Příklady analýzy složitosti mimo Turingovi stroje

Reprezentace grafů

Graf $G = (V, E)$, kde V je množina vrcholů a E je množina hran

- neorientovaný graf: $E = \{\{u, v\} \mid u, v \in V \wedge u \neq v\}$
- orientovaný graf: $E = \{(u, v) \mid u, v \in V\}$

❖ Jak efektivně reprezentovat E

	prostor	dotaz na hranu mezi u, v	vrať následníky u
seznam hran	$O(\lvert E\rvert)$	$O(\lvert E\rvert)$	$O(\lvert E\rvert)$
matice sousednosti	$O(\lvert V\rvert^2)$	$O(1)$	$O(\lvert V\rvert)$
seznam následníků	$O(\lvert V\rvert + \lvert E\rvert)$	$O(deg(u)) \leq O(\lvert V\rvert)$	$O(deg(u)) \leq O(\lvert V\rvert)$
množina následníků	$O(\lvert V\rvert + \lvert E\rvert)$	$O(log(deg(u)) \leq O(\lvert V\rvert)$	$O(deg(u)) \leq O(\lvert V\rvert)$

* $deg(u)$ značí počet následníků vrcholu u (výstupní stupeň).

❖ Poznámka: V případě reprezentace pomocí množiny následníků, složitost dotazu na hranu záleží na datové struktuře použité pro reprezentaci množiny (stromy, hashovací tabulka). Pro jednoduchost budeme předpokládat, že dotaz na hranu umíme v $O(1)$.

Souvislost v neorientovaném grafu

- Problém: Je daný graf souvislý: $\forall u, v \in G : \mathrm{cesta}(u, v)$

$\mathrm{cesta}(v_1, v_n) \Leftrightarrow \exists$ posloupnost $v_1 e_1 v_2 e_2 \ldots e_{n-1} v_n \; : \forall 1 \leq i < n : e_i = \{v_i, v_{i+1}\} \wedge e_i \in E$

- Naivní algoritmus

$s := \mathrm{randomNode}(G)$

$reach := \{s\}, new := true$

while new **do**

    $new := false$

    foreach $\{u, v\} \in E$ **do**

        if $|\{u, v\} \cap reach| = 1$ **then**

            $reach := reach \cup \{u, v\}$

            $new := true$

return $V = reach$

Souvislost v neorientovaném grafu

- Naivní algoritmus

$s := \text{randomNode}(G)$
$reach := \{s\}, new := true$
while new **do**
    $new := false$
    foreach $\{u, v\} \in E$ **do**
        if $|\{u, v\} \cap reach| = 1$ **then**
            $reach := reach \cup \{u, v\}$
            $new := true$
return $V = reach$

- Složitost: $O(|V| \cdot |E|)$

Souvislost v neorientovaném grafu

- Asymptoticky optimální algoritmus

```
foreach v ∈ V do v.visited := false
s := randomNode(G), s.visited := true, reach := {s}
Q.enqueue(s)
while Q.notEmpty() do
    v := Q.dequeue()
    foreach u ∈ Adj(v) do
        if u.visited = false then
            u.visited := true, reach := reach ∪ {u}
            Q.enqueue(u)
return V = reach
```

Souvislost v neorientovaném grafu

- Asymptoticky optimální algoritmus

foreach $v \in V$ **do** $v.visited := false$

$s :=$ randomNode(G), $s.visited := true$, $reach := \{s\}$

Q.enqueue(s)

while Q.notEmpty() **do**

  $v := Q$.dequeue()

  foreach $u \in Adj(v)$ **do**

    if $u.visited = false$ **then**

      $u.visited := true$, $reach := reach \cup \{u\}$

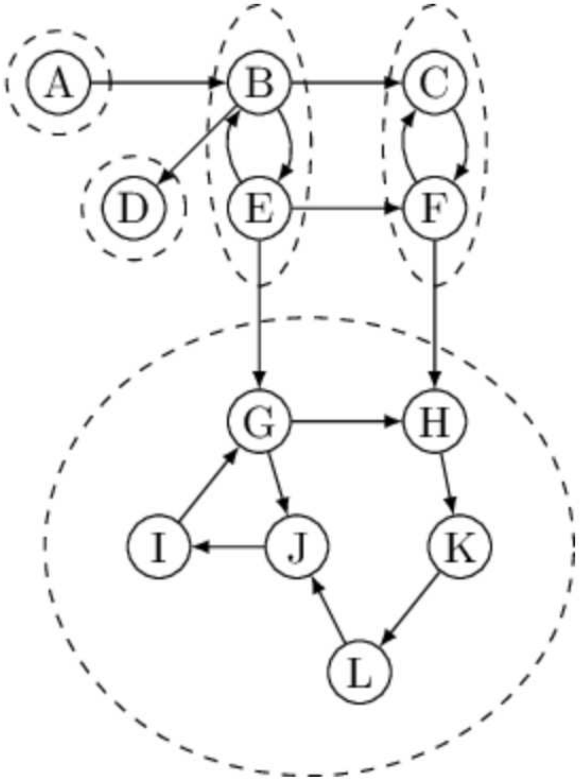
      Q.enqueue(u)

