

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	$O(E)$	$O(E)$	$O(E)$
matice sousednosti	$O(V ^2)$	$O(1)$	$O(V)$
seznam následníků	$O(V + E)$	$O(deg(u)) \leq O(V)$	$O(deg(u)) \leq O(V)$
množina následníků	$O(V + E)$	$O(\log(deg(u)) \leq O(V)$	$O(deg(u)) \leq O(V)$

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

❖ Poznámka: V případě reprezentace pomocí množiny následníků, složitost dotazu na hranu závisí 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: $\forall u, v \in G : \text{cesta}(u, v)$

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

❖ Naivní algoritmus

```
1  $s := \text{randomNode}(G)$ 
2  $reach := \{s\}, new := true$ 
3 while  $new$  do
4    $new := false$ 
5   foreach  $\{u, v\} \in E$  do
6     if  $|\{u, v\} \cap reach| = 1$  then
7        $reach := reach \cup \{u, v\}$ 
8        $new := true$ 
9 return  $V = reach$ 
```

Souvislost v neorientovaném grafu

❖ Naivní algoritmus

```
1  $s := \text{randomNode}(G)$ 
2  $reach := \{s\}, new := true$ 
3 while  $new$  do
4    $new := false$ 
5   foreach  $\{u, v\} \in E$  do
6     if  $|\{u, v\} \cap reach| = 1$  then
7        $reach := reach \cup \{u, v\}$ 
8        $new := true$ 
9 return  $V = reach$ 
```

❖ Složitost: $O(|V| \cdot |E|)$

Souvislost v neorientovaném grafu

❖ Asymptoticky optimální algoritmus

```
1 foreach  $v \in V$  do  $v.visited := false$ 
2  $s := \text{randomNode}(G)$ ,  $s.visited := true$ ,  $reach := \{s\}$ 
3  $Q.enqueue(s)$ 
4 while  $Q.notEmpty()$  do
5    $v := Q.dequeue()$ 
6   foreach  $u \in Adj(v)$  do
7     if  $u.visited = false$  then
8        $u.visited := true$ ,  $reach := reach \cup \{u\}$ 
9        $Q.enqueue(u)$ 
10 return  $V = reach$ 
```

Souvislost v neorientovaném grafu

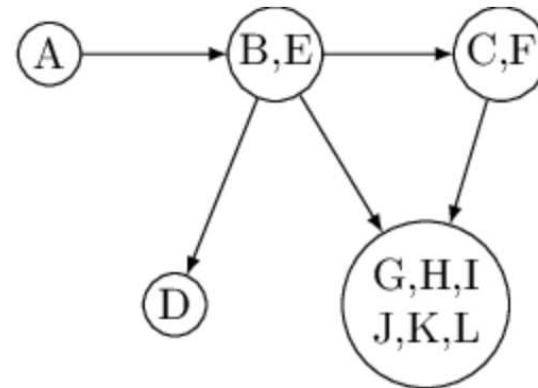
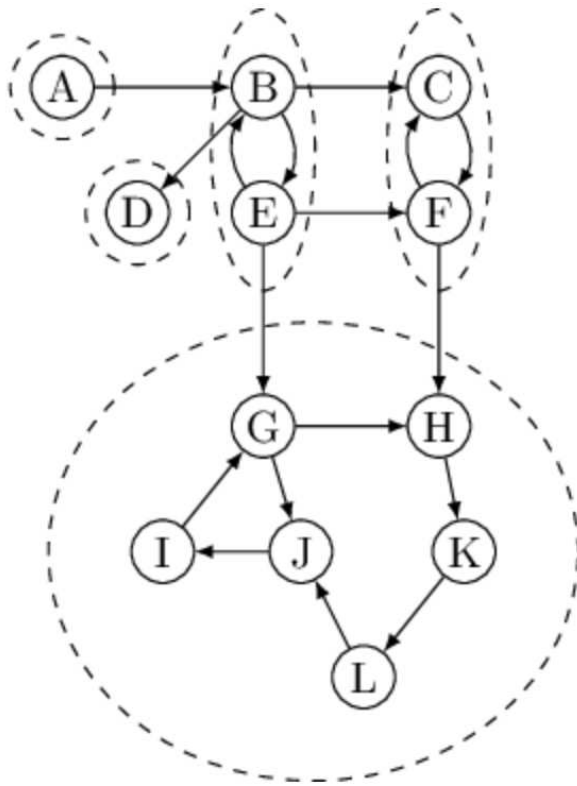
❖ Asymptoticky optimální algoritmus

```
1 foreach  $v \in V$  do  $v.visited := false$ 
2  $s := \text{randomNode}(G)$ ,  $s.visited := true$ ,  $reach := \{s\}$ 
3  $Q.enqueue(s)$ 
4 while  $Q.notEmpty()$  do
5    $v := Q.dequeue()$ 
6   foreach  $u \in Adj(v)$  do
7     if  $u.visited = false$  then
8        $u.visited := true$ ,  $reach := reach \cup \{u\}$ 
9        $Q.enqueue(u)$ 
10 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

