EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Introduction to the Theory of Computation

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA

Introduction to the Theory of Computation

Lesson 1

2.5. State minimization

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Strukturális és Beruházási Alapok

BEFEKTETÉS A JÖVŐBE

Introduction

- Lecturer: Dr. István Heckl, Istvan.Heckl@gmail.com
- http://oktatas.mik.uni-pannon.hu
  - registration
    - full name
    - neptun code
  - course: A számítástudomány alapjai (Heckl)
    - enlisting code: kincs
    - see: download materials
  - course: Theory of the elements of computation (Heckl)

Subject information

- Subject code:
  - for full time student: VEMISAB512S
  - for part time student: VEMLSAB512S
- Signature: at least 50% result at ZH
- Subject name in Hungarian: A számítástudomány alapjai
- Literature: Harry R. Lewis, Christos H. Papadimitriou: Elements of the Theory of Computation, Prentice Hall, Inc., 1998. (second edition)
  - this presentation is based on this book
  - Bach Iván: Formális nyelvek
- Precondition: A digitális számítás elmélete

The semester

- Finite automaton:
  - state minimization
  - algorithmic aspects of finite automata
- Context-free languages: Parse trees
- Algorithms for context-free grammars, Determinism and parsing

The semester

- Extensions of Turing machines
- Random access Turing machines, Nondeterministic Turing machines
- Grammars
- Numerical functions

The semester

- Undecidability
  - unsolvable problems about Turing machines
  - unsolvable problems about grammars
  - an unsolvable tiling problem
  - properties of recursive languages
- Computational complexity: The class P, Problems from P

The semester

- Boolean satisfiability, The class NP
- Polynomial-time reductions
- Cook’s Theorem
- More NP-complete problems
- Coping with NP-completeness

Must be known

- Definition of:
  - deterministic finite automaton, M
  - state diagram of a DFA
  - configuration of a DFA $M=(K, \Sigma, \delta, s, F)$
  - yield in one step of a DFA, $|-_M$
  - computation by DFA M
  - yield of a DFA, $|-_M^*$
  - word accepted by DFA
  - language accepted by DFA M, L(M)
  - equivalence of two DFAs

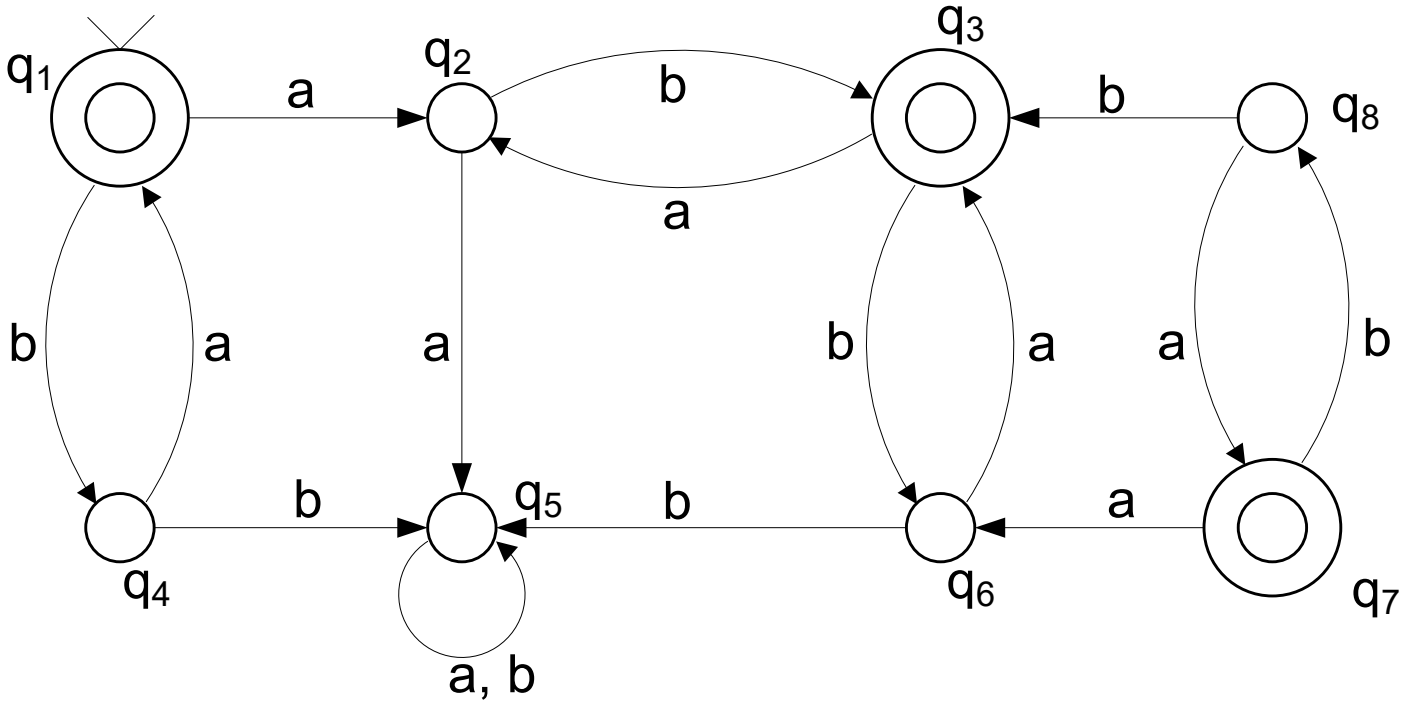
Must be known

- Definition of:
  - regular expression
  - equivalence relation
  - closure
  - language

State minimization

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- For practical reasons it is important to be able to minimize the number of states of a given DFA
- Motivating example: let DFA M such that L(M) = (ab ∪ ba)*

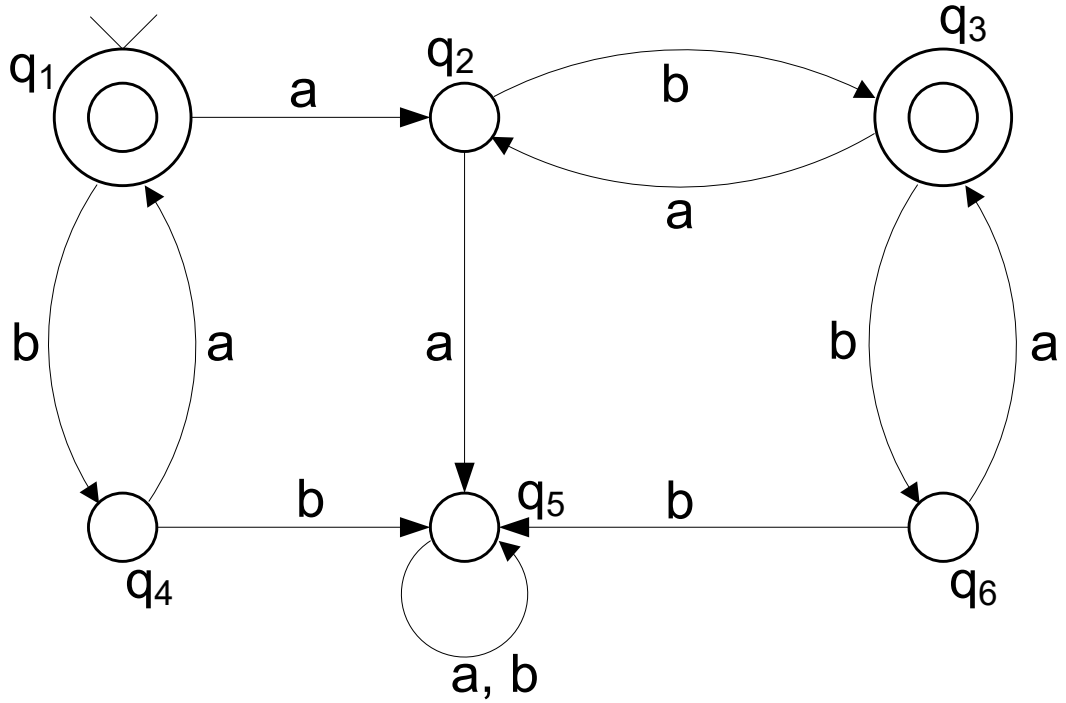


Eliminating unreachable states

- This is the simplest kind of optimization
- State q_7 and q_8 are unreachable because there is no directed path from the start state to them
- Identifying the reachable states can be done in polynomial time
  - the set of reachable states can be defined as the closure of {s} under the relation {(p, q) : δ(p, a) = q for some a $\in$ Σ}
  - all closure can be implemented in polynomial time

– reachable states can be computed by this algorithm:

```
R := {s};
while there is a state p ∈ R and a ∈ Σ
  such that δ(p, σ) ∉ R do
  add δ(p, σ) to R
```



Merging equivalent states

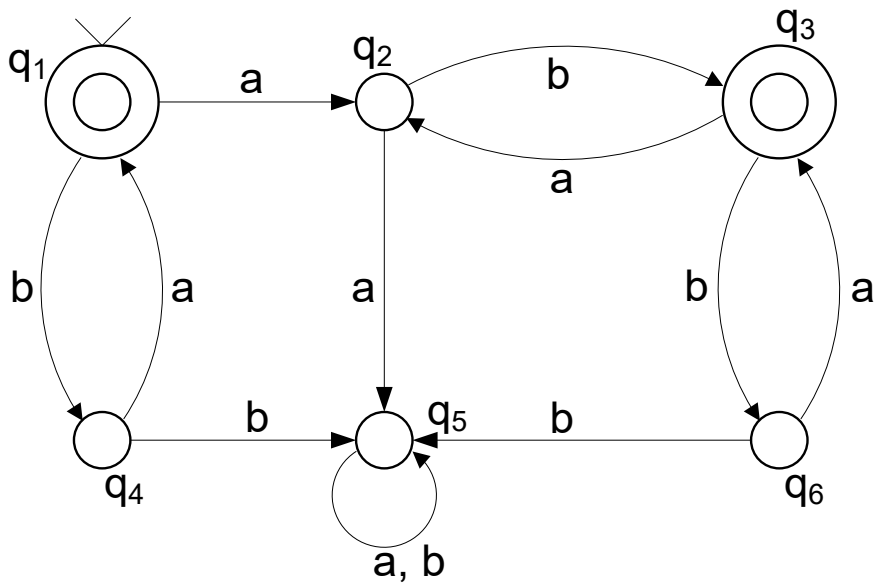
- States q_1, and q_3 are equivalent
  - from either state, precisely the same strings lead the automaton to acceptance
- Equivalent states can be merged into one state

Equivalent strings

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

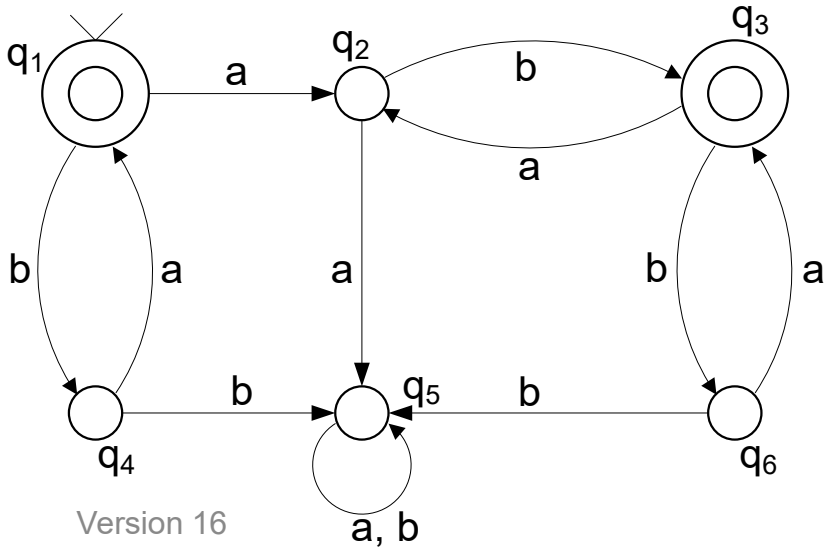
- Definition of x and y are equivalent with respect to L, $x \approx_L y$:
  - let $L \subseteq \Sigma^*$, $x, y \in \Sigma^*$
  - if $\forall z \in \Sigma^*, xz \in L \leftrightarrow yz \in L \rightarrow x \approx_L y$
    - either both concatenated strings are in L or both are not in L
  - DFA is not mentioned here, you cannot check all z
  - we don't know if xz and yz drive the DFA into the same state

- E.g.:
  - ab $\approx_L$ baba
    - z = aa $\rightarrow$ abaa $\notin$ L, babaaa $\notin$ L
    - z = ba $\rightarrow$ abba $\in$ L, bababa $\in$ L
  - a $\approx_L$ baaba (we do not say that x, y $\in$ L)
    - z = ab $\rightarrow$ aab $\notin$ L, baabaab $\notin$ L
    - z = b $\rightarrow$ ab $\in$ L, baabab $\in$ L



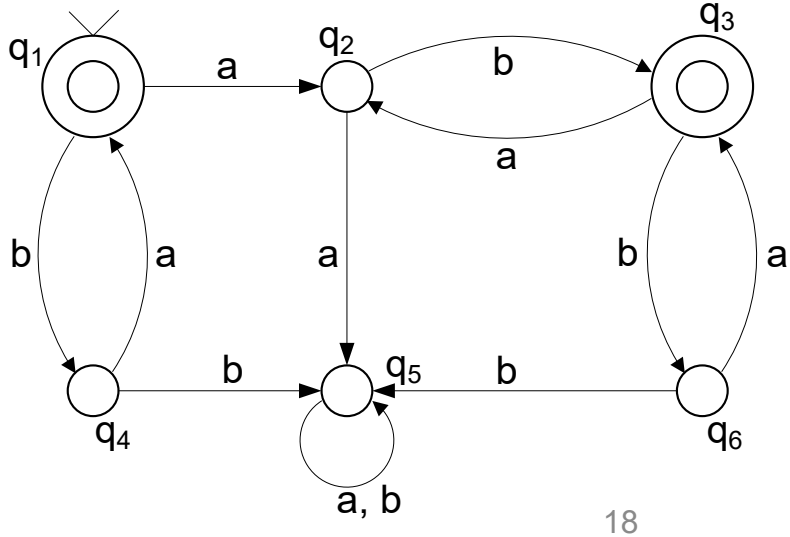
Equivalent strings

- $\approx_L$ is an equivalence relation, so it defines equivalence classes
  - $\approx_L$ the relation holds between any two elements of a class
  - e.g.: $e \approx_L ab \approx_L ba \approx_L abab \approx_L$ ...
  - an equivalence class is denoted with its simplest element (lexicographically)
    - e.g.: [e]



Equivalent strings

- The equivalence classes of $\approx_L$:
  - [e] = L, $\{q_1, q_3\}$
    - the elements of the language
  - [a] = La, $\{q_2\}$
    - a word of the language followed by 'a'
    - any $x \in La$ needs z of the form bL in order for xz to be in L
  - [b] = Lb, $\{q_4, q_6\}$
  - [aa] = $L(aa \cup bb)\Sigma^*$, $\{q_5\}$
    - no matter what z is $xz \notin L$



Equivalent strings

- Definition of x and y are equivalent with respect to M, $\sim_M$:
  - let M = (K, Σ, δ, s, F) be a DFA, x, y $\in$ Σ*
  - if $\exists$ q $\in$ K such that (s, x) $\vdash_M^*$ (q, *e*), (s, y) $\vdash_M^*$ (q, *e*) → x $\sim_M$ y
    - both strings drive M from s to the same state

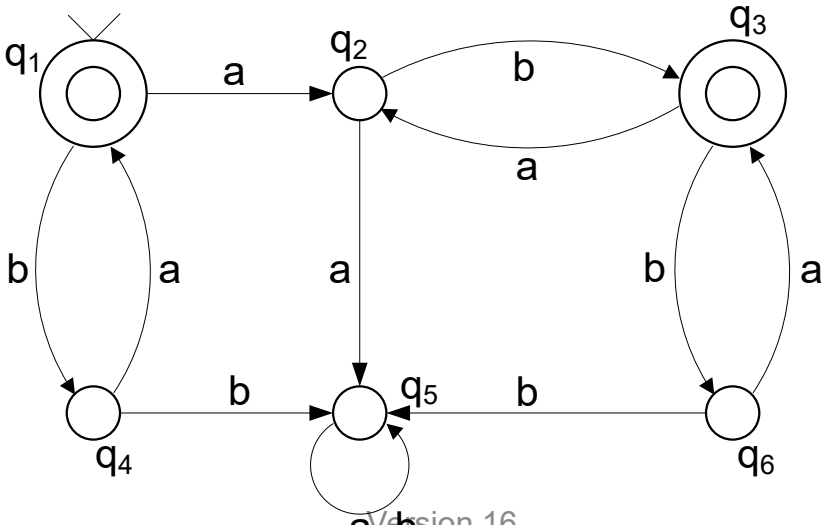
Equivalent strings

- E.g.:
  - ba $\sim_M$ baba
    - (s, ba) $|-_M^*$ (q_1, *e*)
    - (s, baba) $|-_M^*$ (q_1, *e*)
  - a $\sim_M$ baababa (we do not say that x, y $\in$ L)
    - (s, a) $|-_M^*$ (q_2, *e*)
    - (s, baababa) $|-_M^*$ (q_2, *e*)

Equivalent strings

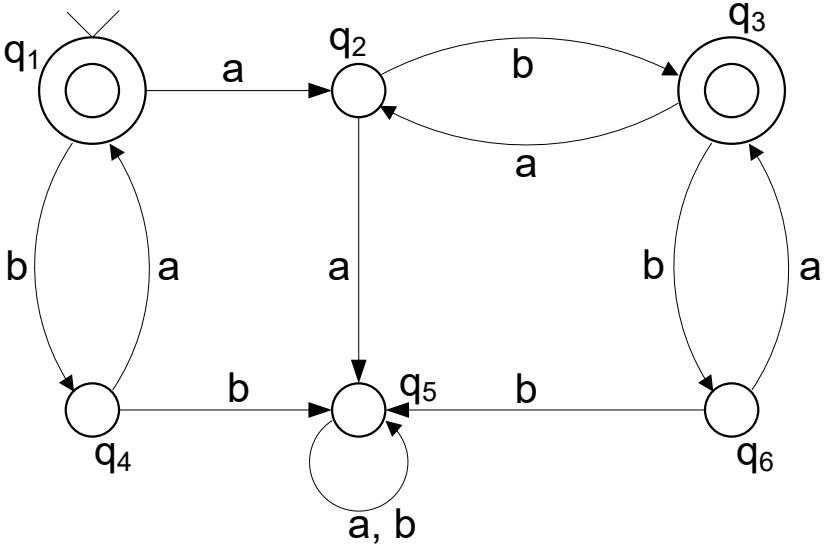
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- $\sim_M$ is an equivalence relation, so it defines equivalence classes
  - $\sim_M$ the relation holds between any two elements of a class
  - e.g.: *e* $\sim_M$ ba $\sim_M$ baba $\sim_M$ ...
  - there is |K| equivalence class denoted by: E_{q1}, E_{q2}, ...



Equivalent strings

- The equivalence classes of $\sim_M$:
  - $E_{q1} = (ba)^*$
  - $E_{q2} = La$
  - $E_{q3} = (ab)^*abL$
    - the elements of the language preceded by ab
  - $E_{q4} = b(ab)^*$
  - $E_{q5} = L(bb \cup aa)\Sigma^*$
  - $E_{q6} = (ab)^*abLb$



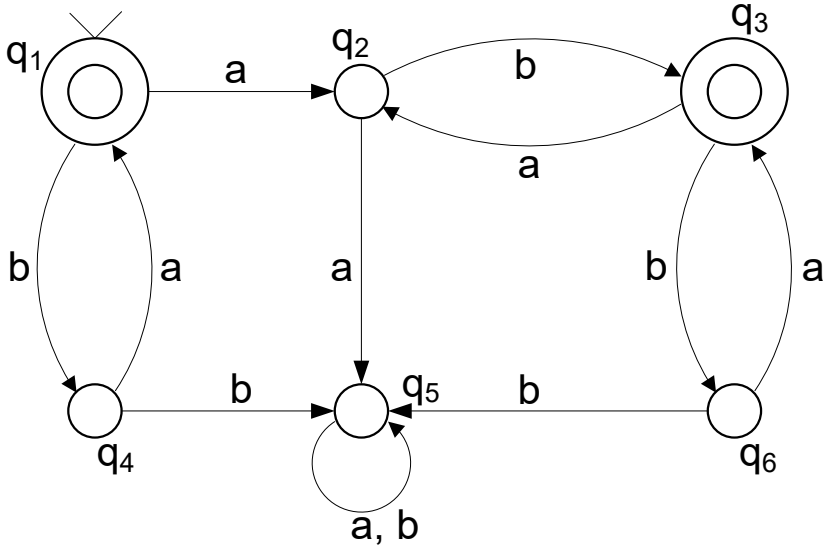
- Theorem: for any DFA $M = (K, \Sigma, \delta, s, F)$, $x, y \in \Sigma^*$, if $x \sim_M y \rightarrow x \approx_{L(M)} y$ (the left side is more strict)
- Proof:
  - informal: if x and y drives M to the same $q \rightarrow xz$ and yz drives M to the same $q \rightarrow$ both xz and yz are accepted or rejected
  - let $q(x) \in K$ such that $(s, x) \vdash_M^* (q(x), e)$
  - $(s, x) \vdash_M^* (q(x), e) \leftrightarrow (s, xz) \vdash_M^* (q(x), z)$
  - let $z \in \Sigma^*$ such $xz \in L(M) \leftrightarrow (s, xz) \vdash_M^* (f, e)$, $f \in F$
  - $xz \in L(M) \leftrightarrow (s, xz) \vdash_M^* (q(x), z) \vdash_M^* (f, e)$, $f \in F$, from the previous two
  - $q(x) = q(y)$, by the definition of $x \sim_M y$
  - $(s, yz) \vdash_M^* (q(y), z) \vdash_M^* (f, e)$, $f \in F$ (previous two)
  - $(s, yz) \vdash_M^* (f, e)$, $f \in F \leftrightarrow yz \in L(M)$
  - both $xz, yz \in L(M)$ which is the definition of $x \approx_{L(M)} y$

Equivalent strings

- The equivalence relation ~ is a refinement of ≈:
  $x \sim y \rightarrow x \approx y$
  - each equivalence class of ≈ is the union of one or more equivalence classes of ~

Equivalent strings

- E.g.:
  - $\sim_M$ is a refinement of $\approx_{L(M)}$
    - $[e] = E_{q1} \cup E_{q3}$
    - $[a] = E_{q2}$
    - $[b] = E_{q4} \cup E_{q6}$
    - $[aa] = E_{q5}$



Merging equivalent states

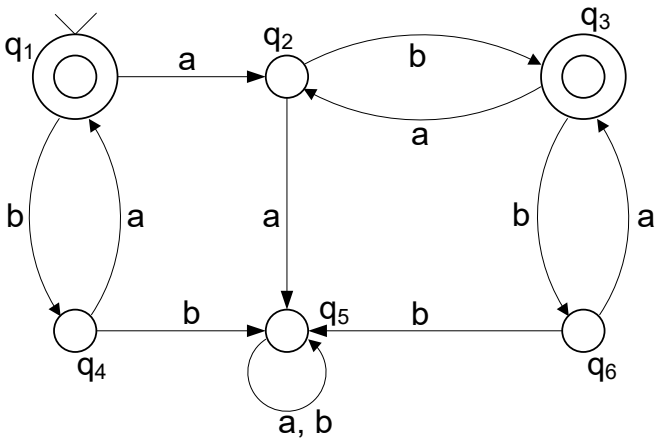
- Theorem (Myhill-Nerode theorem): if $L \subseteq \Sigma^*$ is a regular language, L = L(M) where M is a DFA $\rightarrow$ |K| = the number of equivalence classes in $\approx_L$
  - K is the set of state of M
  - for regular languages always exists M with minimal states

Definitions

- Definition of set A_M: let $M = (K, \Sigma, \delta, s, F)$ be a DFA, $(q, w) \in A_M \leftrightarrow (q, w) \vdash_M^* (f, e)$, $f \in F$
  - w drives M from q to a final state

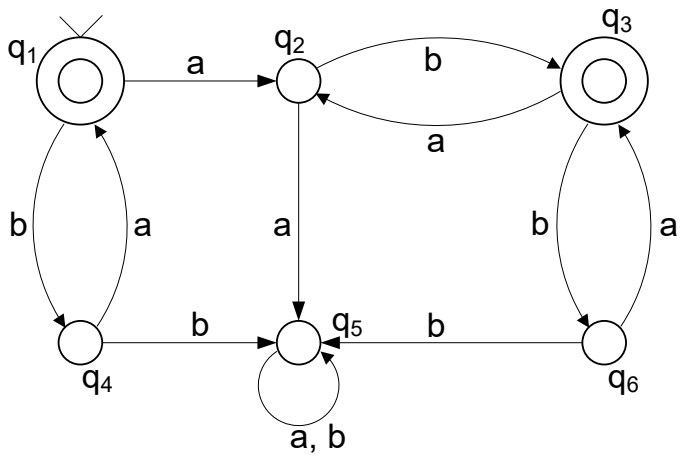
Definitions

- Definition of equivalent states of DFA M, $q \equiv p$:
  if $(\forall z \in \Sigma^*, (q, z) \in A_M \leftrightarrow (p, z) \in A_M) \rightarrow q \equiv p$
  - if each z which drives M from q to a final state also drives M from p to a final state $\rightarrow q \equiv p$
    - also true for non-final states
  - if each string drives M from q and p to the same state (in terms of acceptance) $\rightarrow q \equiv p$
  - the problem again is we cannot check all z



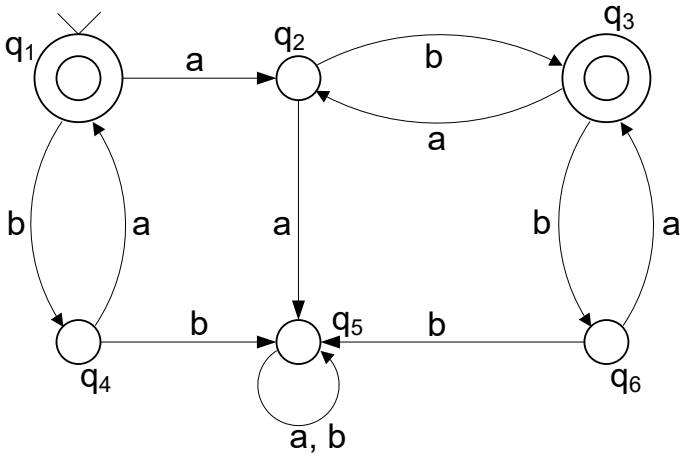
Merging equivalent states

- If $q \equiv p \rightarrow E_q, E_p$ are subsets of the same equivalence class of $\approx_L$
  - e.g.: $q_1 \equiv q_3 \rightarrow E_{q1}, E_{q3} \subseteq [e]$
- Our strategy is not to determine relation ≡ directly but regard each state equivalent and check if some refinements can be done



Merging equivalent states

- Definition of relation $\equiv_n$: if ($\forall$ z $\in \Sigma^*$, $|z| \leq n$, $(q, z) \in A_M \leftrightarrow (p, z) \in A_M$) $\rightarrow$ $q \equiv_n p$
  - if each z of length up to n which drives M from q to a final state also drives M from p to a final state $\rightarrow$ $q \equiv_n p$
  - if n long strings drive M from q and p to the same state (in terms of acceptance) $\rightarrow$ $q \equiv_n p$

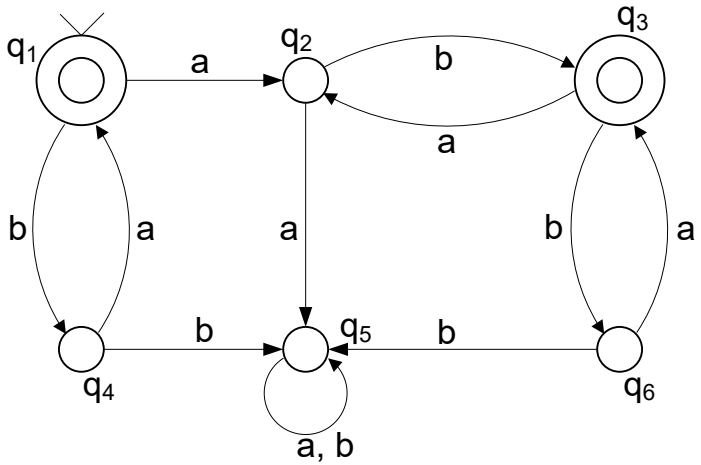


Merging equivalent states

- $q \equiv_0 p$
  - $z = e$
  - if e drives M from q to a final state also drives M from p to a final state $\rightarrow$ $q \equiv_0 p$
  - if $q, p \in F$ or $q, p \notin F \rightarrow$ $q \equiv_0 p$
    - e means 0 yield step
    - $q \equiv_0 p$ holds if both states are either final or both are non-final
    - this can be checked easily
  - $\equiv_0$ has two equivalence classes: F, K-F

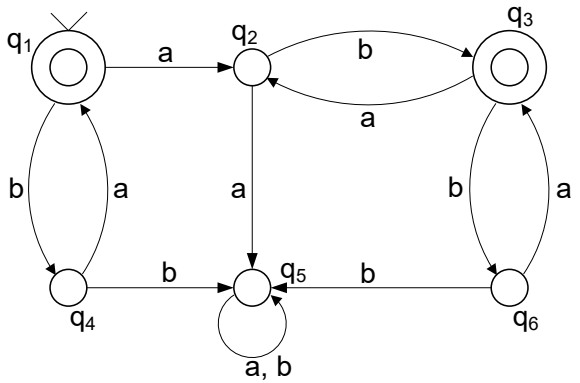
Merging equivalent states

- Theorem: $q, p \in K, n \geq 1, q \equiv_n p \leftrightarrow q \equiv_{n-1} p$ and $\forall \sigma \in \Sigma, \delta(q, \sigma) \equiv_{n-1} \delta(p, \sigma)$



Merging equivalent states

- Proof:
  - $q \equiv_n p \rightarrow q \equiv_{n-1} p$
    - if n long strings drive M to the same state (in terms of acceptance) then n-1 long strings do the same
  - $w = \sigma v$, $|w| = n$
  - left side: $\forall$ n long, $w = \sigma v$, strings drives q and p the same way (in terms of acceptance)
  - right side: $\forall$ n-1 long v strings drives states $\delta(q, \sigma)$ and $\delta(p, \sigma)$ the same way for $\forall$ σ



Merging equivalent states

- If $\forall \sigma \in \Sigma$ drives M from q and p to same equivalence classes of $\equiv_{n-1} \rightarrow q \equiv_n p$
  - if the equivalence classes of $\equiv_{n-1}$ are known, we can check if $\delta(q, \sigma) \equiv_{n-1} \delta(p, \sigma), \forall \sigma \in \Sigma$
- The equivalence classes of $\equiv_0$ is known

Merging equivalent states

- If two elements of an equivalence class of $\equiv_{n-1}$ are not equivalent according to $\equiv_n$ then the class has to be refined (split)
- Each equivalence relation in $\equiv_0, \equiv_1, \equiv_2, \ldots$ is a refinement of the previous one

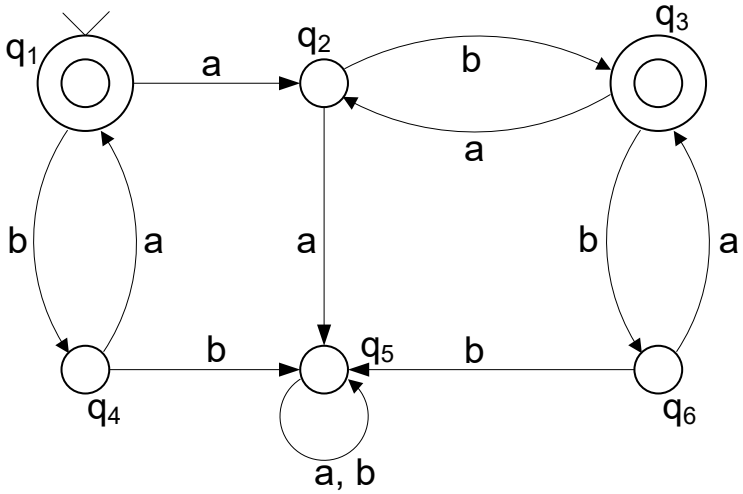
Merging equivalent states

initially the equivalence classes of $\equiv_0$ are F and K-F
repeat, n = 1, 2, ...
  compute equivalence classes of $\equiv_n$ from $\equiv_{n-1}$
until $\equiv_n$ and $\equiv_{n-1}$ are the same

- The algorithm is finite as the maximum number of iterations is |K|-1

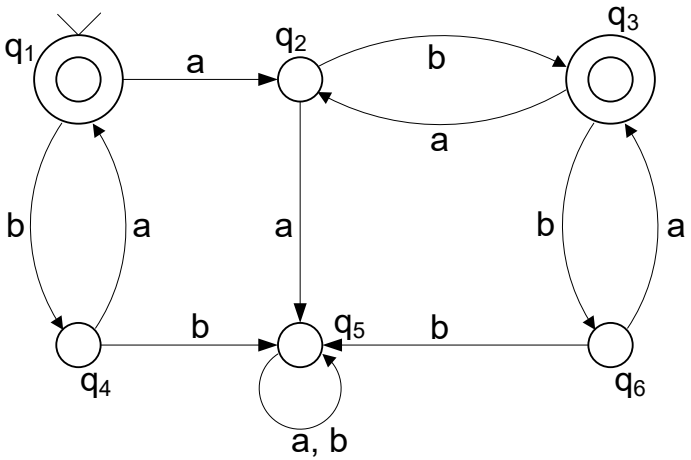
Example

- The equivalence classes of $\equiv_0$ are: $\{q_1, q_3\} = F$, $\{q_2, q_4, q_5, q_6\} = K - F$
  - denote a class with its smallest element
    - e.g.: $\{q_1, q_3\} = Q_1$, $\{q_2, q_4, q_5, q_6\} = Q_2$
  - the sets are disjoint, so the identifiers are unique



Example

- Determine the equivalence classes of $\equiv_1$
  - do we split $\{q_1, q_3\}$?
    - $\delta(q_1, a) \equiv_0 \delta(q_3, a)$, $q_2 \equiv_0 q_2$, $Q_2 = Q_2$
    - $\delta(q_1, b) \equiv_0 \delta(q_3, b)$, $q_4 \equiv_0 q_6$, $Q_2 = Q_2$
    - 'a' and 'b' drive M to the same equivalence class of $\equiv_0$ so we do not split $\{q_1, q_3\}$

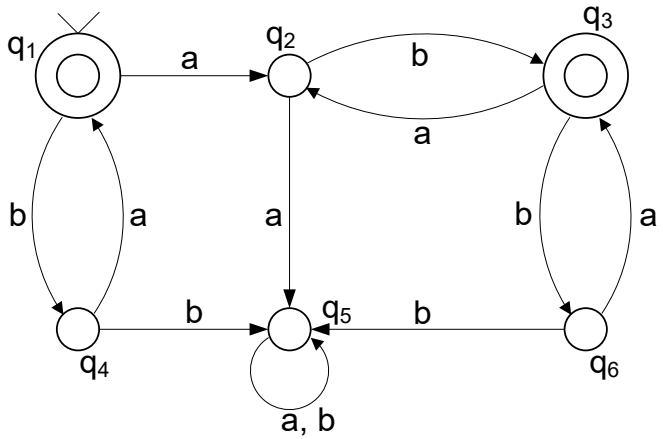


Example

– if $\delta(q_j, \sigma_1) = Q_{i1}$, $\delta(q_j, \sigma_2) = Q_{i2}$, ... → let us denote $\delta(q_j, \Sigma) = (Q_{i1}, Q_{i2}, ...)$