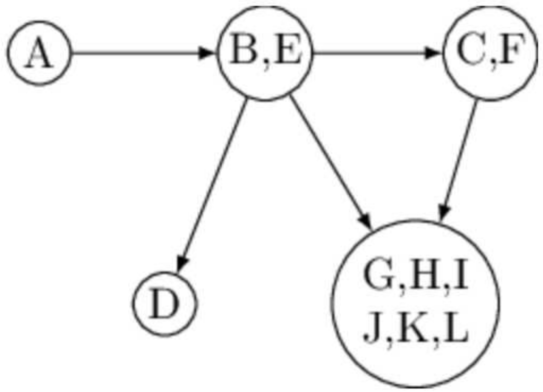
return $V = reach$

- Složitost: $O(|V| + |E|)$

Rozklad na silně souvislé komponenty

- Rozklad orientovaného grafu na silně souvislé komponenty





Rozklad na silně souvislé komponenty

- Naivní algoritmus

$SCC := \emptyset$

while $|V| \neq 0$ **do**

    $s := \mathrm{randomNode}(G)$

    $FWD := Reach(s, G)$ `// states reachable from` s `including` s

    $BWD := Reach(s, G^T)$ `// on transposed G (backward edges)`

    $C := FWD \cap BWD$

    $SCC := SCC \cup \{C\}$

    $V := V \setminus C$

return SCC

Rozklad na silně souvislé komponenty

- Naivní algoritmus

$SCC := \emptyset$
while $|V| \neq 0$ **do**
  $s := \mathrm{randomNode}(G)$
  $FWD := Reach(s, G)$ `// states reachable from` s `including` s
  $BWD := Reach(s, G^T)$ `// on transposed G (backward edges)`
  $C := FWD \cap BWD$
  $SCC := SCC \cup \{C\}$
  $V := V \setminus C$
return SCC

- Složitost: $O((|V| + |E|) \cdot |V|)$

Rozklad na silně souvislé komponenty

- Tarjanův algoritmus – modifikace DFS

- Začněme s DFS, který pro každý vrchol počítá čas návštěvy ($v.time$)

Function `Main`(G)
  foreach $v \in V$ **do** $v.time = -1$
  $gTime := 0$
  foreach $v \in V$ **do**
    if $v.time = -1$ **then**
      $gTime := gTime + 1$
      $v.time := gTime$
      `DFS`(v)

Function `DFS`$(v \in V)$
  foreach $u \in Adj(v)$ **do**
    if $u.time := -1$ **then**
      $gTime := gTime + 1$
      $u.time := gTime$
      `DFS`(u)

- Složitost: $O(|V| + |E|)$

Tarjanův algoritmus

- Budeme počítat $v.low$ and udržovat zásobník vrcholů, které nejsou v SCC.

$$v.low = \min\{u.time \mid \mathrm{Reach}(v,u) \text{ a } u \text{ byl objeven během volani } \texttt{DFS}(v)\}$$

Function Main(G)
  foreach $v \in V$ **do**
    $v.time = -1$
  $gTime := 0$
  foreach $v \in V$ **do**
    if $v.time = -1$ **then**
      $gTime := gTime + 1$
      $v.time := gTime$
      $v.low := gTime$
      $stack.\mathrm{push}(v)$
      DFS(v)
  return $SCCs$

Function DFS($v \in V$)
  foreach $u \in Adj(v)$ **do**
    if $u.time = -1$ **then**
      $gTime := gTime + 1$
      $u.time := u.low := gTime$
      $stack.\mathrm{push}(u)$
      DFS(u)
      $v.low := \min\{v.low, u.low\}$
    else if $u \in stack$ **then**
      $v.low = \min\{v.low, u.time\}$
  if $v.time = v.low$ **then** pop the $stack$ down to v to create new SCC

Tarjanův algoritmus

- Budeme počítat $v.low$ and udržovat zásobník vrcholů, které nejsou v SCC.

$$v.low = \min\{u.time \mid \mathrm{Reach}(v,u) \text{ a } u \text{ byl objeven během volani } \texttt{DFS}(v)\}$$

Function $\texttt{Main}(G)$
  foreach $v \in V$ **do**
    $v.time = -1$
  $gTime := 0$
  foreach $v \in V$ **do**
    if $v.time = -1$ **then**
      $gTime := gTime + 1$
      $v.time := gTime$
      $v.low := gTime$
      $stack.\mathrm{push}(v)$
      $\texttt{DFS}(v)$
  return $SCCs$