```
1  $SCC := \emptyset$ 
2 while  $|V| \neq 0$  do
3    $s := \text{randomNode}(G)$ 
4    $FWD := \text{Reach}(s, G)$  // states reachable from  $s$  including  $s$ 
5    $BWD := \text{Reach}(s, G^T)$  // on transposed  $G$  (backward edges)
6    $C := FWD \cap BWD$ 
7    $SCC := SCC \cup \{C\}$ 
8    $V := V \setminus C$ 
9 return  $SCC$ 
```


Rozklad na silně souvislé komponenty

❖ Naivní algoritmus

```
1  $SCC := \emptyset$ 
2 while  $|V| \neq 0$  do
3    $s := \text{randomNode}(G)$ 
4    $FWD := \text{Reach}(s, G)$  // states reachable from  $s$  including  $s$ 
5    $BWD := \text{Reach}(s, G^T)$  // on transposed  $G$  (backward edges)
6    $C := FWD \cap BWD$ 
7    $SCC := SCC \cup \{C\}$ 
8    $V := V \setminus C$ 
9 return  $SCC$ 
```

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

Rozklad na silně souvislé komponenty

❖ Tarjanův algoritmus – modifikace DFS

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

```
1 Function Main( $G$ )
2   foreach  $v \in V$  do  $v.time = -1$ 
3    $gTime := 0$ 
4   foreach  $v \in V$  do
5     if  $v.time = -1$  then
6        $gTime := gTime + 1$ 
7        $v.time := gTime$ 
8       DFS( $v$ )
```

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

❖ Složitost: $O(|V| + |E|)$

Tarjanův algoritmus

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

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

```
1 Function Main( $G$ )
2   foreach  $v \in V$  do
3     |  $v.time = -1$ 
4    $gTime := 0$ 
5   foreach  $v \in V$  do
6     | if  $v.time = -1$  then
7       |  $gTime := gTime + 1$ 
8       |  $v.time := gTime$ 
9       |  $v.low := gTime$ 
10      |  $stack.push(v)$ 
11      | DFS( $v$ )
12  return  $SCCs$ 
```

```
1 Function DFS( $v \in V$ )
2   | foreach  $u \in Adj(v)$  do
3     | if  $u.time = -1$  then
4       |  $gTime := gTime + 1$ 
5       |  $u.time := u.low := gTime$ 
6       |  $stack.push(u)$ 
7       | DFS( $u$ )
8       |  $v.low := \min\{v.low, u.low\}$ 
9     | else if  $u \in stack$  then
10      |  $v.low = \min\{v.low, u.time\}$ 
11  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.

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

```
1 Function Main( $G$ )
2   foreach  $v \in V$  do
3      $v.time = -1$ 
4    $gTime := 0$ 
5   foreach  $v \in V$  do
6     if  $v.time = -1$  then
7        $gTime := gTime + 1$ 
8        $v.time := gTime$ 
9        $v.low := gTime$ 
10       $stack.push(v)$ 
11      DFS( $v$ )
12  return  $SCCs$ 
```

```
1 Function DFS( $v \in V$ )
2   foreach  $u \in Adj(v)$  do
3     if  $u.time = -1$  then
4        $gTime := gTime + 1$ 
5        $u.time := u.low := gTime$ 
6        $stack.push(u)$ 
7       DFS( $u$ )
8        $v.low := \min\{v.low, u.low\}$ 
9     else if  $u \in stack$  then
10       $v.low = \min\{v.low, u.time\}$ 
11  if  $v.time = v.low$  then pop the
    $stack$  down to  $v$  to create new SCC
```

