Einführung in die Informatik

Computational Thinking

Prof. O. Nierstrasz
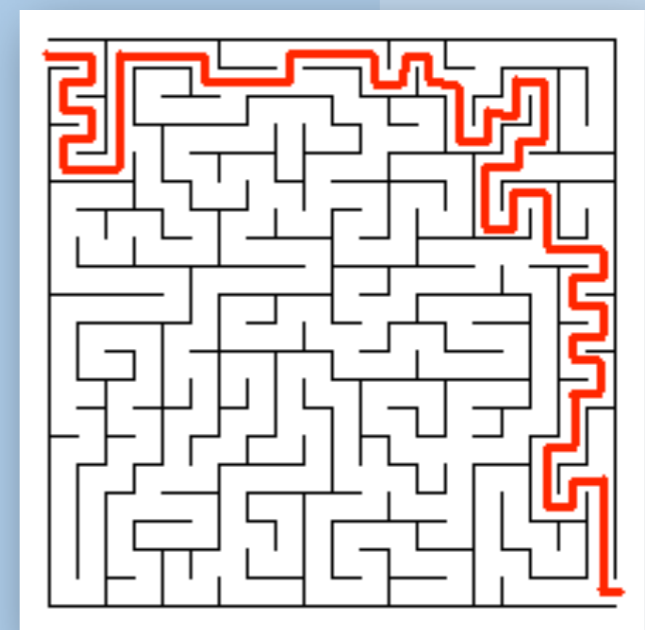
Schedule 2021

23.09.2021	**Computational thinking**	Oscar Nierstrasz
30.09.2021	**Programming languages**	Oscar Nierstrasz
07.10.2021	**Endliche Automaten**	Thomas Studer
14.10.2021	**Berechenbarkeit und Komplexität**	Lukas Zenger
21.10.2021	**Datenbanken**	Thomas Studer
28.10.2021	**Digitale Nachhaltigkeit**	Matthias Stürmer
4.11.2021	**Data handling and visualization**	Athina Tzovara
11.11.2021	**Randomisierte Algorithmen**	Christian Cachin
18.11.2021	**Kryptographie**	Christian Cachin
25.11.2021	**Discrete Representations**	Paolo Favaro
2.12.2021	**Discrete Optimization Methods**	Paolo Favaro
09.12.2021	**Modellierung und Simulation 1**	David Bommes
16.12.2021	**Modellierung und Simulation 2**	David Bommes
23.12.2021	**Q&A**	

Course Material

> ## ILIAS
> —Schedule
> —Slides
> —Videos
> —Zoom links
> —Exercises

This is a note (a hidden slide). You will find some of these scattered around the PDF versions of the slides.

Roadmap

> **Computational Thinking**
> The Thinking Tools
> Tool Zoom In: Recursion
> Tool Zoom In: Modeling
> Outlook

Computational Thinking
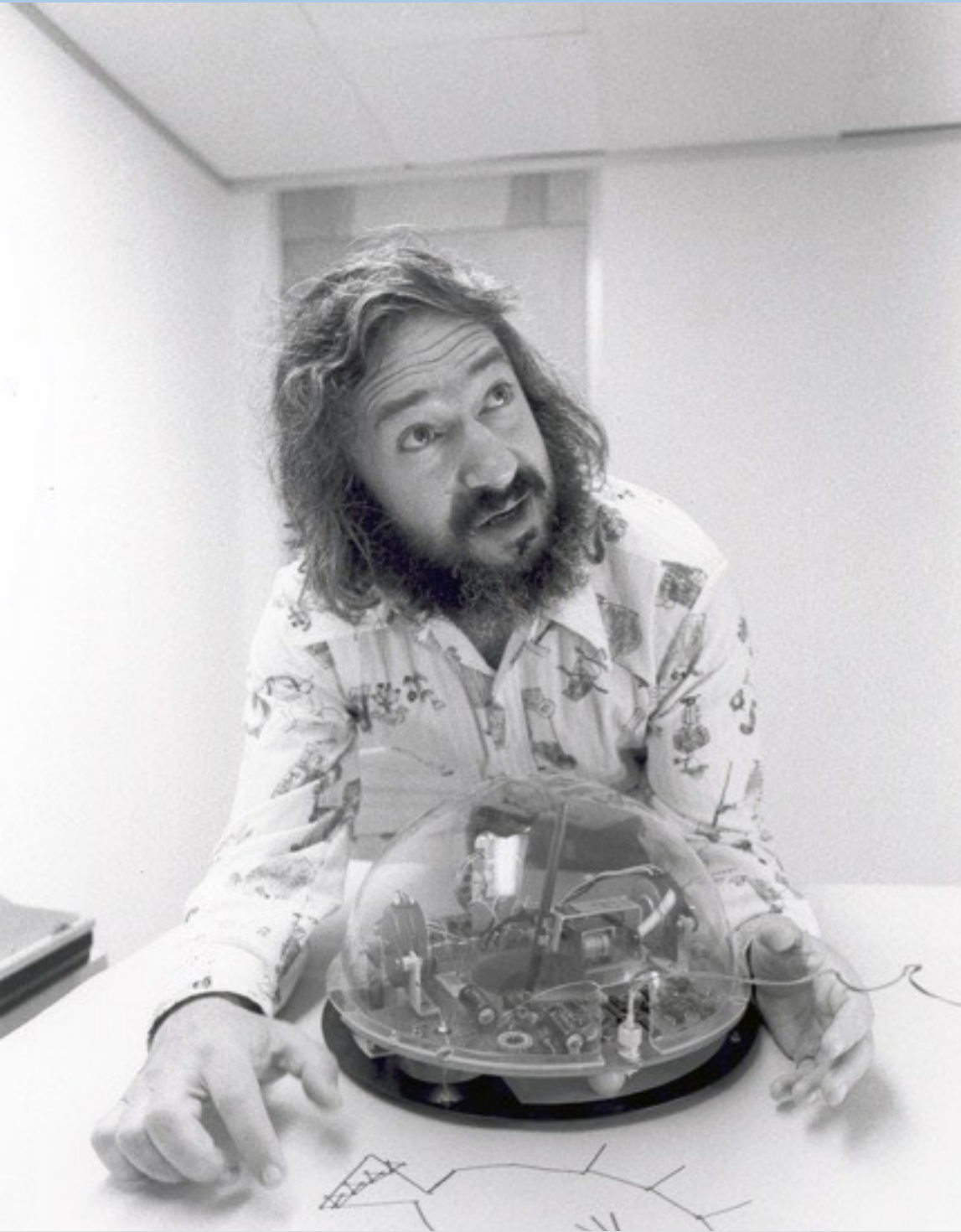
Seymour Papert

"... is a way of <u>solving problems</u>, <u>designing systems</u>, and <u>understanding human behavior</u> that draws on concepts fundamental to computer science"

— Jeannette M. Wing, CACM 2006

In German: "*Rechenbetontes Denken*"

The term was coined by Seymour Papert in his paper from 1996 and popularized by Jeannette Wing. Consider it to mean "thinking like a computer scientist".

http://www.papert.org/articles/AnExplorationintheSpaceofMathematicsEducations.html

Papert was a student of Piaget, the Swiss developmental psychologist. Papert was a proponent of educating children with the help of computing and argued for an experience-based learning. He argued that the tools of the computer scientist are useful in general for the developing child.

The article of Wing is a slalom through the many tools of the computer scientist, of which this lecture presents several.

http://cacm.acm.org/magazines/2006/3/5977-computational-thinking/fulltext

Übersicht

Informatikstudium

Andere Studiengänge

Schnittstellen zur Aussenwelt
(Mensch-Maschine Schnittstelle, Computer-vision, Computergrafik, Sensornetze, Künstliche Intelligenz, Computerlinguistik)

Mathematik

Wirtschaftsinformatik

Wissenschaftliche Anwendungen
(Modellierung und Simulation, Biologie, Physik, Chemie, Sozialwissenschaften, etc.)

Informatik

Praxis
(Programmiersprachen, Betriebssysteme, Netzwerke &Verteilte Systeme, Software Engineering, Datenbanken, Rechnerarchitektur)

Theorie
(Automaten und formale Sprachen, Berechenbarkeit, Komplexität, Logik, Algorithmen)

Anwendungs-software

Informatik = Computer Science

Computer Science requires many different kinds of skills: abstraction, communication, reading, mathemathics, modeling, planning …

If you are a practitioner, you also have to have strong communication skills. A software engineer as an architect has to talk to customers, explain them what is possible and what is not, to convince them of her solution.

Software Engineering

Software Engineering consists of
- processes and *techniques*
- to develop software *products*
- within a given *budget* and *deadline* and
- satisfying *functional* and *quality requirements*

Engineering is about best practices in building *physical* products.

Software engineering is about building software products (i.e., virtual products).

Many of the concerns are the same (planning, quality control, process control), but many are very different due to the non-physical nature of software (production, manufacturing, constraints).

How Software Engineering bridges domains

quality assurance and testing

capture and model requirements ⟷ *architecture, design, implementation*

Real-world domain   **Technology domain**

A large part of the job of software engineering is *gathering and understanding the requirements*. It is not purely a technical issue.

Another large part has to do with *managing the development process*. (Teamwork, planning, budgeting, politics etc etc)

Many different kinds of models are needed to bridge these worlds.

Roadmap

> Computational Thinking
> **The Thinking Tools**
> Tool Zoom In: Recursion
> Tool Zoom In: Modeling
> Outlook

Abstraction

Abstractions strip away details to help us cope with complexity

Abstractions allow us to deal with complexity by *stripping away needless detail*.

They also provide us with some nice *properties*, for example, queues ensure *fairness*.

Any organization that we deal with (like a bank) provides us with a simple *interface* and a much more complex *implementation*.

Abstractions only really exist in our heads. A queue, or even a bank, does not exist in reality, only by convention. What exists is only a physical manifestation (a bunch of people in a line, a building) but that is not the abstraction.