$\{q_1, q_3\} = Q_1$,
$\{q_2, q_4, q_5, q_6\} = Q_2$

– do we split $\{q_1, q_3\}$?
  - $\delta(q_1, \Sigma) = (Q_2, Q_2)$
  - $\delta(q_3, \Sigma) = (Q_2, Q_2)$
    – the elements within a column are the same → no split

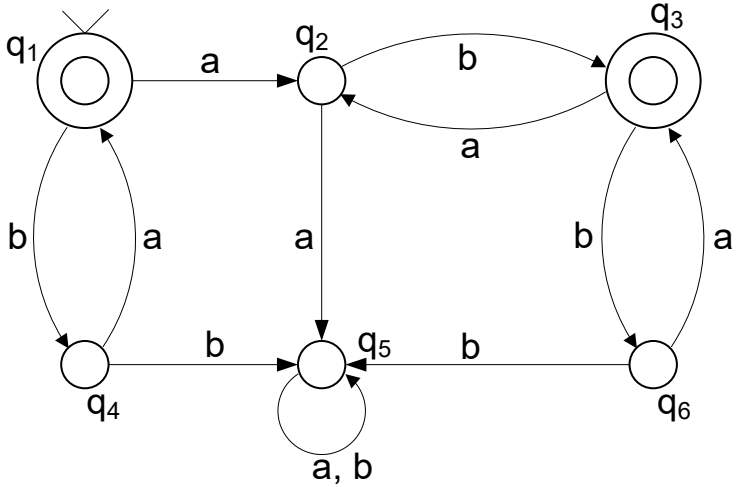


$\{q_1, q_3\} = Q_1$
$\{q_2, q_4, q_5, q_6\} = Q_2$

- do we split $\{q_2, q_4, q_5, q_6\}$?
  - $\delta(q_2, \Sigma) = (Q_2, Q_1)$
  - $\delta(q_4, \Sigma) = (Q_1, Q_2)$
  - $\delta(q_5, \Sigma) = (Q_2, Q_2)$
  - $\delta(q_6, \Sigma) = (Q_1, Q_2)$
  - the elements within a column are not the same → split
  - $\delta(q_2, \Sigma) = (Q_2, Q_1)$

  - $\delta(q_4, \Sigma) = (Q_1, Q_2)$
  - $\delta(q_6, \Sigma) = (Q_1, Q_2)$

  - $\delta(q_5, \Sigma) = (Q_2, Q_2)$

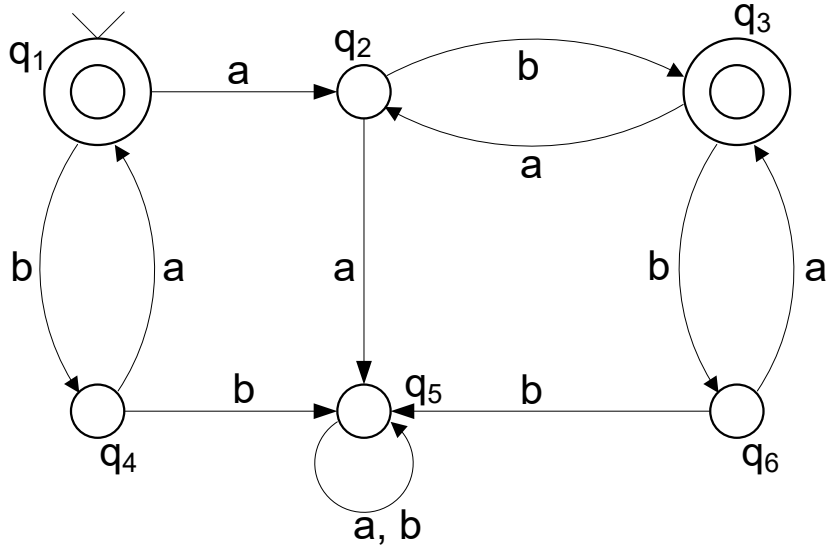


- the equivalence classes of $\equiv_1$ are: $\{q_1, q_3\}$, $\{q_2\}$, $\{q_4, q_6\}$, $\{q_5\}$

Example

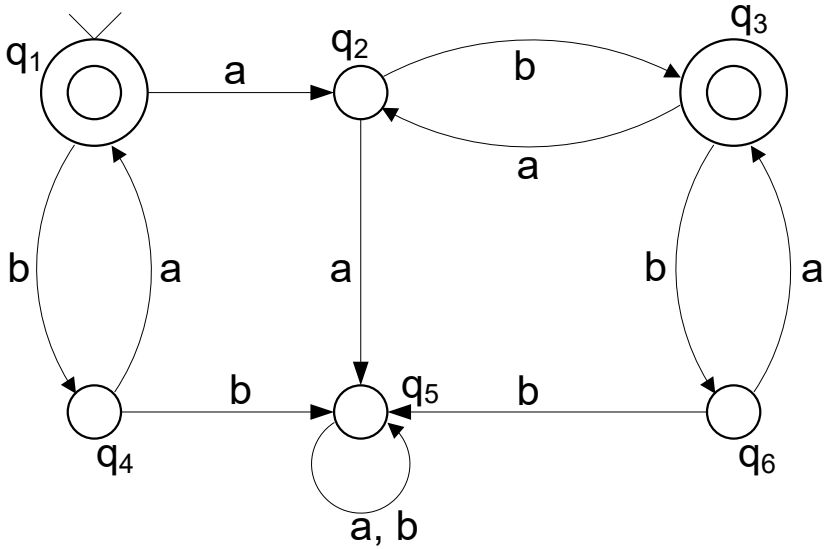
$\{q_1, q_3\} = Q_1$
$\{q_2\} = Q_2$
$\{q_4, q_6\} = Q_4$
$\{q_5\} = Q_5$

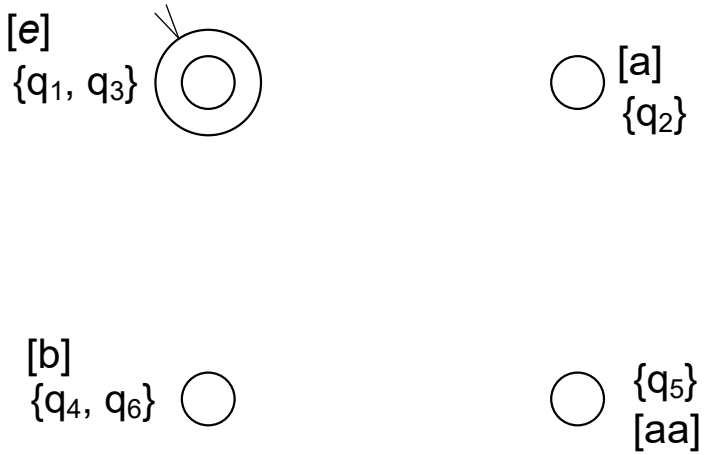
- do we split $\{q_4, q_6\}$?
  - singleton set like $\{q_2\}$ can not be split
  - $\delta(q_4, \Sigma) = (Q_1, Q_5)$
  - $\delta(q_6, \Sigma) = (Q_1, Q_5)$
  - the elements within a column are the same → no split



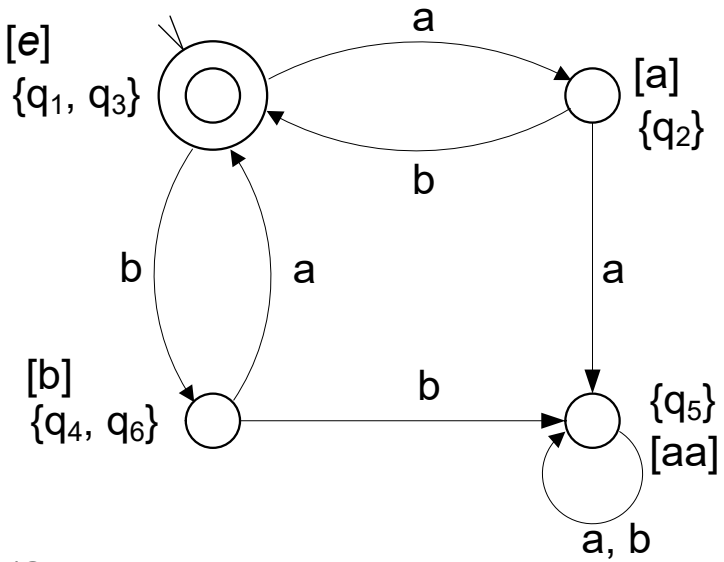
Merging equivalent states

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen





- Draw the nodes
- Draw the arcs



Introduction to the Theory of Computation

Lesson 2

2.6. Algorithmic aspects of finite automat

3.2. Parse trees

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Strukturális és Beruházási Alapok

BEFEKTETÉS A JÖVŐBE

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Theorem: there are algorithms to accomplish the next tasks with the given complexities
  - NFA → DFA exponential
  - RE → NFA polynomial
  - NFA → RE exponential
  - DFA → minimal DFA polynomial

Space-time tradeoff

- Theorem: regard RE r and NFA M such that L(M) = L(r) → |K| ~ |r|
- Proof:
  - the basic machines which accept a simple symbol have two states
  - union, Kleene star introduce 1 new state
  - K < 2*|r| according to the Thomson's construction theorems
- There is at most 2 transitions from each state, |δ| ~ |r|
  - it is not true for NFA in general but only to those which were created by the construction theorems

Space-time tradeoff

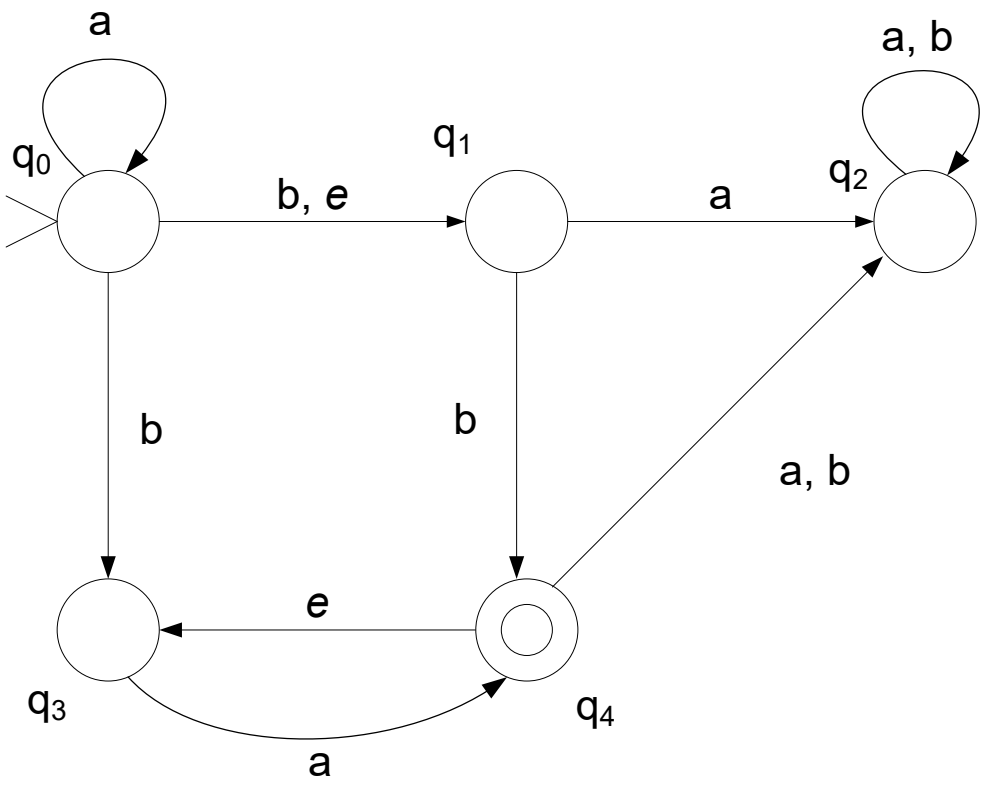
- It is possible to simulate an NFA by constructing a DFA on-the-fly
  - S is the current DFA state, i.e., a set of NFA states

Space-time tradeoff

$S_0 = E(s)$
a = nextchar()
while a ≠ eos
  n = n + 1
  $S_n = \cup\{E(q), \forall p \in S_{n-1}, (p, a, q) \in \Delta)\}$
  a = nextchar()
end
if $S_{n-1} \cap F = \emptyset$ return no
else return yes

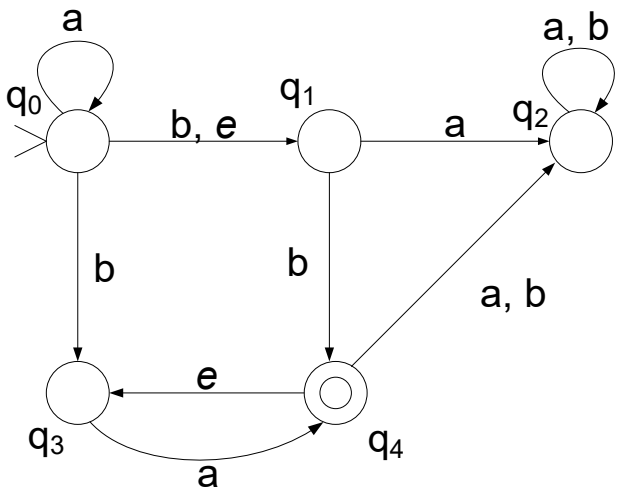
Example

- Check directly if aaaba is accepted by the NFA below!



Example

- The various values for the set S_n are shown below
  - $S_0 = \{q_0, q_1\}$
  - $S_1 = \{q_0, q_1, q_2\}$
  - $S_2 = \{q_0, q_1, q_2\}$
  - $S_3 = \{q_0, q_1, q_2\}$
  - $S_4 = \{q_1, q_2, q_3, q_4\}$
  - $S_5 = \{q_2, q_3, q_4\}$
    - it contains a final state so the word is accepted



Space-time tradeoff

- Definition of the size of a finite automaton, $|\delta|$: the size of the transition table
- Definition of the time (speed) of a finite automaton: the number of step required to process $w \in \Sigma^*$

Space-time tradeoff

- Regard RE r, $w \in \Sigma^*$, decide if $w \in L(r)$
  - method 1:
    - construct NFA M_1 such that $L(M_1) = L(r)$
    - construct DFA M_2 such that $L(M_2) = L(M_1)$
    - space: $|\delta_2| \sim 2^{|\Delta 1|} \sim 2^{|r|}$
      - if K increases → δ also increases
    - time: $|w|$
      - one symbol is processed in every step

Space-time tradeoff

- method 2:
  - construct NFA M_1 such that $L(M_1) = L(r)$
  - simulate M_1 directly
  - space: $|\delta_1| \sim |r|$
  - time: $|w|*|r|^2 \sim |w|*|K_1|^2$
    - $|w|$ iteration of the while loop
    - at most $|K_1|^2$ steps is required to calculate the new S set
      - » $|K_1|^2$ triplet, $(p, a, q) \in \Delta$, is to be checked

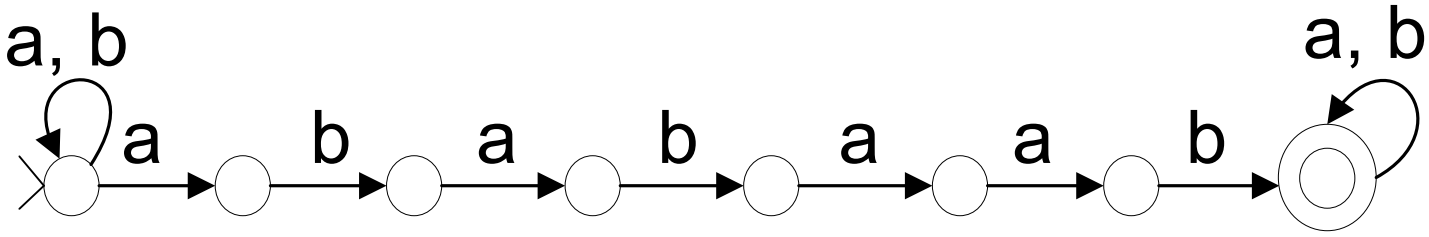
Space-time tradeoff

	Method 1	Method 2
space	$O(2^{\|r\|})$	$O(\|r\|)$
time	$O(\|w\|)$	$O(\|w\|*\|r\|^2)$

- There are combined techniques which is better for certain type of problems

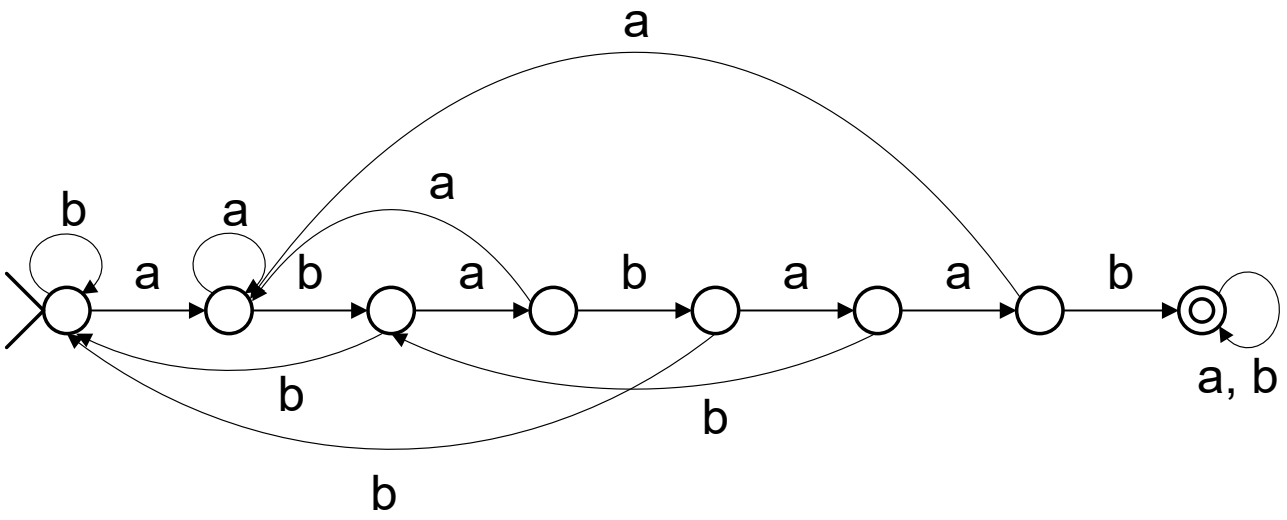
String matching

- Given a fixed string x, determine if x is a substring of w!
  - remember that x is fixed and not an input
- The NFA below recognize pattern ababaab
  - Σ = {a, b}



String matching

- Though the NFA → DFA conversion may increases the number of states exponentially at this example this does not happen
  - the size of the DFA does not increase exponentially
  - in practice, another kind of algorithm is used because Σ is usually large



Must be known

- Definition of:
  - regular grammar
  - context-free grammar
  - derivation of a string
  - partially defined string

Example

- Give grammar G which can create a sequence of balanced parentheses!
  - every right parenthesis can be paired with a unique preceding left parenthesis
  - this is not a regular language
- Let G be the grammar (V, Σ, R, S), where
  - V = {S, (,)}
  - Σ = {(,)}
  - R = {S → *e* | SS | (S)}
    - it is in CFG but not in RG

Parse trees

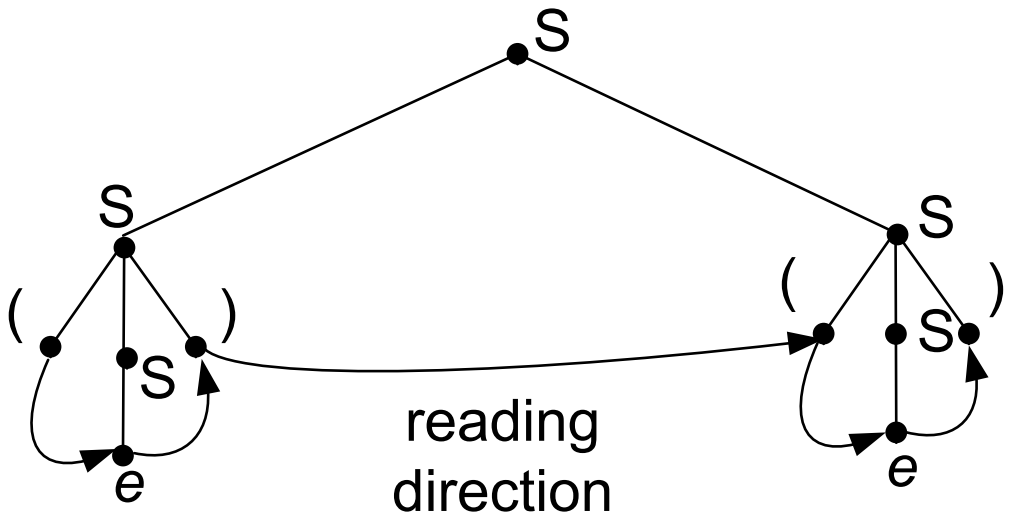
- A string w $\in$ L(G) may have many derivations in G
- E.g.: if G is the CFG that generates the language of balanced parentheses, then the string ()() can be derived from S by at least two distinct derivations
  - S=>SS=>(S)S=>()S=>()(S)=>()()
  - S=>SS=>S(S)=>(S)(S)=>(S)()=>()()

Parse trees

- These two derivations are "the same" in a sense
  - the rules used are the same
  - the rules are applied for the same non-terminals in the partially defined string
  - the only difference is in the order in which the rules are applied

Parse trees

- Both derivations can be pictured like this



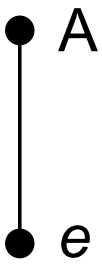
Parse trees

- Such a picture is called a parse tree
  - the points are called nodes
  - each node carries a label that is a symbol in V
  - the topmost node is called the root
  - the nodes along the bottom are called leaves
  - all leaves are labeled by terminals or possibly the empty string *e*
- Definition of the yield of the parse tree: concatenate the labels of the leaves from left to right
  - the derived string

Parse trees

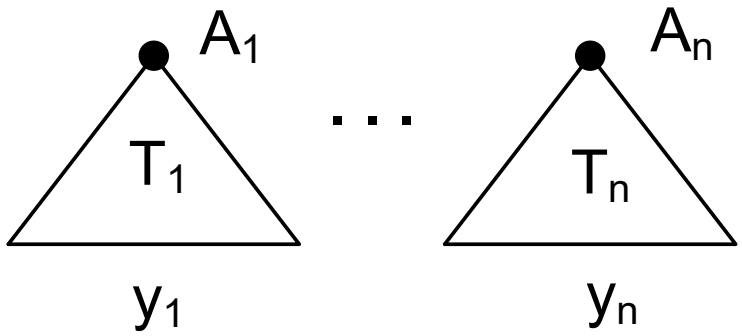
- Definition of the parse tree for CFG G(V, Σ, R, S):
  - a
    - this is the parse tree for each a $\in$ Σ
      - its root: 'a'
      - its leaf: 'a'
      - its yield: 'a'

Parse trees



– this is a parse tree if $A \rightarrow e$ is a rule in R
  - its root: A
  - its leaf: *e*
  - its yield: *e*

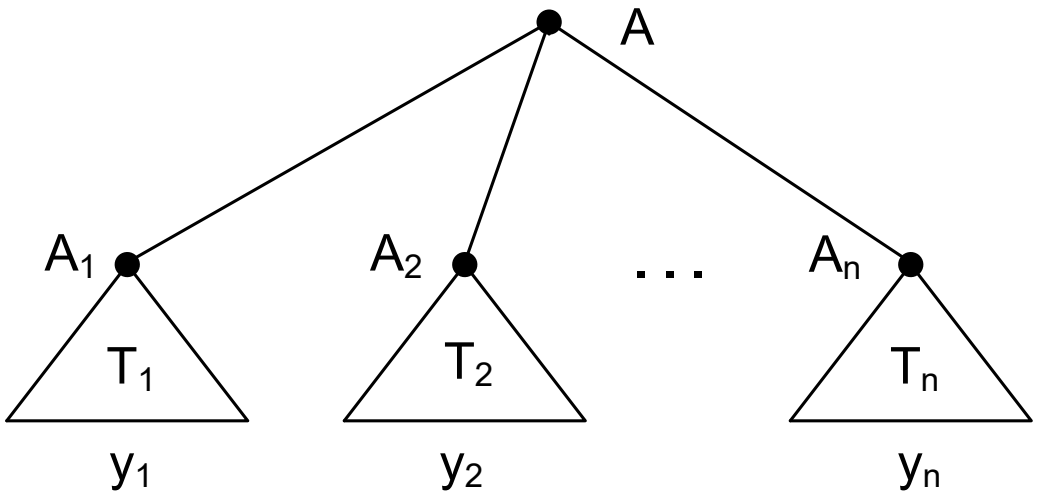
Parse trees



– if the above graphs are parse trees, where n ≥ 1, with roots labeled A_1, A_2, … A_n respectively, and with yields $y_1 \ldots y_n$, and $A \rightarrow A_1 A_2 \ldots A_n$ is a rule in R then:

Parse trees

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



- this is a parse tree
  - its root: A
  - its leaves: the leaves of its constituent parse trees
  - its yield: $y_1 y_2 \ldots y_n$
- nothing else is a parse tree

Parse trees

- Parse trees are ways of representing derivations of strings in L(G)
  - the superficial differences between derivations, owing to the order of application of rules, are suppressed

Parse trees

- Parse trees represent equivalence classes of derivations
  - several derivation may belong to the same parse tree
  - there is a precedence relation between the derivations belonging to the same parse tree

Precedence relation

- We say a derivation precedes another if
  - the two derivations are identical except for two consecutive steps
  - in these steps the two nonterminals are replaced by the same two strings but in opposite order in the two derivations
  - derivation A precedes derivation B if in derivation A the leftmost of the two mentioned nonterminals is replaced first

Precedence relation

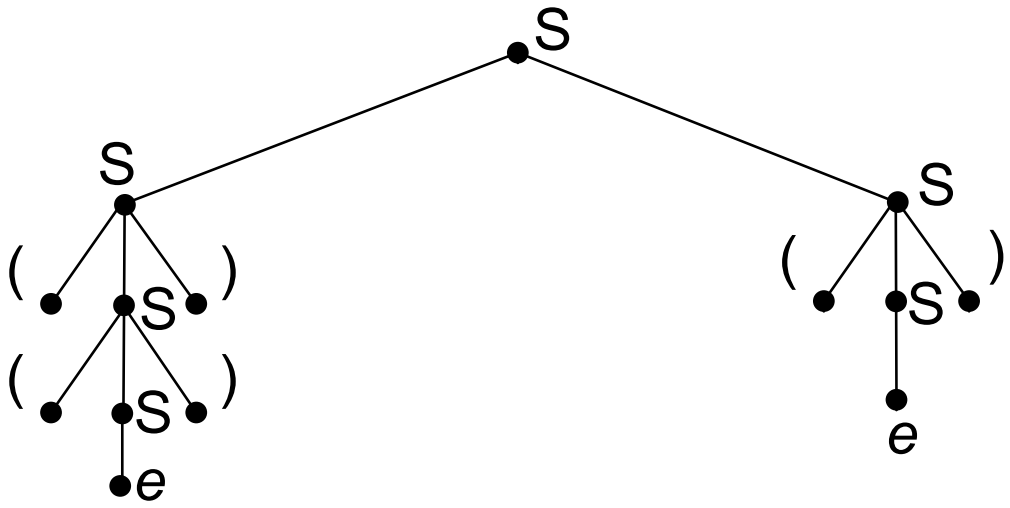
- Definition of precedence relation, ≤:
  - let G(V, Σ, R, S) is a CFG
  - D=x_1 => x_2 =>…=> x_n, D'=x_1' => x_2' =>…=> x_n' are derivations in G
    - where x_i, x_i' ∈ V* for i = 1, …, n, x_1, x_1' ∈ V-Σ, x_n = x_n' ∈ Σ*
  - D and D' are derivations of terminal strings from a single nonterminal

Precedence relation

- we say that D precedes D', written $D \leq D'$, if $n>2$, $\exists$ k, $1 < k < n$ such that
  - $\forall\ i \neq k,\ x_i = x_i'$
  - $x_{k-1} = x_{k-1}' = uAvBw$ (they have this form)
    - $u, v, w, y, z \in V^*$, $A, B \in V - \Sigma$
  - $x_k = uyvBw$, where $A \rightarrow y \in R$
  - $x_k' = uAvzw$, where $B \rightarrow z \in R$
  - $x_{k+1} = x_{k+1}' = uyvzw$

Example

- Recall CFG G generating all strings of balanced parentheses
- Consider these three derivations D_1, D_2, D_3 of (())()
  - D_1=S=>SS=>(S)S=>((S))S=>(())S=>(())(S)=>(())()
  - D_2=S=>SS=>(S)S=>((S))S=>((S))(S)=>(())(S)=>(())()
  - D_3=S=>SS=>(S)S=>((S))S=>((S))(S)=>((S))()=>(())()



Example

- $D_1 \leq D_2$
  - they differ for k=5
  - D_1: S→() for the 1st S, S→(S) for the 2nd S
  - D_2: S→(S) for the 2nd S, S→() for the 1st S
- $D_2 \leq D_3$
  - they differ for k=6
- neither $D_1 \leq D_3$ nor $D_3 \leq D_1$ is true because the two derivations differ in more than one intermediate string
  - ≤ is not total order

Similarity relation

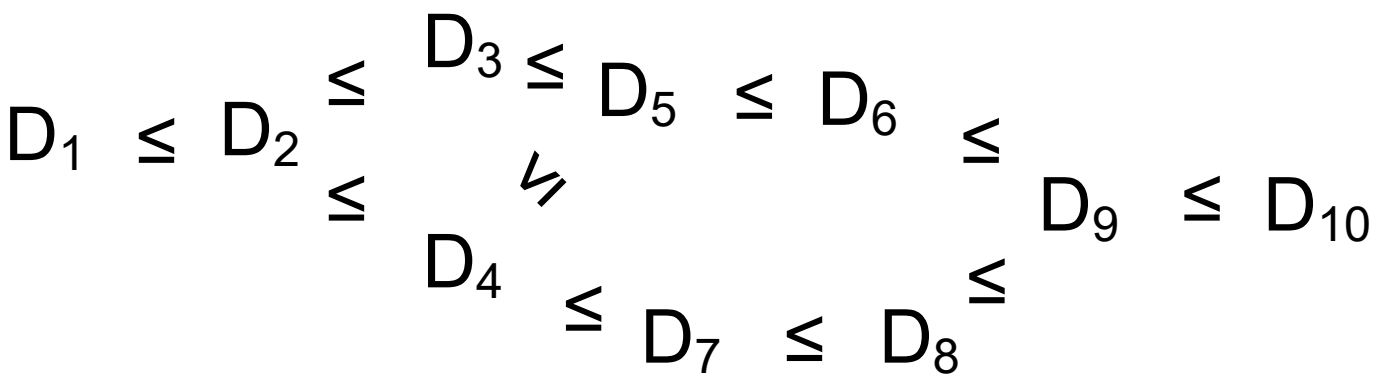
- Definition of similarity relation: the reflexive, symmetric, transitive closure of ≤
  - it is an equivalence relation
- Two derivations are similar if
  - they can be transformed into each other via a sequence of "switching" in the order in which rules are applied
    - a "switching" transform a derivation either by one that precedes it, or by one that it precedes
  - they belong to the same parse tree

Example

- The equivalence class of the derivations of (())() corresponding to the tree showed previously contains the derivations D_1, D_2, D_3 and also
  - D_4=S=>SS=>(S)S=>(S)(S)=>((S))(S)=>(())(S)=>(())()
  - D_5=S=>SS=>(S)S=>(S)(S)=>((S))(S)=>((S))()=>(())()
  - D_6=S=>SS=>(S)S=>(S)(S)=>(S)()=>((S))()=>(())()
  - D_7=S=>SS=>S(S)=>(S)(S)=>((S))(S)=>(())(S)=>(())()
  - D_8=S=>SS=>S(S)=>(S)(S)=>((S))(S)=>((S))()=>(())()
  - D_9=S=>SS=>S(S)=>(S)(S)=>(S)()=>((S))()=>(())()
  - D_{10}=S=>SS=>S(S)=>S()=>(S)()=>((S))()=>(())()
- These derivations are all similar

Example

- These ten derivations are related by ≤ as shown here



Leftmost derivation

- Definition of the leftmost derivation: each equivalence class of derivations under similarity relation, i.e., each parse tree, contains a derivation that is not preceded by any other derivation
- Definition of one step leftmost derivation, $=>^{L}$:
  - if $x = wA\beta$, $y = w\alpha\beta$, $w \in \Sigma^*$, $\alpha, \beta \in V^*$, $A \in V - \Sigma$, $A \rightarrow \alpha \in R$
  - then $x =>^{L} y$
- A leftmost derivation exists in every parse tree
  - repeatedly replace the leftmost nonterminal in the current string
  - a leftmost derivation contains only $=>^{L}$ derivation

Rightmost derivation

- Definition of the rightmost derivation: each equivalence class of derivations under similarity relation, i.e. each parse tree, contains a derivation that does not preceded any other derivation
- Definition of one step rightmost derivation $=>^R$:
  - if $x = \beta A w$, $y = \beta \alpha w$, $w \in \Sigma^*$, $\alpha, \beta \in V^*$, $A \in V - \Sigma$, $A \rightarrow \alpha \in R$
  - then $x =>^R y$
- A rightmost derivation exists in every parse tree
  - repeatedly replace the rightmost nonterminal in the current string
  - a rightmost derivation contains only $=>^R$ derivation

Parse trees

- The leftmost derivation is unique since at each step the leftmost nonterminal is to be replaced
- The rightmost derivation is unique since at each step the rightmost nonterminal is to be replaced
- In the previous example D_1 is the leftmost derivation and D_{10} is the rightmost one

Parse trees

- Theorem:
  - G(V,Σ,R,S) is a CFG, $A \in V-\Sigma$, $w \in \Sigma^*$
  - the following statements are equivalent
    - $A \Rightarrow^* w$
    - there is a parse tree with root A and yield w
    - there is a leftmost derivation $A \Rightarrow_L^* w$
    - there is a rightmost derivation $A \Rightarrow_R^* w$

Ambiguity

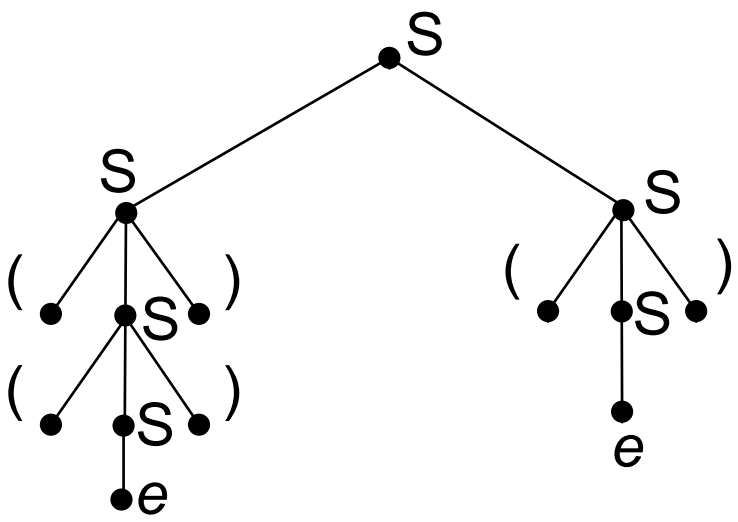
- There may be a string in a language generated by a context-free grammar with two derivations that are not similar
  - the string has distinct parse trees (or equivalently distinct rightmost derivations, and two distinct leftmost derivations)

