# Prohledávání do hloubky (DFS) – rekurzivně

```
function dfs(G, v)
    mark v as visited
    preVisit(v)
    for (v, w) ∈ E(G) do
        edgeVisit(v, w)
        if w not visited then
            dfs(G, w)
    postVisit(v)
```

# Prohledávání do šířky (BFS)

```
function bfs(G, v)
    Q ← empty queue
    mark v as visited
    enqueue(Q, v)
    while Q not empty do
        v ← dequeue(Q)
        vertexVisit(v)
        for (v, w) ∈ E(G) do
            edgeVisit(v, w)
            if w not visited then
                mark w as visited
                enqueue(Q, w)
```

# Prohledávání do šířky (BFS)

```
function bfs(G, v)
    Q ← empty queue
    mark v as visited
    enqueue(Q, v)
    while Q not empty do
        v ← dequeue(Q)
        vertexVisit(v)
        for (v, w) ∈ E(G) do
            edgeVisit(v, w)
            if w not visited then
                mark w as visited
                enqueue(Q, w)
```

- Co se stane když vyměníme frontu za zásobník?

# Tzv. „obecné schéma“ prohledávání grafu

```
function search(G, v)
    B ← empty bag
    put(B, v)
    while B not empty do
        v ← get(B)
        if v not visited then
            mark v as visited
            vertexVisit(v)
            for (v, w) ∈ E(G) do
                edgeVisit(v, w)
                put(B, w)
```

- Je toto tvrzení pravdivé? „Pokud je *bag* fronta, je toto schéma BFS, pokud je *bag* zásobník, je toto schéma DFS.“

# Prohledávání do hloubky (DFS) – iterativně

**Chceme skutečné iterativní DFS**

- musí mít všechny vlastnosti DFS, tj. zejména *postorder*
- potřebujeme simulovat rekurzi pomocí zásobníku
  - co všechno je na zásobníku rekurze?

# Prohledávání do hloubky (DFS) – iterativně

**Chceme skutečné iterativní DFS**

- musí mít všechny vlastnosti DFS, tj. zejména *postorder*
- potřebujeme simulovat rekurzi pomocí zásobníku
  - co všechno je na zásobníku rekurze?

**Varianta 1: indexy následníků (iterátory)**

- s každým prvkem v zásobníku si pamatujeme index
  - ten určuje, které následníky ještě musíme zpracovat
- použitelnost závisí na reprezentaci grafu
  - index může být obecnější iterátor
  - použitelné pro explicitní grafy (matice sousednosti, seznam následníků)
  - i pro některé implicitní grafy

# Prohledávání do hloubky (DFS) – iterativně

**Chceme skutečné iterativní DFS**

- musí mít všechny vlastnosti DFS, tj. zejména *postorder*
- potřebujeme simulovat rekurzi pomocí zásobníku
  - co všechno je na zásobníku rekurze?

**Varianta 1: indexy následníků (iterátory)**

- s každým prvkem v zásobníku si pamatujeme index
  - ten určuje, které následníky ještě musíme zpracovat
- použitelnost závisí na reprezentaci grafu
  - index může být obecnější iterátor
  - použitelné pro explicitní grafy (matice sousednosti, seznam následníků)
  - i pro některé implicitní grafy

**Varianta 2: bez iterátorů**

- u některých implicitních grafů nejsme schopni rozumně iterovat přes následníky
- vygenerujeme všechny následníky a uložíme je na zásobník
  - dva typy prvků na zásobníku (vrcholy a hrany)

# Prohledávání do hloubky (DFS) – iterativně

```
function dfs(G, v)
    S ← empty stack
    mark v as visited
    preVisit(v)
    push(S, (v, 1))
    while S not empty do
        (v, k) ← pop(S)
        if k ≤ number of v's successors in G then
            push(S, (v, k + 1))
            w ← kth successor of v in G
            edgeVisit(v, w)
            if w not visited then
                mark w as visited
                preVisit(w)
                push(S, (w, 1))
        else
            postVisit(v)
```