All software projects make heavy use of abstraction to cope with complexity.

Decomposition & Separation of concerns

We decompose complex tasks by separating concerns

Printing a book entails a large number of complex activities. By breaking them down and specializing skills, the entire process can be handled efficiently.

Contrast this with the laborious and inefficient way books were copied before the invention of the printing press.

Software systems and software development projects exploit decomposition and separation of concerns at all levels.

Parallel algorithms

By distributing tasks to many workers, we get more done in the same amount of time

We see this in any organization. We can add more factory workers to increase production.

This works only if the tasks are truly independent. Adding more members to a team may actually slow down work!

Computer servers work this way – a farm of servers can be scaled up to handle more queries.

Redundancy

Replicating critical elements of a system increases its resilience

A restaurant employs many waiters, each of which can take over duties of the others.

Redundancy in computing is the replication of critical elements of a system to increase the reliability of the system. This is the case in distributed systems, and in hardware.

When information is transmitted along the line, often an extra parity bit is appended to the actual data for verification purposes.

RAID Disk drives are arrays of hard disks which use redundant units to ensure continuing function even in the presence of individual disk failures.

Prefetching and cacheing

Prefetching = get what you need before you need it

Cacheing = keep handy whatever you will need again

Many problems can be solved more effectively by preparing in advance materials that you need: preparing luggage for travel (instead of buying stuff when you arrive); buying ingredients for a meal; preparing for a meeting.

To-do lists are useful for correctly selecting the items to prefetch, but *simulation* is an even better tool. (Step through the actions you will perform, and check if you have everything you will need.)

Prefetching and cacheing are used especially in modern microprocessor architectures to speed up execution.

Planning and optimization

Planning may require sophisticated analysis of multiple scenarios
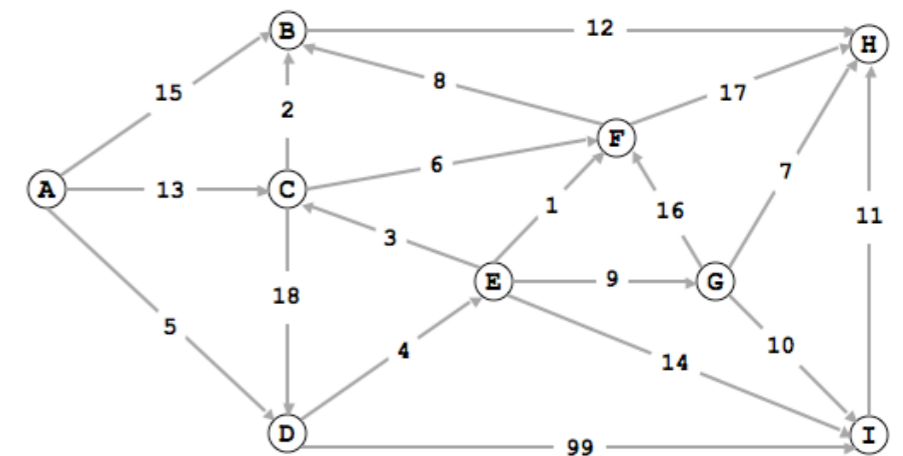
Which queue should you chose at the supermarket? You can consider: the length of the queue, how many items per customer, how awake the clerk looks …

Traffic is blocked in town – what is the fastest route home?

Queuing theory is used in computers to analyse potential performance bottlenecks.

Pipelining

Pipelining makes efficient use of expensive resources

Pipelining is used in factories, administrations, kitchens — anywhere that tasks can be specialized to workers with special skills.

Pipelining in computers is used in hardware and software, especially when data needs to be processed in several different ways.

Problems can occur with pipelining if some stages are faster or slower than others, causing work to pile up, or resources to go idle.

Concurrency control

Contention for shared resources can lead to both *safety* problems and *liveness* problems.
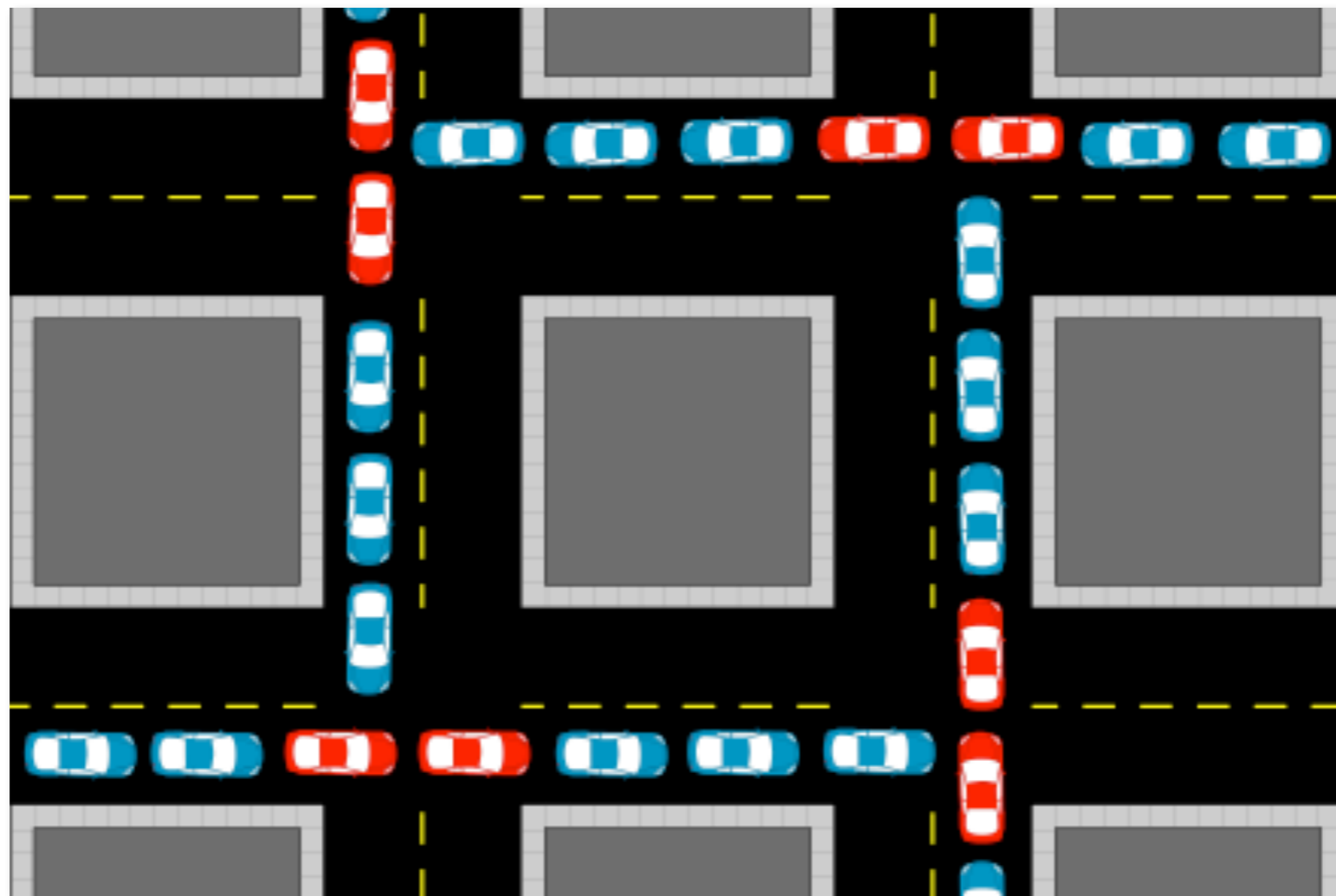
Concurrency problems can basically be divided into *safety problems* (nothing bad should happen) and *liveness* (something good should happen. Traffic accidents are bad. Progress is good. (Traffic jams are safe but not live.)

Traffic lights and roundabouts, together with proper rules and scheduling, can yield good solutions to managing concurrent traffic. Bad designs can create traffic jams or even accidents.

Safety and liveness problems arise because of need for access to shared resources (roads, intersections …).

Software solutions include scheduling algorithms (in the OS or VM) and programming techniques (similar to traffic lights).

In some cases, applying the techniques of concurrent program analysis to real world hospital protocols has detected possible "deadlocks".

