# Fundamental Algorithms

# (WS 2008/09)

Stefan Schwoon

Institut für Informatik (I7)
Technische Universität München

# Fundamental Algorithms

Lectures: Tuesday, 11:45–13:15, Room 00.08.038

Lecturer: Stefan Schwoon, schwoon@in.tum.de

Office hours: by appointment, Room MI 03.011.053

Tutorials: Wednesday, 11:00–12:30, Room tba

Tutor: Dejvuth Suwimonteerabuth, suwimont@in.tum.de

Office hours: by appointment, Room MI 03.011.055

Tutorials are voluntary!

Credits: 3 ECTS credits

Written exam at the end of the semester

# Announcements

Website:

`http://www7.in.tum.de/um/courses/fundalg/ws0809/`

see the website for slides, exercise sheets etc

Email list: (to be collected)

for urgent announcements (sudden schedule changes etc)

# Other things

Intended audience:

CSE master students

Prerequisites:

basic knowledge of (discrete) mathematics

Literature: (for background reading)

Cormen, Leiserson, Rivest, Stein: *Introduction to Algorithms*, 2nd Print, 2001

Sedgewick: *Algorithms*, 2nd Print, 2002

Heun: *Grundlegende Algorithmen*, 2. Auflage, 2003

Vöcking et al: *Taschenbuch der Algorithmen*, 2008

# Part 1: Introduction

# Goals of the course

Learn important algorithmic concepts

Formalize problems, making them amenable to algorithmic solutions

Devise your own algorithmic solutions

Prove algorithms correct

Determine efficiency of algorithms

# What is “fundamental”?

Problems on discrete structures:

- Sorting, searching
- Lists, trees, graphs, other data structures

Arithmetic problems

Complexity measures, proofs of correctness

# What are “algorithms”?

An algorithm is an unambiguously defined sequence of *atomic* instructions transforming some input into some output.

Real-life example: recipes for cooking or baking

Atomic instruction: an action that can be performed by the processor without further explanation.

Alternative view: Given some pre-condition on the inputs, an algorithm establishes a post-condition.

Here, we are interested in algorithms to be performed by computers.

Notice: not all computer programs match this description. Some are not definable in terms of inputs and outputs, but in terms of stimuli and reactions (example: operating systems). The latter are called reactive programs. We are not concerned with reactive programs in this course.

Important properties of computer algorithms:

terminate on all inputs (matching some pre-condition)

finitely described

work on infinitely many different inputs

# Example: Sorting

Problem: We are given some set of *comparable* objects (the input), e.g.

books (in a library), compared by title

numbers

We are supposed to arrange the objects in ascending/descending order (the output).

Important real-life problem (e.g., library: books are ordered in the shelves)

# Fundamental issues

Notation for writing down algorithms (branching, repetition)

Proofs of correctness

Determining how many instructions an algorithm takes (complexity)

# Notation for algorithms

We introduce some basic notation for algorithms.
Later, we may extend it as needed.

Data: A variable denotes some storage location in the computer. Variables are identified by their name; variables with different names denote different storage locations. For now, each variable is assumed to store an integer value.

Examples: `i`, `j`, `k`

We also consider arrays. An array denotes a sequence of storage locations. A particular element of the sequence is referenced by an index (written in brackets). An index is any integer expression, including variables. An array is specified along with its range of indices.

Examples: array `a` with indices `0..n`

```
a[2], a[i], a[i+j]
```

Assignment: An assignment manipulates data; it takes the form

`v:=e`,

where `v` is a variable and `e` some arithmetic expression (involving, e.g., variables, array locations, and constants). It puts the value of the expression into the storage location denoted by the variable.

Examples: `i = j+3`, `a[i] = a[j]`, `i=i-1`

Sequence: Given two instructions (e.g., assignments), we can arrange them in a sequence separated by semicolon, denoting that the first should be performed before the second.

Example: `a[i] = a[j]; i = i+1`

Branching: Allows to do different things, depending on the input.

Example: `if (i = 1) then` ⟨do something⟩ `else` ⟨something else⟩ `fi`

Repetition: Allows to repeat (a sequence of) actions multiple times until some condition is met (or rather, violated).

Example: `while (i > 0) do i = i-1 od`

# An abbreviation

Sometimes we want to perform the same sequence of actions for different values of some variable. If `i` is a variable and `S` is a (sequence of) actions, then

```
for i = 1 to n do S od
```

is an abbreviation for

```
i = 1;

while (i <= n) do S; i=i+1 od
```

# An example

Here's a small example algorithm that adds up numbers from 1 to $n$, storing the sum in variable `s`. In other words the algorithm has input $n$, and its output is the aforementioned sum.

```
s = 0;

for i = 1 to n do

  s = s + i

od
```

In the alternative view, since we place no particular condition on the input $n$, its precondition is $n \geq 0$, and its postcondition is $s = \frac{1}{2}n(n+1)$.

# Correctness

We claim that after the algorithm $s = \frac{1}{2}n(n+1)$ holds.
How can we prove it?

From mathematics, we know the principle of induction. It can be used to prove that something holds for all $n \geq 0$, such as $\sum_{i=1}^{n} i = \frac{1}{2}n(n+1)$.

We first prove that some claim holds for $n = 0$.

Then we prove that, assuming that the claim holds for $n = i$, for some $i \geq 0$, then it follows that the claim holds also for $n = i + 1$.

# Proving loops correct

To prove that some postcondition $\mathcal{Q}$ holds in a program with a loop such as

```
for i = 1 to n do S od
```

we use a technique similar to induction.

More precisely, our aim is to find some condition $\mathcal{I}(i)$ (called an invariant), which depends on $i$, and enjoys the following three properties:

When the execution first reaches the loop, $\mathcal{I}(0)$ holds.

If $\mathcal{I}(j-1)$ holds for some value $j$, and S is executed for $i = j$, then after execution of $S$ $\mathcal{I}(j)$ holds.

$\mathcal{I}(n)$ implies $\mathcal{Q}$.

In our example, we could choose $\mathcal{I}(i)$ to be $s = \frac{1}{2}i(i+1)$.
(Prove the three properties!)

Recall that a `for` loop is just an abbreviation for a `while` construct. Assume that we are given a precondition $\mathcal{P}$, a postcondition $\mathcal{Q}$, and an algorithm of the form

```
while E do S od
```

We can generalise the invariant method to while constructs. The goal is to find some invariant $\mathcal{I}$ with these properties:

$\mathcal{P}$ implies $\mathcal{I}$;

if $\mathcal{I} \wedge E$ holds, then after execution of $\mathtt{S}$ $\mathcal{I}$ holds again;

$\mathcal{I} \wedge \neg E$ implies $\mathcal{Q}$.

# A simple sorting algorithm (Insertion Sort)

Input: an array `a` with indices $1 \ldots n$.
Output: the original array `a`, but sorted in ascending order

```
for i = 1 to n do
  j = i-1; k = a[i];
  while j ≥ 1 ∧ a[j] > k do
    a[j+1] = a[j]; j = j-1
  od;
  a[j+1] = k
od
```

How can we prove this program correct? Which invariants should we use?

# Complexity / Efficiency

It will be interesting for us to see how *fast* (or: efficient) algorithms are, i.e. how many instructions they need to execute in order to produce the desired output.

In the Insertion Sort example, a good measure for efficiency is the number of times that the body of the while loop is executed; all other instructions are executed $n$ times.

In the worst case (when the array is originally sorted in descending order), this will happen $\frac{1}{2}n(n-1)$ times. (Why?)

In the best case (when the input is already sorted in ascending order), the while body is never executed.

# Discussion of complexity

The Insertion Sort algorithm behaves as follows:

In the best case, the number of instructions is a linear function of $n$.

In the worst case, it is a quadratic function.

In the average case, . . . ?

Complexity analysis usually focuses on worst-case analysis.

In this sense, Insertion Sort is “quadratic”. We shall see that there are better solutions for sorting!

# Comparing functions

Is every linear function better than any quadratic function?

Let us compare $f(n) = 100 \cdot n$ and $g(n) = n^2$.

| $n$ | $f(n)$ | $g(n)$ |
|---|---|---|
| 1 | 100 | 1 |
| 5 | 500 | 25 |
| 10 | 1000 | 100 |
| 20 | 2000 | 400 |
| 50 | 5000 | 2500 |
| 100 | 10000 | 10000 |
| 200 | 20000 | 40000 |
| 500 | 50000 | 250000 |

# Growth of functions

We observe: $g(n)$ eventually becomes bigger than $f(n)$.

Let us generalize this observation: consider $f(n) = d \cdot n$, for some constant $c$.

Independently of the choice of $d$, $g(n)$ eventually "overtakes" $f(n)$!

Complexity analysis is concerned with the behaviour of algorithms for large inputs. ("What happens if we want to solve BIG problems?")

Therefore, we shall always consider linear functions as better than quadratic functions.

We also want to abstract from "pesky" things like constant factors and minor terms in the complexity. We shall formalize this in the next couple of slides.

# Landau symbols

The following definitions characterize how functions compare asymptotically, i.e. for large inputs.

Let $f, g : \mathbb{N} \to \mathbb{R}^+$. We define $\mathcal{O}(g)$ as the set of functions that "at most as fast" as $g$. Formally:

$$f \in \mathcal{O}(g) \quad \Leftrightarrow \quad \exists c > 0 \; \exists n_0 \; \forall n \geq n_0 : f(n) \leq c \cdot g(n)$$

Intuitively, from some point on $g$ is at least as big as $f$, modulo constant factors.

# Examples

Let $f(n) = d \cdot n$ and $g(n) = n^2$. We show that $f \in \mathcal{O}(g)$.

Proof: Choose $c = 1$ and $n_0 = d$. Then for all $n \geq n_0$:

$$f(n) = d \cdot n = n_0 \cdot n \leq n^2 = c \cdot g(n)$$

Let $f(n) = \frac{1}{2}n(n-1)$ and $g(n) = n^2$. Again, $f \in \mathcal{O}(g)$.

Proof: Immediate, choose $c = n_0 = 1$.

# Some important classes

"Constant" functions: $\mathcal{O}(1)$

"Logarithmic" functions: $\mathcal{O}(\log n)$

"Linear" functions: $\mathcal{O}(n)$

"Quadratic" functions: $\mathcal{O}(n^2)$

"Polynomial" functions: $\bigcup_{k=0}^{\infty} \mathcal{O}(n^k)$

"Exponential" functions: $\bigcup_{c \in \mathbf{R}^+} \mathcal{O}(c^n)$

# Other Landau symbols

$\mathcal{O}$ is the one Landau symbol that ist most often used in practice. (Especially because it is useful for worst-case analysis.) But there are other, similar notions:

$f$ grows “at least as fast” as $g$:

$$f \in \Omega(g) \quad \Leftrightarrow \quad \exists c > 0 \; \exists n_0 \; \forall n \geq n_0 : f(n) \geq c \cdot g(n)$$

$f$ grows “just as fast” as $g$:

$$\Theta(g) \quad = \quad \mathcal{O}(g) \cap \Omega(g)$$

$f$ grows “strictly slower” than $g$:

$$f \in o(g) \quad \Leftrightarrow \quad \forall c > 0 \; \exists n_0 \; \forall n \geq n_0 : \; f(n) \leq c \cdot g(n)$$

$f$ grows “strictly faster” than $g$:

$$f \in \omega(g) \quad \Leftrightarrow \quad \forall c > 0 \; \exists n_0 \; \forall n \geq n_0 : \; f(n) \geq c \cdot g(n)$$

# Example

Let $f(n) = d \cdot n$ and $g(n) = n^2$. We show that $f \in o(g)$.

Proof: Given any $c > 0$, choose $n_0 \geq \frac{d}{c}$. Then for all $n \geq n_0$:

$$d \cdot n = c \cdot \frac{d}{c} \cdot n \leq c \cdot n_0 \cdot n \leq c \cdot n^2$$

# Useful laws

Transitivity: $f \in \mathcal{O}(g)$ and $g \in \mathcal{O}(h)$ implies $f \in \mathcal{O}(h)$.

Reflexivity: $f \in \Theta(f)$

Symmetry: $f \in \Theta(g)$ implies $g \in \Theta(f)$

Antisymmetry: $f \in \mathcal{O}(g)$ implies $g \in \Omega(f)$

# Part 2: Sorting

# Another sorting algorithm (Bubble Sort)

Input: an array `a` with indices $1 \ldots n$.
Output: the original array `a`, but sorted in ascending order

```
for k = 1 to n-1 do
  for i = 1 to n-k do
    if a[i] > a[i + 1] then
      tmp = a[i]; a[i] = a[i+1]; a[i+1] = tmp
fi   od  od
```

Note: Line 4 swaps elements at two positions. In the future, we will abbreviate this with a `swap` instruction, for better clarity.

# Another sorting algorithm (Bubble Sort)

Input: an array `a` with indices $1 \ldots n$.
Output: the original array `a`, but sorted in ascending order

```
for k = 1 to n-1 do
  for i = 1 to n-k do
    if a[i] > a[i + 1] then
      swap a[i], a[i+1]
fi   od  od
```

# How does Bubble Sort work?

Each iteration of the “outer loop” (with $k$) ensures that the $k$-th biggest element is in the right position.

After one iteration, the biggest element is in the right position.

After two iterations, the two biggest elements are in the right position.

...

Invariant: $\mathcal{I}(k) \mathrel{\widehat{=}}$ “elements $a[n-k+1]$ to $a[n]$ are in the right order”

The inner loop (with $i$) “pushes” bigger elements “downwards”.

Invariant: $\mathcal{J}(i) \mathrel{\widehat{=}}$ “element $a[i]$ is the biggest among the first $i$ elements”

# Analysis of Bubblesort

One easily sees that the algorithm makes $\frac{1}{2}n(n-1)$ steps, regardless of the input.

This means the algorithm *always* takes $\mathcal{O}(n^2)$ steps, (even $\Theta(n^2)$).

Comparison with Insertion Sort:

Insertion Sort also takes $\mathcal{O}(n^2)$ steps for every input ...

...but not $\Theta(n^2)$ for every input (just in the worst case!).

For some inputs, Insertion Sort takes only linear time, but Bubble Sort always takes quadratic time.

Therefore, Bubble Sort can be seen to be inferior to Insertion Sort, even though both have the same worst-case complexity.

# Can we do better?

Both Insertion Sort and Bubble Sort take quadratic time in the worst case?
A natural question is, can one do better?

In the following, we will see that the answer is positive.

There are algorithms with $\mathcal{O}(n \log n)$ worst-case running time.
Notice that $n \log n \in o(n^2)$, i.e. grows strictly slower.

Moreover, one can prove that an $n \log n$ bound is optimal.

# Heaps

A heap is a binary forest with certain properties.

> Forest: directed acyclic graph. Nodes without incoming edges are called *roots*. All other nodes have one incoming edge.
>
> Binary forest: Every node has at most two outgoing edges. Nodes without outgoing edges are called *leaves*. If an edge leads from node $v$ to node $w$, $w$ is a *child* of $v$.

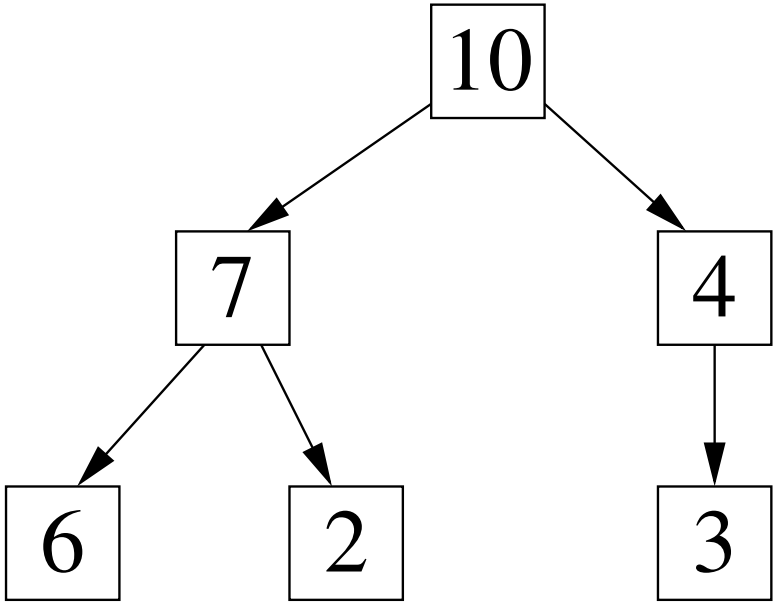
(We will deal more formally with trees and other graphs later in the course. For now this informal definition is sufficient.)

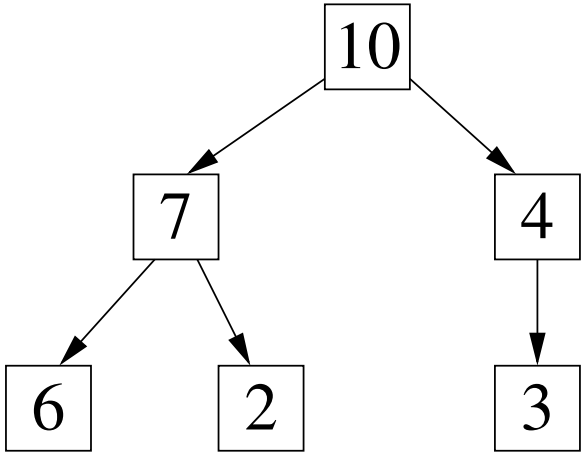
Additional properties of heaps:

Every node is labelled by a number.

For each node, its number is larger than those of its children.

Example:






Insight: An array (slice) can be interpreted as a binary forest.

The children of the element at position $i$ are those at positions $2i$ and $2i + 1$ (if these positions exist).

| 10 | 7 | 4 | 6 | 2 | 3 |
|---|---|---|---|---|---|

# Array slices and heaps

Consider the array slice from positions $\lfloor \frac{n}{2} \rfloor + 1$ to $n$.

Every position is a root.

Therefore, this slice is a heap.

If the slice $1 \ldots k$ is a heap, then so is the slice $1 \ldots k - 1$.

If the slice $k \ldots n$ is a heap, then the slice $k - 1 \ldots n$ is a heap except for position $k - 1$.

We may need to "slide" element $k - 1$ "downwards".

# Algorithm Heapify

Input: an array `a` with indices $1 \ldots n$, indices $k$ and $i$

Output: if the slice $i+1 \ldots k$ is a heap initially, then afterwards $i \ldots k$ is a heap.

```
while i ≤ k do
  largest = i;
  if 2i ≤ k and a[i] < a[2i] then largest = 2i fi;
  if 2i + 1 ≤ k and a[largest] < a[2i + 1] then largest = 2i+1 fi;
  if largest ≠ i then
    swap a[i], a[largest]; i = largest
  else   i = k+1   fi
od
```

# Algorithm Heapsort

Input: an array `a` with indices $1 \ldots n$.
Output: the original array `a`, but sorted in ascending order

```
for i = ⌊n/2⌋ downto 1 do
  heapify a,n,i
od
for i = n downto 2 do
  swap a[1],a[i];
  heapify a,i-1,1
od
```

# Notation for algorithms II

The Heapsort algorithm (as denoted on the preceding slides) consists of two parts.

the Heapify algorithm;

the Heapsort algorithm proper, which uses Heapify

We should clarify what exactly a line like this means:

```
heapify a,i-1,i
```

# Procedures and functions

In mathematics, we are used to define functions in terms of other functions, or even recursively.

$$n! = \begin{cases} 1 & \text{n=0} \\ n \cdot (n-1)! & \textit{otherwise} \end{cases}$$

This means: In order to evaluate $n!$, first evaluate the expression $(n-1)!$. Only once one knows that value, one can continue to compute $n!$.

For algorithms, we adopt this idea:

> A procedure $P$ is simply an algorithm (with certain inputs and outputs). Another algorithm (or procedure) $Q$ can call $P$. Doing so means that $Q$ suspends its operation until $P$ has finished, and then continues.
>
> In the Heapsort algorithm, Heapify is such a procedure.

# Procedures with Parameters

In order for procedures to interoperate, they need to share data.

For instance, both Heapify and Heapsort procedures work on array `a`.

But also, both work on the variable `i`, but use them for different purposes.

We need to distinguish these cases:

- We will declare what data is shared between procedures, and how.
- We call the data shared between two processes parameters.
- We distinguish in/out parameters, in parameters, and return values.

# In/out parameters

`a` is called an in/out parameter.

The Heapify procedure takes `a` from Heapsort.

It modifies `a`.

Once it terminates, the modified `a` is used by Heapsort.

In contrast, `i` is called an in parameter.

The Heapify procedure takes a parameter `i`. In the first call, this is the variable `i` of Heapsort, in the second call it is the value 1.

Heapify then modifies `i`. But in Heapsort, we assume that, after the calls to Heapify, `i` has not changed.

Conclusion: Heapify has its own copy of `i`, and does not pass its value back to Heapsort.

# List of parameters

In the future, we will distinguish these cases carefully: Each algorithm/procedure will be equipped with a list of parameters, i.e. its inputs and outputs.

Order in the list matters!

Each parameter is of the type *in* or in/out.

In Heapify, the list of parameters is `a, k, i` (in this order!)

`a` is an in/out parameter, `k` and `i` are in parameters.

In Heapsort, the parameters are `a` (in/out) and `n` (in).

# An operational model for parameters

The mechanics of procedures and parameters can be made more precise with the idea of a stack.

A stack is an ordered list of data items with the following operations:

We can add another element to the “top” of the stack.

We can remove the topmost element from the stack.

We can read or write on the top element of the stack.

(Imagine a stack of papers!)

# Procedures and stacks

When procedures interoperate, we can imagine that each call creates a new element on top of the stack.

That stack element contains the variables (including the parameters) of the procedure, which we can subsequently read and write to.

Each time a procedure is called, a fresh stack element is created.

> Upon creation (call), the parameters are initialised with the data specified by the caller.
>
> For instance, in the first call from Heapsort to Heapify,
>
> > Heapify's `a` becomes Heapsort's `a`.
> >
> > Heapify's `k` becomes Heapsort's `n`.
> >
> > Heapify's `i` becomes Heapsort's `i`.

Notice that this may lead to multiple copies of the same variable, but on different stack elements. Only the topmost stack element can be accessed.

A procedure could call itself (this is called recursion), in which case each call creates a new copy of its variables.

Upon termination of a procedure, its in/out parameters are copied back to the respective variables in the calling procedure.

In the example, Heapify's `a` is copied into Heapsort's `a` when Heapify terminates.

# A recursive procedure: Factorial

Here's an example of a recursive procedure, which computes a factorial number. Its parameters are `n` and `k`, where `n` is an in parameter, and `f` is an in/out parameter.

If `n` is a non-negative integer, then afterwards `f` equals the factorial of `n`.

This procedure (called Factorial) calls itself recursively.

```
procedure Factorial
if n=0 then
  f = 1
else
  Factorial n-1,f;
  f = f*n
fi
```

# Return values

When a procedure “computes something” (like a factorial), it is more convenient to express this directly, rather than through an in/out parameter.

We introduce a `return` statement that takes an expression, e.g. `return 0`.

The call to such a procedure (say, `f`) is now an assigment, e.g. `x = f(...)`.

The meaning of this is as follows:

> `f` has got an addition unnamed/invisible in/out parameter.
>
> When the procedure is called, the left-hand side variable of the assignment becomes that parameter.
>
> The `return` statement terminates the procedure and assigns the value of the expression to that parameter.

Note: We can generalise this to multiple return values!

# Factorial expressed with return values

The parameter is still a non-negative integer `n`, and the return value has got the value `n!`

```
procedure Factorial

if n=0 then

  return 1

else

  return n*Factorial(n-1)

fi
```

# A recursive sorting procedure

In the following, we introduce another sorting procedure (yet another one!)

This one works *recursively* and employs a divide-and-conquer strategy.

Its worst case complexity will turn out to be quadratic, but on average it performs very well.

The procedure employs non-determinism (i.e., an element of random is involved).

The procedure is called Quicksort.

# Concept of Quicksort

We are given a (slice of an) array `a`.

Suppose the lowest index of the array is $\ell$ and the highest one $h$.

Suppose we have a way to partition the array into two parts, left and right, such that all values in the left-hand part are smaller than the values in the right-hand part. Then we just need to sort both parts individually.

To achieve this partiioning, Quicksort picks a pivot element $p$ (any element of the array).

# Partitioning

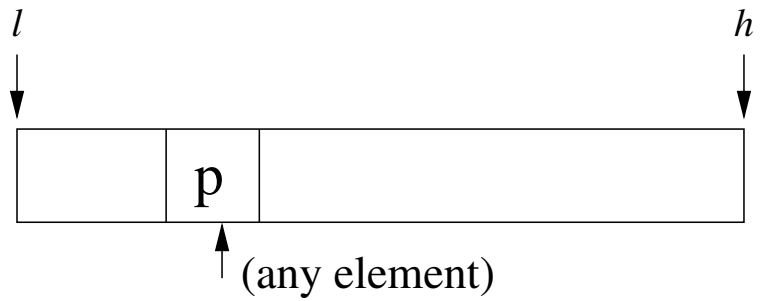
Quicksort actually creates *three* partitions:

The leftmost partition contains all values smaller than *p*.

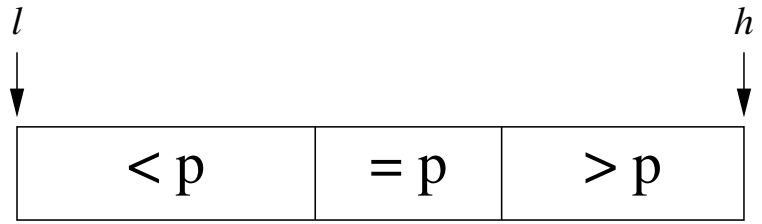
All elements in the central partition are equal to *p*.

The rightmost partition contains all values larger than *p*.

Before:




After:

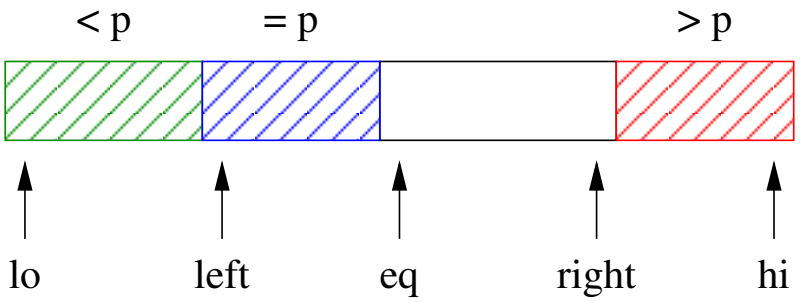


# The partitioning algorithm

This algorithm is called `partition`. Its parameters are `a` (in/out, an array), `lo,hi` (in). Upon termination, the part of `a` between indices `lo` and `hi` is a partitioning (of the kind mentioned in the previous slide), and two values are returned that delimit the partitions.

```
procedure partition
p = any element of a[lo] through a[hi];
left = eq = lo; right = hi;
while (eq <= right) do
  if (a[right] > p) then
    right = right - 1
  else
    swap a[eq], a[right];
    if (a[eq] < p) then
      swap a[left], a[eq];
      left = left + 1;
    fi
    eq = eq + 1
  fi
od;
return left-1,eq
```

The partitioning algorithm maintains the following invariant in the while loop:




- The indices from `lo` to `left-1` are smaller than `p`.
- The indices from `left` to `eq-1` are equal to `p`.
- The indices from `eq` to `right` are arbitrary.
- The indices from `right+1` to `hi` are greater than `p`.

Notice that some of these ranges can be empty (e.g., initially).

The while loop ends when the third range becomes empty.

# The Quicksort procedure

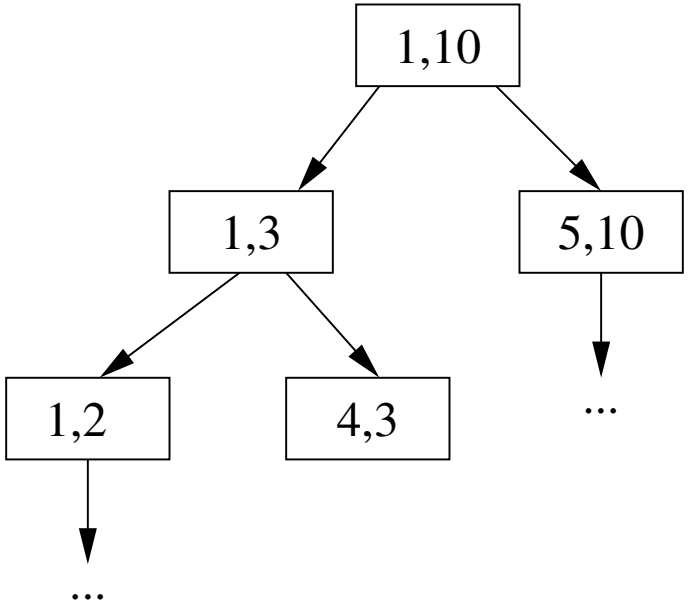
Quicksort takes an array `a` (in/out), two parameters `lo` and `hi` (in), and sorts it between indices lo and hi.

```
procedure quicksort

if hi <= lo then terminate;

p,q := partition(a,lo,hi);

quicksort(a,lo,p);

quicksort(a,q,hi);
```

# Analysis of Quicksort

Let us regard the “tree” of procedure calls that occurs when Quicksort runs on a particular array. I.e. if the Quicksort procedure is called with the range $x, y$ and calls itself recursively with $x', y'$, then we draw an edge from $x, y$ to $x', y'$.

E.g., if Quicksort is used on an array of length 10, the call tree may look like this:



Each partitioning takes linear time (in the size of its array slice).

Thus, the running time of Quicksort is proportional to the sum of the sizes of the array slices that occur in the call tree.

Let us regard the case where all elements in the array are different. (This is the worst case because each pivot is unique.) Then each call will "put" one element into its correct position, and there will be $n$ calls altogether.

The height of the call tree is at least (approximately) $\mathrm{lb}\, n$ and at most $n$.

In the first case, each pivot is the median of the array values, so the partitions are always equal (plus/minus one). Then the running time can be seen to be $\mathcal{O}(n \cdot \log n)$.

In the latter case, the pivot is always the smallest or the largest element, so one partition is empty and the other just one less than the original slice. Then the running time is $\mathcal{O}(n^2)$.

Thus, the worst case running time is $\mathcal{O}(n^2)$.

# The role of non-determinism

The pivot is picked randomly (i.e., we pick any of the elements).

Therefore, it would be correct to pick, e.g., always the rightmost element of the slice (i.e., deterministically).

But then, the algorithm behaves “badly” (quadratic) when used on arrays that are already sorted. (Why?)

In fact, any deterministic strategy for picking the pivot has inputs that lead to a quadratic running time.

On the other hand, if we pick the pivot non-deterministically, then (depending on the choices) every input can lead to either $n \log n$ or $n^2$ running time.

On average, however, non-deterministic choice will partition the array with the proportions $1 : 3$. (Why?)

Then the depth of the call tree will be $\log_{4/3} n$. Therefore, the running time is still $\mathcal{O}(n \log n)$ on average.

> Note: We have $\log_a n = \log_a b \cdot \log_b n$, i.e. for all $n$ two logarithms with bases $a, b$ differ only by a constant factor. Therefore, Landau notation abstracts from the logarithm base.

# Another look at worst-case/average complexity

Our algorithms have multiple possible inputs, and in non-deterministic algorithms there may be multiple possible executions for every input, whereas in deterministic algorithms there is just one execution for each input. Complexity can be measured w.r.t. both quantities.

If $\mathcal{A}$ is some algorithm, let us denote by $I_{\mathcal{A}}(n)$ the set of all (valid) inputs to $\mathcal{A}$ of size $n$. (E.g., all arrays of size $n$ for some sorting algorithm).

Now, if $i$ is an input to algorithm $\mathcal{A}$, then let $R_{\mathcal{A}}(i)$ denote the set of all $k$ such that some execution of $\mathcal{A}$ on input $i$ takes $k$ steps. (For deterministic algorithms, $|R_{\mathcal{A}}(i)| = 1$.)

We can now denote the worst (resp. average) behaviour of $\mathcal{A}$ on $i$ as the maximum (resp. average) of the numbers in $R_{\mathcal{A}}(i)$, denoted $\max_{\mathcal{A}}(i)$ (resp. $\text{avg}_{\mathcal{A}}(i)$). For deterministic algorithms, both conincide.

However, we are usually not interested in individual inputs, but some aggregate measure over all inputs. The standard way to do this is to consider the *worst possible input*. (E.g., the worst that some malicious adversary could do to provoke a bad behaviour in $\mathcal{A}$.)

This can be captured by the functions

$$f_{\mathcal{A}}(n) := \max\{ \max_{\mathcal{A}}(i) \mid i \in I_{\mathcal{A}}(n) \} \qquad \text{and} \qquad g_{\mathcal{A}}(n) := \max\{ \text{avg}_{\mathcal{A}}(i) \mid i \in I_{\mathcal{A}}(n) \}$$

Again, for deterministic algorithms, $f_{\mathcal{A}}(n) = g_{\mathcal{A}}(n)$.

If $\mathcal{A}$ is some deterministic version of Quicksort, we have both $f_{\mathcal{A}}(n), g_{\mathcal{A}}(n) \in \mathcal{O}(n^2)$.

If $\mathcal{A}$ is the non-deterministic version of Quicksort, we have $f_{\mathcal{A}}(n) \in \mathcal{O}(n^2)$ but $g_{\mathcal{A}}(n) \in \mathcal{O}(n \cdot \log n)$.

If $\mathcal{A}$ is Heapsort, we have both $f_{\mathcal{A}}(n), g_{\mathcal{A}}(n) \in \mathcal{O}(n \cdot \log n)$.

Summary:

Non-determinism is advantageous for Quicksort.

Non-det. Quicksort is slower than Heapsort only if we're "unlucky".

Det. Quicksort is slower than Heapsort if we are given a "difficult" input.

# A lower bound for sorting

We have seen that there are algorithms that take $\mathcal{O}(n \log n)$ running time in the worst case (e.g., Heapsort, Quicksort on average).

Question: Can we do better?
Answer: No (at least not in general)

To reason about this, let us make two assumptions:

> We restrict the inputs to the case where all array elements are different. (The above holds even with this restriction.)
>
> We consider only algorithms that work with comparisons to determine the relative order of elements. (This is a proper restriction. But all known algorithms that work differently have other restrictions.)

Any sorting algorithm (under the above restrictions) needs to distinguish $n!$ different relative orderings of its inputs.

Every comparison excludes at most half of the relative orderings.

Thus, every algorithm needs to make at least $\operatorname{lb} n!$ comparisons.

Using Stirling's inequality ($n! \geq n^{n/2}$) we get $\operatorname{lb} n! \in \mathcal{O}(n \log n)$.

Therefore, an algorithm like Heapsort is asymptotically optimal under the given assumptions.

# Sorting in less than $n \log n$ time

The lower bound shown on the previos slides holds under the given assumptions (i.e. all we know is that the array elements are all different).

If we have additional knowledge about our input, we can design algorithms with better running times. (But they will work correctly only if the "additional knowledge" holds.)