Function $\texttt{DFS}(v \in V)$
  foreach $u \in Adj(v)$ **do**
    if $u.time = -1$ **then**
      $gTime := gTime + 1$
      $u.time := u.low := gTime$
      $stack.\mathrm{push}(u)$
      $\texttt{DFS}(u)$
      $v.low := \min\{v.low, u.low\}$
    else if $u \in stack$ **then**
      $v.low = \min\{v.low, u.time\}$
  if $v.time = v.low$ **then** pop the $stack$ down to v to create new SCC

- Složitost: $O(|V| + |E|)$

Příklad ze zkoušky

- Nechť $G = (V, E)$ je orientovaný graf, kde V je množina vrcholů a E je množina hran. E je representovaná pomocí seznamu následníků $Succ$ a seznamu předchůdců $Pred$, tj. $Succ(v)$ vrací v konstantním čase množinu následníků vrcholu v a $Pred(v)$ vrací v konstantním čase množinu předchůdců vrcholu v.

- Dále uvažme níže popsaný algoritmus $allReach(G, v_t)$, který pro $v_t \in G$ vrací množinu vrcholů $V' \subseteq V$, která obsahuje všechny vrcholy v', pro které existuje v G cesta z v' do v_t.

i) Analyzujte asymptotickou časovou složitost algoritmu $allReach(G, v_t)$ v nejhorším případě.

ii) Analyzujte asymptotickou časovou složitost algoritmu $allReach(G, v_t)$ v nejlepším případě.

Poznámka 1: Předpokládejte uniformní cenové kritérium, kde složitost každého řádku programu je 1.

Poznámka 2: Časovou složitost analyzujte vzhledem k velikosti množin V a E.

Příklad ze zkoušky

```
Function allReach(G, v_t)
    reach := ∅
    foreach v' ∈ V do
        foreach v ∈ V do
            v.visited := false
        v'.visited := true
        Q := emptyQueue()
        Q.enqueue(v')
        while Q.notEmpty() do
            v := Q.dequeue()
            if v == v_t then
                reach := reach ∪ {v'}
                break
            foreach u ∈ Succ(v) do
                if u.visited = false then
                    u.visited := true
                    Q.enqueue(u)
    return reach
```

Příklad ze zkoušky

```
Function allReach(G, v_t)
    reach := ∅
    foreach v' ∈ V do
        foreach v ∈ V do
            v.visited := false
        v'.visited := true
        Q := emptyQueue()
        Q.enqueue(v')
        while Q.notEmpty() do
            v := Q.dequeue()
            if v == v_t then
                reach := reach ∪ {v'}
                break
            foreach u ∈ Succ(v) do
                if u.visited = false then
                    u.visited := true
                    Q.enqueue(u)
    return reach
```

- Složitost v nejhorším případě
  $O(|V| \cdot (|V| + |E|))$

Příklad ze zkoušky

```
Function allReach(G, v_t)
    reach := ∅
    foreach v' ∈ V do
        foreach v ∈ V do
            v.visited := false
        v'.visited := true
        Q := emptyQueue()
        Q.enqueue(v')
        while Q.notEmpty() do
            v := Q.dequeue()
            if  v == v_t then
                reach := reach ∪ {v'}
                break
            foreach u ∈ Succ(v) do
                if  u.visited = false then
                    u.visited := true
                    Q.enqueue(u)
    return reach
```

- Složitost v nejhorším případě
  $O(|V| \cdot (|V| + |E|))$
- Složitost v nejlepším případě
  $O(|V|^2)$

Příklad ze zkoušky

❖ Nechť $G = (V, E)$ je orientovaný graf, kde V je množina vrcholů a E je množina hran. E je representovaná pomocí seznamu následníků $Succ$ a seznamu předchůdců $Pred$, tj. $Succ(v)$ vrací v konstantním čase množinu následníků vrcholu v a $Pred(v)$ vrací v konstantním čase množinu předchůdců vrcholu v.

❖ Dále uvažme níže popsaný algoritmus $allReach(G, v_t)$, který pro $v_t \in G$ vrací množinu vrcholů $V' \subseteq V$, která obsahuje všechny vrcholy v', pro které existuje v G cesta z v' do v_t.

iii) Navrhněte algoritmus $allReach^+(G, v_t)$, který řeší stejný problém a má lepší asymptotickou časovou složitost v nejhorším případě.

iii) Analyzujte asymptotickou časovou složitost algoritmu $allReach^+(G, v_t)$ v nejhorším případě.

Poznámka 1: Předpokládejte uniformní cenové kritérium, kde složitost každého řádku programu je 1.

Poznámka 2: Časovou složitost analyzujte vzhledem k velikosti množin V a E.

