NP-úplnost a další

Karel Richta a kol.

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Datové struktury a algoritmy, B6B36DSA
02/2021, Lekce 13

https://moodle.fel.cvut.cz/course/view.php?id=5973

Osnova: Co nám ještě chybí

- Návrh algoritmů zametací technikou.
- Návrh algoritmů technikou "prořezávej a hledej".
- Polynomiálně řešitelné problémy, NP-úplnost.
- Turingův stroj.

Zametací technika - úvod

- Je to postup převzatý z výpočetní geometrie, kdy postupujeme mezi objekty zleva doprava (zpravidla x souřadnice) a postupně zpracováváme místa (vyjádřená souřadnicí po které postupujeme), která nás zajímají.
  - Pro představu se používá idea svislé zametací přímky, která představuje onu hranici posouvající se zleva doprava.
- Základní charakterizující body jsou:
  - Suneme svislou přímku (scanline, SL, zametací přímku) zleva doprava nad množinou objektů.
  - Přímku posouváme na souřadnice, které nás zajímají (např. mezi body na grafu - nikoliv po x++).
  - Souřadnice, na které se chceme dostat, jsou v prioritní frontě (tu nazýváme postupový plán, x-struktura).
  - O již projitých místech/souřadnicích si ukládáme informace (nazývá se y-struktura).

Zametací technika

- Prostorem P, na němž probíhá řešení, proložíme zametací nadrovinu r (přímku v 2D, rovinu v 3D) a tuto nadrovinu posouváme mezi dvěma mezními polohami prostoru P.
- Spolu s nadrovinou udržujeme datovou strukturu S (y-struktura). Nadrovina rozděluje prostor řešení na „levý“ poloprostor, v němž již úloha byla vyřešena a „pravý“ poloprostor, v němž se řešení bude hledat.
- Struktura S obsahuje všechny informace o stavu řešení úlohy v levém podprostoru, které jsou relevantní pro řešení v pravém podprostoru. Stav struktury S se mění s polohou zametací nadroviny. Nadrovina se posouvá do pozic, kde se mění stavy řešení. Tyto pozice se nazývají postupový plán (x-struktura).

Zametací technika - pseudokód

Označme stav zametací přímky **y-struktura**. Zametací přímku budeme posouvat řešeným prostorem zleva doprava podle postupového plánu, který označíme **x-struktura**. Zametací techniku můžeme vyjádřit pseudokódem:

1: Inicializace x-struktury a y-struktury;

2: **while** x-struktura $\neq \emptyset$ **do**

begin p:=minimální prvek z x-struktury;

Posuň zametací přímku do p;

Vyjmi minimální prvek z x-struktury;

Uprav y-strukturu;

Generuj nové řešení odpovídající stavu y-struktury;

Uprav x-strukturu;

end;

Zametací technika - Příklad

- Hledám minimální body (takové, kde v levém dolním segmentu nejsou žádné body):
  1. Seřadím body dle x a uložím do x-struktury.
  2. Pamatuji si v y-struktuře jednu hodnotu - minimální (nejspodnější) y souřadnici. (Prvotní hodnota je nekonečno).
  3. Zametací přímku umisťuji do bodů x-struktury (tedy procházím zleva doprava body a v každém provedu aktualizaci y-struktury).
  4. Aktualizuji y-strukturu - pokud je aktuální souřadnice menší než hodnota v y-struktuře, nahradím.

Př.: Algoritmus pro Konvexní Obal

1. Setřídíme body podle x-ové souřadnice, označíme je $b_1,\ldots,b_n$.
2. Vložíme do horní a dolní obálky bod b_1: $H \leftarrow D \leftarrow (b_1)$.
3. Pro každý další bod $b = b_2,\ldots,b_n$:

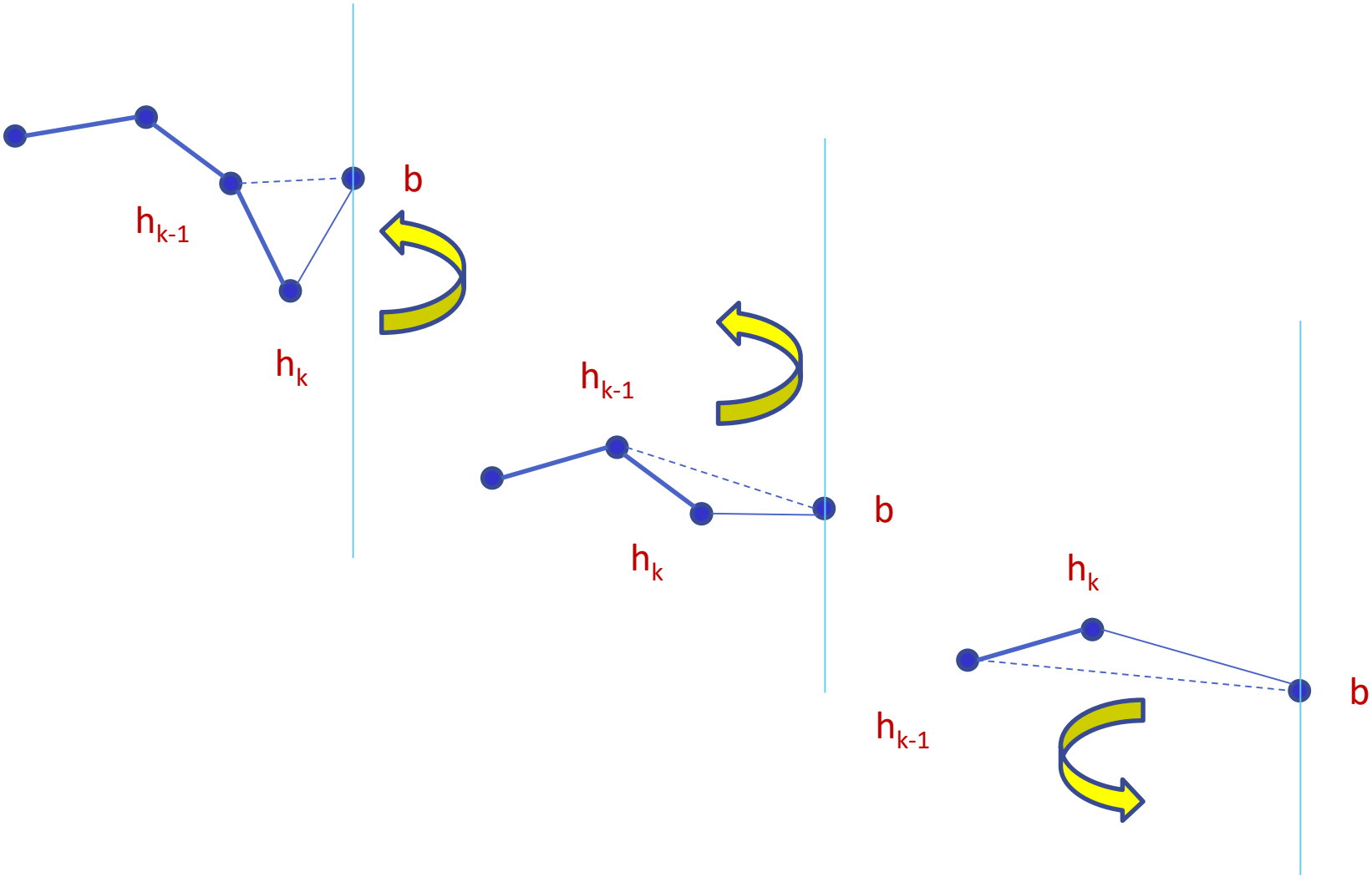
   Přepočítáme horní obálku:

   a) Dokud $|H| \geq 2$, $H = (\ldots,h_{k-1},h_k)$ a úhel $h_{k-1}h_kb$ je orientovaný doleva:
      Odebereme poslední bod h_k z obálky H.

   b) Přidáme bod b na konec obálky H.

   Symetricky přepočteme dolní obálku (s orientací doprava).
4. Výsledný obal je tvořen body v obálkách H a D.

Složitost: $\Theta(n)$


b
h_{k-1}
h_k
h_{k-1}
b
h_k
h_k
b
h_{k-1}

Technika Prořezávej a Hledej

- Tento postup optimalizace hledání je založen na postupném vyřazování části prohledávaných dat a tím redukováním složitosti hledání.
- Toto paradigma je velmi podobné algoritmům typu rozděl a panuj (divide and conquer), zásadní rozdíl je ovšem v tom, že při prořezávání neprocházíme všechny větve, ale pouze ty, které pro nás dávají smysl.
- V dynamickém programování se snažíme neopakovat výpočty, tady se snažíme je nedělat vůbec.

Příklad na Prořezávej a Hledej

- Hledáme i-té nejnižší číslo v neseřazeném seznamu čísel. Jednoduchý postup je seřadit seznam a vybrat i-tý prvek - tento postup však není nejefektivnější.
- Využijeme dělící funkce použité v algoritmu QUICKSORT, která v daném poli vybere pivota a prvky menší než pivot přehází do levé části pole a prvky větší do pravé části. Z pozice pivota v tomto poli můžeme určit, zda hledat i-té nejnižší číslo v levé nebo pravé části - v té rekurzivně opakujeme postup, druhou části se pak dále nemusíme zabývat.
- Příklad operuje nad globálním polem array a využívá následující funkci: **`int RSPLIT(l,r)`** - v části pole určené indexy **`l`** a **`r`** vybere pivota a prohází prvky pole tak, aby prvky menší než pivot byly nalevo a větší napravo. Návratová hodnota je index pivota.