We shall look at examples of such "special-case" algorithms:

- Counting Sort
- Radix Sort

# Counting Sort

Counting Sort works under the following assumptions:

- The size of the array $a$ to be sorted is $n$.
- The elements in the array are all in the range from $1$ to $k$.

The algorithm works as follows:

- It uses an array $b$ with indices $1 \ldots k$. All elements are zero initially.
- It traverses the elements of $a$ once; every time it sees an element with value $i$, it increases $b[i]$.
- After this, it fills $a$ from left to right with $b[i]$ elements of value $i$ each.

Running time: $\mathcal{O}(n + k)$ (linear!)

Parameters are array `a` (in/out) and `k,n` (in).

```
procedure countingsort
for i = 1 to k do  b[i] = 0  od;
for i = 1 to n do  b[a[i]] = b[a[i]] + 1  od;
j = 0;
for i = 1 to k do
  for ℓ = 1 to b[i] do
    j = j + 1;
    a[j] = i
  od
od
```

# Radix Sort

Radix Sort works under the following assumptions:

- The size of the array $a$ to be sorted is $n$.
- The elements in the array are all in the range from $0$ to $2^k - 1$.

The algorithm works as follows:

- It uses an array $b$ with indices $1 \ldots n$.
- It traverses the elements of $a$ once and fills $b$ "from the margins"; every element whose most significant bit is $0$ is put to the left, and to the right otherwise.
- Then, the two partitions of $b$ are sorted recursively according to the second most significant bit and so forth.

Running time: $\mathcal{O}(n \cdot k)$

To give an algorithm that implements Radix Sort, we assume an auxiliary procedure `radix_aux` (given on the next slide), which takes five parameters:

an array `a` (in/out);

indices `lo, hi` (in);

numbers `upper, lower` (in).

Pre-condition: In the slice from index `lo` to `hi`, `a` contains elements whose values are at least `lower` and at most `upper`.

On return from `radix_aux`, `a` is sorted between the indices `lo` and `hi`.

Then Radix Sort can be implemented by a procedure call `radix_aux(a,1,n,0,`$2^k - 1$`)`.

```
procedure radix_aux
if lo >= hi then return fi;
if upper = lower then return fi;
k = lo-1;  ℓ = hi+1;  mid = (lower+upper)/2;
for i = lo to hi do
  if a[i] <= mid then
    k = k + 1;  b[k] = a[i]
  else
    ℓ = ℓ - 1; b[ℓ] = a[i]
  fi
od
radix_aux(b,lo,k,lower,mid);
radix_aux(b,ℓ,hi,mid+1,upper);
for i = lo to hi do  a[i] = b[i]  od
```

# Final notes about sorting algorithms

We discussed methods that work with comparisons only, and some special ones. We also discussed some programming paradigms (such as recursion, divide-and-conquer, non-determinism).

For comparison algorithms, we cannot do better than $\mathcal{O}(n \cdot \log n)$. With special knowledge about the inputs, we can do better.

Notice that comparison algorithms do not “compute” with the array elements. All they need to work is that array elements are comparable. Thus, comparison algorithms work even on non-numeric data for which there exists some total ordering (e.g., names).

# Part 3: Searching

# Searching

In general, searching is the following problem:

> We are given a collection of data items (e.g., a set of numbers) and a so-called key value (or just: key). We want to know whether the key is contained among the data items.

Example: An array of integers and an integer value as key.

We will consider this problem in arrays and other data structures (such as trees).

# Searching in arrays

Input: an array `a` of integers (range `1..n`, in-parameter) and an integer `k` (in).

Return value: an index `i` s.t. `a[i]=k`, or special value `NIL`

```
procedure array_search

for i = 1 to n do

  if a[i] = k then return i fi

od;

return NIL
```

Complexity (worst-case): $\mathcal{O}(n)$

# Search in sorted arrays

Inputs and return values as before, but we assume that `a` is sorted in ascending order. Then we can stop searching as soon as the array elements get bigger than `k`.

```
procedure array_sorted_search

for i = 1 to n do

  if a[i] > k then return NIL fi;

  if a[i] = k then return i fi

od;

return NIL
```

Complexity (worst-case): still $\mathcal{O}(n)$!

# Binary search

Searching a sorted array in linear order isn't smart - if we ask (unsuccessfully) whether the key is in the first element, we must look for the key in the remaining $n - 1$ elements.

Starting in the middle is much smarter - it eliminates half the indices at once! (Think of searching in a telephone directory...)

The next slide shows a searching algorithm (working on a sorted array) that takes $\mathcal{O}(\log n)$ steps by applying the "starting in the middle" principle recursively!

So searching in sorted arrays can be much quicker (which is why telephone directories are sorted...)

```
procedure binary_search
lo = 1; hi = n;
while lo <= hi do
  mid = (lo+hi)/2;
  if a[mid] = k then return mid fi;
  if a[mid] < k then lo = mid+1 fi;
  if a[mid] > k then hi = mid-1 fi;
od;
return NIL
```

# Derived data structures

In the following, we work with basic data types (integers, for now) and derived data types.

Derived data types are:

- arrays (an array of $T$, where $T$ is some data type, with a given range of indices, as before);
- records (a collection of data items belonging to some "logical" object);
- pointers (a pointer to $T$, where $T$ is some data type, an "edge" to some piece of data).

Variables can be of basic or derived types.

Using this, we can enrich our vocabulary for expressing algorithms. Each expression is of some type. Assignments $x = E$ make sense only if both $x$ and $E$ have the same type.

# Records

A record is type. It is a collection of data items that logically belong together.

Each item has got a name and a type.

> For instance, we can define a type for coordinates in two-dimensional space consisting of two data items: two integers with names $x$ and $y$. Let's call this type *TwoD*.

If $r$ is a record and $n$ the name of some item in $r$ (of type $T$), then $r.n$ denotes the corresponding data item. The expression $r.n$ has type $T$.

Records can be nested, e.g. some item of a record may of a (different) record type.

# Pointers

A pointer is an “edge” pointing to some object (e.g., some variable) in memory. The A pointer is parametrized by a type (the type of the variable it points to).

(In most computer architectures, a pointer is simply a variable containing a memory address. Accordingly, the “value” of a pointer is the *object* it points to – not the *value* of that object.)

A special value for pointers is *NIL*, meaning that the pointer points to “nothing”.

A pointer can be to any type (basic, array, record, or some other pointer).

# Working with pointers

When using pointers, we distinguish carefully between the pointer itself and the object it points to.

If $p$ is a pointer, then $p$ denotes the pointer itself. Assigning to $p$ makes the pointer point to something else (e.g., $p = NIL$).

If $p$ is a pointer, then $p\rightarrow$ denotes the object that $p$ points to. Assigning to $p\rightarrow$ changes not $p$, but the object to which $p$ points. Examples:

> if $p$ is a pointer to integers, then $p\rightarrow = 5$ changes the variable pointed to by $p$ to 5.
>
> If $p$ is a pointer to $TwoD$, then $p\rightarrow.x = 5$ changes the x-element of the coordinate to 5.

Careful: if $p$ is NIL, then $p\rightarrow$ is undefined.

# Creating pointers

If $x$ is a variable of type $T$, then $\rightarrow x$ is of type "pointer to $T$". Its value is the object $x$.

If $T$ is a type, then the expression *new* $T$ will create a "fresh" object of type $T$ in memory (whose value is undefined). The type of the expression will be "pointer to $T$".

# Binary trees

In the following, we will consider searching in binary trees. Trees consist of nodes, which are labelled with some number and have at most two children.

For these purposes, we'll define a data type called *BinNode*. It is a record with three items:

*value*, an integer;

*left*, a pointer to *BinNode*;

*right*, another pointer to *BinNode*.

Example: Let *p* be a pointer to *BinNode*. We create a new node with value 5 without any children:

```
p = new BinNode;

p→.value = 5;

p→.left = NIL;

p→.right = NIL
```

Now, let *n* be some BinNode. Setting `n.left=p` will make *p*→ the left child node of *n*.

# Searching in binary trees

We define an algorithm for searching in binary trees. Its arguments are *p* (a pointer to *BinNode*) and *k* (an integer, the key).

We assume that the objects reachable from *p* (directly or transitively) form a finite binary tree consisting of *BinNode* objects (i.e., no cycles, no infinite paths). We assume that the tree is ordered, i.e. for every node *n* of the tree, the following holds: all nodes reachable via *n.left* (resp. *n.right*) have smaller (resp. bigger-or-equal) values than *n.value*.

If *k* is contained in the tree, our algorithm returns a pointer to a node whose value is *k*. Otherwise, it returns *NIL*.

```
procedure binary_tree_search

while p ≠ NIL do

  if p→.value = k then return p fi;

  if p→.value < k then p = p→.right else p = p→.left fi;

od;

return NIL
```

The worst-case running time of this algorithm is proportional to the longest path (to the “deepest” leaf) in the tree.

# Maintaining trees

How can we create such binary trees? Let us define an algorithm that does the following:

It takes two parameters, *p* and *k*, with the same assumptions as in the previous case, but this time *p* is an in-out parameter.

It extends the tree starting at *p* with another BinNode, whose value is *k*. This new node is a “leaf” in the tree, and it is inserted at the “correct” place in the tree, so that it remains ordered (in the aforementioned sense).

Let *q* be a pointer to pointer to BinNode.

```
procedure binary_tree_insert
q = →p;
while q→ ≠ NIL do
  if k >= q→→.value then q = q→→.right else q = q→→.left fi;
od;
q→ = new BinNode;
q→→.value = k;
q→→.left = NIL;
q→→.right = NIL
```

Note: The shape of the tree generated by the previous algorithm depends entirely on the *order* in which elements are inserted.

E.g., if the elements are inserted in ascending order, the tree will be “lopsided” (essentially, a list).

Such a lopsided tree will create bad running times for searching and inserting.

Ideally, we would want to create “perfectly balanced” trees, i.e. where the depth of a tree with $n$ nodes is $\lceil \log_2 n \rceil$.

However, balancing trees perfectly may be a costly operation. In the following, we will create an insertion operation that limits the amount of lopsidedness to create “reasonably” balanced trees with little overhead.

# AVL trees

An AVL tree is a binary tree with the following additional balancing property:

> Let $n$ be any node in the tree, and $\ell, r$ the height of its left and right subtrees (where the height of the tree is the length of its longest path, and the length of a path is the number of nodes in it). Then $\ell$ and $r$ differ by at most one.

For our purposes, we assume that AVL trees consist of records of type *BinHNode*, which are like the *BinNode* type, but with an additional integer item called *height*.

# Inserting into AVL trees

A procedure for inserting nodes into an AVL tree differs from the previous procedure (`binary_tree_insert`) in two respects:

- The procedure needs to update the height of the tree.
- Inserting a node may violate the balancing property, in which case the property must be restored.

We will tackle these two problems separately. It will be more practical do use a recursive procedure.

Our procedure for inserting into AVL trees takes two parameters, *p* (pointer to `BinHNode`, pointing to the tree root) and *k* (the key to be inserted). Both are *in* parameters. The procedure returns a pointer to the (possibly new) root of the tree.

# Insertion procedure (incomplete)

```
procedure avl_tree_insert
if p = NIL then
  p = new BinHNode;
  p→.value = k;  p→.height = 1;
  p→.left = NIL;  p→.right = NIL;
  return p
fi;
if k >= p→.value
  p→.right = avl_tree_insert(p→.right,k)
else
  p→.left = avl_tree_insert(p→.left,k)
fi;
```

Update the tree height (to be specified)

Re-balance if necessary (to be specified)

```
return p
```

# Updating tree height

Updating the tree height is simple: Measure the height of the left and right subtree and add one.

```
if p→.left = NIL then  lh = 0  else  lh = p→.left→.height fi;

if p→.right = NIL then  rh = 0  else  rh = p→.right→.height fi;

p→.height = 1 + max(lh,rh);
```

# Re-balancing a tree

Assume that the balance property holds in $p$ before we insert a new node into the tree starting at $p$. Then, if the balance property is violated afterwards, the height of the left and right subtrees differs by exactly two.

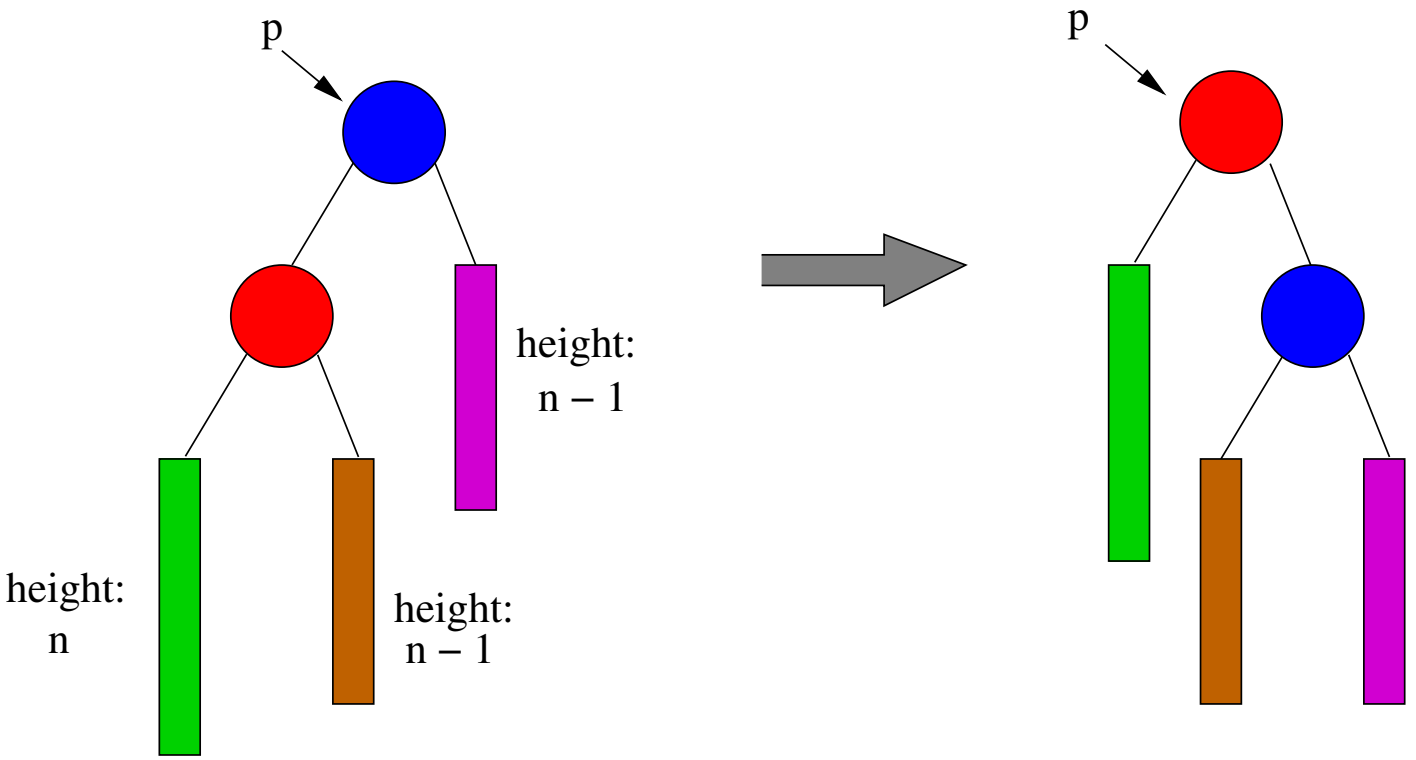
In the following, we assume that the height of the left subtree is bigger than that of the right subtree (the other case is analogous). There are two situtations:

Case 1: The “left-left” subtree is higher (by one) than the “left-right” subtree.

Case 2: The “left-right” subtree is higher (by one) than the “left-left” subtree.

# Case 1:

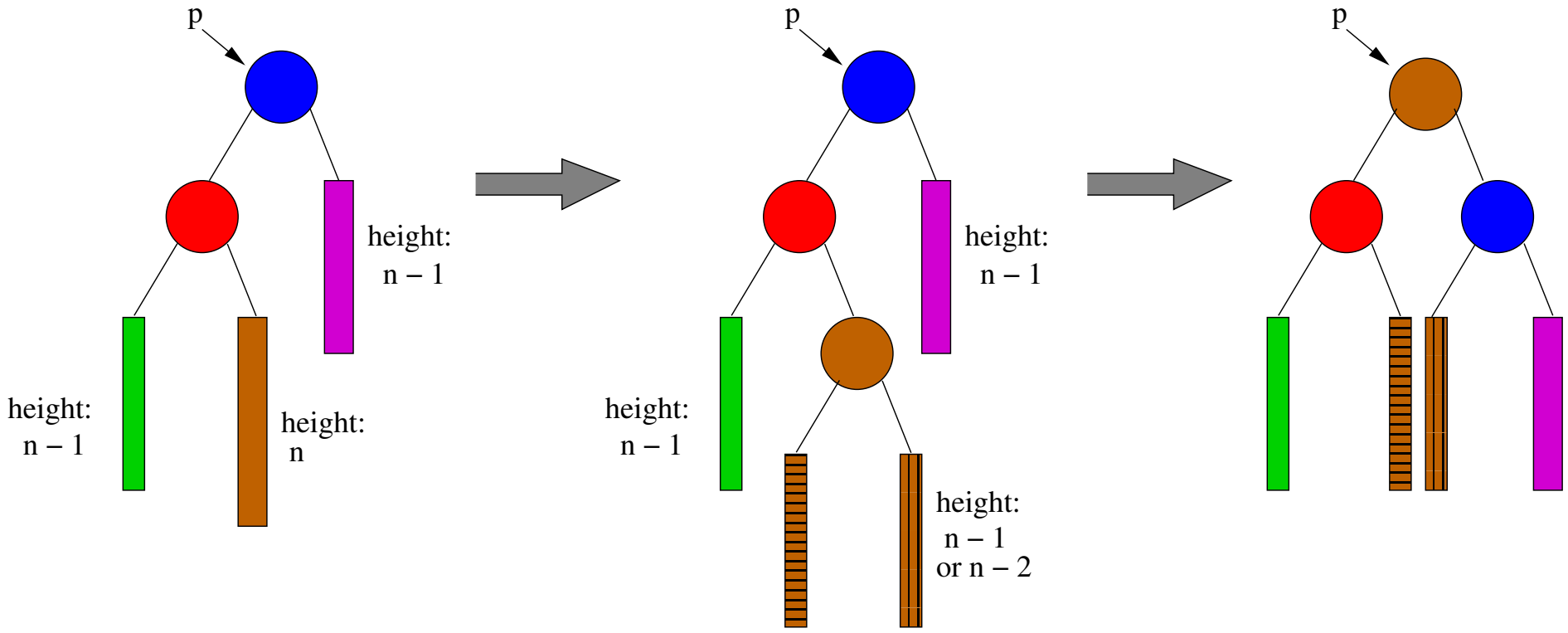
In Case 1, the situation can be redressed by making the red node the new root of the tree:




Notice that this change preserves the order of the tree.

# Case 2:

In Case 2, we make the brown node (the root of the left-right subtree) the new root. One of the brown subtrees is of height $n - 1$, the other of height $n - 2$.



# Algorithm for Case 1

```
blue = p;

red = p→.left;

brown = p→.left→.right;

red→.right = blue;

blue→.left = brown;

p = red
```

Also, the heights of red and blue need to be updated, as discussed before.

The algorithm for Case 2 is analogous.

# Height of AVL trees

Consider an AVL tree of height $n$. We shall prove that the height of such a tree is $\Theta(\log n)$. (From this it follows that searching and inserting into an AVL tree with $n$ nodes takes $\Theta(\log n)$ times.)

Minimal height of AVL trees:

> Consider an AVL tree of height $k$. Then the tree contains at most $2^k - 1$ nodes.
>
> Proof: easy, by induction.
>
> Therefore, an AVL tree with $n$ nodes has at least a height proportional to $\log n$.

The maximal height of AVL trees is related to the Fibonacci numbers, defined as $F_0 := 0$, $F_1 := 1$, and $F_{k+2} := F_k + F_{k+1}$ for $k \geq 0$.

Consider an AVL tree with height $k$. We prove (by induction) that it contains at least $h_k := F_{k+2} - 1$ nodes.

It is immediate to see that $h_1 = 1$ and $h_2 = 2$.

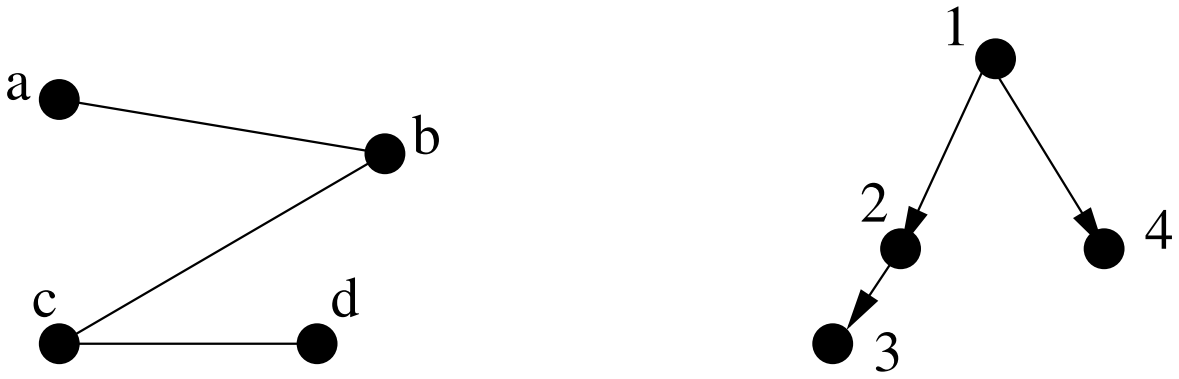
For $k \geq 3$, we have that one of the subtrees is at least of height $k - 1$, the other of height at least $k - 2$. Therefore, $h_k = h_{k-2} + h_{k-1} + 1 = F_{k+2} - 1$.

It is known that $F_k$ is proportional to some constant to the power of $k$.

# Part 4: Graphs

# Graphs

A *graph* is a collection of nodes (also called “vertices”) together with a collection of edges connecting the nodes.




Graphs can be either directed or undirected, meaning that the edges have (or do not have) a “direction”. In the figure above, the left is an *undirected* graph with the set of nodes $\{a, b, c, d\}$, the right is a directed graph with nodes $\{1, 2, 3, 4\}$.

Undirected edges can be understood as a connection that goes “either way”, whereas a directed edge means that one can go from one node to the other but not vice versa.

# Mathematical notation

Mathematically, a graph is a tuple $(V, E)$, where $V$ is the set of nodes (vertices), whereas $E$ is the set of edges.

In *undirected* graphs, the nodes touched by an edge have the same "priority", therefore we denote edges by $\{u, v\}$, where $u, v$ are nodes. (Notice that $\{u, v\} = \{v, u\}$.) In the example, the set of edges is $\Big\{\{a, b\}, \{b, c\}, \{c, d\}\Big\}$.

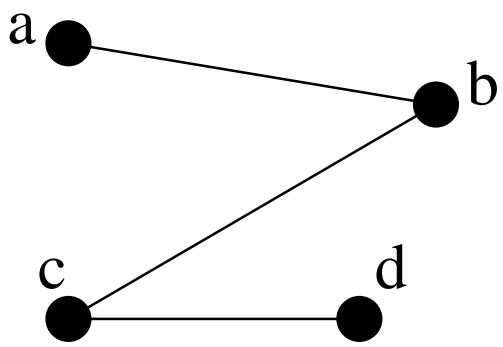
In *directed* graphs, the direction matters, therefore we denote edges as tuples, with the source node first, e.g. $(u, v)\}$. In the example, the set of edges is $\Big\{(1, 2), (2, 3), (1, 4)\Big\}$.

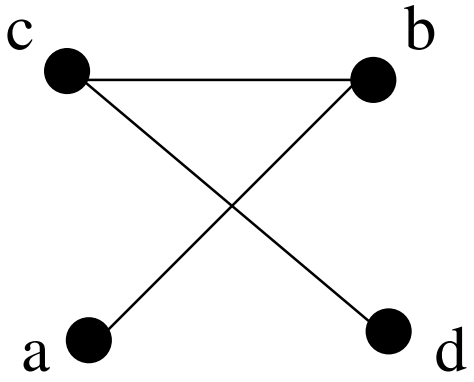
Note: Binary trees are directed graphs.

# Mathematical notation vs graphical display

In the mathematical notation, it only matters which nodes are there and which of them are connected.

A graphical notation adds *topological* information, i.e. where nodes are located. However, we consider this information “unimportant”, i.e. just for convenient visualization. The following two graphs are considered the same:



# Adjacency and incidence

Let $G = (V, E)$ be an undirected graph.

We call nodes $u, v \in V$ adjacent iff $\{u, v\} \in E$.

We call nodes $u \in V$ and edge $e \in E$ incident iff $u \in E$.

We call edges $e, f \in E$ adjacent iff $e \cap f \neq \emptyset$.

We call an edge $e \in E$ a loop iff $e = \{u\}$ for some node $u$.

# Neighbourhood

Let $G = (V, E)$ be an undirected graph *without loops*.

The neighbourhood of node $v \in V$ is the set of adjacent nodes:
$N(v) := \{ u \mid \{u, v\} \in E \}$.

$|N(v)|$ is called the degree of $v$, written $deg(v)$.

In the previous example, the degree of $a, d$ is 1, that of $b, c$ is 2.

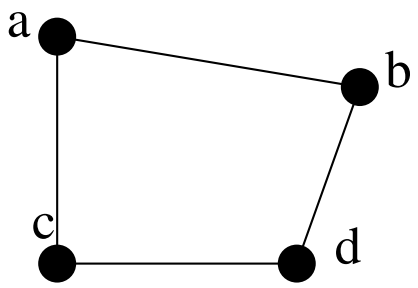
Fact: It holds that $\sum_{v \in V} deg(v) = 2 \cdot |E|$.

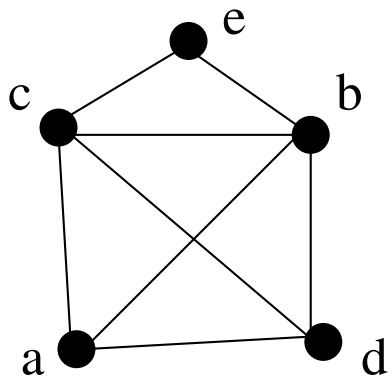
# Euler paths and circles

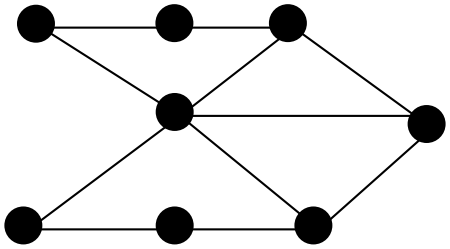
Let $G = (V, E)$ be an undirected graph without loops.

An Euler path is a sequence of adjacent edges containing each edge in $E$ exactly once. An Euler circle is an Euler path starting and ending at the same node.

If a graph has an Euler path, it can be drawn "in one stroke". The left figure below has an Euler circle, the one in the middle an Euler path, but not an Euler circle, and the one on the right does not have an Euler path.



It can be shown that the existence of Euler paths and circles is tied to the distribution of degrees of the nodes.

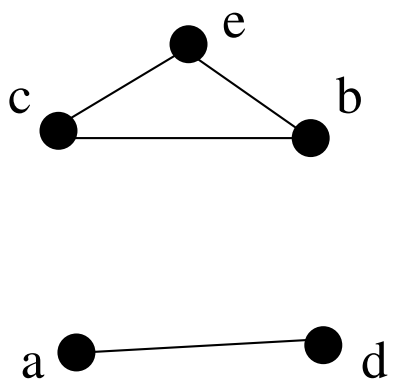
Suppose that a graph has an Euler circle. Then every node must have even degree (it is touched an even number of times).

Suppose that a graph has an Euler path, but not a circle. Then exactly two nodes have odd degrees (the start and the end node) and all others have even degrees.

Suppose that a graph has more than two nodes with odd degrees. Then no Euler path or circle can exist.

# Connectedness

An undirected graph is called connected if from every node one can reach every other node via zero or more edges. All graphs shown so far were connected. The one below is not, it is said to have two components.



# Finding Euler circles

Let $G = (V, E)$ be a connected undirected graph without loops where every node has even degree.

We show how to construct an Euler circle in $G$.

> Start at an arbitrary node $v$ and select arbitrary adjacent edges until you come back to $v$. The result is a circle (which does not necessarily contain all edges).
>
> Remove the selected edges from the graph. In the resulting graph, every node still has even degree (why?). Therefore, select some node $u$ along the selected circle with non-zero degree and find a new circle using the previous step. “Insert” this circle into the first circle and repeat.

# Data structures for graphs

In the following, we concern ourselves with algorithms for *directed* graphs.

For simplicity, we shall assume that graphs have nodes $\{1, \ldots, n\}$, for some $n$.

An edge can be represented by a record type with items `src`, `dst`. An edge $(u, v)$ is represented by a record where the `src` item is $u$ and the `dst` item is $v$.

For each node $u$, we will often require its list of neighbours, i.e. the nodes $v$ such that $(u, v)$ is an edge. For this, we extend the record with another item called `next`, a pointer to another edge. The resulting record type is called `Edge`.

If we have a directed graph with $n$ nodes, then we assume the existence of an array $E$ with indices $1 \ldots n$ whose entries are pointers to type *Edge*.

Then the neighbours of node $i$ can be traversed as follows:

```
p = E[i];
while p ≠ NIL do
  j = p→.dst;
  ...
  p = p→.next
od
```

Here, `p` is assumed to be a pointer to type `Edge`. In every iteration, $j$ is a node such that $(i, j)$ is an edge.

In the following, we shall abstract from this notation an simply write `for j in N(i) do ... od` as an abbreviation.

# Traversing graphs

A traversal of a graph is (informally) a procedure that visits all nodes of a graph once.

Two well-known traversal strategies are:

breadth first: visit some node, then visit all of its neighbours, after that the neighbours of the neighbours and so forth.
Applications: finding shortest paths, ...

depth first: visit some node, then visit *one* of its neighbours, one of the neighbours of that neighbour etc.
Applications: topological sorting, finding components, ...

# Depth-first search

Assume that we have a directed graph with $n$ nodes, named $1 \ldots n$.

We discuss two procedures, called `dfs_pre` and `dfs_post`. Both takes the graph and perform a depth-first search on it.

All nodes receive a number. The numbers are written into an array `dfsnum`, i.e. `dfsnum[i]` containes the number of the node $i$. We assume that initially `dfsnum` is filled with zeros.

If the number is given to a node before the neighbours are considered, we call this pre-order numbering. If the number is given after the neighbours are considered, we call it post-order numbering.

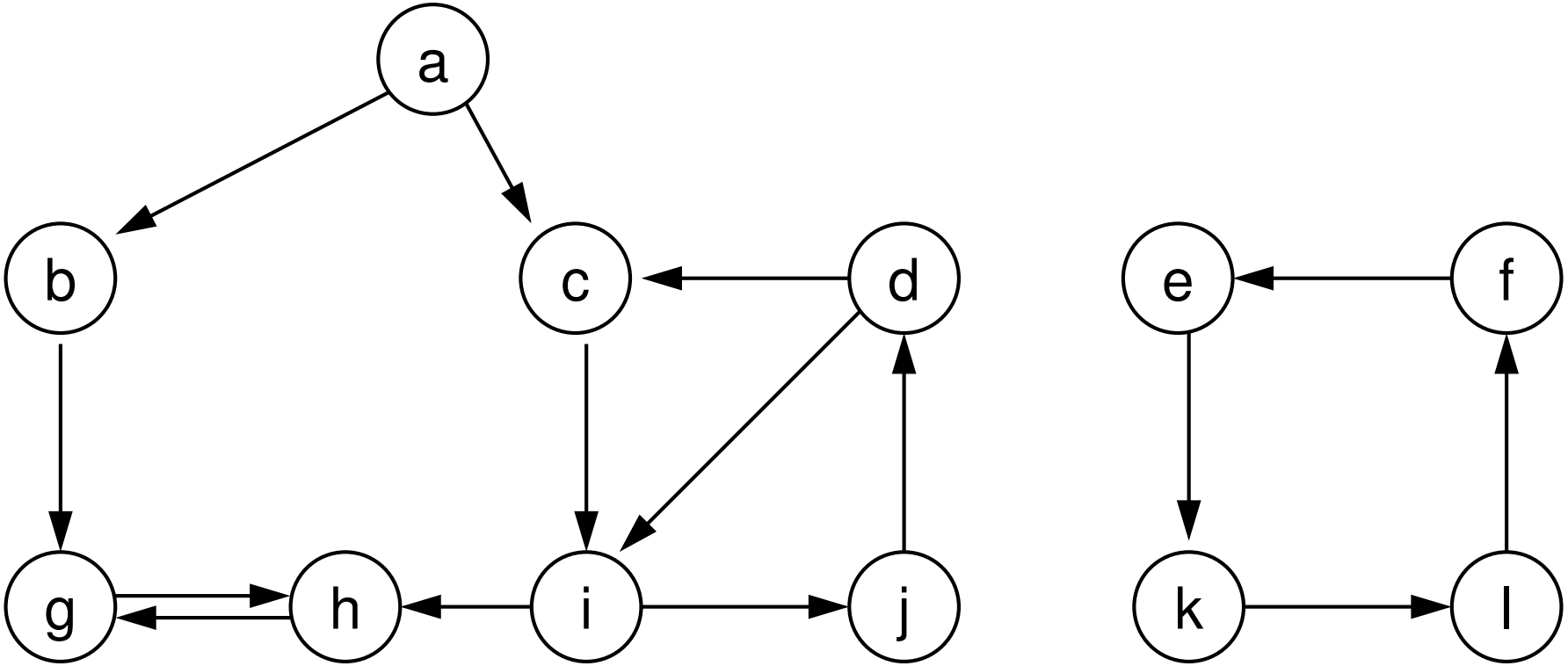
We make use of an auxiliary procedure, which takes an in-parameter `i`, and an in-out parameter `counter`.