Příklad ze zkoušky

Function `allReach`(G, v_t)

  $reach := \emptyset$

  foreach $v \in V$ **do**

    $v_t.visited := false$

  $v_t.visited := true$

  $Q :=$ emptyQueue()

  Q.enqueue(v_t)

  while Q.notEmpty() **do**

    $v := Q$.dequeue()

    foreach $u \in Pred(v)$ **do**

      $reach := reach \cup \{u\}$

      if $u.visited = false$ **then**

        $u.visited := true$

        Q.enqueue(u)

  return $reach$

Příklad ze zkoušky

```
Function allReach(G, v_t)
    reach := ∅
    foreach v ∈ V do
        v_t.visited := false
    v_t.visited := true
    Q := emptyQueue()
    Q.enqueue(v_t)
    while Q.notEmpty() do
        v := Q.dequeue()
        foreach u ∈ Pred(v) do
            reach := reach ∪ {u}
            if u.visited = false then
                u.visited := true
                Q.enqueue(u)
    return reach
```

- Složitost v nejhorším případě
  $$O((|V| + |E|))$$
- Složitost v nejlepším případě
  $$O(|V|)$$

Příslušnost do bezkontextového jazyka

- Uvažme bezkontextovou gramatiku $G = (N, \Sigma, P, S)$ v CNF a slovo w.
- Platí, že $w \in L(G)$?
- Připomeňme, že pokud $w \in L(G)$ pak $S \Rightarrow_G^p w \wedge p = 2|w| - 1$ (délka odvození je daná)

Příslušnost do bezkontextového jazyka

- Uvažme bezkontextovou gramatiku $G = (N, \Sigma, P, S)$ v CNF a slovo w.
- Platí, že $w \in L(G)$?
- Připomeňme, že pokud $w \in L(G)$ pak $S \Rightarrow_G^p w \wedge p = 2|w| - 1$ (délka odvození je daná)

Příslušnost do bezkontextového jazyka

- Uvažme bezkontextovou gramatiku $G = (N, \Sigma, P, S)$ v CNF a slovo w.
- Platí, že $w \in L(G)$?
- Připomeňme, že pokud $w \in L(G)$ pak $S \Rightarrow_G^p w \wedge p = 2|w| - 1$ (délka odvození je daná)

```
foreach derivation tree T in G of the depth p do
    if leafs of T form w then
        return true
return false
```

Příslušnost do bezkontextového jazyka

- Uvažme bezkontextovou gramatiku $G = (N, \Sigma, P, S)$ v CNF a slovo w.
- Platí, že $w \in L(G)$?
- Připomeňme, že pokud $w \in L(G)$ pak $S \Rightarrow_G^p w \wedge p = 2|w| - 1$ (délka odvození je daná)

```
foreach derivation tree T in G of the depth p do
    if leafs of T form w then
        return true
return false
```

- Složitost: $2^{O(|w|)}$ algoritmus projde všechny odvození s délkou $O(|w|)$
  - připomeňme
    $$2^{O(f(n))} = \{g(n) \in \mathcal{F} \mid \exists c \in \mathbb{R}^+ \; \exists n_0 \in \mathbb{N} \; \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow g(n) \leq 2^{c.f(n)}\}$$
  - pro jednoduchost zanedbáváme $|P|$

Příslušnost do bezkontextového jazyka

❖ Cocke-Younger-Kasami (CYK) algoritmus (dynamické programování)

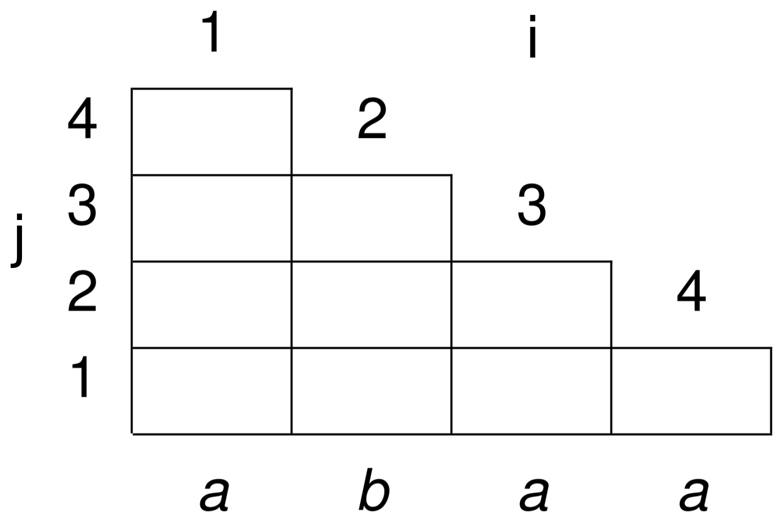
❖ Hlavní myšlenka: pro každé neprázdné podslovo u slova w (začíná na indexu i a má délku j) spočítáme množinu všech neterminálů z kterých lze u odvodit

$$T_{i,j} = \{X \in N \mid S \Rightarrow_G^* w_i w_{i+1} \dots w_{i+j-1}\}$$

❖ Příklad

$S \rightarrow AB \mid SS \mid a$
$A \rightarrow AA \mid BC \mid a$
$B \rightarrow AB \mid b$
$C \rightarrow SA \mid b$