Example

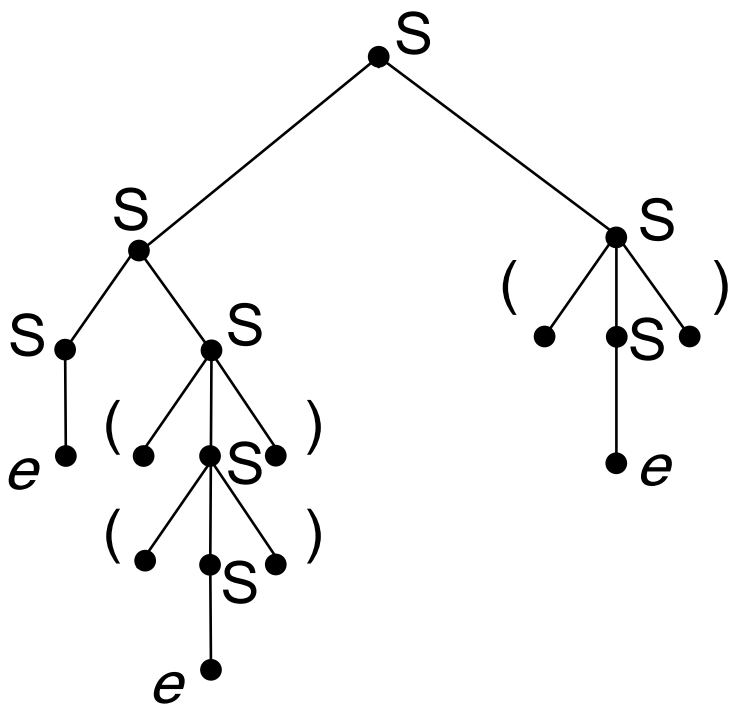
- Recall CFG G generating all strings of balanced parentheses
- There are other derivations of (())() that are not similar to the ones already introduced (D_1 - D_{10})
- They do not belong to the parse tree shown previously
- E.g.:
  - S => SS => SSS => S(S)S => S((S))S => S(())S => S(())(S) => S(())() => (())()

Example

old



new

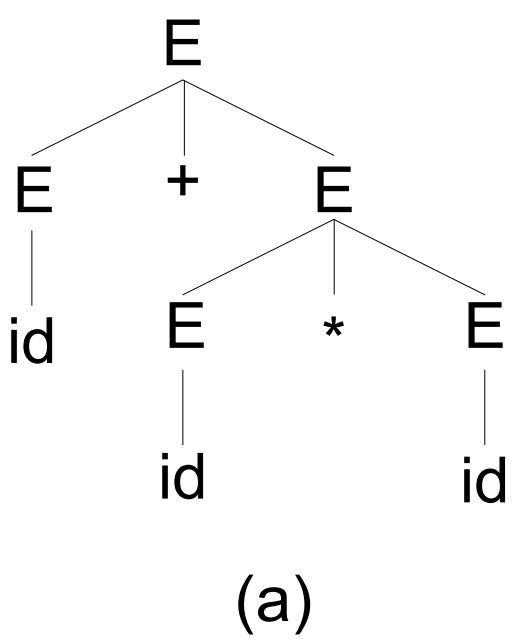


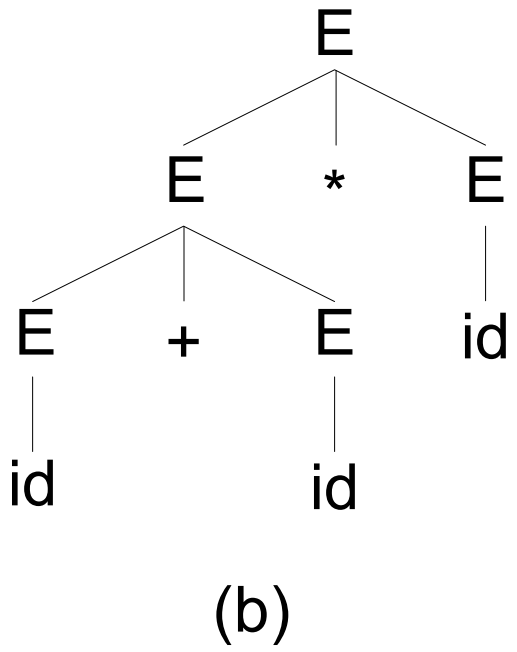
Example

- Consider G'(V, Σ, R, E), where
  - V = {+ ,* , (,), id, E}
  - Σ = {+ , *, (,), id}
  - R = {E → E+E | E*E | (E) | id}
- G' generates all arithmetic expressions over id
  - recall that we already generated this language with grammar G which contained nonterminals for term and factor
  - L(G') = L(G), but G' does not express that addition and multiplication has different precedence

Example

- There are two parse trees for id+id*id in G'
- The first parse tree corresponds to the natural meaning of this expression (with * taking precede over +), the other is "wrong"





Ambiguity

- Definition of ambiguous grammar: such a CFG whose language contains such a string that have two or more distinct parse trees
  - e.g.: G'
- Definition of parsing: assigning a parse tree to a given string
- Unambiguous grammars are important for programming languages

Ambiguity

- We can "disambiguate" some ambiguous grammar
  - e.g. for G' we can introduce the nonterminals T and F
  - e.g. there is an unambiguous grammar to create balanced strings of parentheses
- Definition of inherently ambiguous languages: there is no unambiguous CFG to generate them
  - programming languages are never inherently ambiguous

Introduction to the Theory of Computation

Lesson 3

Chomsky normal form, deterministic context-free languages

3.6. Algorithms for context-free grammars

3.7. Determinism

Dr. István Heckl

Istvan.Heckl@gmail.com

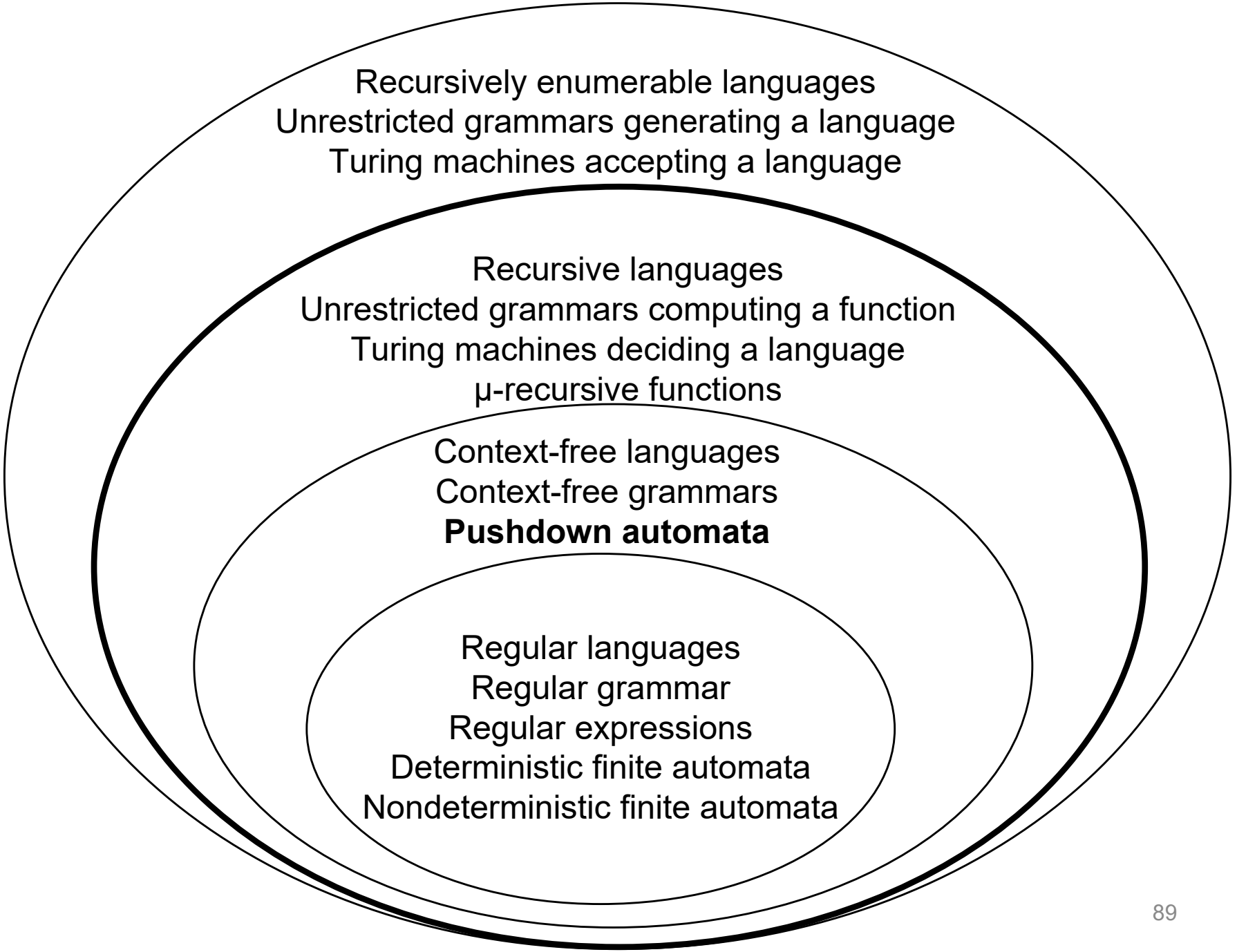
University of Pannonia

MAGYARORSZÁG KORMÁNYA

BEFEKTETÉS A JÖVŐBE

Case		English Name
Upper	Lower	
A	α	alpha
B	β	beta
Γ	γ	gamma
Δ	δ	delta
E	ϵ	epsilon
Z	ζ	zeta
H	η	eta
Θ	θ	theta
I	ι	iota
K	κ	kappa
Λ	λ	lambda
M	μ	mu

Case		English Name
Upper	Lower	
N	ν	nu
Ξ	ξ	xi
O	o	omicron
Π	π	pi
P	ρ	rho
Σ	ς	sigma
T	τ	tau
Υ	υ	upsilon
Φ	ϕ	phi
X	χ	chi
Ψ	ψ	psi
Ω	ω	omega


Recursively enumerable languages
Unrestricted grammars generating a language
Turing machines accepting a language
Recursive languages
Unrestricted grammars computing a function
Turing machines deciding a language
μ-recursive functions
Context-free languages
Context-free grammars
Pushdown automata
Regular languages
Regular grammar
Regular expressions
Deterministic finite automata
Nondeterministic finite automata

Must be known

- Definition of:
  - PDA
  - simple PDA
  - PDA → simple PDA conversion
  - simple PDA → CFG conversion

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Theorem: there is a polynomial algorithm which, given a
  - context-free grammar, construct an equivalent pushdown automaton
  - pushdown automaton, construct an equivalent context-free grammar
  - a context-free grammar G and a string w, decides whether $w \in L(G)$

Comparing theorems about regular and context-free languages

- Similarity: language acceptors can be transformed to generators and vice versa
- Differences:
  - it is trivial to test if $w \in \Sigma^*$ belongs to a given regular language
    - construct the appropriate DFA, run it with w
    - the introduced PDA is nondeterministic, so it is not obvious that w is accepted or not
  - there is no algorithm to
    - test if $L(M_1) = L(M_2)$, M_1 and M_2 are PDA

Chomsky normal form

- Definition of Chomsky normal form: the rules of a CFG has the next form $R \subseteq (V - \Sigma) \times V^2$
  - the right-hand side of every rule must have length two
  - grammars in this form cannot produce strings of length less than 2
    - otherwise they may produce anything

Chomsky normal form

- Theorem: for any context-free grammar G ∃ a context-free grammar G' in Chomsky normal form such that L(G') = L(G) - {Σ ∪ {*e*}}
  - the construction of G' can be carried out in time polynomial in the size of G

Chomsky normal form

- Proof by construction:
  - the rules of G may violate the constraints of Chomsky normal form
    - long rules: right-hand side has length 3 or more
    - *e*-rules: $A \rightarrow e$
    - short rules: $A \rightarrow a$ or $A \rightarrow B$

Chomsky normal form

- Step 1: eliminate $\forall$ long rule: $A \rightarrow B_1B_2 \ldots B_n \in R$, $B_1, B_2, \ldots, B_n \in V$, $n \geq 3$
  - replace the previous long rule with n-1 new rules:
    $A \rightarrow B_1A_1$
    $A_1 \rightarrow B_2A_2$
    ...
    $A_{n-2} \rightarrow B_{n-1}B_n$
  - $A_1, A_2, \ldots A_{n-2}$ are new nonterminals
    - they do not appear anywhere else

Chomsky normal form

- the resulting grammar is
  - equivalent to the original one
  - has rules with right-hand sides of length 2 or less

Chomsky normal form

– Step 2: eliminate $\forall$ *e*-rule: $A \rightarrow e$, $A \in V - \Sigma$
  - determine the set of erasable nonterminals: ε
    $\varepsilon = \{A \in V - \Sigma : A \Rightarrow^* e\}$

```
ε := ∅
while ∃ α ∈ ε*, ∃ A → α, and A ∉ ε
  add A to ε
```

  - delete the *e*-rules

Chomsky normal form

- for $\forall$ rule: $A \rightarrow BC$ or $A \rightarrow CB$, $B \in \varepsilon$ add new rule
  $A \rightarrow C$
  - before deleting the $B \rightarrow e$: A => BC =>* C
  - now: A =>* C
- the new grammar can not generate *e*

Chomsky normal form

- Step 3a: eliminate $\forall$ short rule: $A \rightarrow a$ or $A \rightarrow B$
  - determine the replacement sets:
    $D(A) = \{B \in V : A =>^* B\}, \forall A \in V$
    - D(A) - set of single symbols that can be derived from A in the grammar

```
D(A) := {A}
while ∃ B ∈ D(A), ∃ B → C, C ∉ D(A)
    add C to D(A)
```

Chomsky normal form

- delete the short rules
- for $\forall$ rule: $A \rightarrow BC$ add new rules $A \rightarrow B'C'$, $B' \in D(B)$, $C' \in D(C)$
  - $|D(B)| * |D(C)|$ new rules

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- before deleting B → B': A => BC =>* B'C
- now: A =>* B'C

– Step 3b: for ∀ rule A → BC where A ∈ D(S) - {S} add rules S → BC

- before deleting S → A: S =>* A => BC
- now: S =>* BC

• the new grammar can not generate strings of length 1 but it produces the same language as the previous one

Example

- Transform the next grammar into Chomsky normal form!
  - the CFG generates the set of balanced parenthesis
    - R = {S → SS, S → (S), S → *e*}
  - removing long rule: S → (S)
    - adding new rules: S → (S_1, S_1 → S)
    - if the long rule would have been S → ((())), the new rules would be: S → (S_1, S_1 → (S_2, S_2 →)S_3, S_3 → (S_4, S_4 →))

Example

- removing *e*-rules
  - ε = {S}
  - omit *e*-rule: S → *e*
  - add new rules:
    - S → S because of S → SS
    - S_1 →) because of S_1 → S)

Example

– if the rules would have been S → A, A → B, B → *e*, C → B, D → AB, E → AC, F → Aa then
  - ε = {B, A, S, C, D, E}

Example

- the current set of rules: $S \rightarrow SS$, $S \rightarrow (S_1$, $S_1 \rightarrow S)$, $S \rightarrow S$, $S_1 \rightarrow)$
- removing short rules:
  - determine the replacement sets:
    - $D(S_1) = \{S_1,)\}$
    - $D(A) = \{A\}, \forall A \in V - \{S_1\}$
  - omit the short rules: $S_1 \rightarrow)$, $S \rightarrow S$
  - add new rule: $S \rightarrow ()$ because of $S \rightarrow (S_1$
- the grammar in Chomsky form is: $S \rightarrow SS$, $S \rightarrow (S_1$, $S_1 \rightarrow S)$, $S \rightarrow ()$

Example

- If the rules would have been S → A, A → B, B → C, B → E, F → AB, F → bB then
  - D(S) = {S, A, B, C, E}
  - D(A) = {A, B, C, E}
  - D(B) = {B, C, E}
  - D(F) = {F}

Example

- If the replacement sets would have been D(A) = {A, B, D}, D(B) = {B, D}, and there is a rule S → AB
  - the new rules would be S → AB, S → AD, S → BB, S → BD, S → DB, S → DD

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Lemma: the next algorithm determines all N[i, i + s]
  - the algorithm is based on dynamic programming
    - first it determines the smallest sub-problems: N[1, 1], N[2, 2], ...
    - then gradually the bigger ones using all the previous results (dynamic programming)

- Definition of N[i, i + s]: $\{A \in V : A =>^* x_i \ldots x_{i+s}, x_j \in V\}$
  - given CFG G in Chomsky normal form
  - given string $x = x_1 x_2 \ldots x_n$, $n \geq 2$
  - N[i, i + s] is the set of symbols which can derive in G string $x_i \ldots x_{i+s}$

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

```
for i:= 1 to n do N[i, i] := {x_i}
for s:= 1 to n – 1 do
  for i:= 1 to n – s do
     for k:= i to i + s – 1 do
          if ∃ A → BC ∈ R, B ∈ N[i, k],
  C ∈ N[k+1, i+s] then
               add A to N[i, i+s]
Legend:
    s: the size of the current problem - 1
    i: the start of the first subproblem
    k: the end of the first subproblem
    first subproblem: N[i, k]
    second subproblem: N[k+1, i+s]
```

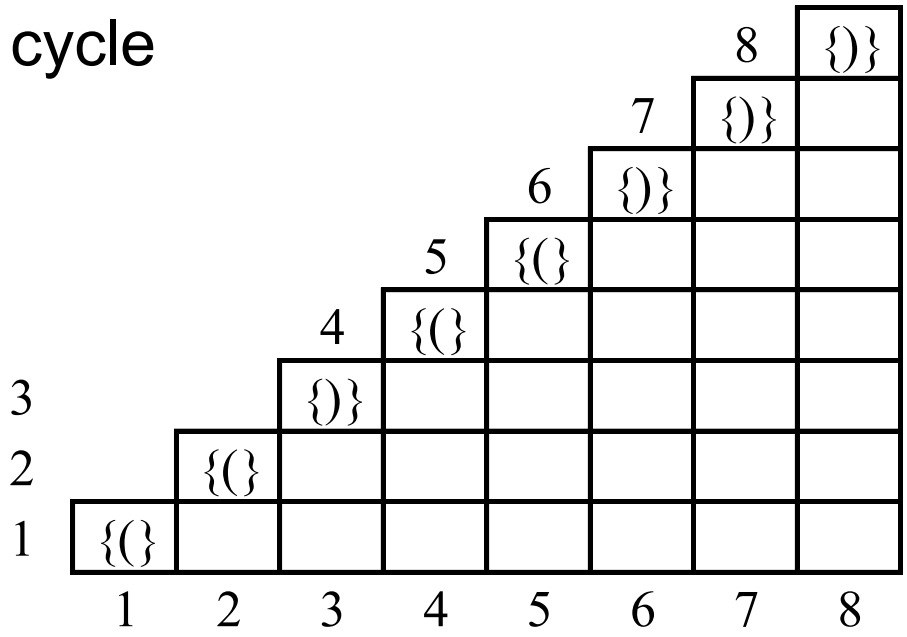
- For a given G the complexity is $O(n^3)$
  - calculation of the if depend on |R| and |Σ|
- Main idea:
  - if: $A \rightarrow BC$, $B =>^* x_i \ldots x_k$, $C =>^* x_{k+1} \ldots x_{i+s}$
  - then: $A =>^* x_i \ldots x_{i+s}$
    - $A => BC =>^* x_i \ldots x_k C =>^* x_i \ldots x_k x_{k+1} \ldots x_{i+s}$

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Theorem:
  - given:
    - CFG G in Chomsky normal form
    - string $x = x_1x_2 \ldots x_n$, $n \geq 2$
    - the N[i, i + s] sets
  - $x \in L(G) \leftrightarrow S \in N[1, n]$
- Proof: according to the definition of N[i, i + s]
  if $S \in N[1, n] \rightarrow S \Rightarrow^* x_1x_2 \ldots x_n = x$

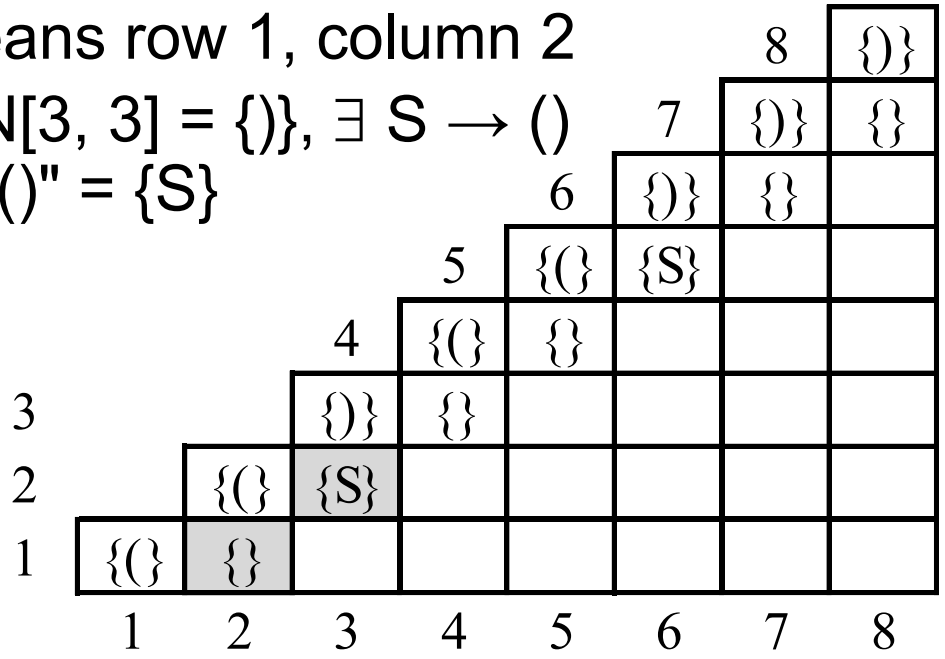
Example

- Decide if ((())(())) can be generated by the next grammar!
  - the CFG generates the set of balanced parenthesis
    - R = {S → SS, S → (S_1, S_1 → S), S → ()}
  - perform the initial cycle



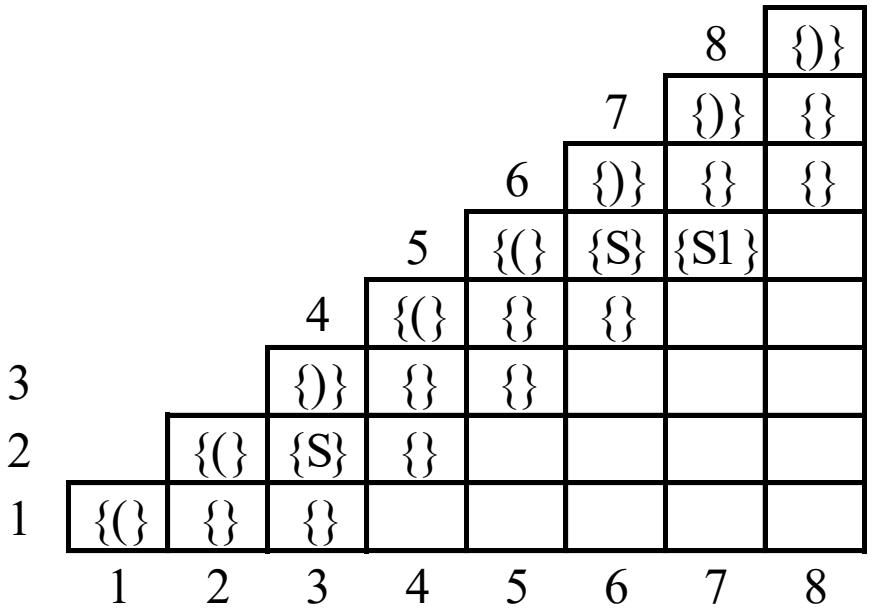
Example

– calculate the s = 1 iteration
  - N[1, 1] = "(" = {(}, N[2, 2] = "(" = {(}, there is no rule such A → ((, A ∈ V → N[1, 2] = "((" = {}
    – N[1, 2] means row 1, column 2
  - N[2, 2] = {(}, N[3, 3] = {)}, ∃ S → () → N[2, 3] = "()" = {S}



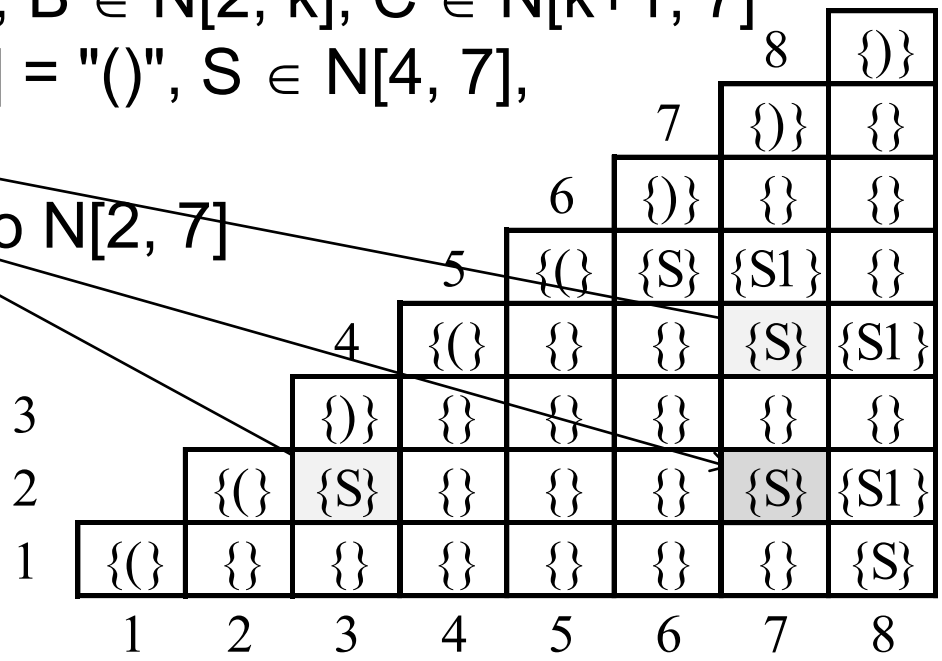
Example

- calculate the table one diagonal after another



Example

- a string can be divided in different ways, e.g. to calculate N[2, 7] = "()(())"
- examine each partition N[2, k], N[k + 1, 7], 2 ≤ k < 7
- check if ∃ A → BC, B ∈ N[2, k], C ∈ N[k+1, 7]
  k = 3, S ∈ N[2, 3] = "()", S ∈ N[4, 7],
  ∃ S → SS,
  so S is added to N[2, 7]



- S ∈ N[1, 8] so the (()(())) can be generated in G

Deterministic PDA

- Definition of consistent strings, x and y: x is a prefix of y or y is a prefix of x
- Definition of compatible transitions ((p, a, β), (q, γ)) and ((p, a', β'), (q', γ')): 'a' is consistent with a' and β is consistent with β'
  - $\exists$ such a configuration when both transitions are applicable
  - in configuration (q, aabba, xyx) both ((q, aa, x), (p, y)) and ((q, a, xy), (p, *e*))
- Definition of deterministic PDA: it has no two distinct compatible transitions
  - it may have *e* transition

Deterministic context-free language

- Definition of deterministic context-free language L: L$ = L(M) for some deterministic PDA M
  - $ is appended to each input string, it marks the end of string, $ $\notin \Sigma$
  - L is recognized by such a deterministic PDA M which can sense the end of the input string
- Theorem: every deterministic context-free language is a context-free language

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- If the end of string is not marked → L = a* ∪ $a^n b^n$ can not be recognized by a deterministic PDA
  - the number of 'a' must be counted if we want to check the number of b by pushing symbols into the stack
  - on the other hand a* must be also accepted
    - if $ is met then the stack can be cleared

Deterministic context-free language

- We suppose that not every context-free language is a deterministic context-free language
  - e.g.: L = $\{a^n b^m c^p : m, n, p \geq 0, n \neq m \text{ or } m \neq p\}$
  - L is context-free
  - L can be deterministic context-free language if we know in advance that the 'a' or the 'b' is to be counted

- Definition of dead end configuration of PDA M, (q, ω, α): for each configuration (q', ω', α') which follows (q, ω, α), (q, ω, α) $|-_M^+$ (q', ω', α'), the followings are true: ω' = ω and |α'| ≥ |α|
  - no progress can be made starting from dead end in terms of either reading the input or reducing the stack
  - M executes a never-ending sequence of *e*-moves

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- If M does not have dead end then
  - for any configuration we can find such a following configuration which has less input or less stack
  - the procedure is finite as the input and the number of elements in the stack are finite

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Theorem: the class of deterministic context-free languages is closed under complement
- Proof by construction:
  - let M = (K, Σ, Γ, Δ, s, F) is any deterministic PDA
  - transform M to M', such that
    - L(M) = L(M')
    - M' is a simple PDA (and still deterministic)

- M' may reject the input if
  - the input is read and M' is not in final state
  - the input is read and the stack is not empty
  - M' is stuck as there is no applicable transition
  - M' reaches a dead end (does not read input)

- the dead ends of a simple, deterministic PDA can be determined
  - not detailed here, how exactly
- transform M' to M'', such that
  - L(M') = L(M'')
  - M'' have some new, non-final states
  - M'' moves into the new states and empty the stack whenever M' would reject the input based on condition 2, 3, or 4
  - M'' has no dead end
- create M''' by inverting F''
  - $L(M''') = L(M'')^C = L(M')^C = L(M)^C$

- Theorem: the class of deterministic context-free languages is properly contained in the class of context-free languages

Deterministic context-free language

- Proof by contradiction:
  - suppose $L = \{a^n b^m c^p : m, n, p \geq 0, n \neq m \text{ or } m \neq p\}$ is deterministic context-free
  - $L^C = \{a^n b^m c^p : m, n, p \geq 0, n = m = p\} \cup \{w \notin a^*b^*c^*\}$ is deterministic context-free according to the previous theorem
  - $L^C \cap \{w \in a^*b^*c^*\}$ is context-free according to a former theorem (context-free $\cap$ regular = context free)
  - but $L^C \cap \{w \in a^*b^*c^*\} = \{w \in a^n b^n c^n\}$ is not context-free

Deterministic context-free language

- Interesting:
  - DFA is equivalent with NFA
  - PDA is not equivalent with deterministic PDA
  - Turing-machine is equivalent to nondeterministic Turing-machine

Introduction to the Theory of Computation

Lesson 4

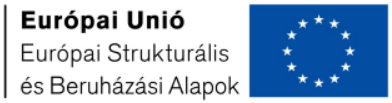
3.7. Parsing

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA



BEFEKTETÉS A JÖVŐBE

Top-down parsing

- More information in the course: Compilers
- Definition of parsing: given some grammar G, use a deterministic automaton (parser)
  - to decide if $w \in L(G)$
    - see previous lesson
  - if $w \in L(G)$, give the derivation of w

Top-down parsing

- Top-down parsing constructs the leftmost derivation of w
  - start the construction from the root
- CFGs and deterministic PDAs are considered here
- Our aim is to construct a deterministic PDA for a CFG

Example 1

- Given CFG G = ({a, b, S}, {a, b}, {S → aSb, S → *e*}, S) construct a parser to it!
  - L(G) = {$a^n b^n$}
  - construct a PDA equivalent with G
    - PDA M_1 = ({p, q}, {a, b}, {a, b, S}, Δ_1, p, {q})
      - Δ_1 = {((p, *e*, *e*), (q, S)), ((q, *e*, S), (q, aSb)), ((q, *e*, S), (q, *e*)), ((q, a, a,), (q, *e*)). ((q, b, b), (q, *e*))}
    - M_1 is not deterministic because in configuration (q, abb, Sb) both transition 2 and 3 are applicable

Lookahead symbol

- Idea of lookahead:
  - non-determinism can be resolved by reading 1 character from the input before decision
  - at this example if
    - next symbol: 'a' then use S → aSb
    - next symbol: b then use S → *e*

Lookahead symbol

- in general
  - read 1 symbol without consulting the stack
  - go to a new state corresponding to the read symbol
    - q_a is a state coming from state q after reading 'a'
  - with the new states a deterministic PDA M_2 can be constructed such that $L(M_1) = L(M_2)$

Deterministic PDA

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- $M_2 = (\{p, q, q_a, q_b, q_\$\}, \{a, b\}, \{a, b, S\}, \Delta_2, p, \{q_\$\})$
  - Δ_2 contains the following transitions:
    - (1) ((p, *e*, *e*), (q, S))
    - (2) ((q, a, *e*), (q_a, *e*)) // new
    - (3) ((q_a, *e*, S), (q_a, aSb)) // S is top of the stack
    - (4) ((q_a, *e*, a), (q, *e*))
    - (5) ((q, b, *e*), (q_b, *e*)) // new
    - (6) ((q_b, *e*, S), (q_b, *e*)) // S is top of the stack
    - (7) ((q_b, *e*, b), (q, *e*))
    - (8) ((q, $, *e*), ($q_\$$, *e*)) // new
      - we can decide between 3 and 6 based on the state (which is based on the lookahead)

Operation

Step	State	Unread input	Stack	Transition used	Rule
0	p	ab$	*e*	-	
1	q	ab$	S	1	
2	q_a	b$	S	2	
3	q_a	b$	aSb	7	S → aSb
4	q	b$	Sb	3	
5	q_b	$	Sb	4	
6	q_b	$	b	8	S → *e*
7	q	$	*e*	5	
8	$q_\$$	*e*	*e*	6	

LL(1) parser

- Definition of LL(1) parser: such a deterministic PDA which is constructed to be a top-down parser and to read at most 1 input symbol at a time
- Not all CFG have a deterministic PDA equivalent
- Some have a deterministic PDA equivalent but not an LL(1) parser
  - reading ahead just 1 character may not be enough to resolve uncertainties
- Some grammar need transformations before an LL(1) parser can be constructed

Example 2

- Given CFG G that generates arithmetic expressions with operations + and *, construct the LL(1) parser!
- G = (V, Σ, R, E), where
  - V = {+ ,* , (,), id, T, F, E}
    - E - expression, T - term, F - factor
  - Σ = {+ , *, (,), id}
  - R = {E → E + T, E → T, T → T * F, T → F, F → (E), F → id, F → id(E)}
    - the last rule corresponds to functions such as sqrt(x * x + 1), f(z)

Example 2

- The PDA equivalent with G contains the following transitions:
  - (0) ((p, *e*, *e*), (q, E))
  - (1) ((q, *e,* E), (q, E + T))
  - (2) ((q, *e*, E), (q, T))
  - (3) ((q, *e,* T), (q, T * F))
  - (4) ((q, *e*, T), (q, F))
  - (5) ((q, *e*, F), (q, (E)))
  - (6) ((q, *e*, F), (q, id))
  - (7) ((q, *e*, F), (q, id(E)))
  - $((q, a, a), (q, e)) \in \Delta$ for $\forall\ a \in \Sigma$

Left factoring

- Looking ahead one symbol does not help to choose between transitions 6 and 7
  - they both can be applied in configuration (q, id(5)$, F)
- The source of the problem are the rules:
  $F \rightarrow id$, $F \rightarrow id(E)$
  - identical left side, F, and identical first nonterminal, id

Left factoring

- Construct again the PDA
  - (6') ((q, *e*, F), (q, idA))
  - (7') ((q, *e*, A), (q, *e*))
  - (8') ((q, *e*, A), (q, (E)))
  - if the lookahead = '(' then transition 8' is to be used

Left factoring

- Given CFG G
  - if it has rules
    - $A \rightarrow \alpha\beta_1 \mid ... \mid \alpha\beta_n$
      - $\alpha \neq e$, $n \geq 2$
  - then replace them by
    - $A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 \mid ... \mid \beta_n$
      - A' is a new nonterminal

Eliminating left recursion

- Looking ahead one symbol does not help to choose between transitions 1 and 2 (belonging to E → E + T | T)
  - they both can be applied in configuration (q, id+id$, E)
    - if transition 2 is selected then the PDA will stuck
    - if transition 1 is repeated 2, 3, ... times then the PDA will stuck

Eliminating left recursion

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Definition of left recursion in CFG G: G contains such a rule whose left side is the same as the leftmost element of the right side
  - at the problem in question: E → E + T
- Replace the previous rules by: E → TE', E' → +TE' | *e*
  - if the lookahead = '+' then we know which rule must be used

