for each h in hrany    // h je hrana z u do v
    u := h.počátek
    v := h.konec
    if u.vzdálenost + h.délka < v.vzdálenost
      v.vzdálenost := u.vzdálenost + h.délka
      v.předchůdce := u

// krok 3: kontrola záporných cyklů
for each h in hrany
  u := h.počátek
  v := h.konec
  if u.vzdálenost + h.délka < v.vzdálenost
    error "Graf obsahuje záporný cyklus."
```