Simulation
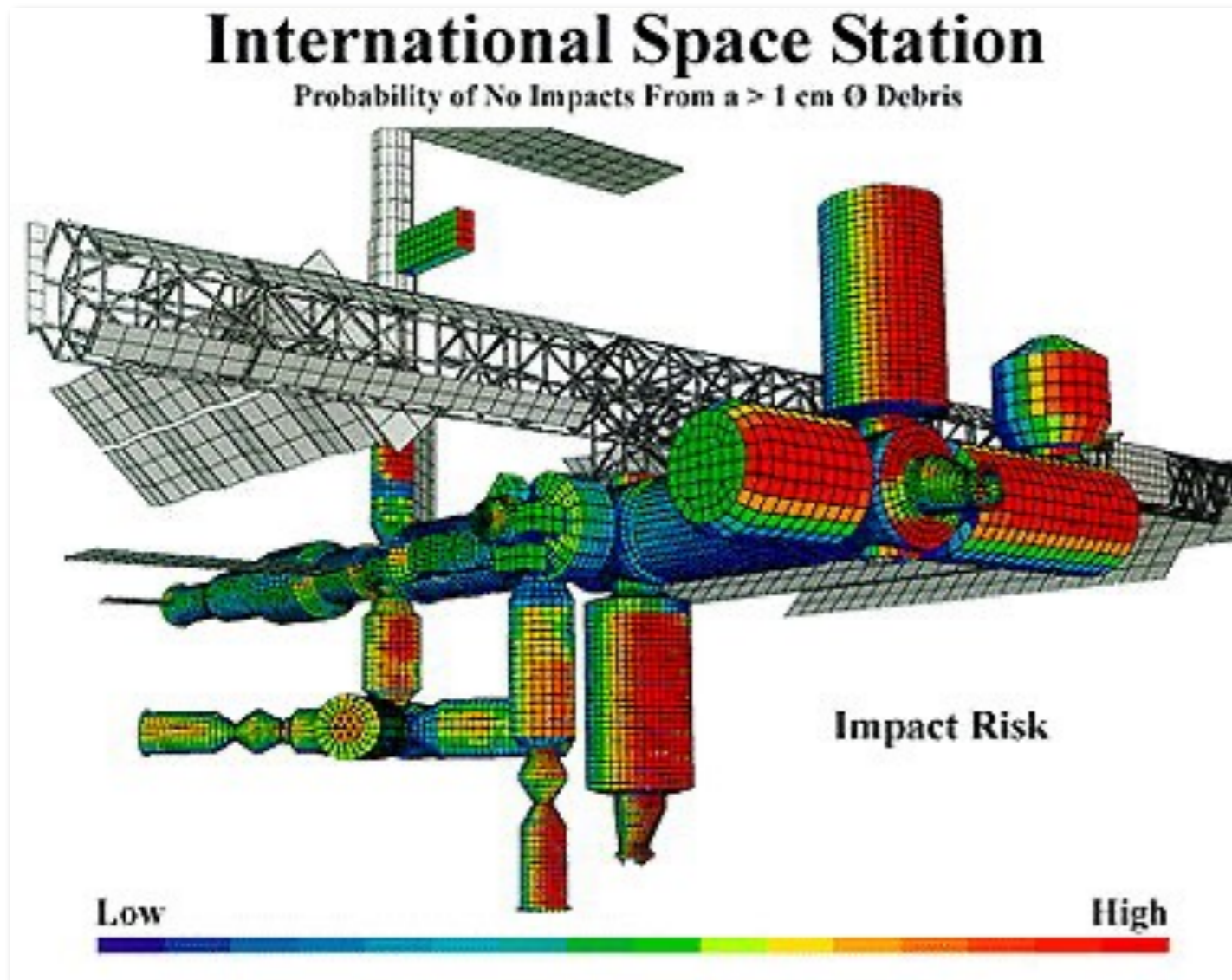
Expensive experiments can be replaced by inexpensive simulations

Simulation is a technique that abstracts from some real thing by *capturing the interesting properties so we can try them out.*

Simulation is a form of modeling that usually focuses on some physical properties that are *hard* or *expensive* to test in real life.

We often simulate with paper or blackboard models, but nowadays we often use computers to simulate physical properties.
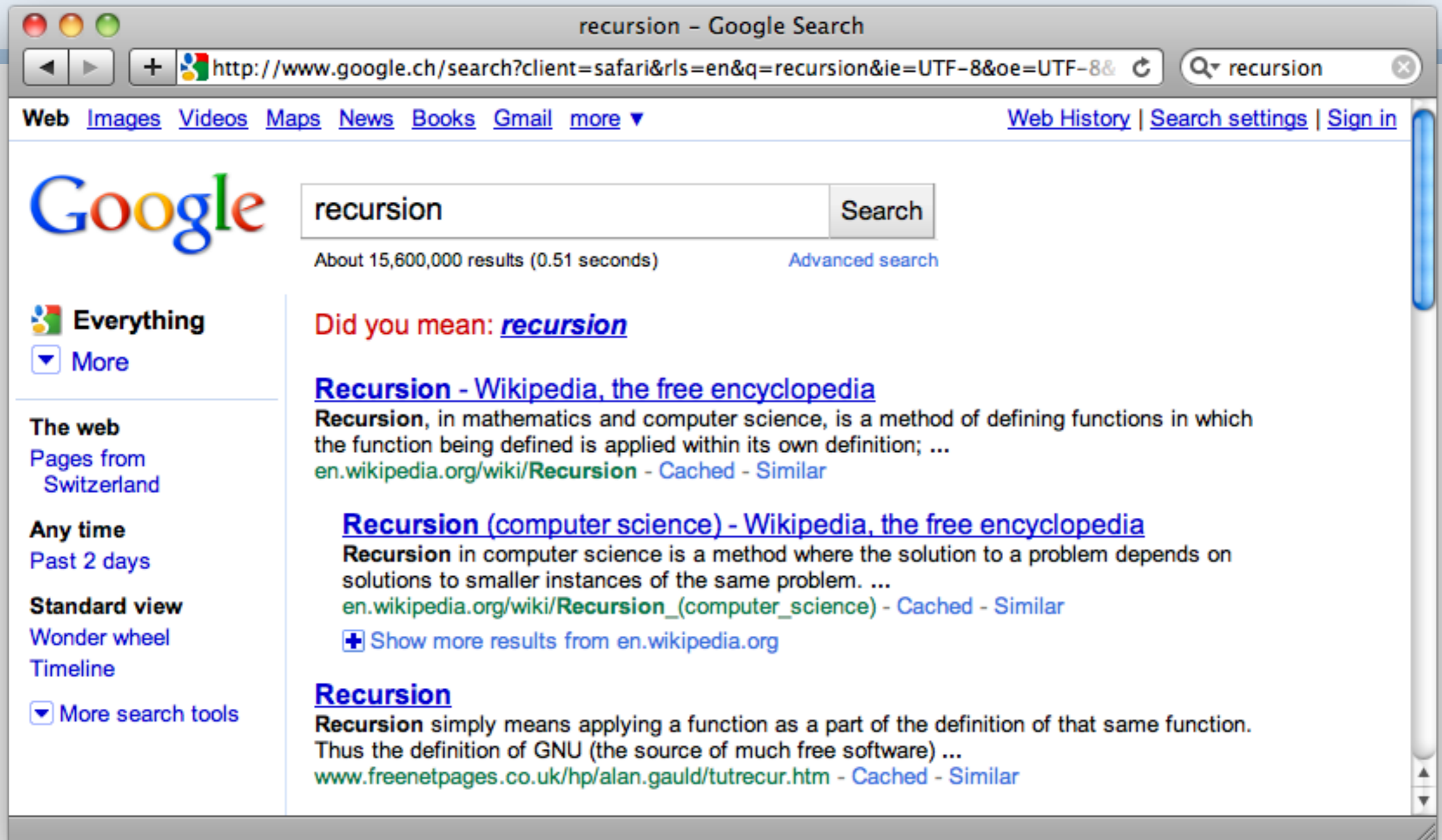
The figure shows a model of the International Space Station (ISS) and the regions which are most at risk of impact with an asteroid.

Roadmap

> Computational Thinking
> The Thinking Tools
> **Tool Zoom In: Recursion**
> Tool Zoom In: Modeling
> Outlook

Recursion

Recursion is when one operation is defined in terms of itself. For example:

Phrase = Phrase + conjunction + sentence.

Infinite recursion is to be avoided in computer science.

Avoiding infinite recursion

re•cur•sion |riˈkər ZH ən|
noun Mathematics & Linguistics
If you still don't get it, see RECURSION.

To prove termination, you must show that recursion will reach a *base case*.

Unless you are very slow, eventually you will "get it", and break out of the recursion.

Phrase = Phrase + conjunction + sentence.

Phrase = Sentence

Proper recursion in Computer Science requires there to be a *base case* that is not recursive. Furthermore, you must ensure that the recursive cases always reach a base case.

Example: factorial

$$5! = 5*4*3*2*1$$

fact(n) = n × fact(n-1)
fact(0) = 1

fact(5) = 5 × fact(4)
= 5 × 4 × fact(3)
= 5 × 4 × 3 × fact(2)
= 5 × 4 × 3 × 2 × fact(1)
= 5 × 4 × 3 × 2 × 1 × fact(0)
= 120

Factorial terminates assuming n is non-negative: since every recursive step reduces the argument by 1, eventually we reach the base case (n=0).

Caveat: For negative n, this does not hold.

Example: greatest common divisor

gcd(a,0) = a
gcd(a,b) = gcd(b, a mod b)

gcd(63, 91) = gcd(91, 63)
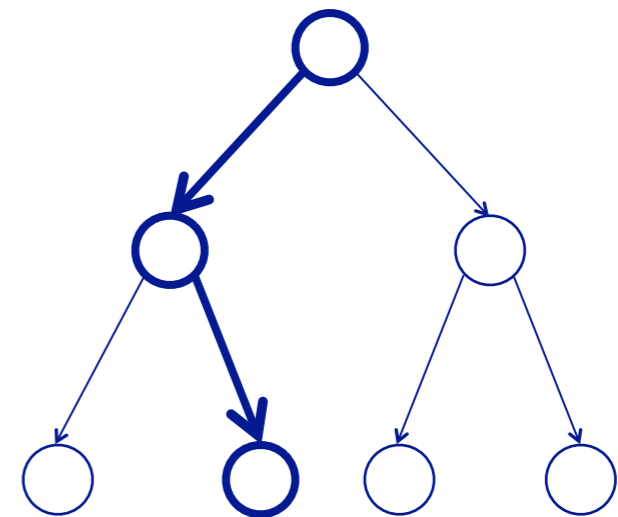= gcd(63, 28)
= gcd(28, 7)
= gcd(7, 0)
= 7

How do we know that gcd(a,b) always terminates for positive integers a and b?

GCD = Größter gemeinsamer Teiler.

This is Euclid's algorithm: Since *a mod b* is always less than *a*, the arguments get smaller in each step and must eventually reach the base case.

Binary search

Many navigation problems can be effectively solved using recursion.

Recursion is used in many human tasks: searching for a word in a dictionary can be solved recursively using binary search.

Passing down orders (or budget cuts) in a large organization can be done recursively (and in parallel).

There is always a danger with non-termination. Think of Ponzi schemes, which collapse due to the impossibility of endless recursion:

- https://en.wikipedia.org/wiki/Ponzi_scheme
- https://en.wikipedia.org/wiki/Bernard_Madoff

Recursion is heavily used in computer programs, especially to navigate tree-like structures, or to search for an answer in a tree-like search space.