Pseudokód pro Prořezávej a Hledej

```
int[] array; //
int RSELECT(int l, r, i) {
   // Vrátí index i-tého nejmenšího prvku
   int q = RSPLIT(l, r); // q je pivot
   int m = q - l + 1;    // m je počet prvků před pivotem
   if i < m then return RSELECT(l, q - 1, i);
        // Vlevo je vice prvků než i - tedy i-tý nejmenší musí ležet tam
   elsif i = m then return q;
        // Vlevo je právě i prvků - pivot je i-tý nejmenší
   else return RSELECT(q + 1, r, i - m);
        /* Vlevo je méně prvků než i - i-tý nejmenší musí ležet
           v pravé podčásti. i je zmenšeno o počet menších prvků,
           které jsme již našli (nalevo od pivota) */
   endif;
}
```

NP-problémy

- Téměř všechny dosud probírané algoritmy byly polynomiální v čase, tj. pro vstup rozsahu n, jejich časová složitost byla nejvýše $O(n^k)$ pro nějaké k.
- Můžeme se ptát, zda jsou všechny problémy řešitelné v polynomiálním čase.
- Odpověď zní ne. Např. existují problémy jako klasický Turingův "Halting Problem", které nemohou být vyřešeny žádným algoritmem, bez ohledu na čas.
- Také existují problémy, které jsou algoritmicky řešitelné, ale ne v polynomiálním čase.
- Obecně problémy, které nejsou řešitelné v polynomiálním čase nazýváme „těžké (NP-hard)" – jsou řešitelné v superpolynomiálním čase.

NP-úplnost

- Existuje zajímavá třída problémů, nazývaných „NP-úplné (NP-complete)”, jejichž statut není zatím známý. Nebyl zatím objeven žádný polynomiální algoritmus, který by je řešil, ale nepodařilo se dokázat, že by takový algoritmus nemohl existovat.
- Tento problém je označován jako otázka: **P ≠ NP** ?
- Tato otázka je stále jednou z největších výzev teoretické informatiky od roku 1971, kdy byla poprvé formulována.
- Vztah mezi P a NP je jedním ze sedmi problémů tisíciletí, které vypsal Clayův matematický ústav 24. května 2000, za rozhodnutí vztahu nabízí 1 000 000 dolarů.
- Přitom některé NP-úplné problémy se liší jen velmi jemně od problémů řešitelných polynomiálně.

Drobný krok od P k NP

Problém nejkratší versus nejdelší jednoduché cesty:

- Nejkratší cestu v orientovaném grafu G = (V,E) lze nalézt v čase $O(|V|\cdot|E|)$ – Bellman-Ford, resp. $O(|V|\cdot\log_2|E|)$ – Dijkstra.
- Nalezení nejdelší jednoduché cesty mezi dvěma uzly je velmi obtížné.
- Dokonce i pouhé určení, zda graf obsahuje jednoduchou cestu s alespoň určitým počtem hran, je NP problém (dokonce NP-úplný).

Eulerova cesta versus Hamiltonovský cyklus:

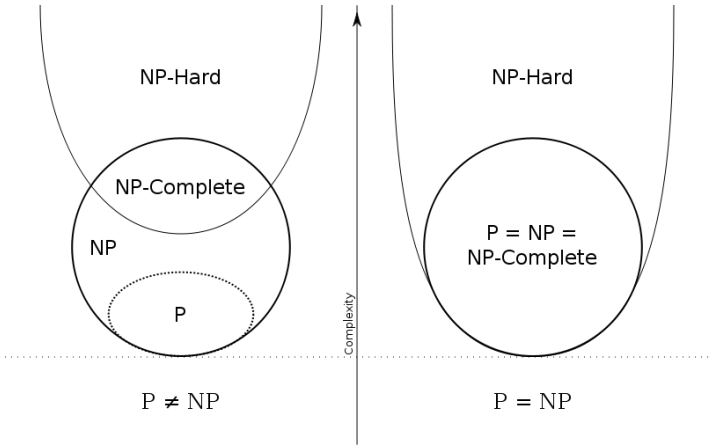
- Eulerova prohlídka souvislého, orientovaného grafu G = (V,E) je cyklus, který prochází každou hranou z E právě jednou, vrcholy je dovoleno navštívit více než jednou. Zjistit, zda existuje Eulerova cesta a která posloupnost hran ji tvoří, lze v čase $O(|E|)$.
- Hamiltonovský cyklus v orientovaném grafu je jednoduchý cyklus, který obsahuje každý vrchol z V (projde všemi uzly právě jednou). Určení, zda orientovaný graf má hamiltonovský cyklus je NP problém (dokonce NP-úplný).

NP-úplnost a třídy P a NP

- Definujeme tři třídy problémů: P, NP, a NPC (NP-úplné problémy, NP-Complete problems).
- Třída P sestává z problémů, které jsou řešitelné v polynomiálním čase, tj. problémy, které lze vyřešit s časovou složitostí $O(n^k)$ pro nějakou konstantu k.
- Třída NP sestává z problémů, které jsou "ověřitelné" v polynomiálním čase. Ověřitelností (verifiability) se myslí fakt, že pokud získáme nějaké potenciální „řešení“, pak umíme ověřit, že je správné. Problémy, kde toto ověření umíme v polynomiálním čase, patří do NP (nedeterministicky polynomiální).
- Např. pro Hamiltonovský cyklus: mějme graf G=(V,E) a jistou posloupnost uzlů $<v_1,v_2,v_3, \ldots ,v_{|V|}>$, pak můžeme v polynomiálním čase ověřit, že $v_i,v_{i+1} \in E$ a také $v_{|V|},v_1 \in E$.

NP-úplnost a třídy P a NP (pokr.)

- Každý problém z P je také v NP, protože v případě, že problém je v P, pak jej můžeme řešit v polynomiálním čase, aniž bychom předem znali řešení. Můžeme věřit, že **P** $\subseteq$ **NP**.
- Otevřenou otázkou je, zda P je vlastní podmnožinou NP.
- Problém patří do třídy NPC **(NP-úplný, NP-Complete)**, pokud je v NP a je stejně těžký (hard), jako všechny ostatní problémy v NP (je NP-hard).
- Důsledek: Pokud by byl některý NP-úplný problém řešitelný v polynomiálním čase, pak existuje polynomiální algoritmus pro každý problém z NP.



Autor: Behnam Esfahbod, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=3532181

NP-úplnost a třídy P a NP (pokr.)

- Většina odborníků se domnívá, že NP-úplné problémy jsou polynomiálně neřešitelné, protože s ohledem na širokou škálu NP-úplných problémů, které byly studovány, zatím nikdo neobjevil polynomiální řešení některého z nich.
- Bylo by vskutku ohromující, pokud všechny z nich byly řešitelné v polynomiálním čase. Přes velké úsilí věnované prokázání, že NP-úplné problémy jsou neřešitelné, nejsou zatím průkazné výsledky ani v tomto směru.
- Nelze tedy vyloučit možnost, že NP-úplné problémy jsou ve skutečnosti řešitelné v polynomiálním čase.

Důkaz NP-úplnosti

- Pro důkaz, že daný problém je NP-úplný, lze použít jakoukoliv techniku, která ověří, že je převeditelný na nějaký existující NP-úplný problém.
- Je třeba uvážit rozdíl mezi rozhodovacími a optimalizačními problémy. NP-úplnost se přímo vztahuje na rozhodovací problémy, tj. na algoritmy, kde výsledkem je „ano“ nebo „ne“. Optimalizační problémy je nutno upravit tak, aby se převedly na problémy rozhodovací.
- Poté se zabýváme redukcí problému na existující NP-úplný problém.
- Ukážeme jeden existující NP-úplný problém (na něj pak můžeme převést všechny ostatní).

Redukce

- Definice NP-úplnosti ukazuje, že důkaz NP-úplnosti lze provést redukcí problému na jiný NP-úplný problém pomocí polynomiálního algoritmu.
- Předpokládejme, že máme algoritmus, který transformuje jakoukoliv instanci α problému A do nějaké instance β problému B a má následující charakteristiky:
  - Transformace proběhne v polynomiálním čase.
  - Odpovědi jsou po transformaci stejné. To znamená, že odpověď pro α je "ano" tehdy a jen tehdy, pokud odpověď pro β je také "ano."
  - Problém B je NP-úplný.
- Pak i problém A je NP-úplný.
- Poznámka: Pokud je problém B z P, pak i A je z P.

První NP-úplný problém

- Vzhledem k tomu, že třída NPC je definována pomocí stejné obtížnosti problému jako má jiný NP-úplný problém, je třeba definovat "První" NP-úplný problém. Tento problém budeme používat pro ustavení stejné obtížnosti jako zkoumaný problém.
- „První“ NP-úplný problém je **problém splnitelnosti**, ve kterém budeme mít logický kombinační obvod složený z bran realizujících logické operace AND, OR a NOT, který má n vstupů a jeden výstup. Chceme vědět, zda existuje nějaký vektor logických dvouhodnotových (Boolean) vstupů do tohoto obvodu, který způsobí, že na jeho výstupu bude hodnota 1.
- Bývá také nazývám **problém splnitelnosti Booleovské formule**.
- Složitost tohoto problému je $O(2^n)$, je tedy NP.

