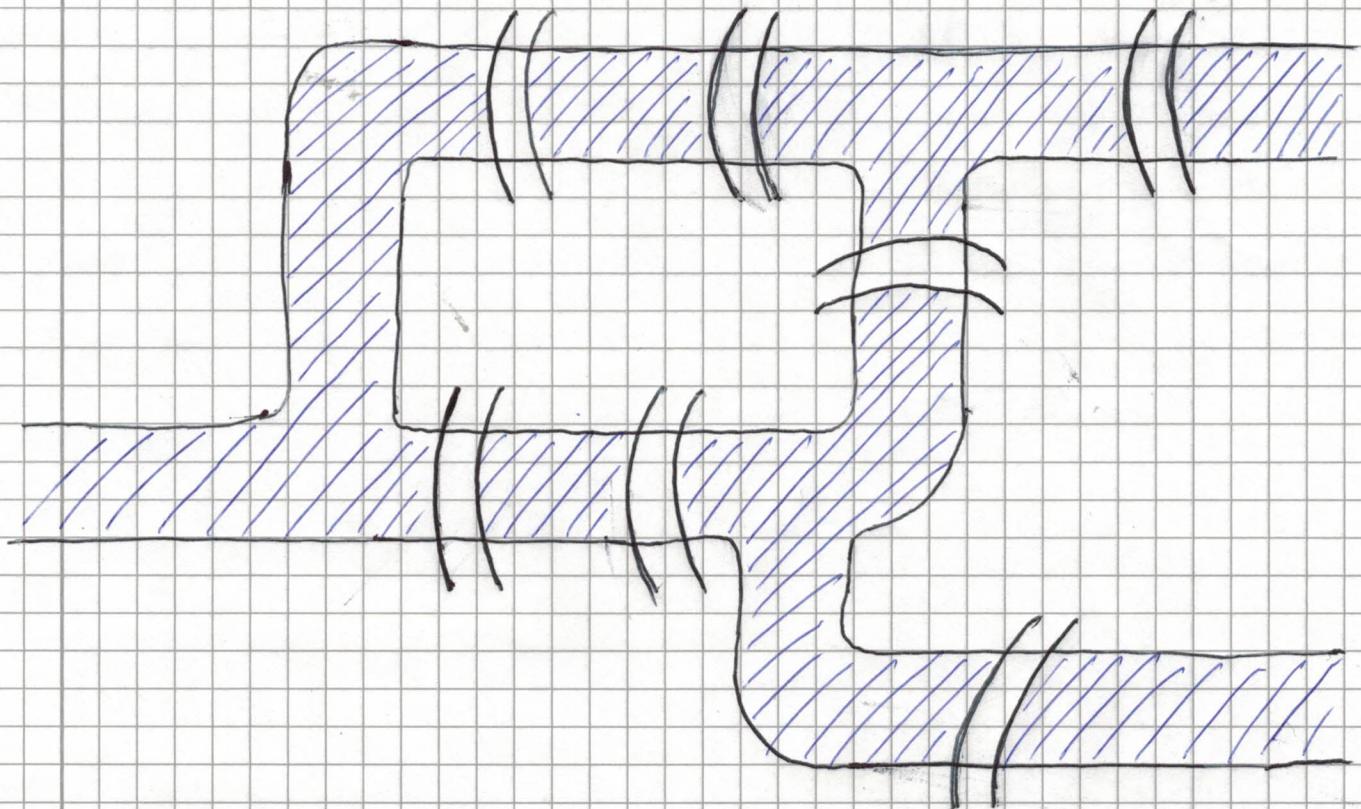


7 bridges of Königsberg (now Kaliningrad)

Problem solved by Euler in 1736

Laid the foundations for graph theory  
(central topic in algorithms)



PROBLEM:

Walk across all bridges exactly once  
and return to starting position

# ALGORITHM DESIGN AND ANALYSIS

II

- Model problem mathematically
- Design algorithm (method) to solve the problem
- Prove correctness
- Analyze complexity - is the algorithm efficient?

## COMPLEXITY MEASURES

- Running time  $\Leftarrow$  MAIN FOCUS IN THIS COURSE
- Memory usage
- Parallelism
- Energy consumption
- Et cetera

## COMPUTATIONAL MODEL

To do rigorous mathematical analysis of algorithms, need to give a precise definition of what we mean by "computer"

In this introductory course, think of

- standard laptop
- your favourite programming language (Python, Java, C, C++)

The actual mathematical model (which we won't talk about in this course) is very robust to variations of details

Let's model the problem and design an algorithm we can analyze!