Depth-first-search with backtracking
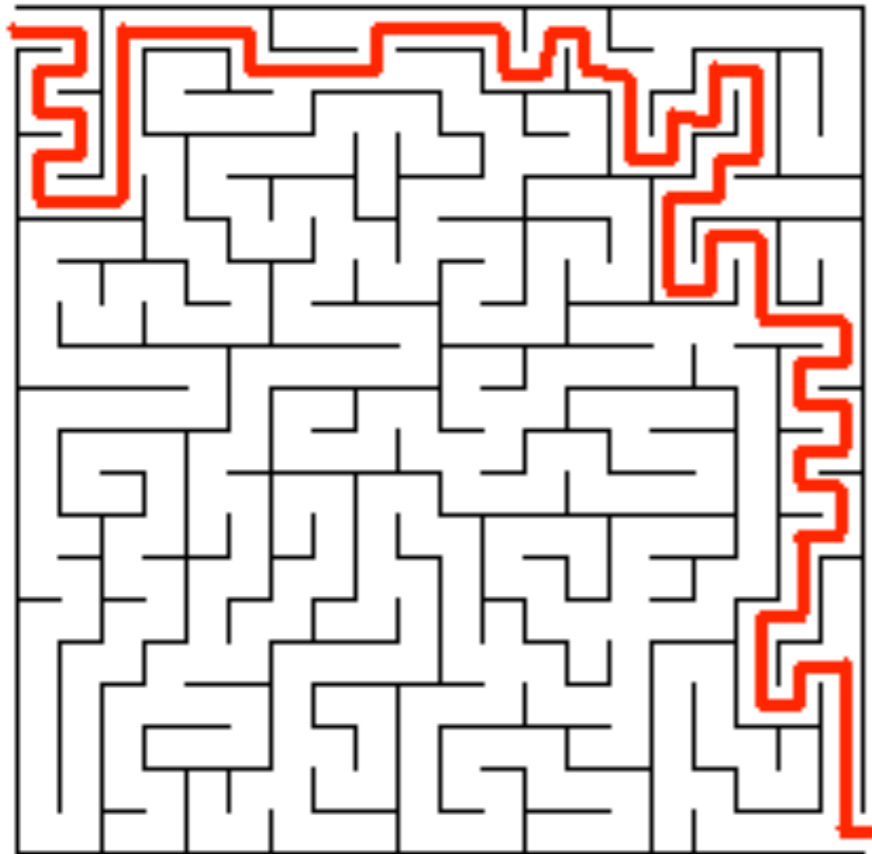
Many search problems can be solved with the help of backtracking

Backtracking requires you to keep track of every decision you make in a search space: searching through a maze; looking for something you lost (keys, glasses); trying out winning strategies.

If you discover you made an error, you backtrack to the last decision you make and change it.
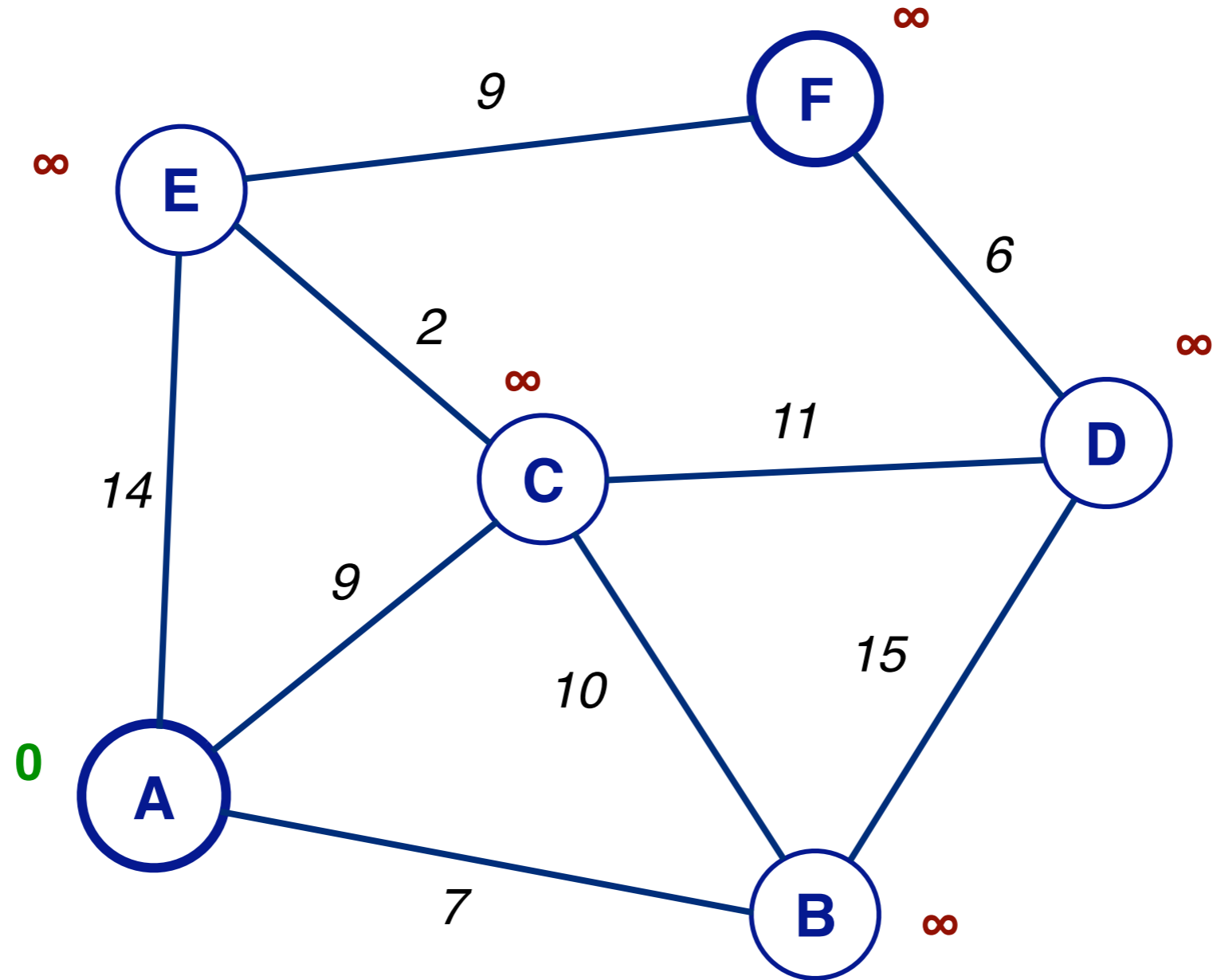
Essentially you are *searching depth-first through a tree of possibilities*.

A Stack is a useful data structure for keeping track of progress: the last decision is always on top of the stack.

(The stack is one of the most basic programming abstractions.)

Depth-first search = *Tiefensuche*

Finding the shortest path from A to F

Edsgar Dijkstra invented this famous algorithm to find the shortest path between two nodes in a network. The nodes of the graph represent locations and the edges are labeled with the distance (or "cost") to travel between them. The task is to find the shortest distance.

We start by labeling the start node (A) with 0, representing the total distance to travel from A, and we label all other nodes with ∞.

The algorithm proceeds *breadth-first* (rather than depth-first), incrementally updating each node with the shortest distance found thus far.

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Breadth-first search = *Breitensuche*

Finding the shortest path from A to F
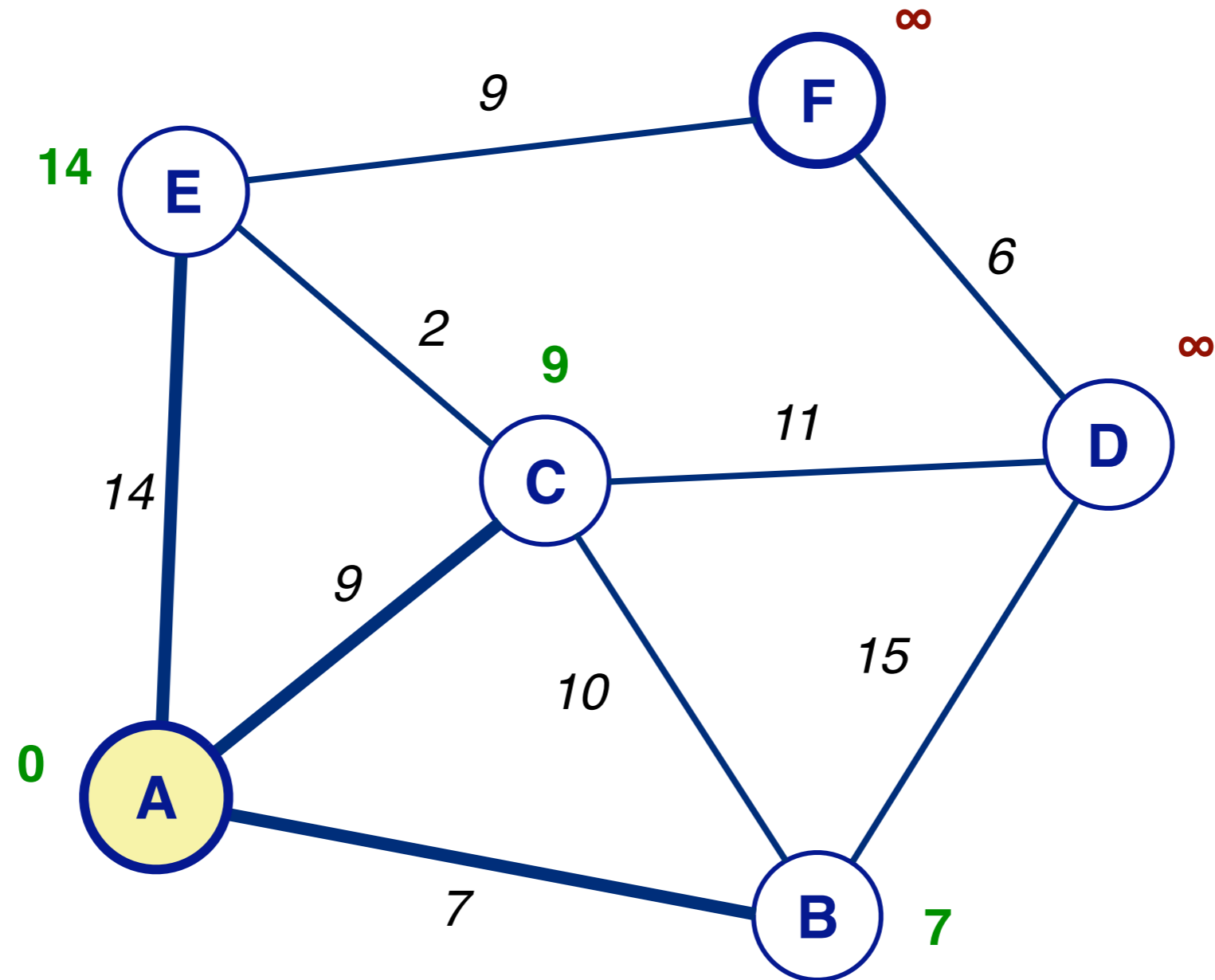
In the first step, we travel from A to all nodes not yet visited along a single edge. We reach nodes B, C and E, and we update the distances to the minimum of the current value and the distance traveled from A. (They are all less than ∞, so we update all of them.)