Příklady NP-úplných problémů

- Problém splnitelnosti Booleovské formule.
- Problém dvou loupežníků.
- Problém obchodního cestujícího.
- Problém batohu.
- Problém tříbarevnosti grafu.
- . . .

Turingův stroj

- Teoretický model počítače popsaný matematikem Alanem Turingem. Obdoba RAM počítače.
- Skládá se z procesorové jednotky, tvořené konečným automatem, programu ve tvaru pravidel přechodové funkce a nekonečné pásky pro zápis mezivýsledků. Využívá se pro modelování algoritmů v teorii vyčíslitelnosti.
- Church-Turingova teze říká, že ke každému algoritmu existuje ekvivalentní Turingův stroj.
- Od výpočetní síly Turingova stroje se odvozuje turingovská úplnost: turingovsky úplné jsou právě ty programovací jazyky nebo počítače, které mají stejnou výpočetní sílu jako Turingův stroj.

Turingův stroj - definice

- Turingův stroj je sedmice TS = (Q , Γ , b , Σ , s , δ , F), kde:
- Q je konečná množina vnitřních stavů,
- Γ je konečná abeceda symbolů na pásce,
- b ∈ Γ je symbol reprezentující prázdný symbol (b není součástí vstupní abecedy přijímaného řetězce),
- Σ ⊆ Γ \ b je konečná množina vstupních symbolů,
- s ∈ Q je počáteční stav,
- δ : Q × Γ → Q × Γ × { L , R, ε } je přechodová funkce, kde:
  - L znamená posun hlavy vlevo, R znamená posun hlavy vpravo, ε znamená bez posuvu,
- F ⊆ Q je množina koncových stavů.

Konfigurace Turingova stroje

- **Konfigurace** Turingova stroje je trojice:

  $\langle q , s \rangle \in Q \times (\Gamma \cup \{\#\})^*$,

  kde *q* je aktuální stav, *s* je nejmenší souvislá část pásky obsahující všechny neprázdné symboly a znakem # je vyznačena pozice čtecí hlavy.

  Pokud například páska obsahuje 1234, stroj je ve stavu q a čtecí hlava stojí na symbolu 2, zapíše se tato konfigurace jako:

  $\langle q , 1\#234 \rangle$.

- **Počáteční konfigurace** Turingova stroje pro vstup $w \in \Gamma^*$ je:

  $\langle q , \#w \rangle$.

Výpočet Turingova stroje

Na začátku výpočtu je Turingův stroj v počáteční konfiguraci a na pásce je zapsané vstupní slovo. Dále pracuje po jednotlivých krocích:

- pokud je aktuální stav zároveň stavem koncovým, výpočet končí, jinak
- čtecí hlava přečte jeden vstupní symbol z buňky, na které se právě nachází a pokud je v přechodové funkci pro aktuální stav a pro přečtený symbol definovaný přechod:
  - změní se stav stroje,
  - na aktuální pozici hlavy se zapíše příslušný symbol,
  - hlava se příslušným způsobem posune, či neposune.

Příklad Turingova stroje

- Popišme jednoduchý TS, který bude mít za úkol zjistit, zda slovo napsané na pásku má tvar 0^n1^n, $n > 0$ (zda řetězec obsahuje n nul a následně n jedniček).
- Základní myšlenkou algoritmu je, že náš Turingův stroj vždy vymaže počáteční 0, přesune hlavu na konec slova, vymaže z něj poslední 1 a přesune se zpět na začátek slova. Tuto činnost opakuje, dokud slovo nevymaže nebo se nedostane do stavu, kdy další krok není definován (v tom případě slovo do jazyka nepatří).
- Formální zápis: $Q = \{ q_0 , q_1 , q_2 , q_3 , q_4 , q_5 , q_6 , q_7 \}$,

  $\Gamma = \{0, 1, b\}$, počáteční stav je q_0 a $F = \{q_7\}$.

Příklad Turingova stroje (pokr.)

Přechodová funkce:

1. $\delta(q_0, 0) = (q_1, b, R)$ - vymažeme počáteční nulu a zahájíme přesun čtecí hlavy doprava,
2. $\delta(q_1, 0) = (q_1, 0, R)$ - přesouváme hlavu doprava přes nuly, pásku neměníme,
3. $\delta(q_1, 1) = (q_2, 1, R)$ - narazili jsme na první jedničku, změníme stav a pokračujeme v pohybu doprava,
4. $\delta(q_2, 1) = (q_2, 1, R)$ - přesouváme hlavu doprava přes jedničky,
5. $\delta(q_2, b) = (q_3, b, L)$ - ocitli jsme se za slovem, vrátíme se před poslední symbol,
6. $\delta(q_3, 1) = (q_4, b, L)$ - vymažeme poslední jedničku a přesuneme se na předchozí znak,
7. $\delta(q_4, b) = (q_7, b, R)$ - pokud jsme v kroku 6 vymazali poslední symbol slova, algoritmus končí přechodem do koncového stavu,
8. $\delta(q_4, 1) = (q_5, 1, L)$ - jinak zahájíme přesun doleva,

Příklad Turingova stroje (pokr.)

Přechodová funkce (pokr.):

9. $\delta(q_5, 1) = (q_5, 1, L)$ - přesouváme hlavu doleva přes jedničky,
10. $\delta(q_5, 0) = (q_6, 0, L)$ - narazili jsme na poslední nulu, změníme stav a pokračujeme v pohybu doleva,
11. $\delta(q_6, 0) = (q_6, 0, L)$ - přesouváme hlavu doleva přes nuly,
12. $\delta(q_6, b) = (q_0, b, R)$ - dostali jsme se před slovo, vrátíme se na jeho první znak a začínáme zase od začátku, slovo je nyní o dva znaky kratší.

Ukázka výpočtu

Ukázka výpočtu výše uvedeného Turingova stroje při vstupu „0011“. Zapíšeme ji jako posloupnost po sobě následujících konfigurací. Nad šipkou představující jeden krok výpočtu je vždy uvedeno číslo použitého pravidla:

$< q_0, \#0011> \rightarrow^{1} < q_1, \#011> \rightarrow^{2} < q_1, 0\#11> \rightarrow^{3} < q_2, 01\#1> \rightarrow^{4}$
$< q_2, 011\#> \rightarrow^{5} < q_3, 01\#1> \rightarrow^{6} < q_4, 0\#1> \rightarrow^{8} < q_5, \#01> \rightarrow^{9}$
$< q_6, \#01> \rightarrow^{10} < q_6, \#01> \rightarrow^{1} < q_1, \#1> \rightarrow^{2} < q_2, 1\#> \rightarrow^{5}$
$< q_3, \#1> \rightarrow^{6} < q_5, \#> \rightarrow^{7} < q_7, \#>$

Turingův stroj dospěl do koncového stavu, výpočet končí úspěšně.

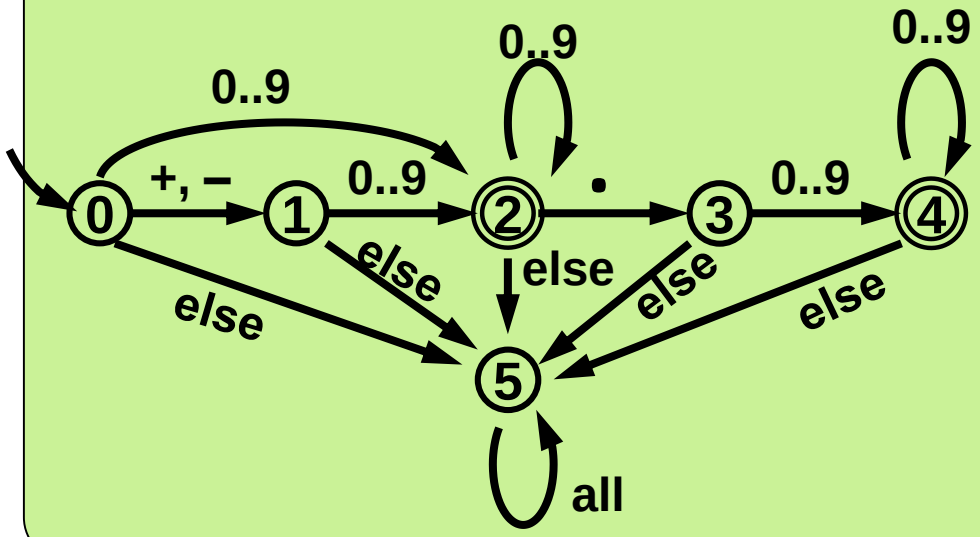
Nedeterministický Turingovův stroj

- Změna spočívá v tom, že čtecí hlava přečte jeden vstupní symbol z pásky v místě, kde se právě nachází a pokud je v přechodové funkci pro aktuální stav a pro přečtený symbol definováno více možných přechodů, vybere se jeden náhodně a podle něj se:
  - změní stav,
  - na aktuální pozici hlavy se zapíše příslušný symbol,
  - hlava se příslušným způsobem posune (či neposune).