Eliminating left recursion

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Construct again the PDA from G' = (V', Σ, R, E)
  - (1) $E \rightarrow TE'$
  - (2) $E' \rightarrow +TE'$
  - (3) $E' \rightarrow e$
  - (4) $T \rightarrow FT'$
  - (5) $T' \rightarrow *FT'$
  - (6) $T' \rightarrow e$
  - (7) $F \rightarrow (E)$
  - (8) $F \rightarrow idA$
  - (9) $A \rightarrow e$
  - (10) $A \rightarrow (E)$

Eliminating left recursion

- Given CFG G
  - if it has rules
    - $A \rightarrow A\alpha_1 \mid ... \mid A\alpha_n$ and $A \rightarrow \beta_1 \mid ... \mid \beta_m$
      - for $\forall i$ β_i does not start with A, n > 0
    - e.g. before: $A => A\alpha_1 =>^* A\alpha_1...\alpha_1 => \beta_1\alpha_1...\alpha_1$
  - then replace them by
    - $A \rightarrow \beta_1 A' \mid ... \mid \beta_m A'$, $A' \rightarrow \alpha_1 A' \mid ... \mid \alpha_n A' \mid e$
      - A' is a new nonterminal
    - e.g. after: $A => \beta_1 A' => \beta_1\alpha_1 A' =>^* \beta_1\alpha_1...\alpha_1 A' => \beta_1\alpha_1...\alpha_1$

Top-down parsing

- From given CFG create a deterministic PDA
  - perform left factoring
  - eliminate left recursion
  - perform CFG → PDA transformation
  - introduce lookahead states
  - apply the PDA to the word
- Inspecting the stack we can see the rules creating a word
- Not every CFG can be transformed in this way

Bottom-up parsing

- Bottom-up parsing constructs the rightmost derivation of w
  - start the construction from the leaves
- There is a new method to construct a PDA equivalent with a CFG
  - given G = (V, Σ, R, S)

Bottom-up parsing

– PDA M = (K, Σ, Γ, Δ, p, F), K = {p, q}, Γ= V, F = {q}
  - Δ contains the following transitions:
    – ((p, a, *e*), (p, a)), for ∀ a ∈ Σ
      » shift the input into the stack symbol by symbol
    – ((p, *e*, α^R), (p, A)), for ∀ rule A → α in R
      » if a reverse of the right side of a rule is at the top of the stack then replace it with the corresponding left side
    – ((p, *e*, S), (q, *e*))
      » end operation by removing the start symbol
– L(G) = L(M)

Bottom-up parsing

- G = (V, Σ, R, E)
  - V = {+, *, (,), id, T, F, E}
  - Σ = {+, *, (,), id}
  - R =
    - E → E + T (R1)
    - E → T (R2)
    - T → T * F (R3)
    - T → F (R4)
    - F → (E) (R5)
    - F → id (R6)

Bottom-up parsing

- Δ of M:
  - (p, a, *e*), (p, a))for $\forall$ a $\in$ Σ (Δ0)
  - (p, *e*, T + E), (p, E)) (Δ1)
  - (p, *e*, T), (p, E)) (Δ2)
  - (p, *e*, F * T), (p, T)) (Δ3)
  - (p, *e*, F), (p, T)) (Δ4)
  - (p, *e,*)E(), (p, F)) (Δ5)
  - (p, *e*, id), (p, F)) (Δ6)
  - (p, *e*, E), (q, *e*)) (Δ7)

step	state	Unread input	Stack	Transition used	Rule
0	p	id*(id)	*e*		
1	p	*(id)	id	Δ0	
2	p	*(id)	F	Δ6	R6
3	p	*(id)	T	Δ4	R4
4	p	(id)	*T	Δ0	
5	p	id)	(*T	Δ0	
6	p	)	id(*T	Δ0	
7	p	)	F(*T	Δ6	R6
8	p	)	T(*T	Δ4	R4
9	p	)	E(*T	Δ2	R2
10	p	*e*	)E(*T	Δ0	
11	p	*e*	F*T	Δ5	R5
12	p	*e*	T	Δ3	R3
13	p	*e*	E	Δ2	R2
14	q	*e*	*e*	Δ7	

Bottom-up parsing

- The implied rightmost derivation:
  - apply the rules from end to start

E => T
=> T * F
=> T * (E)
=> T * (T)
=> T * (F)
=> T * (id)
=> F * (id)
=> id * (id)

Bottom-up parsing

- M is nondeterministic: Δ0 is compatible with all the transitions of type Δ1 through Δ8
- To make M deterministic the next conflicts must be resolved
  - shift or reduce
  - reduce or another reduce

Bottom-up parsing

- Resolving the shift-reduce conflict
- Construct precedence relation P
  - based on
    - b = next input symbol
    - a = top stack symbol
  - $P \subseteq V \times (\Sigma \cup \{\$\})$
  - the construction of P is not given here
- Decision:
  - reduce if $(a, b) \in P$
  - shift otherwise

Bottom-up parsing

- Relation P for the example, e.g.: (T, +) $\in$ P, if there is a mark in the table

	(	)	id	+	*	$	b
(							
)		√		√	√	√	
id		√		√	√	√	
+							
*							
E							
T		√		√		√	
F		√		√	√	√	
a							

Bottom-up parsing

- Resolving the reduce-reduce conflict: reduce the longest possible string from the top of the stack
- Definition of weak precedence grammars: the bottom-up parser corresponding to this grammar is deterministic if the shift-reduce and reduce-reduce conflicts are resolved

Introduction to the Theory of Computation

Lesson 5

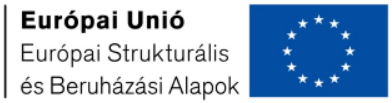
4.3. Extensions of Turing machines

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA



BEFEKTETÉS A JÖVŐBE

Must be known

- Definition of:
  - Turing machine
  - basic machines
  - Machine schema (variable like notation)
  - simple machines (copy, delete, shift, search)

Turing machines

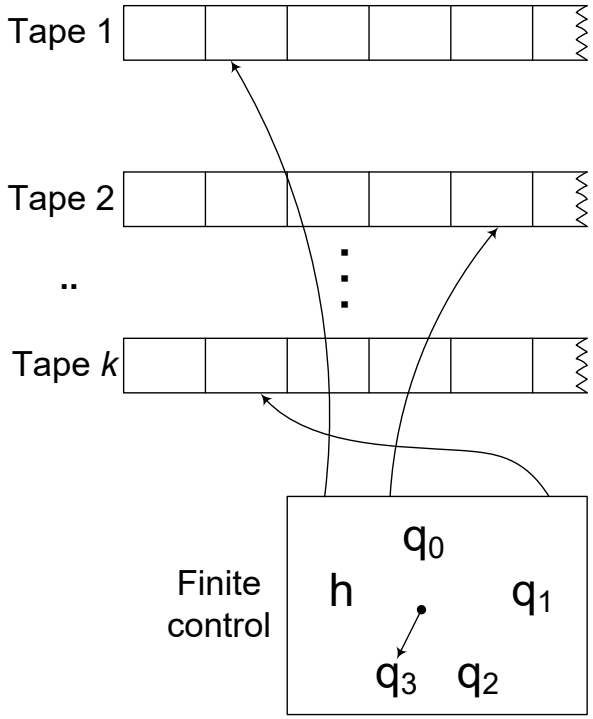
- Turing machines can be extended
- The extensions:
  - do not increase computational power
    - we will see an extended TM that can be simulated with a conventional TM
  - do increase speed and ease of use

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Possible extensions of the Turing machine:
  - multiple tapes
  - two-way infinite tapes
  - two-dimensional tapes
  - multiple heads
  - random access
  - nondeterministic TM

Multiple tape TM

- k-tape TM: a TM equipped with k tapes and corresponding heads ($k \geq 1$)
  - each tape is connected to the finite control by means of a read/write head



Multiple tape TM

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Definition of k-tape TM: a quintuple $(K, \Sigma, \delta, s, H)$, where
  - $k \in \mathbb{N}, k \geq 1$
  - K set of states (finite)
  - Σ alphabet (finite)
    - containing $\sqcup$, $\triangleright$, not containing $\leftarrow$, $\rightarrow$
  - $s \in K$ the initial state
  - $H \in K$ the set of halting states (finite)

Multiple tape TM

- δ transition function, $(K-H) \times \Sigma^k \rightarrow K \times (\Sigma \cup \{\leftarrow, \rightarrow\})^k$
  - $\forall\ q \in K - H, \forall\ (a_1, ..., a_k) \in \Sigma^k$, $\delta(q, (a_1, ..., a_k)) = (p, (b_1, ..., b_k))$
    - p: the new state
    - b_j: the action taken by M at tape j
  - if $\exists$ j such that $a_j = \triangleright \rightarrow b_j = \rightarrow$
  - there is no j such that $b_j = \triangleright$
  - in a single step for each tape the actual symbol can be modified or the head can be moved

Multiple tape TM

- Definition of the configuration of a k-tape TM: an element of $K \times (\triangleright\Sigma^* \times (\Sigma^*(\Sigma - \{\sqcup\}) \cup \{e\}))^k$
  - identify the state, the tape contents, and the head positions for each tape

Multiple tape TM

- The initial configuration
  - contains the input on the 1st tape the same way as on simple TM
  - the other tapes are empty with heads on the leftmost blank position
- The output of a k-tape TM is stored on the first tape

Example

- Construct the copying machine with a 2-tape TM:
  - transform ▷␣w␣ into ▷␣w␣w␣, w ∈ {a, b}*
- Operation:
  - start:
    - first tape: ▷␣w, second tape: ▷␣

Example

– move the heads on both tapes to the right, copying each symbol from the first tape onto the second tape
  - the first square of the second tape should be left blank
  - first tape: ▷␣w␣, second tape: ▷␣w␣
  - part of δ: $\delta(q_1, a, ␣) = (q_2, a, a)$, $\delta(q_2, a, a) = (q_1, \rightarrow, \rightarrow)$

Example

  - move the head on the second tape to the left until blank is found
    - first tape: ▷␣w␣, second tape: ▷␣w
  - move the heads on both tapes to the right copying symbols from the second tape onto the first tape
    - first tape: ▷␣w␣w␣, second tape: ▷␣w␣
- It is easier to implement C (the copy machine) on a 2-tape machine and the operation is quicker

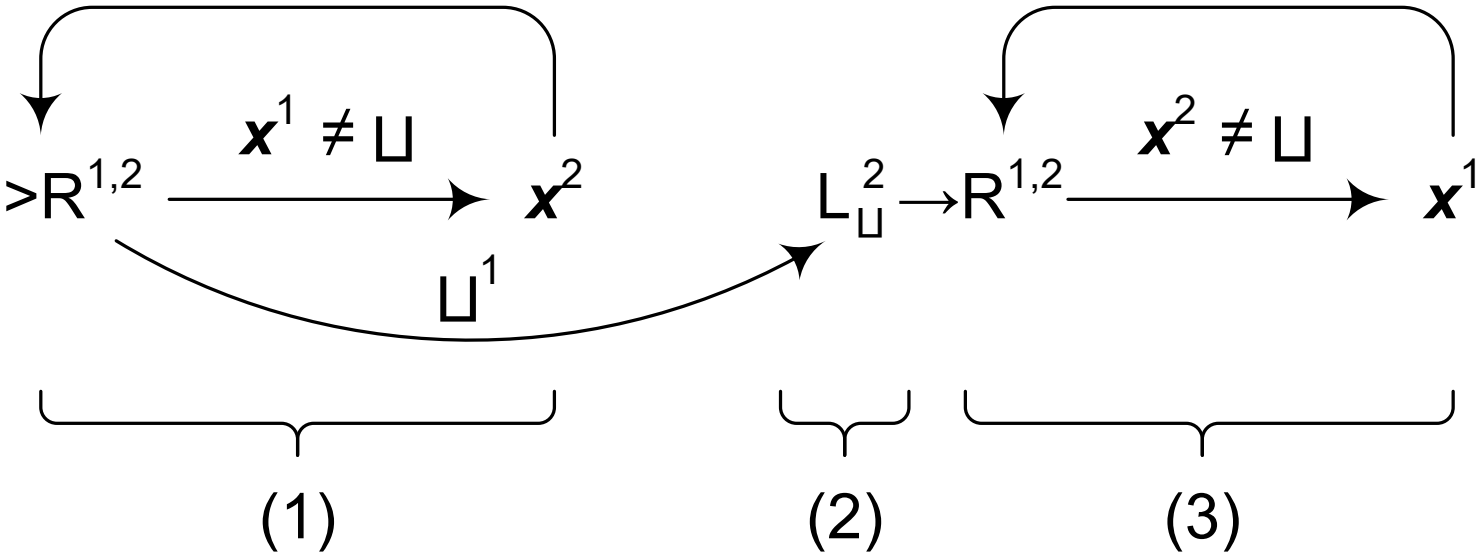
Example

- We use the following conventions to depict a k-tape TM:
  - a superscript is attached to each sub-TM denoting the tape on which it operates
    - e.g.:
      - $␣^2$ writes a blank on the second tape
      - $L_␣^1$ searches to the left for a blank on the first tape
      - $R^{1,2}$ is a shorthand for R^1R^2
    - beware: R^2 meant RR before

Example

- a superscript is attached to the symbols on the arrows
  - e.g.: a^1 means "follow this arrow if the previous sub-TM finished and 'a' is under the head on the first tape"
  - a pair of bits, e.g. 01, on an arrow label is a shorthand for $a^1 = 0$, $a^2 = 1$

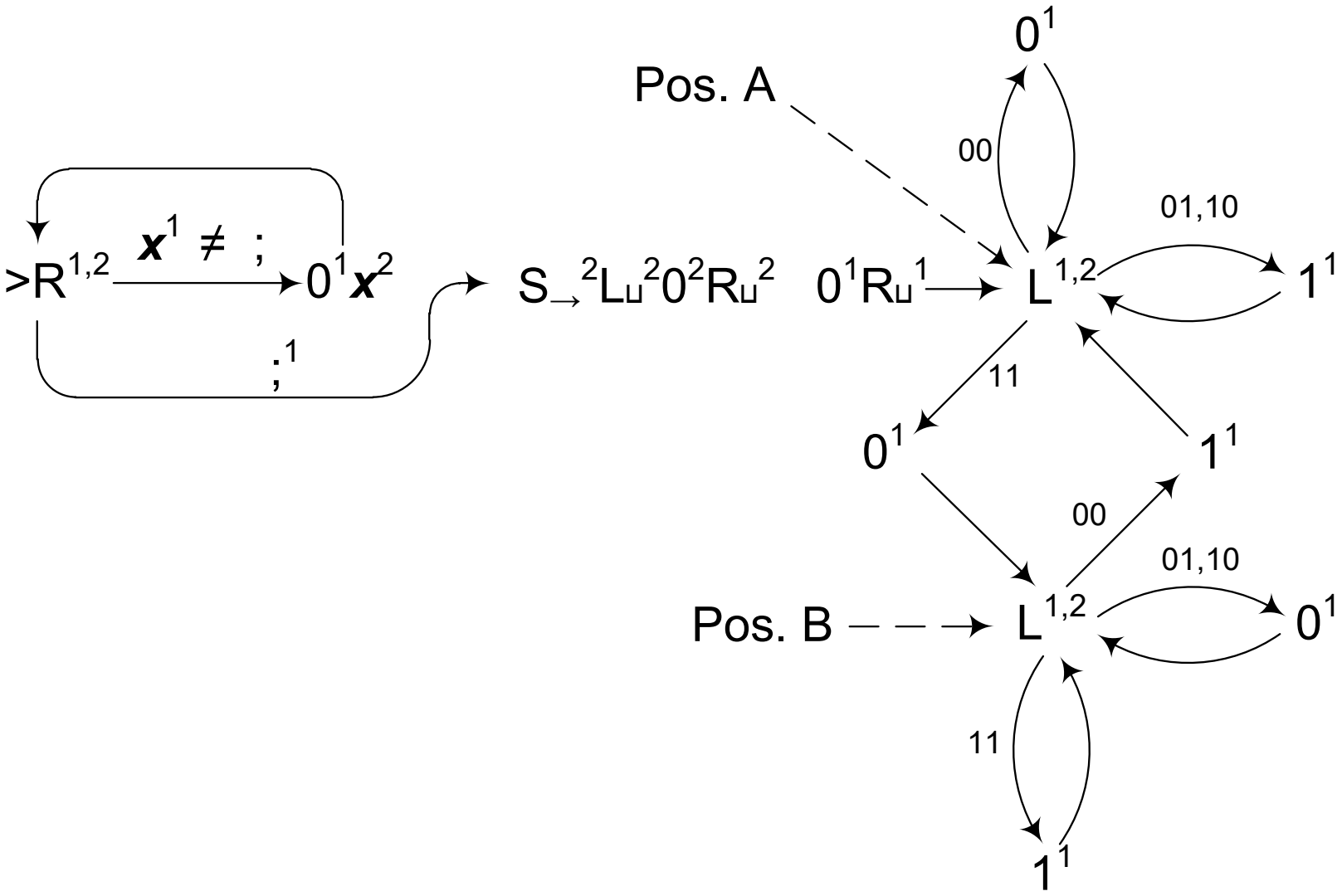
Example



Example

- Construct a machine to add two binary numbers with a 2-tape TM:
  - $n_i \in N$, $w_i = \text{numBin}(n_i)$, $i = 1, 2, 3$, $n_3 = n_1 + n_2$
  - transform $\triangleright \sqcup w_1 ; w_2 \sqcup$ into $\triangleright \sqcup 0^* w_3 \sqcup$, $w_i \in \{0, 1\}^*$

Example


>R1,2
x1 ≠ ;
01x2
;1
S→2L␣20202R␣2
01R␣1
Pos. A
L1,2
01
00
01,10
11
11
Pos. B
L1,2
01
00
01,10
11
11

Example

- Operation:
  - start:
    - first tape: $\triangleright\underline{\sqcup}w_1;w_2\sqcup$, second tape: $\triangleright\underline{\sqcup}$
  - copies the first binary integer in its second tape writing zeros in its place
    - and in the place of the ';' separating the two integers

Example

- first tape: $\triangleright \sqcup 0^* w_2 \underline{\sqcup}$, second tape: $\triangleright \sqcup 0 w_1 \underline{\sqcup}$
  - $S_{\rightarrow}{}^2 L_{\sqcup}{}^2 0^2 R_{\sqcup}{}^2$: $\triangleright \sqcup w_1 \underline{\sqcup} \rightarrow \triangleright \sqcup 0 w_1 \underline{\sqcup}$
    - » the adding stops when one of the numbers is finished, now the carry bit will be written

Example

- performs binary addition by the "school method", starting from the least significant bit of both integers
  - writes the result in the first tape
  - sub-machines indicate if there is a carry bit
    - » there is carry bit at the lower $L^{1,2}$ machine
  - first tape: $\triangleright \sqcup 0^* w_3 \underline{\sqcup}$, second tape: $\triangleright \sqcup w_1 \underline{\sqcup}$

Example

- T1: ▷␣010;011, T2: ▷␣
- T1: ▷␣0000011␣, T2: ▷␣010␣ // Pos. A
- T1: ▷␣0000011␣, T2: ▷␣010␣ // Pos. A
- T1: ▷␣0000011␣, T2: ▷␣010␣
- T1: ▷␣0000011␣, T2: ▷␣010␣ // Pos. A
- T1: ▷␣0000001␣, T2: ▷␣010␣
- T1: ▷␣0000001␣, T2: ▷␣010␣ // Pos. B
- T1: ▷␣0000101␣, T2: ▷␣010␣
- T1: ▷␣0000101␣, T2: ▷␣010␣ // Pos. A

Adding tapes adds no power

- Theorem:
  - let $M = (K, \Sigma, \delta, s, H)$ is a k-tape TM, $k \geq 1$
  - $\exists$ a standard TM $M' = (K', \Sigma', \delta', s', H')$, $\Sigma \subseteq \Sigma'$ such that:
    - M halts on input x with output y on its first tape $\leftrightarrow$ M' halts on input x at the same halting state and with the same output
    - if on input x M halts after t steps $\rightarrow$ M' halts after $O(t^*(|x|+t))$ steps
  - k-tape TM and standard TM are equivalent

Corollary from the k-tape Turing machine

- Proof: in the book
  - idea: each cell of tape of M' contains (2k-tuples)
  - it means $\Sigma' = (\Sigma \times \{0, 1\})^k$
  - {0, 1} is needed to sign the head position of a head

Corollary from the k-tape Turing machine

- Corollary: Any function that is computed or language that is decided or semidecided by a k-tape Turing machine is also computed, decided, or semidecided, respectively, by a standard Turing machine

Two-way infinite tape TM

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- The tape is infinite in both directions
  - there is no ▷ symbol
- All squares are initially blank except those containing the input, tape: ␣w

Two-way infinite tape TM

- It can be simulated by a 2-tape machine:
  - tape 1 contains the part of the tape to the right of the square containing the first input symbol
  - tape 2 contains the part of the tape to the left of the square containing the first input symbol in reverse order
- The simulation needs only linear (and not quadratic) time because in each step only one tape is active

Multiple heads TM

- TM has a single tape and multiple heads
- In one step, the heads all sense the scanned symbols and move or write independently
  - some convention must be used what happens if two heads try to modify the same symbol

Multiple heads TM

- Idea for simulation:
  - the tape is divided into tracks
    - one corresponds to the real tape
    - the others contain only the head positions
- The simulation needs quadratic time

Example

- Construct a 3-heads TM to decide $\{a^n b^n c^n : n \in N\}$!
  - position head 1 to the first 'a', head 2 to the first b, head 3 to the first c
  - enter a loop which moves all head to the right until head 1 reads 'a', head 2 reads b, head 3 read c
  - if at the same time head 1 reads b, head 2 reads c, and head 3 reads ⊔ → the input is accepted

Example

- Construct a 2-heads TM to implement the copying machine!
  - position head 1 to the first input symbol, head 2 to the second ⊔ after the input
  - move both heads to the right while head 2 writes that head 1 reads

Two-dimensional tape TM

- TM store data on a two-dimensional grid instead of a tape
  - infinite to the right and to the bottom
  - there is end of tape markers on the left and the top sides
  - remember both N and N^2 are countably infinite
- The input is placed into the second row
- The head can move on in four different directions
- The simulation needs polynomial time

Two-dimensional tape TM

- The different extension can be combined
- Example:
  - k-tape TM with multiple heads each
  - multiple 2-way infinite tape TM
  - TM with 4 dimensional tape

Theorem for extensions of the TM

- Theorem: Any language decided or semidecided, and any function computed by Turing machines with several tapes, heads, two-way infinite tapes, or multi-dimensional tapes, can be decided, or computed, respectively, by a standard Turing machine

Introduction to the Theory of Computation

Lesson 6

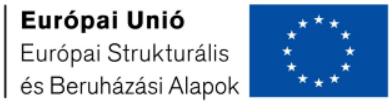
4.3. Random access Turing machines

Dr. István Heckl

Istvan.Heckl@gmail.com

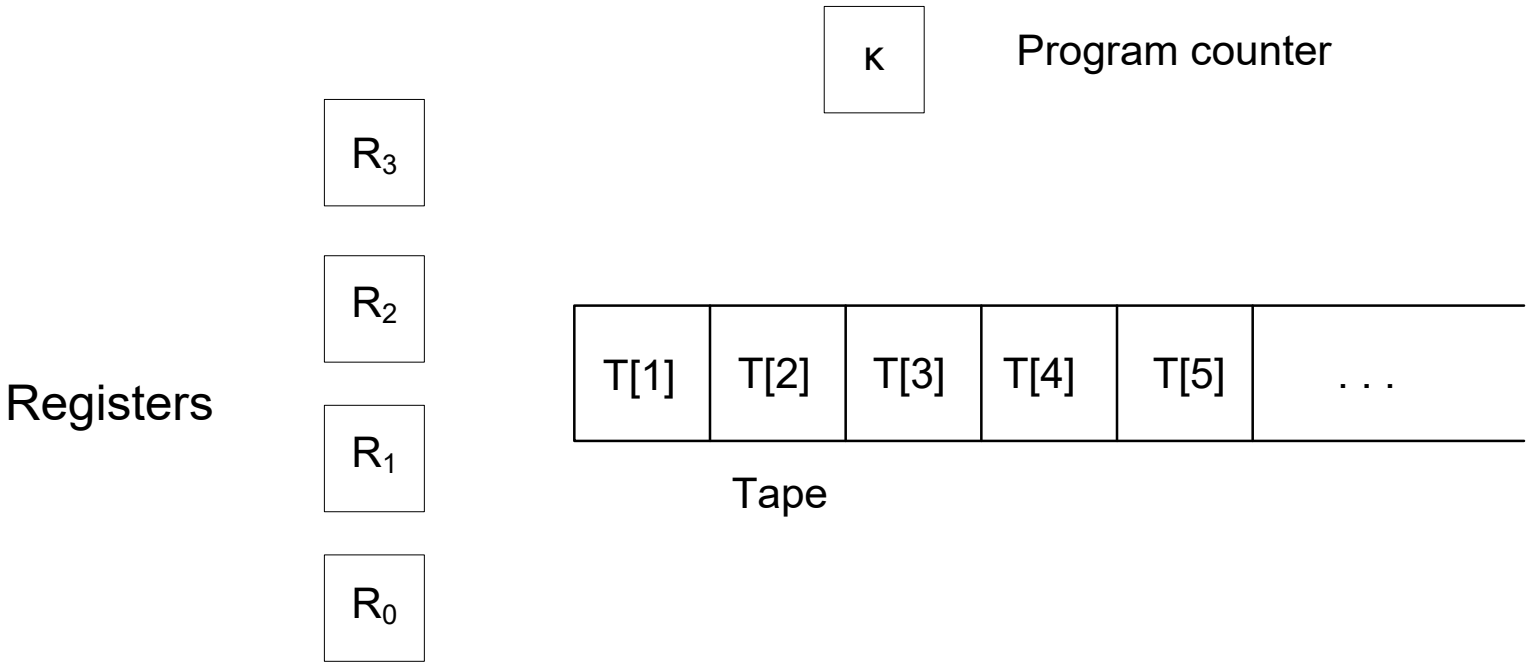
University of Pannonia

MAGYARORSZÁG KORMÁNYA



BEFEKTETÉS A JÖVŐBE

Random access TM



Random access TM

- The TM and its previous enhancements had a common drawback their memories (tapes) are sequential
  - the head must be moved square by square to the desired location before a symbol can be read or written
  - real life computer has random access memory

Random access TM

- A random access TM has:
  - a fixed number of registers
    - initially they contain 0s
  - a one-way infinite tape
    - the registers and the tape squares can be accessed directly
    - the registers and the tape squares can contain natural numbers
    - the tape contains the input

Random access TM

- a program containing a finite sequence of instructions
- a program counter showing which instruction is to be executed next

Instructions

Instruction	Operand	Semantics
read	j	$R_0 := T[R_j]$
write	j	$T[R_j] := R_0$
load	j	$R_0 := R_j$
load	=c	$R_0 := c$
store	j	$R_j := R_0$
add	j	$R_0 := R_0+R_j$
add	=c	$R_0 := R_0+c$
sub	j	$R_0 := \max\{R_0 – R_j, 0\}$

Instructions

Instruction	Operand	Semantics
sub	=c	$R_0 := \max\{R_0 - c, 0\}$
half		$R_0 := \lfloor R_0 / 2 \rfloor$
jump	s	$\kappa := s$
jpos	s	if $R_0 > 0$ then $\kappa := s$
jzero	s	if $R_0 = 0$ then $\kappa := s$
halt		$\kappa := 0$

Instructions

- T[i] denotes the current contents of tape square i
- R_j denotes the current contents of register j, $0 \leq j < k$
  - R_0, called accumulator, has a special role, all arithmetic and logical operations use this register

Instructions

- s denotes any instruction number in the program
- $c \in N$
- All instructions change κ to κ+1, unless explicitly stated otherwise

Random access TM

- Definition of a random access TM: a pair M = (k, Π), where:
  - k >0, the number of registers
  - Π = (π_1, π_2,…, π_p), the program, which is a finite sequence of instructions
    - we assume that π_p = halt, though the program may contain other halt instructions

Configuration

- Definition of the configuration of a random access TM: a (k+2)-tuple of the current value of the program counter, the registers, and the tape
  - denoted by $C = (\kappa, R_0, R_1, \ldots, R_{k-1}, T)$
  - $C \in \{0, 1, \ldots p\} \times N^k \times T$

Configuration

- if $\kappa = 0 \rightarrow$ halted configuration
- $T \subseteq \{N-\{0\}\} \times \{N-\{0\}\}$, there are no pairs (i, m), (j, n) such that i = j
  - $(i, m) \in T \leftrightarrow T[i] = m$
  - if T[i] is not defined $\rightarrow$ T[i] = 0

Yields in one step

- Definition of the yields in one step relation of a random access TM, $|-_M$: a relation between two "neighboring" configurations
  - $C = (\kappa, R_0, R_1, \ldots, R_{k-1}, T)$, $C' = (\kappa', R_0', R_1', \ldots, R_{k-1}', T')$
  - if the difference in the values of κ and κ', R_i and R_i', T and T' is exactly the difference caused by $\pi_\kappa \rightarrow C |- C'$

Yields in one step

– e.g.:
  - if π_κ = read j $\rightarrow$ $R_0' = T[R_j]$, $\kappa' = \kappa+1$, the other components are unchanged
  - if π_κ = add =c $\rightarrow$ $R_0' = R_0+c$, $\kappa' = \kappa+1$, the other components are unchanged
  - and so on

Algorithmic notation

- Reading and writing the program of a random access TM is extremely cumbersome
- The conventional statements of an algorithm can be translated into a sequence of instructions
  - e.g.: $R_1 := R_1 + R_2 - 1$ is an abbreviation of:
    1. load 1
    2. add 2
    3. sub =1
    4. store 1
  - the registers can be renamed to enhance readability
    - let $R_1 := x$; $R_2 := y \rightarrow x := x + y - 1$

Algorithmic notation

```
while x > 0 do
        x := x-3
```

– is an abbreviation of: (R_1 := x)

1. load 1
2. jzero 6
3. sub =3
4. store 1
5. jump 1
6. halt

– if labels are introduced before the important instructions → instruction numbers can be omitted

Example

- Construct a random access TM to implement instruction mply j, which performs $R_0 := R_0 * R_j$!
  - suppose R_0 initially contains x and R_1 initially contains y, after TM will halt R_0 will contain x*y
    - the instructions to move x and y from the tape to the registers is neglected now
- This new instruction can be used later as a function
  - if mply j is given then it is easy to devise mply =c

Example

- Multiplication is done by successive additions
- The simple algorithm needs O(y) iterations
  - w contains the result

```
w := 0
while y > 0 do
begin
    w := w + x
    y := y - 1
end
halt
```

Example

- Idea: perform the multiplication using "smaller" multiplications
  - e.g.: $345*247 = 345*(2*10^2 + 4*10^1 + 7*10^0) = 2*345*10^2 + 4*345*10^1 + 7*345*10^0$
    - $7 = 247 \bmod 10$
    - $4 = \lfloor 247/10 \rfloor \bmod 10$
    - $2 = \lfloor 247/100 \rfloor \bmod 10$

```
345*247
690
1380
 2415
-----
85215
```

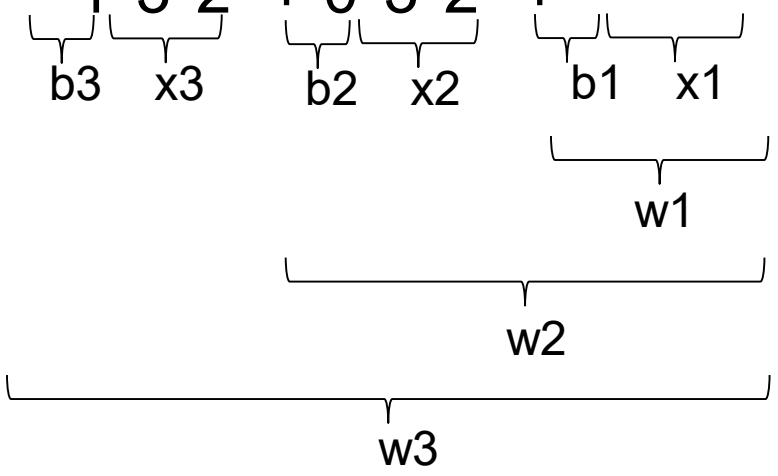
Example

- School method for multiplication
  - w contains the result
  - it needs O(log y) step
  - contains invalid instructions

```
w := 0
for each digit k of y_orig
begin
   b := ⌊y_orig / 10^(k-1)⌋ mod 10
   x := x_orig 10^(k-1)
   w := w + bx
end
halt
```

Example

- Idea: imagine y as a binary number
  - e.g.: $x_{orig} = 3$, $y_{orig} = 5 = 101_2$
  - $3*5 = 3*(1*2^2 + 0*2^1 + 1*2^0) = 1*3*2^2 + 0*3*2^1 + 1*3*2^0$



Example

- The advanced algorithm using binary representation
  - it contains only valid insturctions

```
w := 0
while y > 0 do
begin
   z := half(y)
   if y − z − z ≠ 0 then w := w + x
   x := x + x
   y := z
end
halt
```

```
w := 0
for each digit k of yorig
begin
   b := ⌊yorig / 10^(k-1)⌋ mod 10
   x := xorig10^(k-1)
   w := w + bx
end
halt
```

Example

- calculating b_k (the k-th least significant bit of y):
  - `z := half(y), if y - z - z ≠ 0`
  - e.g.: $\lfloor y_{orig} / 2^{k-1} \rfloor \bmod 2$
    - $b_1 = \lfloor 5 / 2^{1-1} \rfloor \bmod 2 = 5 - 2 - 2 = 1$ (last bit)
    - $b_2 = \lfloor 5 / 2^{2-1} \rfloor \bmod 2 = 2 - 1 - 1 = 0$
    - $b_3 = \lfloor 5 / 2^{3-1} \rfloor \bmod 2 = 1 - 0 - 0 = 1$
      - » $\lfloor y_{orig} / 2^{k-1} \rfloor$ shift to the right
      - » mod 2 check the last bit
- shifting to the right
  - `y := z`

Example

- calculating $x_k = x_{orig}2^{k-1}$
  - `x := x + x`
  - e.g.:
    - $x_1 = 3*2^{1-1} = x_{orig} = 3$
    - $x_2 = 3*2^{2-1} = 3+3 = 6$
    - $x_3 = 3*2^{3-1} = 6+6 = 12$

Example

- calculating $w_k = \Sigma_k b_k x_k$
  - `if` b_k `is 1 → w := w + x`
  - e.g.:
    - $w_1 = \Sigma_1 b_k * x_k = 1*3*2^{1-1} = 1*3 = 3$
    - $w_2 = \Sigma_2 b_k * x_k = 0*3*2^{2-1} + 1*3*2^{1-1} = 0*6 + 1*3 = 3$
    - $w_3 = \Sigma_3 b_k * x_k = 1*3*2^{3-1} + 0*3*2^{2-1} + 1*3*2^{1-1} =$ $1*12 + 0*6 + 1*3 = 15$

Example

- Variables at the start of an iteration:
  - $R_1 = y$, contains $\lfloor y_{orig} / 2^{k-1} \rfloor$
  - $R_2 = x_k$, contains $x_{orig} * 2^{k-1}$
  - $R_3 = z$, contains $\lfloor y_{orig} / 2^{k} \rfloor$
  - $R_4 = w_k$, contains $\Sigma_k b_k * x_k$ (partial result)
- Initial values: $x = x_{orig}$, $y = y_{orig}$, $w = 0$, $z = 0$
- Result: w

Example

- The values of the variables in the algorithm for x = 3, y = 5

Iteration:	0	1	2	3
x:	3	6	12	24
y:	5	2	1	0
z:	0	2	1	0
w:	0	3	3	15

Example

- The advanced algorithm translated to instructions (R0 = R2 = x, R1 = y)

1. store 2
2. load 1
3. jzero 19
4. half
5. store 3
6. load 1
7. sub 3
8. sub 3
9. jzero 13
10. load 4
11. add 2
12. store 4
13. load 2
14. add 2
15. store 2
16. load 3
17. store 1
18. jump 2
19. halt