We mark A as *visited* (yellow) so we do not visit it again.

# Finding the shortest path from A to F
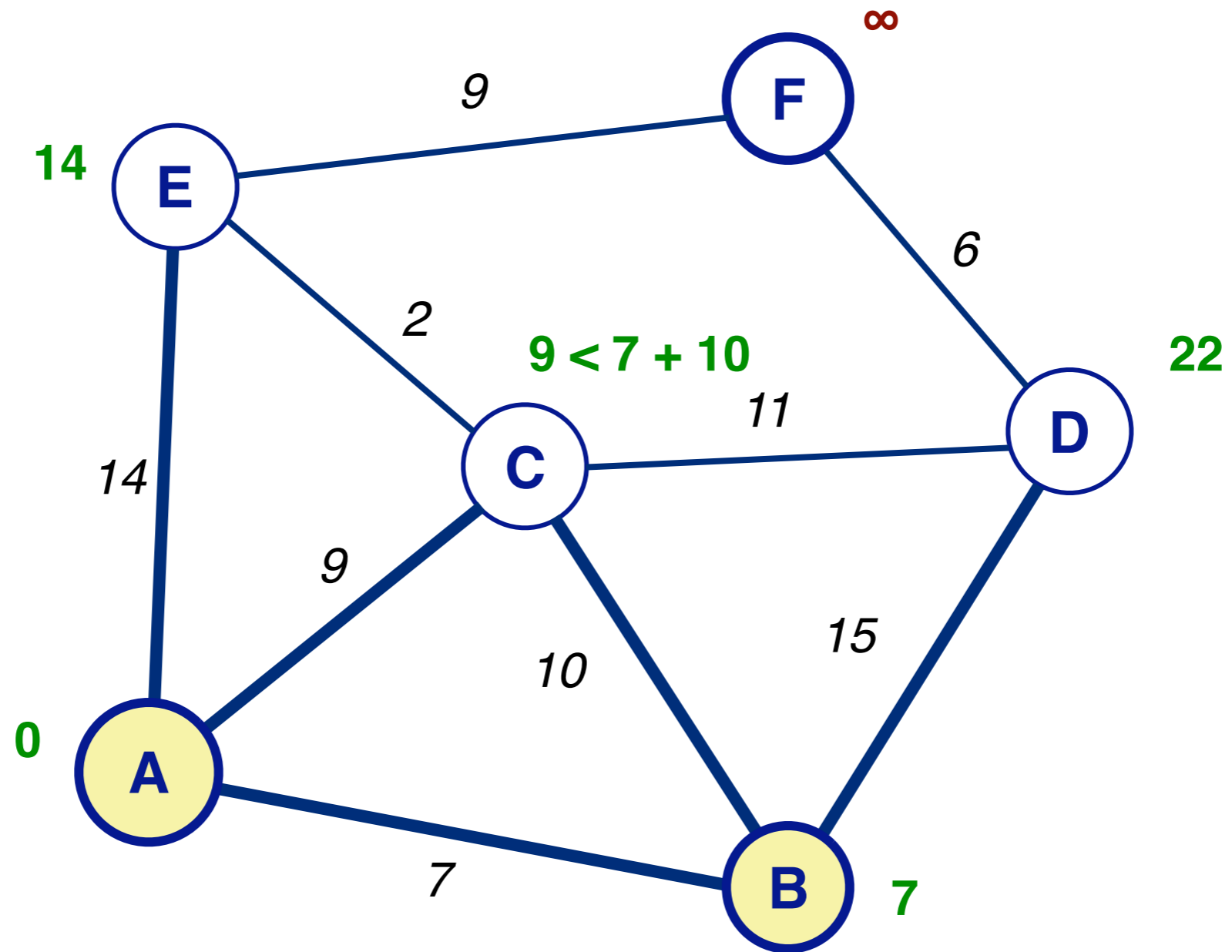
We have already started from A, so we need not consider it again. Now we arbitrarily pick a node we have reached (B), and travel from there to C and D. (No need to go back to A; we have already been there.)

The distance to C via B is 7+10, or 17, which is bigger than the current distance from A to C, so we do not update it. Hoever the distance to D (7+15=22) is an improvement, so we update it.

We mark B as *visited*.
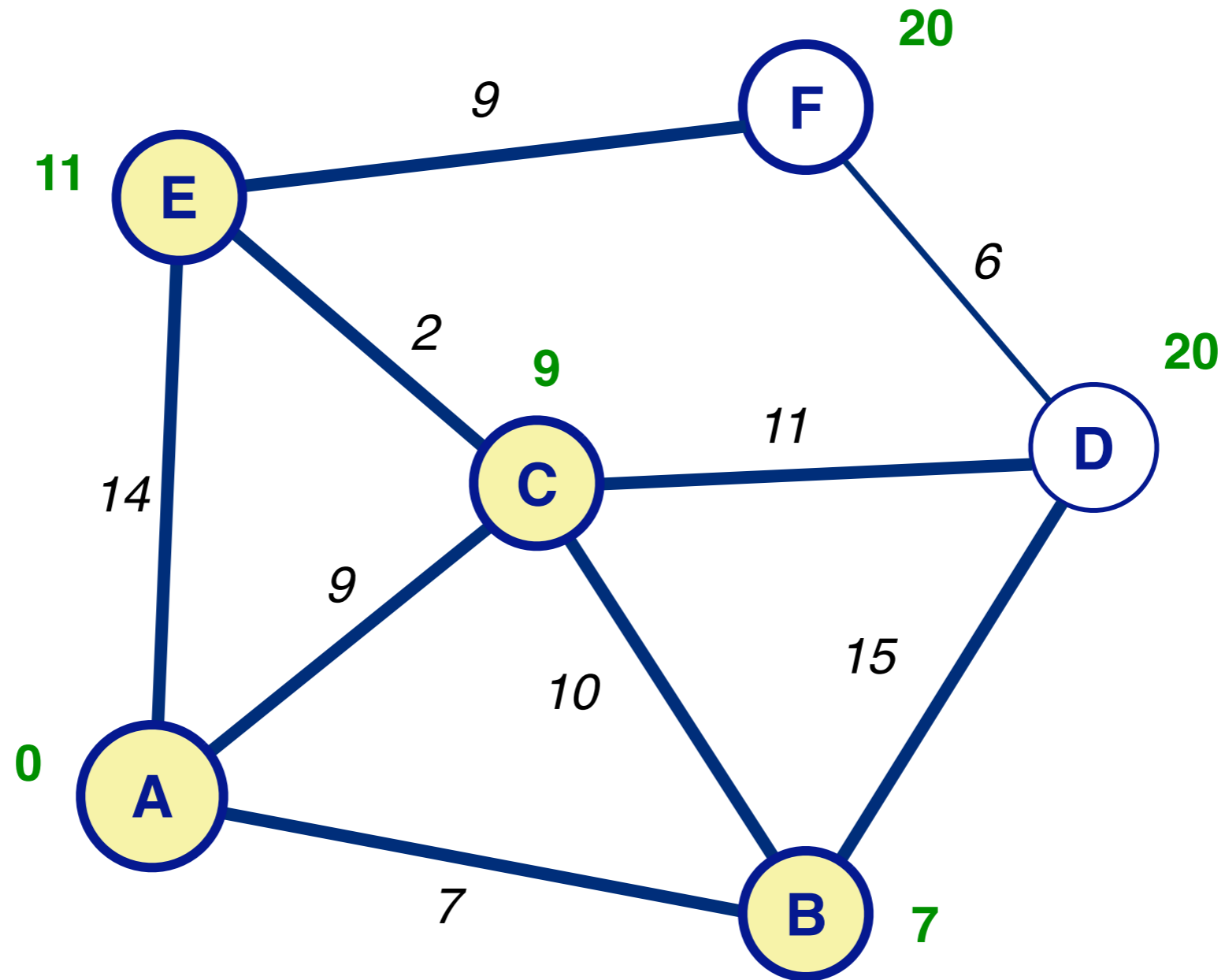
Finding the shortest path from A to F
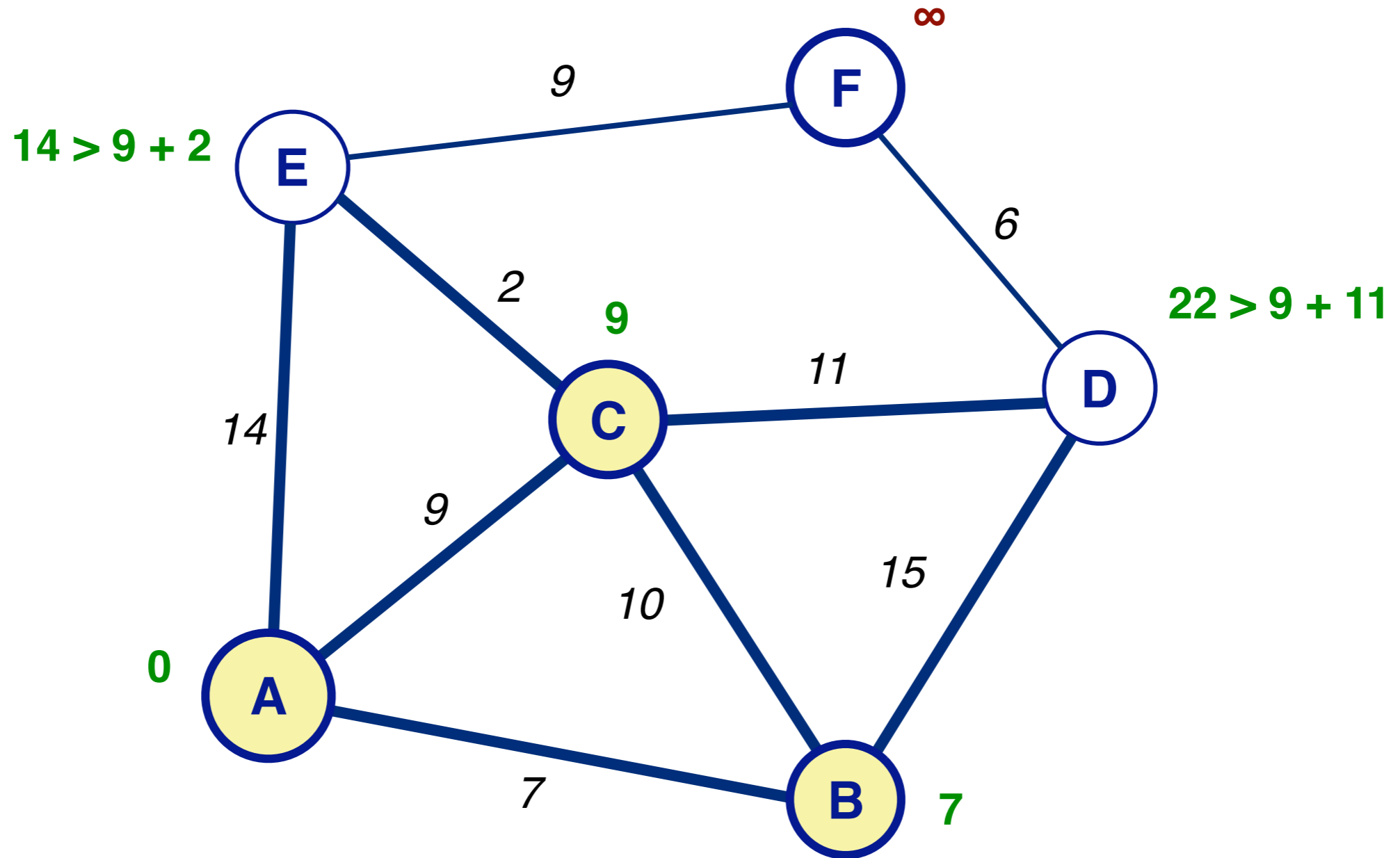
Again we pick an arbitrary node we have reached to continue traveling. (We pick C, but could also continue from D or E.)