- ❖ Složitost: $O(|V| + |E|)$

Příklad ze zkoušky

- ❖ Necht' $G = (V, E)$ je orientovaný graf, kde V je množina vrcholů a E je množina hran. E je reprezentovaná 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

```
1 Function allReach( $G, v_t$ )
2    $reach := \emptyset$ 
3   foreach  $v' \in V$  do
4     foreach  $v \in V$  do
5        $v.visited := false$ 
6      $v'.visited := true$ 
7      $Q := \text{emptyQueue}()$ 
8      $Q.enqueue(v')$ 
9     while  $Q.notEmpty()$  do
10       $v := Q.dequeue()$ 
11      if  $v == v_t$  then
12         $reach := reach \cup \{v'\}$ 
13        break
14      foreach  $u \in Succ(v)$  do
15        if  $u.visited = false$  then
16           $u.visited := true$ 
17           $Q.enqueue(u)$ 
18  return  $reach$ 
```

Příklad ze zkoušky

```
1 Function allReach( $G, v_t$ )
2    $reach := \emptyset$ 
3   foreach  $v' \in V$  do
4     foreach  $v \in V$  do
5        $v.visited := false$ 
6      $v'.visited := true$ 
7      $Q := emptyQueue()$ 
8      $Q.enqueue(v')$ 
9     while  $Q.notEmpty()$  do
10       $v := Q.dequeue()$ 
11      if  $v == v_t$  then
12         $reach := reach \cup \{v'\}$ 
13        break
14      foreach  $u \in Succ(v)$  do
15        if  $u.visited = false$  then
16           $u.visited := true$ 
17           $Q.enqueue(u)$ 
18  return  $reach$ 
```

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

Příklad ze zkoušky

```
1 Function allReach( $G, v_t$ )
2    $reach := \emptyset$ 
3   foreach  $v' \in V$  do
4     foreach  $v \in V$  do
5        $v.visited := false$ 
6      $v'.visited := true$ 
7      $Q := emptyQueue()$ 
8      $Q.enqueue(v')$ 
9     while  $Q.notEmpty()$  do
10       $v := Q.dequeue()$ 
11      if  $v == v_t$  then
12         $reach := reach \cup \{v'\}$ 
13        break
14      foreach  $u \in Succ(v)$  do
15        if  $u.visited = false$  then
16           $u.visited := true$ 
17           $Q.enqueue(u)$ 
18  return  $reach$ 
```

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

❖ Složitost v nejlepším případě
 $O(|V|^2)$

Příklad ze zkoušky

- ❖ Necht' $G = (V, E)$ je orientovaný graf, kde V je množina vrcholů a E je množina hran. E je reprezentovaná 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

```
1 Function allReach( $G, v_t$ )
2    $reach := \emptyset$ 
3   foreach  $v \in V$  do
4     |    $v_t.visited := false$ 
5      $v_t.visited := true$ 
6      $Q := \text{emptyQueue}()$ 
7      $Q.enqueue(v_t)$ 
8     while  $Q.notEmpty()$  do
9       |    $v := Q.dequeue()$ 
10      |   foreach  $u \in Succ(v)$  do
11        |   |    $reach := reach \cup \{u\}$ 
12        |   |   if  $u.visited = false$  then
13          |   |   |    $u.visited := true$ 
14          |   |   |    $Q.enqueue(u)$ 
15   return  $reach$ 
```

Příklad ze zkoušky

```
1 Function allReach( $G, v_t$ )
2    $reach := \emptyset$ 
3   foreach  $v \in V$  do
4      $v_t.visited := false$ 
5    $v_t.visited := true$ 
6    $Q := emptyQueue()$ 
7    $Q.enqueue(v_t)$ 
8   while  $Q.notEmpty()$  do
9      $v := Q.dequeue()$ 
10    foreach  $u \in Succ(v)$  do
11       $reach := reach \cup \{u\}$ 
12      if  $u.visited = false$  then
13         $u.visited := true$ 
14         $Q.enqueue(u)$ 
15  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)

```
1 foreach derivation tree  $T$  in  $G$  of the depth  $p$  do
2   |   if leafs of  $T$  form  $w$  then
3     |   |   return true
4 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)

```
1 foreach derivation tree  $T$  in  $G$  of the depth  $p$  do
2   |   if leafs of  $T$  form  $w$  then
3   |   |   return true
4 return false
```