# Algorithm for pre-order numbering

```
procedure dfs_pre
counter = 0;
for i = 1 to n do
  if dfsnum[i] = 0 then dfs_pre_aux(i,counter) fi
od
```

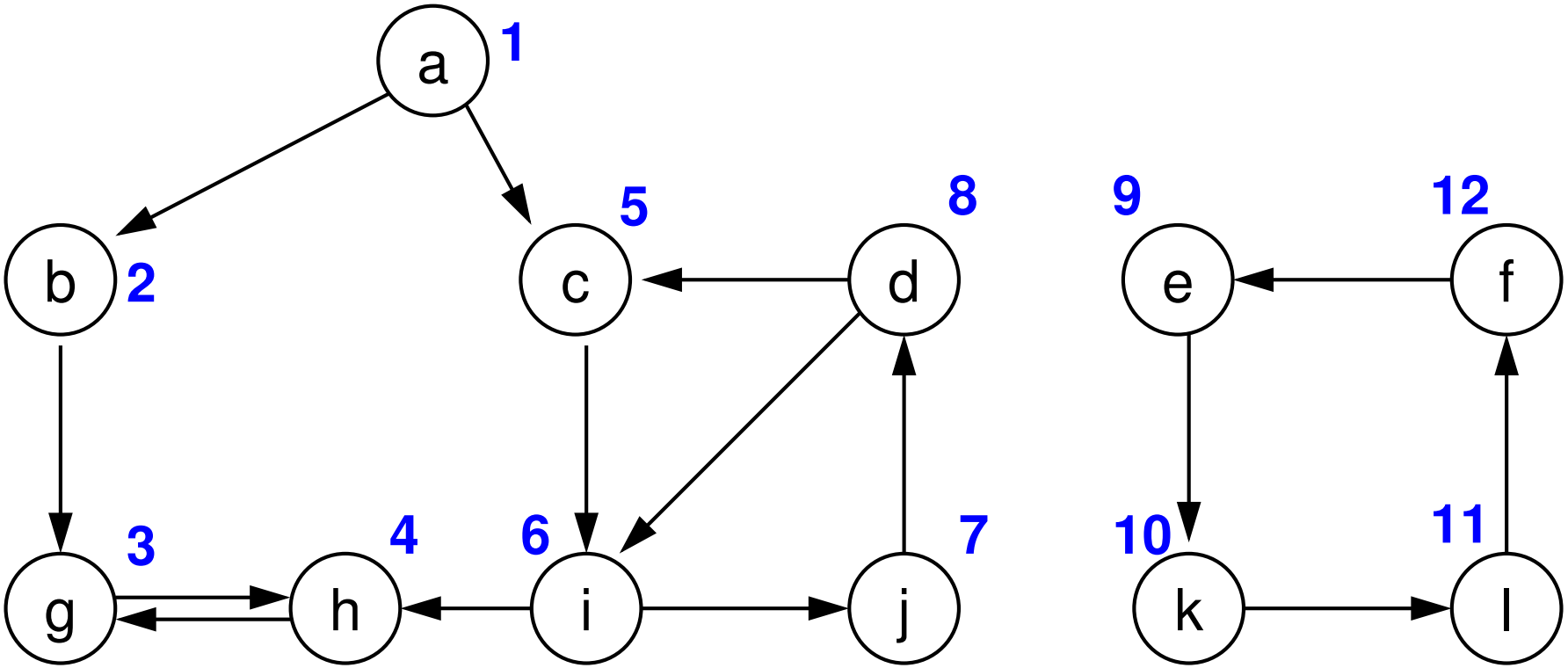
```
procedure dfs_pre_aux
counter = counter + 1;
dfsnum[i] = counter;
for j in N(i) do
  if dfsnum[j] = 0 then dfs_pre_aux(j,counter) fi
od
```

# Running example



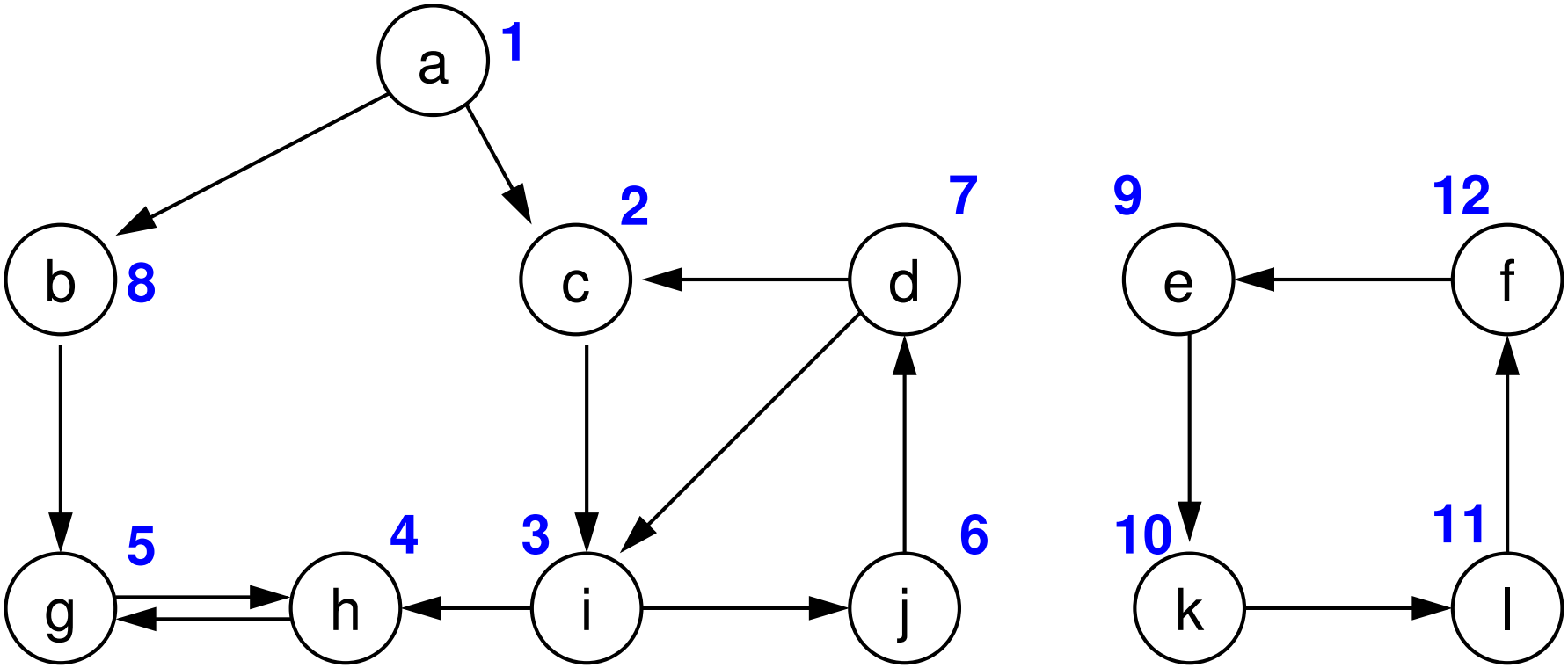

Here's an example of a directed graph.
Note: Nodes labelled with letters *a* . . . *l* instead of numbers 1 . . . 12 (for better distinction from the dfs numbers, shown in the next slide).

# Pre-order numbering (1)



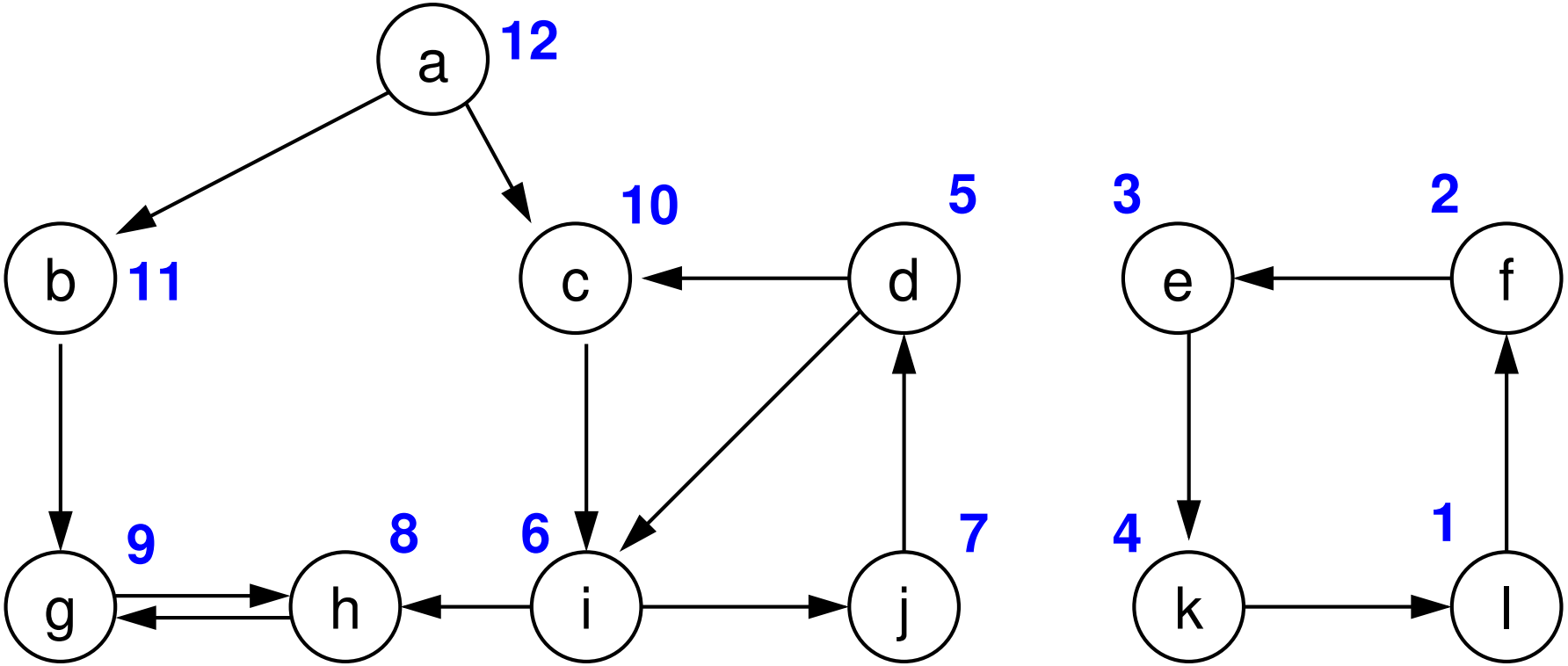

The blue numbers are the contents of `dfsnum` array after the `dfs` procedure.
Note: The algorithm is not completely deterministic if we assume that `for j in N(i)` can visit neighbours in any order.

# Pre-order numbering (2)




Another possible numbering if neighbours are visited in different order.

# Pre-order numbering (3)



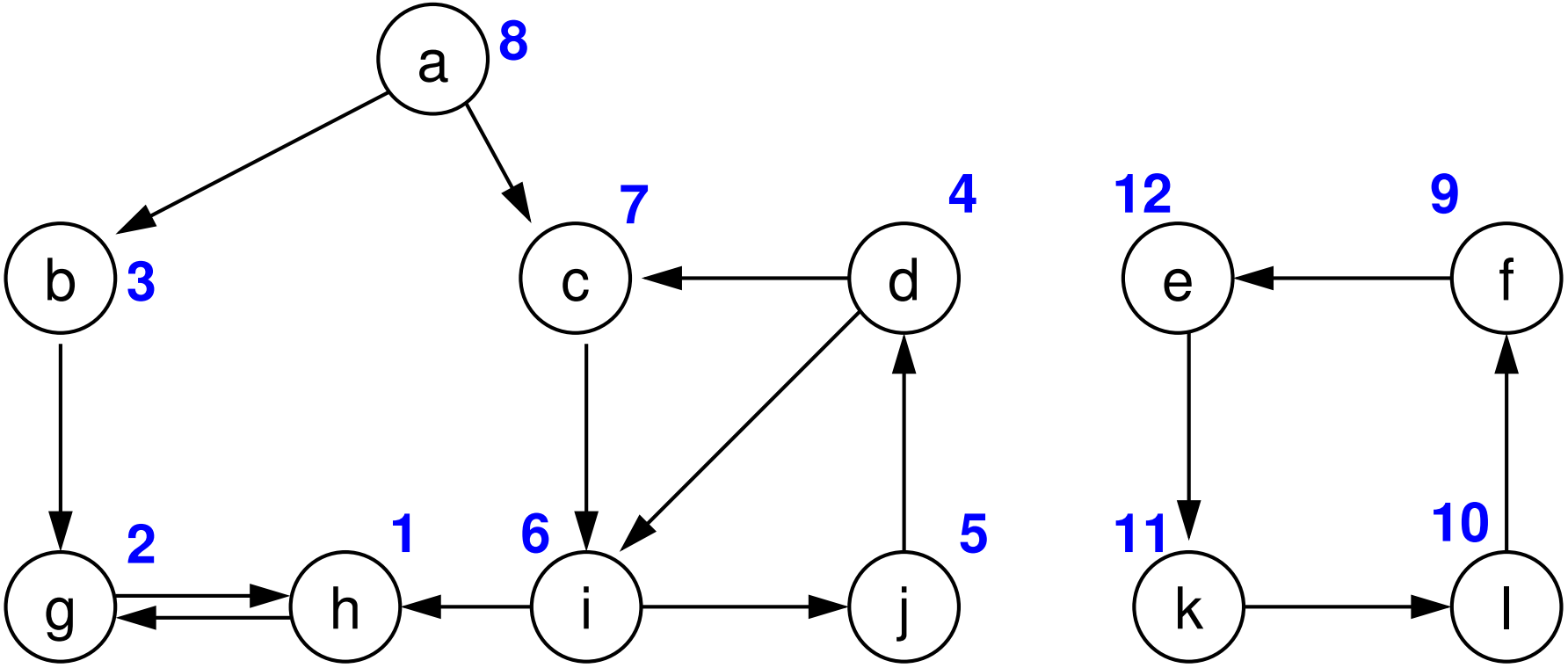

Suppose that the `for i...` loop also visits nodes in any order (rather than strictly from 1...n, or alphabetically in this case). Then the above numbering is also possible.

# Algorithm for post-order numbering

```
procedure dfs_post
counter = 0;
for i = 1 to n do
  if dfsnum[i] = 0 then dfs_aux(i,counter) fi
od
```

```
procedure dfs_post_aux
dfsnum[i] = -1;
for j in N(i) do
  if dfsnum[j] = 0 then dfs_aux(j,counter) fi
od
counter = counter + 1;
dfsnum[i] = counter;
```

# Post-order numbering




Post-order numbering if nodes are visited in the same order as in the first pre-order numbering.

# Applications of depth-first search

We will consider two applications of depth-first search.

- topological sorting
- strongly-connected components

Both make use of post-order numbering.

# Topological sorting

Abstractly speaking, topological sorting is the task of refining a partial order into a total order.

Example: Imagine a “to-do list” with several items.

buy drinks

buy a mop

buy a new pen

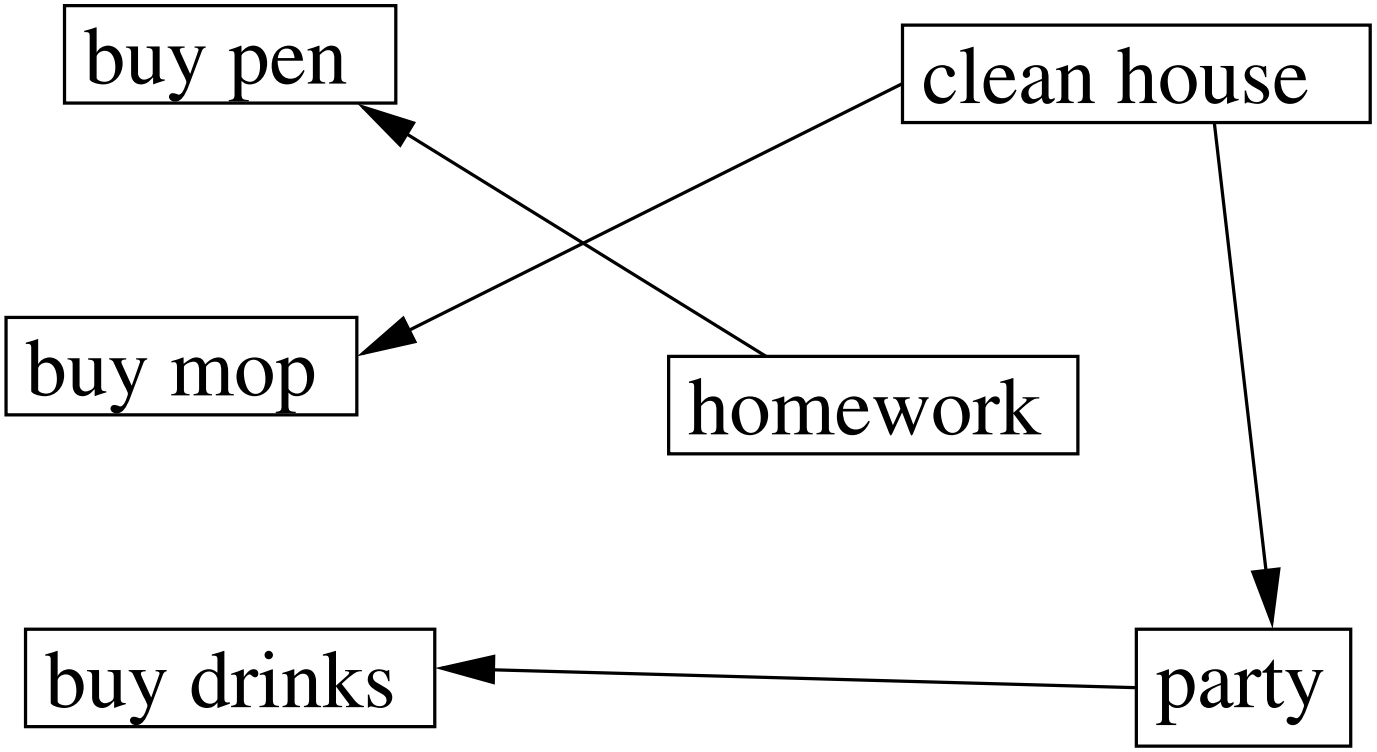
give a party

clean the house

do homework

There are dependencies between some of these items, for instance one would buy drinks *before* giving a party.

# Dependency graph




Dependencies expressed as a graph, where an arrow from item A to item B means that B must be done before A. Note that there are no cyclic dependencies!

# Acyclic graphs and partial orders

A directed graph is called acyclic if for any two different nodes $v$, $w$ we have the following: if $v$ is reachable from $w$, then $w$ is not reachable from $v$.

Example: see previous slide

Let $M$ be a set. A partial order $\prec$ on $M$ is a binary relation between elements of $M$ with the following properties: for all $a, b, c \in M$:

- $a \prec a$ (reflexivity);
- $a \prec b$ implies $b \not\prec a$ (anti-symmetry);
- $a \prec b$ and $b \prec c$ imply $a \prec c$ (transitivity).

However, a partial order is not necessarily total, i.e., it may contain incomparable elements, that is, pairs $a, b$ with neither $a \prec b$ nor $b \prec a$.

Example: For any set $S$, the subset relation is a partial order on the powerset of $S$.

The reachability relation on acyclic graphs is a partial order between the nodes, given by $v \prec w$ if $v$ is reachable from $w$.

Likewise, every partial order corresponds to an acyclic graph.

Thus, we can equate partial orders and acyclic graphs.

Let $G$ be a directed graph. We denote by $G_R$ the graph obtained by reversing the direction of all edges in $G$.

If $G$ is acyclic, then $G_R$ is also acyclic, and corresponds to the reverse partial order.

# Post-order numbering on acyclic graphs

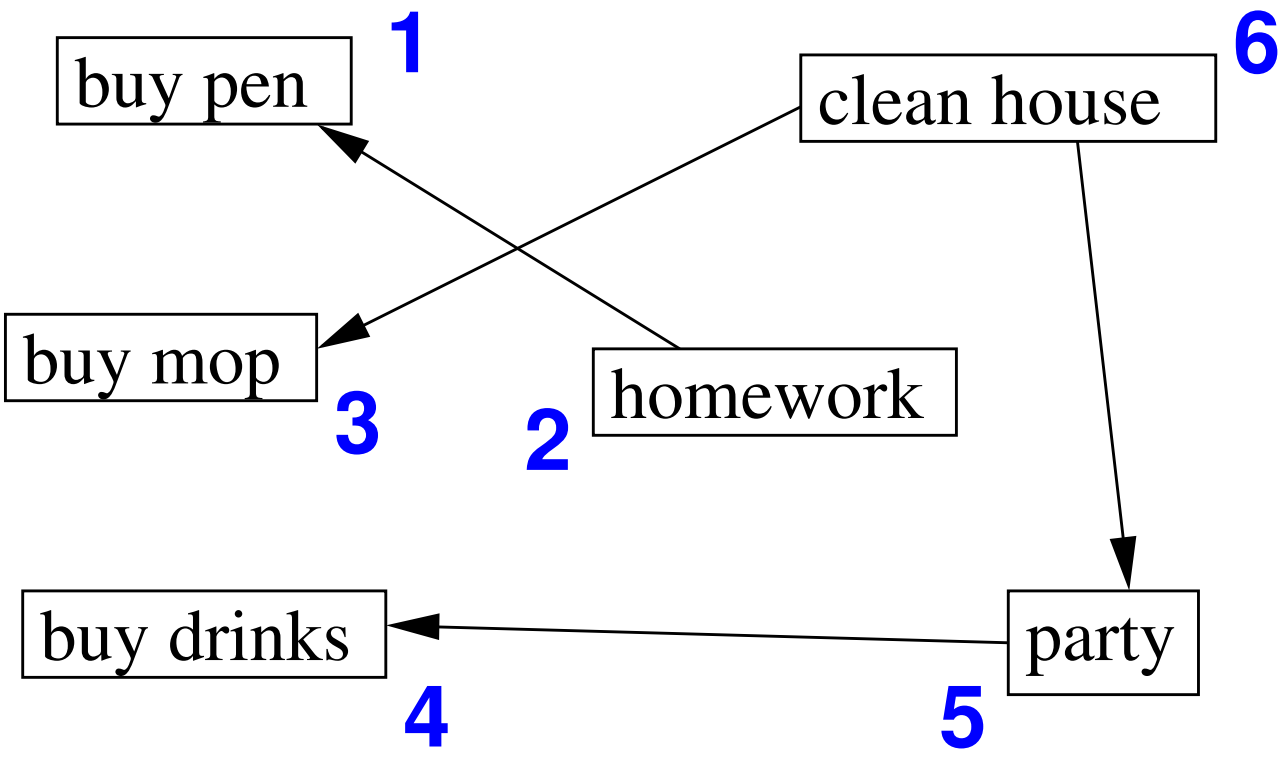
Any post-order numbering on a directed graph has the following property:

> If $v$ is reachable from $w$ but not vice versa, then the dfs number of $v$ will be smaller than the one of $w$.

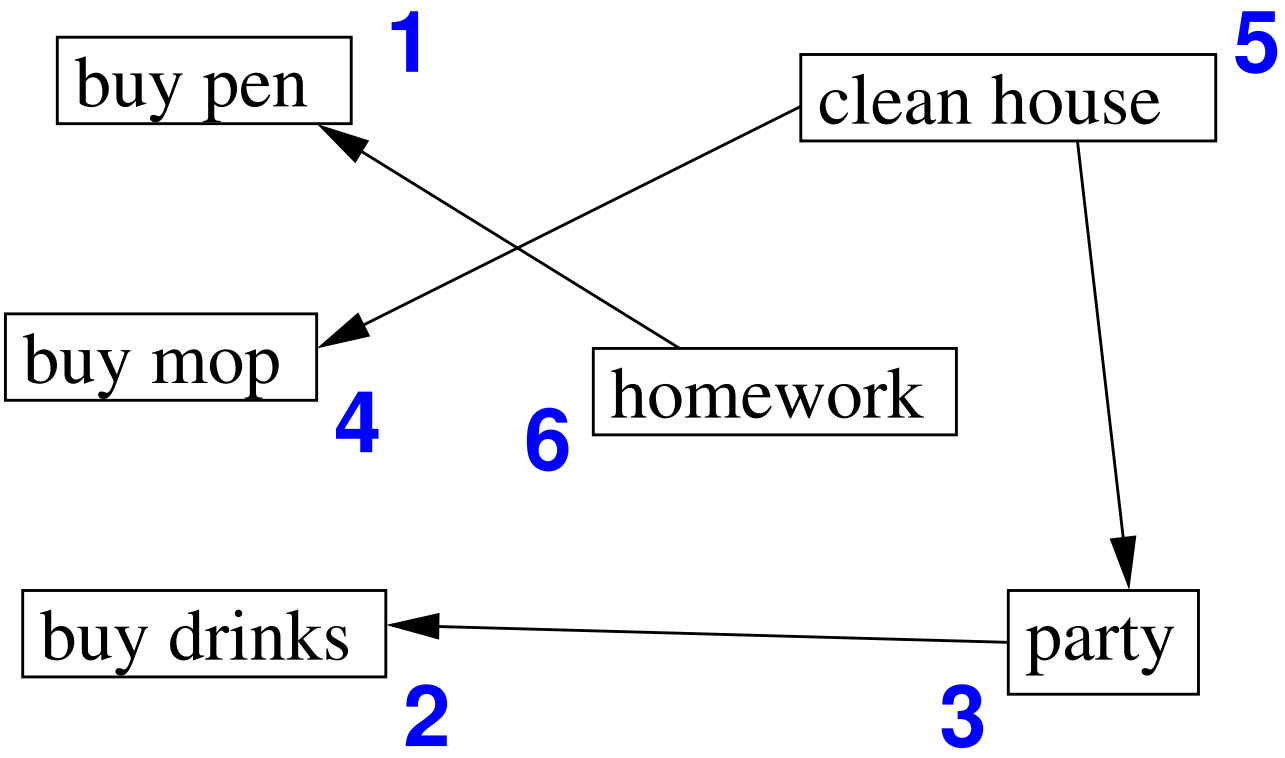
Proof: Either `dfs_post_aux(v)` is called before `dfs_post_aux(w)`. Then, since $w$ is not reachable from $v$, `dfs_post_aux(v)` will terminate and assign a number to $v$ before considering $w$. Or, in the other case, since $v$ is reachable from $w$, the recursive calls will reach $v$ (and assign a number to it) before returning to $w$.

In other words, post-order numbering provides a total order that refines the partial order associated with the graph, i.e. $v \prec w$ implies $dfsnum[v] < dfsnum[w]$ (but not vice versa).

# To-do list revisited




A possible post-order numbering of the to-do-list graph. Doing things in the given order will ensure that all dependencies are resolved correctly.




Another possible post-order numbering.

# Strongly connected components

Let $G = (V, E)$ be a directed graph (not necessarily acyclic). A set of nodes $C \subseteq V$ is called a strongly connected component (SCC) of $G$ if

(1) for all nodes $v, w \in C$, it holds that $v$ is reachable from $w$ (and vice versa);

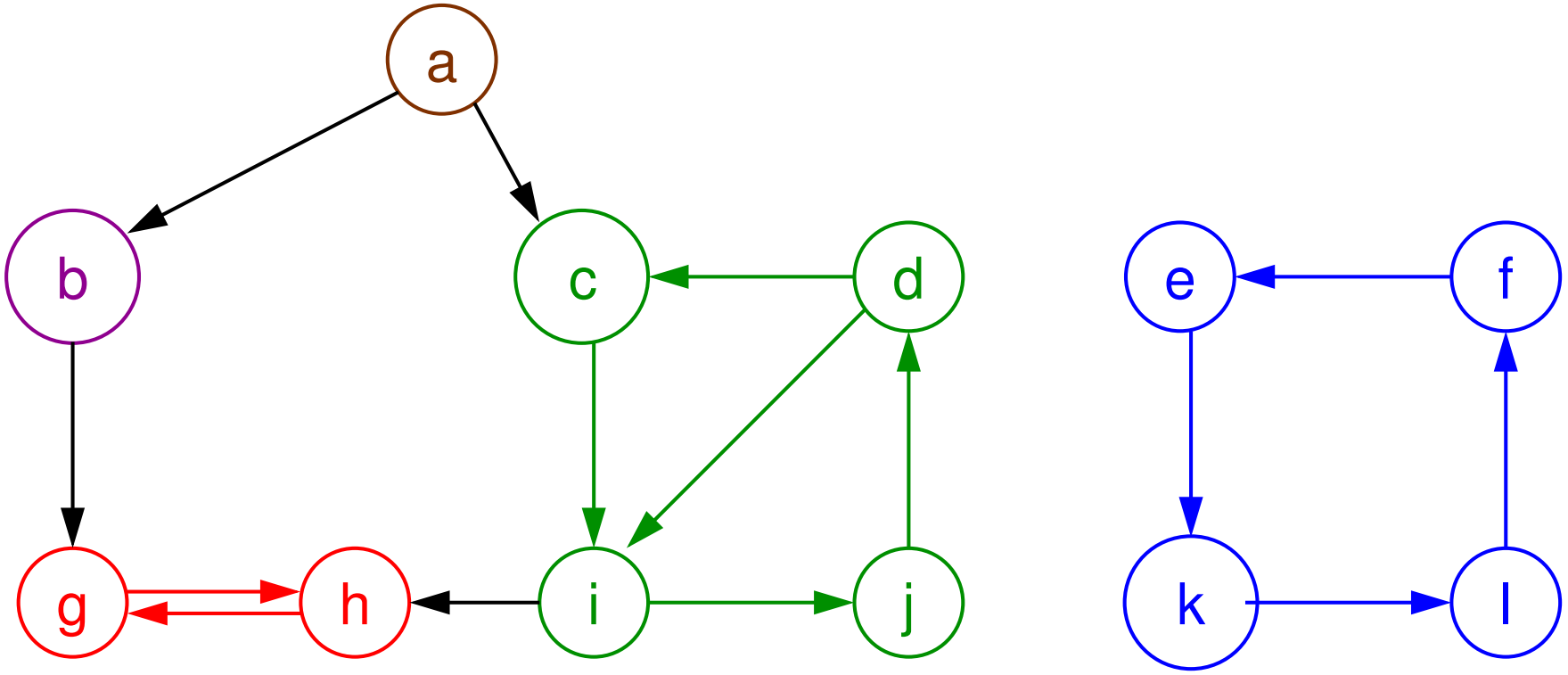
(2) $C$ is *maximal* w.r.t. (1), i.e. no further node can be added to $C$ without violating property (1).

Note: SCCs are a partioning of the nodes, i.e. each node belongs to exactly one SCC. The graph $G_R$ has the same SCCs as $G$.

The relation given by $v \sim w$ if $v, w$ are in the same SCC is an equivalence relation (reflexive, symmetric, transitive).

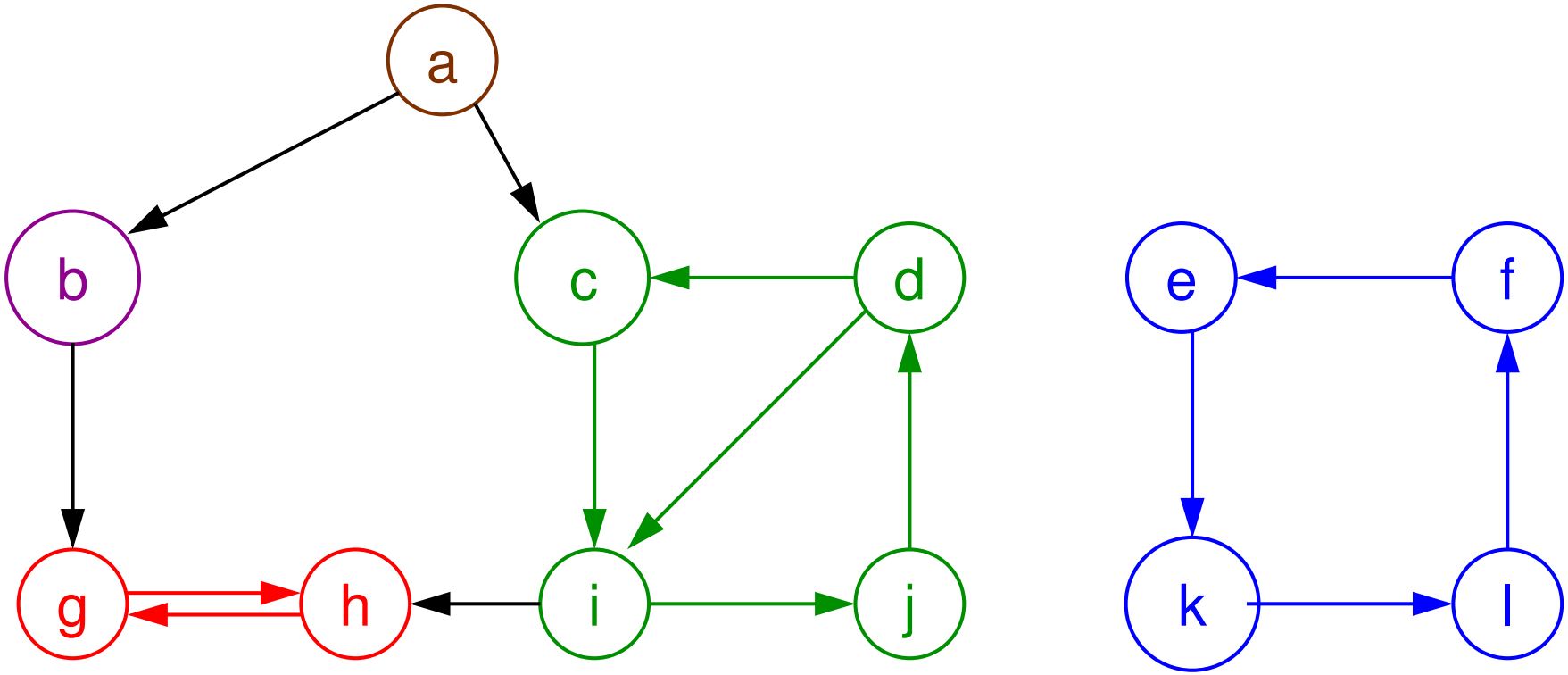
The relation between SCCs given by $C_1 \prec C_2$ if nodes in $C_1$ are reachable from nodes in $C_2$ is a partial order.

# SCCs: Example




In this example, SCCs are indicated by colours. Nodes in the same SCC have the same colour. Nodes *a* and *b* form an SCC by themselves.

# SCCs: Example




Partial ordering of SCCs: *red* $\prec$ *green*, *green* $\prec$ *brown* etc.