Další automaty

- Turingovu pásku můžeme nahradit seznamem – ukazovátko v seznamu zastupuje polohu čtecí hlavy, posun hlavy simulujeme posunem ukazovátka.
- Seznam se dá nahradit dvěma zásobníky – v jednom je obsah pásky před čtecí hlavou, ve druhém obsah za čtecí hlavou. Pohyb čtecí hlavy se simuluje přesunem z jednoho zásobníku na druhý.
- Pokud nahradíme Turingovu pásku omezenou RAM pamětí, dostaneme RAM počítač.
- Pokud nahradíme Turingovu pásku zásobníkem, dostaneme zásobníkový automat.
- Pokud Turingovu pásku zcela odstraníme, dostaneme konečný automat. Ten si pamatuje svou minulost pouze pomocí aktuálního stavu, nemá žádnou vnější paměť.

Konečný automat (finite-state automaton)



	0	1	2	…	9	•	+	–	
0	2	2	2	…	2	5	1	1	
1	2	2	2	…	2	5	5	5	
2	2	2	2	…	2	3	5	5	F
3	4	4	4	…	4	5	5	5	
4	4	4	4	…	4	5	5	5	F
5	5	5	5	…	5	5	5	5	

Q: konečná množina stavů ⓪①②③④⑤

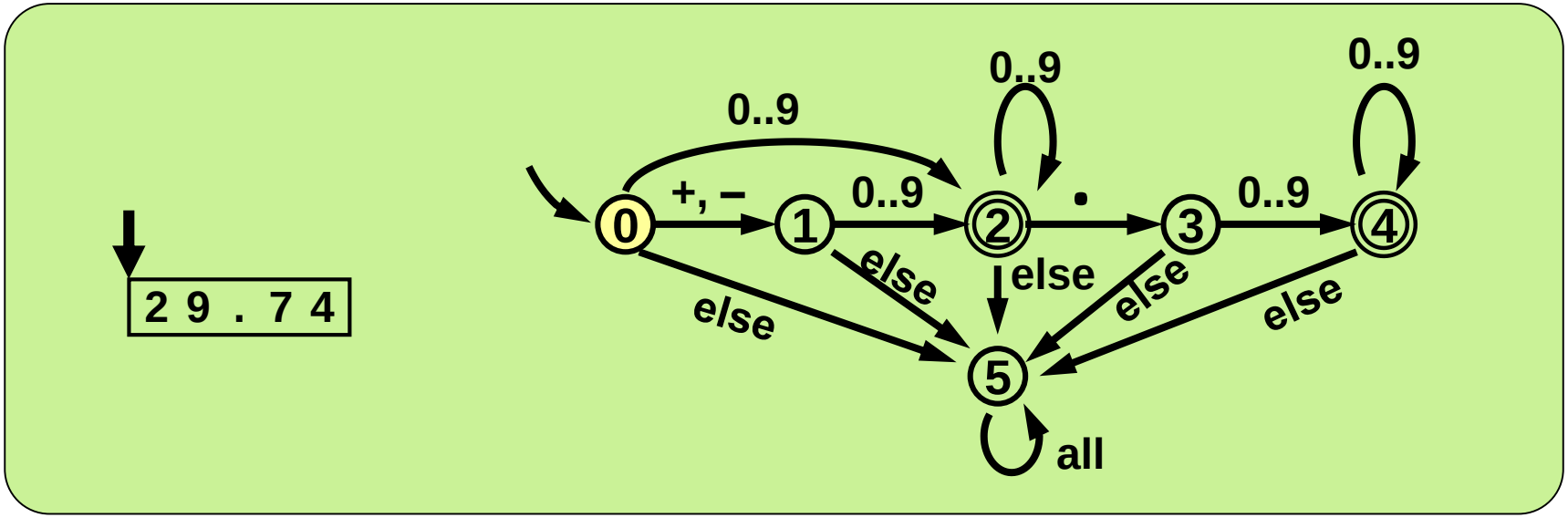
Σ: konečná abeceda {0,1,2,...,9, ,+, –}

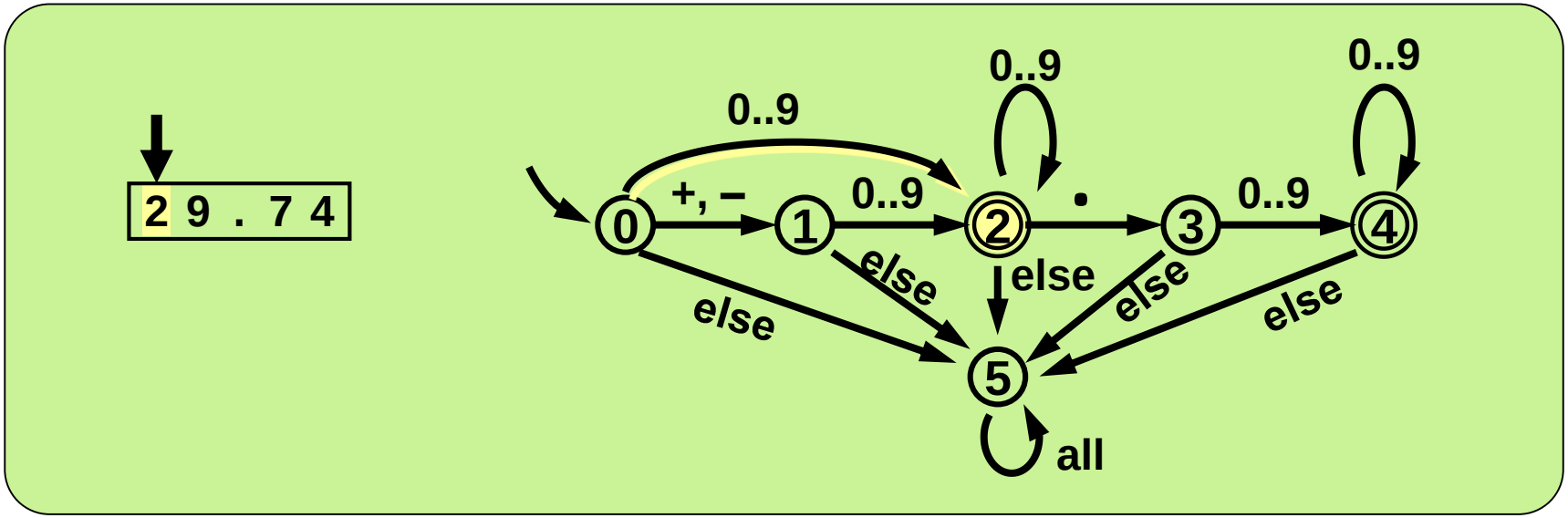
δ: zobrazení $Q \times \Sigma \rightarrow Q$, viz ohodnocení hran nebo tabulka

q_0: počáteční stav ⓪

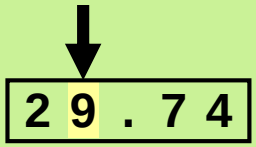
F: množina koncových (akceptujících) stavů ②④