Example

- The computation of 5*3 (and not 3*5 as previously):
  - x = 5, y = 3
  - $5*3 = 5*(1*2^1 + 1*2^0) = 1*5*2^1 + 1*5*2^0$

Iteration:	0	1	2
x:	5	10	20
y:	3	1	0
z:	0	1	0
w:	0	5	15

Example

- The computation of 5*3:

  (1; 5, 3, 0, 0, 0; Ø) |- (2; 5, 3, 5, 0, 0; Ø) |-
  (3; 3, 3, 5, 0, 0; Ø) |- (4; 3, 3, 5, 0, 0; Ø) |-
  (5; 1, 3, 5, 0, 0; Ø) |- (6; 1, 3, 5, 1, 0; Ø) |-
  (7; 3, 3, 5, 1, 0; Ø) |- (8; 2, 3, 5, 1, 0; Ø) |-
  (9; 1, 3, 5, 1, 0; Ø) |- (10; 1, 3, 5, 1, 0; Ø) |-
  (11; 0, 3, 5, 1, 0; Ø) |- (12; 5, 3, 5, 1, 0, Ø) |-
  (13; 5, 3, 5, 1, 5; Ø) |- (14; 5, 3, 5, 1, 5; Ø) |-
  (15; 10, 3, 5, 1, 5; Ø) |- (16; 10, 3, 10, 1, 5; Ø) |-
  (17; 1, 3, 10, 1, 5; Ø) |- (18; 1, 1, 10, 1, 5; Ø) |-
  (2; 1, 1, 10, 1, 5; Ø) |-* (18; 0, 0, 20, 0, 15; Ø) |-
  (2; 0, 0, 20, 0, 15; Ø) |- (3; 0, 0, 20, 0, 15; Ø) |-
  (19; 0, 0, 20, 0, 15; Ø)

Encoding of the tape

- Let
  - Σ is an alphabet such that $\sqcup \in \Sigma$, $\triangleright \notin \Sigma$
  - $E: \Sigma \rightarrow \{0, 1, \ldots |\Sigma|-1\}$ is any bijection
    - $E(\sqcup) = 0$
  - $w = a_1a_2...a_n \in \Sigma^*$ is encoded on the tape as $T = \{(1, E(a_1)), (2, E(a_2)), \ldots (n, E(a_n))\}$

Encoding of the tape

- A random access TM accepts a string if the initial configuration yields a halted configuration with $R_0 = 1$
- A random access TM rejects a string if the initial configuration yields a halted configuration with $R_0 = 0$

Example

- Construct a random access TM deciding the language $\{a^n b^n c^n : n \geq 0\}$!

```
acount := bcount := ccount := 0, n := 1
while T[n] = 1 do :
  n := n +1, acount := acount +1
while T[n] = 2 do :
  n := n +1, bcount := bcount +1
while T[n] = 3 do :
  n := n +1, ccount := ccount +1
if acount = bcount = ccount and T[n] = 0 then
  load =1, halt
else load =0, halt
```

Example

- E(a) = 1, E(b) = 2, E(c) = 3
- We are using the variables acount, bcount, and ccount to stand for the number of 'a', b, and c

Random access TM

- Theorem: any recursive or recursively enumerable language, and any recursive function, can be decided, semidecided, and computed, respectively, by a random access TM
- Proof: book

Random access TM

- Theorem: any language decided or semidecided by and any function computable by a random access TM can be decided, semidecided, and computed, respectively, by a standard TM
  - the simulation needs polynomial time
- Corollary: standard and random access TM are equivalent

Introduction to the Theory of Computation

Lesson 7

4.4. Nondeterministic Turing machines

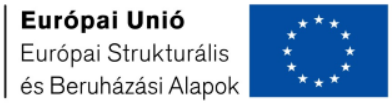
4.6. Grammars

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA



BEFEKTETÉS A JÖVŐBE

Nondeterministic TM

- Definition of nondeterministic TM: a quintuple $(K, \Sigma, \Delta, s, H)$, where
  - $\Delta \subseteq ((K - H) \times \Sigma) \times (K \times (\Sigma \cup \{\leftarrow, \rightarrow\}))$
    - it has transition relation (not function)

String accepted by nondeterministic TM

- Definition of word accepted by nondeterministic TM: $w \in (\Sigma - \{\triangleright, \sqcup\})^*$ is accepted by M if $\exists\ h \in H$, $u, v \in \Sigma^*$, $a \in \Sigma$ such that $(s, \triangleright\underline{\sqcup}w) \vdash_M^* (h, \triangleright\sqcup u\underline{a}v)$
  - at least one final configuration is reachable from the initial configuration through yield operations
  - the input may lead to a lot of non-halting configurations
- M accepts (semidecides) a language L if M accepts all words in L

Nondeterministic Turing decidable

- Definition of deciding a language L by nondeterministic TM M = (K, Σ, Δ, s, {y, n}):
  - if $\forall$ w $\in$ L, L $\subseteq$ (Σ – {␣, ▷})*
  - then
    - $\exists$ n $\in$ N depending on M and w such that there is no configuration C such that (s, ▷␣w) $|-_M^n$ C
      - the length of any computation is bounded by n

Nondeterministic Turing decidable

- $w \in L \leftrightarrow \exists\, u, v \in \Sigma^*, a \in \Sigma$ such that $(s, \triangleright \sqcup w) \vdash_M^* (y, \triangleright \sqcup u \underline{a} v)$
  - at least one accepting configuration is reachable from the initial configuration
  - the input may lead to a lot of rejecting configuration

Nondeterministic Turing computable

- Definition of computing a function f by nondeterministic TM M:
  - if $w \in L$, $L \subseteq (\Sigma - \{\sqcup, \triangleright\})^*$
  - then
    - $\exists\, n \in N$, depending on M, such that there is no configuration C such that $(s, \triangleright\underline{\sqcup}w) \vdash_M^n C$
      - the length of any computation is bounded by n
    - $(s, \triangleright\underline{\sqcup}w) \vdash_M^* (h, \triangleright\underline{\sqcup}v)$, $h \in H$, $f(w) = v$ for $\forall$ computation
      - all computations agrees on the output

Example

- Construct nondeterministic TM M to decide L = {w $\in$ {0, 1}* : w is the binary encoding of a composite number} !
- Operation of M:
  - choose nondeterministically (write 1, 0, or stop) two binary numbers 1 < p, q < w, and write them on the tape, after w, separated by ;
    - e.g.: ▷␣110011;111;1111␣␣

Example

- multiply p and q and put the answer, A, on the tape, in place of p and q
  - e.g.: ▷␣110011;1011111␣␣
- compare A and w, if equal, go to state y, else go to state n

Nondeterministic TM ↔ deterministic TM

- Theorem: if a nondeterministic TM N semidecides or decides a language or computes a function → ∃ deterministic TM D semideciding, deciding the same language or computing the same function

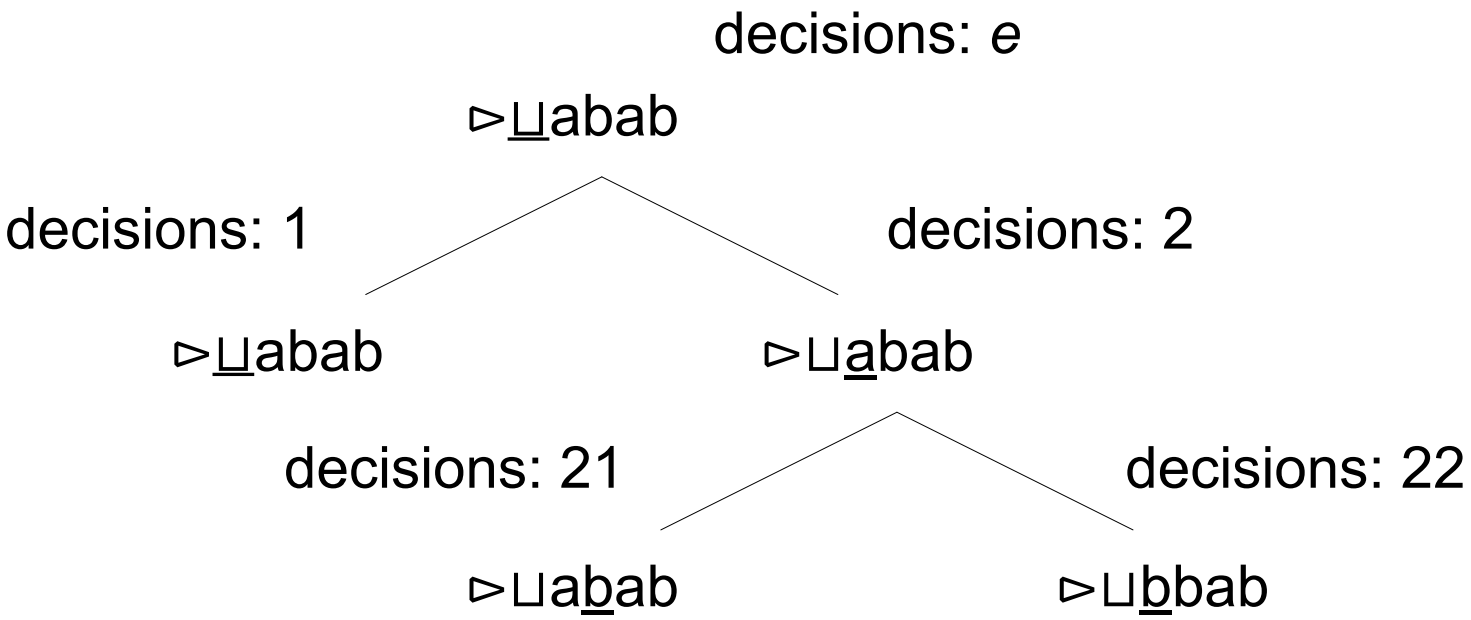
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof for semidecide:
  - construct D which simulates N by systematically trying out every possible computation of N
  - if such a computation is found which accepts the input then D also accepts the input

Computation tree

- Each configuration may lead to at most |K|*(|Σ|+2) configurations
- The configurations can be organized into a computation tree
  - the root is the initial configuration

Computation tree



Nondeterministic TM ↔ deterministic TM

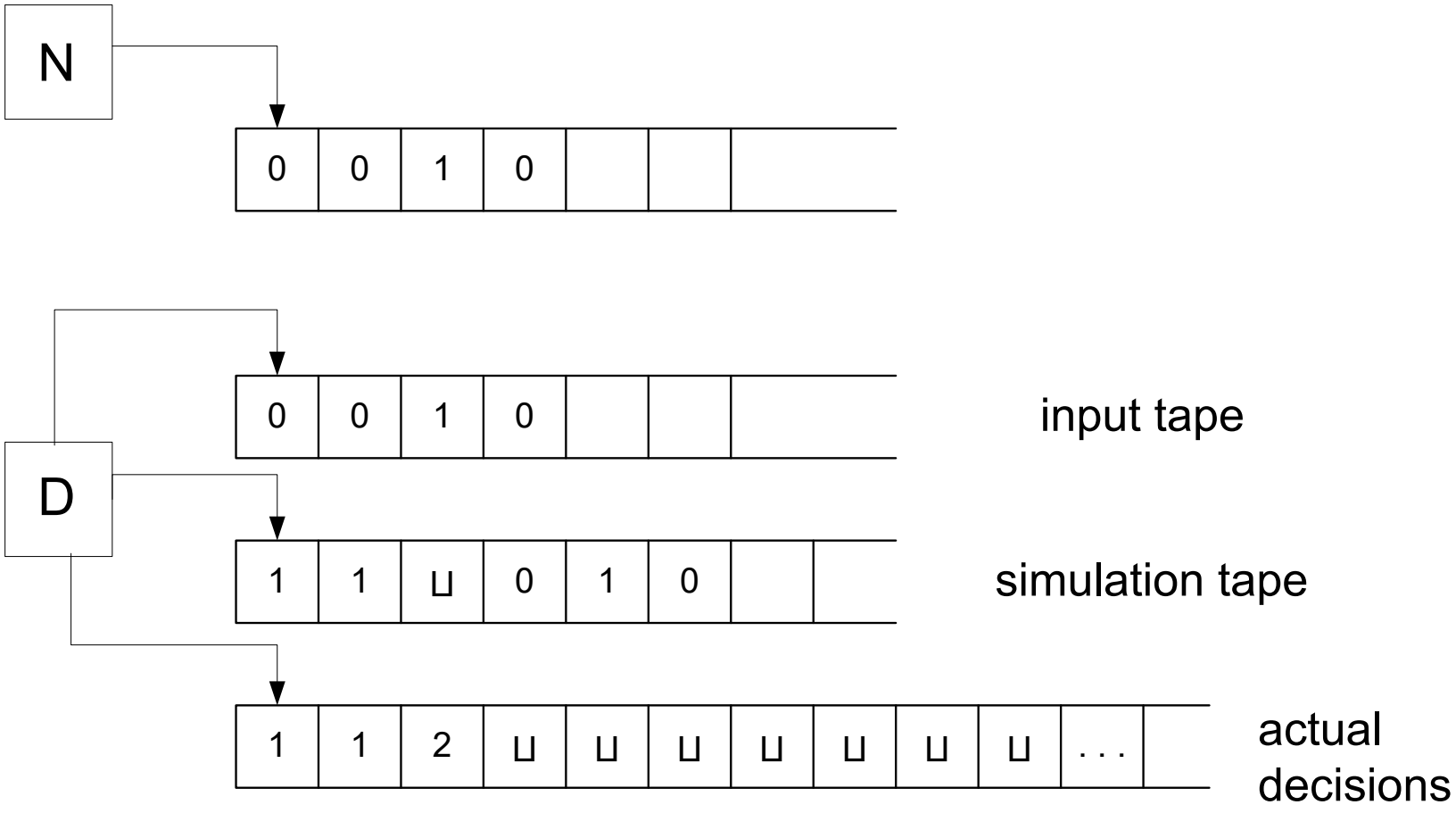
- A path of the computation tree of N represents a possible computation of D
  - a computation tree may be infinite for recursively enumerable languages but some branches are finite
  - a sequence of numbers correspond to each node representing the nondeterministic decisions made to reach that node

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- one may create such a computation tree where every inner node has exactly r children
  - some children may be the same
- visiting the nodes corresponding to decisions *e*, 1, 2, ..., r, 11, 12, ... 1r, 21, 22, ..., rr, 111, ... will encounter every node

Nondeterministic TM ↔ deterministic TM

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Nondeterministic TM ↔ deterministic TM

- D has three tapes:
  - tape 1: contains the input string and it is never altered
  - tape 2: a copy of N's tape, it changes according to a particular computation
  - tape 3: string of the actual non-deterministic decisions

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Operation:
  1. tape 1 contains the input w, tape 2, 3 are empty
  2. copy tape 1 to tape 2
  3. use tape 2 to simulate N with the help of tape 3
     - the current state, the input symbol, and tape 3 together determines the next configuration
       - string 142 in tape 3 means the 4th possible configuration has to be chosen (then go right)

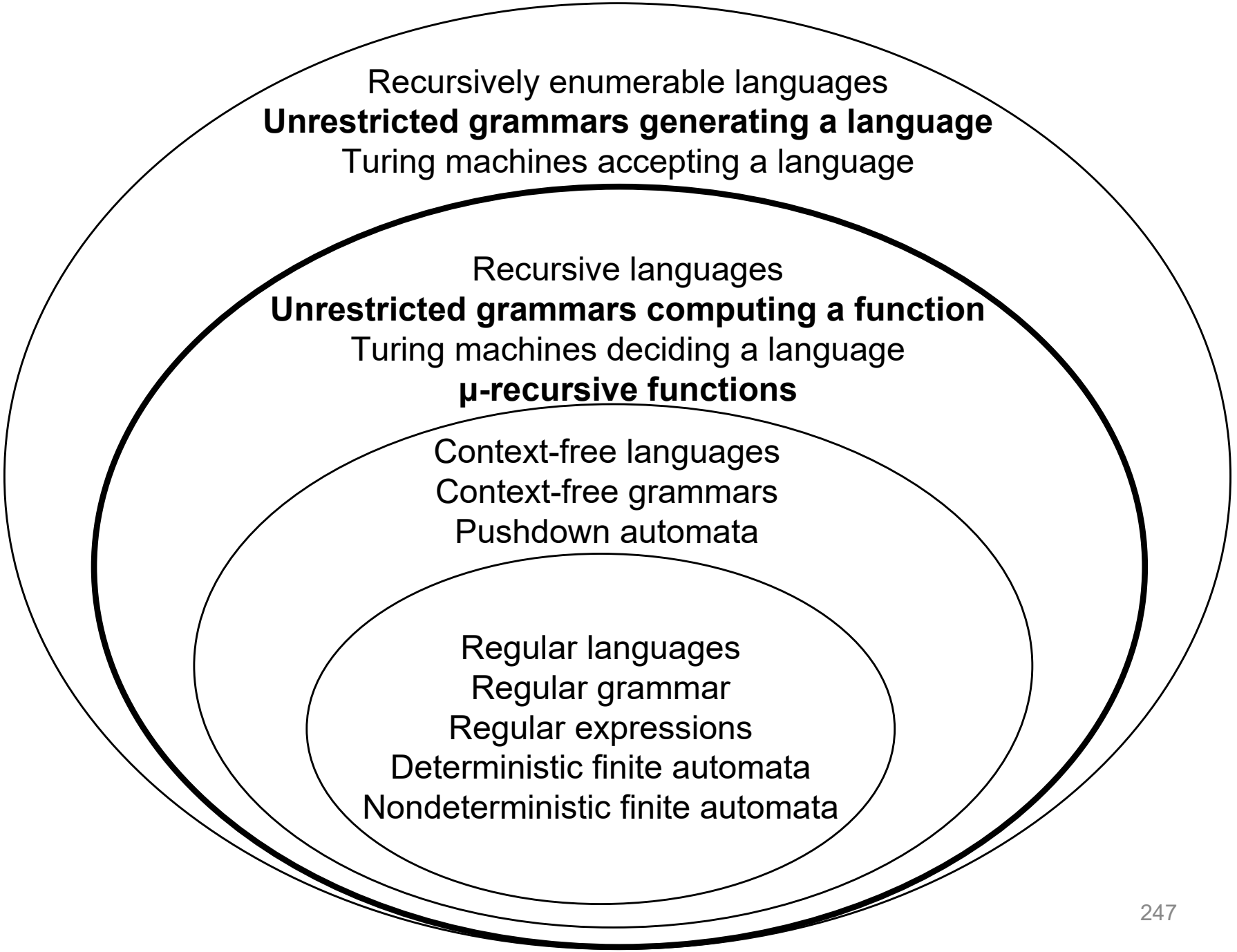
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- if an accepting configuration is encountered, accept the input
- if no more symbols remain on tape 3 go to step 4

4. prepare new computation: replace the string on tape 3 with the lexicographically next string, go to step 2 (*e*, 1, 2, ..., r, 11, 12, ... 1r, 21, 22, ..., rr, ...)

Nondeterministic TM ↔ deterministic TM

- Corollary: every nondeterministic TM has an equivalent deterministic TM


Recursively enumerable languages
Unrestricted grammars generating a language
Turing machines accepting a language
Recursive languages
Unrestricted grammars computing a function
Turing machines deciding a language
μ-recursive functions
Context-free languages
Context-free grammars
Pushdown automata
Regular languages
Regular grammar
Regular expressions
Deterministic finite automata
Nondeterministic finite automata

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- TMs act as languages acceptors like weaker relatives:
  - the finite automata
  - pushdown automata
- TMs can recognize two important families of languages:
  - the recursive languages
  - recursively enumerable languages

Language generators

- Language generators:
  - regular expressions
  - regular grammars
  - context-free grammars
  - new kind of language generator: (unrestricted) grammar

Grammars

- Definition of (unrestricted) grammar: a quadtuple $G = (V, \Sigma, R, S)$, where
  - V is an alphabet
  - $\Sigma \subseteq V$, the set of terminal symbols
    - $V\text{-}\Sigma$, the set of nonterminal symbols
  - $S \in V\text{-}\Sigma$, is the start symbol
  - $R \subseteq (V^*(V\text{-}\Sigma)V^*) \times V^*$, the rules
    - denote: $u \rightarrow_G v$ if $(u, v) \in R$

Derivation

- Definition of the one step derivation of grammar G, $=>_G$:
  - $u, v, x, y \in V^*, u' \rightarrow_G v' \in R$
  - if $u = xu'y, v = xv'y \rightarrow u =>_G v$
- When the grammar to which we refer to is obvious, we write $A \rightarrow w$ and $u => v$ instead of $u' \rightarrow_G v'$ and $u =>_G v$
- Definition of derivation, $=>_G^*$: the reflexive, transitive closure of $=>_G$

Derivation

- We call the sequence $w_0 =>_G w_1 =>_G \ldots =>_G w_n$ a derivation in G of w_n from w_0
  - $w_0, w_1, \ldots w_n \in V^*$
  - $n \in N$, the length of the derivation
    - we say that the derivation has n steps
      - emphasizing the number of steps: $w_0 =>^n w_n$

Language generated by G

- Definition of language generated by grammar G, L(G): the set of strings generated by G
  - $L(G) = \{w \in \Sigma^* : S \Rightarrow_G^* w\}$
- Example: any context-free grammar is also a grammar
  - in CFG the right side of a rule is a member of $V-\Sigma$, rather than $V^*(V-\Sigma)V^*$

Example

- Give grammar G such that $L(G) = \{a^n b^n c^n : n \geq 1\}$!
  - $V = \{S, a, b, c, A, B, C, T_a, T_b, T_c\}$
  - $\Sigma = \{a, b, c\}$, and
  - rules:
    - $S \rightarrow ABCS$
    - $S \rightarrow T_c$
    - $CA \rightarrow AC$
    - $BA \rightarrow AB$
    - $CB \rightarrow BC$

Example

- $CT_c \rightarrow T_c c$
- $CT_c \rightarrow T_b c$
- $BT_b \rightarrow T_b b$
- $BT_b \rightarrow T_a b$
- $AT_a \rightarrow T_a a$
- $T_a \rightarrow e$

Example

- Operation:
  - the first two rules generate string $(ABC)^nT_c$
  - the next three rules allow to sort nonterminals A, B, C, so the string becomes $A^nB^nC^nT_c$

Example

- the remaining rules allow to migrate the token to the left while converting a nonterminal to a terminal one at a time
  - token: one of T_c, T_b, or T_a
  - while T_b migrates left, all B is transformed to b
    - in the last step T_b is converted to T_a
- Beware: some derivation does not yield a string because some nonterminals can not be eliminated

Example

- S => ABCS => ABCABCS => ABCABCT_c => ABACBCT_c => ABABCCT_c => AABBCCT_c => AABBCT_cc => AABB$T_b$cc => AAB$T_b$bcc => AAB$T_b$bcc => AA$T_a$bbcc => A$T_a$abbcc => $T_a$aabbcc => aabbcc
- Stuck:
  - S => ABCS => ABCABCS => ABCABCT_c => ABCABT_bc => ABCA$T_a$bc => ABC$T_a$abc => ABCabc

G ↔ TM

- Theorem: a language is generated by a grammar ↔ it is recursively enumerable
- Proof: only if →
  - G = (V, Σ, R, S) is given, construct TM M such that L(M) = L(G)

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- M is nondeterministic and has 2 tapes:
  - tape 1: contains the input w, never changes
  - tape 2: M reconstructs a derivation of w from S
    - M starts writing S on tape 2
    - M selects nondeterministicly a rule of G or a comparison of tape 1 and 2

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- for each rule u' $\rightarrow_G$ v', M has a number of transitions:
  - » nondeterministicly find u'
  - » delete u'
  - » shift the remaining part of the string to make space
  - » write v'
- go back to select a new rule or to compare

– M stops only if the comparison of tape 1 and 2 results in a positive answer

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof: if ←
  - TM M is given, construct G = (V, Σ, R, S) such that L(M) = L(G)
    - suppose M deletes the tape before accepting
  - V = Σ ∪ K
  - G simulates a backward computation of M
  - configuration (q, ▷u$\underline{a}$w) is represented by the string ▷uaqw◁
    - ◁ is the new end marker symbol

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- rules have to be introduced to each transition of M:
  - if $\delta(q, a) = (p, b)$, $p \in K$, $b \in \Sigma$
    - $(q, \triangleright uaw) \vdash (p, \triangleright ubw)$
    - $\triangleright ubpw \triangleleft \Rightarrow \triangleright uaqw \triangleleft$
    - G rule: $bp \rightarrow aq$, (p, q are nonterminals)

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- if $\delta(q, a) = (p, \rightarrow)$, $p \in K$
  - $(q, \triangleright uabw) \vdash (p, \triangleright uabw)$
  - $\triangleright uabpw\triangleleft \Rightarrow \triangleright uaqbw\triangleleft$
  - G rule: $abp \rightarrow aqb$, for all b

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- if $\delta(q, a) = (p, \rightarrow)$, $p \in K$
  - $(q, \triangleright u\underline{a}) \vdash (p, \triangleright ua\underline{\sqcup})$
  - $\triangleright ua\sqcup p\triangleleft \Rightarrow \triangleright uaq\triangleleft$
  - G rule: $a\sqcup p\triangleleft \rightarrow aq\triangleleft$

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- if δ(q, a) =(p, ←), p ∈ K, a ≠ ⊔ or w ≠ *e*
  - (q, ▷uba<u>a</u>w) |- (p, ▷u<u>b</u>aw)
  - ▷ubpaw◁ => ▷ubaqw◁
  - G rule: pa → aq, for all b
- if δ(q, ⊔) = (p, ←), p ∈ K
  - (q, ▷ub<u>⊔</u>) |- (p, ▷u<u>b</u>)
  - ▷ubp◁ => ▷ub⊔q◁
  - G rule: p◁→ ⊔q◁

– other rules: S → ▷⊔h◁, ▷⊔s → *e*, ◁ → *e*

– a derivation: S => ▷⊔h◁ =>* ▷⊔sw◁ => w◁ => w

Grammatical computation

- Definition of computation of function f by grammar G:
  - $\forall$ w, v $\in$ Σ^*, SwS =>* v $\leftrightarrow$ v = f(w)
  - we say G computes f: $\Sigma^* \rightarrow \Sigma^*$
  - G yield exactly one string, the correct value of f(w) from SwS
    - all derivations agree on the result
    - note that a CFG cannot change terminals
- The computation of f by G is always finite so it can be considered as an algorithm

Grammatical computation

- Definition of grammatically computable function f: ∃ such grammar G that computes it
- Theorem: a function is recursive (Turing computable) ↔ it is grammatically computable
- Proof: similar to previous theorem

Introduction to the Theory of Computation

Lesson 8

4.7. Numerical functions

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Strukturális és Beruházási Alapok

BEFEKTETÉS A JÖVŐBE

Numerical functions

- Functions from numbers to numbers
- They are a new representation of computation different from both TMs and unrestricted grammars

Numerical functions

- E.g.: $f(m, n) = mn^2+3m^{2m+17}$
  - we know that f can be calculated as it is the composition of addition, multiplication, and exponentiation
  - exponentiation can be defined recursively with multiplication
  - multiplication can be defined recursively with addition

Basic functions

- Definition of the basic functions:
  - k-ary zero function: $0 \leq k$, $zero_k(n_1, \ldots, n_k) = 0$, $n_1, \ldots, n_k \in N$
  - j-th k-ary identity function: $0 < j \leq k$ $id_{k,j}(n_1, \ldots, n_k) = n_j$, $n_1, \ldots, n_k \in N$
    - we may omit subscript k from id and zero if it does not cause confusion
  - successor function: $succ(n) = n + 1$, $n \in N$
- E.g.:
  - $zero_2(3, 5) = 0$, $id_{3,1}(6, 2, 0) = 6$, $succ(34) = 35$

Composition

- Definition of composition:
  - let
    - $0 \leq k, l$
    - $g: N^k \rightarrow N$
    - $f, h_1, \ldots, h_k: N^l \rightarrow N$
  - the composition of g with $h_1, \ldots, h_k$:
    $f(n_1, \ldots, n_l) = g(h_1(n_1, \ldots, n_l), \ldots, h_k(n_1, \ldots, n_l))$
    - the 1st parameter of g is given by h_1, the 2nd is given by h_2, …
  - if $k = 1 \rightarrow f(n_1, \ldots, n_l) = g(h(n_1, \ldots, n_l))$

Recursion

- Definition of recursion:
  - let
    - $0 \leq k$
    - $g: N^k \rightarrow N$
    - $f: N^{k+1} \rightarrow N$
    - $h: N^{k+2} \rightarrow N$
  - function f defined recursively by g and h:
    - $f(n_1, \ldots, n_k, 0) = g(n_1, \ldots, n_k)$, base case
    - $f(n_1, \ldots, n_k, m+1) = h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m))$

      $n_1, \ldots, n_k, m \in N$

Primitive recursive function

- Definition of primitive recursive functions: the basic functions and the functions that can be obtained from them with composition and recursion

plus(n, m)

- Show that plus2(n) = n + 2 is primitive recursive!
  - plus2(n) = succ(succ(n))
  - obtained by composition k = l = 1 using f = plus2, g = h_1 = succ

plus(n, m)

- Show that plus(n, m) = n + m is primitive recursive!
  - plus(n, 0) = $id_1(n) = n$
  - plus(n, m+1) = succ(plus(n, m)) = succ(id_3(n, m, plus(n, m)))
    - e.g.: plus(4, 2) = plus(4, 1) + 1
  - f is obtained by primitive recursion: k = 1, f = plus, g = id_1, h(n, m, p) = succ(id_3(n, m, p))
  - h is primitive recursion obtained by composition

mult(n, m)

- Multiplication mult(n, m) = n*m is defined recursively:
  - mult(n, 0) = zero(n) = 0
  - mult(n, m + 1) = plus(n, mult(n, m))
    = plus(id_1(n, m, mult(n, m)), id_3(n, m, mult(n, m)))

exp(n, m)

- Exponent function exp(n, m) = n^m is defined as:
  - exp(n, 0) = succ(zero(n)) = 1
  - exp(n, m + 1) = mult(n, exp(n, m))

Constant

- Constant functions such as $f(n_1, \ldots, n_k) = 3$ are primitive recursive
  - $f(n_1, \ldots, n_k) = succ(succ(succ(zero(n_1, \ldots, n_k)))$
- From now on we can use the usual arithmetic notation: x*y+3 instead of plus(mult(x, y), 3)

sgn(m)

- The sign function, sgn(m) is zero if m = 0 otherwise, it is 1 is also primitive recursive, defined with recursion
  - sgn(0) = 0 = zero(), k = 0
  - sgn(m + 1) = 1 = succ(zero(m, sgn(m)))

pred(m)

- The pred function, pred(m) = m-1 if m > 0, 0 otherwise, is also primitive recursive, defined with recursion
  - pred(0) = 0
  - pred(m+1) = m

- The nonnegative subtraction function, n ~ m = max {n-m, 0}, is also primitive recursive, defined with recursion:
  - n ~ 0 = n
  - n ~ (m + 1) = pred(n ~ m)

iszero(n)

- Definition of primitive recursive predicate: such a primitive recursive function which takes only values 1, 0
- The iszero predicate (1 if n = 0, 0 otherwise) is also primitive recursive, defined with recursion
    - iszero(0) = 1
    - iszero(n + 1) = 0

isone(n)

- The isone predicate, (1 if n = 1, 0 otherwise) is also primitive recursive, defined with recursion
    - isone(0) = 0
    - isone(n + 1) = iszero(n)
- The predicate positive(n) is the same as the sgn(n)

Disjunction

- The predicate greater-than-or-equal(m, n): m ≥ n is defined as iszero(n ~ m)
- The predicate less-than(m, n) is defined as: 1 ~ greater-than-or-equal(m, n)
  - the negation of any primitive recursive predicate is also primitive recursive

Disjunction

- The disjunction of two primitive recursive predicates, p(m, n), q(m, n) is also primitive recursive
  - p(m, n) or q(m, n) defined as:
    1 ~ iszero(p(m, n) + q(m, n))
  - e.g.: p(m, n) = 1, q(m, n) = 0 → iszero(p(m, n) + q(m, n)) = 0, 1 ~ 0 = 1 = 1 or 0

Conjunction

- The conjunction of two primitive recursive predicates, p(m, n), q(m, n) is also primitive recursive
  - p(m, n) and q(m, n) defined as:
    p(m, n)*q(m, n)

Conjunction

- If g and h are primitive recursive and p is a primitive recursive predicate, all three with the same arity k → the function defined by cases is also primitive recursive

$$f(n_1,\ldots,n_k) = \begin{cases} g(n_1,\ldots,n_k), & \text{if } p(n_1,\ldots,n_k) \\ h(n_1,\ldots,n_k), & \text{otherwise} \end{cases}$$

  - $f(n_1, \ldots, n_k) = p(n_1, \ldots, n_k)*g(n_1, \ldots, n_k) + (1\sim p(n_1, \ldots, n_k))*h(n_1, \ldots, n_k)$

rem(m, n)

- The reminder function is also primitive recursive defined with recursion
  - 62 % 6 = 61 % 6 + 1 = (60 % 6 +1) + 1
  - 10*6 % 6 = 0 ↔ (10*6-1) % 6 = 6-1
    - x*y % y = 0 ↔ (x*y-1) % y = y-1
  - rem(0, n) = 0
  - rem(m+1,n) = $\begin{cases} 0, & \text{if equal(rem(m,n), pred(n))} \\ \text{rem(m,n)+1}, & \text{otherwise} \end{cases}$

div(m, n)

- The divide function is also primitive recursive defined with recursion
  - 65 / 6 = 64 / 6 = 63 / 6 = 62 / 6 = 61 / 6 = 60 / 6
  - (65 + 1) / 6 = 65 / 6 + 1, rem(65, 6) = pred(6)
  - div(0, n) = 0
  - $$\mathrm{div}(m+1,n) = \begin{cases} \mathrm{div}(m,n)+1, & \text{if equal}(\mathrm{rem}(m,n), \mathrm{pred}(n)) \\ \mathrm{div}(m,n), & \text{otherwise} \end{cases}$$

Minimalization

- Theorem: there are functions which are not primitive recursive

Minimalization

- Proof:
  - primitive recursive functions are countable
    - they can be represented as strings over a finite alphabet
    - the alphabet contains symbols for
      - basic functions
      - recursion and composition
      - 0 and 1 are used to index in binary the basic functions, e.g.: $d_{11,1}$
      - parenthesis, coma

Minimalization

- consider the strings which define primitive recursive functions and list them all in lexicographic order: $f_0, f_1, \ldots$
- let $g(n) = f_n(n)+1$ (Diagonalization)
- suppose g(n) is primitive recursive
  - $\exists\ m \in N$ such that $f_m = g$
  - $f_m(m) = g(m) = f_m(m)+1$ which is contradiction
- g is not primitive recursive

Minimalization

- Definition of minimalization of g:
  - $0 \leq k$, $g: N^{k+1} \rightarrow N$
  - $$f(n_1,\ldots,n_k) = \begin{cases} \text{the least m, such that } g(n_1,\ldots,n_k,m) = 1, \\ \quad \text{if such an m exists} \\ 0, \text{ otherwise} \end{cases}$$
  - f can be denoted: $\mu\, m[g(n_1, \ldots, n_k, m) = 1]$
    - μ m - minimalization in m
  - though f is always well-defined there is no obvious method for computing it

Minimalization

- The obvious method to calculate m is not algorithm, because it may fail to terminate

```
m := 0;
while g(n_1, …, n_k, m) ≠ 1 do m := m + 1;
output m
```

Minimalization

- Definition of a minimalizable function g:
  - $0 \leq k$, $g: N^{k+1} \rightarrow N$, $n_1, \ldots, n_k \in N$, $\exists\, m \in N$ such that $g(n_1, \ldots, n_k, m) = 1$
  - if the above method always terminates

μ-recursive functions

- Definition of μ-recursive function: the basic functions and the functions that can be obtained from them with composition, recursion, and minimalization of minimalizable functions

μ-recursive functions

- Define the logarithm function: log(m, n)!
  - to avoid the mathematical pitfalls the function log is defined as: $\log(m, n) = \lfloor \log_{m+2}(n+1) \rfloor$
  - log(m, n) = μ p[greater-than-or-equal((m + 2)^p, n + 1)]
    - $\exists\ p \geq 0$ such that $(m + 2)^p \geq n$
    - log(10, 1000) = 3
  - log can be also defined without minimalization

μ-recursive functions ↔ recursive functions

- Theorem: a function f: $N^k \rightarrow N$ is μ-recursive ↔ it is recursive (Turing computable)

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof by induction: only if →
  - suppose f is μ-recursive
  - construct random access TM M to compute f
  - the basic functions are recursive:
    - zero function: delete input, write 0
    - identity function: delete part of the input, shift the remaining part to the left
    - successor function: if the number is represented unary → write I, step right

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- composition can be computed by M
  - f: $N^k \rightarrow N$ is the composition of the function g: $N^l \rightarrow N$ with $h_1, \ldots, h_l$: $N^k \rightarrow N$
    - by induction we know that g and h functions are computable

- f is computed as

$m_1 := h_1(n_1, \ldots, n_k);$

$m_2 := h_2(n_1, \ldots, n_k);$

$m_3 := h_3(n_1, \ldots, n_k);$

...