Observation: Only BRIDGES in Königsberg makes

Model as

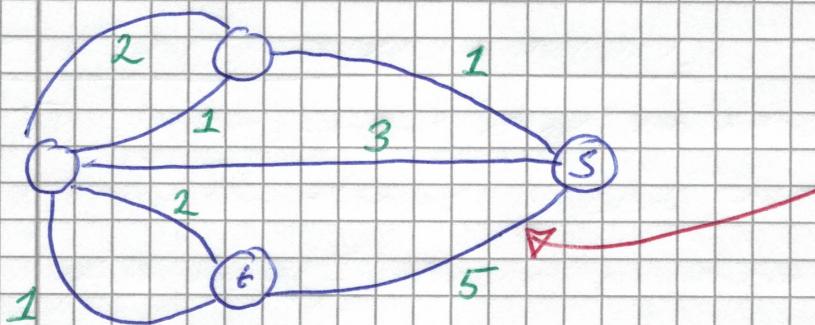
GRAPH  $G = (V, E)$

$V$ : vertices

$E$ : edges / arcs between pairs of vertices

Extension: Edges can have weights  
(what is the length of the bridge, say)

CYCLE: Walk along edges returning to start vertex



edge is INCIDENT to s and t

COMPUTATIONAL PROBLEMS given  $G = (V, E)$  as input

EULERIAN CYCLE

Is there a cycle using every edge once? (exactly)

HAMILTONIAN CYCLE

Is there a cycle visiting every vertex exactly once?

TRAVELING SALESMAN PROBLEM (TSP)

What is the shortest cycle visiting every vertex exactly once?

SHORTEST PATH PROBLEM

What is the shortest route from s to t?

## EULERIAN CYCLE ALGORITHM

OBSERVATION For every vertex, edges (bridges) are used in pairs incoming - outgoing

$\Rightarrow$  After finished cycle, must have even # edges incident to every vertex

Can prove this is also sufficient

Algorithm 1:

EULER ( $G$ )

for every vertex  $v \in V$

| if # edges incident to  $v$  odd

| | return "no"

return "yes"

This is pseudocode — not real program, but can be translated easily to your favourite programming language

## HAMILTONIAN CYCLE ALGORITHM

Simple idea: Try all possible orderings (permutations) of vertices in  $V$  and check if cycle

Algorithm 2:

HAMILTON ( $G$ )

for every ordering  $O = \{v_1, v_2, \dots, v_n\}$  of  $V$

| is-cycle := TRUE

| for  $i := 1$  up to  $n-1$

| | if  $(v_i, v_{i+1})$  not edge

| | | is-cycle := FALSE

| | if  $(v_n, v_1)$  not edge

| | | is-cycle := FALSE

| | if is-cycle return "yes"

return "no"

# TRAVELLING SALESMAN ALGORITHM

IV

Can use similar idea for TSP

Algorithm 3

TSP (G)

best-tour :=  $\infty$

for every ordering  $O = \{v_1, \dots, v_n\}$  of  $V$

curr-tour := 0

is-cycle := TRUE

for  $i := 1$  upto  $n - 1$

if  $(v_i, v_{i+1})$  not edge

is-cycle := FALSE

else

curr-tour := curr-tour + weight( $v_i, v_{i+1}$ )

if  $(v_n, v_1)$  not edge

is-cycle := FALSE

else

curr-tour := curr-tour + weight( $v_n, v_1$ )

if (is-cycle and curr-tour < best-tour)

best-tour := curr-tour

return best-tour

# SHORTEST PATH ALGORITHM

as  
Could use similar idea for TSP, but  
only start counting from vertex s and  
stop at vertex t.

COMPLEXITY ANALYSIS

Measure running time by counting # basic operations  
for instance:

- adding two numbers
- checking if edge  $(u, v)$  exists in graph  $G$

What is an efficient (i.e., fast) algorithm?

Depends on the size of the input

**KEY INSIGHT**

Measure how running time / # operations scales with input size

If input size doubles, then reasonable that running time doubles — we need to read twice as much input

If running time increases by at most constant factor  $K$  when input size doubles, then algorithm **EFFICIENT**

Equivalently: Algorithm is efficient if for input size  $n$ , the running time scales like some polynomial  $n^k$

$K = 2$  polynomial  $n$  — linear in  $n$

$K = 4$  polynomial  $n^2$  — quadratic

$K = 8$  polynomial  $n^3$  — cubic

Are our algorithms above efficient?

## Eulerian cycle

Just scan input — linear time

Often best we can hope for (in this course)

✓

## Hamiltonian cycle and TSP

For graph with  $|V| = n$  vertices,  
have to consider all

$n! = n(n-1)(n-2) \cdots 3 \cdot 2 \cdot 1$   
possible permutations

$$n! \approx 2^n \log n$$

Even worse than exponential time!

For just a thousand vertices, won't get  
results before the sun dies... X

Is it possible to find efficient algorithm  
for Hamiltonian cycle or TSP?

NOT KNOWN!

Many believe that there are no algorithms  
that will always be efficient and give correct answers

But proving this is one of the MILLENNIUM  
PRIZE PROBLEMS with a 1 MUSD award

What about SHORTEST PATH PROBLEM?

There is an efficient algorithm!