$w = abaa$



Příslušnost do bezkontextového jazyka

❖ Cocke-Younger-Kasami (CYK) algoritmus (dynamické programování)

❖ Hlavní myšlenka: pro každé neprázdné podslovo u slova w (začíná na indexu i a má délku j) spočítáme množinu všech neterminálů z kterých lze u odvodit

$$T_{i,j} = \{X \in N \mid S \Rightarrow^*_G w_i w_{i+1} \dots w_{i+j-1}\}$$

❖ Příklad

$S \to AB \mid SS \mid a$
$A \to AA \mid BC \mid a$
$B \to AB \mid b$
$C \to SA \mid b$

$w = abaa$

j \ i	1	2	3	4
4	S,C,A			
3	S,C	A		
2	S,B	$\emptyset$	S,C,A	
1	S,A	B,C	S,A	S,A
	a	b	a	a

❖ $w \in L(G)$ jelikož $S \in T_{1,4}$

Cocke-Younger-Kasami algoritmus

Vstup: gramatika $\mathcal{G} = (N, \Sigma, P, S)$ v CNF, slovo $w = w_1 \dots w_n \neq \varepsilon$
Výstup: množiny $T_{i,j} = \{X \in N \mid X \Rightarrow^* w_i \dots w_{i+j-1}\}$

<u>**for**</u> $i \leftarrow 1$ <u>**to**</u> n <u>**do**</u>
    $T_{i,1} \leftarrow \emptyset$
    <u>**for**</u> každé pravidlo tvaru $(A \to a) \in P$ <u>**do**</u>
        <u>**if**</u> $a = w_i$ <u>**then**</u> $T_{i,1} \leftarrow T_{i,1} \cup \{A\}$
    <u>**od**</u>
<u>**od**</u>
<u>**for**</u> $j \leftarrow 2$ <u>**to**</u> n <u>**do**</u>
    <u>**for**</u> $i \leftarrow 1$ <u>**to**</u> $n - j + 1$ <u>**do**</u>
        $T_{i,j} \leftarrow \emptyset$
        <u>**for**</u> $k \leftarrow 1$ <u>**to**</u> $j - 1$ <u>**do**</u>
            <u>**for**</u> každé pravidlo tvaru $(A \to BC) \in P$ <u>**do**</u>
                <u>**if**</u> $B \in T_{i,k} \wedge C \in T_{i+k,j-k}$ <u>**then**</u> $T_{i,j} \leftarrow T_{i,j} \cup \{A\}$
            <u>**od**</u>
        <u>**od**</u>
    <u>**od**</u>
<u>**od**</u>

Cocke-Younger-Kasami algoritmus

Vstup: gramatika $\mathcal{G} = (N, \Sigma, P, S)$ v CNF, slovo $w = w_1 \dots w_n \neq \varepsilon$
Výstup: množiny $T_{i,j} = \{X \in N \mid X \Rightarrow^* w_i \dots w_{i+j-1}\}$

```
for i ← 1 to n do
    T_{i,1} ← ∅
    for každé pravidlo tvaru (A → a) ∈ P do
        if a = w_i then T_{i,1} ← T_{i,1} ∪ {A}
    od
od
for j ← 2 to n do
    for i ← 1 to n − j + 1 do
        T_{i,j} ← ∅
        for k ← 1 to j − 1 do
            for každé pravidlo tvaru (A → BC) ∈ P do
                if B ∈ T_{i,k} ∧ C ∈ T_{i+k,j−k} then T_{i,j} ← T_{i,j} ∪ {A}
            od
        od
    od
od
```

- Složitost: $O(|w|^3)$
  - opět zanedbáváme $|P|$ – jinak $O(|w|^3 \cdot |P|)$

Univerzalita NKA

- Pro daný nedeterministický konečný automat $A = \{Q, \Sigma, \delta, s_0, F\}$ rozhodni zda

$$L(A) \neq \Sigma^*.$$

- Naivní algoritmus

`Construct deterministic finite automaton` A' `such that` $L(A) = L(A')$;
if *exists in* A' *a path from* $\{s_0\}$ *to* S *where* $F \cap S = \emptyset$ **then**
    return *true*
else
    return *false*

Univerzalita NKA

❖ Pro daný nedeterministický konečný automat $A = \{Q, \Sigma, \delta, s_0, F\}$ rozhodni zda

$$L(A) \neq \Sigma^*.$$

❖ Naivní algoritmus

Construct deterministic finite automataon A' **such that** $L(A) = L(A')$;
if *exists in* A' *a path from* $\{s_0\}$ *to* S *where* $F \cap S = \emptyset$ **then**
  return *true*
else
  return *false*

❖ Prostorová složitost: $2^{O(|Q|)}$

- deterministický automat může mít až exponenciální počet stavů
- dále si musíme uložit přechodovou relaci

❖ Časová složitost: $2^{O(|Q|)}$

- hledání neakceptující cesty v exponenciálně velkém automatu

Univerzalita NKA

- Pro daný NKA $A = \{Q, \Sigma, \delta, s_0, F\}$ rozhodni zda $L(A) \neq \Sigma^*$.