$m_l := h_l(n_1, \ldots, n_k);$

output $g(m_1, \ldots, m_l);$

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- recursion can be carried out by M
  - f is defined recursively from g and h
    - by induction we know that g and h functions are computable

- f(n_1, ..., n_k, m) can be computed:

```
v := g(n_1, ..., n_k);
if m = 0 then output v;
else
   for i := 1, 2,..., m do
     v := h(n_1, ..., n_k, i - 1, v);
output v
```

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- minimalization can be carried out by M
  - f is by the minimalization of g
    - g is minimalizable
    - by induction we know that g is computable
  - $f(n_1, ..., n_k)$ can be computed:

```
m := 0;
while g(n1, …, nk, m) ≠ 1 do m := m + 1;
output m
```

  - the three basic functions and the three operations can be computed with M
- Proof: if ← (in the book)

Other formal models of computation

- Lambda calculus
- Combinatory logic
- Markov algorithm
- Register machine
- P"

- Source: http://en.wikipedia.org/wiki/Theory_of_computation

Introduction to the Theory of Computation

Lesson 9

5.4. Unsolvable problems about Turing machines

5.5. Unsolvable problems about grammars

5.6. An unsolvable tiling problem

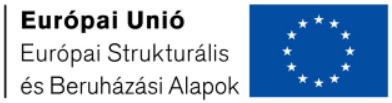
5.7. Properties of recursive languages

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA



Must be known

- Definition of:
  - recursive languages
  - recursively enumerable languages
  - recursive functions
  - universal Turing machine
  - halting problem
  - problems and the corresponding language

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Definition of reduction τ, from language L_1 to L_2: a recursive function (Turing computable) $\tau: \Sigma^* \rightarrow \Sigma^*$ such that $x \in L_1 \leftrightarrow \tau(x) \in L_2$
  - L_2 can contain more strings than L_1, strings which are not the image of any string in L_1
  - the term reduction from L_1 to L_2 is misleading

Undecidable problems about TM

- Theorem: if L_1 is not recursive, $\exists$ a reduction from L_1 to L_2 then L_2 is not recursive either
  - a non-recursive language corresponds to an undecidable problem, L_1 to P_1, L_2 to P_2
  - P_2 is at least as complex as P_1, $P_1 \leq P_2$

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof by contradiction:
  - suppose L_2 is recursive, so it is decided by Turing machine M_2
  - let T be the Turing machine, that computes the reduction τ
  - the machine schema TM_2 would decide L_1
  - L_1 is not recursive, we have reached a contradiction

Undecidable problems about TM

- Theorem: the following problem is undecidable: given a TM M, does M halt on the empty tape?

- Proof:
  - the corresponding language: L = {ρ(M) : M halts on *e*}
  - let M_w write w on its tape then starts to simulate M
    - if $w = a_1 \dots a_n \rightarrow M_w = Ra_1Ra_2R \dots Ra_nL\sqcup M$
  - M halts on w $\leftrightarrow$ M_w halts on *e*
    - $\rho(M)\rho(w) \in H \leftrightarrow \rho(M_w) \in L$
  - there is τ: $\rho(M)\rho(w) \rightarrow \rho(M_w)$ a reduction from H to L
    - it is obvious that τ exists though we do not define it
      - it deletes ρ(w) and modify ρ(M) to $\rho(M_w)$
  - τ exists, H is not recursive → L is not recursive

Undecidable problems about TM

- Theorem: the following problem is undecidable: given a TM M
  - is there any string on which M halts?
  - does M halts on every input?

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof:
  - first language: L' = {ρ(M) : M halts on some input}
  - second language: E = {ρ(M) : M halts on every input}
  - let M' which erases the input and simulate M, M' = DM

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- M halts on *e* ↔ M' = DM halts on any / each w
  - $\rho(M) \in L \leftrightarrow \rho(M') \in L' = E$
  - $L = \{\rho(M) : M \text{ halts on } e\}$
- there is τ: $\rho(M) \rightarrow \rho(M')$ a reduction from L to L' and also from L to E
- τ exists, L is not recursive → L' and E are not recursive

Undecidable problems about TM

- Theorem: the following problem is undecidable: given TM M_1 and M_2, do they halt on the same input strings?

- Proof:
  - the corresponding language: $S = \{\rho(M_1)\rho(M_2) : M_1$ and M_2 halt on the same input$\}$
  - let $M_2 = y$ to be the machine that halts on every input
  - M halts on every input $\leftrightarrow$ both M and y halt on every input, thus, they halt on the same input
    - $\rho(M) \in E \leftrightarrow \rho(M)\rho(y) \in S$
  - there is τ: $\rho(M) \rightarrow \rho(M)\rho(y)$ a reduction from E to S
    - τ exists though we do not define it
  - τ exists, E is not recursive $\rightarrow$ S is not recursive

Undecidable problems about TM

- Theorem: ∃ a fixed TM M for which the following problem is undecidable: does M halt on a given input string?
- Proof: M is the universal TM

Unsolvable problems about grammars

- Theorem: the following problems are undecidable
  - given grammar G and string w, determine if $w \in L(G)$
  - given grammar G, determine if $e \in L(G)$
  - given two grammars G_1, G_2, determine if $L(G_1)=L(G_2)$
  - given grammar G, determine if $L(G) = \emptyset$
  - $\exists$ fixed grammar G_0, determine if string $w \in L(G_0)$

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof:
  - a reduction from the halting problem exists to (a) based on TM unrestricted grammar equivalence
  - simple reductions exist from (a) to the other cases
    - see hints at undecidable problems about TMs

Unsolvable problems about grammars

- Theorem: the following problems about CFG are undecidable
  - a) given CFG G, determine if $L(G) = \Sigma^*$
  - b) given CFG G_1, G_2, determine if $L(G_1) = L(G_2)$
  - c) given PDA M_1, M_2, determine if $L(M_1) = L(M_2)$
  - d) given PDA M, find an equivalent PDA with as few states as possible

Unsolvable problems about grammars

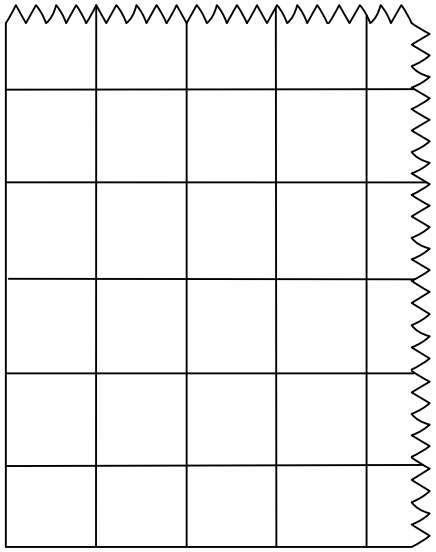
- Proof a: in the book
- Proof b:
  - assume problem b) is decidable
  - consider CFG G_{triv} that generates Σ^*
  - by checking if $L(G) = L(G_{triv})$ we can decide if $L(G) = \Sigma^*$ (problem a) which is contradiction
- Proof c:
  - assume problem c) is decidable
  - $\exists$ CFG G_1, G_2 such that $L(G_1) = L(M_1)$, $L(G_2) = L(M_2)$
  - by checking if $L(M_1) = L(M_2)$ we can decide if $L(G_1) = L(G_2)$ (problem b) which is a contradiction

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Lemma: it is decidable if a one state PDA accepts Σ^*
- Proof d:
  - assume we can optimize PDA M
  - let M' the optimized PDA
  - $\exists$ CFG G such that L(G) = L(M)
  - by checking if M' has one state and $L(M') = \Sigma^*$ we can decide if $L(G) = \Sigma^*$ (problem a) which is a contradiction

An unsolvable tiling problem

- Definition of the tiling problem: given different types of tiles, each one is a unit square, there is an infinite supply of copies of each type; tile the first quadrant of the plane
  - the origin tile must be placed in the corner
  - given which types may adjoin horizontally
  - given which types may adjoin vertically
  - tiles may not be rotated or turned over



An unsolvable tiling problem

- Definition of a tiling system: a quadruple $\Delta = (D, d_0, H, V)$, where
  - D set of the types of the tiles
  - $d_0 \in D$, the origin tile
  - $H \subseteq D^2$ set of those pairs whose 1st and 2nd component may adjoin horizontally
  - $V \subseteq D^2$ set of those pairs whose 1st and 2nd component may adjoin vertically

An unsolvable tiling problem

- Definition of tiling by Δ: f: $N^2 \rightarrow D$ such that
  - $f(0, 0) = d_0$
  - $(f(m, n), f(m+1, n)) \in H , \forall m, n \in N$
  - $(f(m, n), f(m, n+1)) \in V , \forall m, n \in N$

An unsolvable tiling problem

- Theorem: the following problem is undecidable: given a tiling system, determine if ∃ a tiling by that system

An unsolvable tiling problem

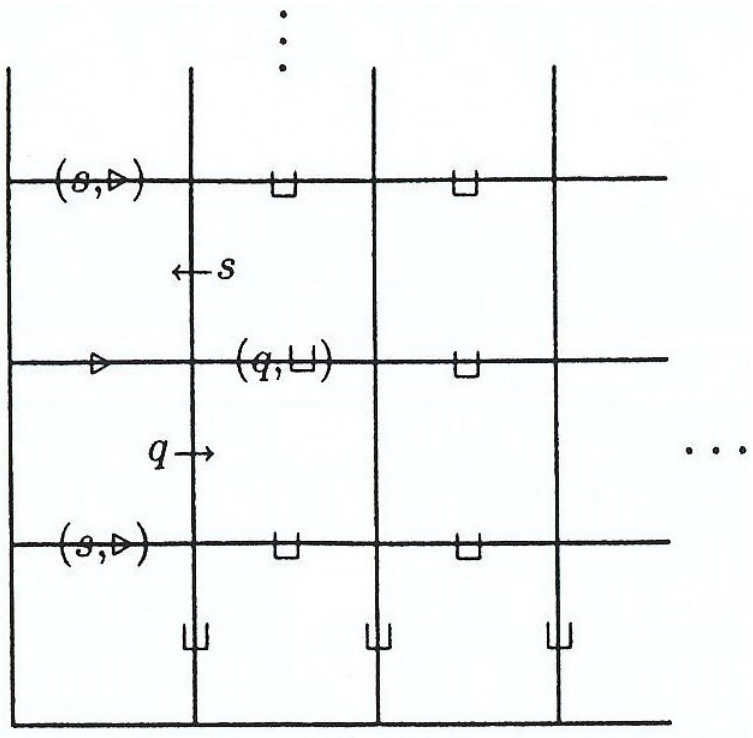
- Proof idea:
  - reduce the complement of the halting problem to tiling problem
    - given TM M, determine if M fails to halt on input *e*
      - it is also undecidable
  - construct from any TM M a tiling system
    - the horizontal dimension represents the tape of M
    - the vertical dimension stands for time

An unsolvable tiling problem

- if M never halts on the empty input, successive rows can be tiled ad infinitum
  - if M halts after k steps → it is impossible to tile more than k rows
- to ensure the relations H and V think of the edges of the tiles as being marked with certain information
  - tiles are allowed to adjoin each other only if the markings on the adjacent edges are identical
  - on the horizontal edges markings are either a member of Σ or $K \times \Sigma$
  - on the vertical edges markings are absent or a member of K with a directional indicator

An unsolvable tiling problem

– the markings on the horizontal edges between the n^{th} and $(n+1)^{st}$ rows of tiles give the configuration of M after n-1 steps of its computation



Properties of recursive languages

- Every TM M semidecides a unique language L(M)
  - there are infinite many other TMs with the same language
    - e.g.: renumber the states of M or make some extra unnecessary head movements
- The complexity of this mapping between TMs and languages implies the next theorem

Properties of recursive languages

- Theorem (Rice's Theorem): suppose that C is a proper subset of the class of all recursively enumerable languages, the following problem is undecidable:
  - given a TM M, is $L(M) \in C$?

Properties of recursive languages

- Corollary: the next problems are undecidable:
  - given a TM M, is L(M)
    - finite?
    - regular?
    - context free?
    - recursive?
    - empty?
    - Σ^*?

Properties of recursive languages

- Every recursive language is recursively enumerable, but the two classes are not the same
  - language H (the halting problem) is recursively enumerable but not recursive

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Definition of Turing enumerable language L: ∃ TM M such that for some fixed state $q \in K$ (not a halting state),
  $L = \{w : (s, \triangleright\underline{\sqcup}) \vdash_M^* (q, \triangleright\underline{\sqcup}w)\}$
  - we say M enumerates L
    - M starts from the blank tape

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- after some computation M reaches state q
  - the current content of the tape is a member of L
  - we say M displays a string
- the computation continues, upon reaching q again a new string in L is generated again, and so on

– M may enumerate the strings of L in any order, possibly with repetitions

Properties of recursive languages

- Theorem: a language is recursively enumerable ↔ it is Turing enumerable

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof: if, ←
  - suppose M enumerates L
  - construct a 2-tape TM M' such that
    - M' saves its input on tape 2
    - simulate M on tape 1
    - whenever the fixed state of M arises the two tapes are compared, if the tapes are
      - the same: M' halts
      - different: M' continues to simulate M on tape 1
  - M' semidecides L

Properties of recursive languages

- Definition of lexicographically Turing enumerable language L: ∃ TM M such that M enumerates L and if $(q, \triangleright\underline{\sqcup}w) \vdash_M^+ (q, \triangleright\underline{\sqcup}w')$, q is the fixed state → w' comes lexicographically after w
  - we say M lexicographically enumerates L
    - M generates the strings of L in lexicographical order

Properties of recursive languages

- Theorem: a language is recursive ↔ it is lexicographically Turing enumerable

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof: only if →
  - suppose M decides L
  - construct M' such that
    - it generates one after the other all strings in the alphabet of L in lexicographic order
    - simulate M on the actual string
      - if accepted → display it by going into the fixed state
      - if rejected → continue with the next string
  - M' lexicographically enumerates L

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof: if ←
  - suppose M lexicographically enumerates L
  - if L finite → L regular → L recursive
    - L = $\{w_1 \cup w_2 \cup ...\}$
  - suppose L is infinite
  - construct M' which works on w input such that
    - simulate M and check each displayed string x
      - if x = w → accept w
      - if x < w → go on simulating M
      - if x > w → reject w
  - M' decides L as there are finitely many x's such that x < w

Introduction to the Theory of Computation

Lesson 10

6.1. The class P

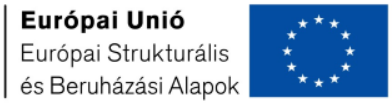
6.2. Problems, problems

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA



BEFEKTETÉS A JÖVŐBE

Must be known

- Definition of:
  - algorithm complexity
  - order of f, *O*(f)
  - relation f ≈ g
  - reflexive, transitive closure

Computational complexity

- Classification of problems, those
  - which can be solved by algorithms
  - which can not be solved by algorithms
- Many problems in the first group can not be solved in any practical sense by computers because of the excessive time requirements

Example

- Definition of the traveling salesman problem, TSP: given n cities and the distances in miles between them, minimize the total distance traversed to visit all the cities exactly once
  - $d_{i,j}$: the distance between city i and j
    - $d_{i,i} = 0$
    - $d_{i,j} = d_{j,i}$
  - a tour is defined by function π, π(i) is the i-th city in the tour
  - the cost of a tour: $c(\pi) = d_{\pi(1),\pi(2)} + d_{\pi(2),\pi(3)} + \ldots$

Example

- The number of possible itineraries: (n-1)!
  - first n-1 city can be selected, then n-2, ...
  - an algorithm can be easily designed that systematically examines all itineraries in order to find the shortest
    - the problem is solvable

Example

- suppose there are 40 cities
  - number of itineraries: 39!, larger than 10^{45}
    - if we could examine 10^{15} tours/s which is quicker than any available or planned computer
    - required time: several billion lifetime of the universe

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- The Church-Turing thesis defines something as algorithm if it can be represented by such a TM which halts on every input
- Similarly the practically feasible algorithm should be defined
- The algorithms (and the solvable problems) can be divided into two subgroups
  - practically feasible algorithms
  - practically infeasible algorithms

Problem of growth

- The difficulty in the traveling salesman problem is the steep growth of the time or steps required by the algorithm on a given input
  - input: number of cities
  - the number of steps required has an exponential rate of growth
    - it is proportional with the number of itineraries (n-1)!
      - grows even faster than 2^n
- If the number of steps required by an algorithm has a polynomial rate of growth then we tend to call it practically feasible

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- It must be decided which computational device should be used as the basis of the practically feasible algorithm
  - TM is the obvious choice but it has a lot of variants
    - multi-tape
    - random access
    - ...

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- these are equally adequate as they can be simulated by a classic TM which requires polynomial time
  - exception: nondeterministic TM
    - its simulation requires exponential time as each possible computation must be examined

P class

- Definition of polynomially bounded deterministic TM M: ∃ a polynomial p(|x|) such that there is no configuration C for any input x for which $(s, > \underline{\sqcup} x) \vdash_M^{p(|x|)+1} C$
  - M always halts at most p(|x|) steps
- Definition of polynomially decidable language: ∃ a polynomially bounded TM that decides it

P class

- Definition of P: the class of all polynomially decidable languages
  - the quantitative analog of the class of recursive languages
- We regard something as a practically feasible algorithm if it belongs to P
- See: http://qwiki.stanford.edu/wiki/Complexity_Zoo for hierarchy of the problem classes

P class

- Theorem: P is closed under complement
- Proof:
  - if language L is decidable by a polynomially bounded TM M
  - then L^C is decided by the version of M that inverts y and n states
    - the polynomial bound is unaffected

P class

- Definition of language E: {ρ(M)ρ(w) : M accepts input w after at most $2^{|w|}$ steps}
  - the quantitative analog of the halting language H
  - E is recursive: a modified version of the universal TM, U can decide it
    - if after at most $2^{|w|}$ steps U does not halt then w is rejected

P class

- Theorem: E $\notin$ P
  - E can be decided but only with a practically infeasible algorithm, i.e., with a TM which is not bounded polynomially based on the length of the input

Time requirements in practice

- How well does class P capture the intuitive notion of practically feasible algorithm?
  - an algorithm with time
    - n^{100} or $10^{100}n^2$ belongs to P
    - $n^{\log \log n}$ does not belong to P
  - $\exists\ n_0$ such that for $\forall\ n>n_0$, $n^{100}< n^{\log \log n}$
    - but n_0 is very big in this case
    - if in practice inputs smaller than n_0 are important then $n^{\log \log n}$ is preferable despite that it is not in P
- Example: simplex method for LP is exponential but widely used in practice

Time requirements in practice

- Though theoretically possible, extreme time bounds rarely come up in practice
- In general
  - polynomial algorithms in computational practice have:
    - small exponents
    - small constant coefficients
  - nonpolynomial algorithms:
    - usually hopelessly exponential
    - quite limited use in practice

P class

- The P class categorizes an algorithm based on its worst-case performance:
  - the largest running time over all possible input of lengths n
- Some suggest an average-case approach would be better
  - it is not clear which distribution on the inputs should be adopted

The reachability problem

- Definition of the reachability problem: given a directed graph, $G \subseteq V \times V$, $V = \{v_1, \ldots, v_n\}$ is there a path from v_i to v_j ?
  - the corresponding language: $R = \{\rho(G)\rho(v_i)\rho(v_j) :$ there is a path from v_i to v_j in $G\}$
    - $\rho(i)$: some encoding of node v_i as strings
    - $\rho(G)$: some encoding of graph G as strings
      - e.g.: adjacency matrix

The reachability problem

- A problem in general is:
  - a set of inputs, typically infinite
  - together with a yes-or-no question asked of each input
    - a property an input may or may not have

The reachability problem

- The corresponding language is the set of the encodings of those inputs which have the desired property
  - if a TM M decides a string that belongs to the language then we know the input have the desired property
- Problems and associated languages are interchangeable

Reachability in P

- Theorem: $R \in P$
- Proof:
  - compute the reflexive, transitive closure of G
    - use the algorithm learned before
      - implement the algorithm with a random access TM
    - the algorithm needs $O(n^3)$ time
    - random access TM can be simulated by ordinary TM in polynomial time
  - inspect if there is an edge between v_i and v_j
    - if there is an edge then there is a path too

Euler cycle problem

- Definition of the Euler cycle problem: given graph G, is there a closed path in G that uses each edge exactly once?
- A graph has Euler cycle if
  - for any pair of nodes (u, v), neither of which is isolated, $\exists$ path from u to v
  - all nodes have equal number of incoming and outgoing arcs
- The Euler cycle problem is in P

Hamilton cycle problem

- Definition of the Hamilton cycle problem: given graph G, is there a closed path in G that passes through each node exactly once?
- The Hamilton cycle problem is in NP

The traveling salesman problem

- Optimization problems do not require a single yes or no answer so they can not be converted into language readily
- Modify the TSP: given a distance matrix and $B > 0$ (budget), is there a tour π such that $c(\pi) < B$?
- The TSP is in NP

Partition problem

- Definition of the partition problem: given a set of nonnegative integer a_1, ..., a_n represented in binary form, is there a set $P \subseteq \{1, ..., n\}$ such that $\Sigma_{i \in P} a_i = \Sigma_{i \notin P} a_i$?

Partition problem

- Definition of the unary partition problem: similar to the partition problem but the input is encoded in unary form
- The partition problem can be solved
  - calculate $B(i) = \{\Sigma_{j \in K} a_j : K \subseteq \{1, ..., i\}\}$ for $\forall$ i
    - sums of all possible partition of the first i number
  - $H = \Sigma_{i \in \{1, ..., n\}} a_i / 2$
  - if $H \in B(n) \rightarrow$ the answer is yes

Partition problem


```
B(0) = {0}
for i = 1, ..., n
   B(i) = B(i-1)
   for ∀ j in B(i-1)
      add j+a_i to B(i)
```

- The number of steps required by the algorithm: $n2^n$
  - the new B set is usually twice as big as the previous one

Partition problem

- The binary (or decimal or any other radix system) encoding of the numbers is powerful
  - to encode the number n we need $\log_2(n)$ digits
    - log is the inverse of exp

Partition problem

- There are n number and each has value about 2^n
  - number of steps: $n2^n$
  - input size:
    - binary encoding: n*n → complexity: exponential
      - the partition problem is in NP
    - unary encoding: $n2^n$ → complexity: polynomial
      - the unary partition problem is in P
- The details about encoding objects does not effect the membership of the corresponding language in P
  - exception: when binary encoding is used instead of unary

Satisfiability problem

- Definition of Boolean variable: a statement which can be true or false
  - the negate of a Boolean variable x is denoted with x^C

Satisfiability problem

- Definition of the clause of conjunctive normal form: an expression from Boolean variables and 'or' operators
- Definition of conjunctive normal form: an expression from clauses and 'and' operators
  - e.g.: $(x_1 \vee x_2 \vee x_4^C) \wedge (x_2 \vee x_3^C \vee x_5^C)$

Satisfiability problem

- Definition of truth assignment, T: a function which assign true or false value for each variable in formula F
  - T satisfies F if the value of F is true
  - F is satisfiable if ∃ a T which satisfies it
- Definition of the satisfiability problem: given a Boolean formula in conjunctive normal form, is it satisfiable ?
- The problem is in NP

Satisfiability problem

- Definition of the 2 satisfiability problem: such a satisfiability problem whose formula contains only such clauses which have at most two variables
- The problem is in P
  - the solution method is described in the book

NP class

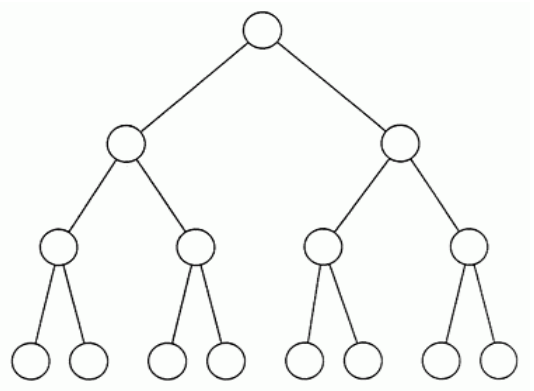
- Definition of polynomially bounded nondeterministic TM M: $\exists$ a polynomial p(|x|) such that there is no configuration C for any input x for which $(s, \triangleright\underline{\ }x) \vdash_M^{p(|x|)+1} C$
  - M always halts at most p(|x|) steps

NP class

- Definition of NP: the class of languages decidable by a polynomially bounded nondeterministic TM
- Nondeterministic behavior is very strong quality
  - nondeterministic acceptance means for $\forall$ w $\in$ L $\exists$ at least one computation which accepts w (all other configuration may reject the w)

NP class

- The operation of a nondeterministic TM can be represented by a tree
  - each node is a configuration
    - the root is the initial configuration
  - each node with 2 or more children represent a nondeterministic choice
  - with each new level
    - the length of the computation increases with just 1
    - the number of nodes may increase exponentially



NP class

- Theorem: the satisfiability problem is in NP
- Proof:
  - the nondeterministic TM which accepts the corresponding language works as follows
    - check if the input is really the encoding of a conjunctive normal form
    - count the variables in the formula
      - write I symbols on a second tape

NP class

- guess the values of the variables nondeterministicly
- check the value of each clause
  - if at least one variable is true → the clause is true
- check the value of the formula
  - if all the clauses are true → the formula is true
- accept or reject based on the results

– all subtasks can be performed in polynomial time

NP class

- Similar proofs can be used to establish that other problems are in NP
  - use a nondeterministic TM to guess a string which helps to decide the input
    - the string is called certificate or witness
  - check deterministically if the certificate helps to accept or reject the input

NP class

- Example: TSP
  - guess the bijection π
  - calculate the cost based on π and the distance matrix
  - check if the cost is smaller than the budget

NP class

- Definition of polynomial balanced language L': if x;y $\in$ L', L' $\subseteq \Sigma^*;\Sigma^* \rightarrow \exists$ polynomial p(n) such that |y| < p(|x|)
  - ; $\notin \Sigma$
  - x will be the input and y certificate

NP class

- Theorem: if $L \in NP \leftrightarrow L' \in P$ and L' is polynomial balanced (the certificate is succinct)
  - L' = {x;y : y is a certificate for input x}
  - the length of the certificates are polynomially bounded
  - the acceptance of the input can be determined in polynomial time from the certificate
- For the prime number problem the certificate is a pair of integer whose product is the input

EXP class

- Definition of exponentially bounded deterministic TM M: $\exists$ a polynomial p(|x|) such that there is no configuration C for any input x for which $(s, \triangleright\underline{\ }x) \vdash_M^{\exp(p((|x|)+1)} C$
  - M always halts at most exp(p(|x|)) steps

EXP class

- Definition of EXP: the class of languages decidable by a exponentially bounded deterministic TM
- Theorem: if $L \in NP \rightarrow L \in EXP$
- Proof: if L is decided by a nondeterministic TM M → the deterministic version of M decides L in exponential time
  - see the equivalence of the nondeterministic and deterministic TM

EXP class

- Theorem: $P \subseteq NP \subseteq EXP$
- Proof:
  - see the previous proof for $NP \subseteq EXP$
  - a deterministic TM can be seen as a special nondeterministic TM, so the definitions of P and NP imply $P \subseteq NP$

EXP class

- We know that $E \notin P$, $E \in EXP$, so at least one of the previous inclusions are proper
  - $E = \{\rho(M)\rho(w) : M$ accepts input w after at most $2^{|w|}$ steps$\}$
- It is strongly supposed that both inclusions are proper but there is no proof for it
  - it is an unresolved question of the theory

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- NP-complete problems have the following properties:
  - all problems in NP can be reduced to them via polynomial-time reductions
  - if there was a quick algorithm for any of these problems then all NP problems could be solved efficiently
- Unfortunately theory fails to prove that P ≠ NP, but we strongly assume that
  - Clay Mathematics Institute has offered a $1 million prize for the first correct proof

Polynomial-time reductions

- Definition of polynomial-time computable function f: ∃ a polynomially bounded TM that computes it
  - f: $\Sigma^* \rightarrow \Sigma^*$

Polynomial-time reductions

- Definition of polynomial reduction from L_1 to L_2, τ: τ is a polynomial-time computable function, $x \in L_1 \leftrightarrow \tau(x) \in L_2$
  - $L_1, L_2 \subseteq \Sigma^*$
  - τ: $\Sigma^* \rightarrow \Sigma^*$
  - notation: $L_1 \circ L_2$
    - L_2 is at least as hard as L_1
  - it is analogue of the reductions we used to establish undecidability of problems with the help of H

Polynomial-time reductions

- Theorem: if $L_2 \in P$, $L_1 \circ L_2 \rightarrow L_1 \in P$
- Proof:
  - L_2 can be decided by M_2 in polynomial time
  - τ can be computed in polynomial time
  - L_1 can also be decided by TM_2 in polynomial time
- Theorem: if $L_1 \in EXP$, $L_1 \circ L_2 \rightarrow L_2 \in EXP$

Introduction to the Theory of Computation

Lesson 11

7.1. Polynomial-time reductions

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Strukturális és Beruházási Alapok

BEFEKTETÉS A JÖVŐBE

Polynomial reduction from Hamilton cycle problem to Satisfiability

- Theorem: ∃ polynomial reduction from Hamilton cycle problem to Satisfiability

Polynomial reduction from Hamilton cycle problem to Satisfiability

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Construction:
  - a G graph must be converted by τ to a Boolean formula in conjunctive normal such a way that τ(G) is satisfiable ↔ G has a Hamilton cycle
    - $G \subseteq V \times V$, $V = \{1, 2, \ldots, n\}$
  - Hamilton cycle: a closed path visiting each node of G exactly once
  - formula τ(G) will involve n^2 Boolean variables: $x_{i,j}$, $1 \leq i, j \leq n$
    - $x_{i,j} = 1$ if node i of G is at the j-th position in the Hamilton cycle

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- the clauses of τ(G) are the requirements for the existence of a Hamilton cycle
- requirement 1: exactly one node should appear j-th in the Hamilton cycle
  - at least one node should appear j-th in the Hamilton cycle
    - $(x_{1,j} \vee x_{2,j} \vee \ldots \vee x_{n,j})$ for $\forall$ j = 1, ..., n
  - two distinct nodes i and k should not appear j-th in the Hamilton cycle
    - $(x_{i,j}{}^{C} \vee x_{k,j}{}^{C})$, for i, j, k = 1, ..., n, and $j \neq k$

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- requirement 2: each node should appear in the Hamilton cycle
  - the j-th should appear at least once in the Hamilton cycle
    - $(x_{j,1} \vee x_{j,2} \vee \ldots \vee x_{j,n})$ for $\forall$ j = 1, ..., n
  - the j-th node should not appear in two distinct positions k and i in the Hamilton cycle
    - $(x_{i,j}{}^{C} \vee x_{i,k}{}^{C})$, for i, j, k = 1, ..., n, and $j \neq k$
- requirement 1 and 2 ensure that x represents a bijection between nodes and positions

Polynomial reduction from Hamilton cycle problem to Satisfiability

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- requirement 3: if there is no (i, k) edge then i and k can not appear consecutively
  - $(x_{i,j}{}^{C} \vee x_{k,\, j+1}{}^{C})$ for j = 1, ..., n, for $\forall$ (i, k) such that (i, k) is not an edge of G
    - if $j = n \rightarrow j+1 = 1$
- $\tau(G)$ has $O(n^3)$ clauses, these clauses are simple
  - it is simple to construct a polynomial-time TM that computes τ

Polynomial reduction from Hamilton cycle problem to Satisfiability

- Proof: G has a Hamilton cycle ↔ τ(G) is satisfiable
  - suppose that G has a Hamilton cycle (π(1), π(2), ..., π(n))
    - let truth assignment T be the following: $T(x_{i,j})$ = true ↔ j = π(i)
    - T satisfies all clauses of τ(G) because of the construction

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- suppose T is a truth assignment that satisfies τ(G)
  - requirement 1 and 2 ensure that T assigns one position for ∀ node
  - denotes with π(i) the unique j for which $T(x_{i,\pi(i)})$ is true
  - requirement 3 ensures that if i = π(j) and k = π(j+1) then (j, k) must be an edge of G
- There is also a polynomial reduction from Satisfiability to Hamilton cycle problem

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Definition of the Two-machine scheduling problem:
  - n tasks must be scheduled on two machines
  - both machines have the same speed
  - each task can be executed on either machine
  - there are no restrictions on the order of the execution

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- given:
  - the execution times a_1, ..., a_n of the tasks
  - deadline D (all in binary form)
- question: can you complete all these tasks on the two machines within the deadline?
  - $\exists$ partition of the numbers into two sets so that the numbers in each set add up to D or less?