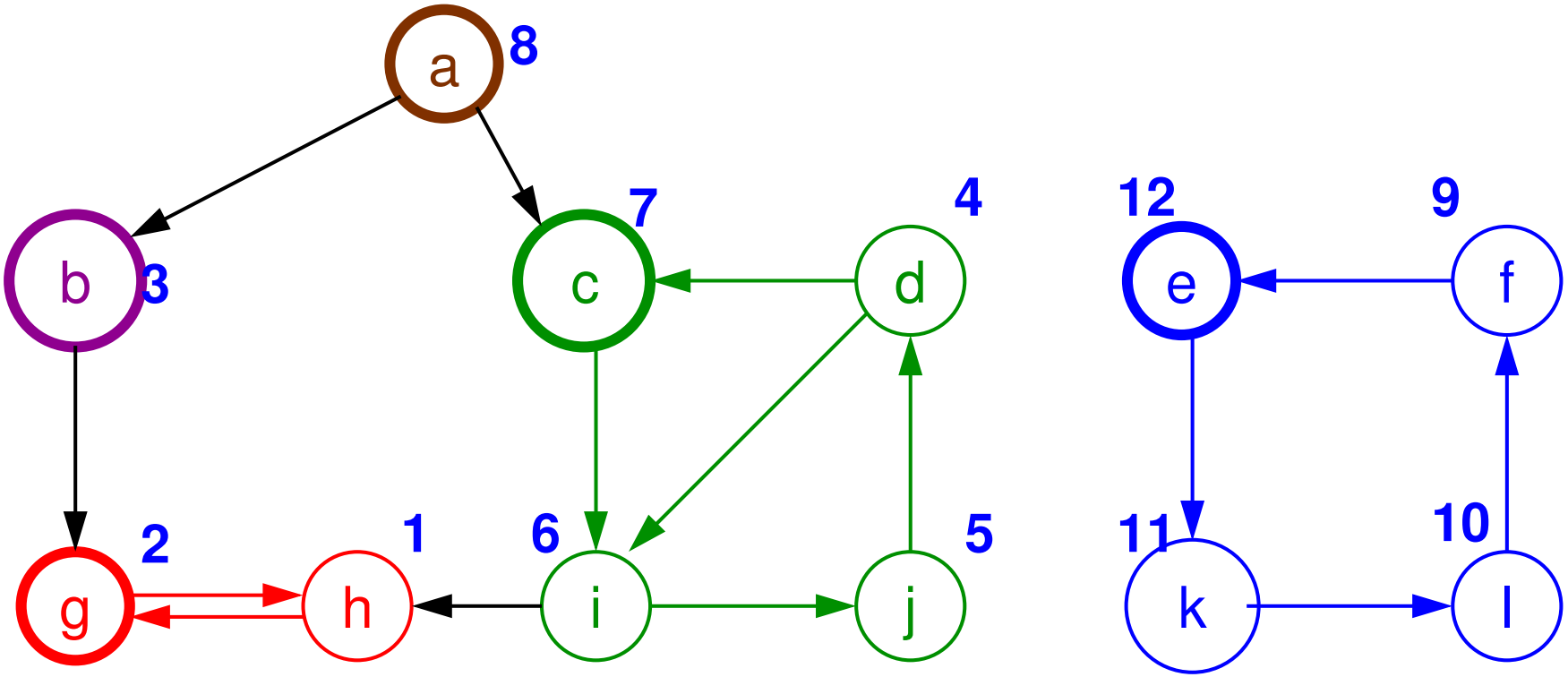
# Roots

For a given *post-order* numbering, the root of an SCC is the node with the *largest* number within the SCC.

This means that the root of an SCC is the node that is visited first (and left last) by the dfs procedure.

Let $C_1, C_2$ be two SCCs such that $C_1 \prec C_2$. Then the root of $C_2$ has a bigger number than the one of $C_1$.

# Post-order numbering




Same post-order numbering as before, roots indicated by thick lines.

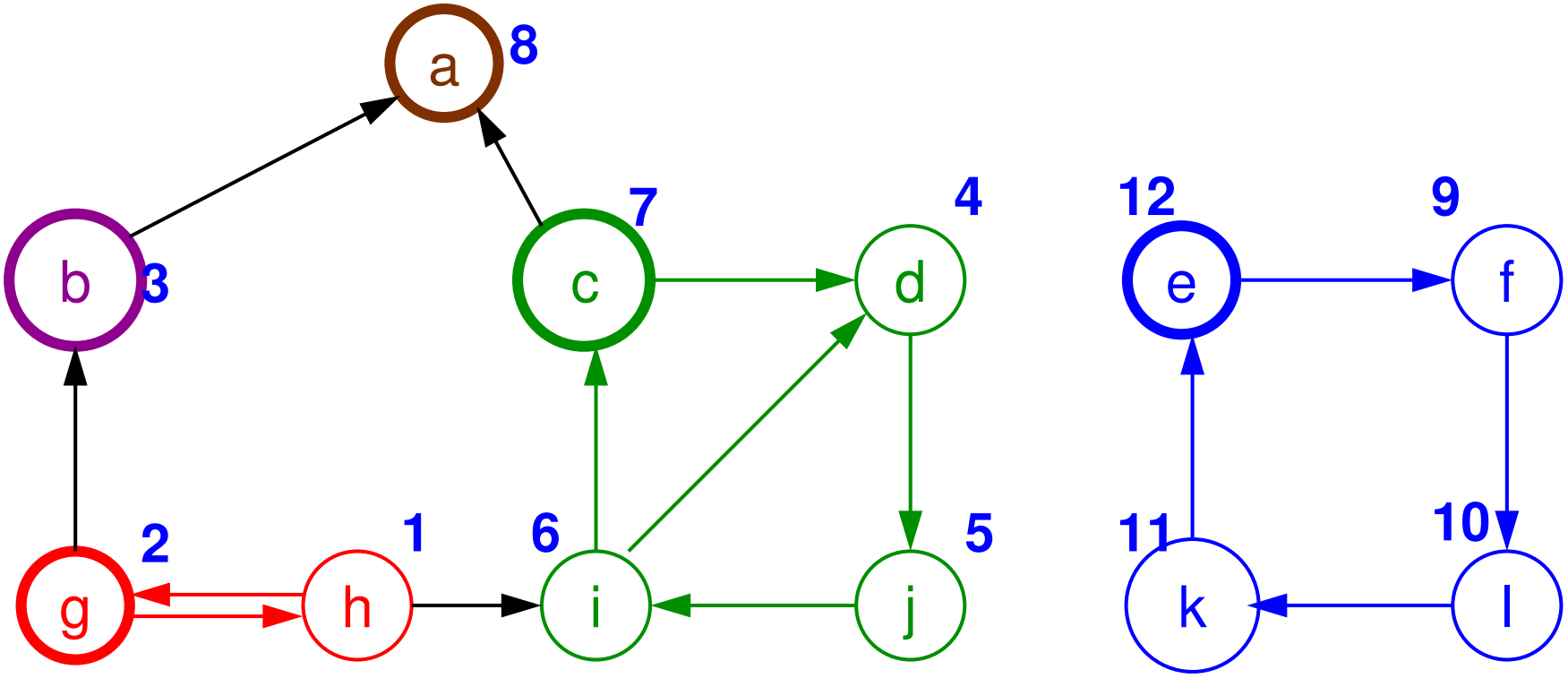
# Algorithm for finding SCCs

Let $v$, $w$ be two nodes of a graph $G$. If $v$ can be reached from $w$ in both $G$ and $G_R$, then $v$ and $w$ are in the same SCC.

Using this, we now give an algorithm for identifying all SCCs of a given graph $G$. It consists of the following steps:

> Perform any post-order numbering on $G$, sort the nodes in *descending* order, and generate the graph $G_R$.
>
> While the graph $G_R$ still contains nodes, do the following:
>
> > Find the node $r$ with the highest number still in $G_R$. (This must be a root!)
> >
> > Find all nodes reachable from $r$ in $G_R$ (using depth-first or breadth-first search). Remove all these nodes from $G_R$. All removed nodes are one SCC.

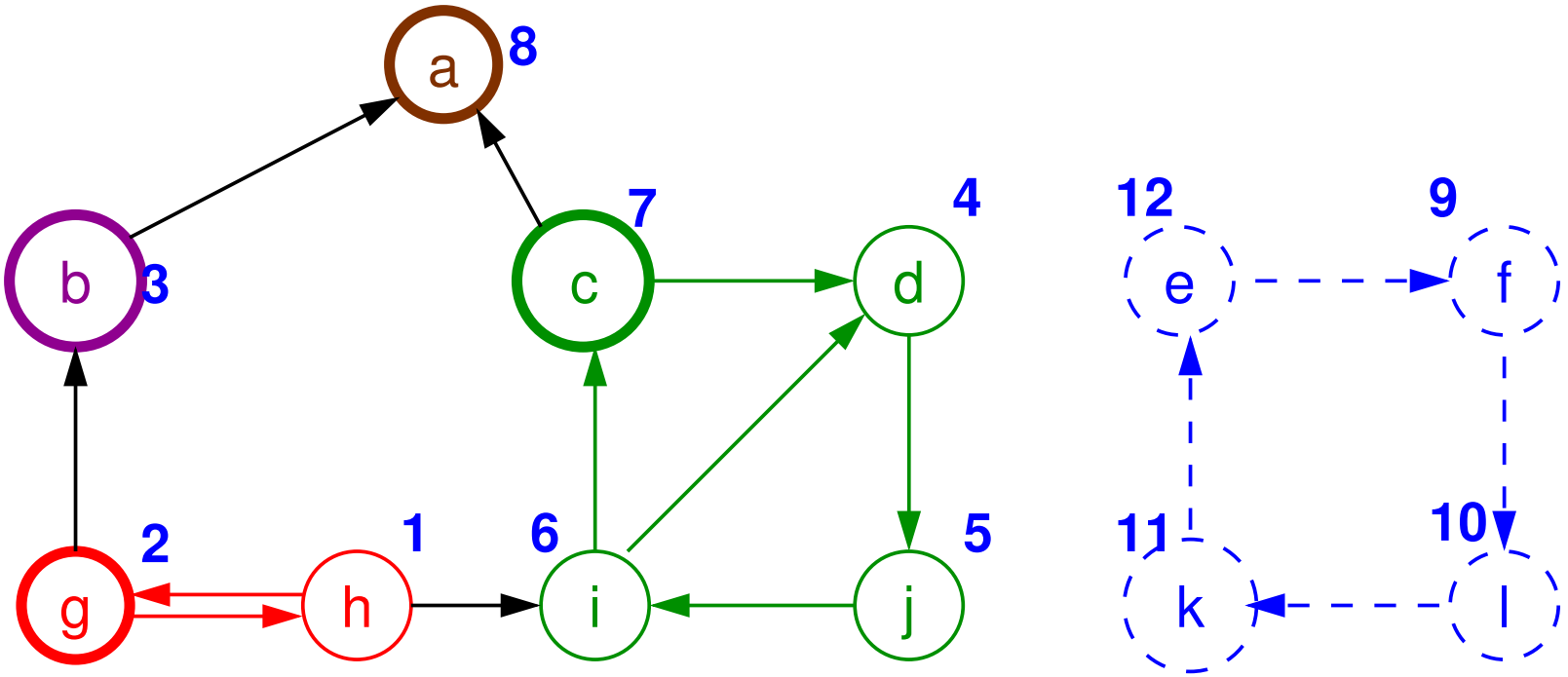
# Example




The graph $G_R$ with a post-order numbering of $G$.

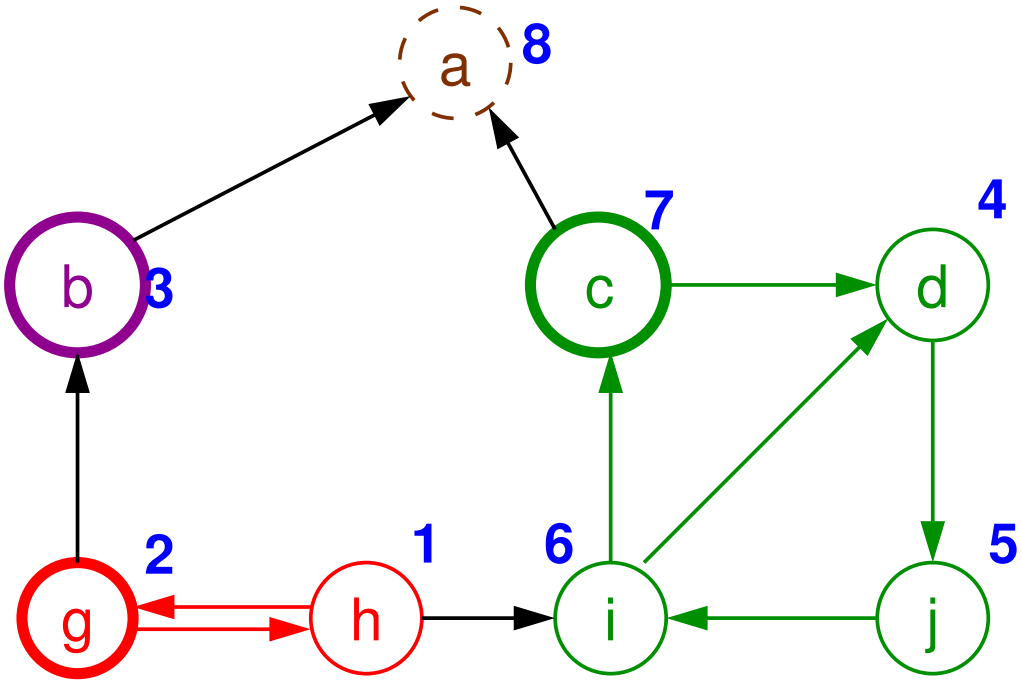
Root nodes will be considered in the order $e$, $a$, $c$, $b$, $g$.

# Running the algorithm (1)



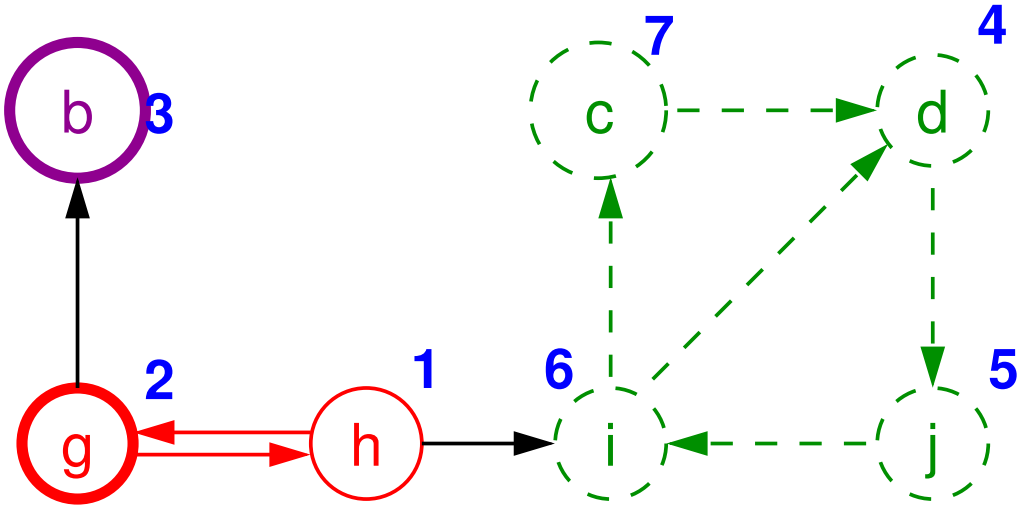

Finding and removing the nodes reachable from *e* identifies the blue SCC.

# Running the algorithm (2)



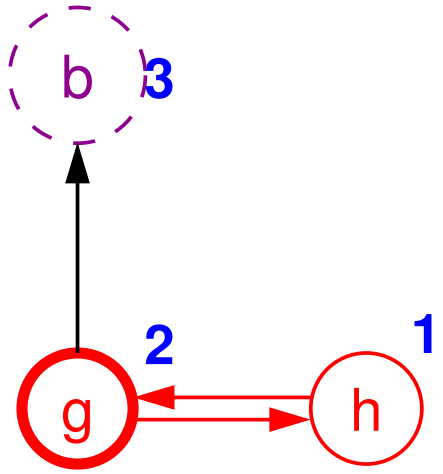

From *a* no other node is reachable, so *a* is an SCC unto itself.

# Running the algorithm (3)



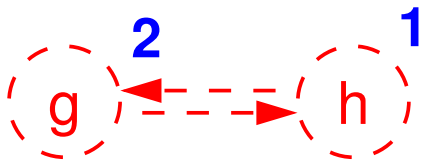

The green SCC is found and removed next, starting from *c*.

# Running the algorithm (4)




*b* is again an SCC in its own right.

# Running the algorithm (5)




Finally, only the red SCC is left over.

# Correctness and complexity

The root with the largest number belongs to a maximal SCC (w.r.t. $\prec$) in $G$, and thus a minimal SCC of $G_R$.

Remember that a minimal SCC is one that can be reached from other SCCs but cannot reach any others. Therefore, the nodes reachable in $G_R$ from its root all belong to that SCC.

After removing the SCC, the next-largest number belongs to an SCC that is now minimal w.r.t. the reduced version of $G_R$, and so forth.

Complexity: Every node and every edge are touched exactly twice by the algorithm, i.e. its run-time is $\mathcal{O}(|V| + |E|)$.

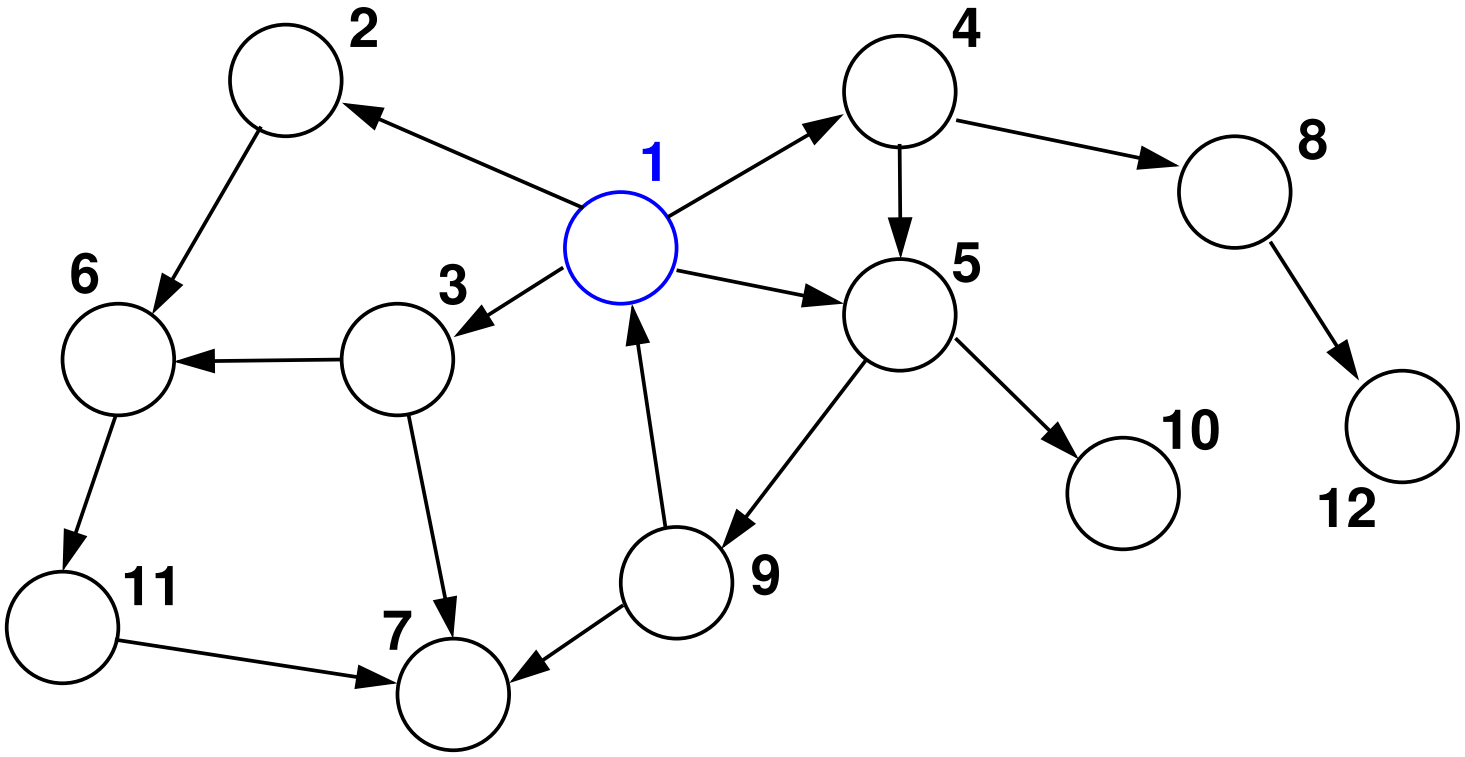
# Breadth-first search

Recall the principle of breadth-first search (BFS):

> visit some node, then visit all of its neighbours, after that the neighbours of the neighbours and so forth.

$n$

# Example




Here's an example of a BFS numbering, starting at the blue node.
We shall see shortly how this numbering was created.

# Breadth-first search

Recall the principle of breadth-first search (BFS):

> visit some node, then visit all of its neighbours, after that the neighbours of the neighbours and so forth.

In the following, we discuss a simple algorithm that performs a BFS on a directed graph and orders nodes in the order it visits them. The following assumptions are made by the algorithm:

> The graph has $n$ nodes, numbered $1 \ldots n$.
>
> The search procedure starts at node $1$.
>
> All nodes are reachable from node $1$.

# Algorithm for BFS

```
procedure bfs
for i = 1 to n do bfsnum[i] = 0 od;
bfsnum[1] = 1; order[1] = 1;
visited = 0; counter = 1;
while visited < counter do
  visited = visited + 1;
  i = order[visited];
  for j in N(i) do
    if bfsnum[j] = 0 then
      counter = counter + 1;
      bfsnum[j] = counter;
      order[counter] = j;
    fi
  od
od
```

# Notes on the BFS algorithm

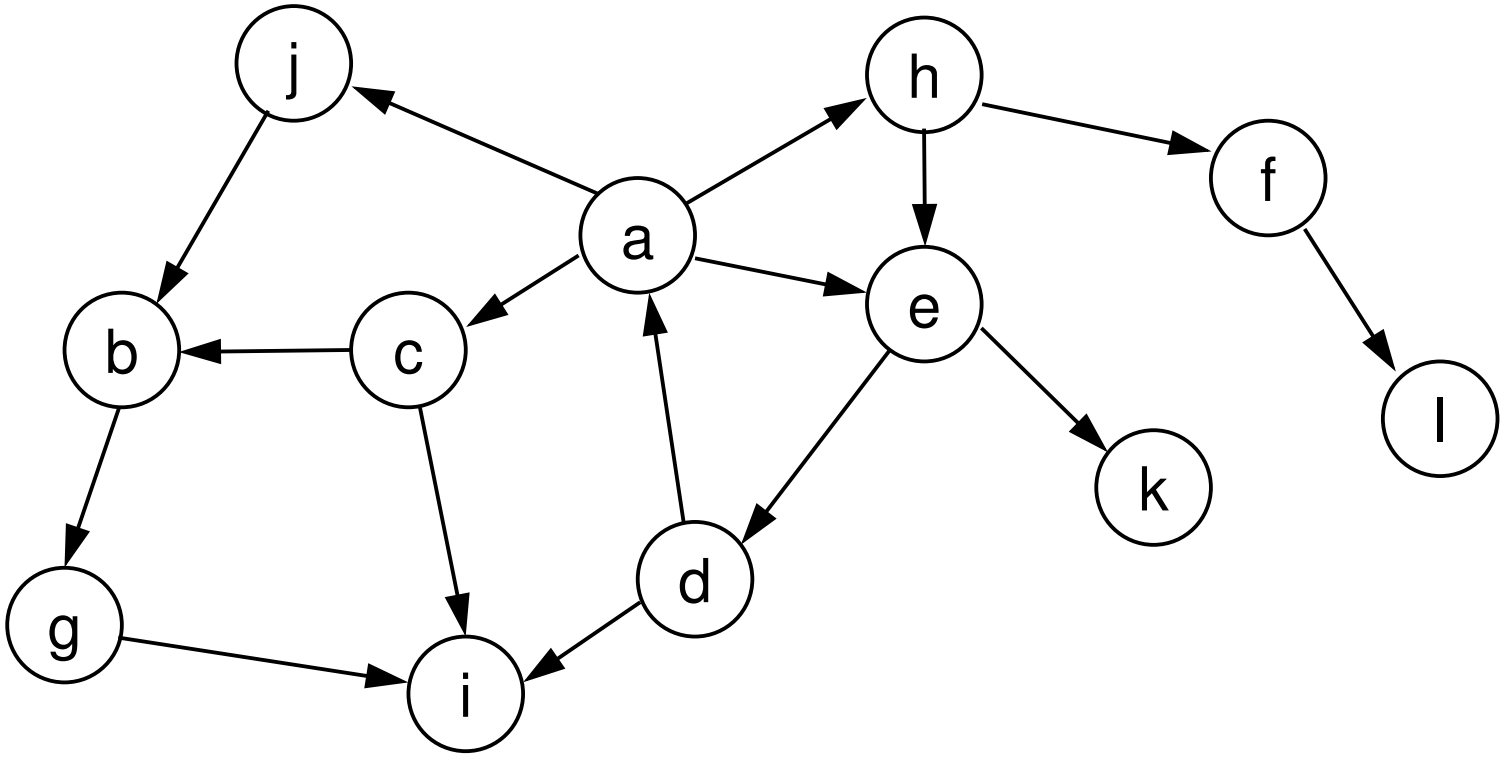
The BFS algorithm uses two arrays:

`bfsnum`: initialized to all zeros; afterwards, `bfsnum[i]` contains the BFS number of node `i`.

`order`: filled in such a way that `order[i]` contains the number of the node with BFS number `i`

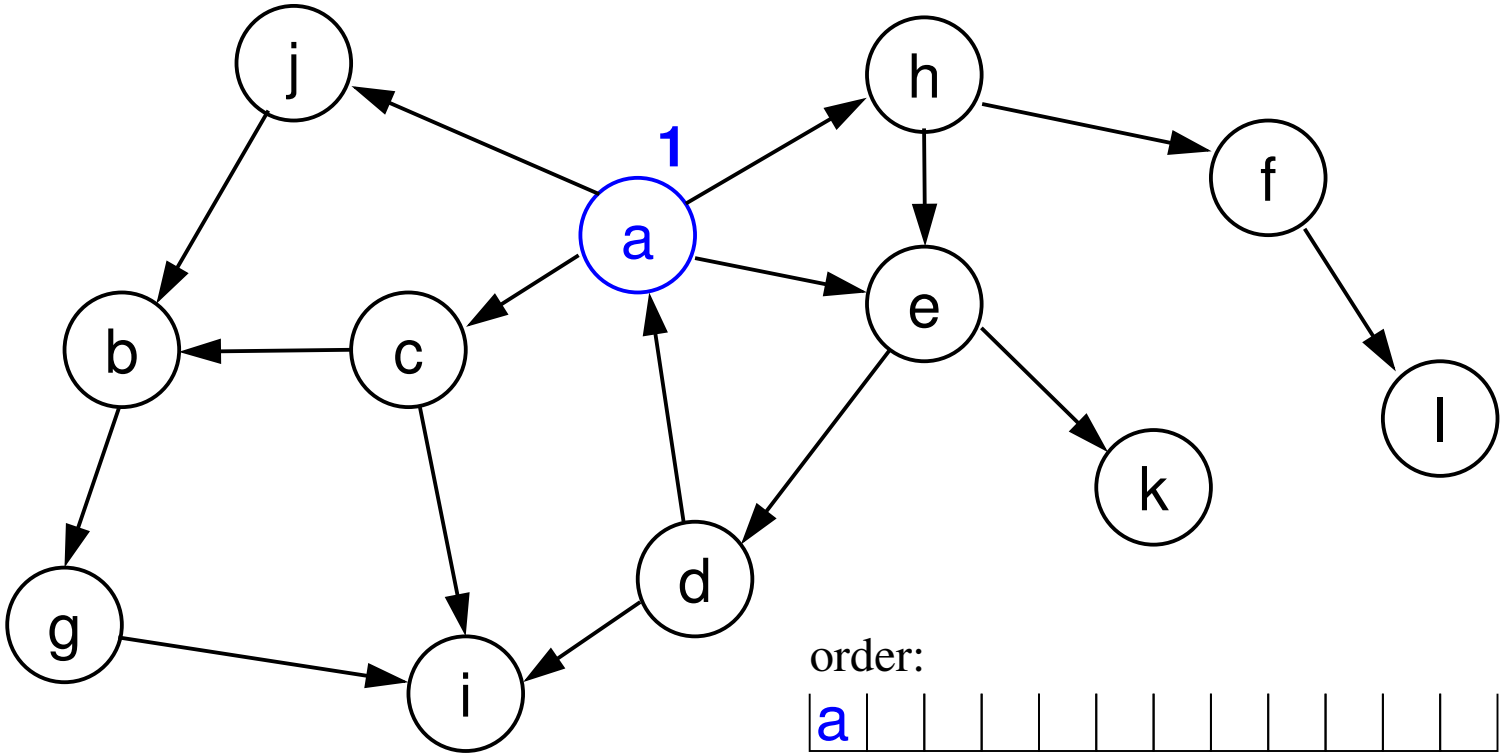
At the end, the two arrays are “inverse”, i.e. if `bfsnum[i]=k`, then `order[k]=i`.

# Running example



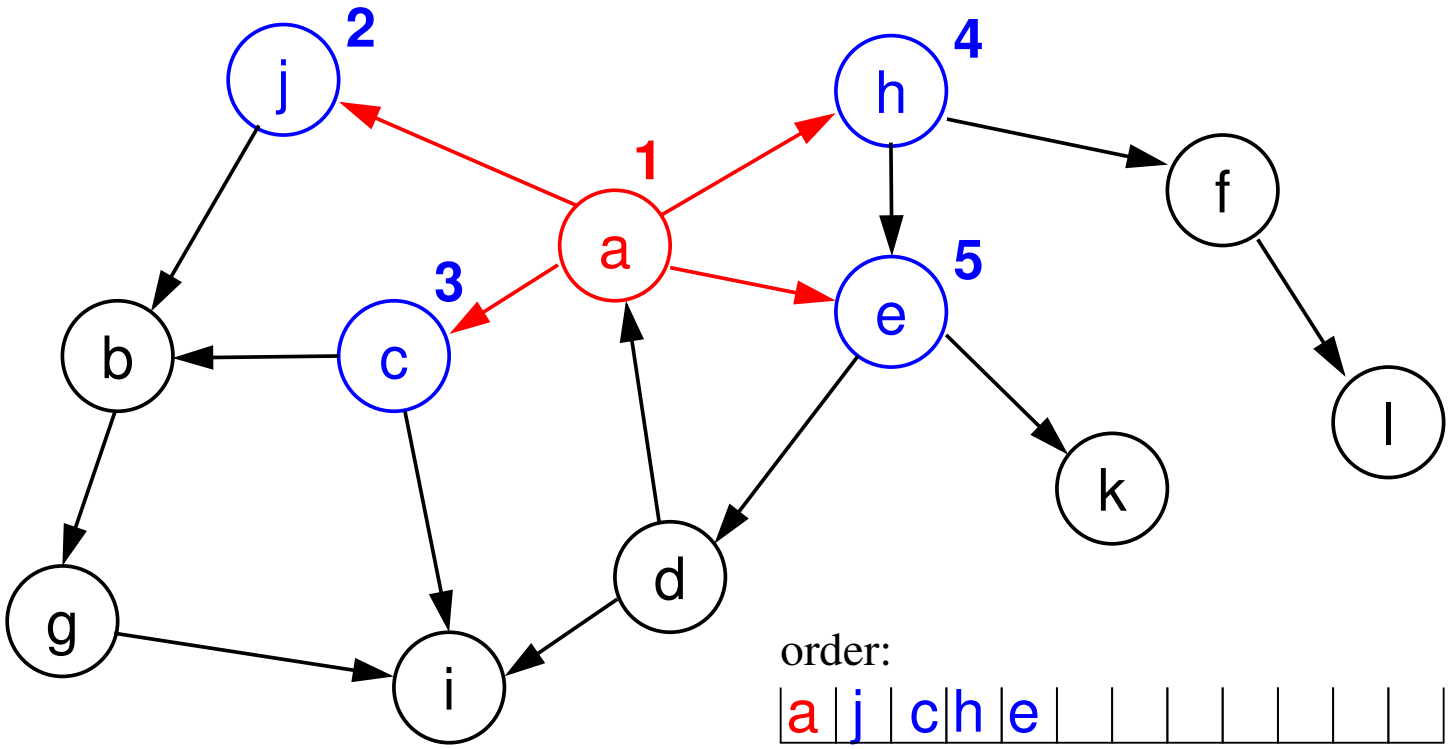

We shall use this graph as a running example. Again, the nodes are numbered *a..l* here instead of `1..12`, so that we can more clearly distinguish between node numbers and their BFS numbers.

# BFS numbering on the example



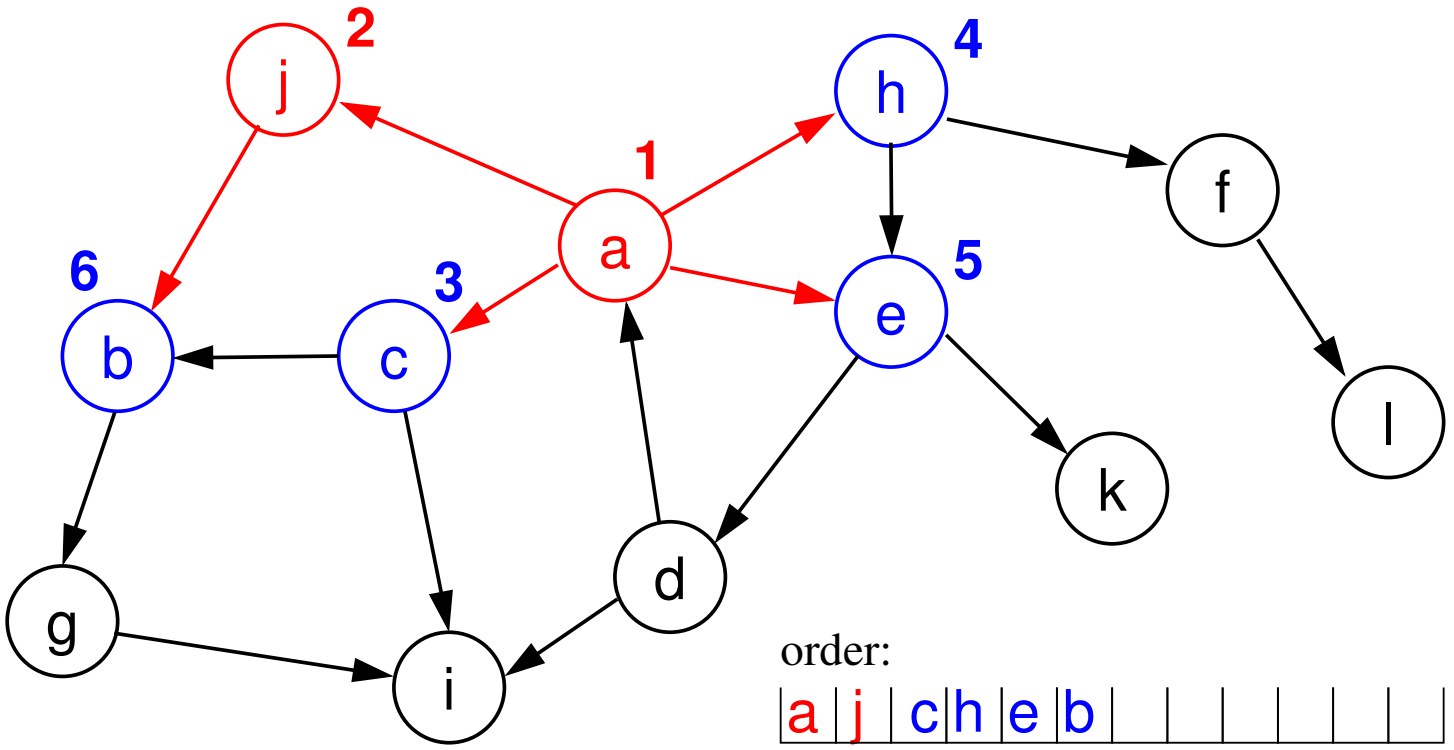

Initially, only the first node (here called *a* instead of 1) gets its BFS number. Assigned BFS numbers shown next each node, and the (partially determined) contents of `order` below the graph.