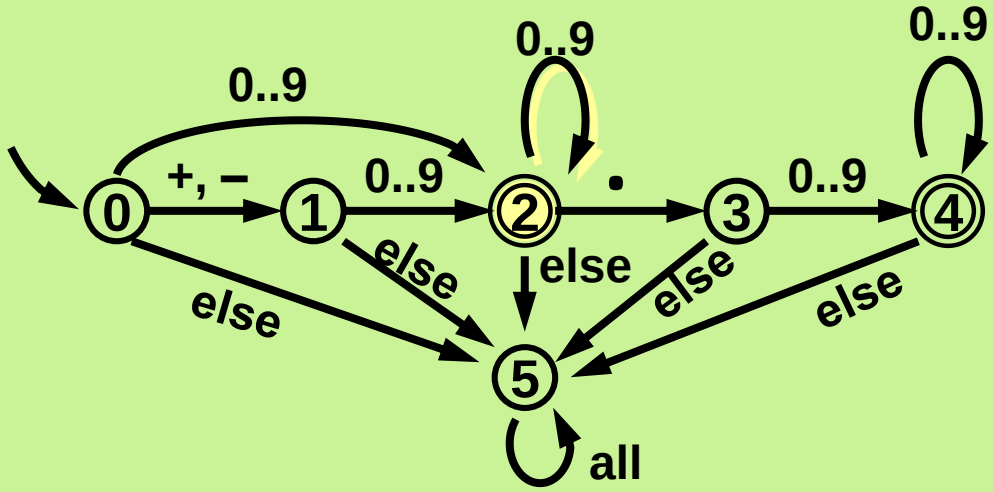
Činnost konečného automatu

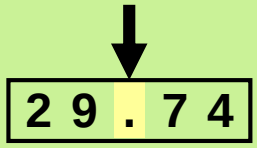


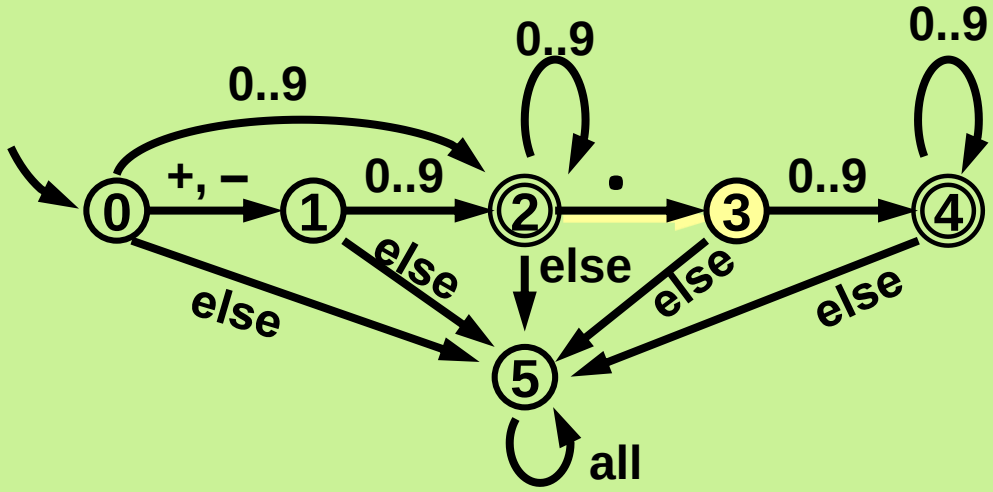


Činnost konečného automatu

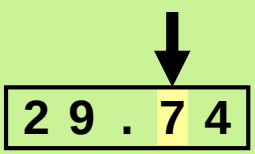


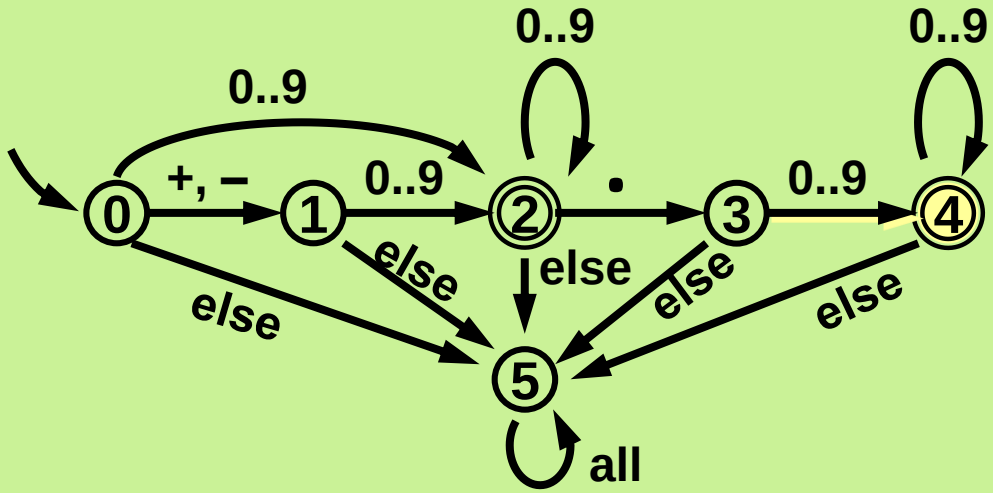


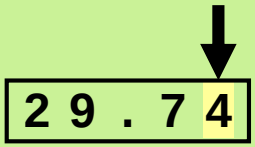