Polynomial-time reductions

- Definition of Knapsack problem:
  - given:
    - a set $S = \{a_1, ..., a_n\}$ of nonnegative integers
    - an integer K (all in binary form)
  - question: $\exists$ a subset $P \subseteq S$ such that $\Sigma_{a \in P} a = K$?
- Between two-machine scheduling problem, Knapsack problem and partition problem there are six polynomial reductions reducing any of these problems to any other
- Example:
  reduction from Knapsack to Partition

Polynomial reduction from Knapsack problem to Partition problem

- Theorem: ∃ polynomial reduction from Knapsack problem to Partition problem
- Construction:
  - given an instance of Knapsack problem, with integers a_1, ..., a_n, and K
  - if $K = H = ½\Sigma_{i=1} a_i$ → the reduction is to erase K from the input
  - if $K \neq H$ → erase K from the input and add two new integers $a_{n+1} = 2H + 2K$, $a_{n+2} = 4H$ to the set of a_i's

Polynomial reduction from Knapsack problem to Partition problem

- Proof:
  - in partition the new integers must be on opposite sides

    $\Sigma_{i=1\ldots n} a_i < a_{n+1} + a_{n+2} = 2H + 2K + 4H$
  - write the partition of the numbers with set P

    $4H + \Sigma_{a \in P} a = 2H + 2K + \Sigma_{a \in S-P} a$
    - $P \subseteq \{1, \ldots, n\}$
  - add $\Sigma_{a \in P} a$ to both sides

    $4H + 2\,\Sigma_{a \in P} a = 4H + 2K$

    $\Sigma_{a \in P} a = K$

Polynomial reduction from Knapsack problem to Partition problem

- the resulting instance of Partition problem has a solution ↔ the original instance of Knapsack problem has one
  - the Knapsack corresponds to P
- the reduction can be carried out in polynomial time

Polynomial reduction from Knapsack problem to Partition problem

- Theorem: ∃ polynomial reduction from Partition problem to Knapsack problem
  - Partition problem is a special case of Knapsack problem
  - the reduction calculates bound $K = ½\Sigma_{i=1} a_i$
    - if K is not an integer → the given instance of Knapsack and of Partition have no solution

Reduction from Partition problem to Two-machine scheduling problem

- Theorem: ∃ polynomial reduction from Partition problem to Two-machine scheduling problem
- Construction:
  - given an instance of Partition problem with integers $a_1, ..., a_n$
  - the reduction keeps $a_1, ..., a_n$ as execution times and calculates deadline $D = ½\Sigma_{i=1} a_i$
    - if D is not an integer → the given instance of Two-machine scheduling and Partition problem have no solution

Reduction from Partition problem to Two-machine scheduling problem

- Proof:
  - if the sums of the two sets are exactly D (Partition problem) → at most D is also true (Two-machine scheduling problem)
  - if the sums of the two sets are at most D (Two-machine scheduling problem) and D is the half-sum → the sums are exactly D (Partition problem)

Reduction from Two-machine scheduling problem to Partition problem

- Theorem: ∃ polynomial reduction from Two-machine scheduling problem to Partition problem

Reduction from Two-machine scheduling problem to Partition problem

- Construction:
  - given an instance of two-machine scheduling with task lengths a_1, ..., a_n and deadline D
  - idle time: $I = 2D - \Sigma_{i=1} a_i$
    - the sum of times while the machines do not work
  - add some new numbers to the set of lengths of tasks
    - the sum of these new numbers is I
    - any number between 0 and I can be created by adding a subset of these new numbers

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- determine the new numbers
  - 1st solution: we could add I copies of 1, but this is not a polynomial-time reduction
    - it would represent a kind of unary encoding of I while all other numbers are given binary
  - 2nd solution:
    - powers of 2 if they are smaller than I/2
    - another integer to bring the sum to I
    - e.g.: if I = 56 → new integers: 1, 2, 4, 8, 16 and 25

Reduction from Two-machine scheduling problem to Partition problem

- Proof:
  - if the sums of the two sets are at most D (Two-machine scheduling problem) → adding the new integers to the problem helps to keep busy the machines in the idle time
    - if there is no idle time → we have a Partition problem
  - if the sums of the two sets are exactly D (Partition problem) → at most D is also true (Two-machine scheduling problem)

Polynomial-time reductions

- Theorem: if τ_1 is a polynomial reduction from L_1 to L_2 and τ_2 is a polynomial reduction from L_2 to $L_3 \rightarrow \tau_1 \circ \tau_2$ is a polynomial reduction from L_1 to L_3

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof:
  - τ_1 is computed by TM M_1 in time p_1 (polynomial)
  - τ_2 is computed by TM M_2 in time p_2 (polynomial)
  - $\tau_1 \circ \tau_2$ can be computed by $M_1 M_2$ in time $p_2(p_1(|x|))$
    - the input of M_2 is longer than the input of M_1
    - $|\tau_1(x)| \leq p_1(|x|)$ in each step M_1 can write at most 1 symbol to the tape
  - $x \in L_1 \leftrightarrow \tau_2(x) \in L_2 \leftrightarrow \tau_1 \circ \tau_2(x) \in L_3$

- Definition: A language $L \subseteq \Sigma^*$ is called $\mathrm{N}\Pi$-complete if
  - $L \in \mathrm{N}\Pi$
  - and for every language $L' \in \mathrm{N}\Pi$, $\exists$ polynomial reduction from L' to L

- Theorem: let L be an $\mathrm{N}\Pi$-complete language, $\Pi = \mathrm{N}\Pi \leftrightarrow L \in \Pi$
- Proof: only if $\rightarrow$
  - $L \in \mathrm{N}\Pi$-complete $\rightarrow L \in \mathrm{N}\Pi$
  - $L \in \mathrm{N}\Pi$, $\mathrm{N}\Pi = \Pi$, $\rightarrow L \in \Pi$
- Proof: if $\leftarrow$
  - $L \in \mathrm{N}\Pi$-complete is decided by a deterministic TM M_1 in time $p_1(n)$, a polynomial

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- let L' $\in$ NP
- $\exists$ a polynomial reduction τ from L' to L according to the definition of NP-complete
  - τ is computed by TM M_2 in time $p_2(n)$, a polynomial
- M_2M_1 decides L' in time $p_1(p_2(|x|))$, a polynomial
  - the input of M_1 is longer than the input of M_2
  - $|\tau(x)| \leq p_2(|x|)$, in each step M_2 can write at most 1 symbol to the tape
- if $\forall$ L' $\in$ NP can be decided in polynomial time $\rightarrow$ P = NP

Homework

- NP-hard, co-NP, co-NP-complete with examples

Introduction to the Theory of Computation

Lesson 12

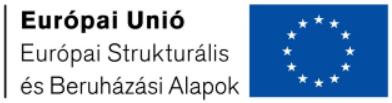
7.2. Cook's Theorem

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA



BEFEKTETÉS A JÖVŐBE

Cook's theorem

- Theorem: if $L_1 \in \mathrm{N\Pi}$-complete, $\exists$ a polynomial reduction from L_1 to $L_2 \rightarrow L_2 \in \mathrm{N\Pi}$-complete
- Proof:
  - $\exists$ polynomial reduction from any $L' \in \mathrm{N\Pi}$ to L_1 because $L_1 \in \mathrm{N\Pi}$-complete
  - $\exists$ polynomial reduction from L_1 to L_2
  - $\exists$ polynomial reduction from any $L' \in \mathrm{N\Pi}$ to L_2 because of the transitivity of the polynomial reductions

Cook's theorem

- Satisfiability is the first problem shown to be NΠ-complete (Stephen A. Cook, 1971)
- We prove that the Bounded tiling problem is NΠ-complete using the definition of NΠ-complete

Cook's theorem

- Definition of Bounded tiling problem:
  - given:
    - a tiling system (D, H, V)
      - D set of tiles
      - H relation of horizontally joinable tiles
      - V relation of vertically joinable tiles
    - the length of the square to be tiled: $s \in Z^+$
    - the first row of tiles as a function $f_0 : \{0, ..., s-1\} \rightarrow D$
      - in tiling problem only the (0, 0) tile is given
  - $\exists$ tiling of an $s \times s$ square? (and not of a whole quadrant)

Cook's theorem

- tiling is a function $f : \{0, 1, ..., s-1\} \times \{0, 1, ..., s-1\} \rightarrow D$, such that
  - $f(m, 0) = f_0(m), \forall\ m < s$
  - $(f(m, n), f(m+1, n)) \in H, \forall\ m < s -1, n < s$
  - $(f(m, n), f(m, n+1)) \in V, \forall\ m < s, n < s -1$

Cook's theorem

- Theorem: Bounded tiling problem is $\mathrm{N}\Pi$-complete
- Proof:
  - certificate of the problem: a complete listing of the s^2 values of the tiling function f

Cook's theorem

- the certificate is succinct:
  - the certificate contains s^2 encoding of tiles
  - the input contains s encoding of tiles in f_0
    - if only the corner was given → the certificate is not succinct
- we can check in polynomial time if f is really a tiling function
- the problem has a succinct certificate, checkable in polynomial time so it is in $\mathrm{N}\Pi$

Cook's theorem

- we must show $\forall$ L $\in$ NΠ can be reduced via a polynomial reduction to the Bounded tiling problem
- consider any L $\in$ NΠ
  - we only know that $\exists$ a nondeterministic TM M deciding L in polynomial time, p(|x|)
- τ, a polynomial-time computable function
  - calculates s = p(|x|) + 2
  - defines (D, H, V) based on M
    - tiles are defined in the same way they were defined at the Tiling problem (not detailed now)
      - » based on K, Σ, δ of M

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

  - the edges of the tiles are signed with the elements of the previous sets
  - tiles with the same signs can be joined
- calculates f_0 the 0^{th} row of the s $\times$ s square
  - $f_0(0)$ is a tile with upper edge marking $>$
  - $f_0(1)$ is a tile with upper edge marking (s, $\int$)
  - $f_0(i+1)$ for i = 1, ..., |x| is a tile with upper edge marking x_i
  - $f_0(i)$ for all i > |x| + 1, is a tile with upper edge marking $\int$

Cook's theorem

- the markings of the horizontal edges between rows t and t+1 represent the tape content of M after the t^{th} step (starting with input x)
  - can be proved with induction
  - the horizontal edge markings between the 0^{th} and the 1^{st} rows will spell the initial configuration of
    $M(s, >\!\!\int x)$ on input x

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- there is a tile not used before with (y, a) lower and upper markings
  - y is the accepting state, a $\in \Sigma$
  - it makes possible to repeat the last row if the input is accepted

Cook's theorem

- $x \in L \leftrightarrow$ M accepts $x \leftrightarrow \tau(x)$ has a tiling
  - suppose $\exists$ $\tau(x)$ has a tiling
    - the last row contains tiles too
  - the computation of M on input x can not continue for more than $p(|x|) = s-2$ steps
    - the upper markings of the $s-2^{nd}$ row must contain one of the symbols y and n
  - since there are tiles in the $s-1^{st}$ (last) row too, and there is no tile with lower marking n, symbol y must appear in the upper markings of the $s-2^{nd}$ row
    - the computation is accepting

Cook's theorem

- conversely if M accepts x then it can be simulated by a tiling
  - we know the horizontal edge markings correspond to tape contents
  - the last row containing an accepting state may be repeated several times to complete the tiling

Cook's theorem

- Theorem: (Cook's theorem) Satisfiability is $\mathrm{N}\Pi$-complete
  - we have already proved that Satisfiability is in $\mathrm{N}\Pi$
  - reducing the Bounded tiling problem to Satisfiability implies that the latter problem is also $\mathrm{N}\Pi$-complete
  - given a tiling system (D, H, V), side s, bottom row f_0, where $D = \{d_1, ..., d_k\}$
  - construct a Boolean formula $\tau(D, H, V, s, f_0)$, such that $\exists$ an $s \times s$ tiling $f \leftrightarrow \tau(D, H, V, s, f_0)$ is satisfiable

Cook's theorem

- Construction:
  - introduce $x_{m,n,d}$ for $\forall$ $0 \leq m, n < s$, $d \in D$
    - $x_{m,n,d}$ = true $\leftrightarrow$ f(m, n) = d
  - the clauses of τ are the requirements for the existence of a tiling
  - requirement 1: exactly one tile should appear in each position
    - each position has at least one tile
      - $(x_{m,n,d1} \vee x_{m,n,d2} \vee \ldots \vee x_{m,n,dk})$, $\forall$ m, n < s
    - a position can not have more than one tile
      - $(x_{m,n,d}{}^{C} \vee x_{m,n,d'}{}^{C})$, $\forall$ m, n < s, d $\neq$ d'

Cook's theorem

- requirement 2: the first row is given by f_0
  - $(x_{i,0,f_0(i)})$, i = 0, ..., s-1
- requirement 3: two tiles can not be next to each other if relation H does not allow it
  - $(x_{m,n,d}{}^{C} \vee x_{m+1,\ n,\ d'}{}^{C})$, $\forall$ n < s, m < s-1, (d, d') $\notin$ H
    - either d can not appear in (m, n) or d' can not appear in (m+1, n) if (d, d') $\notin$ H

Cook's theorem

- requirement 4: two tiles can not be one on another if relation V does not allow it
  - $(x_{m,n,d}{}^{C} \vee x_{m,\, n+1,\, d'}{}^{C})$, $\forall$ n < s-1, m < s, (d, d') $\notin$ V
    - either d can not appear in (m, n) or d' can not appear in (m, n+1) if (d, d') $\notin$ V

Cook's theorem

- Proof: only if →
  - suppose $\tau(D, H, V, s, f_0)$ is satisfiable by the truth assignment T
  - since the clauses of requirement 1 are true T represents a function $f : \{0, ..., s - 1\} \times \{0, ..., s - 1\} \rightarrow D$
  - since the clauses of requirement 2 are true f extends f_0
  - since the clauses of requirement 3 and 4 are true f represents a legal tiling

Cook's theorem

- Proof: if ←
  - suppose f is a legal tiling
  - it is easy to see that all clauses of τ are true

Cook's theorem

- Theorem: 3 - Satisfiability is NΠ-complete
  - special case of Satisfiability in which all clauses involve 3 or fewer literals
  - it is in NΠ, as it is a special case of Satisfiability known to be in NΠ
  - we reduce Satisfiability to 3 - Satisfiability

Cook's theorem

- Construction:
  - leave the short clauses unchanged
  - for $\forall$ long clause $(\lambda_1 \vee \lambda_2 \vee \ldots \vee \lambda_k)$ in F, k > 3
    - introduce new Boolean variables: $y_1, \ldots, y_{k-3}$
      - $y_i = \lambda_{i+2} \vee \ldots \vee \lambda_k$
    - replace the long clause with

      $(\lambda_1 \vee \lambda_2 \vee y_1), (y_1^C \vee \lambda_3 \vee y_2), (y_2^C \vee \lambda_4 \vee y_3), \ldots,$
      $(y_{k-4}^C \vee \lambda_{k-2} \vee y_{k-3}), (y_{k-3}^C \vee \lambda_{k-1} \vee \lambda_k)$
      - $(y_i^C \vee \lambda_{i+2} \vee y_{i+1})$ states if y_i = true $\rightarrow$ λ_{i+2} = true or y_{i+1} = true
  - τ can be carried out in polynomial time

Cook's theorem

- Proof: only if →
  - suppose that truth assignment T satisfies τ(F)
  - T satisfies the short clauses of F as they are the same as in τ(F)
  - regard any long clause C of F, at least one of λ_i of C is true
    - suppose $\forall\ \lambda_i$ of C is false
    - the clauses corresponding to C in τ(F) are true because of the y_i variables
      - y_1 = true, because of $(\lambda_1 \vee \lambda_2 \vee y_1)$
      - y_2 = true, because of $(y_1^C \vee \lambda_3 \vee y_2)$
      - …

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

  - y_{k-3} = true, because of $(y_{k-4}{}^{C} \vee \lambda_{k-2} \vee y_{k-3})$
  - $(y_{k-3}{}^{C} \vee \lambda_{k-1} \vee \lambda_k)$ = false, but we know T satisfies F(F) so the assumption "$\forall$ λ_i of C is false" is not valid
- T satisfies both the short and long clauses of F

- Proof: if ←
  - suppose that truth assignment T satisfies F
  - let T' the extension of T for the y_i values
  - for $\forall$ long clause $C = (\lambda_1 \vee \lambda_2 \vee ... \vee \lambda_k)$ of F
    - j = argmin{$T(\lambda_j)$ = true}
      - the smallest index for which $T(\lambda_j)$ = true
      - j exists because T satisfies F

Cook's theorem

- $T'(y_i)$ = true, $i \leq j - 2$
- $T'(y_i)$ = false, otherwise
- it is easy to see that T' satisfies the clauses corresponding to C (if C is satisfiable)
  - $(\lambda_1 \vee \lambda_2 \vee y_1)$ states λ_1 = true or λ_2 = true or the rest of λ is true
  - $(y_{k-3}{}^{C} \vee \lambda_{k-1} \vee \lambda_k)$ states if y_{k-3} = true $\rightarrow$ λ_{k-1} = true or λ_k = true

Cook's theorem

- Definition of Max set problem: given a set F of clauses, and an integer K, ∃ a truth assignment that satisfies at least K of the clauses?
- Theorem: Max set is NΠ-complete
- Proof:
  - Max set problem is a generalization of Satisfiability
    - an instance of Satisfiability can be thought of as an instance of Max set problem with special K
      - K = the number of clauses in F

Cook's theorem

- Construction:
  - given an instance F of Satisfiability with m clauses
  - create (F, m), the input of the Max set problem, by appending to F the parameter K = m
- Proof:
  - suppose ∃ truth assignment T satisfying F
  - T also satisfies (F, m)
  - suppose ∃ truth assignment T' satisfying (F, m)
  - T' also satisfies F

Introduction to the Theory of Computation

Lesson 13

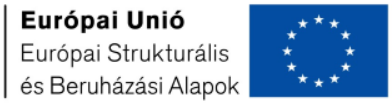
7.3. More NP-complete problems

Dr. István Heckl

Istvan.Heckl@gmail.com

University of Pannonia

MAGYARORSZÁG KORMÁNYA



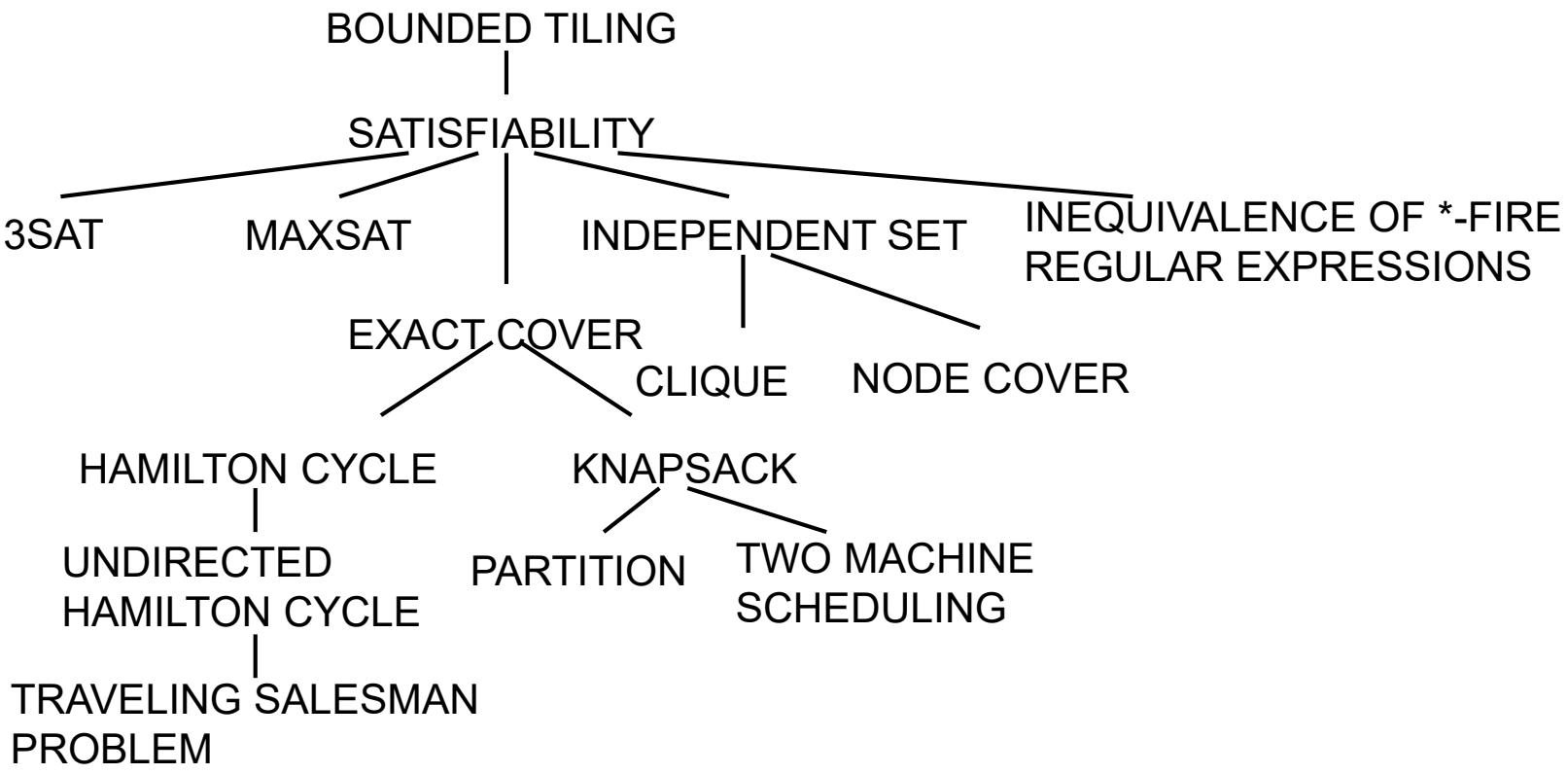
BEFEKTETÉS A JÖVŐBE

More NP-complete problems

- NP-complete problems are important because they
  - come up frequently under various guises in applications
    - do not try to devise an effective algorithm for an NP-complete problem
  - have been studied so much
  - are often handy starting points for NP-complete reductions
- Satisfiability is important for all three reasons

More NP-complete problems

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen



Exact cover

- Definition of the Exact cover problem:
  - given:
    - a finite set $U = \{u_1, \ldots, u_n\}$, called universe
    - $F = \{S_1, ..., S_m\}$, called family, $S_i \subseteq U$, $i = 1, 2 \ldots m$
  - is there an exact cover C called subfamily, such that
    - $C \subseteq F$
    - the sets in C are disjoint
    - $\cup C = U$

Exact cover

- Example:
  - universe U = $\{u_1, u_2, u_3, u_4, u_5, u_6\}$
  - family F = $\{\{u_1, u_3\}, \{u_2, u_3, u_6\}, \{u_1, u_5\}, \{u_2, u_3, u_4\}, \{u_5, u_6\}, \{u_2, u_4\}\}$
  - exact cover C = $\{\{u_1, u_3\}, \{u_5, u_6\}, \{u_2, u_4\}\}$

Exact cover

- Theorem: Exact cover is $\mathrm{N}\Pi$-complete
- Proof:
  - Exact cover is in $\mathrm{N}\Pi$
    - C is a valid certificate of (U, F)
    - C is polynomilally concise
      - C is part of (U, F)
      - can be checked in polynomial time if C is really an exact cover

Exact cover

- Construction:
  - reduce Satisfiability to Exact cover
  - given a Boolean formula F with clauses $(C_1, ..., C_l)$ over the Boolean variables $x_1, ..., x_n$
    - beware F has two meaning now
  - we show how to construct in polynomial time an equivalent instance $\tau(F)$ of the Exact cover problem
  - let $\lambda_{j,k}$ the k^{th} literal of clause C_j
    - C_j has m_j literal

Exact cover

- the universe of $\tau(F)$: $U = \{x_i : 1 \le i \le n\} \cup \{C_j : j = 1, ..., l\} \cup \{p_{j,k} : 1 \le j \le l, k = 1, ..., m_j\}$
  - one element for each Boolean variable
  - one for each clause
  - one for each position in each clause
- the family of $\tau(F)$:
  - $\{p_{j,k}\}$ for $\forall\ p_{j,k}$
    - $p_{j,k}$'s are easy to cover
  - $T_{i,true} = \{x_i\} \cup \{p_{j,k} : \lambda_{j,k} = x_i^C\}$
    - for $\forall\ x_i$
    - it contains the position for all negative occurrences of x_i

Exact cover

- $T_{i,false} = \{x_i\} \cup \{p_{j,k}: \lambda_{j,k} = x_i\}$
  - for $\forall$ x_i
  - it contains the position for all positive occurrences of x_i
- $\{C_j, p_{j,k}\}$ for $\forall$ clause C_i, for $\forall$ literal in it

Exact cover

- Lemma: τ(F) has an exact cover ↔ F is satisfiable
- Proof, →:
  - suppose that an exact cover C exists
    - beware C has two meanings now
  - construct truth assignment T as:
    - if $T_{i,true} \in C \rightarrow T(x_i) = true$
    - if $T_{i,false} \in C \rightarrow T(x_i) = false$
      - beware T has two meaning now
    - since x_i must be covered exactly once, either $T_{i,true} \in C$ or $T_{i,false} \in C$

Exact cover

- T satisfies F
  - consider clause C_j
  - C_j must be covered by one such set $\{C_j, p_{j,k}\}$
  - $p_{j,k}$ must appear in either $T_{i,true}$ or $T_{i,false}$ for some i
    - see the definition of $T_{i,true}$ and $T_{i,false}$

Exact cover

- suppose that the k^{th} literal in C_j is x_i
  - then $T_{i,false}$ contains $p_{j,k}$
  - exact cover C must contain $T_{i,true}$
    - » which does not contain $p_{j,k}$
  - according to the truth assignment $T(x_i)$ = true
  - T makes true the k^{th} literal of C_j
  - if you suppose that the k^{th} literal in C_j is x_i^C then $T(x_i)$ = false, so C_j is true again
- the previous is true for $\forall$ C_j, hence F is satisfiable

Exact cover

- Proof, ←:
  - suppose that truth assignment T satisfies F
  - construct an exact cover C by selecting sets:
    - $T_{i,true}$ if $T(x_i)$ = true
    - $T_{i,false}$ if $T(x_i)$ = false
      - for $\forall$ x_i
    - $\{C_j, p_{j,k}\}$, k should be such that the k^{th} literal of C_j is made true by T (there may be several such literal) and $p_{j,k}$ have not occurred yet in C
      - for $\forall$ C_j
    - $\{p_{j,k}\}$ for such p's which has not occurred yet
  - C covers τ(F) as the x, C, and p variables are covered

Exact cover: Example

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

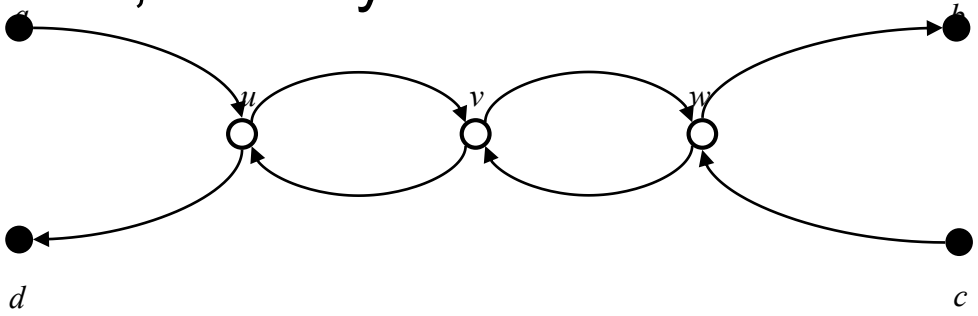
- Formulae $F = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where $C_1 = (x_1 \vee x_2^C)$, $C_2 = (x_1^C \vee x_2 \vee x_3)$, $C_3 = (x_2)$, $C_4 = (x_2^C \vee x_3^C)$
- Universe $U = \{x_1, x_2, x_3, x_4, C_1, C_2, C_3, C_4, p_{1,1}, p_{1,2}, p_{2,1}, p_{2,2}, p_{2,3}, p_{3,1}, p_{4,1}, p_{4,2}\}$
- Family F = {
  - $\{p_{1,1}\}, \{p_{1,2}\}, \{p_{2,1}\}, \{p_{2,2}\}, \{p_{2,3}\}, \{p_{3,1}\}, \{p_{4,1}\}, \{p_{4,2}\}$
  - $T_{1,true} = \{x_1, p_{2,1}\}$, $T_{1,false} = \{x_1, p_{1,1}\}$
  - $T_{2,true} = \{x_2, p_{1,2}, p_{4,1}\}$, $T_{2,false} = \{x_2, p_{2,2}, p_{3,1}\}$
  - $T_{3,true} = \{x_3, p_{4,2}\}$, $T_{3,false} = \{x_3, p_{2,3}\}$
  - $\{C_1,p_{1,1}\}, \{C_1, p_{1,2}\}, \{C_2, p_{2,1}\}, \{C_2, p_{2,2}\}, \{C_2, p_{2,3}\}, \{C_3, p_{3,1}\}, \{C_4, p_{4,1}\}, \{C_4, p_{4,2}\}$ }

Exact cover

- T truth assignment: $T(x_1)$ = true, $T(x_2)$ = true, $T(x_3)$ = false
- Exact cover C = $\{T_{1,true}, T_{2,true}, T_{3,false}, \{C_1,p_{1,1}\}, \{C_2, p_{2,2}\}, \{C_3, p_{3,1}\}, \{C_4, p_{4,2}\}, \{p_{1,2}\}, \{p_{2,1}\}, \{p_{2,3}\}, \{p_{4,1}\}\}$

Hamilton cycle

- Definition of gadget: a graph displayed with abbreviation
  - consider the next sub-graph of graph G
    - there is a series of node: u, v, w, ...
    - there are forward and backward arcs between (u, v), (v, w), ...
  - only the first and the last nodes in the series are connected to the rest of the graph through nodes a, b c, and d, the way as it is indicated

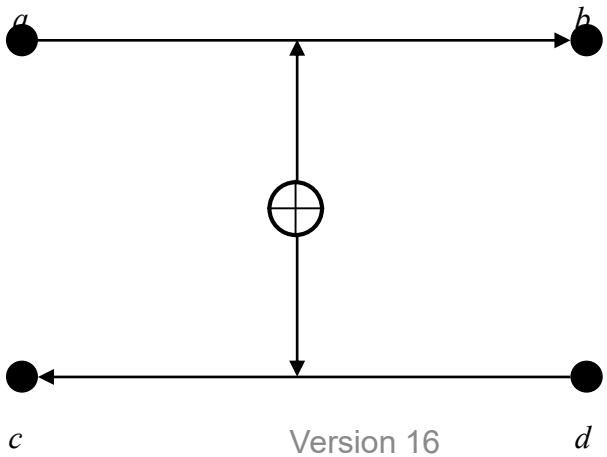


Hamilton cycle

- search for the Hamilton cycle of G
  - a cycle which traverses each node exactly once
- via which edges is the Hamilton cycle is going to traverse the middle nodes: u, v, w?
  - (a, u), (u, v), (v, w), (w, b) or
  - (c, w), (w, v), (v, u), (u, d)
  - all other possibilities omit some nodes or traverse through a node more than once

Hamilton cycle

- sequence (a, u), (u, v), (v, w), (w, b) and (c, w), (w, v), (v, u), (u, d) can be replaced by (a, b) and (c, d) respectively
- in any Hamilton cycle of G either (a, b) is traversed or (c, d) is, but not both (exclusive or)
- this situation is depicted in the figure where the two edges are connected by an exclusive or by a sign



Hamilton cycle

- Theorem: Hamilton cycle is $\mathrm{N}\Pi$-complete
- Proof:
  - we already know that it is $\mathrm{N}\Pi$
  - reduce Exact cover to Hamilton cycle with a polynomial algorithm

Hamilton cycle

- Construction:
  - given an instance (U, F) of Exact cover
    - $U = \{u_1, ..., u_n\}$ and $F = \{S_1, ..., S_m\}$
  - we shall describe a polynomial-time algorithm which produces a directed graph $G = \tau(U, F)$
  - graph G has nodes: $u_0, u_1, ..., u_n$ and $S_0, S_1, ..., S_m$
    - one for each element of the universe, and for each set in the family, plus two more nodes

Hamilton cycle

- graph G has two (S_{i-1}, S_i) arcs, i = 1, ..., m
  - two different edges connecting the same pair of nodes are needed because of the exclusive or
  - one of them is called the long edge, the other is the short edge
- graph G has k (u_{j-1}, u_j) arcs, j = 1, ..., n, where k is the number of appearance of u_j in the S sets
- graph G has arcs (u_n, S_0) and (S_m, u_0)

Hamilton cycle

- join $\forall$ copy of edge (u_{j-1}, u_j) with the long edge (S_{i-1}, S_i) by an exclusive or such that $u_j \in S_i$
  - as each copy of edge (u_{j-1}, u_j) corresponds to an appearance of u_j in some S_i

Hamilton cycle

- Lemma: τ(U, F) has a Hamilton cycle ↔ (U, F) has an exact cover
- Proof, →:
  - suppose that τ(U, F) has a Hamilton cycle
    - do not forget that the exclusive or constraints must hold
      - if we ignore them it is easy to give Hamilton cycle
    - the cycle contains the nodes in this order: S_0, S_1, ..., S_m, u_0, u_1, ..., u_n, S_0
      - there are no backward arcs and the starting point is irrelevant

Hamilton cycle

- let C be the set of all such S_i sets where the short edge (S_{i-1}, S_i) is traversed by the Hamilton cycle
- the Hamilton cycle traverses exactly one of the copies of (u_{j-1}, u_j) for $\forall u_j \in U \rightarrow \forall u_j$ is contained in exactly one S_i set in C
  - there may be several S_i sets which contains u_j
  - there is one S_i such that $u_j \in S_i \in C$
    - if the Hamilton cycle traverses a copy of $(u_{j-1}, u_j) \leftrightarrow$ it does not traverse the long edge (S_{i-1}, S_i) connected to the previous edge through the exclusive or

Hamilton cycle

- if the Hamilton cycle does not traverse the long edge $(S_{i-1}, S_i) \leftrightarrow$ it do traverses the short edge
  (S_{i-1}, S_i) because there are just two arcs between S_{i-1}, S_i
- if the Hamilton cycle traverses the short edge $(S_{i-1}, S_i) \rightarrow S_i \in C$ by definition

Hamilton cycle

- there is no more than one S_i such that $u_j \in S_i \in C$
  - if the Hamilton cycle traverses a copy of (u_{j-1}, u_j) $\leftrightarrow$ it does not traverse the other copies
  - if the Hamilton cycle does not traverse the other copies of (u_{j-1}, u_j) $\leftrightarrow$ it traverses the long edges (S_{i-1}, S_i) connected to them through the exclusive or
  - if the Hamilton cycle traverses the long edges (S_{i-1}, S_i) $\leftrightarrow$ it does not traverse the short edges (S_{i-1}, S_i)
  - if the Hamilton cycle does not traverse the short edges (S_{i-1}, S_i) $\leftrightarrow$ these S_i are not in C by definition

Hamilton cycle

- if $\forall$ u_j is contained in exactly one S set in C → C is an exact cover by definition
  - the union of the S sets in C results in U without overlap

Hamilton cycle

– conversely, suppose that an exact cover C exists then a Hamilton cycle in the graph τ(U, F) can be constructed as follows:
  - traverse the short copies of all edges (S_{j-1}, S_j) where $S_j \in C$, and the long edges for all other sets
  - for each element u_i traverse the copy of the edge (u_{i-1}, u_i), that corresponds to the unique set in C that contains u_i
  - complete the Hamilton cycle by the edges (u_n, S_0) and (S_m, u_0)

More NP-complete problems: Undirected Hamilton cycle

- Research, in connection to NP-complete problems, often is focused on whether the instance being much less complex, may not be solvable by an efficient algorithm
- Alternatively, it can be often shown that even substantially restricted versions of the problem remain NP-complete
- Example:
  Undirected Hamilton cycle - the Hamilton cycle problem restricted to graphs that are undirected, that is, symmetric without self-loops

More NP-complete problems: Undirected Hamilton cycle

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Theorem: Undirected Hamilton cycle is NP-complete
- Proof:
  - we reduce Hamilton cycle to it
  - given a graph $G \subseteq V \times V$, and we shall construct a symmetric graph $G' \subseteq V' \times V'$ without self-loops
  - G has a Hamilton cycle if and only if G' has one
  - $V' = \{v_0, v_1, v_2 : v \in V\}$ that is, G' has three nodes v_0, v_1, v_2 for each node v of G
    - v_0 is the entry and v_2 is the exit node

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- thus the edges in G': $\{(u_2, v_0), (v_0, u_2) : (u, v) \in G\} \cup \{(v_0, v_1), (v_1, v_0), (v_1, v_2), (v_2, v_1) : v \in V\}$
  - that is the nodes v_0, v_1, v_2 are connected by a path in this order, and there is an undirected edge between u_2 and v_0 whenever $(u, v) \in G$
- now we must prove that G' has a Hamilton cycle if and only if G has one
  - suppose that a Hamilton cycle of G' arrives at node v_0 from an edge of (u_2, v_0)

More NP-complete problems: Undirected Hamilton cycle

- if Hamilton cycle leaves v_0 through an edge other than (v_0, v_1) then it cannot pick up node v_1 (and it won't be a Hamilton cycle then)
- thus, edge (v_0, v_1) must be a part of the cycle, and so is (v_1, v_2)
- then the cycle has to continue through one of the edges (v_2, w_0) where $(v, w) \in G$
- and then to (w_1, w_2), to some z_0, where $(w, z) \in G$ etc.