From C we reach E and D. Both paths through C are shorter so we update the total distances to E and D.
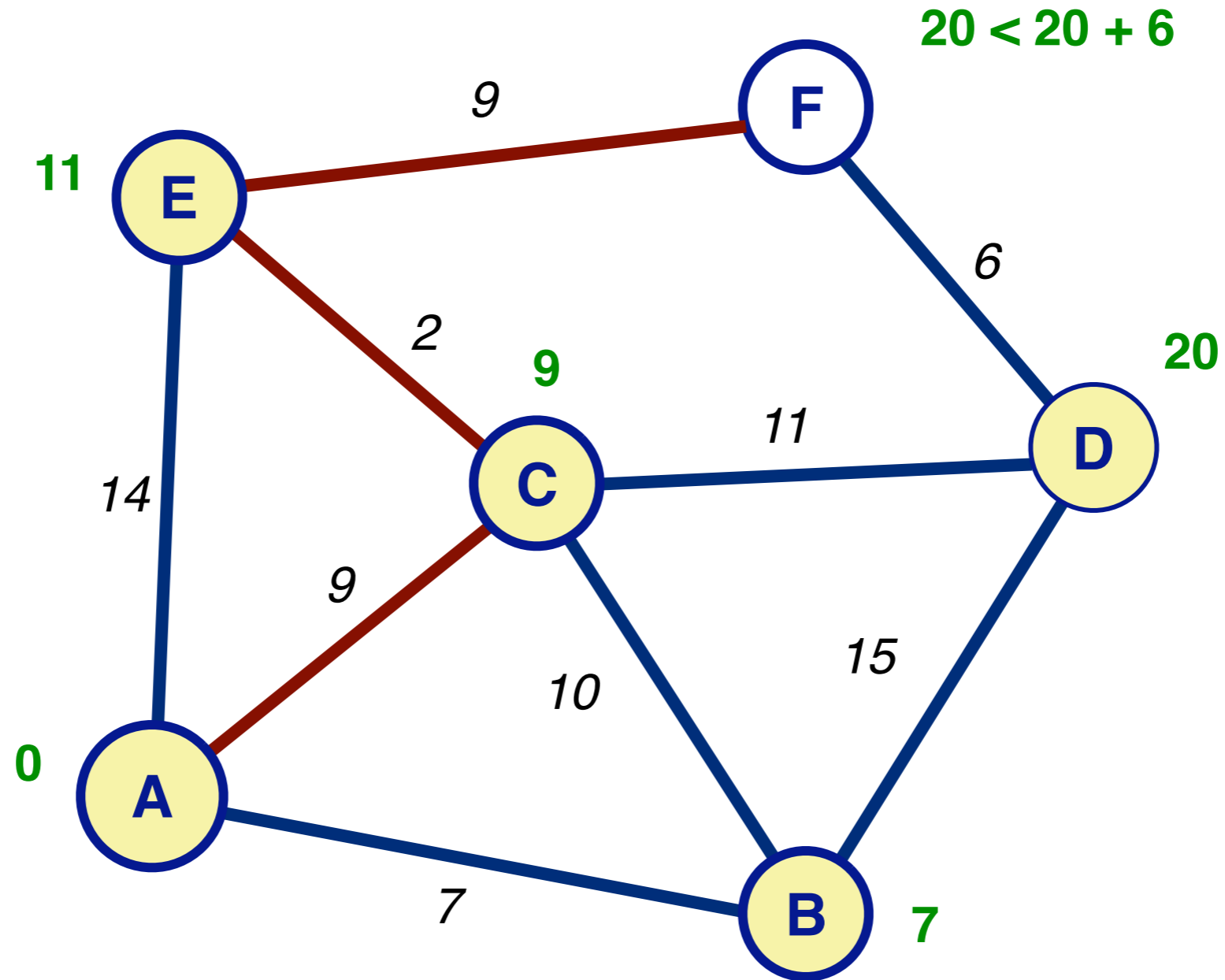
We mark C as *visited*.

Finding the shortest path from A to F

This time we start from E and discover that we can reach F in distance 20 via path ACEF.

We mark E as *visited*.

Finding the shortest path from A to F

Finally we continue from D and find that the path ACDF is longer (26). All nodes have been visited so we conclude the shortest path is ACEF (20).

Breadth-first vs depth first

Breadth-first

> Exhaustive — finds *all* solutions

> Must keep track of all nodes in a level

> Will find shortest path first

Depth-first

> Finds *first* solution

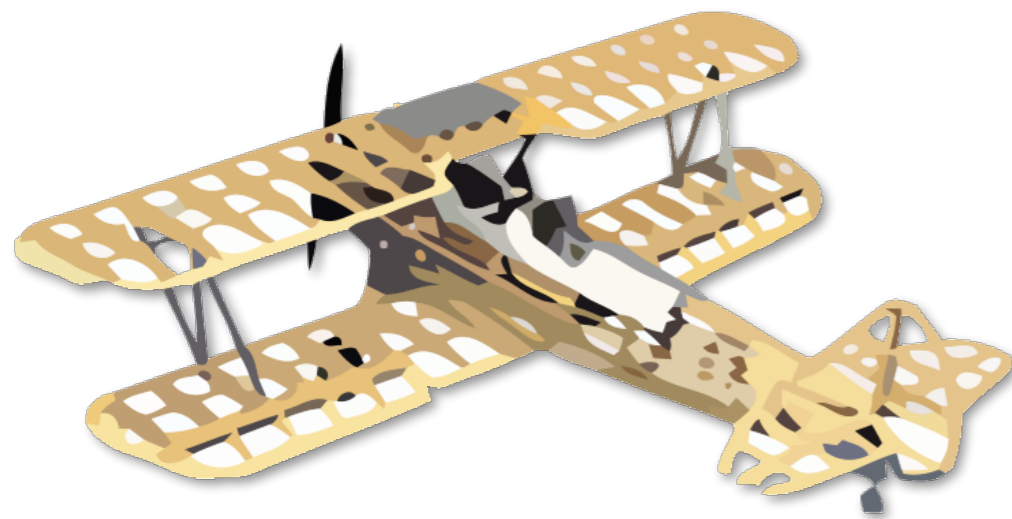> Only keep track of current path — can use a stack

> May be unlucky in searching

Roadmap

Modeling

A model captures certain properties of interest of a subject

A model *abstracts* certain *properties* of a subject in order to specify it, to communicate it, or to reason about it.