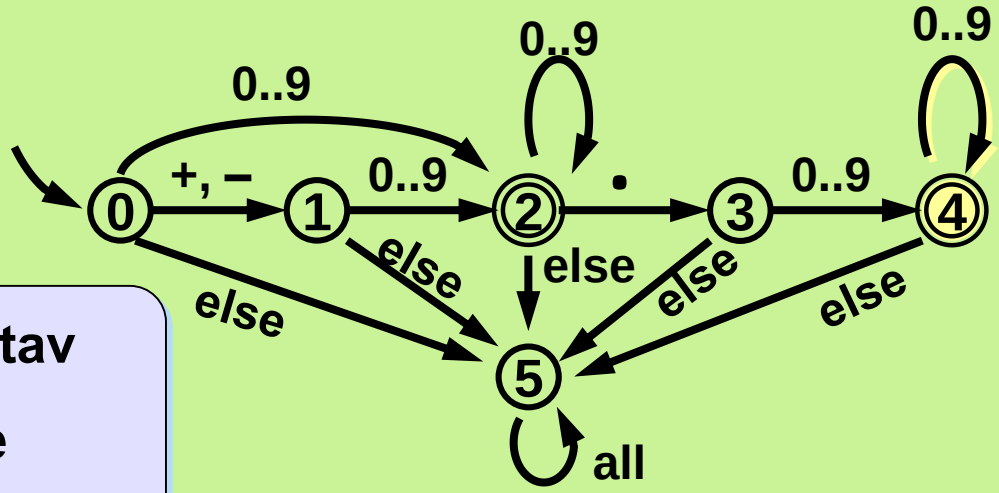


Činnost konečného automatu





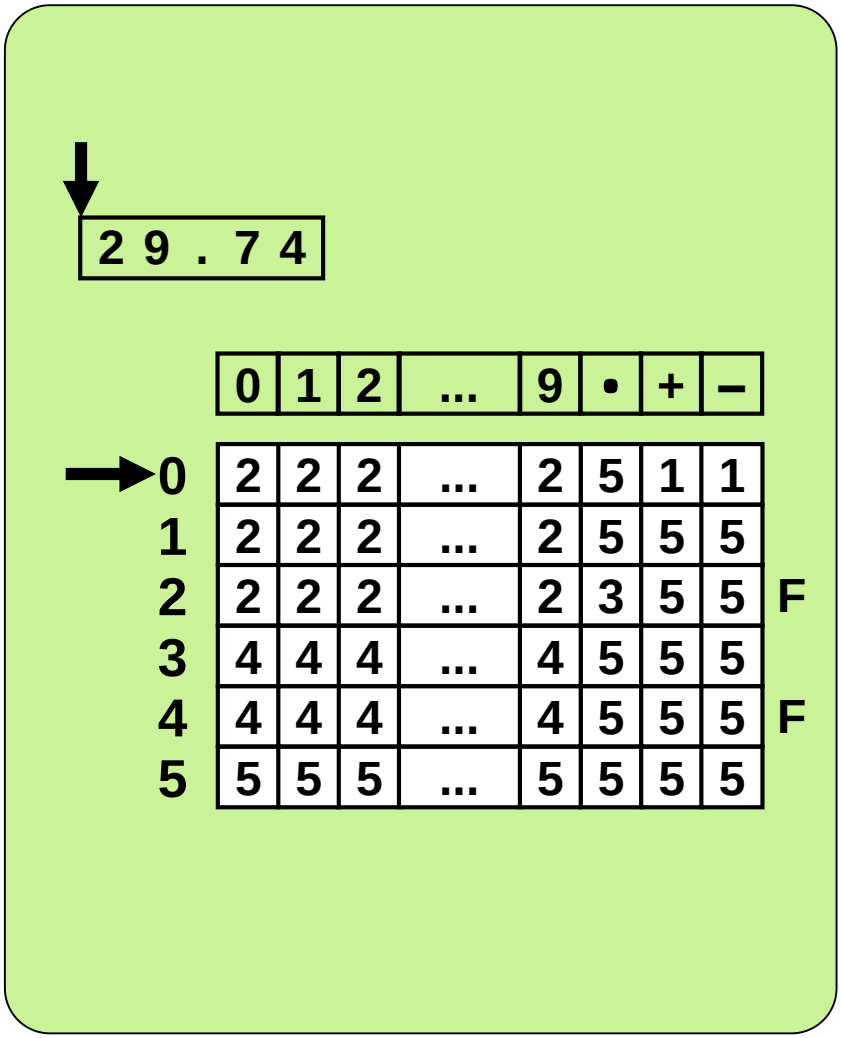


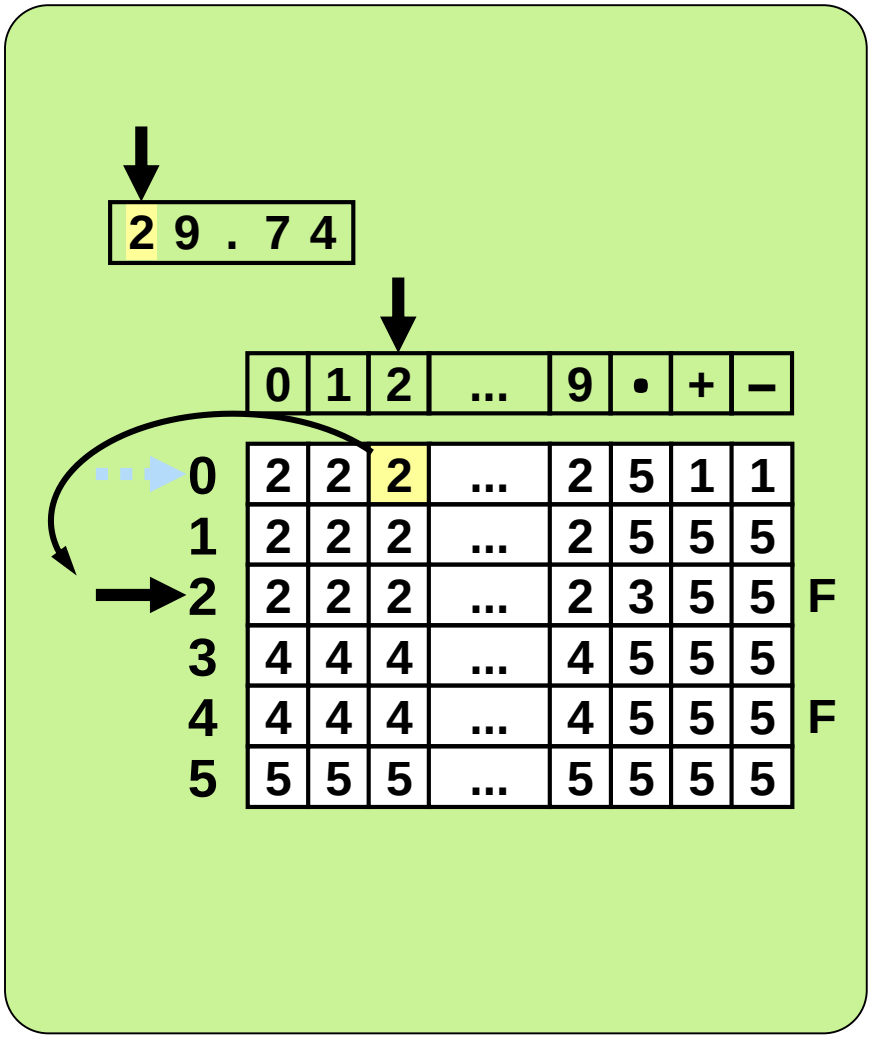


④ je koncový stav

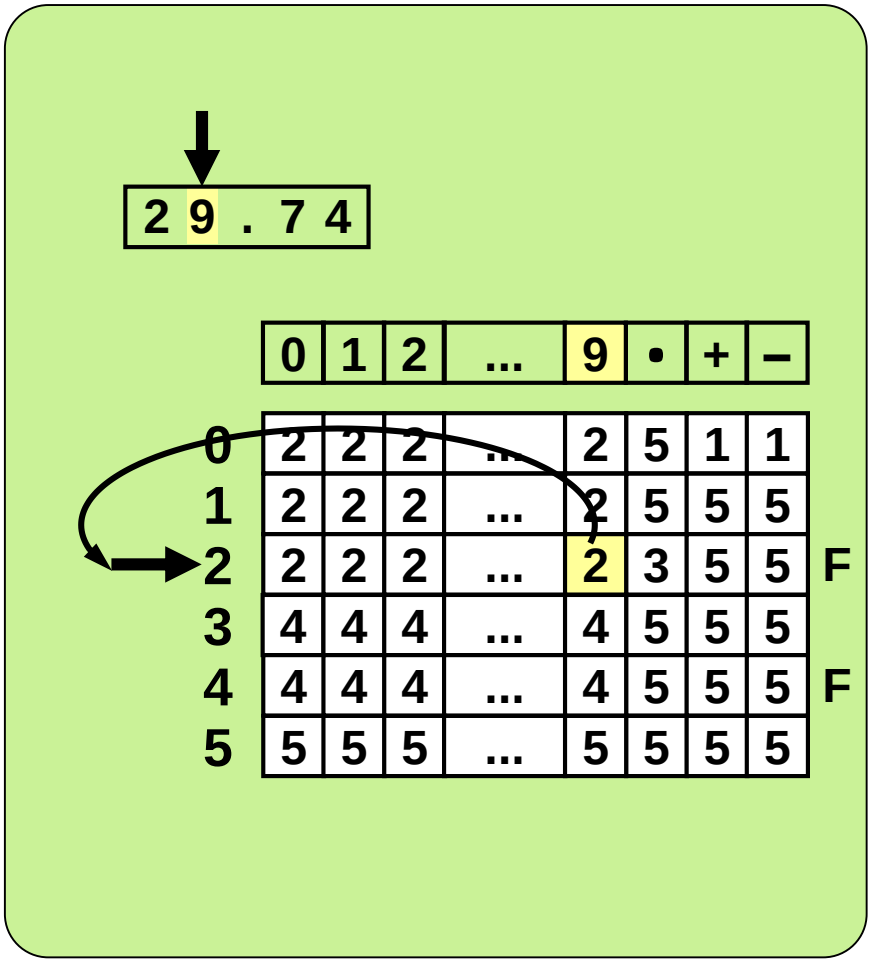
— 29.74 je akceptováno

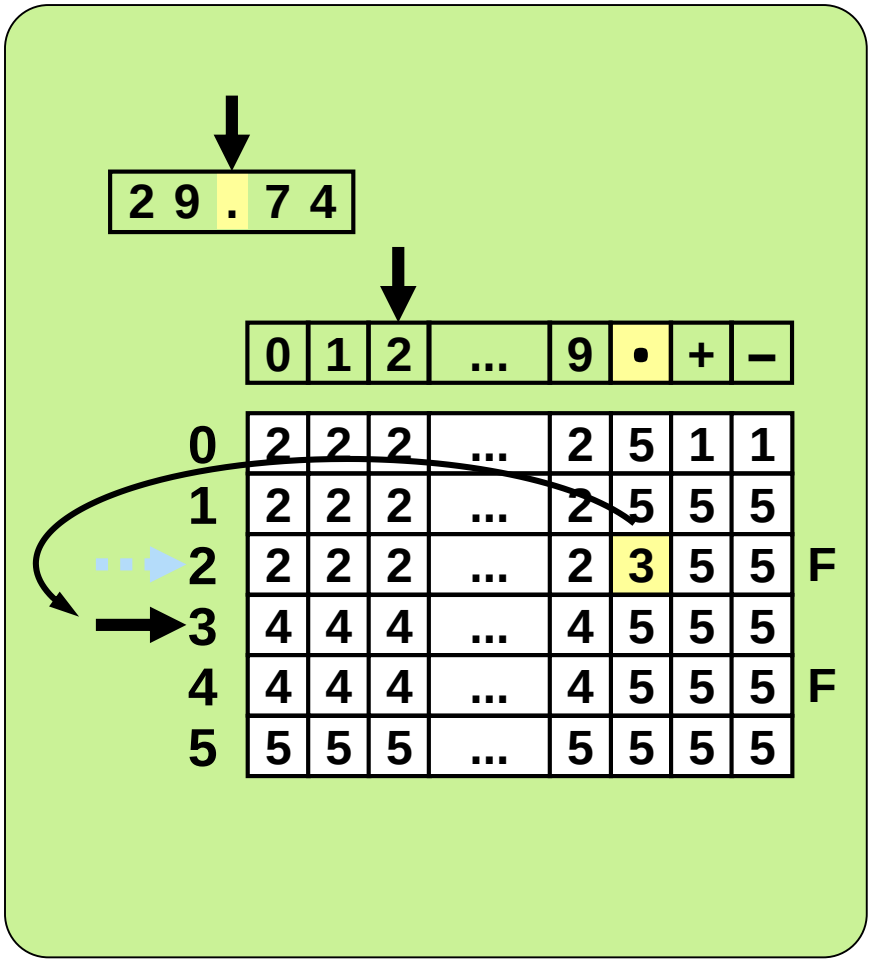
Činnost konečného automatu



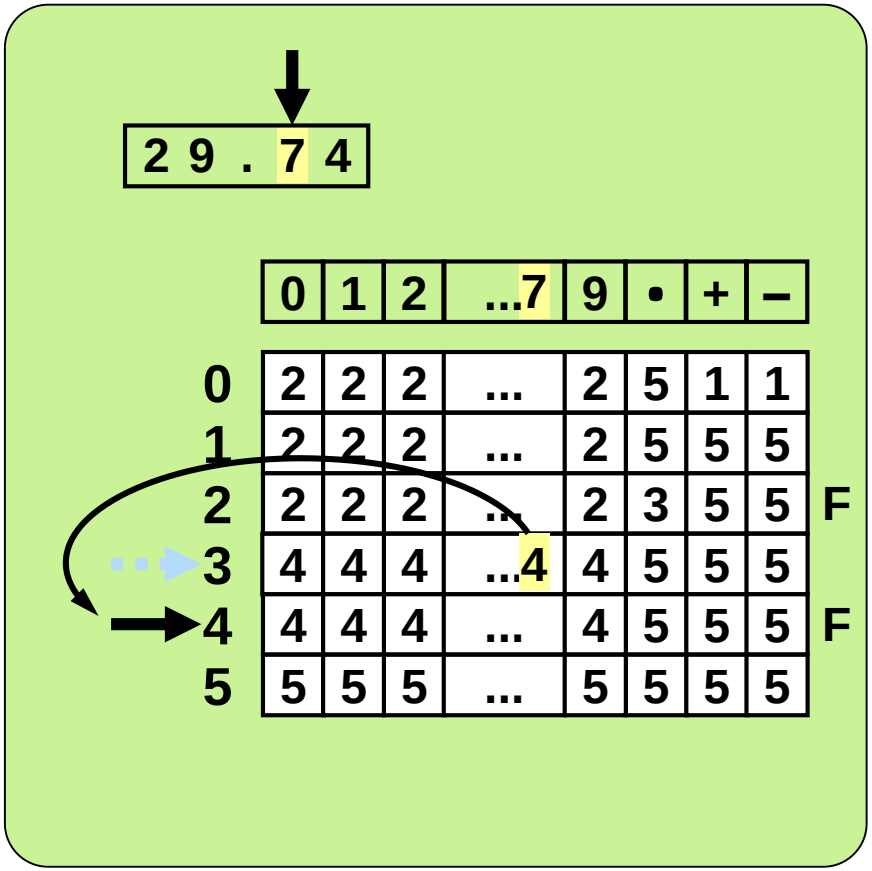