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- therefore the edges of (u_0, v_2) in the Hamilton cycle of G' constitute in fact a Hamilton cycle of G
- conversely, any Hamilton cycle $(v^1, v^2, ..., v^{|V|})$ of G can be converted into a Hamilton cycle of G' as: $(v_0^1, v_1^1, v_2^1, v_0^2, v_2^2, ..., v_0^{|V|}, v_1^{|V|}, v_2^{|V|})$

More NP-complete problems: Traveling Salesman

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Theorem: The Traveling Salesman Problem is NP-complete
- Proof:
  - we already know that it is in NP
  - we reduce Undirected Hamilton cycle to it
  - given a symmetric graph G
  - we construct an instance of the Traveling Salesman

More NP-complete problems: Traveling Salesman

- n, the number of cities, |V|, the distance d_{ij} between any two cities i and j is
  $d_{ij} = 0$, if $i = j$
  $d_{ij} = 1$, if $(v_i, v_j) \in G$
  $d_{ij} = 2$, otherwise
- since G is a symmetric graph without loops, $d_{ij} = d_{ji}$ for all cities i and j
- budget B = n

More NP-complete problems: Traveling Salesman

- any tour of the cities has cost equal to the number of n plus the number of the intercity distances traversed that are not edges of G
- thus, a tour of cost B or less exists if and only if the tour is a Hamilton cycle of G

More NΠ-complete problems: Knapsack

- Theorem: Knapsack is NΠ-complete
- Proof:
  - we know that Knapsack is in NΠ
  - we reduce Exact Cover to Knapsack
  - given an instance of Knapsack a_1, ..., a_n, and K, a subset P of {1, ..., n} such that $\Sigma_{i \in P} a_i = K$ can serve as a certificate that the answer to the given instance is yes
  - we are given a universe $U = \{u_1, ..., u_n\}$, and a family $F = \{S_1, ..., S_m\}$ of subsets of U

More NΠ-complete problems: Knapsack

- we shall construct an instance τ(U, F) of Knapsack, that is
  - nonnegative integers a_1, ..., a_k and another K such that,
  - there is a subset $P \subseteq \{1, ..., k\}$ with $\Sigma_{i \in P} a_i = K$
  - if and only if there is a set of sets $C \subseteq F$ that are disjoint and collectively cover all of U
- this construction relies on an unexpected relationship between set union and integer addition
  - subsets of a set of n elements can be represented as strings over $\{0, 1\}^n$

- such strings can be interpreted as integers between 0 and 2^n - 1, written in binary
- the union of such sets is the same as adding the corresponding integers

– the question whether the disjoint union of the teams makes up the whole U (Exact Cover), seems to be the same as whether there are integers among the given ones that add up to K = 1 + 2 + 4 + … + 2^{n-1} - the binary number with n ones

More NP-complete problems: Knapsack

$S_1 = \{u_3, u_4\}$ $S_1 = 0011$ 0011
$S_2 = \{u_2, u_3, u_4\}$ $S_2 = 0111$ 0111
$S_3 = \{u_1, u_2\}$ $S_3 = 1100$ 1100
1111

- there is only one problem with this correspondence, because in integer addition we may have a carry bit
- e.g. the sum 11 + 13 + 15 + 24 = 63, in binary 001011 + 001101 + 001111 + 011000 = 111111
  if we translate back to subsets of $\{u_1, ..., u_6\}$, the sets
  $\{u_3, u_5, u_6\}$, $\{u_3, u_4, u_6\}$, $\{u_3, u_4, u_5, u_6\}$ and $\{u_2, u_3\}$ are neither disjoint, nor do they cover all of U

– but this problem can be resolved as considering the strings in $\{0, 1\}^n$ as integers not in binary, but in m-ary
  - m is the number of sets in F
  - we have m integers $a_1, \ldots, a_m$, where $a_i = \Sigma_{u_j \in S_i} m^{j-1}$
  - we ask whether there is a subset that adds up to $K = \Sigma_{j=1} m^{j-1}$

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- this way carry is not a problem, because the addition of fewer than m digits in m-ary, with each of the digits either 0 or 1, can never result in carry

– so the resulting instance of Knapsack has a solution if and only if the original instance of Exact Cover has a solution

More NP-complete problems: Two-Machine Scheduling

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Corollary: Partition and Two-Machine Scheduling are NP-complete
- Proof:
  - there are polynomial reductions from Knapsack to both of these problems

More NP-complete problems: Independent Set

- Theorem: Independent Set is NP-complete
- Proof:
  - we know that it is in NP
  - we reduce 3-Satisfiability to it
    - we are given a Boolean formula F with clauses C_1, ..., C_m, each with at most three literals (if a clause has only one or two literals we allow a literal to be repeated to make them three)
    - we construct an undirected graph G and an integer K, such that there is a set of K nodes in G with no edges between them if and only if F is satisfiable

More NP-complete problems: Independent Set

- for each one of the clauses C_1, ..., C_m of F, we have three nodes in G (c_{i1}, c_{i2}, c_{i3}), connected by edges so that they form a triangle
- these are all the nodes of G - a total of 3m nodes
- the goal is K = m
- for defining the remaining edges of G, node c_{ij} is identified with the j^{th} literal of clause C_i
- two nodes are joined by an edge if and only if their literals are the negation of one another

– suppose there is an independent set I in G with K = m nodes

More NP-complete problems: Independent Set

- there is exactly one node in I from each triangle (since any two nodes from the same triangle are connected by an edge)
- a node in I means that the corresponding literal is T
- there are no edges between nodes in I means, that no two such literals are the negation of one another, therefore they can be the basis of a true assignment T
- T may not be fully defined on all variables, for the set of nodes in I may fail to involve them all
- T may take any truth value on such missing variables, so the resulting truth assignment T satisfies all clauses

More NP-complete problems: Independent Set

– conversely, given any truth assignment satisfying F, we can obtain an independent set of size m by picking for each clause a node corresponding to a satisfied literal

More NP-complete problems: Clique and Node Cover

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Theorem: Clique and Node Cover are NP-complete
- Proof:
  - they are both in NP
  - Clique: requires that all edges between any two nodes in the set be present

More NP-complete problems: Clique and Node Cover

- we reduce Independent Set to Clique
  - given an instance (G, K) of Independent Set
    $G \subseteq V \times V$ is an undirected graph
    $K \geq 2$ is the goal
  - create (G', K') Clique
    - $G' = V \times V - \{(i, i) : i \in V\} - G$
    - $K' = K$, we keep the same goal
  - the maximum independent set of G is precisely the maximum clique in G'

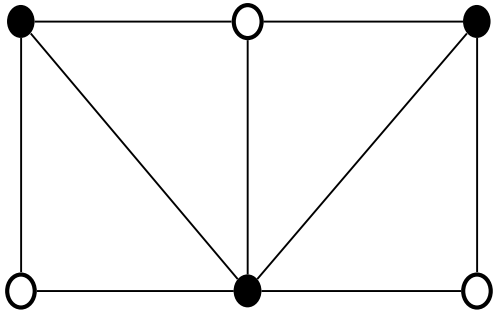
EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- we reduce Independent Set to Node Cover
- since the nodes in a node cover $N \subseteq V$ hit all edges, the set V - N must have no edges between it's elements, and thus is an independent set
  - $N \subseteq V$ is a node cover of G $\leftrightarrow$ V - N is an independent set of G
    - the maximum independent set of G has size K or more $\leftrightarrow$ the minimum node cover of G has size |V| - K or less
  - the reduction leaves the graph the same, simply replaces K by |V| - K

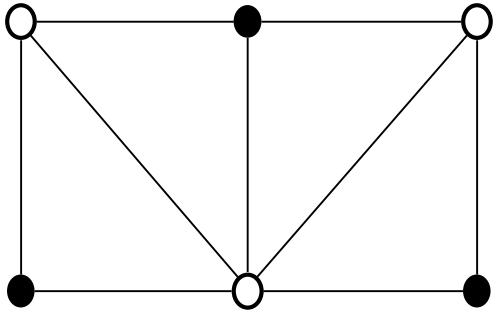
More NP-complete problems: Clique and Node Cover

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Node Cover:
  - the black dots belongs to it



- Independent Set:



More NP-complete problems: Inequivalence of regular expressions

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Remaining question: are two regular expressions (or two NFA) are equivalent?
  - consider the complementary problem
- Given two regular expressions R_1 and R_2, $L(R_1) \neq L(R_2)$?
  - $w \in (L(R_1) - L(R_2)) \cup (L(R_2) - L(R_1))$ is a certificate
  - the certificate should be polynomial succinct
    - in the length of the two expressions $|R_1| + |R_2|$
    - Kleene star causes problems

More NP-complete problems: Clique and Node Cover

- Definition of *-free regular expressions: regular expressions over union and concatenation, not containing any occurrences of Kleene star
  - e.g.: $R = (0 \cup 1)00(0 \cup 1) \cup 010(0 \cup 1)0$
  - if $x \in L(R) \rightarrow |x| \leq |R|$

More NΠ-complete problems: Inequialence of *-Free Regular Expressions

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Theorem: Inequivalence of *-Free Regular Expressions is NΠ-complete
- Proof:
  - we already know that the problem is in NΠ
  - we reduce Satisfiability to it
  - given any Boolean formula F with variables x_1, ..., x_n and clauses C_1, ..., C_m

More NP-complete problems: Inequialence of *-Free Regular Expressions

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- we produce *-free regular expressions R_1 and R_2, such that: $L(R_1) \neq L(R_2) \leftrightarrow$ the given Boolean formula is satisfiable
- R_2: $(0 \cup 1)(0 \cup 1)...(0 \cup 1)$
  - $(0 \cup 1)$ repeated n times
  - $L(R_2) = \{0, 1\}^n$, the set of all binary strings of length n

More NII-complete problems: Inequialence of *-Free Regular Expressions

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- R_1: $\alpha_1 \cup \alpha_2 \cup \ldots \cup \alpha_m$
  - $\alpha_i = \alpha_{i,1}\alpha_{i,2} \ldots \alpha_{i,n}$, where
    - $\alpha_{i,j} = 0$, if x_j is a literal of C_i
    - $a_{i,j} = 1$, if x_j^C is a literal of C_i
    - $a_{i,j} = (0 \cup 1)$, otherwise
- strings in $\{0, 1\}^n$ can be thought of as truth assignments to the Boolean variables $\{x_1, ..., x_n\}$

More NII-complete problems: Inequialence of *-Free Regular Expressions

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- $L(\alpha_i)$ is precisely the set of all truth assignments that fails to satisfy C_i
- $L(R_1)$ is the set of all truth assignments that fails to satisfy at least on of the clauses of the given Boolean formula
- the set of all truth assignments that satisfies the given Boolean formula is $L(R_1)^C$
- the given Boolean formula is satisfiable $\leftrightarrow L(R_1)^C \neq \emptyset \leftrightarrow L(R_1) \neq \{0, 1\}^n = L(R_2)$

- Corollary: the general equivalence problem for RE or NFA can only be harder then the special case

More NΠ-complete problems: DFA with minimal number of states

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Corollary: Unless Π = NΠ, there is no algorithm which (given a RE or a NFA) constructs the minimum-state equivalent DFA in time that is polynomial in the input and in the output

More NII-complete problems: DFA with minimal number of states

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Proof:
  - let M_n denote the simple n + 1 - state DFA accepting $\{0, 1\}^n$
    - there is no DFA with less state to accept $\{0, 1\}^n$
  - the formula F of the previous proof with n variables is unsatisfiable $\leftrightarrow$ $L(R_1) = \{0, 1\}^n$
    - the minimum-state DFA equivalent to R_1 is exactly M_n
  - indirection: suppose that an algorithm as described in the corollary exists, with a time bound of the form p(|x| + |y|), where

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- p is a polynomial
- x is the input of the algorithm
- y is the output of the algorithm

– we could solve Satisfiability in polynomial-time:
- given any Boolean formula F with n variables
- first, perform the reduction to obtain a regular expression R_1
- second, run on input R_1, the purported algorithm for $p(|R_1| + |M_n|)$ steps
- if the algorithm stops within time answer
  – no to Satisfiability, if the output is M_n
  – yes, otherwise

More NΠ-complete problems: DFA with minimal number of states

- if the algorithm does not stop within time $p(|R_1| + |M_n|) \rightarrow$ answer yes
  - the output can not be M_n any more

– since Satisfiability is NΠ-complete, we must conclude that Π = NΠ, if the algorithm does really exist

Coping with NP-completeness

- Once our problem has been shown to be NP-complete, what we can do?
  - examine only special cases
  - settle with approximation solutions
  - wait patiently for a solution
    - use stronger PC
- Special cases: examine if we really need to solve this problem in the full generality
  - e.g.: though Satisfiability is NP-complete, its special case 2-Satisifiability can be solved efficiently
  - often the special case turns out to be itself NP-complete, e.g. 3-Satisfiability

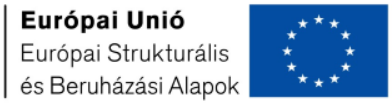
Introduction to the Theory of Computation

Lesson 14

7.4. Coping with NP-completeness

Dr. István Heckl
Istvan.Heckl@gmail.com
University of Pannonia

MAGYARORSZÁG KORMÁNYA



BEFEKTETÉS A JÖVŐBE

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Many NP-complete graph problems become trivial when the graph is a tree (it has no cycles)
- The trees have hierarchical structure:
  - pick an arbitrary root
  - let:
    - T(u) the subtree of node u
    - C(u) the children of node u
    - G(u) the grandchildren of node u

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- The Independent Set problem is easy when the graph is a tree
  - the size of the largest independent set of the tree can now be found by computing
    - $I(u) = \max \{\Sigma_{v \in C(u)} I(v), 1 + \Sigma_{v \in G(u)} I(u)\}$
  - this equation says that, in designing the largest independent set of T(u) we have two choices:
    - we do not put u into the independent set
      - we can put together all maximum independent sets in the subtrees of its children

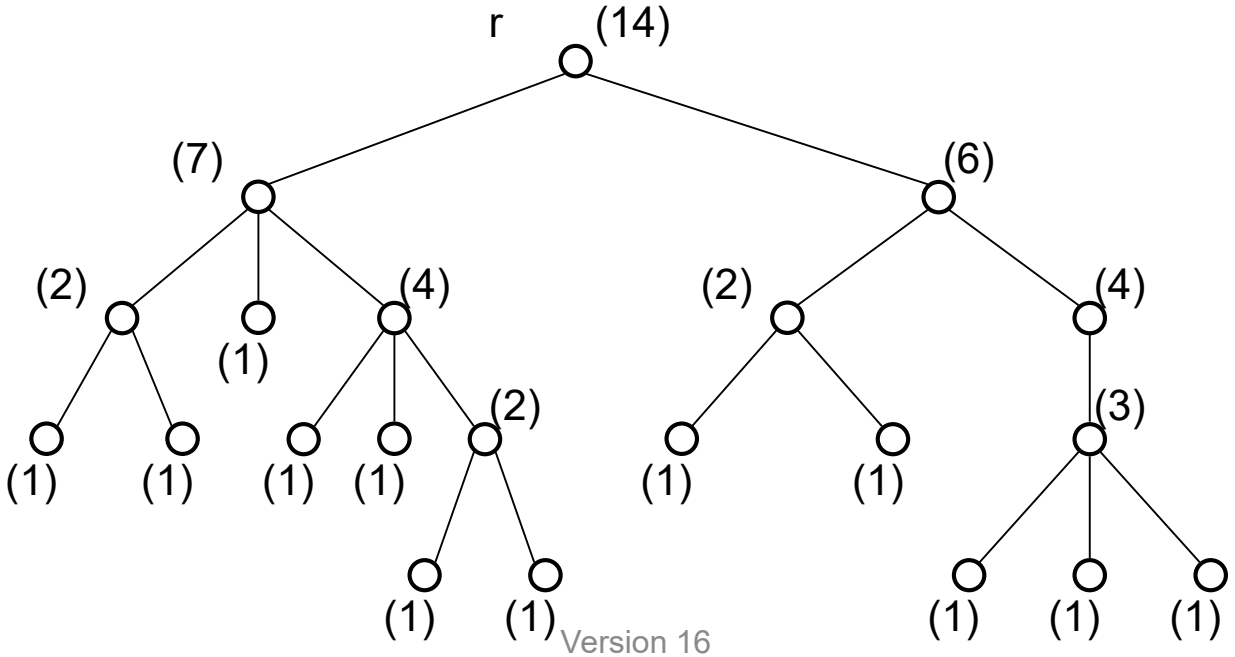
- we put u in the independent set but we must omit all its children, and assemble the maximum independent sets of the subtrees of all its grandchildren

– dynamic programming can calculate I(u)
  - start at the leaves (where I(u) is trivially one)
  - compute I(u) for larger and larger subtrees
    – reuse previously calculated I(u) values
  - the value of I at the root is the size of the maximum independent set of the tree

– the algorithm is polynomial

Example

- The root, denoted r, has two children and five grandchildren
  - the values of I(u) are shown in parentheses, the largest independent set of the tree has size 14



- Approximation solutions: use more simple algorithms producing solutions guarantied to be close to the optimum
- Definition of ε-approximation algorithm A:
  - given an NΠ-complete optimization problem
  - for each instance x the optimal solution is opt(x)
    - assume that opt(x) is always a positive integer
  - A is a polynomial algorithm, returning A(x)
  - the following inequality must hold:

$$\frac{|opt(x) - A(x)|}{opt(x)} \leq \varepsilon$$

Coping with NP-completeness

- All NP-complete optimization problems can be divided into groups:
  - fully approximable: ∃ algorithm for all ε > 0
    - e.g.: Two Machine Scheduling
  - partly approximable: ∃ algorithm for some range of ε's, but not down to zero
    - e.g.: Node Cover, Max Sat
  - inapproximable: there is no algorithm
    - e.g.: Traveling Salesman, Clique, Independent Set

- Theorem: ∃ 1-approximation algorithm for Node Cover
- Proof:

```
C := 0
while ∃ an edge [u, v] in G do
        add u, v to C, delete them from
G
```

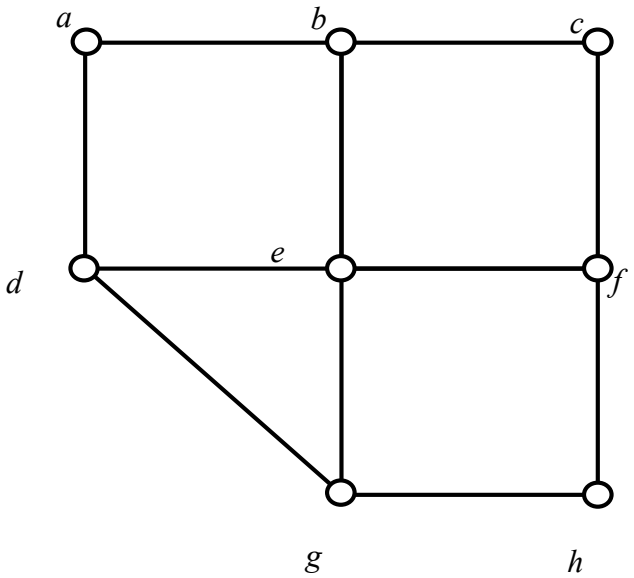
  - it returns a node cover that is at most twice the optimum size

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- let C the node cover returned by the algorithm and C' be the optimum node cover
- the number of the edges chosen by the algorithm is not larger than |C'|
  - at least one end point of each selected edge is in C' (see the definition of Node Cover)
  - if we remove |C'| edges from G, no full edge left
- |selected edges| ≤ |C'| → |C| ≤ 2 · |C'|
  - |C| is exactly twice the number of the edges chosen by the algorithm
- it is indeed a 1-approximation algorithm
  - |C'| = 3, |C| = 6 → |3 - 6| / 3 = 1

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- Select
  - (a, b)
  - (e, f)
  - (g, h)



- Theorem: Two-Machine Scheduling is fully approximable
- Proof:
  - the Partition problem can be solved in O(nS) time
    - n - the number of integers, S - their sum
  - adapt the algorithm to solve Two-Machine Scheduling (for finding the smallest D)
    - the last cycle should go until S (not just up to H = ½S)
    - the smallest sum in B(n) that is greater than or equal to ½S is the desired minimum D

- Acceleration:
  - round up these task lengths to the next hundred
  - normalize them by 100
  - we lost a little accuracy but we have decreased the time requirements a hundredfold
- Example:
  - task lengths:
    - 45362, 134537, 85879, 56390, 145627, 197342, 83625, 126789, 38562, 75402 with n = 10, and $S \approx 10^6$
  - solving it with the unaccelerated algorithm takes 10^7 steps

  - rounded numbers: 45400, 134600, 85900, 56400, 145700, 197400, 83700, 126800, 38600, 75500
  - normalized numbers: 454, 1346, 859, 564, 1457, 1974, 837, 1268, 386, 755
  - thus we can solve this instance in about 10^5 steps
- If we round up to the next k^{th} power of ten, the absolute error is no more than $n10^k$
  - to calculate the relative error, this quantity must be divided by the optimum, which can be not less than S/2
  - we have a $2n10^k/S$-approximation algorithm, with running time $O(nS/10^k)$
  - if we say $2n10^k/S$ = any desirable $\varepsilon > 0$, we will get running time $O(n^2/\varepsilon)$ - which is polynomial

Coping with NP-completeness

- Theorem: Travelling Salesman is inapproximable
- Proof:
  - revisit the reduction from the Hamilton Cycle, x, to Travelling Salesman, τ(x)
  - modify the reduction:
    - the distances $d_{i,j}$ are now the following:
      - $d_{i,j} = 0$, if $i = j$
      - $d_{i,j} = 1$, if $(v_i, v_j) \in G$
      - $d_{i,j} = 2 + n\varepsilon$ otherwise

- this construction means:
  - if x has a Hamilton cycle, then the optimum cost of τ(x) is n
  - else the optimum cost is greater than n - 1 + 2 + nε + 1= n(1 + ε)
    - at least one distance 2 + nε must be traversed, in addition to at least n - 1 others of cost at least 1
    - +1 comes from the greater

- assume ∃ a polynomial-time ε-approximation algorithm A for the Travelling Salesman problem
- run A on the previously constructed τ(x)
  - if $A(\tau(x)) \geq n(1 + \varepsilon) + 1 \rightarrow$ the optimum of τ(x) can not be n, so x has no Hamilton cycle
    - the relative error of A would be at least: $|n(1 + \varepsilon) + 1 - n| / n > \varepsilon$
  - if $A(\tau(x)) \leq n(1 + \varepsilon) \rightarrow$ the optimum solution of τ(x) must be n, so x has a Hamilton cycle
    - τ(x) is designed not to have a tour of cost between n + 1 and $n(1 + \varepsilon)$

EFOP-3.4.3-16-2016-00009
A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

- by applying A on τ(x) we can tell whether x has a Hamilton Cycle or not in polynomial time
- since Hamilton Cycle is NΠ-complete, the assumption is false

- The Travelling Salesman Problem, in which
  - distances $d_{i,j}$ satisfy the triangle inequality is partly approximable
    - $d_{i,j} \leq d_{i,k} + d_{k,j}$ for $\forall i, j, k$
    - the best known error bound is ½
  - cities are restricted to be points on the plane with the usual Euclidean distances, is fully approximable

Coping with NP-completeness

- All NP-complete problems are solvable by polynomially bounded nondeterministic Turing machines
  - we only know exponential methods to stimulate such machines
  - though we can not beat the exponential behavior in the worst case, we can create such algorithms which work more efficiently in average case

Coping with NP-completeness

- A typical NP-complete problem asks, if any member of a large set S_0 of candidate certificates (e.g. permutation of nodes) satisfies certain constraints (e.g. is a Hamilton path?)
  - $|S_0|$ depends exponentially on the size of the problem (e.g. the number of nodes in the graph)

Coping with NΠ-completeness

- The solving of an NΠ-complete problem by a nondeterministic TM produces a tree of configurations
  - corresponds some S to each configurations, a subset of S_0
    - S_0 belongs to the initial configuration
    - we call these subsets subproblems
    - if the children of configuration C are C_1, C_2, … and to these configurations correspond S, S_1, S_2, … respectively → $S = S_1 \cup S_2 \cup \ldots$
  - to tell if S has a valid certificate is a problem similar to tell if S_0 has one, but easier as $|S| < |S_0|$
    - this is called self-reducibility

Coping with NP-completeness

- If we can not tell anything to the original problem → replace it with its subproblems
- This suggests a genre of algorithms for solving NP-complete problems:
  - we always maintain a set of active subproblems, A
    - initially A = {S_0}
  - at each point we
    - choose a subproblem from A
    - replace it with its smaller subproblems
    - this is called branching

- each newly generated subproblem is submitted to a quick heuristic test, the answers can be:
  - empty: the subproblem has no solution, so it can be omitted
    - this is called backtracking
  - a solution of the original problem contained in the current subproblem → terminates
  - ?: we can not prove either of the previous two
    - add the subproblem in hand to A

Backtracking


```
A := {S0}
while A is not empty do
  choose a subproblem S
  delete S from A
  choose a way of branching out of S, say
    to subproblems S1, ..., Sr
  for each subproblem Si in this list
    do
      if test(Si) find solution then
        halt
      else if test(Si) returns "?" add Si
        to A
  return "no solution"
```

Backtracking

- The backtracking algorithm terminates because, in the end, the subproblems will contain just one candidate solution (a leaf of the tree)
- The effectiveness of the backtracking algorithm depends on three important design decisions:
  - how does one choose the next subproblem?
  - how is the chosen subproblem further split into smaller subproblems
  - which test is used?

Example

- Backtracking algorithm for Satisfiability
  - how to create subproblems: choose a variable x and fix its value
    - subproblem 1: T(x) = true
      - the clauses in which x appears are omitted because the whole clause is true
      - x^C is omitted from the clauses in which it appears because its value is fixed
    - subproblem 2: T(x) = false
      - the clauses in which x^C appears are omitted
      - x is omitted from the clauses in which it appears

Example

- how to choose the variable x on which to branch: select a variable that appears in the smallest clause
  - may soon lead to backtracking
- how to choose the next subproblem: select the subproblem that contains the smallest clause
- how to test a subproblem:
  - if ∃ an empty clause return "subproblem is empty"
  - if there are no clauses return "solution found"
  - otherwise return "?"

- Original problem:
  $(x \vee y \vee z), (x^C \vee y), (y^C \vee z), (z^C \vee x), (x^C \vee y^C \vee z^C)$
  - T(x) = true: (y), $(y^C \vee z)$, $(y^C \vee z^C)$
    - T(y) = true: (z), (z^C)
      - T(z) = true: ()
      - T(z) = false: ()
    - T(y) = false: ()
  - T(x) = false: $(y \vee z)$, $(y^C \vee z)$, (z^C)
    - T(z) = true: ()
    - T(z) = false: (y), (y^C)
      - T(y) = true: ()
      - T(y) = false: ()
- The problem is unsatisfiable

Example

- Designing a backtracking algorithm for Hamilton cycle
  - we already have: a path with endpoints 'a' and b, going through a set of nodes $T \subseteq V - \{a, b\}$
  - we are looking for: a Hamilton path from 'a' to b through the remaining nodes in V, to close the Hamilton cycle
  - branching: choose $c \notin T$ to extend the path by edge [a, c]
  - we do not specify how to choose the subproblem

Example

– the test:
  - if G - T - {a, b} is disconnected or G - T has a degree-one node other than a or b, return → "subproblem is empty"
  - if G - T is a path from a to b → return "solution"
  - otherwise return "?"
– at a specific graph with 8 nodes the backtrack algorithm examines only 19 subproblem while total enumeration consider all the leaves (n - 1)! = 5040
  - there is no guarantee for the time in the worst case it is still exponential
  - efficiency depends on the branching and on the selection strategies

Branch and bound

- An interesting variant of backtracking for optimization problems is called branch and bound
  - each solution has a cost associated with it, and we wish to find the candidate solution in S_0 with the smallest cost

Branch and bound

- The branch and bound algorithm:

```
A := {S0}, bestsofar := inf
while A is not empty do
    choose a subproblem S, delete it from
     A
    branch out of S to subproblems S1, …
     Sr
    for each subproblem Si do
     if |Si| = 1 → update bestsofar
      else if lowerbound(Si) < bestsofar
       → add Si to A
 return bestsofar
```

Branch and bound

- Operation
  - bestsofar: the smallest cost of any solution seen so far
    - initially infinite
    - it is an upper bound of the optimal solution of the original solution
  - if $|S_i| = 1 \rightarrow$ the optimal solution of the subproblem is the cost of the single solution in it
    - if this value is smaller than bestsofar → bestsofar is updated
  - lowerbound: a method for obtaining a value which is smaller than the cost of any solution in S
    - the sharper (bigger) the bound is the better

Branch and bound

- Main idea: if lowerbound(S_i) ≥ bestsofar → S_i is disregarded because it can not contain such solution whose cost is better than bestsofar

Local improvements

- Main idea: allow a solution of an optimization problem to change a little, and adopt the new solution if it has improved cost
  - let S_0 be the set of candidate solutions in an minimization problem
  - we define a neighbourhood relation $N \subseteq S_0 \times S_0$
    - it captures the intuitive notion changing a little
  - for $s \in S_0$, the set $\{s' : (s, s') \in N\}$ is called the neighbourhood of s

Local improvements

- The algorithm:

```
s := initialsolution
while ∃ a solution s' such that
(s, s') ∈ N and cost(s') < cost(s) do:
   s := s'
return s
```

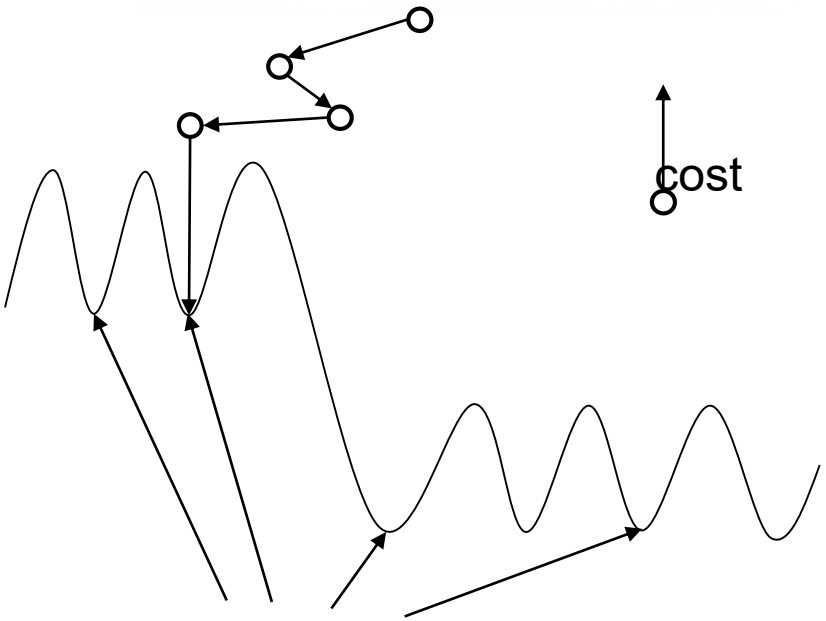
Local improvements

- Operation:
  - the algorithm keeps improving s by replacing with a neighbour s' with a better cost, until there is no more
  - we find only a local optimum
  - trade off: larger neighbourhoods
    - probably better local optimum
    - more iteration, so slower algorithm

Local improvements

- decision points:
  - the method used in finding s'
    - adopt the first better solution we find in the neighbourhood of s
    - wait to find the best neighbour
  - the procedure initialsolution
    - experience shows it should be randomized
    - after restart a different local optima may be obtained

Local improvements



local optima

- An optimization problem can be pictured as an energy landscape
  - local optima are depicted as valleys
  - local improvement heuristics jump from solution to solution, until a local optimum is found

Example

- Local improvement algorithm for the Travelling Salesman problem
  - tours are neighboring when they share all but very few links
    - 2-change algorithm: two tours related by N ↔ they differ in just two links
    - better results can be achieved by adopting the 3-change
    - 4-change does not seems to be advantageous
  - best heuristic algorithm currently known: Lin-Kernighan algorithm, it relies on λ-change

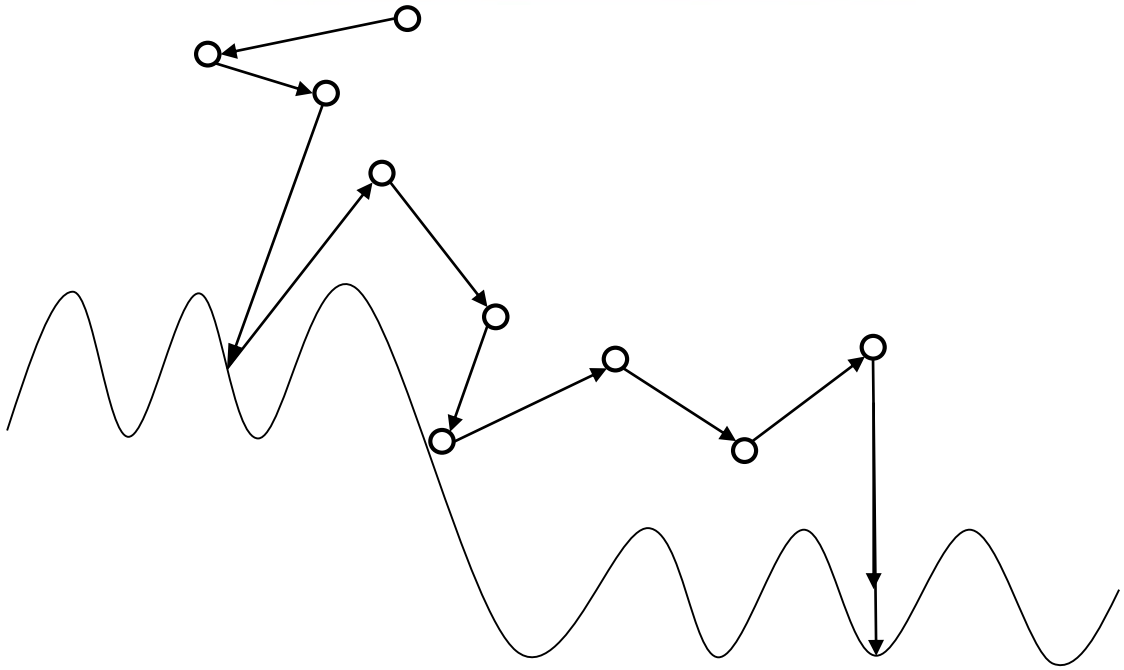
Example

- Local improvement algorithm for Max Set:
  - truth assignments are neighboring when they differ in the value of a single variable
  - experience shows that it is advantageous
    - to adopt as s' the best neighbour of s (instead of the first one found that is better than s)
    - to make lateral moves, that is to adopt a solution even if the new cost is the same as (and not less than) the old one

Simulated annealing

- Main idea: allow the algorithm to escape from bad local optima by performing occasional cost-increasing changes
  - the inspiration comes from the physics of cooling solids

Simulated annealing



- Advantage over the basic local improvement algorithm: because its occasional cost-increasing moves help it avoid early convergence in a bad local optimum
  - this often comes at loss of efficiency

Simulated annealing

- The algorithm:

```
s := initialsolution, T := T0
repeat
   generate randomly s' such that (s,
    s')∈ N, let Δ := cost(s') - cost(s)
   if Δ ≤ 0 then s := s' else
   s := s' with probability e^(-Δ/T)
   update(T)
until T = 0
return the best solution seen
```

Simulated annealing

- The probability that a cost increasing change will be adopted depends on
  - the amount of the cost increase Δ
  - the temperature T
    - as time passes, T decreases as well as the probability to accept a worse solution

Coping with NP-completeness

- Other local improvement methods:
  - genetic algorithms
  - neural networks
- Local improvement algorithms:
  - they do not in general return the global optimum
    - there is no guaranty for the quality of the returned solution
  - they have exponential worst-case complexity
- But for many NP-complete problems, in practice they turn out to be the ones that perform best

EFOP-3.4.3-16-2016-00009

A felsőfokú oktatás minőségének és hozzáférhetőségének együttes javítása a Pannon Egyetemen

Thank you for your attention!

SZÉCHENYI 2020

MAGYARORSZÁG KORMÁNYA

Európai Unió
Európai Strukturális és Beruházási Alapok

BEFEKTETÉS A JÖVŐBE