# BFS numbering on the example



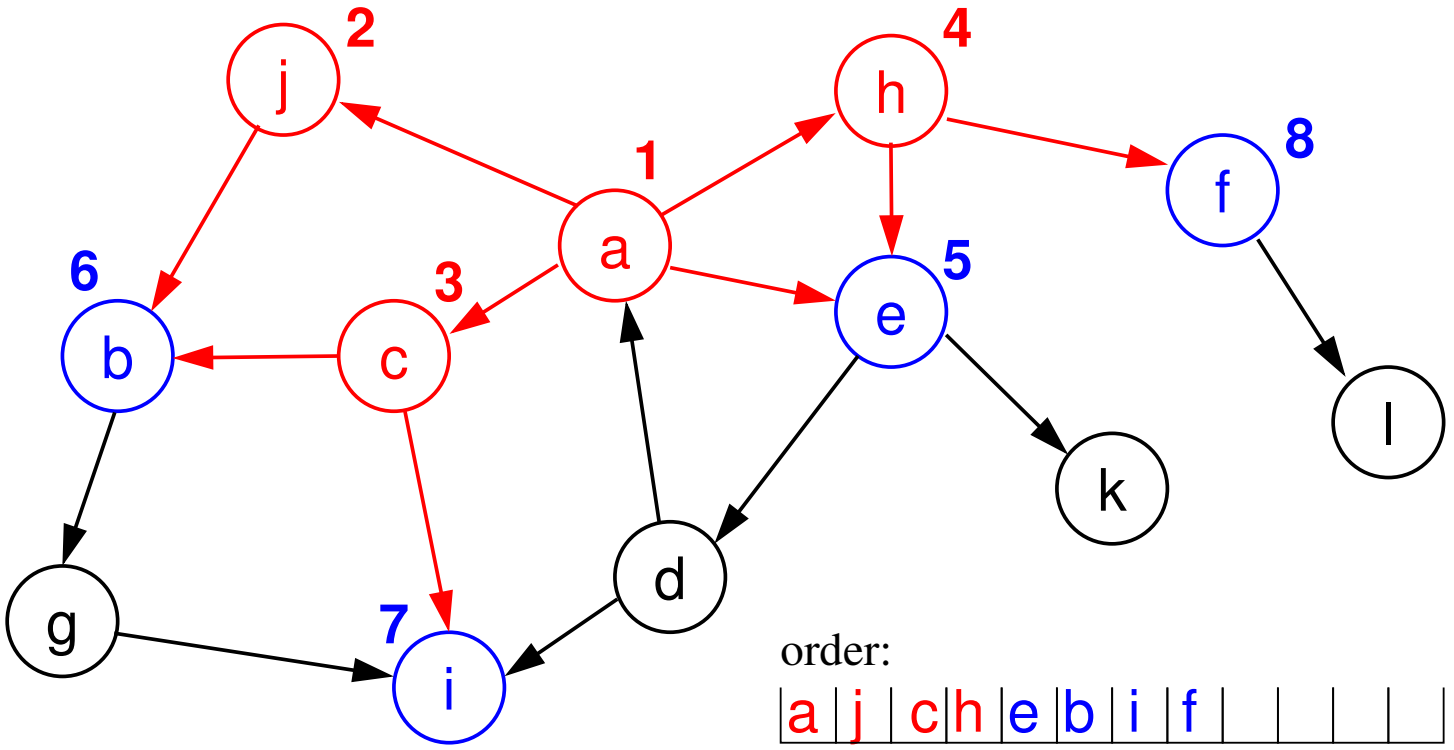

The `while` loop visits *a* first, gives numbers to its neighbours (in *any* order) and puts them into the `order` array.

# BFS numbering on the example



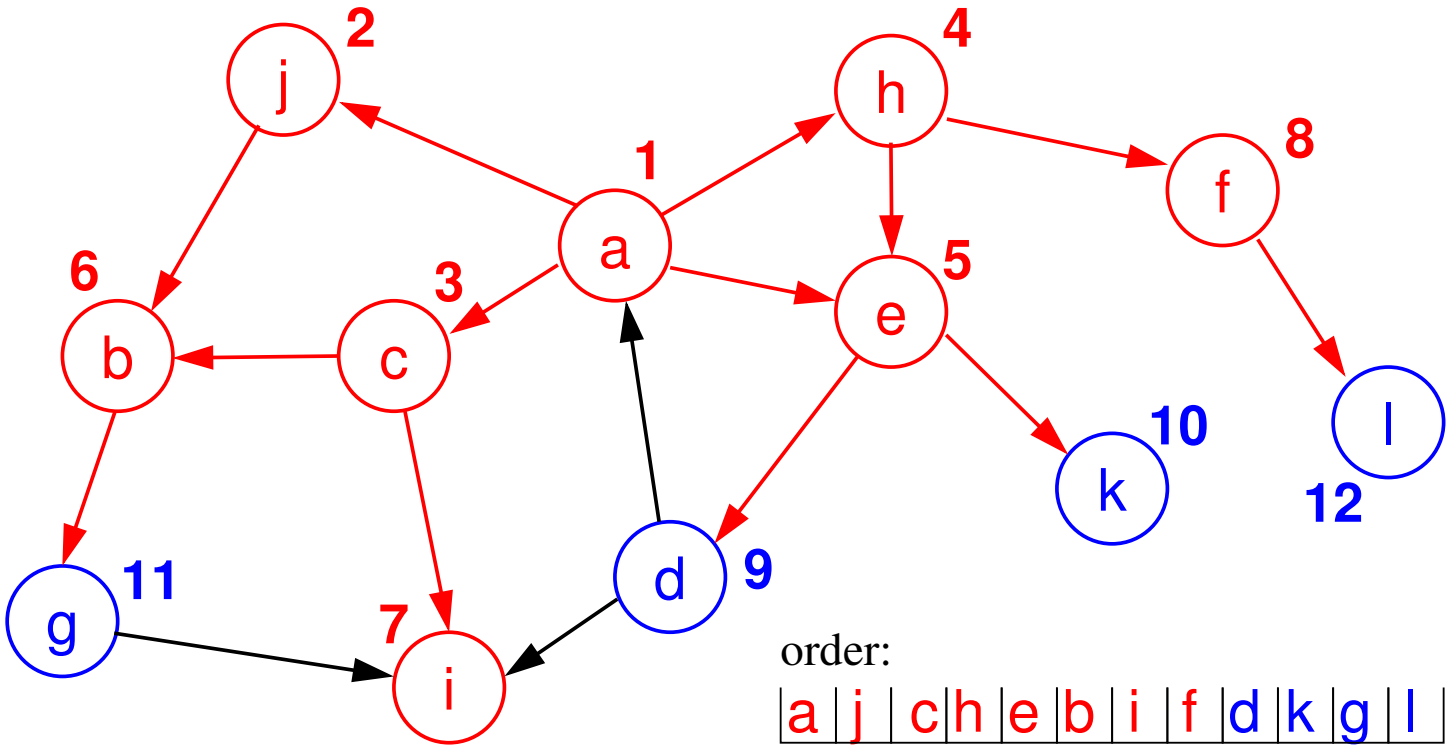

Visited nodes (i.e., nodes whose neighbours have been explored) are shown in red here. “Front” nodes (i.e., nodes with a BFS number that have not yet been visited) are shown in blue. The situation above arises after visiting *j*.

# BFS numbering on the example



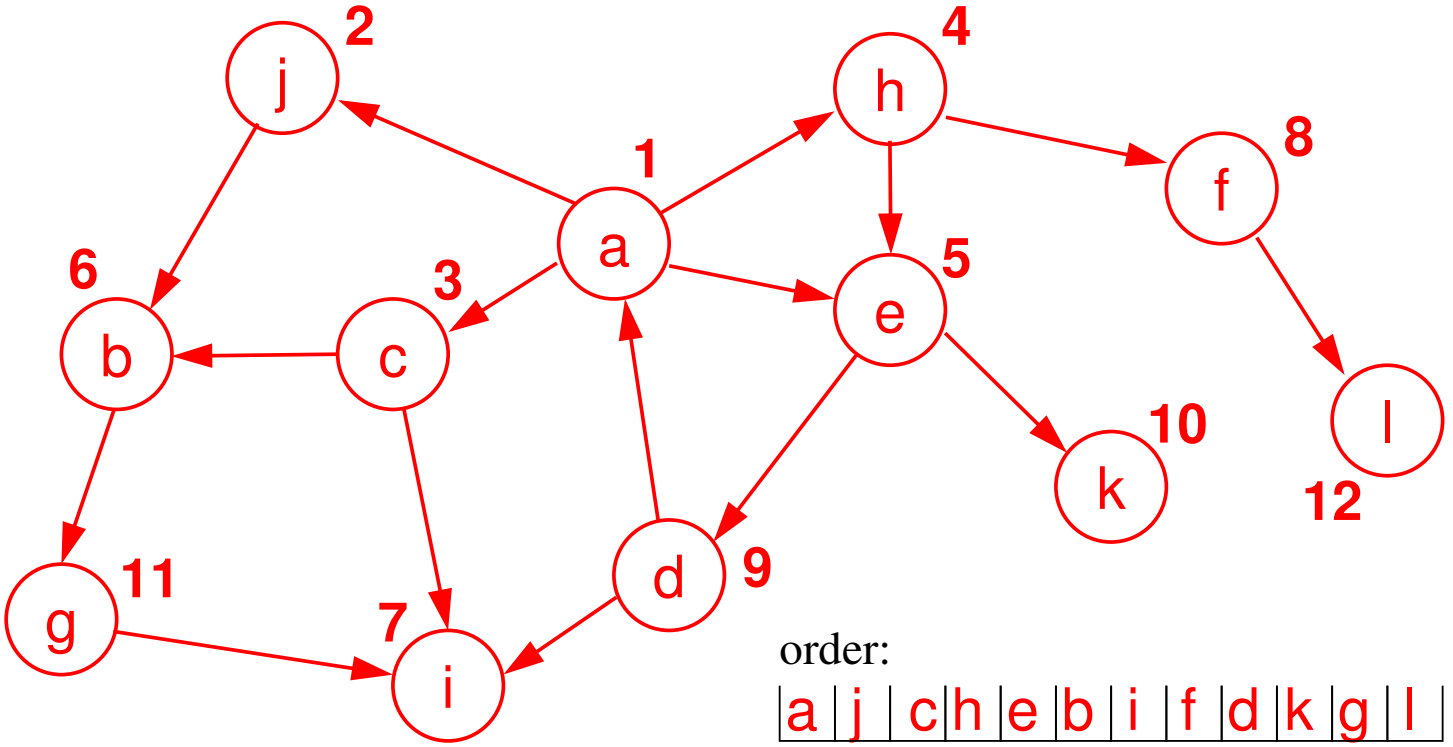

When *c* and *h* are visited next, the BFS numbers of *b* and *e* are left unchanged, only *i* and *f* get fresh numbers.

# BFS numbering on the example



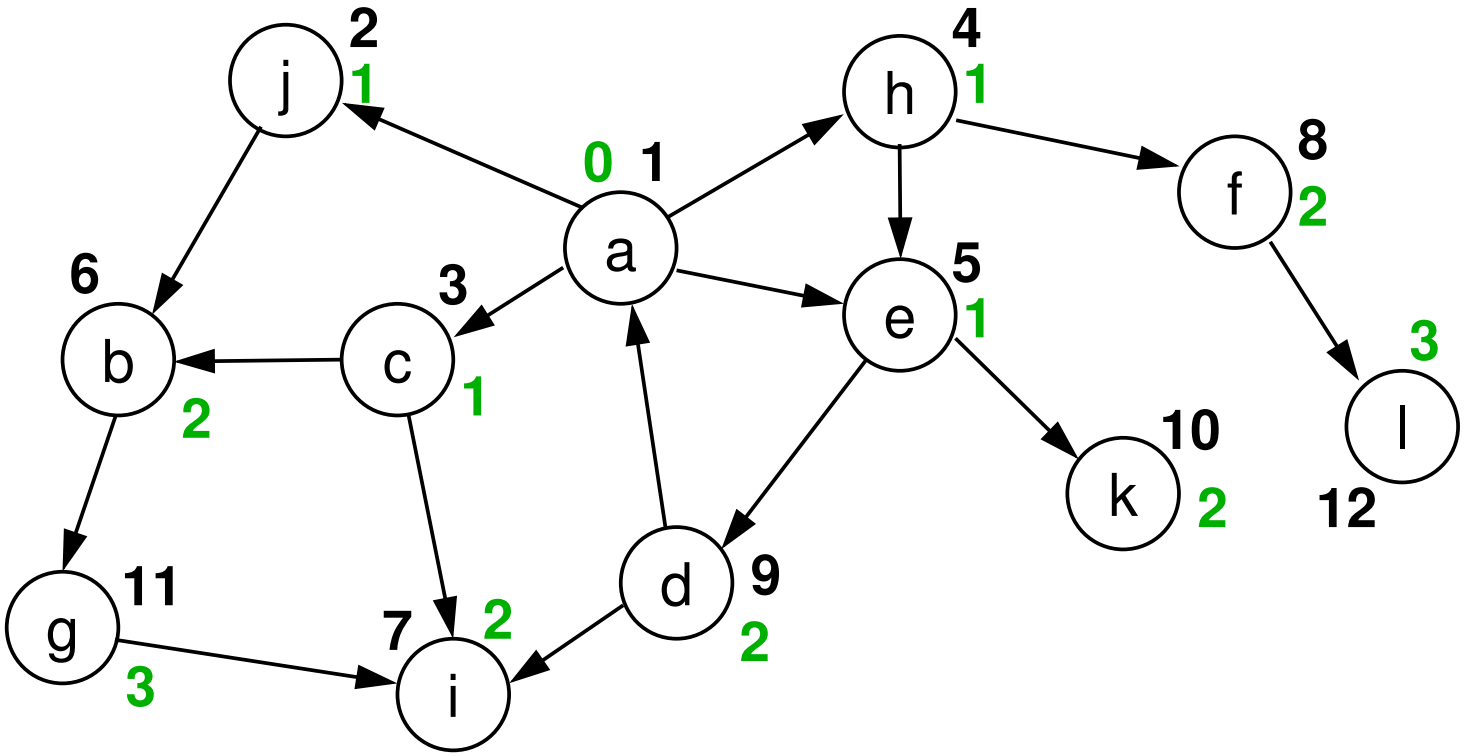

Situation after visiting the nodes that were previously front nodes; now all nodes are numbered. . .

# BFS numbering on the example




. . . so the final four visits do not change anything, and we end up with the numbering shown initially. Notice how the BFS numbers increase with growing distance to *a*.

# BFS numbers and distance




Above, the lengths of the shortest paths from *a* to all other nodes (measured in terms of edges) are shown in green. Indeed, we see that if some node is closer than another to *a*, then that node has a smaller BFS number than the other.

# Computing distances

Remark: We could define a partial order between nodes given by $u \prec v$ iff $u$ has a shorter distance to $a$ than $v$. Then BFS numbering refines this partial ordering into a total ordering.

The distances shown in the previous slide can be computed by a small modification of the BFS algorithm (shown on the next slide). Instead of BFS numbers, that algorithm maintains an array `distance`.

The distance algorithm maintains the following invariant in its `while` loop:

> All visited nodes and all front nodes have the correct distance assigned to them.
>
> None of the unexplored nodes is closer to $a$ than any of the visited or front nodes.

# Algorithm for computing the distances

```
procedure distance
for i = 1 to n do distance[i] = ∞ od;
distance[1] = 0; order[1] = 1;
visited = 0; counter = 1;
while visited < counter do
  visited = visited + 1;
  i = order[visited];
  for j in N(i) do
    if distance[j] = ∞ then
      distance[j] = distance[i] + 1;
      counter = counter + 1;
      order[counter] = j;
    fi
  od
od
```

# Shortest-path problems

We shall now generalize the problem of computing distances.

> Previously, we measured the length of a path as the numbers of edges in it, i.e. all edges had the same “weight” of 1.
>
> Now, let us assume that edges may have different “weights”. These can represent *distances*, *time*, *costs* etc.

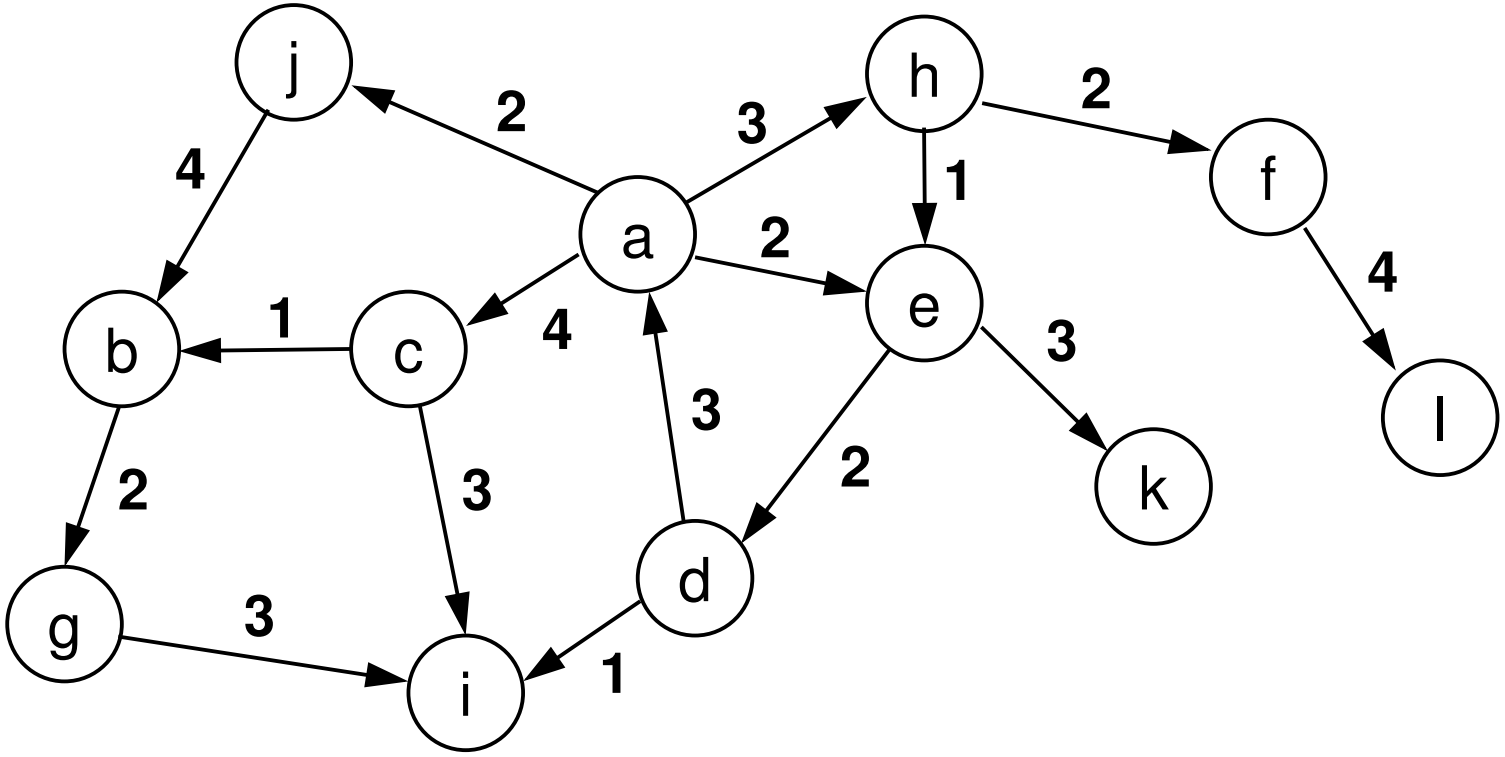
We assume that weights are non-negative numbers.
(In the following, only natural numbers are used as weights. However, the same principles apply to real numbers.)

The shortest-path problem then is:

> Given some directed graph with non-negative weights on the edges and a node $u$ in it, compute the lengths of the shortest paths from $u$ to all other nodes, where the length of a path is taken as the sum of the weights of the edges in it.

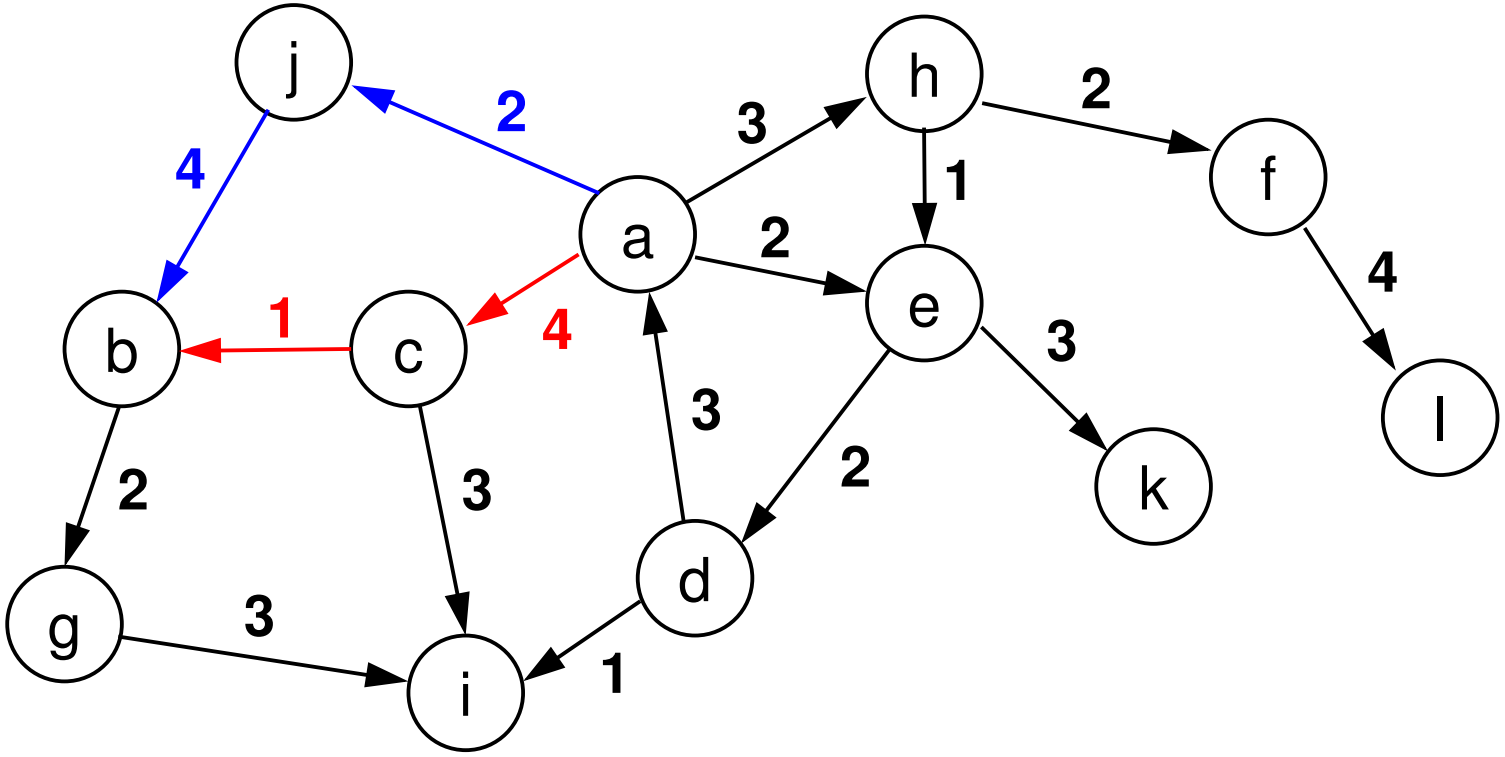
Applications: Finding the shortest route to some target, minimal-cost actions, ...

# Running example with weights



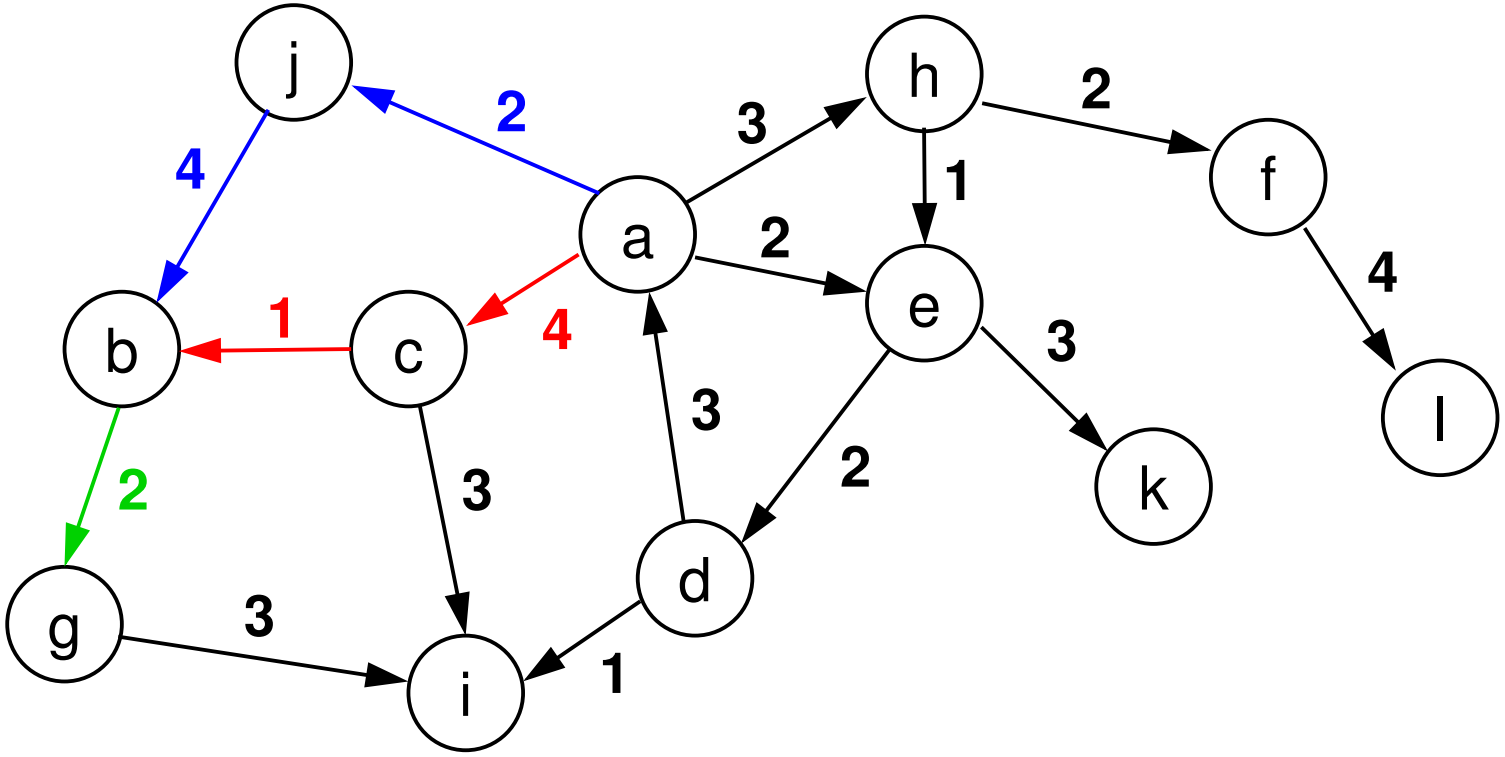

Here's the running example again, this time with some weight attached to each edge. We shall compute the shortest distances starting from *a*.

# Some observations



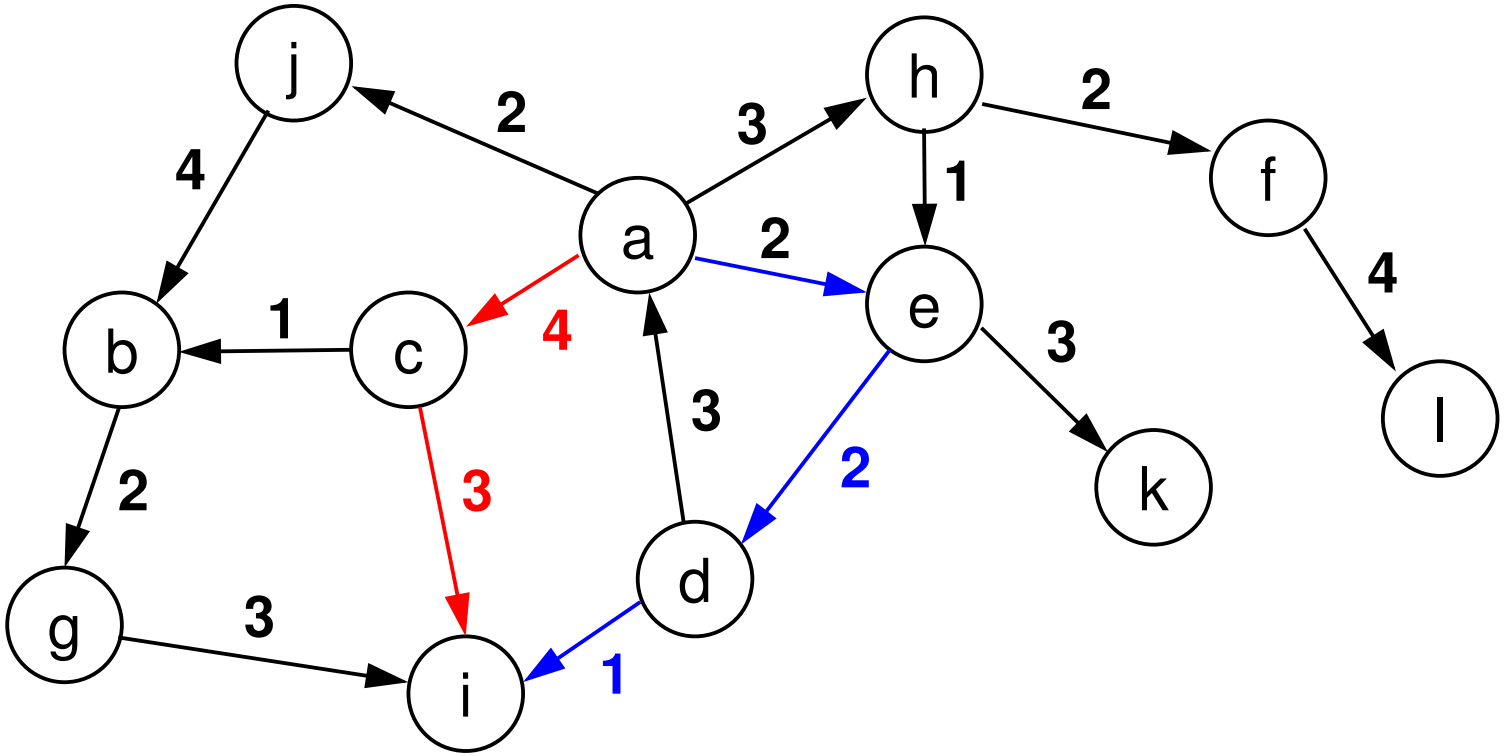

Notice how there different paths from, e.g., from *a* to *b*, can have different lengths. Here, the red path has length 5, whereas the blue path has length 6.

# Some observations




Since we have $x + z < y + z$ iff $x < y$, we can determine the shortest path to $g$ by determining the shortest path to $b$ first. This suggests that we should still employ some “BFS-like” principle.

# Some observations




It may be the case that a path with more edges in it is still shorter than a path with fewer edges. Here, the blue path is shorter than the red one despite containing more edges.

# Towards an algorithm

We shall modify the previous distance algorithm to solve the shortest-path problem.

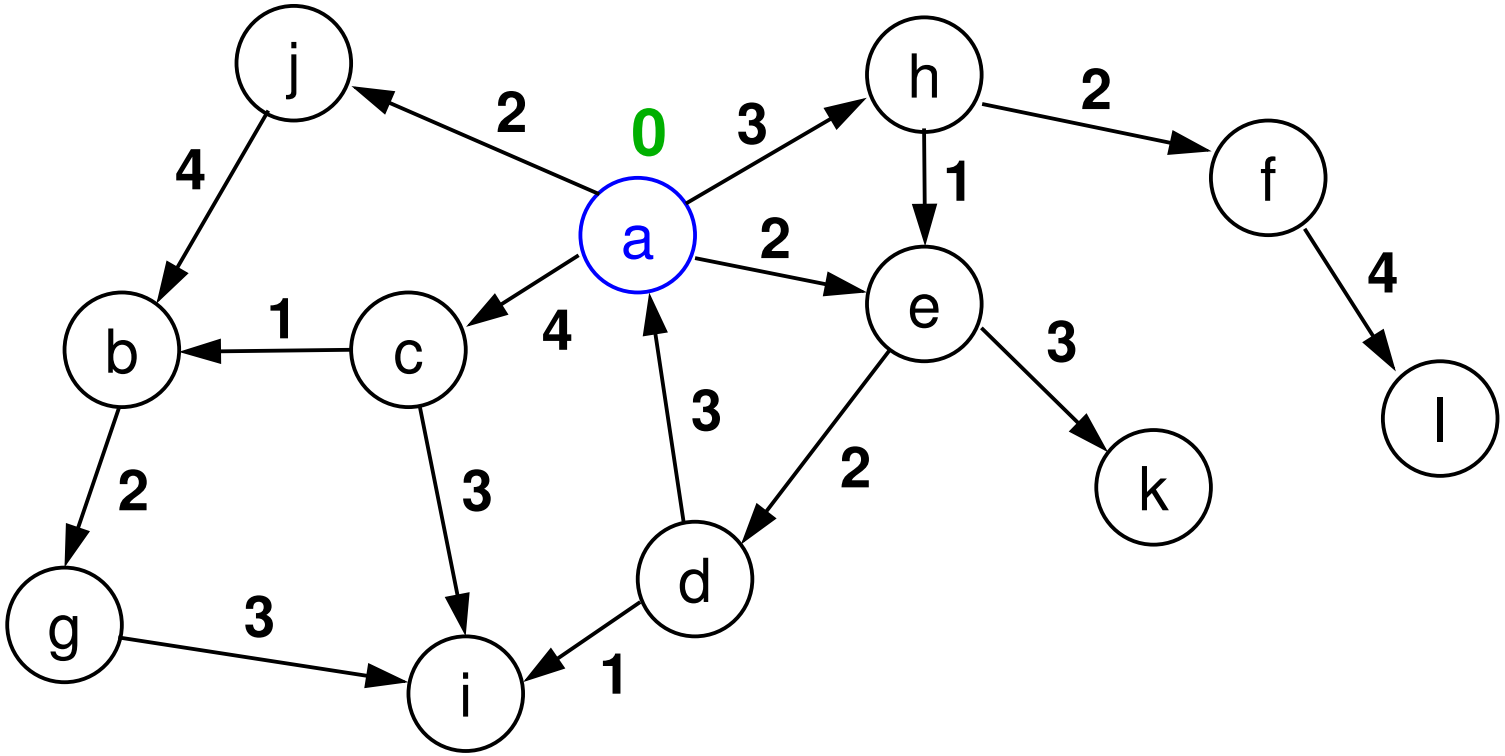
Most elements remain the same:

- We still distinguish visited nodes, front nodes, and unexplored nodes.
- Nodes are still visited in increasing order of distance, starting at *a*.
- Distances “carried forward” from one node to its neighbours.