- Algoritmus pracující v nedeterministickém polynomiálním prostoru
  - on-the-fly determinizace
  - uhodnutí neakceptující cesty

$counter := 0;$
$makroState := \{s_0\};$
while $counter \leq 2^{|Q|}$ **do**
    if $makroState \cap F = \emptyset$ **then**
        return $true$
    $a :=$ `nondeterministically choose from` $\Sigma;$
    $makroState := \bigcup_{q \in makroState} \delta(q, a);$
    $counter := counter + 1;$
return $false$

Univerzalita NKA

- Pro daný NKA $A = \{Q, \Sigma, \delta, s_0, F\}$ rozhodni zda $L(A) \neq \Sigma^*$.

- Algoritmus pracující v nedeterministickém polynomiálním prostoru
  - on-the-fly determinizace
  - uhodnutí neakceptující cesty

$counter := 0;$
$makroState := \{s_0\};$
while $counter \leq 2^{|Q|}$ **do**
    if $makroState \cap F = \emptyset$ **then**
        return $true$
    $a :=$ `nondeterministically choose from` $\Sigma;$
    $makroState := \bigcup_{q \in makroState} \delta(q, a);$
    $counter := counter + 1;$
return $false$

- Prostorová složitost: $O(|Q|) \subseteq$ **NPSPACE** = **PSPACE**
  - pamatujeme si jen 2 makrostavy (každý max $|Q|$) a $counter$ ($log(2^{|Q|}) \in O(|Q|)$)

Univerzalita NKA

- Pro daný NKA $A = \{Q, \Sigma, \delta, s_0, F\}$ rozhodni zda $L(A) \neq \Sigma^*$.

- Algoritmus pracující v nedeterministickém polynomiálním prostoru
  - on-the-fly determinizace
  - uhodnutí neakceptující cesty

$counter := 0$;
$makroState := \{s_0\}$;
while $counter \leq 2^{|Q|}$ **do**
  if $makroState \cap F = \emptyset$ **then**
    return $true$
  $a :=$ `nondeterministically choose from` Σ;
  $makroState := \bigcup_{q \in makroState} \delta(q, a)$;
  $counter := counter + 1$;
return $false$

- Časová složitost stále v $2^{O(|Q|)}$
  - pozor není v **NP**, jelikož délka cesty může být exponenciální k $|Q|$ – její ověření není v **P**

Amortizovaná složitost

Amortizovaná složitost

- Technika dovolující přesnější analýzu složitosti v nejhorším případě pro danou posloupnost operací.

- Klasický přístup analyzuje složitost jednotlivých operací. Výsledná složitost je součtem jednotlivých operací.

- Technika amortizace analyzuje posloupnost jako celek. Stojí zejména na pozorování, že drahé operace mohou nastat pouze jednou za delší dobu a tudíž se jejich cena amortizuje.

- Existují různé metody dovolující analyzovat amortizovanou složitost: **seskupování**, **metoda účtů**, potenciálové funkce

Amortizovaná složitost – příklad

- Uvažme zásobník S a operace PUSH(S, x), POP(S) a MULTIPOP(S, k) – odebere k prvků případně vyprázdní zásobník pokud $|S| \leq k$).

- Uvažujme libovolnou posloupnost n operací.

- Každá operace PUSH(S, x) a POP(S) má složitost 1. V posloupnosti n operací má MULTIPOP(S, k) v nejhorším případě složitost $O(n)$.

- Naivní analýza časové složitosti (v nejhorším případě) pro n operací je $\mathbf{O(n^2)}$.

- Existuje posloupnost operací, která má tuto složitost? Můžeme tuto analýzu zpřesnit?

Amortizovaná složitost – příklad

- Uvažme zásobník S a operace $\textsc{Push}(S, x)$, $\textsc{Pop}(S)$ a $\textsc{MultiPop}(S, k)$ – odebere k prvků případně vyprázdní zásobník pokud $|S| \leq k$.

- **Metoda seskupování:** Rozdělíme operace do skupin a analyzujeme jejich složitost. Celková složitost je menší nebo rovna součtu složitostí jednotlivých skupin.