As with simulations, it is easier, cheaper or only feasible to work with the model than with the real thing.

- A *specification* may serve to build the thing that is modeled.
- A model as a form of *communication* avoids the cost of sending the thing itself.
- As a *reasoning tool*, a model lets us reason about size, weight, strength, speed, cost etc.

We may need multiple models of the same subject.

Software systems are all about modeling (what we store in computers are models; *computation* consists in manipulating models or reasoning about them).

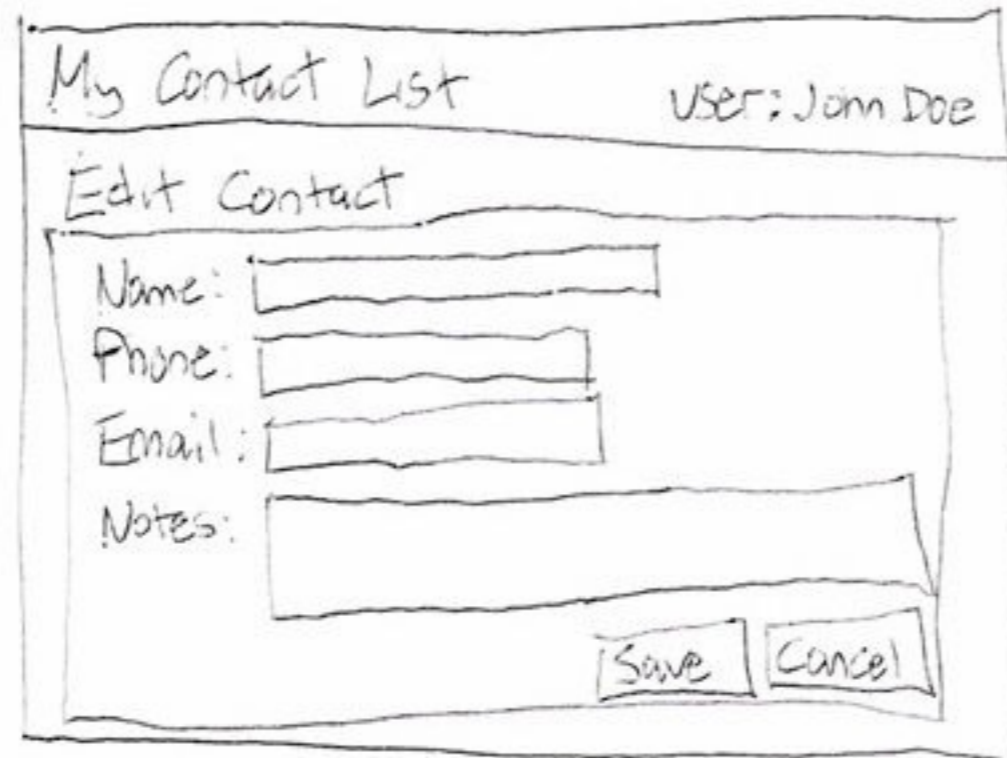# Computer graphics models capture visual aspects of real (or imagined) objects
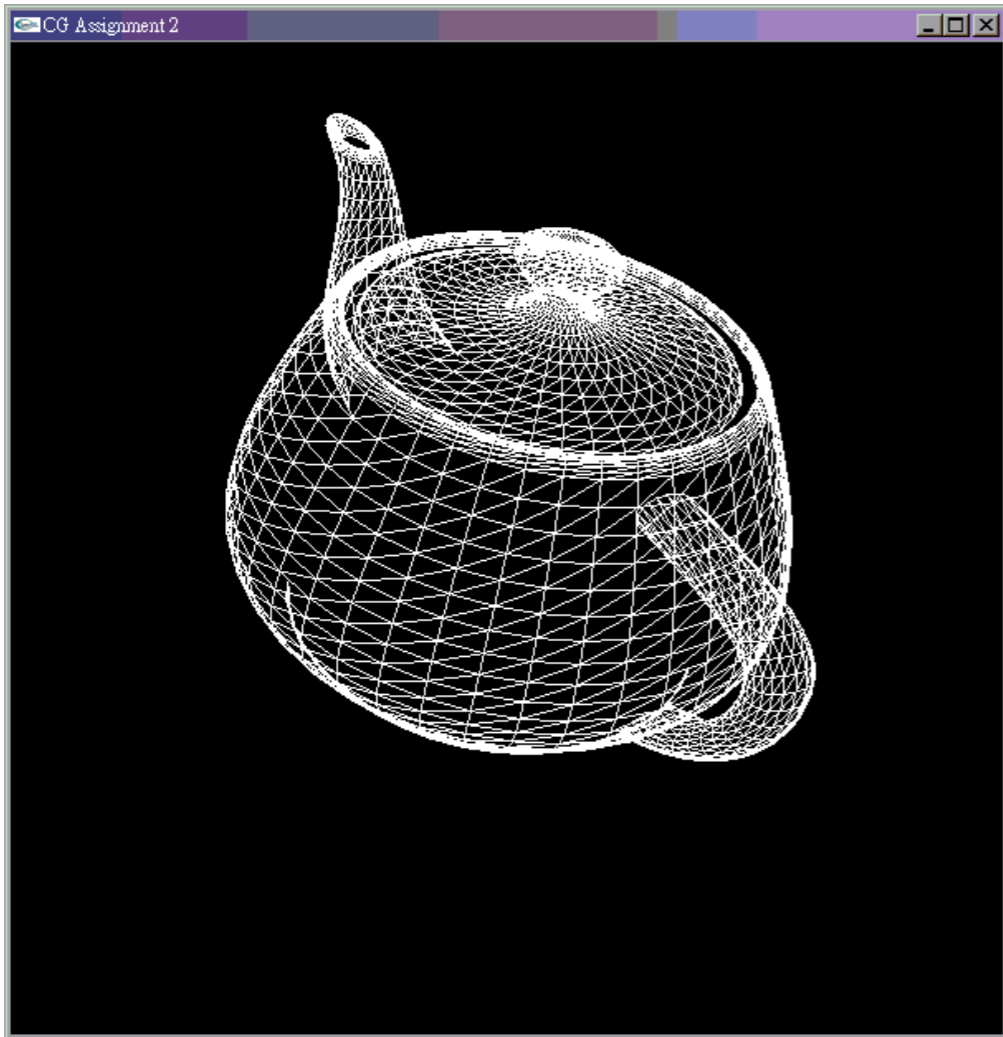
Different models express different properties

The teapot is the archetypical example of a computer graphics model.

Wire frame models capture geometry. Other aspects model light, perspective, texture, ambient characteristics, and so on.

Prototyping

A *prototype* is a primitive model of a system that allows it to be studied and evaluated.

In Computer Science, prototypes may be executable or not. *Paper prototypes* of user interfaces are often used to test out scenarios before designing or implementing the actual system.

http://www.infoq.com/articles/agile-useability-churchville

Data modeling

A data model captures the *persistent* entities of an information system, their attributes and relationships
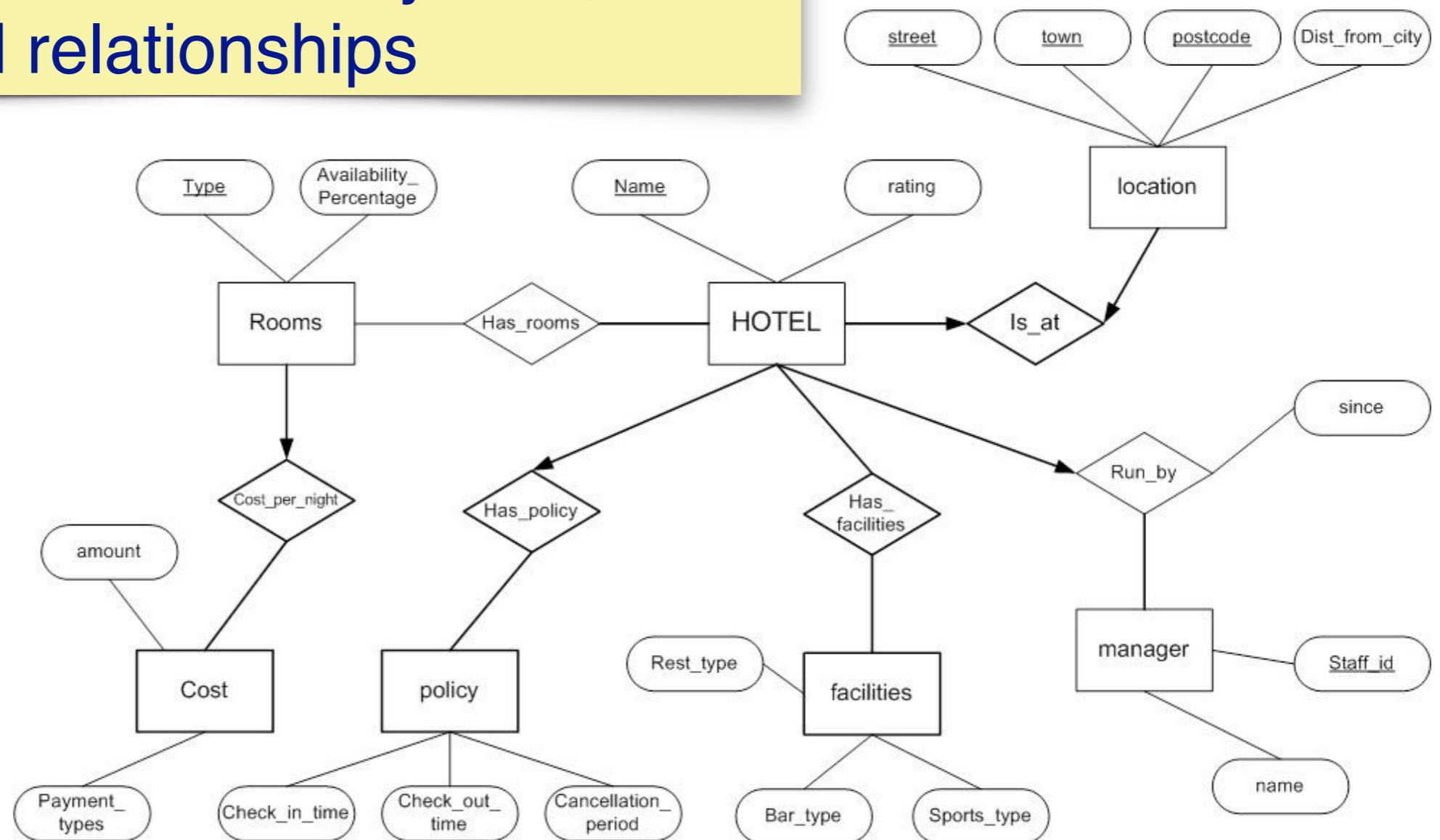
Here we are modeling only the persistent entities that the application must maintain over time. Typically they will be stored in a *relational database*.