There are also some changes:

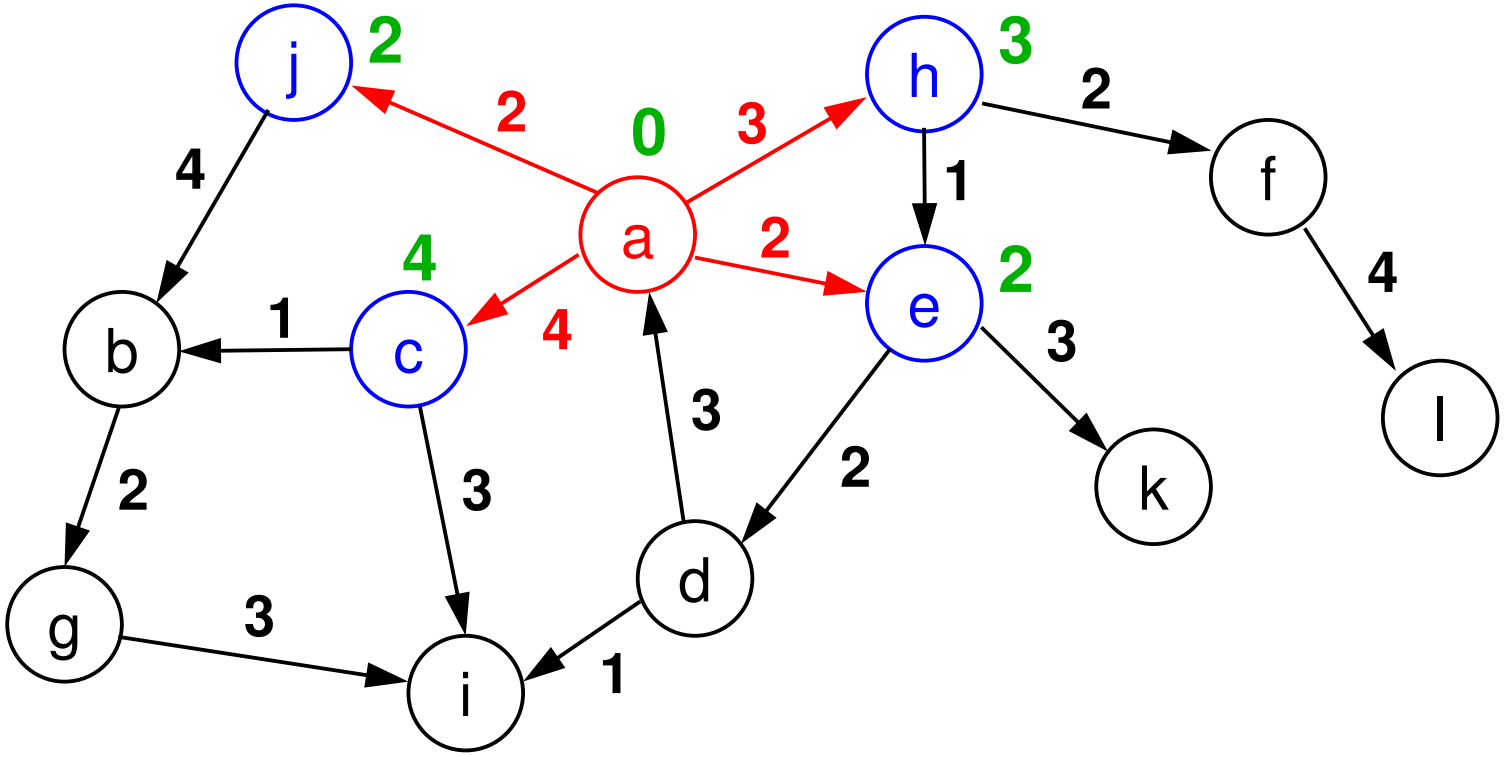
- The distance of a node may be updated more than just once, i.e. we may find one path and afterwards another, shorter path.
- For this reason, the order of the front nodes may change in between.
- We pick a front node with minimal distance to visit next.

# Computing distance: Example



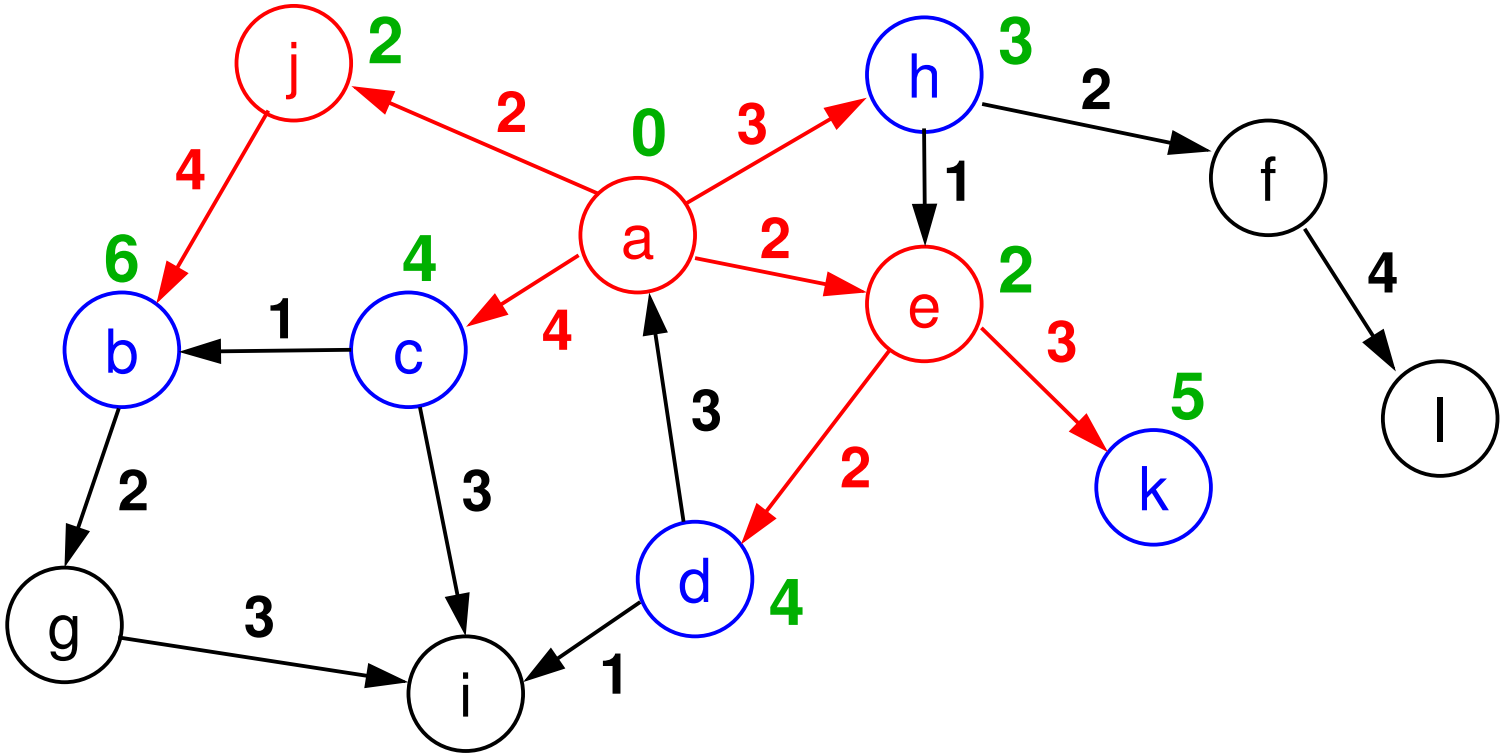

Initially, only *a* is a front node, with distance 0. Nodes where no distance is shown are “unexplored” and still have distance $\infty$ assigned to them.

# Computing distance: Example



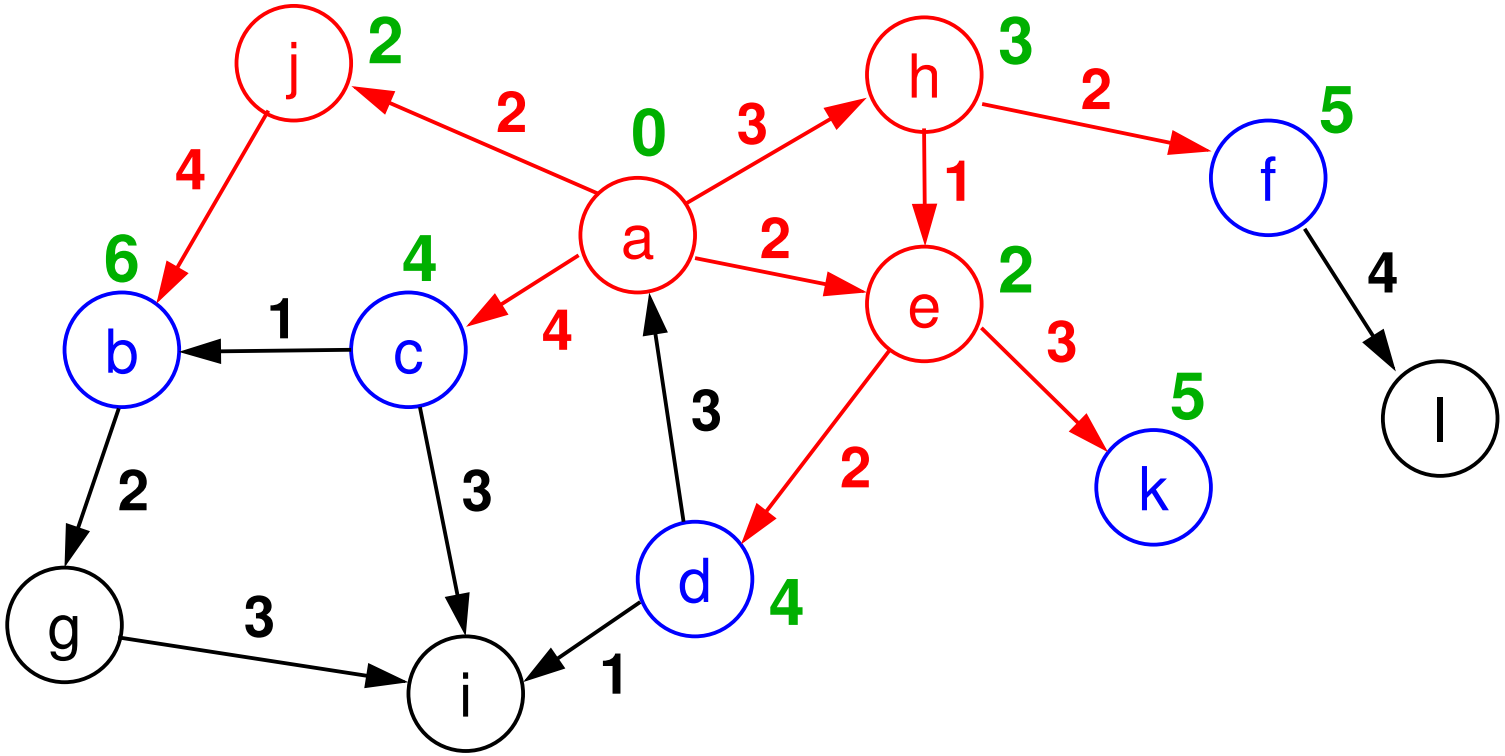

Situation after visiting *a*. Its neighbours have been assigned (preliminary) distances and have been made front nodes. We next pick a front node with minimal distance, i.e. either *j* or *e*, in either order.

# Computing distance: Example



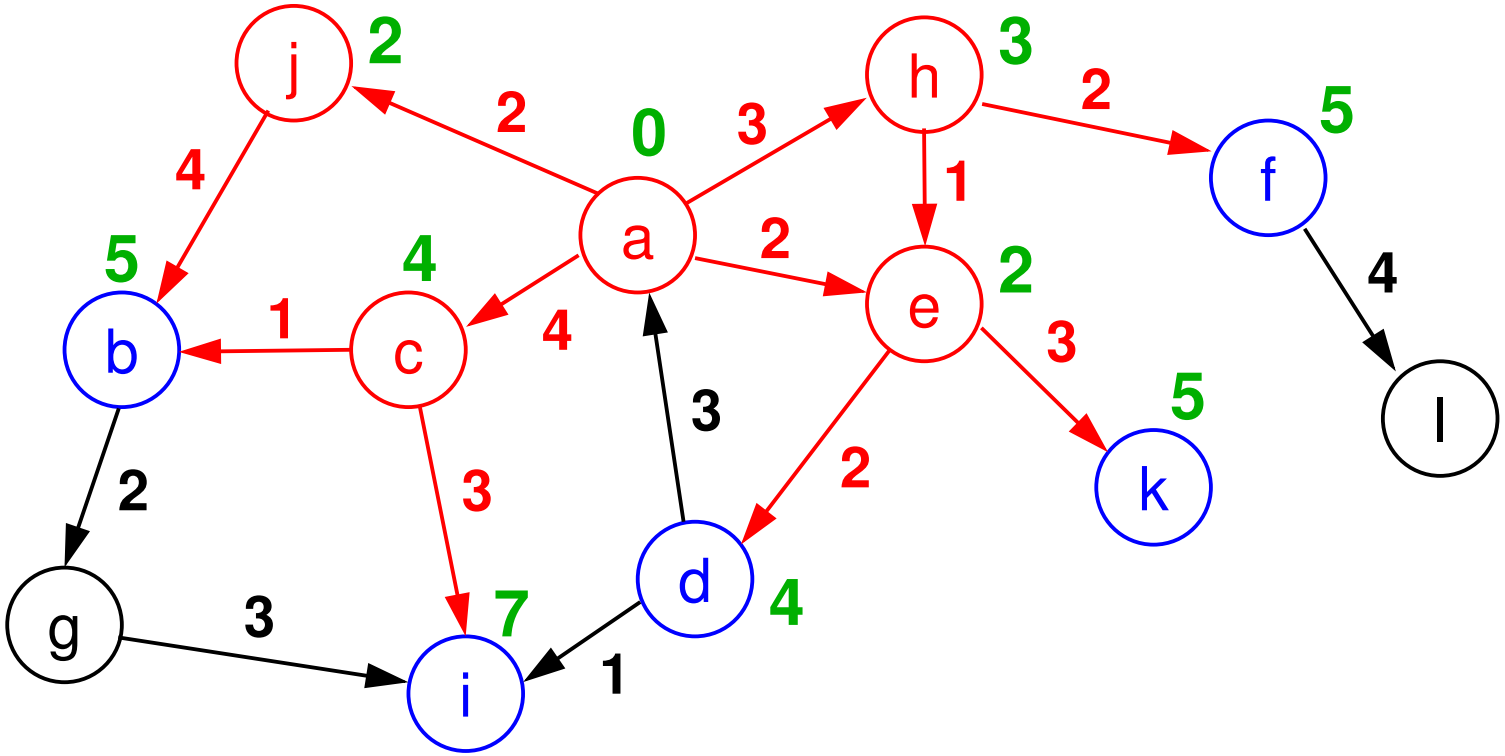

After visiting both *j* and *e*, the front node with minimal distance is *h*.

# Computing distance: Example



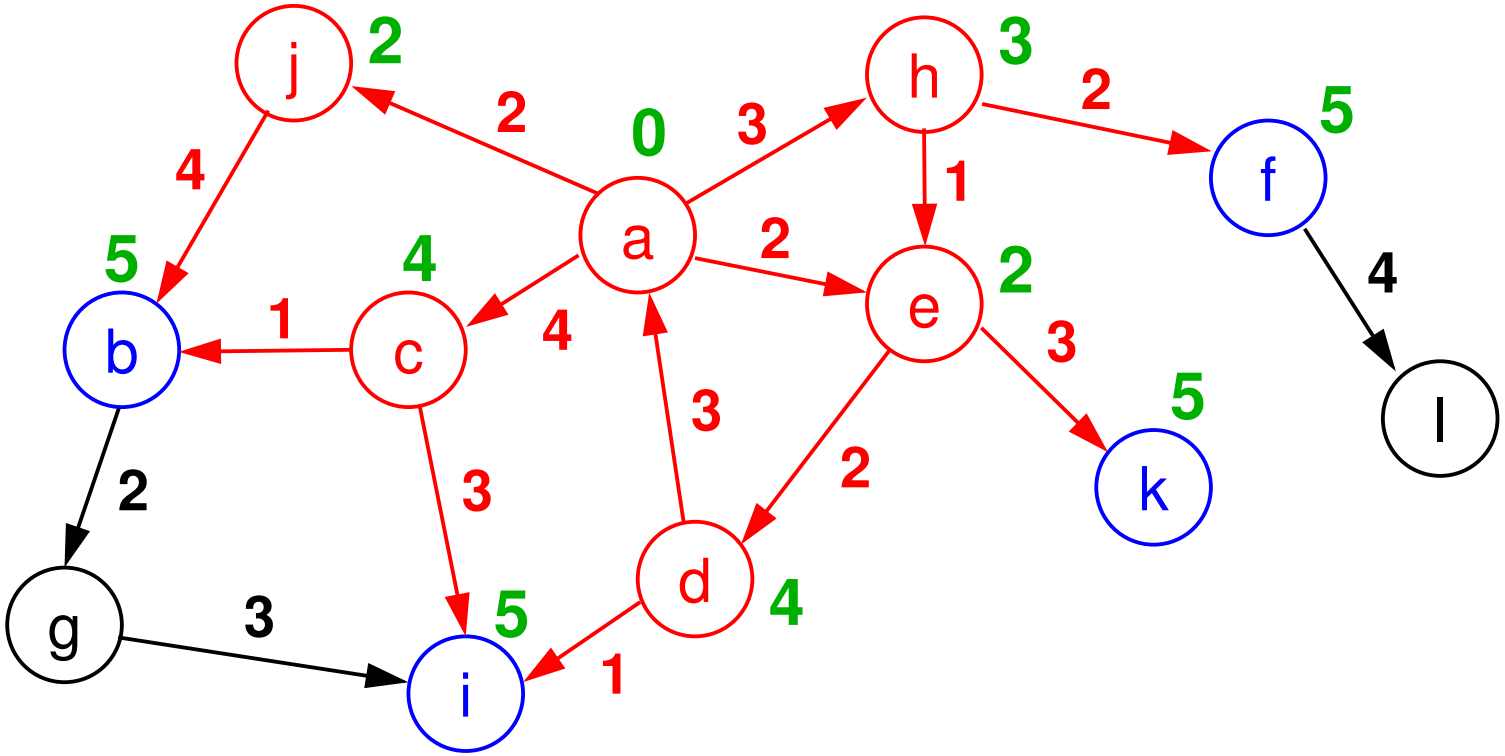

Visiting *h* will make *f* a front node. The distance of *e* remains unchanged because the path to it via *h* is longer than the previously found path.

# Computing distance: Example



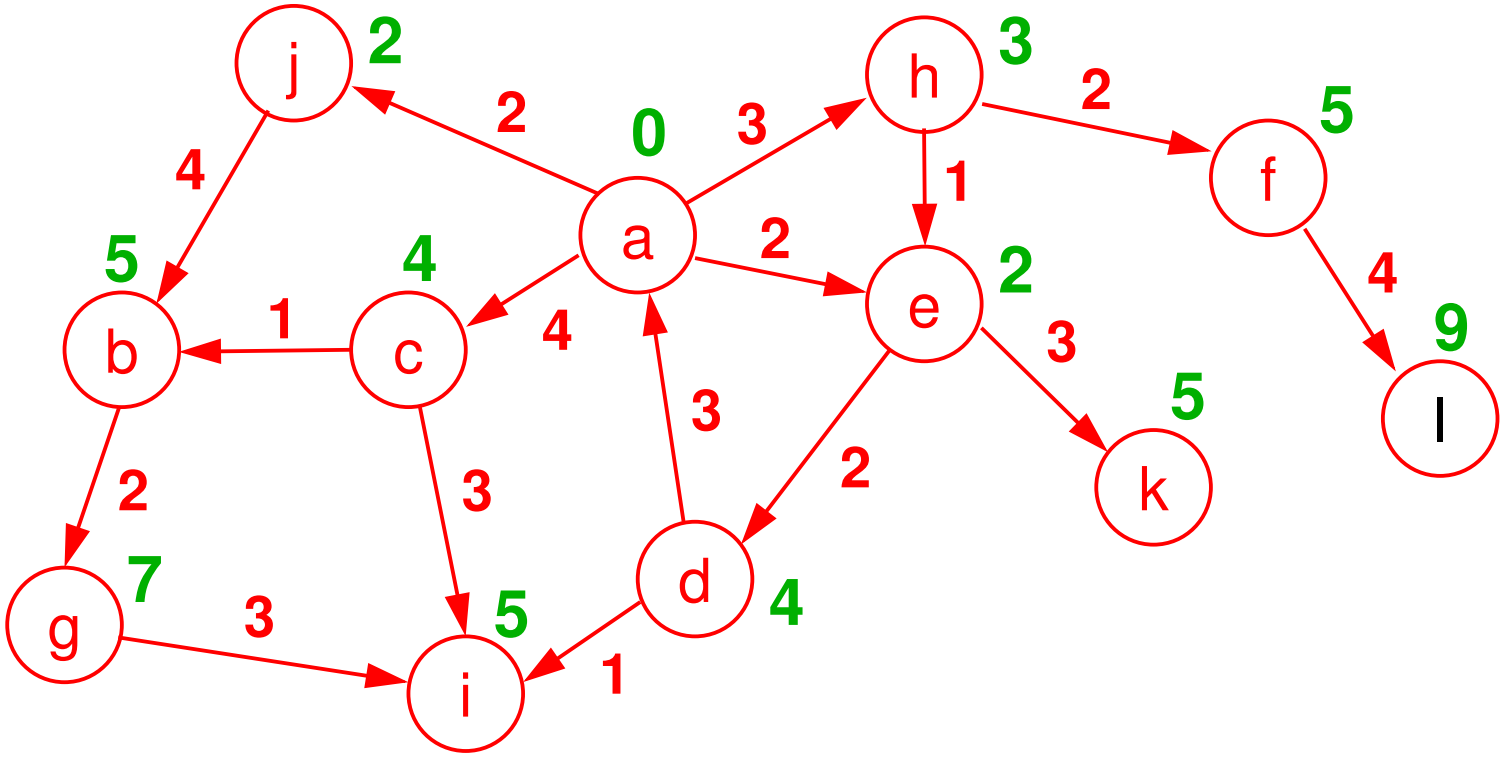

However, visiting *c* next will cause the distance of *b* to be updated because the newly-found path is shorter than the previous one.

# Computing distance: Example




Likewise, visiting *d* will update the distance of *i* to 5.

# Computing distance: Example




The final result is shown above.

# Dijkstra's Algorithm

The method we just used for obtaining the lengths of the shortest paths is also called Dijkstra's algorithm, after its inventor. It is summarized in full below.

1. Initially, all nodes are unexplored and have distance $\infty$.

2. Make $u$ a front node with distance $0$.

3. If no front nodes are left, terminate.

4. Pick $v$ among the front nodes with minimal distance and make $v$ a visited node.

5. For all neighbours $w$ of $v$, if the currrent distance of $w$ is larger than the distance of $v$ plus the weight of the edge from $v$ to $w$, then update the distance of $w$ to the latter and make $w$ a front node.

6. Continue at step 3.

# Correctness of Dijkstra's algorithm

The invariants maintained by Dijkstra's algorithm are just a little different from our previous distance algorithm:

> All visited nodes have the correct shortest distance assigned to them.
>
> The distance assigned to all front node is the length of the shortest path among those that use only visited nodes.
>
> There is no path to any unexplored node using only visited nodes.

The correctness of the algorithm follows from this invariant.

In particular, it follows from the invariant that the front node with minimal distance must have the correct shortest distance. (Why?)

The correctness arguments hinge vitally on the fact that there are no edges with negative weights. (What would happen otherwise?)

# Complexity of Dijkstra's algorithm

Dijkstra's algorithm visits each node and every edge exactly once, contributing $\mathcal{O}(|V| + |E|)$ time if run on a directed graph $G = (V, E)$.

Picking a front node with a minimal distance takes some effort, too. If the front nodes are organized in a heap (cf. the Heapsort algorithm), then picking the minimal-distance node is trivial. However, removing a node from the heap and inserting one takes at worst $\mathcal{O}(\log |V|)$ time (although usually, there are far fewer than $|V|$ front nodes in the heap).

Still, the worst case complexity is $\mathcal{O}((|V| + |E|) \cdot \log |V|)$.

# Directed search

Dijkstra's algorithm computes the shortest distance from a given node $u$ to *all* other nodes.

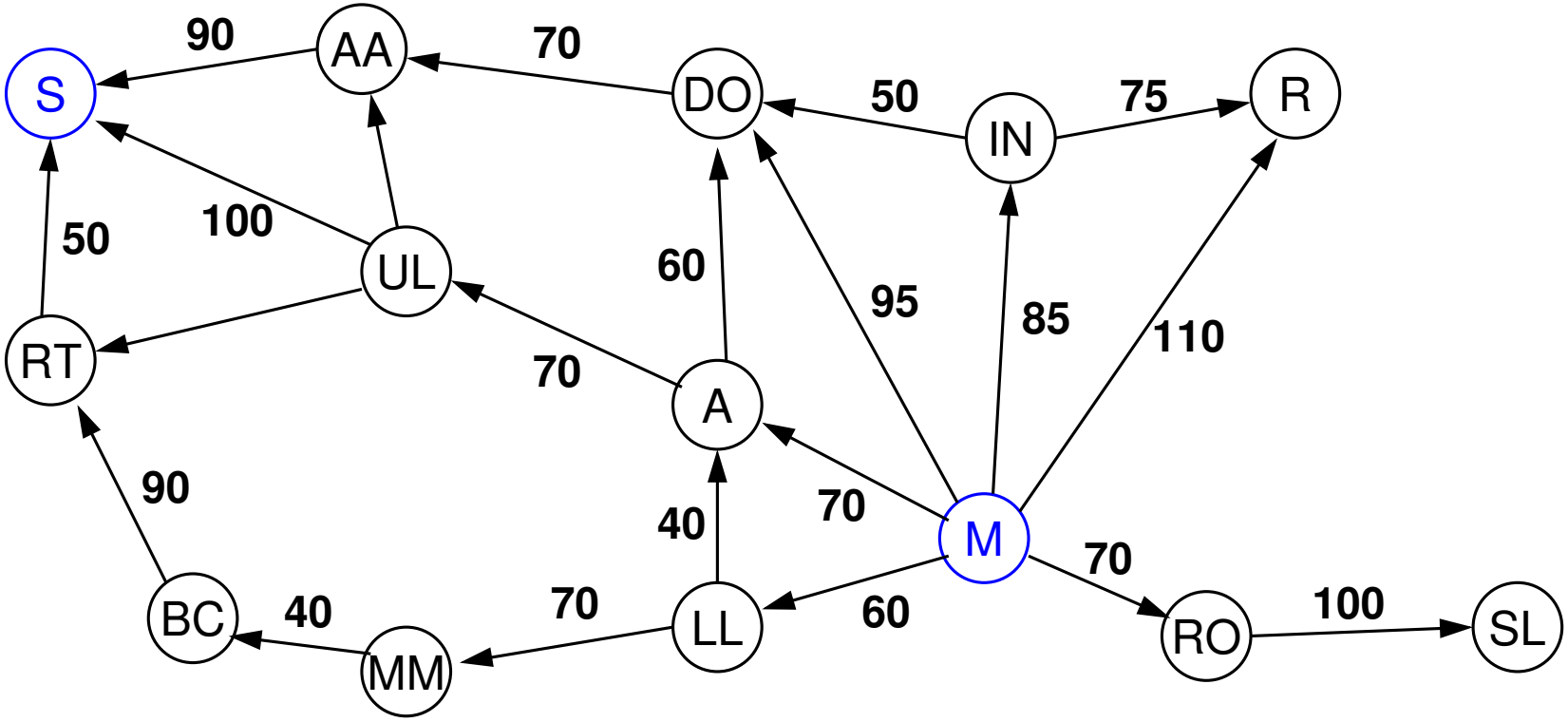
Suppose that we are only interested in the distance between two nodes $u$ and $v$.

Obviously, Dijkstra's algorithm could be used for this; start at $u$ and abort when $v$ is about to be visited.

However, the BFS principle explores in all directions at once from $u$. It may explore many nodes which are not "relevant" in order to reach $v$.

Therefore, we may want to optimize the procedure in order to find the shortest path to $v$ more quickly. This is called directed search.

# Example




Suppose that you want to ride from Munich (M) to Stuttgart (S).
Intuitively, you would limit your attention to paths that approximately in the right direction and ignore others.

# Using intuition

When searching in this intuitive way, we are using some heuristic that takes into account additional knowledge about the situation. In the example, our additional knowledge is that we are searching a “real” map.

For instance, we “know” that going via Salzburg will be no good because it is farther away from Stuttgart than even Munich. I.e., we *already* have some estimate how far it will be from Munich resp. Salzburg to Stuttgart, and this estimate plays a role for our decision.

We shall discuss how this intuition can be used to find the target faster.

# Adding heuristic values

Suppose that we have a heuristic function $h$ for conservatively estimating the distance from any node to the target $v$.

I.e. for each node $w$ we have a value $h(w)$ that gives an estimate for the distance from $w$ to $v$.

$h$ is called a monotone heuristic if it satisfies the "triangle property", i.e.:

> Let $w, z$ be two nodes, and let $D$ be the *actual* shortest distance from $w$ to $z$. Then $h(w) \leq D + h(z)$.

Note: If $h(v) = 0$, then this property implies that $h(w)$ may be at least as large as the real distance from $w$ to $v$.

Note: When searching on a real map, taking the distance "as the crow flies" satisfies this property.

# The $A^*$ algorithm

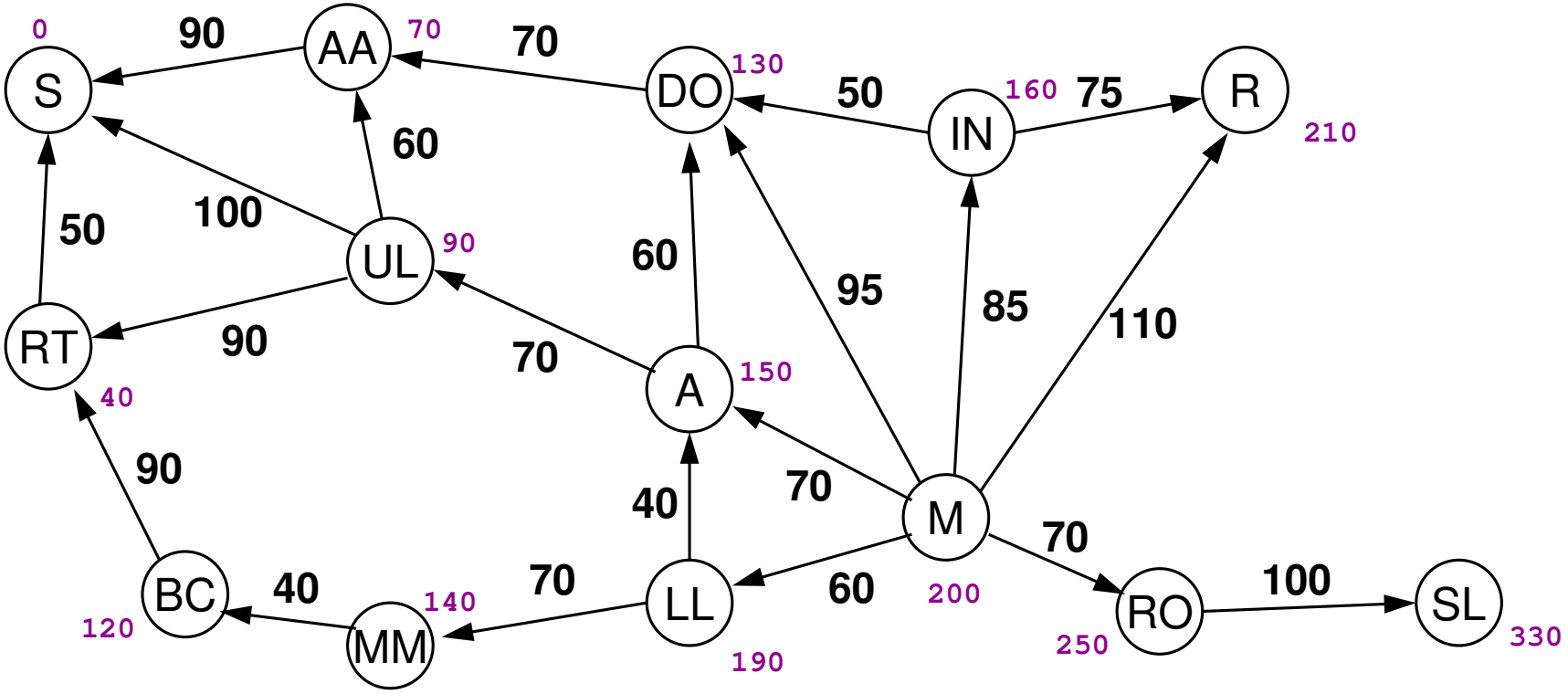
The so-called $A^*$ algorithm is a variant of Dijkstra's that takes heuristics into account. Its only modification is the selection of the next front node to be visited:

> In each iteration of the `while` loop, the $A^*$ algorithm picks a front node $w$ for which the sum of its distance (from $u$) and the value $h(w)$ is minimal among all front nodes.

Moreover, the $A^*$ algorithm terminates when $w = v$.