- Skupina 1: Posloupnost n operací $\textsc{Push}(S, x)$ má složitost n

- Skupina 2: Složitost posloupnosti operací $\textsc{Pop}(S)$ a $\textsc{MultiPop}(S, k)$ je daná počtem prvků odebraných z S a tudíž nemůže být větší než počet provedených operací $\textsc{Push}(S, x)$. Složitost celé skupiny je tedy menší než n.

- Celková složitost libovolných n operací je menší než $\mathbf{2n}$.

Amortizovaná složitost – metoda účtů

❖ Každé operaci přiřadíme kredit. Při realizaci operace zaplatíme její cenu dle následujících pravidel:

- pokud cena operace $\leq$ kredit operace, tak operaci zaplatíme kredity a zbylé kredity vložíme na účet
- pokud cena operace $\geq$ kredit operace, tak scházející kredity odebereme z účtu

❖ Na začátku je na účtě 0 kreditů. Pokud během celého výpočtu je počet kreditů na účtě **nezáporný**, tak platí, že součet kreditů vykonaných operací $\geq$ složitost těchto operací.

Amortizovaná složitost – metoda účtů

❖ Každé operaci přiřadíme cenu a kredit. Při realizaci operace zaplatíme její cenu dle následujících pravidel:

- pokud cena operace $\leq$ kredit operace, tak operaci zaplatíme kredity a zbylé kredity vložíme na účet
- pokud cena operace $\geq$ kredit operace, tak scházející kredity odebereme z účtu

❖ Na začátku je na účtě 0 kreditů. Pokud během celého výpočtu je počet kreditů na účtě **nezáporný**, tak platí, že součet kreditů vykonaných operací $\geq$ složitost těchto operací.

❖ Zpět k našemu příkladu

operace	cena	kredit
PUSH(S, x)	1	2
POP(S)	$\min\{1, \|S\|\}$	0
MULTIPOP(S, k)	$\min\{k, \|S\|\}$	0

Při operaci PUSH(S, x) se předplatíme jeden kredit na případné odstranění vloženého prvku

❖ Zřejmě platí invariant: Počet kreditů na účtu je rovný počtu prvkou v S. Tudíž skutečně platí, že počet kreditů na účtě je vždy nezáporný. Celková cena n operací je $\leq 2n$.

Amortizovaná složitost – ukázka

- Příklad: dynamicky alokovaná tabulka T (odebraní záznamu zneplatní řádek)
  - neznámá velikost – dynamická realokace mění kapacitu
  - při naplnění tabulky alokujeme větší tabulku a záznamy do ní přesuneme
  - při vyprázdnění na určitou mez alokujeme menší tabulku a záznamy do ní přesuneme
  - $\alpha(T)$ je poměr platných záznamů (velikost) ku kapacitě tabulky
  - pokud je $\alpha(T) = 1$ vložení záznamů vede k alokaci 2-krát větší tabulky a přesunu
  - pokud je $\alpha(T) \leq 0.5$ odebrání záznamů vede k alokaci 2-krát menší tabulky a přesunu
- Asymptotická složitost nejhoršího případu
  - složitost jedné operace vložení i odebrání je rovna n (velikost tabulky)
  - složitost n operací je v nejhorším případě $O(n^2)$

Amortizovaná složitost – ukázka

- Amortizovaná složitost – seskupování
  - n operací vložení:
    - c_i – cena i-té operace vložení
    - $c_i = i$ pokud $i - 1$ je mocninou 2 , jinak $c_i = 1$
    - $\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + 2n = 3n$

- Amortizovaná složitost – metoda účtů
  - každé operaci vložení dáme 3 kredity
    - 1 kredit zaplatí vložení,
    - 1 kredit zůstane na účtě pro přesun tohoto záznamu
    - 1 kredit zůstane na účtě pro přesun záznamu, který je v tabulce, ale nemá žádný kredit
  - nechť byla alokována tabulka o velikosti m a bylo sem přesunuto $m/2$ záznamů
    - těchto $m/2$ záznamů nemá na účtě žádný kredit
    - než se tabulka znovu zaplní, bude na účtě m kreditů, které zaplatí přesun
  - celková cena n operací vložení je $\leq 3n$

Amortizovaná složitost – ukázka 2

- existuje posloupnost n operací (vložení, odebrání) se složitostí $\Theta(n^2)$
  - střídavě přidáváme odebíráme záznamy kolem faktoru $\alpha(T) = 0.5$
  - každá třetí operace má cenu n

❖ Existuje implementace s lineární amortizovanou složitostí?

Amortizovaná složitost – ukázka 2

- existuje posloupnost n operací (vložení, odebrání) se složitostí $\Theta(n^2)$
  - střídavě přidáváme odebíráme záznamy kolem faktoru $\alpha(T) = 0.5$
  - každá třetí operace má cenu n

❖ Existuje implementace s lineární amortizovanou složitostí?