# Prohledávání do hloubky (DFS) – iterativně (bez iterátorů)

```
function dfs(G, v)
    S ← empty stack
    expand(G, S, v)
    while S not empty do
        top ← pop(S)
        if top is an edge (v, w) then
            edgeVisit(v, w)
            if w not visited then expand(G, S, w)
        else
            postVisit(top)

function expand(G, S, v)
    mark v as visited
    preVisit(v)
    push(S, v)
    for (v, w) ∈ E(G) do
        push(S, (v, w))
```

# Hledání nejkratších cest – Dijkstrův algoritmus

```
function dijkstra(G, s)
    P ← empty priority queue
    d[s] ← 0
    for v ∈ V(G) do
        if v ≠ s then d[v] ← ∞
        insert(P, (v, d[v]));
    while P not empty do
        v ← extractMin(P)
        mark v as closed
        for (v, w) ∈ E(G) do
            if w not closed and d[v] + δ(v, w) < d[w] then
                d[w] ← d[v] + δ(v, w)
                decreaseKey(P, w, d[w])
```

# Hledání nejkratších cest – Dijkstrův algoritmus

```
function dijkstra(G, s)
    P ← empty priority queue
    d[s] ← 0
    insert(P, (s, 0))
    while P not empty do
        v ← extractMin(P)
        mark v as closed
        for (v, w) ∈ E(G) do
            if w not closed and d[v] + δ(v, w) < d[w] then
                d[w] ← d[v] + δ(v, w)
                if w is in P then
                    decreaseKey(P, w, d[w])
                else
                    insert(P, (w, d[w]))
```

- Vkládáme do prioritní fronty vrcholy až za běhu.

# Hledání nejkratších cest – Dijkstrův algoritmus (modifikovaný)

```
function dijkstra(G, s)
    P ← empty priority queue
    d[s] ← 0
    insert(P, (s, 0))
    while P not empty do
        v ← extractMin(P)
        for (v, w) ∈ E(G) do
            if d[v] + δ(v, w) < d[w] then
                d[w] ← d[v] + δ(v, w)
                if w is in P then
                    decreaseKey(P, w, d[w])
                else
                    insert(P, (w, d[w]))
```

- Je tento algoritmus korektní? Jakou má časovou složitost?

# Hledání nejkratších cest – A*

### Motivace

- často chceme hledat pouze cestu ze startu do cíle
- preferujeme cesty, které *vypadají*, že vedou směrem k cíli

### Heuristika

- pro každý vrchol máme hodnotu $h(v)$, která je *odhadem* vzdálenosti k cíli
- **přípustná**: $h(v)$ nesmí být větší než skutečná vzdálenost k cíli
- **konzistentní**: pro každé $v$, $w$ platí: $h(v) \leq \delta(v, w) + h(w)$

### K zamyšlení:

- Je každá konzistentní heuristika přípustná?
- Je každá přípustná heuristika konzistentní?

### Algoritmus A*

- varianta Dijkstrova algoritmu: místo $d[v]$ je klíčem $d[v] + h(v)$

# Hledání nejkratších cest – A*

```
function aStar(G, s, t)
    P ← empty priority queue
    d[s] ← 0
    insert(P, (s, 0 + h(s)))
    while P not empty do
        v ← extractMin(P)
        if v = t then stop
        for (v, w) ∈ E(G) do
            if d[v] + δ(v, w) < d[w] then
                d[w] ← d[v] + δ(v, w)
                if w is in P then
                    decreaseKey(P, w, d[w] + h(w))
                else
                    insert(P, (w, d[w] + h(w)))
```

- Kterou podmínku musíme na heuristiku klást, aby byl algoritmus korektní? Jakou má složitost?

# Hledání nejkratších cest

**Zajímavé odkazy**

http://theory.stanford.edu/~amitp/GameProgramming/
http://www.redblobgames.com/pathfinding/a-star/introduction.html
http://zerowidth.com/2013/05/05/jump-point-search-explained.html

(poslední odkaz vysvětluje algoritmus Jump Point Search, heuristiku pro vylepšení algoritmu A*)