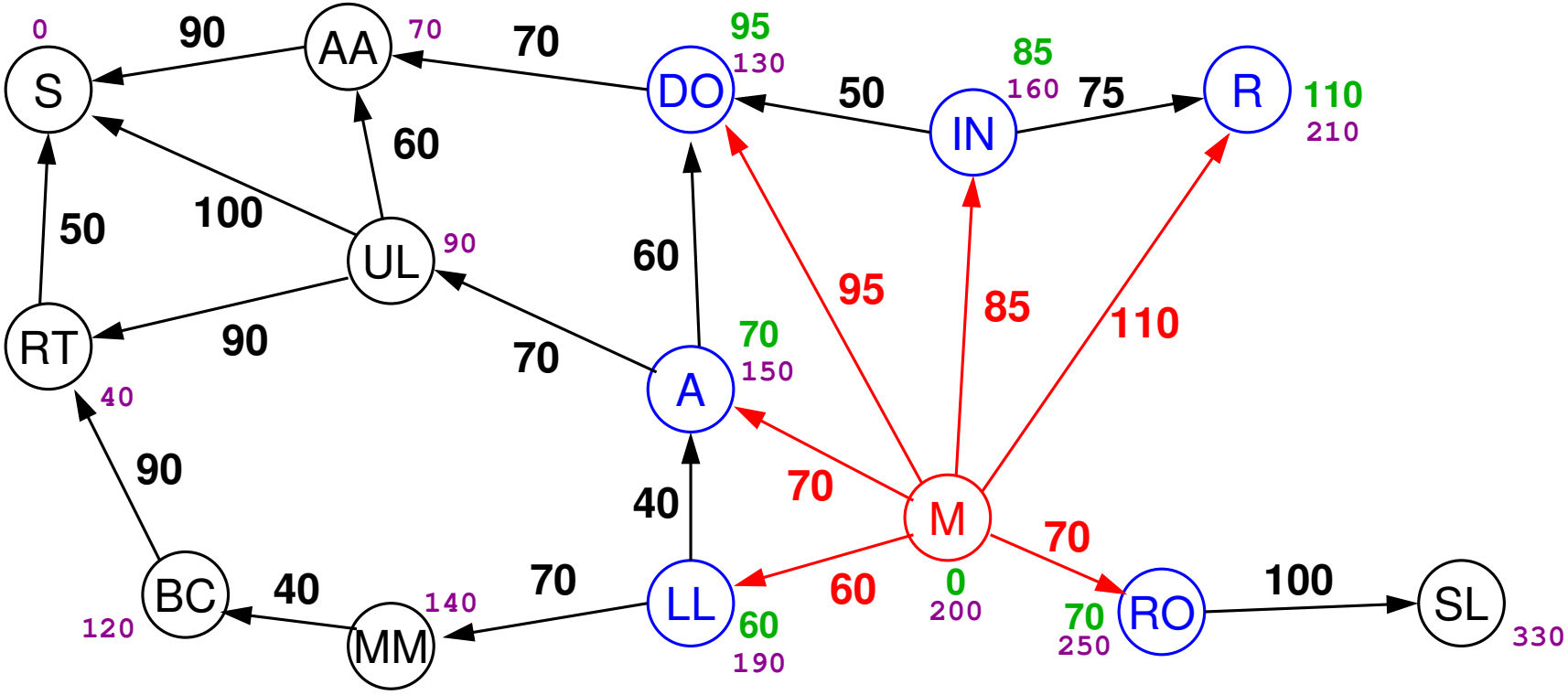
The $A^*$ algorithm can be shown correct if $h$ is monotone, i.e. in that case it will still compute the shortest distance from $u$ to $v$, but in general it will consider fewer nodes.

# Example



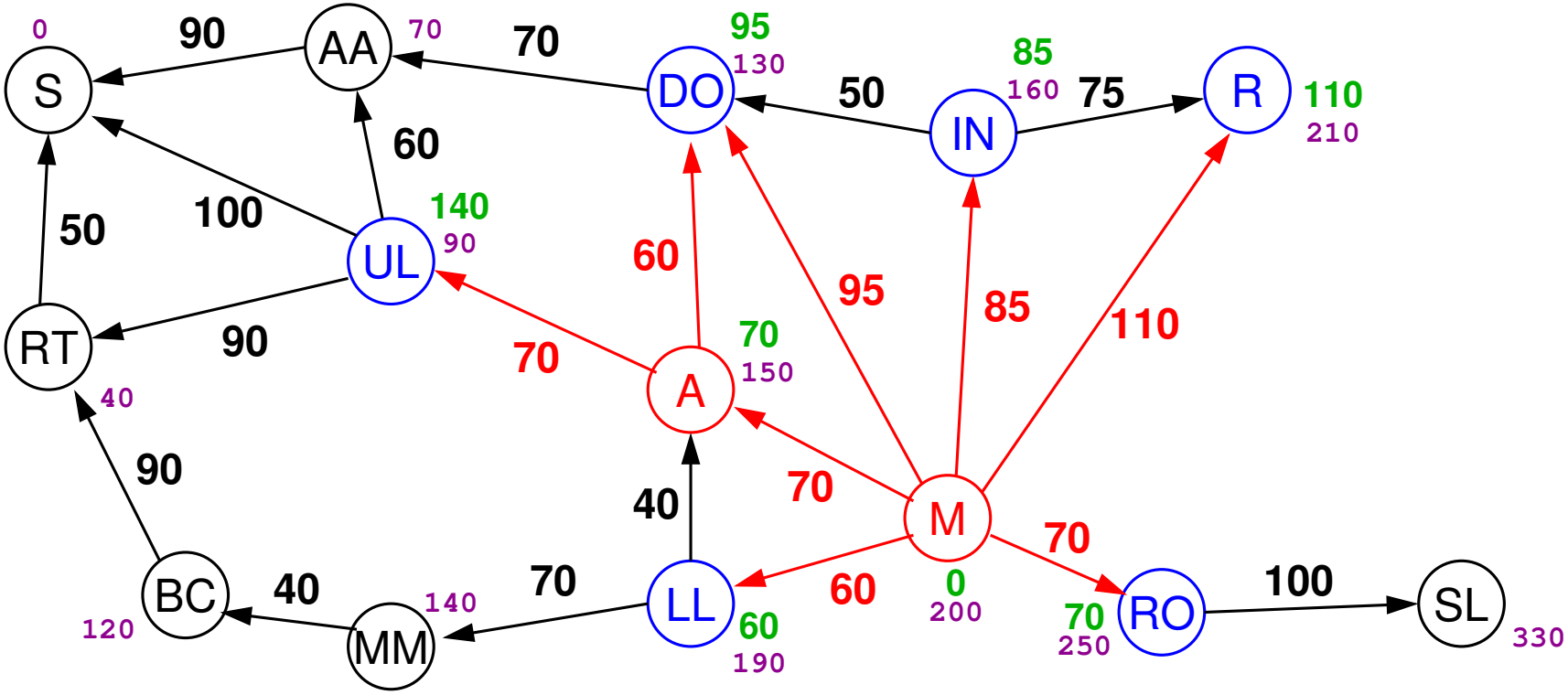

The example together with a monotone heuristic, given in purple typewriter script.

# Running $A^*$ on the example



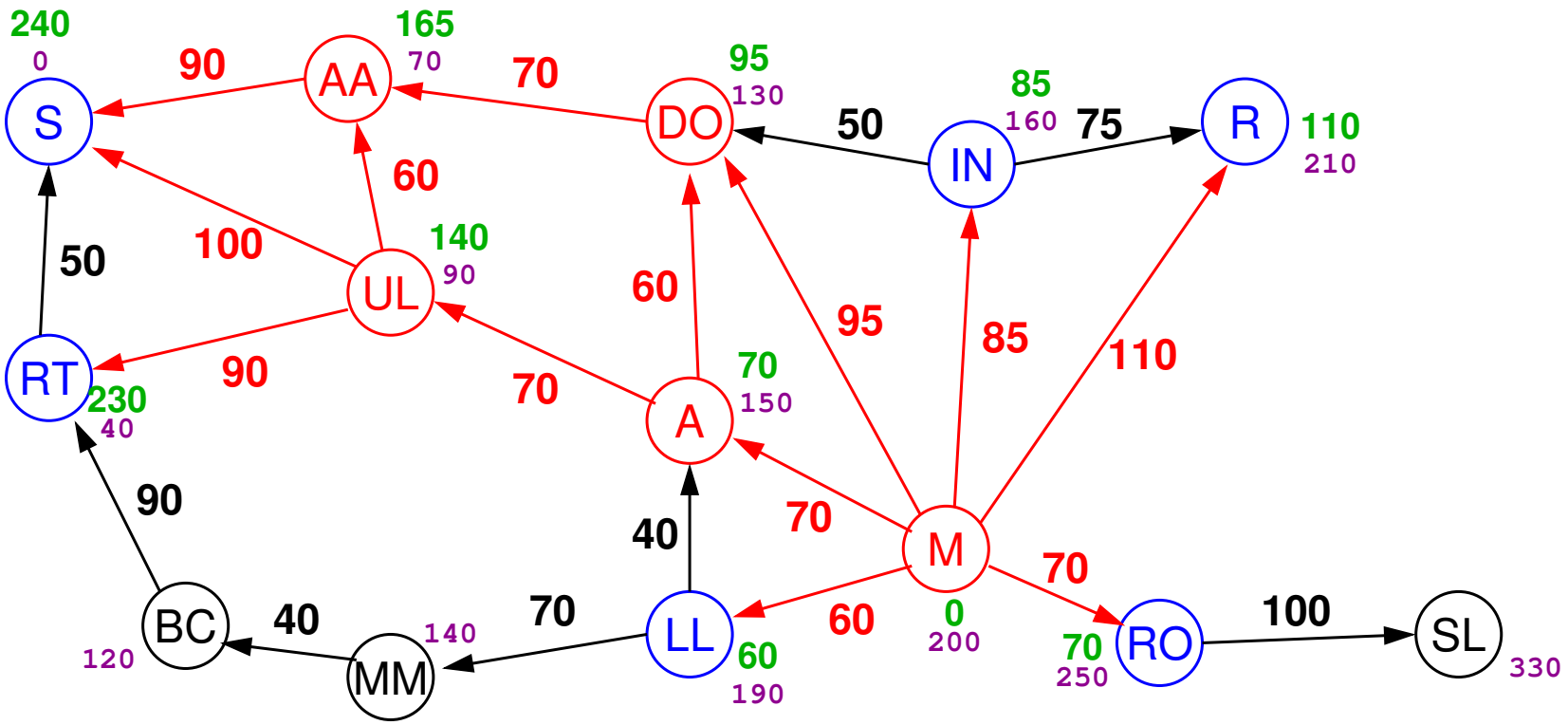

The result after visiting Munich (M); computed distances are given in green. The front node with the smallest sum is Augsburg (A), with a sum of $70 + 150 = 220$.

# Running $A^*$ on the example




Result after visiting Augsburg (A). The next node to be visited is Donauwörth (DO), with a sum of $95 + 130 = 225$.

# Running $A^*$ on the example




After Donauwörth (DO), the next nodes to be visited are Ulm (UL), Aalen (AA), and Stuttgart (S), at which point we terminate. The computed distance to Stuttgart is 240, which is indeed the shortest distance. Only those nodes which are roughly between Munich and Stuttgart were visited.

# Correctness of $A^*$

Previously we claimed that $A^*$ works correctly (i.e., finds the shortest distance to $v$) provided that the heuristic $h$ is monotone.

$A^*$ differs from Dijkstra only in the order in which nodes are visited. However, we shall prove that – like in Dijkstra – that when a front node is chosen to be visited, it is already annotated with the correct shortest distance.

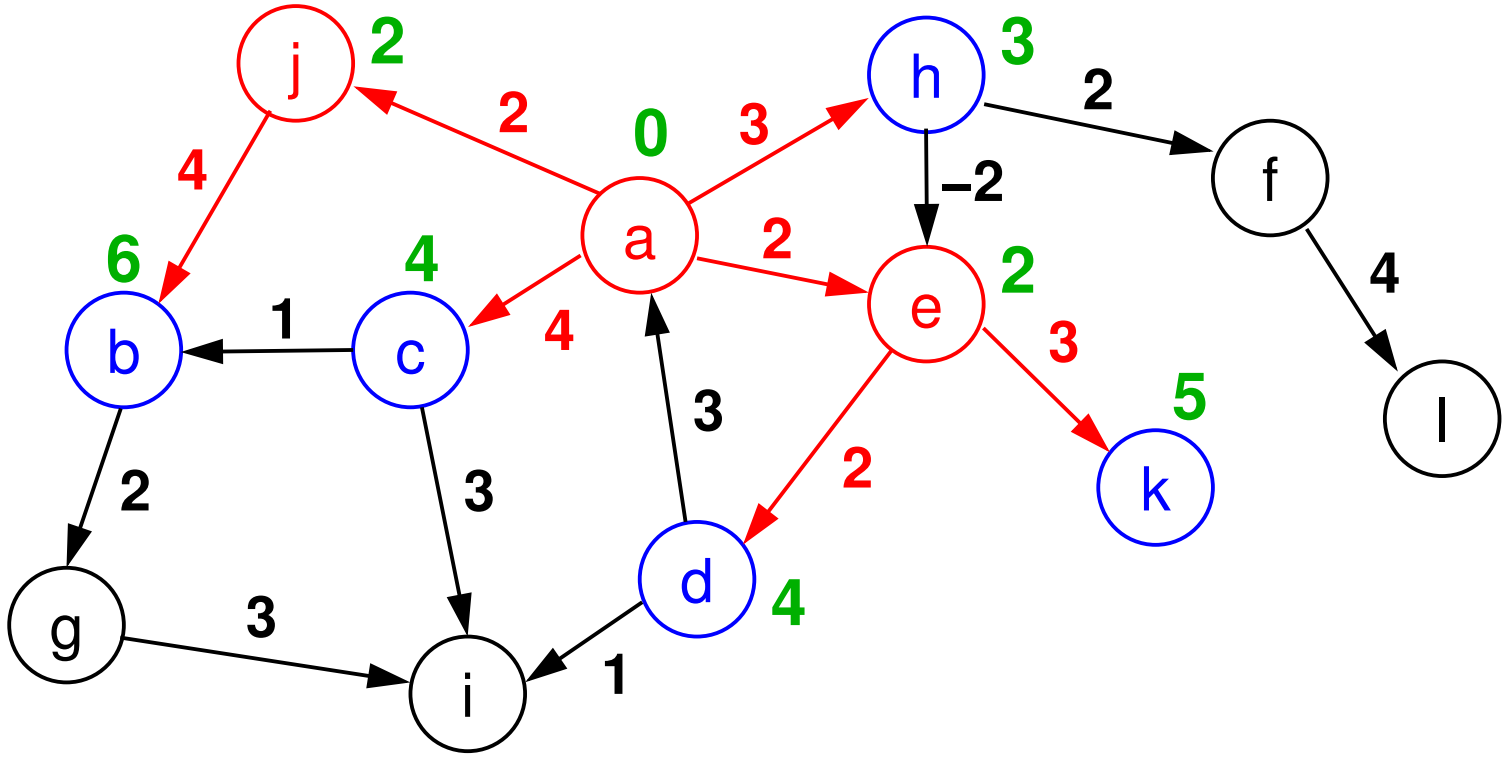
> Suppose, by contradiction, that we visit a node $z$ (currently annotated with distance $A$) such that $A$ is not the shortest distance from $u$ to $z$.
>
> Then, there must be some node $w$ that is visited later such that the shortest distance from $u$ to $w$ is $B$, there is an edge with weight $C$ from $w$ to $z$, and $B + C < A$, so that $z$ requires updating.
>
> Since $z$ was visited before $w$, we must have $A + h(z) \leq B + h(w)$.
>
> However, this together implies $B + C + h(z) < B + h(w)$, and therefore $h(w) > C + h(z)$, which contradicts the assumption that $h$ is monotone.

# Dijkstra's algorithm and negative weights

As mentioned, Dijkstra's algorithm assumes that the weights of the edges are non-negative.

If there are non-negative weights, the algorithm can fail.

# Negative weights: Example




Here, the edge from *h* to *e* has a negative weight. Starting from *a*, we first visit *j* and *e*. Only when *h* is visited next, we would update the distance of *e* to `1`, but the distance of *k* and *d* would remain incorrect.

# Dijkstra's algorithm and negative weights

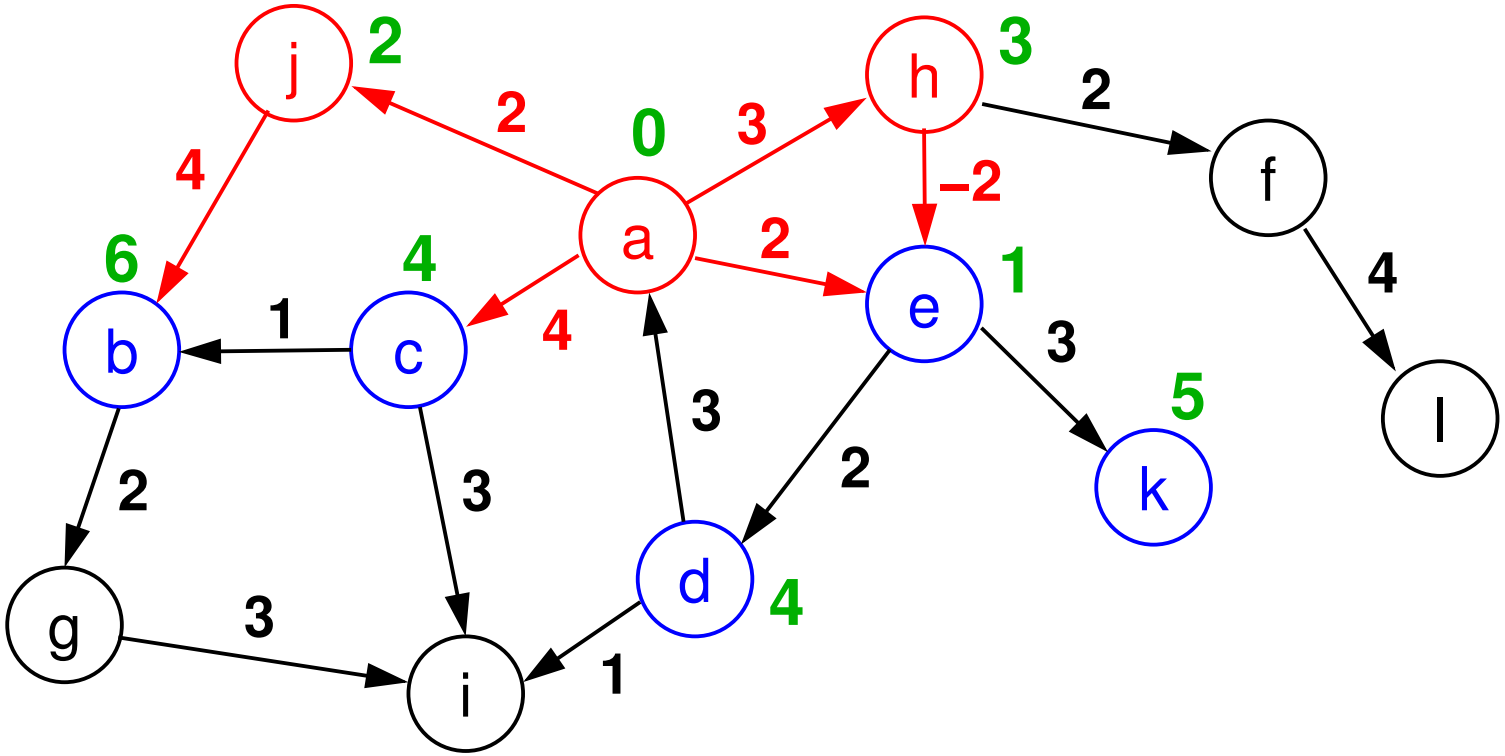
As mentioned, Dijkstra's algorithm assumes that the weights of the edges are non-negative.

If there are non-negative weights, the algorithm can fail.

However, there is a way to adapt the algorithm in case there are non-negative edges but no negative cycles (i.e., cycles in which the sum of the weights is negative).

> Whenever the distance visited (red) node is updated, make this node a front (blue) node again.

# Negative weights: Example




With the aforementioned fix, *e* becomes a front node once more when *h* is visited. This will cause the distance of *k* and *d* to be corrected eventually.

# Dijkstra's algorithm and negative weights

As mentioned, Dijkstra's algorithm assumes that the weights of the edges are non-negative.

If there are non-negative weights, the algorithm can fail.

However, there is a way to adapt the algorithm in case there are non-negative edges but no negative cycles (i.e., cycles in which the sum of the weights is negative).

> Whenever the distance visited (red) node is updated, make this node a front (blue) node again.

This will mean that the algorithm may visit notes more than once, so the stated complexity will no longer be true.

Beware: With this change, the algorithm may never terminate in the presence of negative cycles!

# Shortest paths in the presence of negative cycles

If a negative cycle exists in the graph, then there is no meaningful “shortest path” towards to nodes in the graph.

The Bellman-Ford algorithm (discussed on the next slide) can be used in this case instead. It has the following properties:

The Bellman-Ford algorithm *always* terminates.

If there is a negative cycle, the algorithm detects it (and aborts).

Otherwise, it will find the shortest paths, like Dijkstra’s algorithm.

However, the price to pay is a longer running time.

Therefore, using the Bellman-Ford algorithm makes sense only if the graph under consideration has negative weights but one is uncertain whether they form a negative cycle.

# Bellman-Ford algorithm

The Bellman-Ford algorithm computes a shortest distance $d[v]$ for each node $v$ from some starting node $u$. It works as follows (sketch):

In the following, a "round" means to take each edge $(v, w)$ with weight $x$ and set $d[w] := \min\{d[w], d[v] + x\}$.

1. Set $d[v] := \infty$ for all nodes $v \neq u$, and $d[u] := 0$.

2. Perform $n - 1$ rounds, where $n$ is the number of nodes.

3. Perform one more round. If any distance value changes in this last round, then abort, saying that a negative cycle has been detected. Otherwise, $d[v]$ provides the shortest distance for all nodes $v$.

The correctness of the algorithm uses on the fact that, in the absence of negative cycles, a shortest path will contain at most $n - 1$ edges. On the other hand, every cycle can be completed with at most $n$ edges.

# All-pairs shortest distances

Suppose that we have a graph $G$ (possibly with negative edge weights or even negative cycles) with $n$ nodes and $m$ edges.

Let $d(u, v)$ denote the shortest distance between nodes $u$ and $v$ in a graph $G$. Computing $d(u, v)$ for all pairs $u, v$ is called the all-pairs shortest-path problem.

Reminder: The Bellman-Ford algorithm computes $d(u, v)$ for a fixed node $u$ and all nodes $v$.

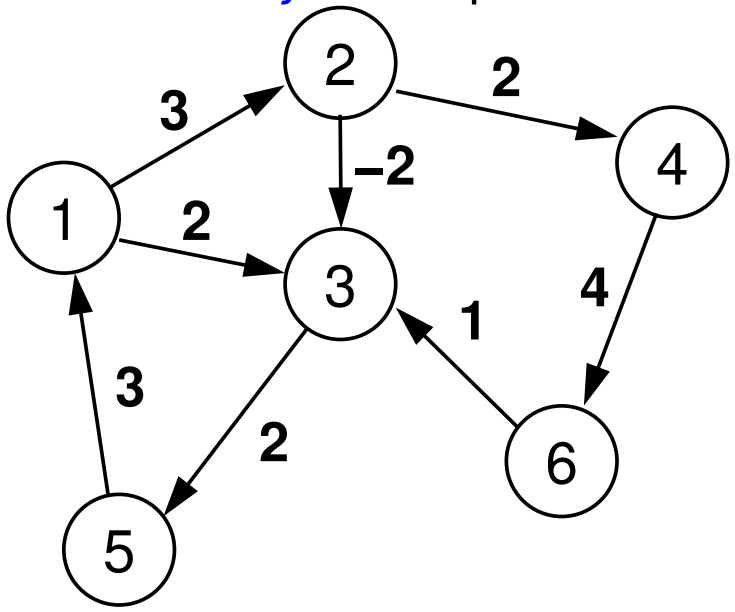
One could obtain the solution of the all-pairs shortest path problem by using Bellman-Ford once for each node $u$. The running time for this is $\mathcal{O}(n^2 \cdot m)$ (even $\Theta(n^2 \cdot m)$).

Note that (for connected graphs), $m$ is usually bigger than $n$. There is a better solution for the all-pairs problem that takes only $\mathcal{O}(n^3)$ time.

# Distance matrix

In the following, we again assume that the nodes are numbered $1..n$.

Now, we can arrange the edge weights in an $n \times n$ matrix. The entry at position $(i, j)$, denoted $M(i, j)$ is the weight of the edge from $i$ to $j$, $\infty$ if no such edge exists, and $0$ if $i = j$. Example:




$$\begin{pmatrix} 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & 0 & -2 & 2 & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & 4 \\ 3 & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix}$$

# Constrained path

Let $i, j$ be nodes and $0 \leq k \leq n$. A $k$-constrained path from $i$ to $j$ is a path that visits only nodes whose numbers are at most $k$ (apart from $i$ and $j$).

Let $c(i, j, k)$ denote the length of the shortest $k$-constrained path from $i$ to $j$. Obviously, $c(i, j, n) = d(i, j)$ and $c(i, j, 0) = M(i, j)$.

The algorithm that we shall discuss (called the Floyd-Warshall algorithm) makes use of these two facts.

> It starts with the values $c(i, j, 0)$, given by the distance matrix. Then it makes $n$ iterations. In the first iteration, the matrix will be replaced by one that gives the values $c(i, j, 1)$, then in the next iteration $c(i, j, 2)$, and eventually $c(i, j, n)$, giving the solution.

For computing shortest $k+1$-constrained paths from shortest $k$-constrained paths, the following property helps:

$$c(i,j,k+1) := \min\{c(i,j,k),\ c(i,k+1,k) + c(k+1,j,k)\}$$

Explanation: If we already know the shortest $k$-constrained path, then the only additional paths that we may take are via the node $k+1$.

Below, the values for $c(i,j,0)$ (left) and $c(i,j,1)$ (right) are shown.

$$\begin{pmatrix} 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & 0 & -2 & 2 & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & 4 \\ 3 & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & 0 & -2 & 2 & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & 4 \\ 3 & 6 & 5 & \infty & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix}$$

The second iteration, values for $c(i,j,1)$ (left) and $c(i,j,2)$ (right). On the left, the values for going through node $2$ are indicated in magenta, on the right, the changed values in red.

$$\begin{pmatrix} 0 & 3 & 2 & \infty & \infty & \infty \\ \infty & 0 & -2 & 2 & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & 4 \\ 3 & 6 & 5 & \infty & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 3 & 1 & 5 & \infty & \infty \\ \infty & 0 & -2 & 2 & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & 4 \\ 3 & 6 & 4 & 8 & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix}$$

One more iteration:

$$\begin{pmatrix} 0 & 3 & 1 & 5 & \infty & \infty \\ \infty & 0 & -2 & 2 & \infty & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & 4 \\ 3 & 6 & 4 & 8 & 0 & \infty \\ \infty & \infty & 1 & \infty & \infty & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 3 & 1 & 5 & 3 & \infty \\ \infty & 0 & -2 & 2 & 0 & \infty \\ \infty & \infty & 0 & \infty & 2 & \infty \\ \infty & \infty & \infty & 0 & \infty & 4 \\ 3 & 6 & 4 & 8 & 0 & \infty \\ \infty & \infty & 1 & \infty & 3 & 0 \end{pmatrix}$$

Final result:

$$\begin{pmatrix} 0 & 3 & 1 & 5 & 3 & 9 \\ 3 & 0 & -2 & 2 & 0 & 6 \\ 5 & 8 & 0 & 10 & 2 & 14 \\ 10 & 13 & 5 & 0 & 7 & 4 \\ 3 & 6 & 4 & 8 & 0 & 12 \\ 6 & 9 & 1 & 11 & 3 & 0 \end{pmatrix}$$

# Notes on the Floyd-Warshall algorithm

The algorithm makes $n$ iterations, in each iteration each pair of nodes will be considered. Therefore, the complexity is $\Theta(n^3)$, independently of the number of edges.

The final matrix gives the correct values for $d(i,j)$ unless there are negative cycles in the graph (in which case no "shortest" paths exist for some nodes).

However, if a node $i$ is located on a negative cycle, then the entry $(i,i)$ in the final matrix will be negative. If such a node exists, then the final values should be discarded.

# Minimum spanning tree

We consider one more problem with graphs, this time on *undirected* graphs.

Let $G = (V, E)$ be a *connected* undirected graph with weights on the edges. Let $w(e)$ denote the weight of edge $e$. (Negative weights are allowed.)

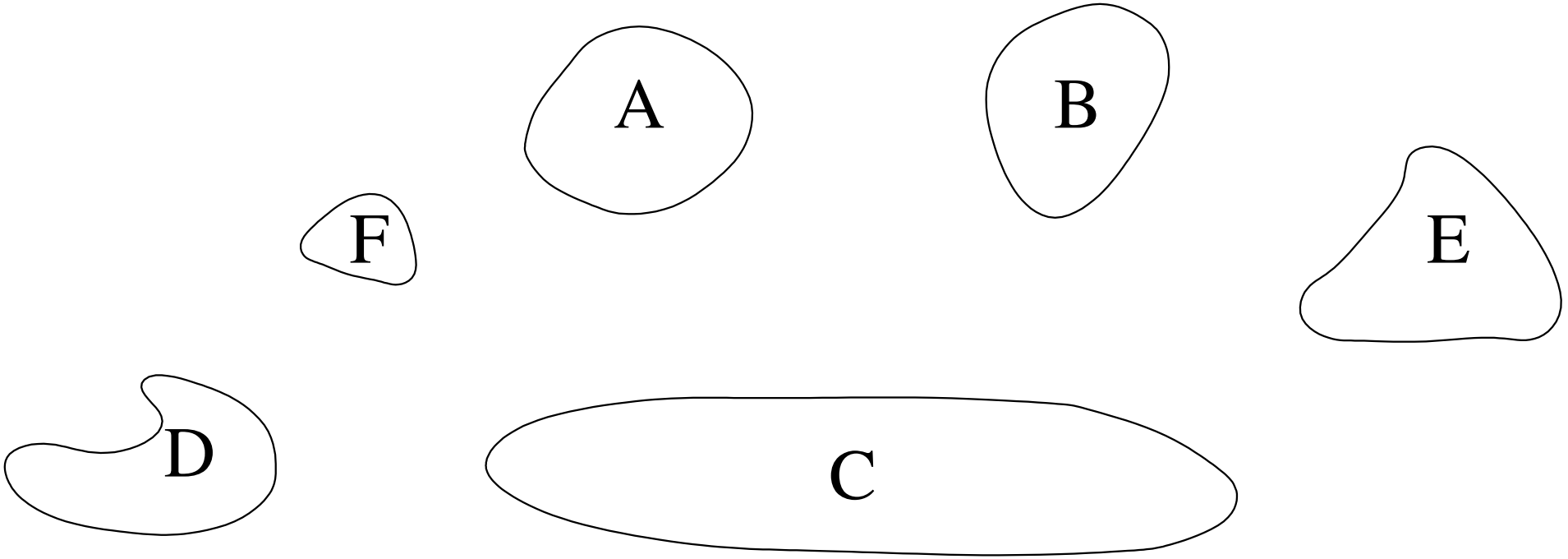
A spanning tree of $G$ is a graph $G' = (V, E')$, where $E' \subseteq E$, such that $G'$ is connected and a tree, i.e. a graph without cycles.

The minimum spanning tree of $G$ is a spanning tree $G' = (V, E')$ such that $w(G') := \sum_{e \in E'} w(e)$ is minimal among all spanning trees.

Note: Any graph $G'' = (V, E'')$ is a spanning tree if and only if it is connected and has $|V| - 1$ edges.

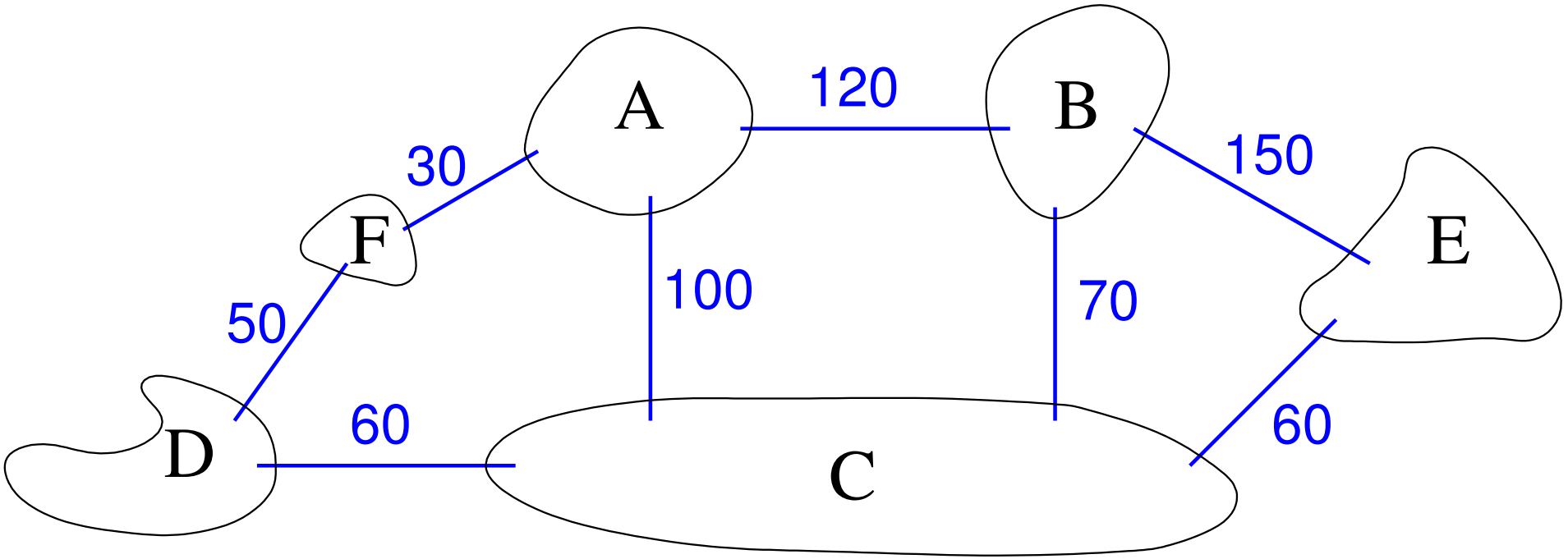
Application: Connecting a set of objects with least cost (e.g., bridges, cabling).

# Example



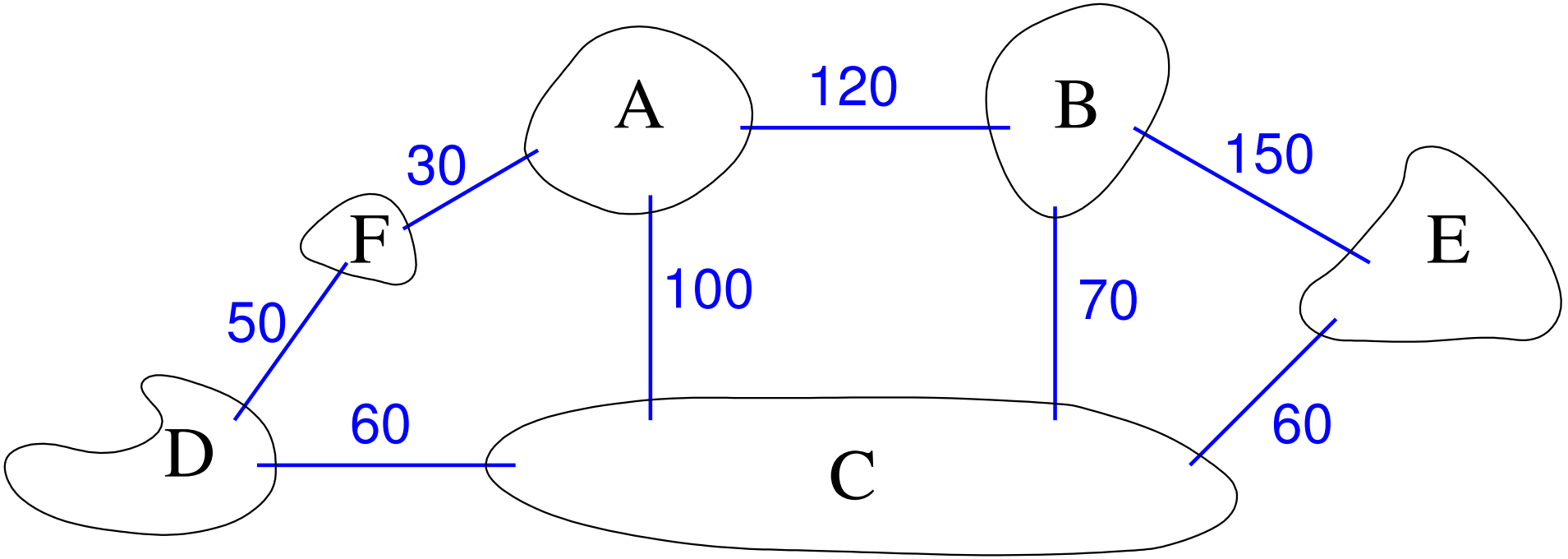

Here's an archipelago consisting of six islands. Until recently, the islanders were content rowing from one island to another when necessary.