- ❖ Složitost: $2^{O(|w|)}$ algoritmus projde všechny odvození s délkou $O(|w|)$

- připomeňme

$$2^{O(f(n))} = \{g(n) \in \mathcal{F} \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \in \mathbb{N} : n \geq n_0 \Rightarrow g(n) \leq 2^{c \cdot f(n)}\}$$

- 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

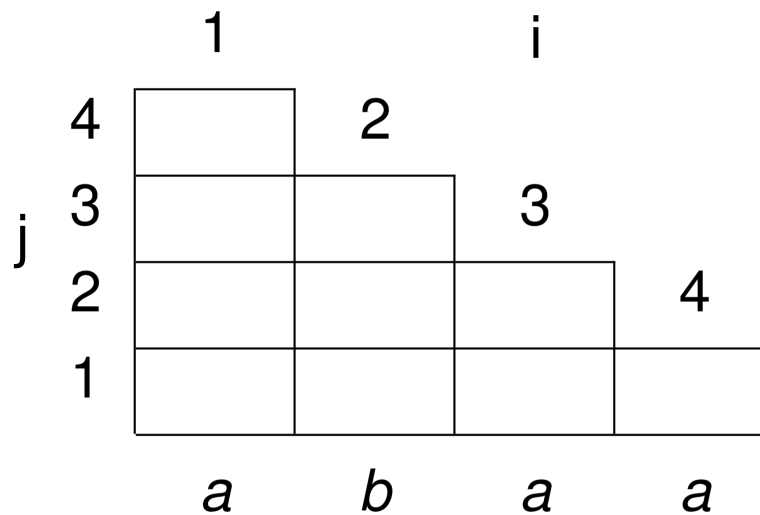
$S \rightarrow AB \mid SS \mid a$

$A \rightarrow AA \mid BC \mid a$

$B \rightarrow AB \mid b$

$C \rightarrow SA \mid b$

$w = abaa$



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

$S \rightarrow AB \mid SS \mid a$

$A \rightarrow AA \mid BC \mid a$

$B \rightarrow AB \mid b$

$C \rightarrow SA \mid b$

$w = abaa$

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

		1		i	
	4	S,C,A	2		
	3	S,C	A	3	
j	2	S,B	∅	S,C,A	4
	1	S,A	B,C	S,A	S,A
		a	b	a	a

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 $i \leftarrow 1$ **to** n **do**

$T_{i,1} \leftarrow \emptyset$

for každé pravidlo tvaru $(A \rightarrow a) \in P$ **do**

if $a = w_i$ **then** $T_{i,1} \leftarrow T_{i,1} \cup \{A\}$

od

od

for $j \leftarrow 2$ **to** n **do**

for $i \leftarrow 1$ **to** $n - j + 1$ **do**

$T_{i,j} \leftarrow \emptyset$

for $k \leftarrow 1$ **to** $j - 1$ **do**

for každé pravidlo tvaru $(A \rightarrow BC) \in P$ **do**

if $B \in T_{i,k} \wedge C \in T_{i+k,j-k}$ **then** $T_{i,j} \leftarrow T_{i,j} \cup \{A\}$

od

od

od

od

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 $i \leftarrow 1$ **to** n **do**

$T_{i,1} \leftarrow \emptyset$

for každé pravidlo tvaru $(A \rightarrow a) \in P$ **do**

if $a = w_i$ **then** $T_{i,1} \leftarrow T_{i,1} \cup \{A\}$

od

od

for $j \leftarrow 2$ **to** n **do**

for $i \leftarrow 1$ **to** $n - j + 1$ **do**

$T_{i,j} \leftarrow \emptyset$

for $k \leftarrow 1$ **to** $j - 1$ **do**

for každé pravidlo tvaru $(A \rightarrow BC) \in P$ **do**

if $B \in T_{i,k} \wedge C \in T_{i+k,j-k}$ **then** $T_{i,j} \leftarrow T_{i,j} \cup \{A\}$

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

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

Univerzalita NKA

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

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

- ❖ Naivní algoritmus

```
1 Construct deterministic finite automaton  $A'$  such that  $L(A) = L(A')$ ;  
2 if exists in  $A'$  a path from  $\{s_0\}$  to  $S$  where  $F \cap S = \emptyset$  then  
3 | return true  
4 else  
5 | 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