Software models express architecture and design
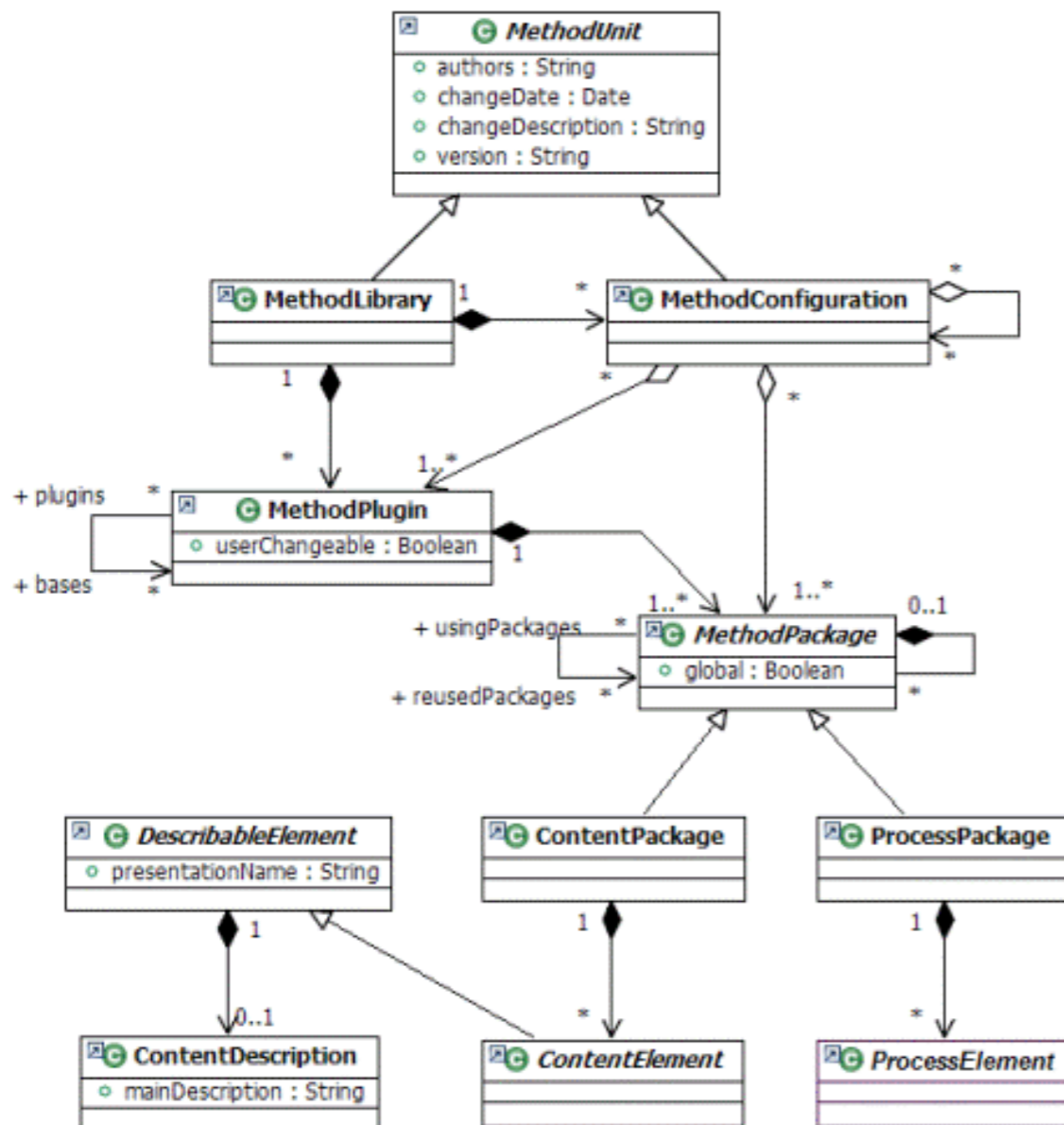
The Unified Modeling Language (UML) offers various diagrams to describe software designs, architectures, and requirements models.

This is a class diagram describing a small part of the Eclipse development environment.

The model here abstracts from the details of the source code. Unfortunately, reading the code will eventually be necessary.

We can see which classes depend on which other classes, which components specialize other ones, and we can see some of the kinds of information that they manage.

Finite state models
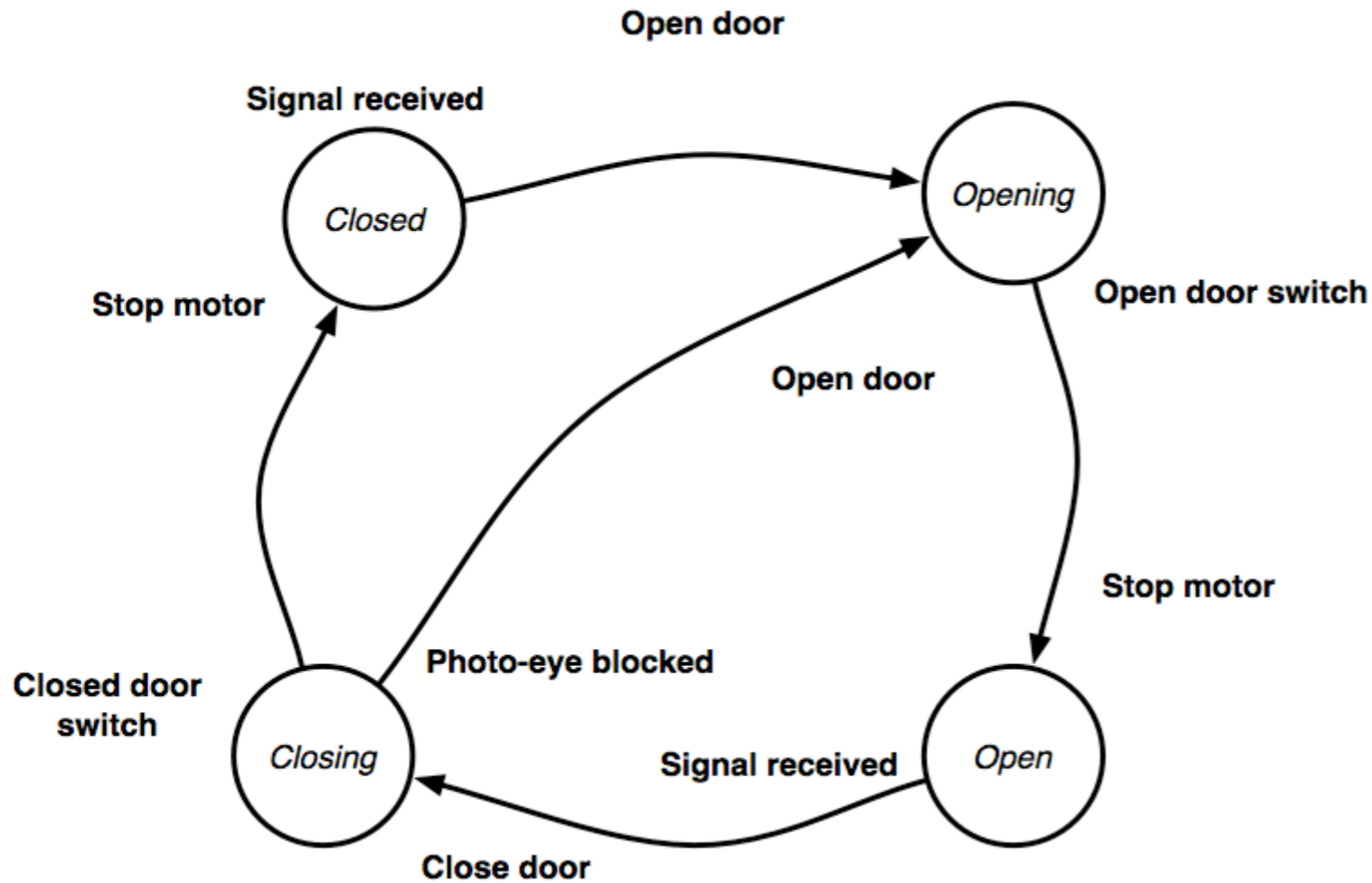
Finite state models express states and transitions of a process

Finite state models are ideal for modeling closed systems. They can be exhaustively analyzed, and readily implemented.

Finite state models occur in many branches of computer science, including string matching (regular expressions), concurrency control (model checking), and software modeling (UML).

https://en.wikipedia.org/wiki/Finite-state_machine

Roadmap

> Computational Thinking
> The Thinking Tools
> Tool Zoom In: Recursion
> Tool Zoom In: Modeling
> **Outlook**

Computational thinking...

means: thinking like a computer scientist

is a skill that is useful for everybody

involves a broad range of mental tools

involves solving problems, designing systems, understanding human behavior

creative commons

Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

    Share — copy and redistribute the material in any medium or format

    Adapt — remix, transform, and build upon the material for any purpose, even commercially.

    The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/