# Example



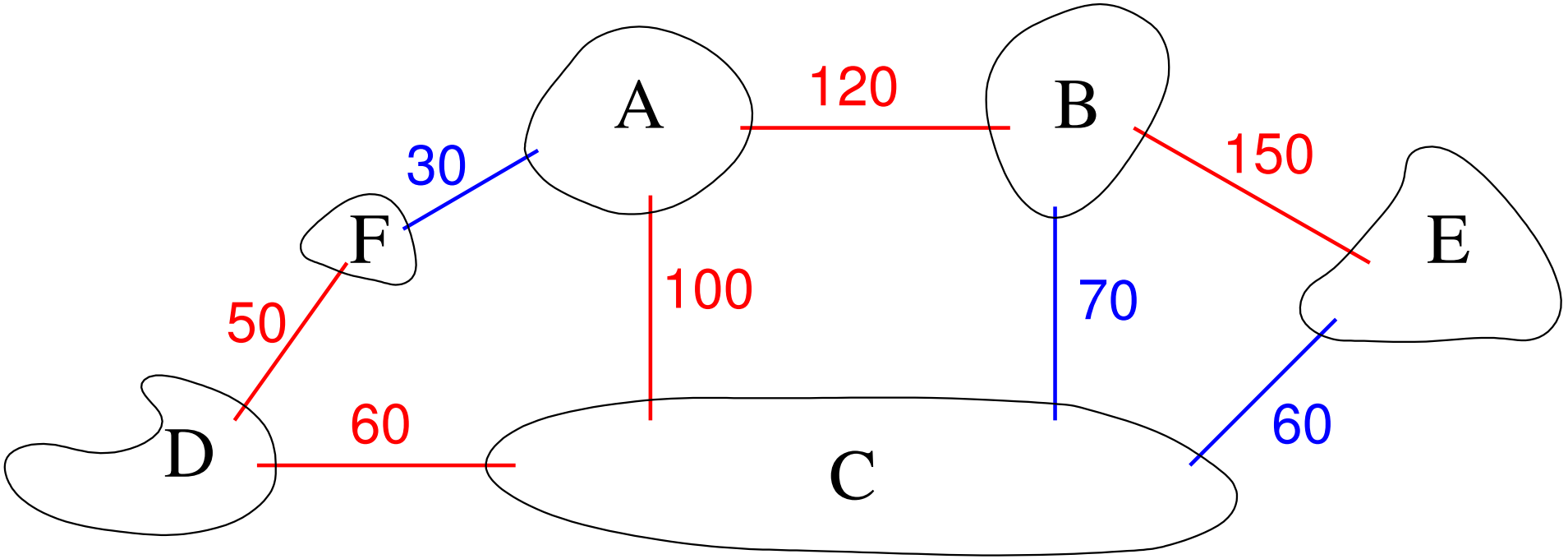

But now, the islanders want to connect their islands with bridges. They have figured out possible location of bridges and the costs of building them, as indicated.

# Example




Owing to global recession, the islanders cannot afford to build all bridges. They just want to make sure that all islands are connected somehow.

# Example




In short, they want to select a subset of edges forming a tree. The bridges indicated in red indicate a possible solution, but not one with minimal cost. How can they do better?

# Prim's algorithm

An algorithm due to Prim computes a minimal spanning tree. It works as follows:

1. Set $E' := \emptyset$ and $V' := \emptyset$.

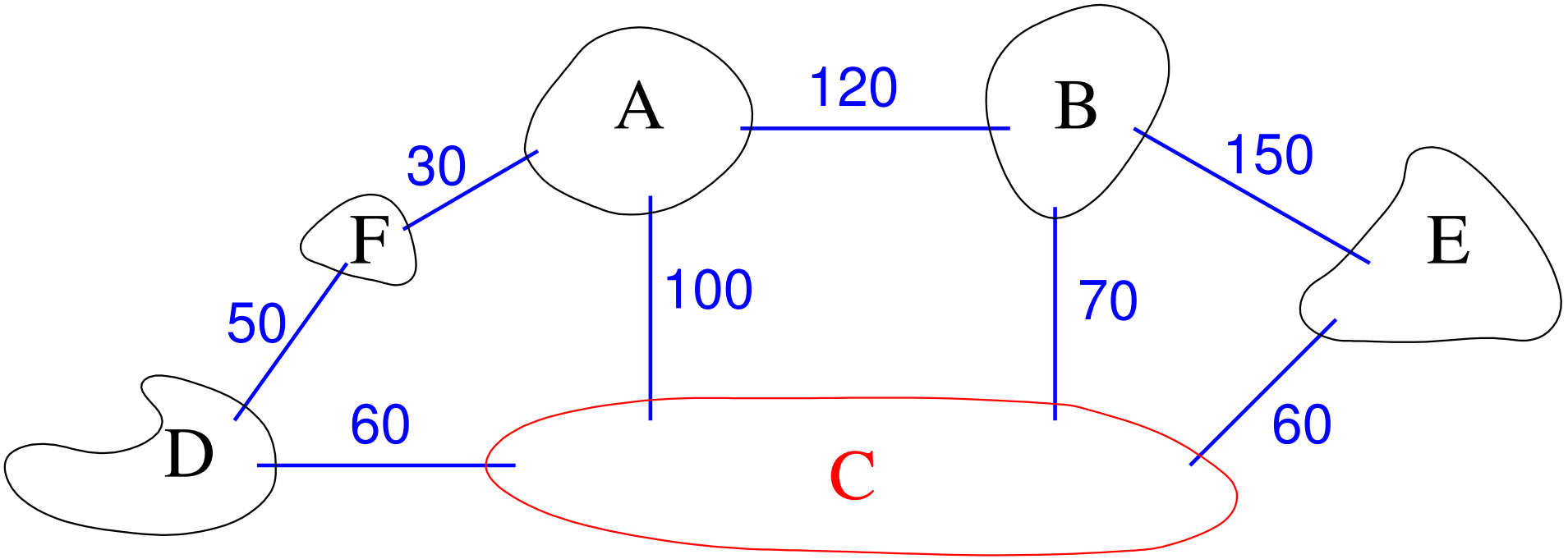
2. Select any node $v \in V$ and add it to $V'$.

3. While $V' \neq V$, repeat the following:

   Among all the edges $\{v, w\}$ such that $v \in V'$ and $w \notin V'$, let $e$ be one with minimal weight. Add $e$ to $E'$ and $w$ to $V'$.

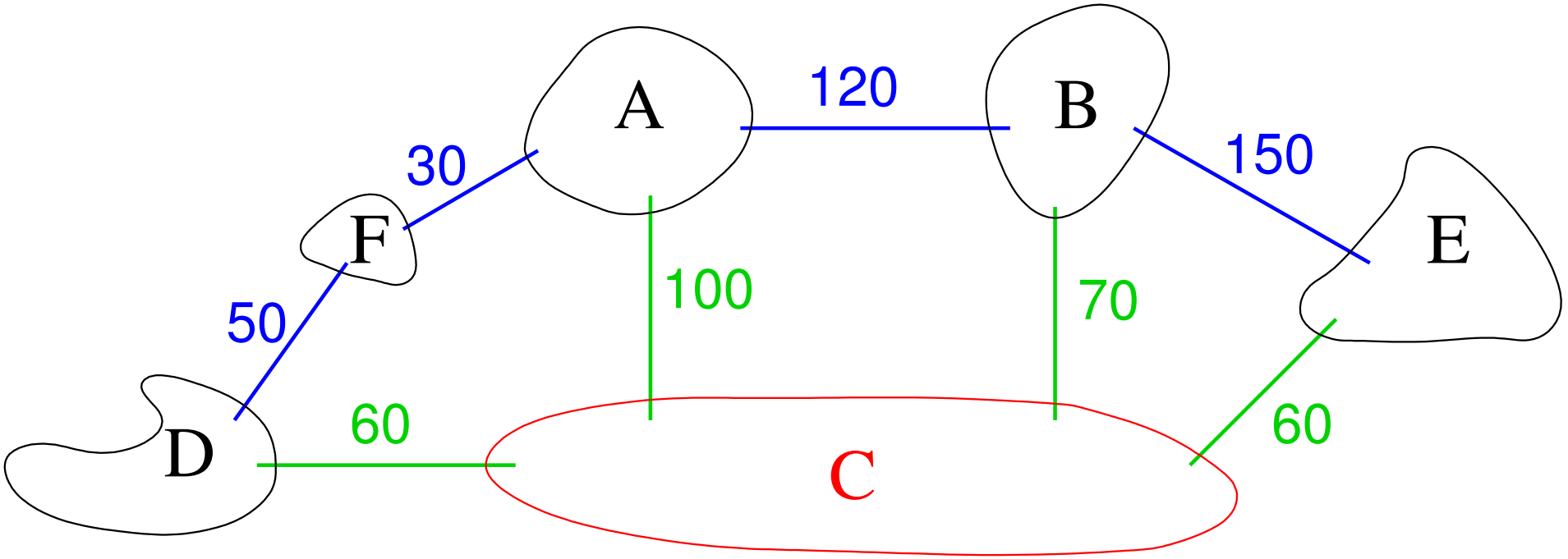
The result, $G' := (V, E')$, is a minimal spanning tree.

# Example



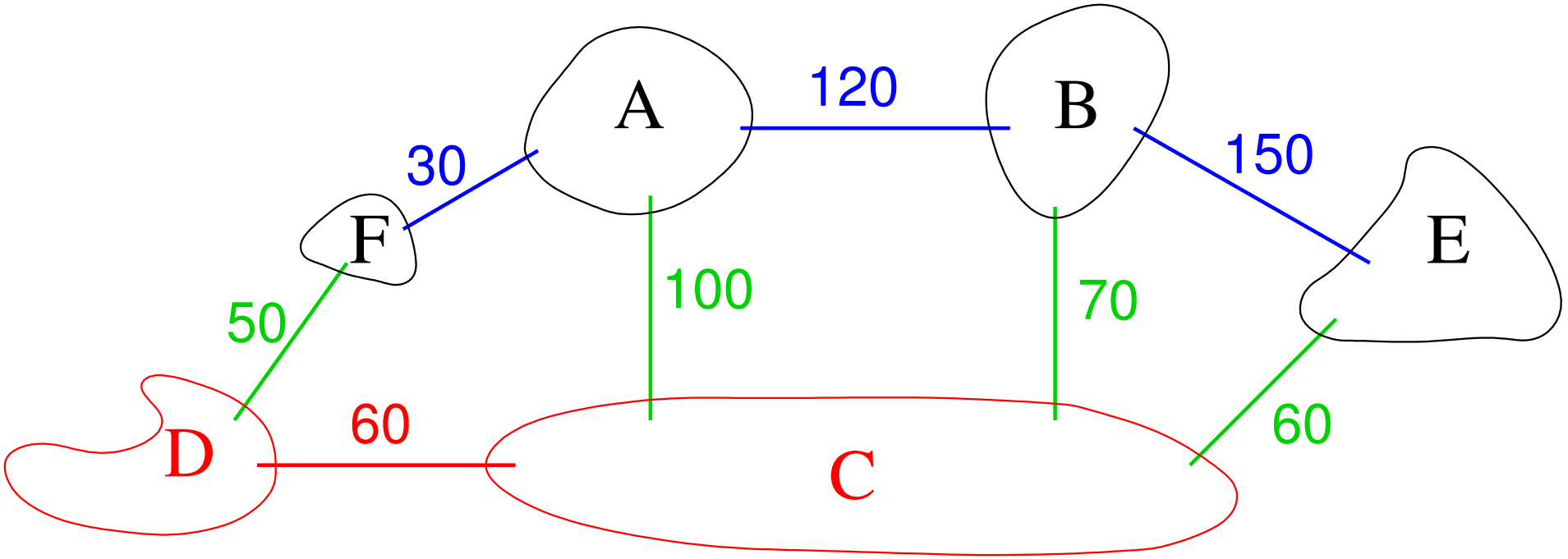

In the following, the contents of $V'$ and $E'$ are indicated in red. Suppose that we select $C$ in step 2.

# Example



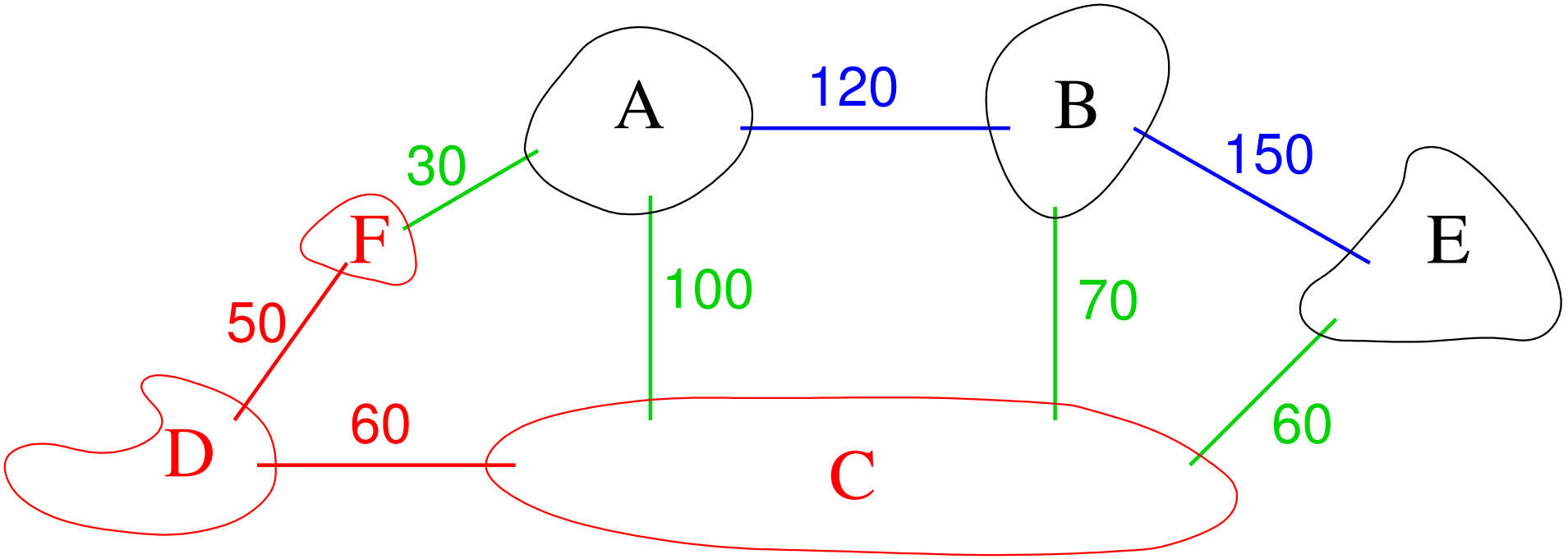

Now, the green edges are those connecting red nodes (in $V'$) to black nodes (not in $V'$). There are two edges with minimal weight, the ones to $D$ and $E$.

# Example



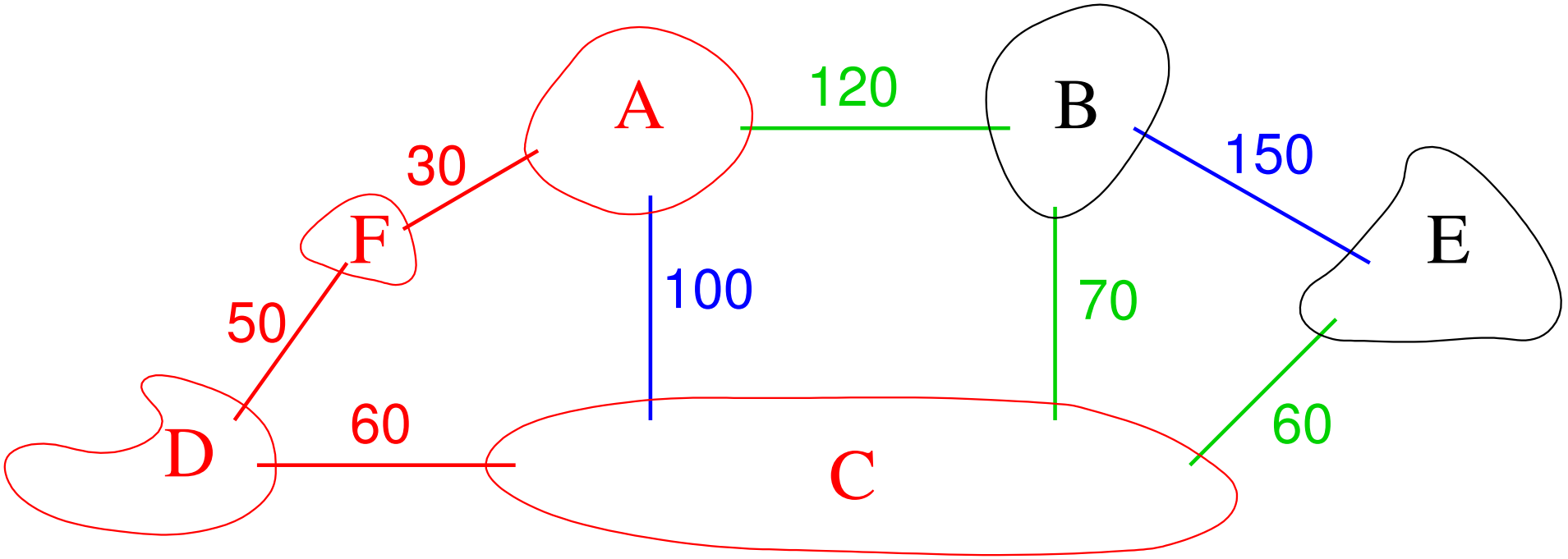

Let's say we pick the one to *D* and add it to *E′*. Here's the result. The next minimal green edge is the one from *D* to *F*.

# Example



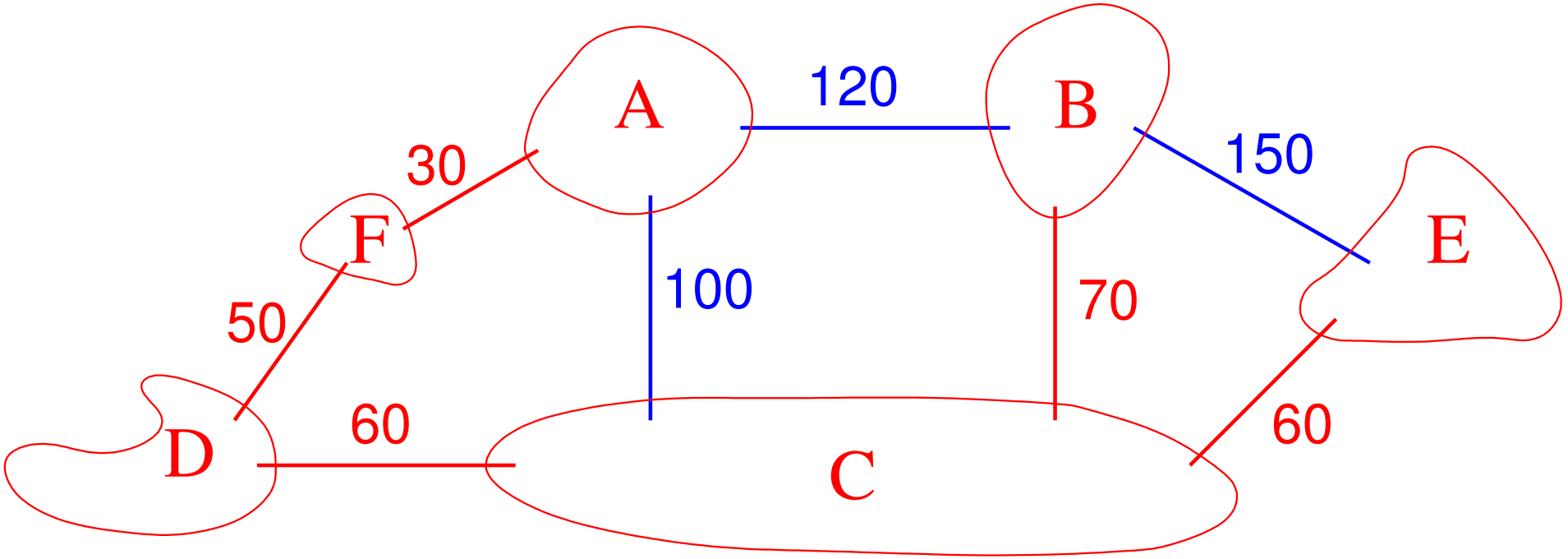

The situation after “building the bridge” from *D* to *F*. The next minimal green edge is the one from *F* to *A*.

# Example




Notice that the edge from *C* to *A* is removed from consideration as *A* is added.

# Example

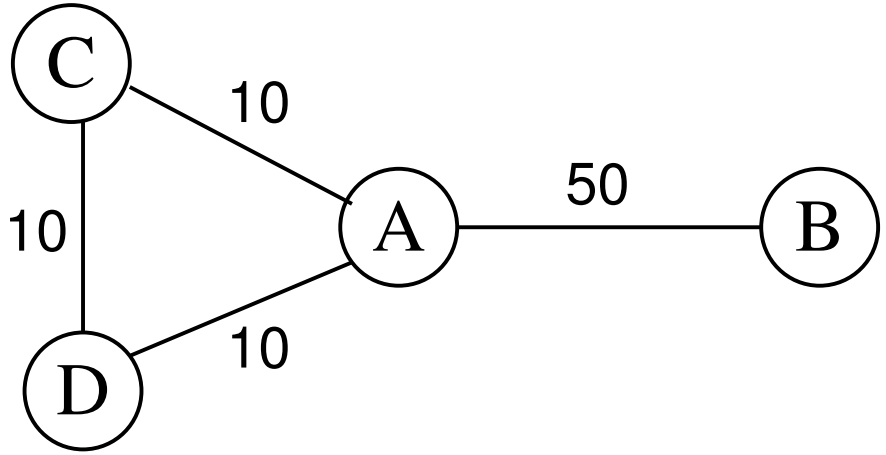



In the final two iterations, bridges to *E* and *B* are added (in that order). The final result is shown above.

# Caveat

In the previous example, the minimal spanning tree could also have been obtained by simply picking the $|V| - 1$ edges with minimal cost.

But doing so is not guaranteed to yield the correct result in general, as the example below shows:



# Correctness of Prim's algorithm

It is easy to see that the algorithm indeed computes a spanning tree.
In each iteration of step 3, another node is connected to $(V', E')$ until all are connected, and the choice of $e$ ensures that no cycles are created.

As for minimality, let $G'$ be the tree obtained by the algorithm, and let $H$ be some minimal spanning tree (there might be several). We shall show that $w(G') = w(H)$.

> Either $G' = H$, then we are done. Otherwise, let $e_1, \ldots, e_k$ be the edges added to $G'$ by the algorithm, in that order, and let $e_m = \{v, w\}$ be the first edge not contained in $H$. Let $V_m$ be the set $V'$ at the time $e_m$ was added. Thus, $v \in V_m$ but $w \notin V_m$.
>
> Now, since $H$ is a spanning tree, it must contain some path from $v$ to $w$, and let $f = \{v', w'\}$ be the first edge along the path such that $v' \in V_m$ and $w' \notin V_m$.

Thus, since $f$ was not chosen as the $m$-th edge in $G'$, we have $w(f) \geq w(e_m)$. We obtain a new graph $H'$ from $H$ by removing $f$ and adding $e_m$. Obviously, $w(H') \leq w(H)$.

$H'$ is still connected: for every two nodes connected via $f$ in $H$, there is now a new connection in $H'$ via $e_m$. Moreover, $H'$ is also a tree since it is connected and has $|V| - 1$ edges, so $H'$ is another minimal spanning tree that contains all of $e_1, \ldots, e_m$.

(Incidentally, since $H$ was already a minimal spanning tree, we can now conclude that $w(f) = w(e_m)$.)

Now, either $G' = H'$ and we're done, or there is some edge $e_{m'}$ contained in $G'$ but not in $H'$, with $m' > m$. We can now repeat the same argument as before and obtain a new minimal spanning tree that agrees with $G'$ on the first $m'$ edges etc until we get one that is identical to $G'$.

# Additional notes on Prim's algorithm

The algorithm does $|V| - 1$ iterations of step 3. In each iteration, the only non-trivial operation is to pick the minimal-weight "green" edge.

In order to do that efficiently, we can organize the green edges in a heap, in which case picking the minimal element is easy.

But then, we need to add edges to the heap whenever one of their nodes is added to $V'$. Each edge needs to be added at most once, so at most $|E|$ edges can be in the heap, and each addition takes $\mathcal{O}(\log |E|)$ time.

Thus, the total time of the algorithm is $\mathcal{O}(|V| + |E| \log |E|)$.

Another algorithm for computing minimal spanning trees (with different, usually worse, complexity) is Kruskal's algorithm (not discussed here).

# Addendum: Finding important nodes in a graph

# Example: The Munich subway

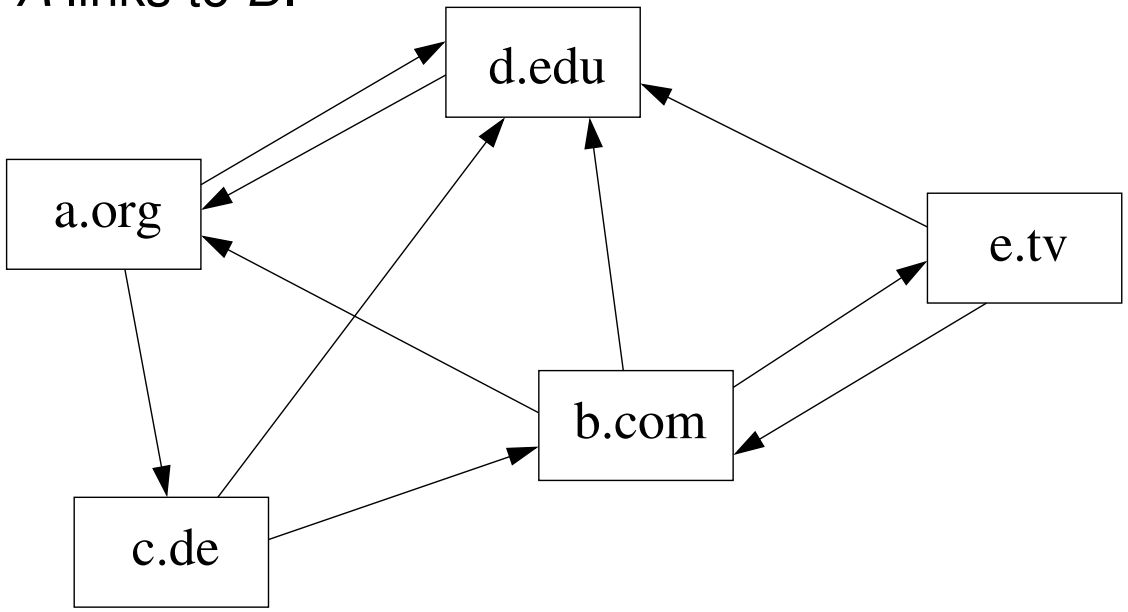
Question: Which are the most important stations in Munich's subway network? (assuming they're not on strike. . . )

> Intuitively, the ones in the centre seem to be important, i.e. those where several lines intersect (Hauptbahnhof, Sendlinger Tor, Odeonsplatz, possibly Marienplatz and Karlsplatz if the S-Bahn is also taken into account).

We shall find a mathematical concept that describes "importance" and matches this intuition.

# Example 2: The Internet

Consider the following (fictional) websites on the Internet. We draw an edge from site *A* to site *B* if *A* links to *B*.




Question: Which is the most important website?

Or: What is the most important website containing a given keyword?
(This is the question one asks when searching the Internet using, e.g., Google.)

# Ideas

In the following, we regard directed graphs (representing, e.g., the subway network, the Internet, . . . )

We shall distill a notion of importance from the structure of the graph.

In the Internet example, a site could be considered important if many other site link to it (i.e., a link can be considered a “recommendation”).

> However, not all links might have the same quality.
>
> In particular, links from sites that are already important are more relevant than links from less important sites.
>
> Note: The previous statement seems to imply some intractable recursive definition. However, we shall overcome this problem.

# Probabilities as weights

Moreover, not all links from the same site may have the same importance. (e.g., depending on their position on the site and other factors)

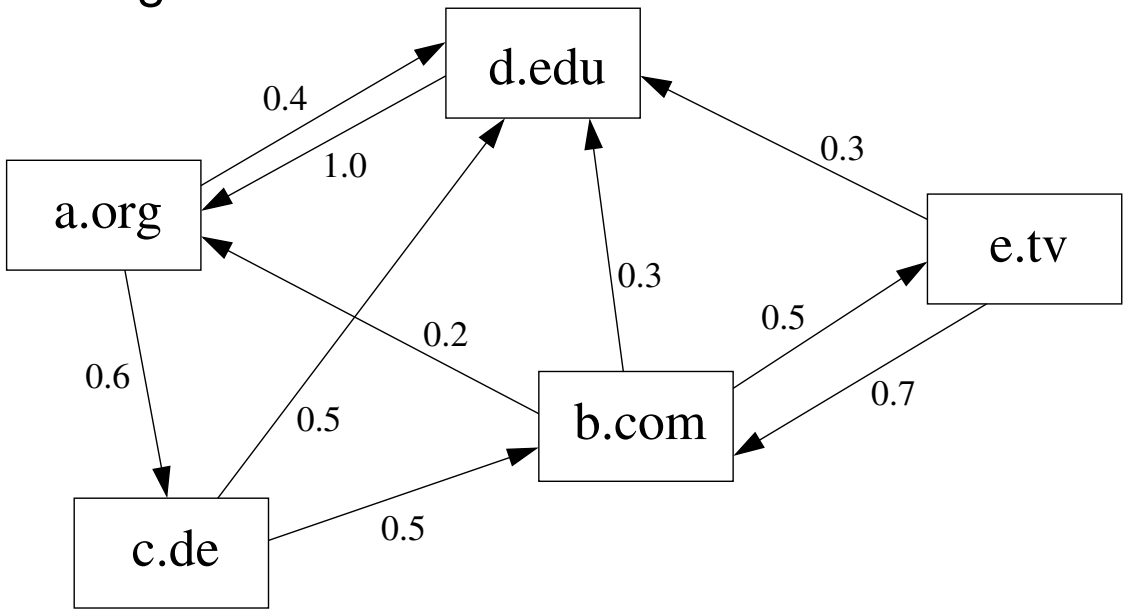
In the following, let us express the relative importance of links from the same page by weights.

More to the point, our weights shall be probabilities:

> The weight on the edge from $A$ to $B$ represents the probability of clicking on the link to $B$ when visiting site $A$.
>
> E.g., if all links on $A$ are equally important, then all outgoing edges from $A$ will have the same weights.
>
> The sum of the weights on all outgoing edges from $A$ needs to be $1$.

# Markov chains

As an example, consider the websites from Example 2, with additional probabilities on the edges.




Notice that the sum of the weights on the outgoing edges for each node is indeed 1.

Such a directed graph with probabilities is also called a Markov chain.

# Random walk

Let us imagine a random walk on a Markov chain:

A random walk starts at some node, say $u$.

In every step, the next node is chosen according to the probabilities.

I.e., if we are at node $u$ in step $n$ and the probability on the edge from $u$ to $v$ is $p$, then in step $n+1$ we are at $v$ with probability $p$.

Let $p^n_{u,v}$ denote the probability of, when starting in $u$, ending up in $v$ after $n$ steps.

To compute these values for a fixed $n$, we simply list all the paths of length $n$ leading from $u$ to $v$ and add up their probabilities.

E.g., to go from `b.org` to `d.edu` in exactly two steps, we can go via *a.org* or *e.tv*, so

$$p^2_{b,d} = 0.2 \cdot 0.4 + 0.5 \cdot 0.3 = 0.23$$

The values for $p^1_{u,v}$ are directly given by the edge probabilities.

For $n \geq 2$, we can compute the values $p^{n+1}_{u,v}$ from the respective values for $p^n_{u,v}$, as follows, where $V$ is the set of all nodes:

$$p^{n+1}_{u,v} = \sum_{w \in V} p^n_{u,w} \cdot p^1_{w,v}$$

Note: Google uses a model like this to determine the importance of websites. However, they use a random-walk model with a "reset" probability: In each step, one can go to *any* website with small probability. This reflects the possibility that a web user gets bored with the present website and types in some new URL. The Markov chain can be adapted accordingly.

# Probabilities and “importance”

Intuitively, a random walk likely to visit an “important” website more often than a less important one.

Therefore, we are interested in the frequency with which sites are visited on average in random walks. This corresponds to the probabilities “in the long run”.

In the following, we shall use some facts from probability theory (without proof). It should be noted that these hold under certain benign conditions:

irreducibility, i.e. the nodes of the graph form one big SCC;

aperiodicity, i.e. it must not be possible to always walk in cycles of a period larger than 1.

The Markov chains we consider (and the one considered by Google) easily fulfil these conditions.

# Facts from probability theory (without proof)

Let $u, v, w$ be nodes of a Markov chain. Then
$\lim_{n\to\infty} p^n_{u,w} = \lim_{n\to\infty} p^n_{v,w} =: p_w$.

In other words, in the long run, it does not matter where we start a random walk, the probabilities for going to some node $w$ are going to converge to a value we shall call $p_w$. Indeed, it turns out that the values of $p_w$ are unique.

The value of $p_w$ is proportional to the relative frequency with which $w$ is visited by random walks "in the long run". E.g., if $p_u$ is twice as high as $p_v$, then $u$ will be visited twice as often as $v$, on average.

Thus, the "importance" of nodes is given by the values $p_w$.
(Importance values are *relative*.)

# Stationary probabilities

The values $p_w$ are called stationary probabilities.

This is because, if the values are converging, then the following must hold:

$$p_w = \sum_{v \in V} p_v \cdot p^1_{v,w}$$

Notice that the latter gives a system of linear equations, which can be solved by standard methods (Gauß-Seidel, Newton, ...).

To obtain a unique solution, one must make use of the additional property $\sum_{w \in V} p_w = 1$.