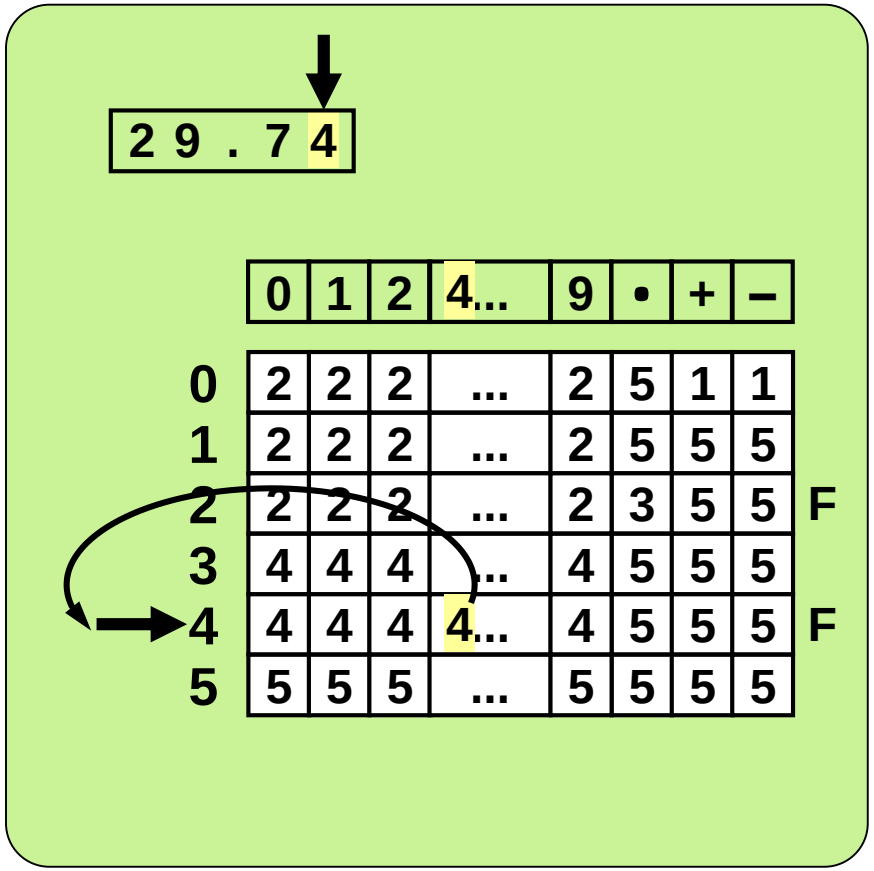
Činnost konečného automatu



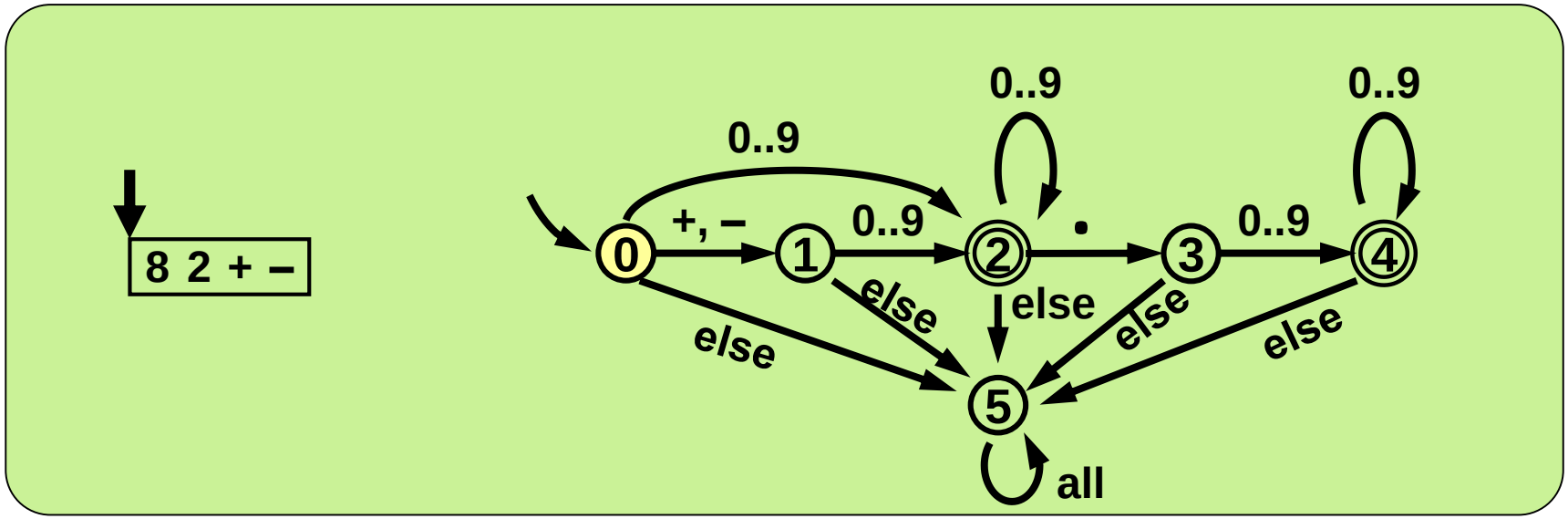


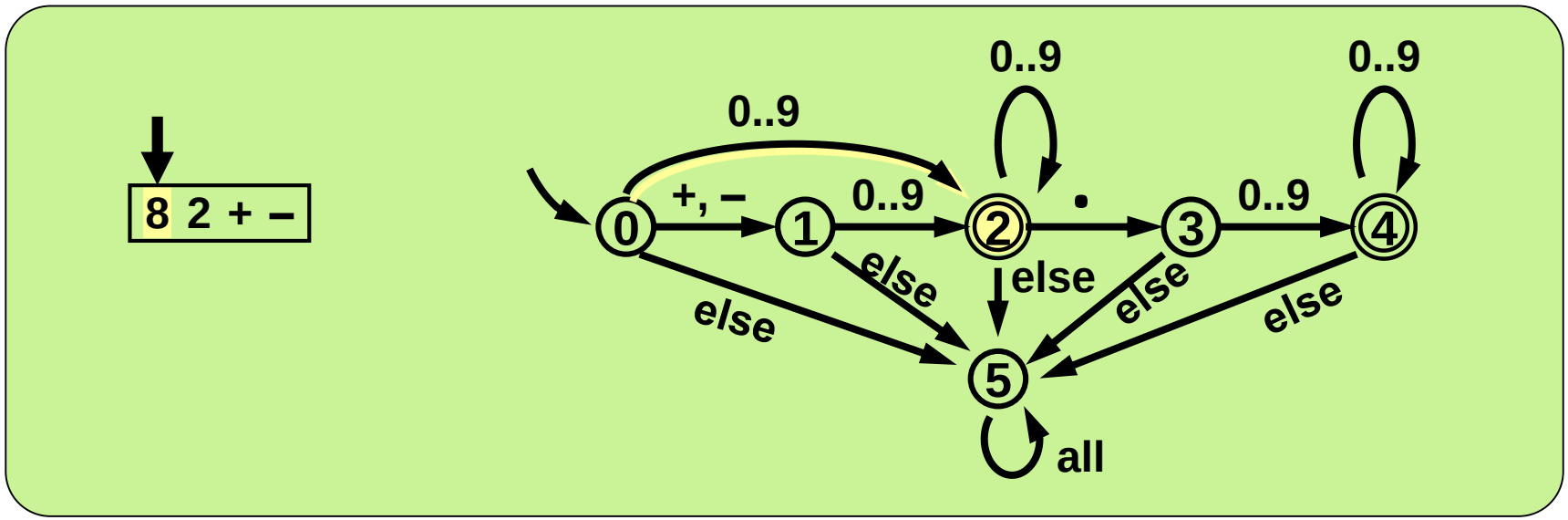
Činnost konečného automatu



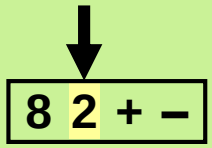


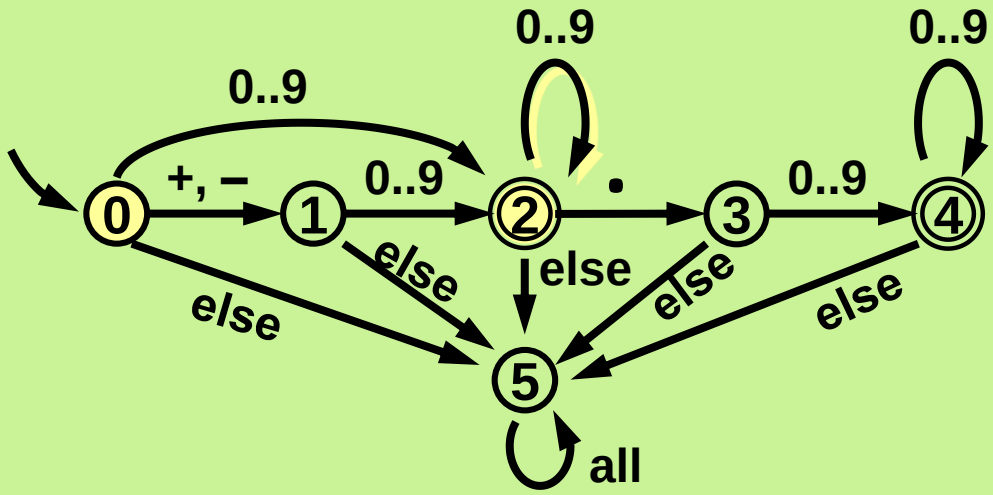
Jiný příklad činnosti konečného automatu

