today

- algorithms, instances, correctness
- efficiency, order of growth, tractability
- a paradigm: exhaustive search
  - pseudocodes
- design of algorithms (always!)

chapter 2, section 1 and 2 from
Jon Kleinberg, Eva Tardos

Algorithm design

any edition, any printing
today

• algorithms, instances, correctness
• efficiency, order of growth, tractability
• a paradigm: exhaustive search
  – pseudocodes
• design of algorithms (always!)

chapter 2 (3) and most of chapter 1 (2) ⭐ from
Thomas. H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Introduction to algorithms

any edition, any printing

⭐ green chapter numbers for the second edition!
some logistics

• sessions:
  - two hours of lectures on average per week
    • compensate in block C
  - one hour of instruction on average per week
    • but some skipping in block C
    • one extra session of instruction before the exam
  - sheets of the week on:
    http://www.es.ele.tue.nl/education/5MC10/sheets.htm

• exam:
  - written exam in january:
    • problem from electrical engineering
    • statements about complexity of the problem
    • algorithm for a relaxed problem in pseudocode
    • analysis of algorithm
  - at home:
    • implementation of algorithm
    • defense of deviations from written exam
    • demonstration on given example
a nand gate
a simple example

\[ ab + c + d + (f+g) (e+h) \]
a simple example
an assignment problem

given:
- a symbolic transistor-network \((T, N, I)\)
- a sequence of successive gate sites \(S\)
- \(|T| = |S|\)

required
- an assignment \(T \rightarrow S\),
  such that \((T, N, I)\) can be completed with metal strips
an algorithm is correct
if, for every instantiation of the problem,
the algorithm stops after a finite number of steps,
and the output is a correct answer.
a small example

consider an example network

with a possible assignment

check:

- a has a contact in common with b
- b has a contact in common with c

conclusion:

the assignment is consistent!

however,

the first attempt does not lead to a correct algorithm
second attempt

• generate all assignments of \( m \) oriented transistors to \( m \) sites
• check for each assignment whether transistors assigned to neighbors are incident to the same node with the right contact
• if so, complete the network with metal strips

is this "algorithm" correct? YES, but

• many assignments are going to be generated: \( m! \ 2^m \)
  - or is the algorithm sufficiently efficient?
• many assignments do not really have a chance
  - can we avoid most "pointless" (partial) assignments
second attempt

- generate all assignments of \( m \) oriented transistors to \( m \) sites
- check for each assignment whether transistors assigned to neighbors are incident to the same node with the right contact
- if so, complete the network with metal strips

is this "algorithm" correct?

- many assignments are going to be generated: \( m! \cdot 2^m \)
  - or is the algorithm sufficiently efficient?
- many assignments do not really have a chance
  - can we avoid most "pointless" (partial) assignments

To conclude inefficiency we need an lowerbound on runtime
To conclude efficiency we need an upperbound on runtime
asymptotic bounds

For $n > n_o$, $f(n)$ always stays above $c g(n)$ \hspace{1cm} (c > 0)

For $n > n_o$, $f(n)$ always stays below $c g(n)$ \hspace{1cm} (c > 0)

an algorithm is efficient if the number of steps to finish grows at most polynomially with the input size on a random access machine
asymptotic tight bounds

for \( n > n_0 \), \( f(n) \) always stays in between \( c_1 g(n) \) and \( c_2 g(n) \) (\( c_1, c_2 \) positive constants)

an algorithm is efficient if the number of steps to finish grows at most polynomially with the input size on a random access machine
second attempt

- generate all assignments of \( m \) oriented transistors to \( m \) sites
- check for each assignment whether transistors assigned to neighbors are incident to the same node with the right contact
- if so, complete the network with metal strips

Is this "algorithm" **correct**?  **YES, but**

- many assignments are going to be generated: \( m! \cdot 2^m \)
  - or is the algorithm sufficiently **efficient**?
- many assignments do not really have a chance
  - can we avoid most "pointless" (partial assignments)
systematic configuration generation

required:

a finite sequence of objects \((a_1, a_2, a_3, \ldots)\)
from \(E_1 \times E_2 \times E_3 \times \ldots\)
satisfying certain constraints

example:

- elementary objects: oriented transistors
- constraints: common nodes, no duplications

a partial sequence \((a_1, a_2, \ldots, a_{k-1})\)
a candidate set \(S_k \subseteq E_k\)

\(a_k \in S_k\) if \(a_k \in E_k\), has not been tried and is consistent
pseudo code

- block structure by indentation
- control structures: iteration (`while`, `for`, `repeat`) and conditional (`if ... then ... else`)
- comments: after `→`
- assignment: `←` also multiple assignments
- variables: local to the given procedure unless explicitly indicated
- arrays:
  - elements accessed by index between square brackets
  - attributes are named with the object in square brackets
- parameters are passed by value
the backtracking paradigm

\[ S_1 \leftarrow E_1 \]

\[ k \leftarrow 1 \]

\[ \text{while } k > 0 \quad \text{do} \]

\[ \text{while } S_k \neq \emptyset \quad \text{do} \]

\[ a_k \leftarrow \text{element in } S_k \]

\[ S_k \leftarrow S_k \setminus \{a_k\} \]

\[ \text{if } (a_1, a_2, \ldots, a_k) \text{ is a solution} \]

\[ \text{then put on the list} \]

\[ k \leftarrow k + 1 \]

\[ \text{determine } S_k \]

\[ k \leftarrow k - 1 \]

\[ \text{return the list!} \]
exhaustive assignment generation
reformulation

critique of exhaustive assignment generation
  • a graph can have exponentially many paths
  • most of the work is done when there is no solution

input:
a graph with the nodes as vertices, the transistors as edges

output:
a path in that graph in which every edge occurs exactly once

attempt nr 4:
  • select a node as a starting point
  • traverse as many edges as possible to form a maximal path
  • if all edges are in the path, report the solution
maximal path generation

**MAXPAD** \((V, E, v)\)

\[
P ← \emptyset \\
U ← E \\
w ← v
\]

**while** \(I(w) \cap U \neq \emptyset\) **do**

\[
e ← \text{element van } I(w) \cap U \\
U ← U \setminus \{e\} \\
P ← P \cup e \\
w ← \text{het andere element van } e
\]

**return** \(P\)
invariants for some special cases

all vertices have even degree:

• any "MAXPAD" ends in the start vertex
• removal of all edges of "MAXPAD" from the graph leaves a graph with only even-degree vertices

exactly two vertices have odd degree:

• any "MAXPAD" starting at an odd-degree vertex ends in the other odd-degree vertex
• removal of all edges of that "MAXPAD" from the graph leaves a graph with only even-degree vertices
inserting maximal paths

EXTEND ( V, E, v )
P ← MAXPAD ( V, E, v )
if P ⊆ E
then w ← P ∩ E \ P
P ← P ∪ EXTEND ( V, E \ P, w )
return P
köningsbergen
evaluation

- correct: if at most two vertices have odd degree, then "EXTEND" produces the required path
  - if more than two vertices have odd degree, then no such path can exist

- efficient: create a doubly-linked list for the first P; perform the if-clause for every vertex on P, progressing in spite of a changing P
  - constant amount of work per edge $\rightarrow O(m)$

- solution: what if the transistor network leads to a graph with more than two odd-degree vertices?
tricks and treats

what if the network graph has no euler path?

for example:
\[ a ( b + c ) + ( d + e ) f \]

logically equivalent with
\[ a ( b + c ) + f ( d + e ) \]