- zabráníme tomu, že kapacita může oscilovat mezi dvěma hodnotami při malém počtu operací (viz výše)
- pokud je $\alpha(T) \leq 0.25$ odebrání záznamu vede k alokaci 2-krát menšího pole a přesunu
- amortizovaná složitost libovolných n operací (seskupením, hlavní myšlenka)
  - $2^k + 1$ operací vložení, kapacita 2^{k+1}, cena $< 3(2^k + 1)$
  - realokaci vyvolá $2^{k-1} + 1$ operací odebrání, kapacita je 2^k, cena $2^{k-1} + 2^k + 1$
  - další realokaci vyvolá $2^{k-1} + 1$ operací vložení, kapacita 2^{k+1}, cena $2^{k-1} + 2^k + 1$
  - celkově operací: $2^k + 1 + 2 * 2^{k-1} + 2 = 2^{k+1} + 3$
  - celkově cena: $2 * 2^{k-1} + 5 * 2^k + 5 = 6 * 2^k + 5 = 3 * 2^{k+1} + 5$
  - ukázali jsem, že n operací má cenu $O(n)$

Amortizovaná složitost – ukázka 2

- existuje posloupnost n operací (vložení, odebrání) se složitostí $\Theta(n^2)$
  - střídavě přidáváme odebíráme záznamy kolem faktoru $\alpha(T) = 0.5$
  - každá třetí operace má cenu n

❖ Existuje implementace s lineární amortizovanou složitostí?

- zabráníme tomu, že kapacita může oscilovat mezi dvěma hodnotami při malém počtu operací (viz výše)
- pokud je $\alpha(T) \leq 0.25$ odebrání záznamu vede k alokaci 2-krát menšího pole a přesunu
- amortizovaná složitost libovolných n operací (metodou účtů, hlavní myšlenka)
  - každé operaci vložení dáme 3 kredity (viz výše)
  - každé operaci odebraní dáme 2 kredity: 1 kredit na účet pro realokaci při odebírání
  - nechť byla alokována tabulka o velikosti m a bylo sem přesunuto $m/2$ záznamů bez kreditů
  - realokace při vkládání (viz výše)
  - než se tabulka zmenší na $m/4$, bude na účtě $m/4$ kreditů, které zaplatí přesun
  - celková cena n libovolných operací je $\leq 5n$

Příklad ze zkoušky

Uvažme dynamicky alokovanou datovou strukturu $mnozina$ implementující konečnou množinu přirozených čísel pomocí uspořádaného jednosměrně vázaného lineárního seznamu (předpokládejte, že uspořádání je rostoucí posloupnost) s ukazatelem $head$ na začátek. $size$ označuje počet prvků v seznamu.

- Operace $mnozina.insert(x)$ vloží na správnou pozici do seznamu prvek x, pokud v seznamu není (jinak jde o prázdnou operaci). Tato operace má lineární časovou složitost k velikosti seznamu.
- Operace $mnozina.prune(y)$ odstraní z množiny všechny prvky menší než y a je implementována následujícím způsobem:

```
Function prune(y)
    while head ≠ null ∧ head → value < y do
        tmp := head
        head := head → next
        print(tmp → value)
        free(tmp)
```

Příklad ze zkoušky

```
Function prune(y)
    while head ≠ null ∧ head → value < y do
        tmp := head
        head := head → next
        print(tmp → value)
        free(tmp)
```

❖ Uvažme posloupnost n operací $mnozina.insert(x)$ a $mnozina.prune(y)$.

- Analyzujte a zdůvodněte amortizovanou časovou složitost jedné operace $mnozina.prune(x)$.

Poznámka: Předpokládejte uniformní cenové kritérium, kde složitost každého řádku programu je 1.

Příklad ze zkoušky

Function prune(y)
  while $head \neq \texttt{null} \wedge head \rightarrow value < y$ **do**
    $tmp := head$
    $head := head \rightarrow next$
    print($tmp \rightarrow value$)
    free(tmp)

operace	cena	kredit
$insert(x)$	$O(size)$	$O(size) + 5$
$prune(x)$	$1 + 5\cdot$ počet odstraněných prvků	1

- Každá operace $insert(x)$ vloží na účet 5 kreditů.

- Platí invariant: na účtě je vždy $5 \cdot size$ kreditů. Stačí si uvědomit, že každá operace $prune(x)$ zaplatí z účtu 5 kreditů (podmínka + tělo while cyklu) za každý odstraněný prvek. 1 kredit operace $prune(x)$ je na zaplacení podmínky while cyklu, která se vyhodnotí na false.

- Z invariantu přímo plyne, že počet kreditů neklesne pod 0. Amortizovaná cena operace $prune(x)$ je 1 (počet kreditů, který za ni platíme).