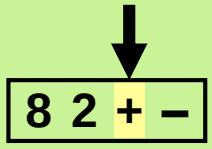


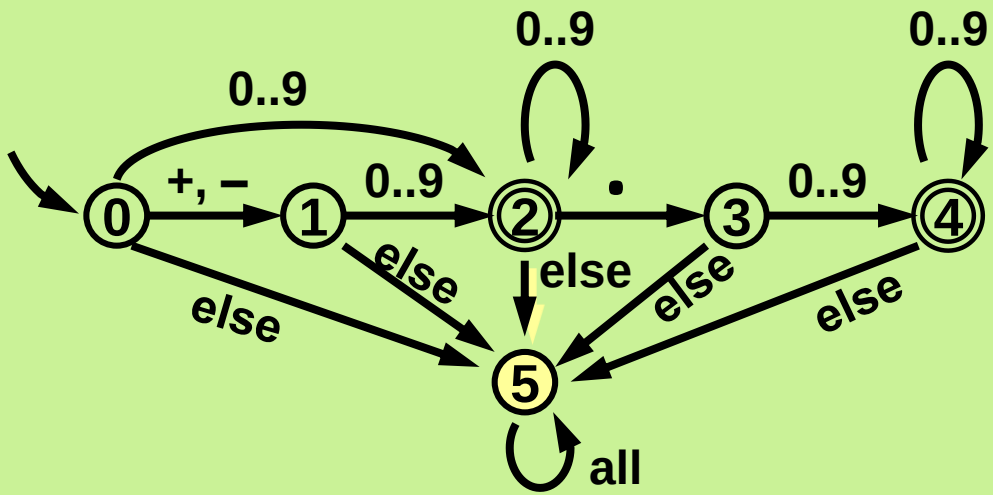


Jiný příklad činnosti konečného automatu

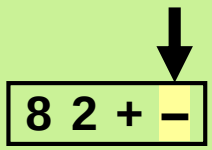


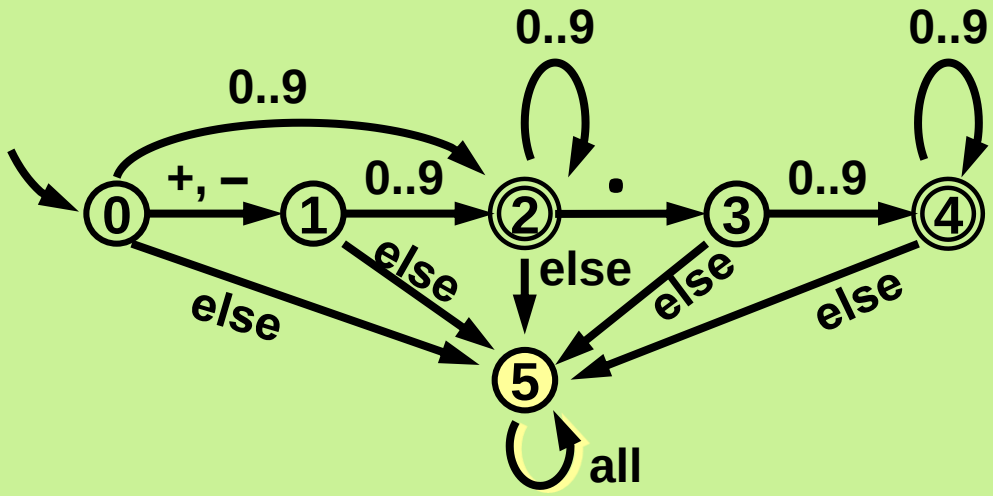






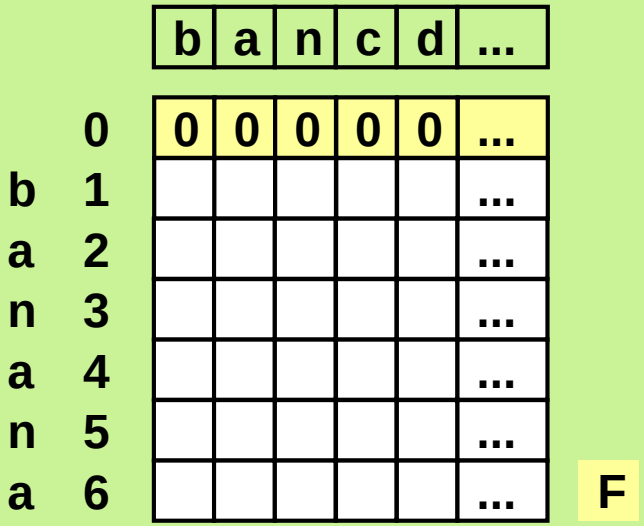
Jiný příklad činnosti konečného automatu





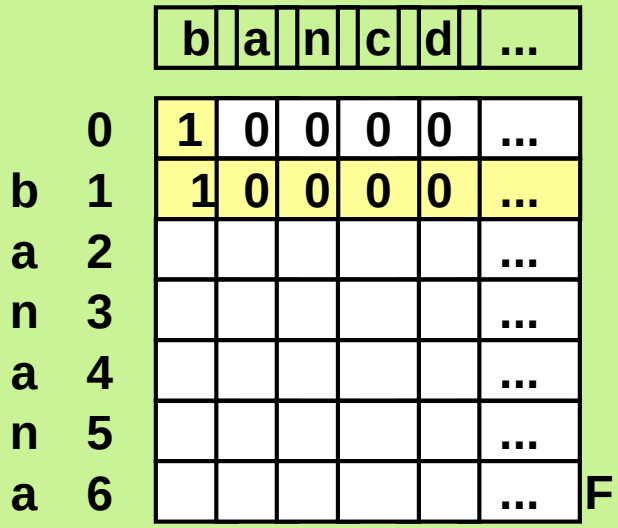
⑤ není koncový stav — “82+–” není akceptováno

Př.: Konstrukce KA pro hledání vzorku textu



1: $M \leftarrow (\{q_0, q_1, \ldots, q_m\}, \Sigma, \delta, q_0, \{q_m\})$
2: **for** $\forall a \in \Sigma$ **do**
3: $\delta(q_0, a) \leftarrow \{q_0\}$ {self-loop of the initial state}
4: **end for**

Konstrukce KA pro hledání vzorku textu




```
5: for i ← 1..m do
6:     r ← δ(q_{i−1}, p_i)
7:     δ(q_{i−1}, p_i) ← q_i {forward transition}
8:     for ∀a ∈ Σ do
9:         δ(q_i, a) ← δ(r, a)
10:    end for
11: end for
```

Konstrukce KA pro hledání vzorku textu

		b	a	n	c	d	...	
	0	1	0	0	0	0	...	
b	1	1	2	0	0	0	...	
a	2	1	0	0	0	0	...	
n	3						...	
a	4						...	
n	5						...	
a	6						...	F

i = 2

r = δ(1,a) = 0

```
 5: for i ← 1..m do
 6:     r ← δ(q_{i−1}, p_i)
 7:     δ(q_{i−1}, p_i) ← q_i {forward transition}
 8:     for ∀a ∈ Σ do
 9:         δ(q_i, a) ← δ(r, a)
10:     end for
11: end for
```

Konstrukce KA pro hledání vzorku textu

		b	a	n	c	d	...	
	0	1	0	0	0	0	...	
b	1	1	2	0	0	0	...	
a	2	1	0	3	0	0	...	
n	3	1	0	0	0	0	...	
a	4						...	
n	5						...	
a	6						...	F

i = 3

r = δ(2,n) = 0

```
 5: for i ← 1..m do
 6:     r ← δ(q_{i−1}, p_i)
 7:     δ(q_{i−1}, p_i) ← q_i {forward transition}
 8:     for ∀a ∈ Σ do
 9:         δ(q_i, a) ← δ(r, a)
10:     end for
11: end for
```

Konstrukce KA pro hledání vzorku textu

		b	a	n	c	d	...
	0	1	0	0	0	0	...
b	1	1	2	0	0	0	...
a	2	1	0	3	0	0	...
n	3	1	4	0	0	0	...
a	4	1	0	5	0	0	...
n	5	1	6	0	0	0	...
a	6	1	0	0	0	0	...

i = 6

F r = δ(5,a) = 0

```
 5: for i ← 1..m do
 6:     r ← δ(q_{i−1}, p_i)
 7:     δ(q_{i−1}, p_i) ← q_i {forward transition}
 8:     for ∀a ∈ Σ do
 9:         δ(q_i, a) ← δ(r, a)
10:     end for
11: end for
```

The End