```
1 counter := 0;
2 makroState := {s0};
3 while counter ≤ 2|Q| do
4   | if makroState ∩ F = ∅ then
5     | return true
6   | a := nondeterministically choose from Σ;
7   | makroState := ∪q∈makroState δ(q, a);
8   | counter := counter + 1;
9 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

```
1 counter := 0;
2 makroState := {s0};
3 while counter ≤ 2|Q| do
4   | if makroState ∩ F = ∅ then
5   |   | return true
6   | a := nondeterministically choose from Σ;
7   | makroState := ∪q∈makroState δ(q, a);
8   | counter := counter + 1;
9 return false
```

- ❖ Prostorová složitost: $O(|Q|) \subseteq \mathbf{NPSPACE} = \mathbf{PSPACE}$

- 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

```
1 counter := 0;
2 makroState := {s0};
3 while counter ≤ 2|Q| do
4   | if makroState ∩ F = ∅ then
5     |   return true
6   | a := nondeterministically choose from Σ;
7   | makroState := ∪q∈makroState δ(q, a);
8   | counter := counter + 1;
9 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 $\text{PUSH}(S, x)$, $\text{POP}(S)$ a $\text{MULTIPOP}(S, k)$ – odebere k prvků případně vyprázdní zásobník pokud $|S| \leq k$.
- ❖ Uvažujme libovolnou posloupnost n operací.
- ❖ Každá operace $\text{PUSH}(S, x)$ a $\text{POP}(S)$ má složitost 1. V posloupnosti n operací má $\text{MULTIPOP}(S, k)$ v nejhorším případě složitost $O(n)$.
- ❖ Naivní analýza časové složitosti (v nejhorším případě) pro n operací je $O(n^2)$.
- ❖ Existuje posloupnost operací, která má tuto složitost? Můžeme tuto analýzu zpřesnit?

Amortizovaná složitost – příklad

- ❖ Uvažme zásobník S a operace $\text{PUSH}(S, x)$, $\text{POP}(S)$ a $\text{MULTIPOP}(S, k)$ – odebere k prvků případně vyprázdí 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 $\text{PUSH}(S, x)$ má složitost n
- ❖ Skupina 2: Složitost posloupnosti operací $\text{POP}(S)$ a $\text{MULTIPOP}(S, k)$ je daná počtem prvků odebraných z S a tudíž nemůže být větší než počet provedených operací $\text{PUSH}(S, x)$. Složitost celé skupiny je tedy menší než n .
- ❖ Celková složitost libovolných n operací je menší než $2n$.

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.
- ❖ Cena n libovolných operací je pak $\leq n$ -krát počet kreditů za nejdražší operaci.

Amortizovaná složitost – metoda účtů

- ❖ 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.
- ❖ Cena n libovolných operací je pak $\leq n$ -krát počet kreditů za nejdražší operaci.
- ❖ 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 prvků v S . Tudíž skutečně platí, že počet kreditů na účtě je vždy **nezáporný**.
- ❖ Celková cena n operací je $\leq n \cdot \max\{2, 0\} = 2n$.

Amortizovaná složitost – ukázka

- ❖ Příklad: dynamicky alokovaná tabulka T (odebrání 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 $\alpha(T) < 0.5$ 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 (vlození, 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 $\alpha(T) < 0.25$ odebrání záznamu vede k alokaci 2-krát menšího pole a přesunu
- amortizovaná složitost libovolných n operací (seskupením, hlavní myšlenka)
 - $2^k + 1$ operací vlození, 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 vlození, 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 (vlození, 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 $\alpha(T) < 0.25$ 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 odebrání 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 $\leq n \cdot \max\{2, 3\} = 3n$

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.

- 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)* je implementována následujícím způsobem:
- *size* označuje počet prvků v seznamu.

```
1 Function prune(y)
2   while head ≠ null ∧ head → value < y do
3     tmp := head
4     head := head → next
5     print(tmp → value)
6     free(tmp)
```

Příklad ze zkoušky

```
1 Function prune(y)
2   while head ≠ null ∧ head → value < y do
3     tmp := head
4     head := head → next
5     print(tmp → value)
6     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

```
1 Function prune(y)
2   while head ≠ null ∧ head → value < y do
3     tmp := head
4     head := head → next
5     print(tmp → value)
6     free(tmp)
```

operace	cena	kredit
<i>insert</i> (<i>x</i>)	$O(size)$	$O(size) + 5$
<i>prune</i> (<i>x</i>)	$1 + 5 \cdot \text{počet odstraněných prvků}$	1

❖ Každá operace *insert*(*x*) vloží na účet 5 kreditů.

❖ Platí invariant: na účte 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).