# DIJKSTRA'S SHORTEST PATH ALGORITHM

VIII

Computes distances from start vertex  $s$  to all other vertices

Algorithm 4

DIJKSTRA ( $G_1, s$ )

for all  $v \in V$

$d(v) := \infty$

$d(s) := 0$

  while exists unprocessed vertices

$v :=$  unprocessed vertex with smallest  $d(v)$

    for all neighbours  $u$  of  $v$

      update  $d(u)$  to  $d(v) + \text{weight}(v, u)$

      if smaller

    mark  $v$  as processed

## CLAIM 1

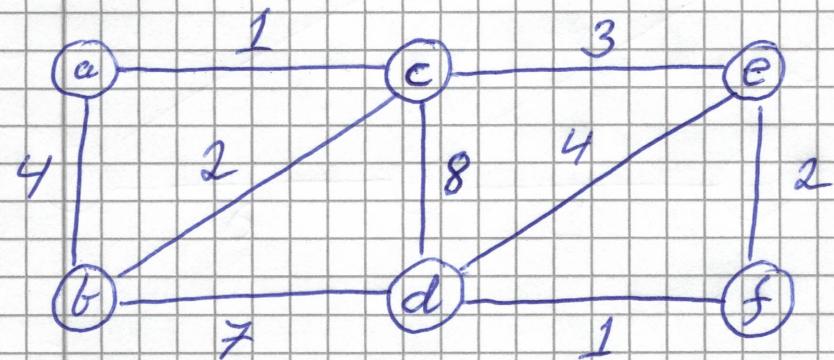
At the end of the algorithm, for all  $v$   
 $d(v)$  is length of shortest path from  $s$  to  $v$

## CLAIM 2

Dijkstra's algorithm can be implemented  
to run in nearly linear time

We will prove all this during the  
last week of the course...

For now, let's see an example day-in  
of the algorithm for graph with  
starting vertex a



DISTANCE ESTIMATE UPDATES

<u>Vertex processed</u>	a	c	b	e	f	d	SHORTEST PATHS FROM a
$d(a)$	0	0	wavy	wavy	wavy	wavy	0
$d(b)$	$\infty$	4	3	3	wavy	wavy	3
$d(c)$	$\infty$	1	1	wavy	wavy	wavy	1
$d(d)$	$\infty$	$\infty$	9	9	8	7	7
$d(e)$	$\infty$	$\infty$	4	4	4	wavy	4
$d(f)$	$\infty$	$\infty$	$\infty$	$\infty$	6	6	wavy

Neither correctness nor efficiency  
is obvious, but we will get back to this

# RAM MODEL

How we think of computer

Big chunk of MEMORY partitioned into WORDS

In one time unit, we can

- read word from memory
- store word in memory
- perform operation

word 1
word 2
word 3
word 4
word 5
:

## WORD ( $w$ )

A number of bits (0s & 1s)

We assume "basic objects" fit in a word

- character
- integer (not ridiculously large)
- pointer (to other location in memory)

What does NOT fit in a word:

- arrays / tables
- lists
- graphs

## OPERATIONS

Assume following operations take constant time

- arithmetic +, -, \*, /
- Boolean operations and, or, not
- Bitwise operations
  - & (bitwise and)
  - | (bitwise or)
  - ~ (bitwise not)

This is a **MODEL**, not a description of reality  
But good enough for our purposes

## PSEUDO-CODE

Not actual code

But "close enough" that you can easily code it up in your favourite programming language

- Focus on essential details
- Ignore irrelevant syntax details
- Assume flat "obvious"/"reasonable" methods exist in some (pseudo-) standard library

What is important?

- NOT the exact syntax
- But that we get a crisp, clear, unambiguous description of the algorithm

## COMPLEXITY ANALYSIS

Given algorithm A solving same task

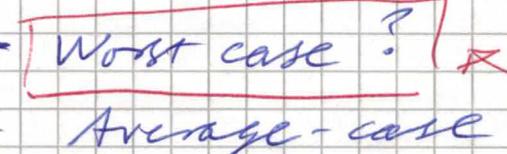
Given input of size  $n$

How many steps does algorithm need for input of size  $n$ ?

Count:

- Reading, assigning, storing
- arithmetic operations
- comparisons
- program flow constructs
  - if - then - else
  - for loops
- method / function calls

What time complexity?

- Best case ?
- = Worst case ? 
- Average-case
- "Real-case"

Focus on worst case (in this course)

Often most relevant for the basic algorithms we will study

But certainly not the only way to measure!

### Simplifying assumptions

All operations take the same time

Only care about highest-order term

### Example

Algorithm A running time  $n^2 - 4n + 3$

Algorithm B  $- 11 - 1000n + 1000000$

A scales like  $\sim n^2$

B scales like  $\sim n$

So B will become (much) better as input size grows

In reality, factor 1000 is of course important

But this is an introductory course - focus on the fundamentals. And linear-time algorithm will quickly beat quadratic-time algorithm in practice